TECH TODAY

Реверс-инжиниринг программного обеспечения x86, кракинг и контрмеры

STEPHANIE DOMAS
CHRISTOPHER DOMAS
@voidptr0xff @reverse_domain

WILEY

Вступление

Реверс-инжиниринг и взлом программного обеспечения - дисциплины с долгой и богатой историей. На протяжении десятилетий разработчики программного обеспечения пытались встроить средства защиты в свои приложения для защиты интеллектуальной собственности или предотвращения внесения изменений в программный код. Искусство взлома существует почти с тех пор, как реверс-инженеры изучали и модифицировали код для развлечения или получения прибыли.

Прежде чем углубляться в детали того, как работает реверс-инжиниринг, полезно понять контекст, в котором находятся эти дисциплины. В этой главе описывается, чего ожидать от этой книги, и рассматриваются история и юридические аспекты реверс-инжиниринга программного обеспечения и взлома.

Кому следует прочитать эту книгу

От профессионалов в области безопасности до любителей - эта книга предназначена для всех, кто хочет научиться разбирать, понимать и модифицировать программное обеспечение "черного ящика". Эта книга знакомит любознательного человека, склонного к безопасности, с тем, как происходит взлом программного обеспечения и как работают компьютеры. Изучение того, как работает компьютер x86, не только полезно с точки зрения реверс-инжиниринга и взлома, но и сделает каждого читателя более сильным разработчиком, обладающим передовыми знаниями, которые он сможет применить для оптимизации кода, повышения эффективности, отладки, настройки компилятора и выбора чипа. Затем занавес продолжает опускаться, поскольку читатели узнают, как происходит взлом программного обеспечения. Читатели узнают об инструментах и методах, которые используют взломщики программного обеспечения в реальном мире, и они проверят свои новообретенные знания, самостоятельно взламывая реальные приложения в многочисленных практических лабораториях. Затем мы возвращаемся к пониманию защитных приемов борьбы со взломом программного обеспечения. Изучив как наступательные, так и оборонительные приемы, читатели станут опытными взломщиками программного обеспечения или защитниками программного обеспечения.

Чего ожидать от этой книги

Эта книга основана на трех основных принципах реверс-инжиниринга:

- Не существует такого понятия, как программное обеспечение, которое невозможно взломать.
- Цель в нападении - попытаться действовать быстрее.
- Цель в защите - попытаться замедлиться.

Основываясь на этой философии, любое программное обеспечение может быть подвергнуто обратной разработке, в результате чего его секреты могут быть украдены, а средства защиты обойдены. Это всего лишь вопрос времени.

Как и в других областях кибербезопасности, как атакующие, так и оборонительные реверс-инженеры выигрывают от наличия схожего набора навыков. Эта книга предназначена для ознакомления с этими тремя взаимосвязанными наборами навыков:

- **Реверс-инжиниринг**: Реверс-инжиниринг - это процесс разборки программного обеспечения на части и выяснения того, как оно работает.
- **Взлом**: Взлом основан на обратном проектировании путем манипулирования внутренностями программы, чтобы заставить ее делать то, для чего она не предназначалась.
- **Защита**: Хотя все программное обеспечение поддается взлому, защита может сделать взлом программы более сложным и отнимающим много времени.

Как атакующие, так и защитные реверс-инженеры извлекают выгоду из одного и того же набора навыков. Без понимания реверс-инжиниринга и взлома защищающийся не сможет создать эффективную защиту. С другой стороны, злоумышленник может более эффективно обходить и преодолевать эти средства защиты, если он может понимать, как работает программа, и манипулировать ею.

Структура книги

Эта книга организована на основе этих трех основных способностей и наборов навыков. Структура следующая:

Часть	Топик	Описание
Часть 1: Бэкграунд	Ускоренный курс по истории и правовым аспектам x86	Разберитесь в x86 и научитесь быстро работать.
Часть 2: Обратная разработка программного обеспечения	Разведка Средства проверки ключей Генераторы ключей Мониторинг процессов Манипулирование ресурсами Статический анализ Динамический анализ Написание ключевых слов Взлом программного обеспечения	Освоение инструментов, подходов и мышления, необходимых для разбора программного обеспечения на части и понимания его внутренней работы.
Часть 3: Взлом программного обеспечения	Ручное исправление Автоматические средства исправления Расширенный динамический анализ Отслеживание выполнения Расширенный статический анализ Пробные периоды Раздражающие экраны Больше ключевых генераций Больше взлома	Овладейте инструментами, подходами и мышлением, необходимыми для изоляции поведения и модификации программного обеспечения.
Часть 4: Средства защиты, контрмеры и расширенные темы	Запутывание/ деобфускация Анти-отладка/anti-debugging Упаковка/распаковка Шифровальщики/дешифраторы Архитектурная защита Законный Вневременная отладка	Осваивайте приемы защиты и контрзащиты. Оценивайте оборонительную позицию и компромиссы. Изучайте продвинутые темы. Применяйте инструменты обратного проектирования и взлома, техники и мышление.

	Бинарный инструментарий Промежуточные представления Декомпиляция Автоматическое восстановление структуры Визуализация Доказательства теорем Символический анализ Феерия взлома

Практический опыт и лабораторные работы

Лучший способ научиться обратному проектированию и взлому программного обеспечения - это делать это на практике. По этой причине в эту книгу будет включено несколько практических лабораторных работ, демонстрирующих концепции, описанные в тексте.

Цель этой книги не в том, чтобы обучить определенному набору инструментов и техник. Хотя основное внимание уделяется программному обеспечению x86, работающему в Windows, многие подходы и техники будут перенесены на другие платформы. В этой книге будет предпринята попытка продемонстрировать широкий спектр инструментов, включая решения с открытым исходным кодом, бесплатные, условно-бесплатные программы и коммерческие решения. Понимая, какие инструменты доступны, а также их относительные сильные и слабые стороны, вы сможете более эффективно выбирать подходящий инструмент для работы.

Практические лабораторные работы и упражнения также будут посвящены обратному проектированию и взлому множества различных целей, включая следующее:

- Реальное программное обеспечение: В некоторых упражнениях будет использоваться реальное программное обеспечение, тщательно отобранное во избежание нарушений авторских прав.
- Готовые примеры: Программное обеспечение, написанное специально для этой книги, чтобы проиллюстрировать концепции, которые практически невозможно продемонстрировать на примерах из реального мира.
- Crackmes: Готовое программное обеспечение, разработанное crackers для иллюстрации концепции или вызова другим.

Сопутствующие файлы для скачивания

В книге упоминаются некоторые дополнительные файлы, такие как лабораторные работы или инструменты. Эти элементы доступны для скачивания с https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES.

История

Прежде чем углубляться в мельчайшие детали взлома и обратного инжиниринга, полезно ознакомиться с их историей. Средства защиты программного обеспечения, а также приемы и техники, используемые для их преодоления, развивались десятилетиями.

Первые средства защиты программного обеспечения

Первые средства защиты от копирования программного обеспечения появились в 1970-х годах. Некоторые из первых новаторов в этой области были следующими:

Apple II: В Apple II были встроены проприетарные драйверы дисков, которые позволяли записывать на дорожках, записывать дополнительные кольца, а также чередующиеся и перекрывающиеся сектора. Целью этого было сделать диски непригодными для использования машинами и программным обеспечением, не принадлежащими Apple, которые не знали бы, как читать и записывать с такими нечетными смещениями.
Atari 800: Системы Atari 800 намеренно включали поврежденные сектора в свои диски и пытались загрузить эти сектора. Если при загрузке не возвращалась ошибка “поврежденный сектор”, то программное обеспечение знало, что это недействительный диск, и останавливало выполнение.
Commodore 64: Законное программное обеспечение Commodore 64 распространялось только на дисках, доступных только для чтения. Программное обеспечение пыталось перезаписать диск, и, если это удавалось, оно знало, что диск поддельный.
Все эти средства защиты зависели от необычного поведения программного обеспечения, такого как использование недопустимой памяти или попытка перезаписать собственный код программы. Устранение этих средств защиты требовало понимания того, как работает программное обеспечение.

Рост числа взломов и обратного инжиниринга
Расцвет крекинга и обратного инжиниринга начался в 1980-х годах. Однако первые взломщики занимались этим не ради денег. Крекинг был соревнованием, целью которого было определить, кто быстрее сможет вычислить и обойти защиту программного обеспечения.

В течение следующих нескольких десятилетий сцена реверс-инжиниринга и взлома развивалась. Это некоторые из ключевых дат в истории реверс-инжиниринга:

1987: Организация Fairlight в 1987 году компанией Bacchus определяет одну из первых оперативных групп. Позже Fairlight получит известность в ходе репрессий ФБР в начале 2000-х. Для получения более исторических подробностей посетите www.fairlight.to и csdb.dk.
1990: Эллиот Дж. Чикофски и Джеймс Х. Кросс II определили реверс-инжиниринг как “процесс анализа предметной системы с целью идентификации компонентов системы и их взаимосвязей и создания представлений системы в другой форме или на более высоком уровне абстракции". (“Обратное проектирование и восстановление конструкции: таксономия”. Программное обеспечение IEEE, Том 7, выпуск 1, январь 1990).
1997: Old Red Cracker (handle +ORC) основывает основанный в Интернете Университет высокого уровня взлома (+HCU), чтобы каждый мог узнать о взломе. + ORC выпустил онлайн-уроки “как взломать” и стал автором научных статей. + У студентов HCU были ручки, которые начинались на +.
1997-2009: Появляется “варез-сцена”, где группы соревнуются за то, чтобы первыми выпустить материал, защищенный авторским правом. Инсайдеры (они же “поставщики”) предоставляли ранний доступ к своим группам, “взломщики” взламывали защиту, а “курьеры” распространяли взломанное программное обеспечение на FTP-сайтах. В период с 2003 по 2009 год на “сцене” было примерно 3164 активных группы, конкурирующие в первую очередь за гордость и право хвастаться, а не за деньги.
2004: ФБР и другие страны начинают рейды против “места преступления”. Операция Fastlink (2004) привела к осуждению 60 членов warez, а операция Site Down (2005) уничтожила 25 группировок warez.
Гонка вооружений между средствами защиты программного обеспечения и взломщиками продолжает бушевать, и обратное проектирование является бесценным набором навыков для обеих сторон. Взломщикам необходимо понимать, как работает программа, чтобы манипулировать ею и обходить защиту. С точки зрения защиты, важно понимать новейшие

методы взлома, чтобы разработать средства защиты интеллектуальной собственности и других конфиденциальных данных.

Законность

Лучший способ учиться - это делать. Вот почему в эту книгу включены лабораторные работы и упражнения с использованием реального программного обеспечения, а также готовые примеры и взломы. Мы не юристы, и тем, у кого есть сомнения, следует проконсультироваться с юристом. Мы рекомендуем Electronic Frontier Foundation (www.eff.org). Глава 15 посвящена юридическим темам, поскольку мы считаем, что для всех важно понимать законы США, которые влияют на эту область. Есть два основных закона, о которых следует знать: Закон об авторском праве и Закон об авторском праве в цифровую эпоху (DMCA).

Пункт о добросовестном использовании Закона об авторском праве (Copyright Act, 17 U.S.C. § 107) гласит, что обратное проектирование подпадает под “добросовестное использование”, когда выполняется для “...таких целей, как критика, комментарии, новостные репортажи, преподавание (включая многократные копии для использования в классе), стипендии или исследования....” Это исключение уравновешивается “влиянием использования на потенциальный рынок или стоимость защищенной авторским правом работы”. По сути, обратный инжиниринг, используемый в образовательных целях, является законным, если вы не распространяете и не продаете взломанное программное обеспечение.

В октябре 2016 года DMCA также добавила исключение для добросовестных исследований в области безопасности. В нем говорится: “доступ к компьютерной программе исключительно в целях добросовестного тестирования... когда такая деятельность осуществляется в контролируемой среде, предназначенной для предотвращения какого-либо вреда отдельным лицам или общественности... и не используется или не поддерживается способом, способствующим нарушению авторских прав”.

Программное обеспечение, рассмотренное в этой книге и используемое в упражнениях, было тщательно отобрано, чтобы подпадать под исключения добросовестного использования и DMCA. Если вы планируете осуществлять реинжиниринг и взламывать программное обеспечение для чего-либо, кроме самообразования, вам следует проконсультироваться с юристом. Юридические аспекты обратного инжиниринга также будут рассмотрены более подробно в одной из последующих глав.

Реверс-инжиниринг программного обеспечения и взлом программного обеспечения имеют богатую историю, и этот набор навыков может применяться как в наступательных, так и в оборонительных целях. Однако важно понимать законы, касающиеся этих дисциплин, и убедиться, что ваша деятельность подпадает под действие положений о добросовестном тестировании и добросовестном использовании.

Эта книга предназначена для того, чтобы заложить прочную основу в навыках и инструментах, используемых для реверс-инжиниринга программного обеспечения и взлома. Начиная с основ, книга будет проходить через разделы, посвященные реверс-инжинирингу программного обеспечения и взлому, и завершится рассмотрением передовых методов нападения и защиты.

Глава 1
Декомпиляция и архитектура
Эффективный реверс-инженер или взломщик - это тот, кто понимает системы, которые он анализируют. Программное обеспечение предназначено для работы в определенной среде, и если вы не понимаете, как работает эта среда, вам будет трудно разобраться в программном обеспечении.

В этой главе рассматриваются шаги, необходимые для начала обратного проектирования приложения. Декомпиляция имеет решающее значение для преобразования приложения из машинного кода во что-то, что может быть прочитано и понято человеком. Чтобы действительно проанализировать полученный код, необходимо также понимать архитектуру компьютеров, на которых он предназначен для работы.

Декомпиляция
Большинство программистов пишут, используя языки программирования более высокого уровня, такие как C/C++ или Java, которые предназначены для чтения человеком. Однако компьютеры предназначены для выполнения машинного кода, который представляет инструкции в двоичном формате.

Компиляция - это процесс преобразования языка программирования в машинный код. Это означает, что декомпиляция будет процессом перевода машинного кода обратно на исходный язык программирования, восстановления исходного кода. Когда это доступно, это самый простой подход к обратному проектированию, поскольку исходный код предназначен для чтения и интерпретации человеком. Большая часть этой книги будет посвящена более типичному случаю, когда декомпиляция невозможна. Но в целях изучения важно понимать, что иногда вы можете выполнить декомпиляцию обратно к исходному коду, и когда это возможно, вам следует воспользоваться этим.

Когда полезна декомпиляция?
Для многих языков программирования полная декомпиляция невозможна. Эти языки преобразуют код непосредственно в машинный код, и некоторая информация, такая как имена переменных, теряется в процессе. Хотя некоторые продвинутые декомпиляторы могут создавать псевдокод для этих языков, процесс не идеален.

Однако некоторые языки программирования используют так называемую JIT-компиляцию. Когда программы, написанные на JIT-языках, «собираются», они преобразуются из исходного кода в промежуточный язык (IL), а не в машинный код. JIT-компиляторы хранят копию кода в этом IL до запуска программы, после чего код преобразуется в машинный код. Примерами JIT-языков являются Java, Dalvik (Android) и .NET.

Например, Java хорошо известна тем, что в значительной степени не зависит от платформы, и причиной этого является использование IL (байт-кода Java) и виртуальной машины Java (JVM). Распространяя программный код в виде байт-кода и компилируя его только во время выполнения, JVM преобразует Java IL в машинный код, специфичный для машины, на которой он запущен. Хотя такой подход может негативно повлиять на размер файла и производительность, он окупается переносимостью.

JIT-компиляция также значительно упрощает обратное проектирование этих приложений. Эти промежуточные языки достаточно похожи на исходный код, чтобы их можно было декомпилировать или преобразовать обратно в пригодный для использования исходный код. Исходный код разработан таким образом, чтобы быть удобочитаемым человеком, что

значительно облегчает понимание логики приложения и выявление программных средств защиты или других встроенных секретов.

Декомпиляция JIT-языков программирования
Для JIT-языков, таких как .NET, доступно несколько бесплатных декомпиляторов. Одним из широко используемых .NET-декомпиляторов является JetBrains dotPeek, который доступен по адресу www.jetbrains.com/decompiler. На рисунке 1.1 показан пример .NET-кода, декомпилированного в dotPeek.

Как показано на рисунке, .NET код легко читается после декомпиляции, поскольку промежуточный язык кодирует огромное количество информации в виде метаданных, что позволяет более точно реконструировать исходный код. Любая конфиденциальная информация или коммерческие секреты, содержащиеся в коде, легко доступны для реинжиниринга.

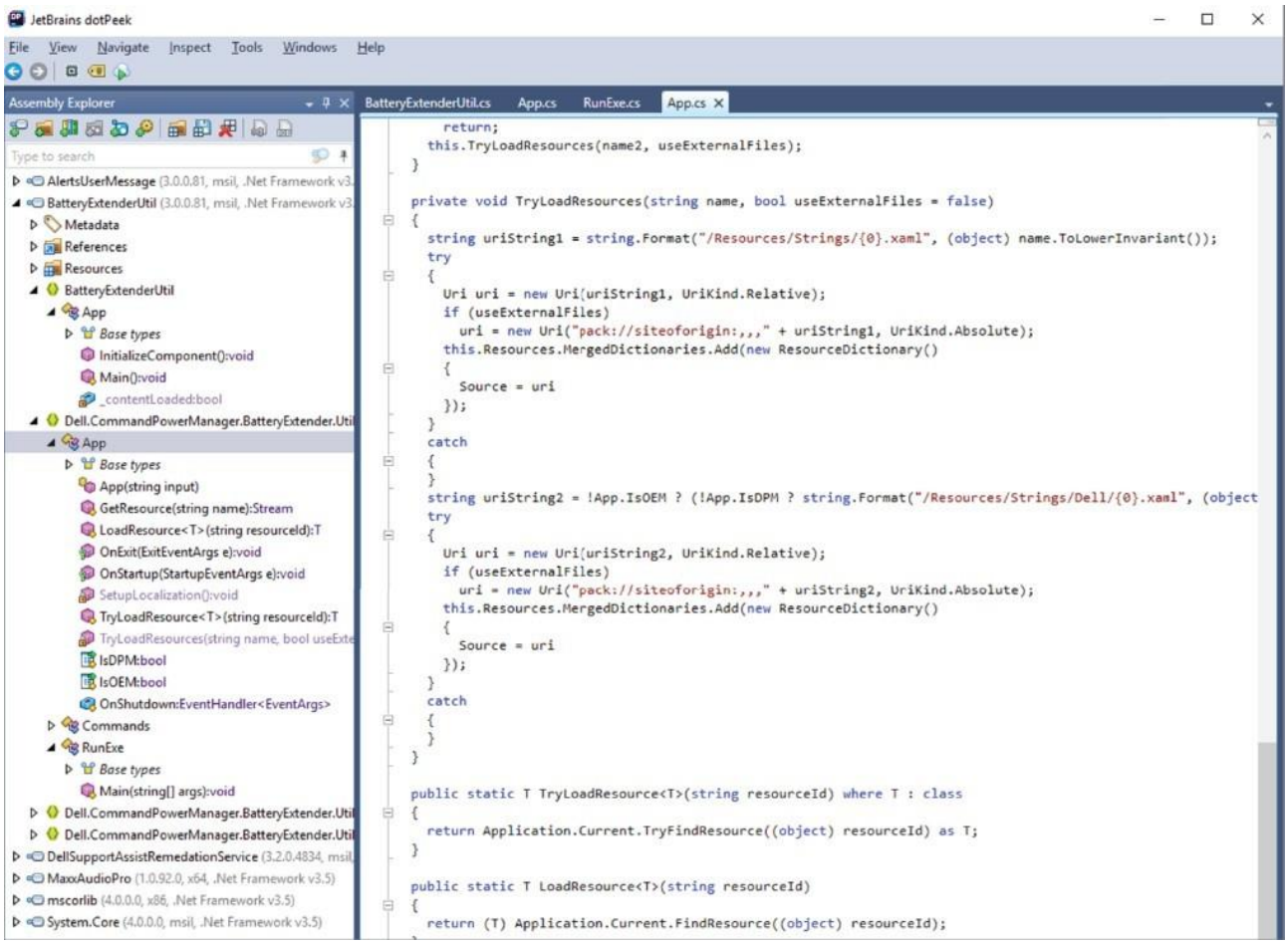


Рисунок 1.1: JetBrains dotPeek .NET декомпилятор

Защита JIT-языков
В отличие от программ с настоящим машинным кодом, JIT-скомпилированные программы часто могут быть преобразованы в исходный код. Снижение планки для обратного проектирования кода делает многие средства защиты от обратного проектирования x86, обсуждаемые в последующих главах, ненужными и избыточными.

Для декомпилируемых языков обычно используемой защитой от обратного проектирования является обфускация. На рисунке 1.2 показан пример .NET-приложения до и после обфускации.

Верхняя половина рисунка содержит код до того, как произойдет запутывание, где имена функций, переменных и строки легко читаются. Информация, содержащаяся в этих именах переменных, облегчает реверс-инженеру понимание назначения каждой функции и того, как работает приложение в целом.

В нижней половине изображения мы видим запутанную версию того же кода. Теперь имена функций, переменных и строк искажены, что значительно затрудняет понимание назначения показанной функции, не говоря уже о приложении в целом.

Еще одна важная рекомендация в области безопасности - избегать написания кода, критически важного для безопасности или конфиденциальности, на JIT-языках, где обратная инженерия проста. Вместо этого напишите этот код на ассемблерном языке, таком как C/C++, где обратная инженерия значительно сложнее. Этот код может быть включен в библиотеки DLL, которые связаны с исполняемым файлом, содержащим нечувствительный код, написанный на языке JIT.

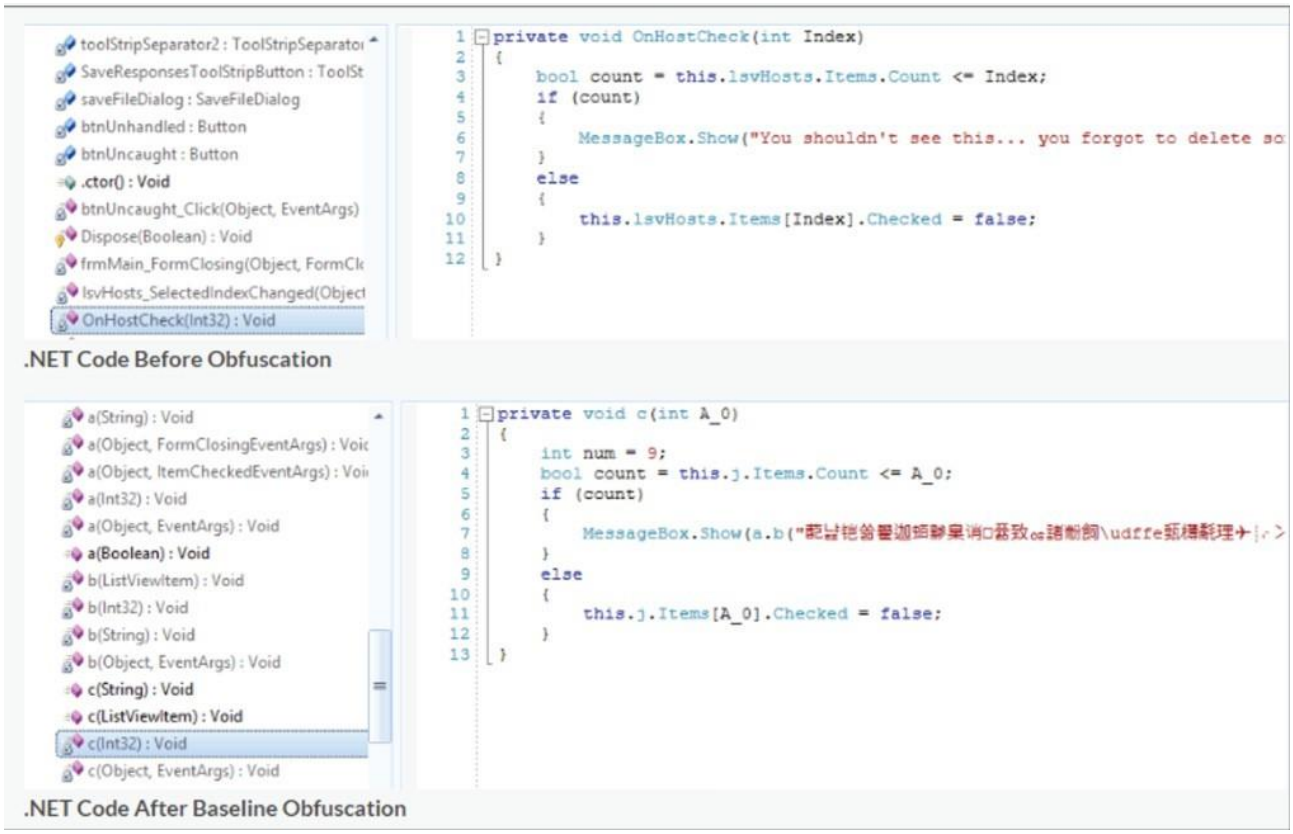


Рисунок 1.2: Обфускация в JetBrains dotPeek

Лабораторная работа 1: Декомпиляция

Это первая практическая лабораторная работа в этой книге. Лабораторные работы и все связанные с ними инструкции можно найти в соответствующей папке здесь: https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
Для этой лабораторной работы, пожалуйста, найдите раздел "Лабораторная декомпиляция" и следуйте приведенным инструкциям.

Навыки для практики

Каждая лабораторная работа в этой книге предназначена для обучения определенным навыкам и предоставления практического опыта. Навыки для практики в этой лабораторной включают следующее:

- Декомпиляция
- Выполнение вводного обратного проектирования

Чтобы освоить эти навыки, вы будете использовать JetBrains dotPeek для обратного проектирования и модификации .NET-приложения.

Выводы
Декомпиляция - это мощный и простой подход к пониманию и модификации программы. Однако он работает не в каждой программе. Хотя программы, написанные на таких языках, как C/C++, могут быть декомпилированы с помощью таких инструментов, как IDA Hex-Rays Decompiler или Ghidra, результат часто получается некачественным и сложным в использовании.

При разработке приложений, содержащих конфиденциальную информацию или которые вы не хотите изменять, лучше использовать язык, который нелегко декомпилировать. Например, C/C++ является лучшим выбором для конфиденциальной функциональности, чем язык .NET, такой как C#.

Архитектура
Декомпиляция - это простой подход к обратному проектированию, поскольку он возвращает вас к языкам более высокого уровня и логическим структурам. Однако этот простой путь не часто доступен. Для языков, которые строятся на машинном коде, нам нужно углубиться и понять, как работают компьютерные архитектуры, машинный код и ассемблерный код.

Компьютерная архитектура
Обычно считается, что среднестатистическому программисту не требуется глубокого понимания того, как работают компьютеры. При написании программы на процедурном языке операционная система обрабатывает все низкоуровневые операции. Программа отображается как процесс, который имеет доступ к процессору, памяти и файловой системе всякий раз, когда они ему нужны. Процессы, по-видимому, имеют свои собственные непрерывные области памяти, а файлы - это просто последовательность байтов для чтения и записи.

Однако на самом деле ничего из этого не соответствует действительности, и ваша операционная система скрывает от вас правду (чтобы упростить программирование). Глубокое понимание того, как на самом деле работает компьютерная архитектура, важно для реинжиниринга. На рисунке 1.3 показаны основные компоненты, из которых состоит компьютер, включая центральный процессор, мост, память и периферийные устройства.

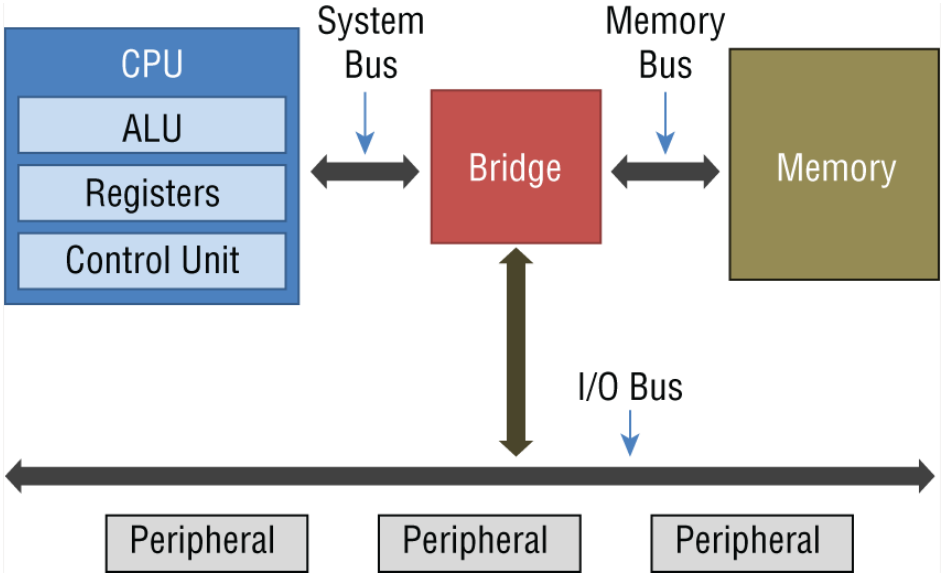


Рисунок 1.3: Архитектура компьютера

Центральный процессор

Центральный процессор (CPU) - это место, где происходит обработка на компьютере. Внутри центрального процессора находятся следующие компоненты:

Арифметико-логический блок (ALU): ALU выполняет математические операции внутри компьютера, такие как сложение и умножение.
Регистры: Регистры выполняют временное хранение данных и используются в качестве основных входов и выходов команд x86. Регистры обеспечивают чрезвычайно быстрый доступ к одному слову данных и обычно доступны по имени.
Блоки управления: Блоки управления выполняют код. Это включает в себя чтение инструкций и организацию работы других элементов в компьютере.
Мосты и периферийные устройства
Центральный процессор подключен через шину к мосту. Назначение моста заключается в подключении центрального процессора к другим компонентам системы, включая память и шину ввода-вывода, по которой к системе подключаются периферийные устройства, такие как клавиатура, мышь и динамики. В то время как информация передается по шине, мост отвечает за управление этим трафиком и обеспечение того, чтобы трафик, поступающий по одной шине, перенаправлялся по соответствующей шине.

Периферийные устройства, подключенные через шину ввода-вывода, позволяют компьютеру взаимодействовать с внешним миром. Это включает отправку и прием данных с видеокарты, клавиатуры, мыши, динамиков и других систем.

Память и регистры

Как следует из названия, память - это место, где хранятся данные на компьютере. Данные хранятся в виде линейной последовательности байтов, доступ к которым осуществляется по их адресу. Такая конструкция обеспечивает умеренно быстрый доступ к данным, хранящимся в системе.

Когда программа хочет получить доступ к данным в памяти, центральный процессор отправляет запрос по шине на мост, который перенаправляет его в память, где осуществляется доступ к данным по указанному адресу. Затем запрошенные данные должны пройти по этому маршруту и вернуться в центральный процессор, прежде чем они смогут быть использованы программой. Напротив, регистр физически расположен внутри

центрального процессора, что делает его гораздо более доступным.

Регистры - это хранилище, которое находится внутри центрального процессора и, в отличие от памяти, не представляет собой линейный ряд байтов. Регистры имеют специальные имена и заданные размеры, связанные с каждым из них.

Регистры и память служат одной и той же цели: они хранят данные. Однако у них разная специализация (качество в сравнении с количеством). Регистров немного и они дороги, но они обеспечивают чрезвычайно быстрый доступ к данным. Память дешевая и в большом количестве, но обеспечивает более низкую скорость доступа.

Основная часть данных, связанных с программой, сам код и его данные будут храниться в памяти. Во время выполнения программы небольшие фрагменты данных будут скопированы в регистры для обработки.

Система счисления

Компьютеры работают на двоичной, цифровой логике. Все либо включено (1), либо выключено (0). Сюда входят программы, запущенные на компьютере. Все языки высокого уровня в конечном итоге преобразуются в последовательность битов, называемую машинным кодом. Этот машинный код определяет набор инструкций, выполняемых компьютером для выполнения желаемой функции.

Введение в машинный код

Каждый программист начинает изучение языка с программы “hello world”. В x86 машинный код для “hello world” выглядит следующим образом:

55 89 e5 83 e4 f0 83 ec 10 b8 b0 84 04 08 89 04 24 e8 1a ff ff ff b8 00 00 00 00 c9 c3 90

Этот машинный код написан в шестнадцатеричном формате для удобства чтения и компактности, но его истинное значение представляет собой двоичную строку из 1 и 0. Эта двоичная строка содержит инструкции по переключению транзисторов для вычисления информации, извлечению данных из памяти, передаче сигналов по системным шинам, взаимодействию с видеокартой и, наконец, распечатке текста “hello world”. Если эта строка символов кажется немного короткой для выполнения всего этого, то это потому, что эти инструкции запускают операционную систему (в данном примере Linux) на помощь.

Машинный код управляет процессором на максимально детализированном уровне. Некоторые из функций, которые выполняет машинный код, включают следующее:

- Перемещение данных в память и из нее
- Перемещение данных в регистры и из них
- Управление системной шиной
- Управление ALU, блоком управления и другими компонентами

Такое низкоуровневое управление означает, что приложения, написанные в машинном коде, могут быть невероятно мощными и эффективными. Однако, хотя запоминание и ввод различных последовательностей битов для выполнения определенных задач довольно увлекательны, они неэффективны и подвержены ошибкам.

От машинного кода к сборке

В машинном коде последовательность битов представляет конкретное действие. Например, 0x81 или 10000001 - это команда, которая складывает два значения вместе и сохраняет результат в определенном месте.

Ассемблерный код разработан таким образом, чтобы представлять собой удобочитаемую версию машинного кода. Вместо запоминания двоичной или шестнадцатеричной строки, такой как 0x81 или 10000001, программист может использовать add. Мнемоника add сопоставлена с 0x81, так что это сокращение упрощает программирование без потери каких-либо преимуществ написания машинного кода.

Перевод машинного кода в ассемблерный значительно облегчает его понимание. Например, предыдущий пример кода “hello world” можно преобразовать в серию понятных инструкций.

MACHINE CODE	ASSEMBLY
55	push ebp
89 e5	mov ebp,esp
83 e4 f0	and esp, 0xfffffff0
83 ec 10	sub esp, 0x10
b8 b0 84 04 08	mov eax
89 04 24	mov [esp], eax
e8 1a ff ff ff	call 80482f4
b8 00 00 00 00	mov eax, 0x0
c9	leave
c3	ret
90	nop

Если вы разбираетесь в машинном коде, писать непосредственно в нем может быть интересно, и есть случаи, когда это может иметь смысл. Однако в большинстве случаев это неэффективно и непрактично. Написание на ассемблере дает те же преимущества, что и написание в машинном коде, но гораздо практичнее.

После того, как код был написан на ассемблере, он может быть переведен в машинный код с помощью ассемблера в процессе, называемом сборкой. Программа, уже находящаяся в машинном коде, может быть разобрана на ассемблерный код с помощью дизассемблера.

Определение

Ассемблеры преобразуют ассемблерный код в машинный код. Дизассемблеры преобразуют машинный код в ассемблер.

Многие программисты не пишут на машинном коде или ассемблере. Вместо этого они используют языки более высокого уровня, которые абстрагируют больше деталей. Например, следующий псевдокод аналогичен многим процедурным языкам высокого уровня.

int x=1, y=2, z=x+y;
В процессе компиляции эти языки более высокого уровня преобразуются в ассемблерный код, аналогичный следующему:

```
mov [ebp-4], 0x1
mov [ebp-8], 0x2
mov eax, [ebp-8]
mov edx, [ebp-4]
```

```
lea eax, [edx+1*eax]
mov [ebp-0xc], eax
```
Затем можно использовать ассемблер для преобразования ассемблерного кода в следующий машинный код, который может использовать компьютер:

```
c7 45 fc 01 00 00 00 c7 45 f8 02 00 00 00 8b 45 f8 8b 55 fc 8d 04 02 89 45 f4
```

Архитектуры наборов команд и микроархитектуры

Слово "компьютер" охватывает широкий спектр систем. Умные часы и настольный компьютер работают схожим образом. Однако их внутренние компоненты могут существенно отличаться.

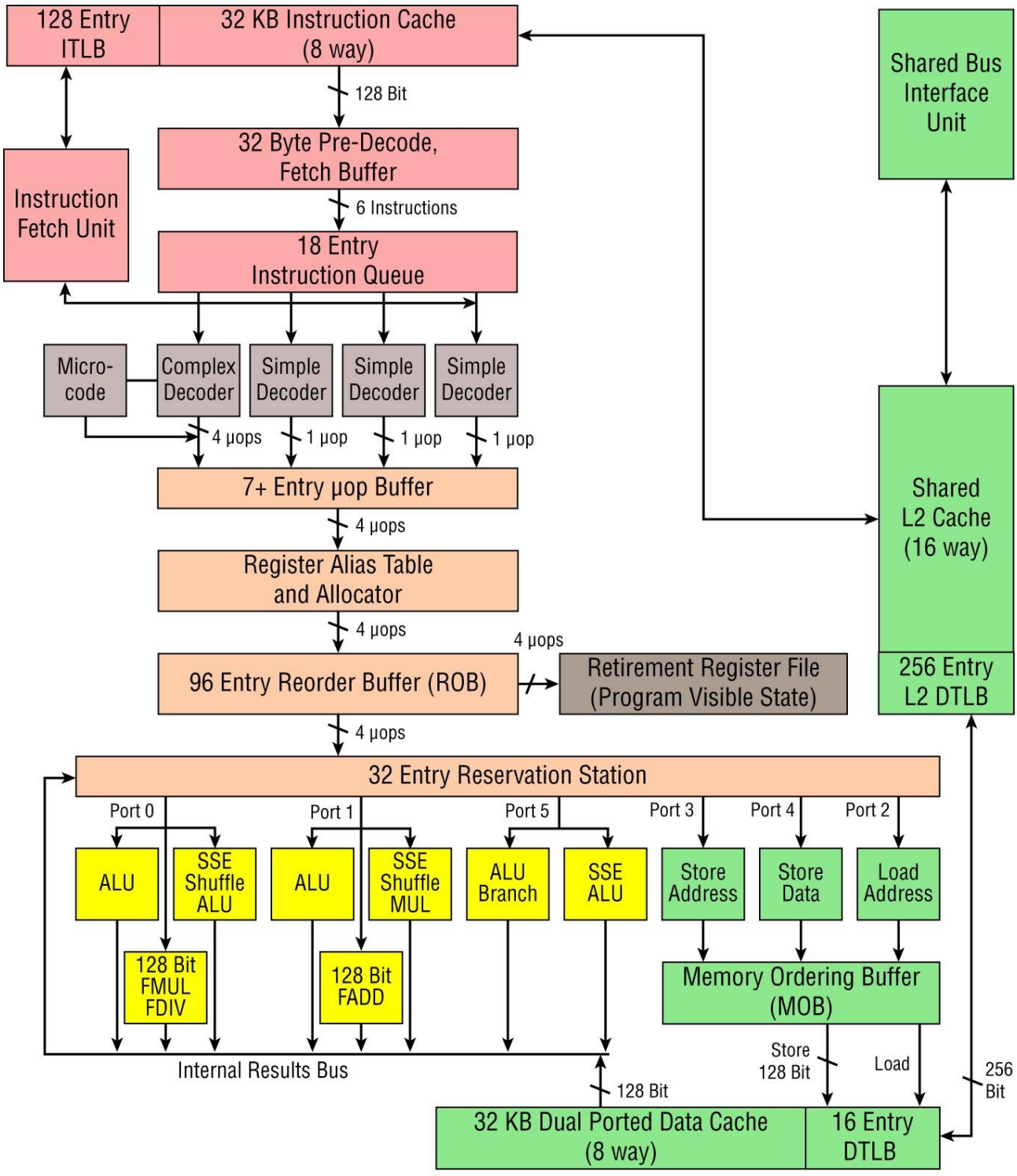
Архитектура набора инструкций (ISA) описывает экосистемы, в которых выполняются программы. Некоторые из факторов, определяемых ISA, включают следующее:

- **Регистры**: ISA определяет, имеет ли процессор один регистр или сотни. Он также определяет размер этих регистров, содержат ли они 8 бит или 128 бит.
- **Адреса и форматы данных**: ISA определяет формат адресов, используемых для доступа к данным в памяти. Он также определяет, сколько байт система может извлекать из памяти за один раз.
- **Машинные инструкции**: Разные ISA могут поддерживать разные наборы инструкций. ISA определяет, поддерживаются ли сложение, вычитание, равенство, остановка и другие инструкции.

Определяя возможности физической системы, ISA также косвенно определяет язык ассемблера. ISA определяет, какие низкоуровневые инструкции доступны и что эти инструкции выполняют.

Микроархитектура описывает, как конкретный ISA реализован на процессоре. На рисунке 1.4 показан пример архитектуры Intel Core 2.

Вместе ISA и микроархитектура определяют компьютерную архитектуру. Существование тысяч ISA и тысяч микроархитектур означает, что существуют также тысячи компьютерных архитектур.



Intel Core 2 Architecture

Рисунок 1.4: Архитектура Intel Core 2

Определение

Архитектура набора команд определяет, как работают регистры, адреса, форматы данных и машинные инструкции. Микроархитектуры реализуют ISA на процессоре. Вместе ISA и микроархитектура определяют архитектуру компьютера.

Компьютерные архитектуры RISC и CISC

Несмотря на то, что существуют тысячи компьютерных архитектур, их можно в общих чертах разделить на две основные категории. Архитектуры вычислений с сокращенным набором команд (RISC) определяют небольшое количество более простых инструкций. В целом, архитектуры RISC дешевле и проще в создании, а аппаратное обеспечение физически меньше и потребляет меньше энергии.

Напротив, архитектура complex instruction set computing (CISC) определяет большее количество более мощных инструкций. Процессоры CISC более дороги и сложны в создании, а также, как правило, больше по размеру и потребляют больше энергии.

Хотя архитектуры CISC могут показаться объективно хуже, чем архитектуры RISC, их основное преимущество заключается в простоте и эффективности программирования. Например, рассмотрим гипотетический пример, когда программа хочет умножить значение на 5 в системе RISC по сравнению с системой CISC.

CISC	RISC
`mul [100], 5`	`load r0, 100`
	`mov r1, r0`
	`add r1, r0`
	`add r1, r0`
	`add r1, r0`
	`add r1, r0`
	`mov [100], r1`

В этом примере процессор CISC может выполнить вычисление за одну команду, если у него есть операция умножения, которая может загружать значение из памяти, умножать его и сохранять результат в том же месте памяти. Однако в RISC-процессоре может отсутствовать оператор умножения, поскольку это сложная операция. Вместо этого RISC загружает значение из памяти, добавляет его к себе четыре раза и сохраняет результат в одной и той же ячейке памяти в течение семи шагов.

Архитектуры RISC и CISC имеют свои преимущества, недостатки и варианты использования. Например, оператору RISC может потребоваться 100 инструкций для выполнения той же операции, которую оператор CISC может выполнить за одну. Однако выполнение одной операции CISC может занять в 100 раз больше времени или в 100 раз больше мощности.

Сегодня широко используются как RISC, так и CISC-наборы команд. Некоторые примеры широко используемых RISC-архитектур включают следующее:

ARM (используется телефонами, планшетами)
MIPS (используется встроенными системами и сетевым оборудованием)
PowerPC (используется оригинальными компьютерами Mac и Xbox360)
В этой книге мы сосредоточимся на языке ассемблера x86, который представляет собой архитектуру CISC. Эта архитектура используется на всех современных ПК и серверах и поддерживается всеми основными операционными системами (Windows, Mac, Linux) и даже некоторыми игровыми системами, такими как Xbox One. Что делает его одним из самых мощных для изучения методов взлома программного обеспечения.

Резюме

Машинный код, который фактически выполняется на компьютерах, не предназначен для чтения и понимания людьми. Чтобы его можно было использовать, его необходимо преобразовать в другую форму.

Одним из вариантов для этого является декомпиляция, которая приводит к результату, подобному или идентичному исходному коду. Однако декомпиляция не всегда возможна.

Для полностью скомпилированных языков, таких как C/C++ и многих других языков, необходимо разобрать скомпилированный исполняемый файл и проанализировать его в ассемблере. Однако это требует гораздо более глубокого понимания архитектуры компьютера и того, как он на самом деле работает, чем написание и чтение кода на языке более высокого уровня. Теперь, когда мы знаем, какую роль может играть декомпиляция и необходимость дизассемблирования, в следующих нескольких главах мы рассмотрим, как работают компьютеры, чтобы научиться дизассемблировать как профессионал.

Глава 2
Сборка x86: данные, режимы, регистры и доступ к памяти
Большая часть обратной разработки программного обеспечения требует дизассемблирования скомпилированного исполняемого файла и анализа результата. Результатом такой дизассемблирования является ассемблерный код, а не язык более высокого уровня.

Хотя существует несколько языков ассемблера, x86 является одним из наиболее широко используемых. В этой главе представлены некоторые ключевые концепции ассемблера x86, обеспечивающие основу для последующих глав.

Введение в x86
Существуют тысячи компьютерных архитектур. Хотя все они работают одинаково, компьютер есть компьютер, но между ними есть незначительные или существенные различия.

Чтобы изучить реверс-инжиниринг, нам нужно выбрать архитектуру, на которой мы сосредоточимся. В этой книге мы будем использовать x86, который был выбран по нескольким причинам:

- Повсеместность: x86 является наиболее широко используемым языком ассемблера, что делает его широко применимым для обратного проектирования.
- Компьютерная поддержка: приложения x86 могут быть созданы, запущены и подвергнуты обратному проектированию на любом настольном компьютере, ноутбуке или сервере.
- Доля рынка: x86 является ядром основных операционных систем (Windows, Linux и macOS), поэтому используется в миллиардах систем.
Архитектура x86 существует уже несколько десятилетий и за эти годы значительно эволюционировала. Впервые он был представлен в 1974 году компанией Intel, и некоторые из основных вех в истории x86 включают следующее:

- Intel 8080: 8-разрядный микропроцессор, представленный в 1974 году
- Intel 8086: 16-разрядный микропроцессор, представлен в 1978 году
- Intel 80386: 32-разрядный микропроцессор, представлен в 1985 году
- Intel Prescott, AMD Opteron и Athlon 64: 64-разрядные микропроцессоры, представленные в 2003/2004 годах
За свою почти 50-летнюю историю архитектура x86 регулярно добавляла новые функции, сохраняя при этом обратную совместимость. Даже если функция была признана неиспользуемой, она никогда не удалялась из системы. В результате программы, написанные для процессора Intel 8086, выпущенного в 1978 году, все еще могут работать на новейших чипах x86 без каких-либо изменений.

Такой акцент на обратной совместимости позволил создать огромную, сложную и интересную архитектуру. Последнее руководство разработчика программного обеспечения Intel (www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html) содержит более 5000 страниц и лишь поверхностно описывает возможности этой архитектуры. Эта книга посвящена пониманию основ x86, а это все, что необходимо для чтения, записи и манипулирования большинством x86-кода.

Поскольку архитектура x86 изменилась, термин x86 стал общим термином для всех архитектур, которые развились из 16-разрядной архитектуры Intel 8086. Это включает архитектуру Intel 80286, которая содержит как 16-разрядную, так и 32-разрядную архитектуры, и архитектуру Intel 80886, которая добавляет 64-разрядную архитектуру. Термин x64 конкретно относится к 64-разрядной версии x86.

В этой книге будут приведены примеры в 32-разрядной архитектуре x86. Все концепции из 32-разрядной архитектуры x86 точно переводятся в x64. По мере изучения значительно проще работать с примерами в 32-разрядной архитектуре по сравнению с 64-разрядной. Изучив 32-разрядный x86 на протяжении всей этой книги, вы сразу же сможете взглянуть на x64-разрядную сборку и разобраться в ней. Однако ваши глазные яблоки будут благодарны вам за то, что вам не нужно постоянно смотреть на 64 бита, поскольку даже на 32 бита смотреть немного больно. Итак, не позволяйте примерам, написанным в 32-разрядной версии, заставить вас задуматься о том, что это устарело или что вам следует сосредоточиться на 64-разрядной версии. Мы оба сначала изучили 32-разрядную версию, и мы многому научились взламывать программное обеспечение и можем с уверенностью сказать, что если вы сначала освоите прочную 32-разрядную основу, 64-разрядная версия станет просто еще несколькими именами регистров и более длинными значениями.

Синтаксис ассемблера

Выбор x86 из тысяч возможных компьютерных архитектур важен, но этого недостаточно. Хотя архитектура набора команд (ISA) определяет такие факторы, как регистры, формат данных и машинные инструкции, она не определяет синтаксис.

Пока язык ассемблера следует всем правилам для регистров, адресации и т.д. и определяет правильный набор инструкций, он является допустимым языком x86. Например, язык x86 должен иметь операцию умножения. Однако его мнемоническим обозначением может быть mul, MUL, multiply или любая другая вариация на любом языке.

Синтаксис языка ассемблера полностью определяется ассемблером. Не существует стандартного синтаксиса для языка ассемблера в целом или для сборки x86 в частности. В результате существуют сотни различных вариаций.

Однако есть два распространенных варианта синтаксиса x86, которые, как вы обнаружите, используются большинством инструментов сборки x86: синтаксис AT&T и синтаксис Intel. В каждой из этих основных ветвей представлены сотни вариантов, специфичных для ассемблера.

Хотя Intel и AT&T assembly оба являются x86, они выглядят очень по-разному. Например, рассмотрим инструкцию, предназначенную для перемещения памяти по адресу ebx+4*ecx+2020 в регистр eax.

Эта инструкция выглядит очень по-разному в синтаксисах Intel и AT&T:

СИНТАКСИС INTEL AT&T SYNTAX

INTEL SYNTAX	AT&T SYNTAX
`mov eax, [ebx+4*ecx+2020]`	`mov 0x7e4(%ebx,%ecx,4),%eax`

В синтаксисе Intel после команды mov следует местоположение, в котором будет сохранен результат. Доступ к памяти обозначен квадратными скобками, и вычисление адреса памяти [ebx+4*ecx+2020] выполняется внутри этих скобок.

Синтаксис AT&T отличается от синтаксиса Intel несколькими способами:

- Упорядочение: Аргументы меняются местами, поэтому местоположение назначения указано вторым.

- Регистры: AT&T указывает регистры, используя знак процента (%), в то время как Intel этого не делает.
- Доступ к памяти: AT&T использует круглые скобки для обозначения доступа к памяти, в то время как Intel использует квадратные скобки.
- Вычисление: Вычисление желаемого адреса памяти выглядит очень по-разному в синтаксисе AT&T и Intel.
- Инструкции: Хотя здесь они не показаны, AT&T часто использует другую, более длинную мнемонику инструкций, чем Intel.

Для ясности и последовательности в примерах, приведенных в этой книге, был выбран синтаксис Intel. Вот некоторые из причин выбора Intel вместо AT&T:

- Поддержка Intel: Intel является ведущим разработчиком процессоров, и они используют синтаксис Intel.
- Использование инструментов: Большинство основных инструментов реверс-инжиниринга, таких как IDA, используют синтаксис Intel.
- Удобочитаемость: Синтаксис Intel широко считается более чистым и простым для чтения и записи, чем синтаксис AT&T.

Представление данных

В отличие от людей, компьютеры работают в двоичном формате, поэтому большинство инструментов обратного проектирования не отображают числа в системе base-10. Чтобы понять, что делает приложение, необходимо понимать данные, которые оно обрабатывает, и как эти данные могут быть представлены.

Основы системы счисления

База в системе счисления определяет количество символов, используемых для представления значения цифры. Большинство людей выполняют математические вычисления в базе 10, в которой используются символы 0, 1, 2, 3, 4, 5, 6, 7, 8, и 9.

Однако это не единственный вариант. Можно использовать любую базу, если у вас достаточно символов для представления значений. Например, база 5 использует символы 0-5, а база 8, также известная как восьмеричная, использует символы 0-7.

Совет

Основание, в котором записано число, может быть обозначено нижним индексом. Например, 1010 записывается по основанию 10, в то время как 102 является двоичным числом (по основанию 2).

В каждой базе данных нам нужна возможность представлять значения, превышающие базовое число. Для этого мы используем несколько цифр. Например, подсчет в базе 10 выполняется следующим образом ... 8, 9, 10, 11, ..., 98, 99, 100, 101, ... В базе 16 отсчет идет... 8, 9, a, b, c, d, e, f, 10, 11, ... 19, 1a, ... 1f, 20, ...

Компьютеры представляют собой двоичные системы и выполняют все свои операции по хранению и обработке данных, используя единицы и 0. Однако они неэффективны и быстро становятся громоздкими для записи. Например, значение 201410 равно 111110111102.

В то время как компьютеры работают с двоичным кодом, значения часто отображаются инструментами в шестнадцатеричном формате или “hex” для удобства чтения. Значения, записанные в шестнадцатеричном формате, могут быть обозначены нижним индексом (1d16) с префиксом 0x (0x1d) или суффиксом h (1dh).

Одним из преимуществ шестнадцатеричной системы счисления (основание 16) является то, что шестнадцатеричная система счисления имеет степень 2. Это означает, что значения могут

быть легко преобразованы из двоичных в шестнадцатеричные с помощью замены символов. На рисунке 2.1 показано, как каждый шестнадцатеричный символ соответствует основанию 10 и основанию 2 (двоичному).

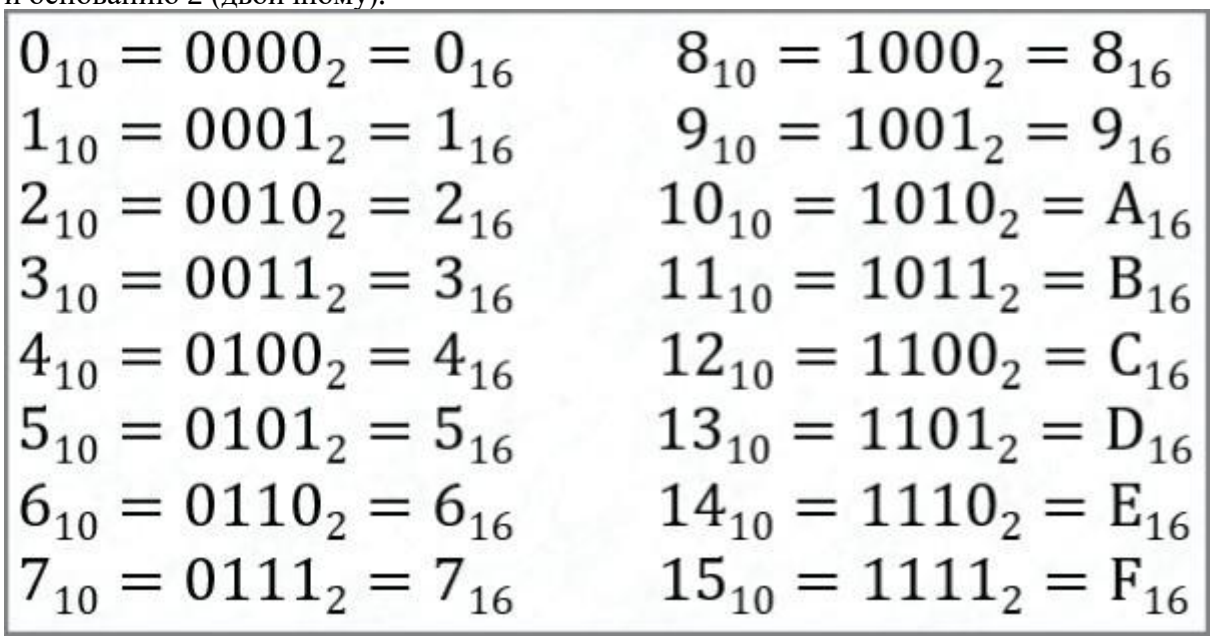


Рисунок 2.1: Шестнадцатеричный формат

Для примера рассмотрим двоичное значение 111110111102. Каждая шестнадцатеричная цифра представляет собой четыре двоичные цифры, поэтому это значение можно разбить на три части, начиная справа: 111, 1101 и 1110. Из рисунка мы видим, что эти фрагменты равны шестнадцатеричным цифрам 7, d и e, и все значение может быть представлено как 0x7de.

111110111102	Двоичное число
111 1101 11102	Разбивается на 4-разрядные фрагменты справа налево
7 d e	Каждый фрагмент преобразуется в шестнадцатеричный формат
0x7de	Результирующее шестнадцатеричное значение

Хотя эти преобразования можно выполнить вручную, часто быстрее и точнее использовать инструмент. На рисунке 2.2 показан пример выполнения базовых преобразований с помощью калькулятора Windows.

Биты, байты и слова

Биты - это базовая единица измерения, используемая компьютерами. Однако они слишком малы, чтобы обеспечить большую полезность. Вместо доступа к отдельным битам и их обработки компьютеры используют байты в качестве наименьшей единицы памяти. Во всех современных системах байт содержит 8 бит.

Хотя байты больше битов, они также слишком малы для многих операций. Компьютеры спроектированы таким образом, чтобы оптимально получать доступ к определенному количеству байтов за раз. Это количество байтов называется словом, обычно имеет степень 2

и может варьироваться в зависимости от компьютера. Например, микроконтроллеры имеют небольшие размеры слов, часто используя слова, содержащие 1 или 2 байта (8 или 16 бит). На компьютерах общего назначения размер слова обычно составляет 4 или 8 байт (32 или 64 бита).

Биты, байты и слова - наиболее важные термины, которые необходимо знать при работе с памятью, но они не единственные. Полный список распространенных терминов выглядит следующим образом:

- Бит: двоичная цифра, 0 или 1
- Байт: 8 бит
- Фрагмент: 4 бита
- Двухбайтовый: 16 бит
- Четырехбайтовый: 32 бита
- Слово: Зависит от архитектуры, некоторое количество байт
- Полуслово: Половина слова
- Двойное слово (DWORD): Два слова
- Четырехсловный (QWORD): Четыре слова
- Восьмисложие, двойное четырехсловие (DQWORD): восемь слов

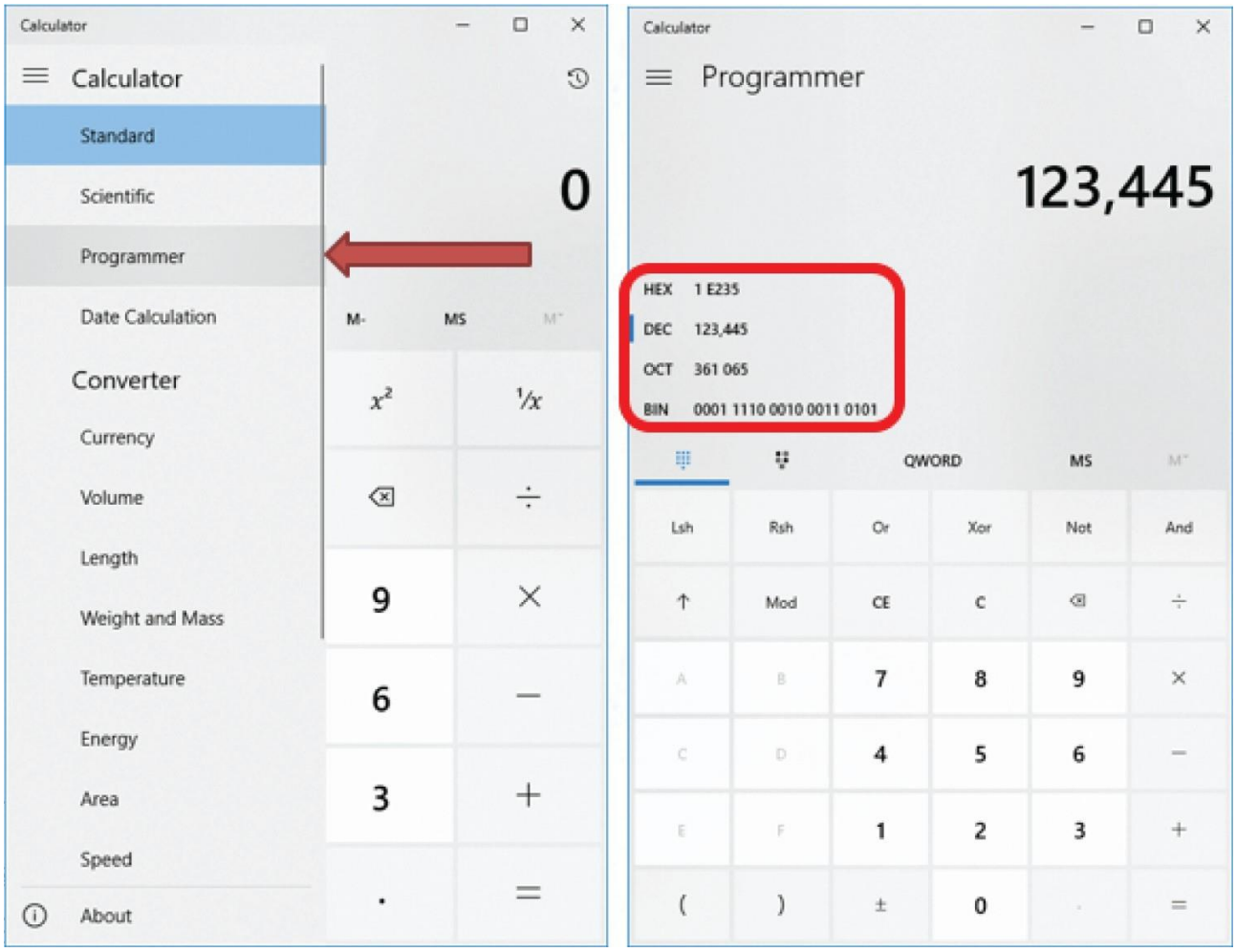


Рисунок 2.2: Базовые преобразования в калькуляторе Windows

Эта книга посвящена 32-разрядной архитектуре. В традиционной 32-разрядной архитектуре

это требовало бы, чтобы слово было 32-разрядным. Но это особенность архитектуры x86. Поскольку x86 сохранила свою обратную совместимость с оригинальной 16-разрядной архитектурой, слово на x86 составляет 16 бит, а двойное слово - 32 бита.

Совет
В 32-разрядной системе x86 байт равен 8 битам, а двойное слово - 32 битам.

Работа с двоичными значениями
Реверс-инжиниринг обычно включает в себя работу с большими двоичными числами, которые охватывают несколько разных байтов. При работе с этими числами необходимо понимать такие понятия, как расширение до нуля, значение битов и байтов и порядковый номер, чтобы правильно интерпретировать число, которое представляет двоичная строка.

Нулевое расширение и удобочитаемость
Двоичные значения обычно дополняются нулем или расширяются до размера слова архитектуры. В 32-разрядной архитектуре это означает добавление 0 слева от значения до тех пор, пока его длина не достигнет 32 бит. Например, значение 110012 было бы дополнено нулем до 00000000 00000000 00000000 000110012.

Обратите внимание, что биты также разбиты на группы по четыре или восемь для улучшения удобочитаемости. Это похоже на то, как запятые иногда добавляются каждые три цифры в базе 10 (т.е. 1000 вместо 1000). Когда значения записываются в шестнадцатеричном формате, они также группируются по байтам с двумя символами на байт. Например, значение 4D216 (эквивалентно 123410) может быть записано как 0416 D216.

Значение битов и байтов
Биты и байты в двоичном числе могут быть помечены на основе их относительного веса в числе. На рисунке 2.3 показаны некоторые из этих распространенных меток.

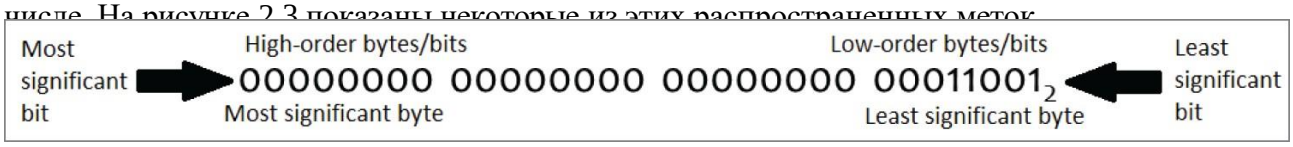


Рисунок 2.3: Метки значимости битов и байтов

В значении 00000000 00000000 00000000 000110012 младшим значащим битом (LSB) является тот, который находится в крайнем правом углу и имеет значение 1. Самый старший бит (MSB) находится в крайнем левом углу и имеет значение 0. При преобразовании из двоичного кода в базовый 10 MSB будет умножен на 231, в то время как LSB будет умножен на 20.

В дополнение к MSBs и LSBs, существуют также понятия наименьшего и наиболее значимого байтов. В значении 00000000 00000000 00000000 000110012 наименее значимый байт имеет значение 000110012, в то время как наиболее значимый байт имеет значение 000000002.

Биты и байты также могут быть помечены на основе их близости к концам значения. Например, биты и байты рядом с LSB считаются младшими по порядку, в то время как биты и байты рядом с MSB являются старшими по порядку.

Последовательность
В памяти данные хранятся в байтах. Однако многие типы данных используют несколько байтов. Например, значение int равно 32 битам или 4 байтам.

Порядок следования описывает порядок, в котором эти байты хранятся в памяти. В системе с младшим порядком байт с наименьшим значением сохраняется первым (по самому низкому адресу). В системе с большим порядком байт с наибольшим значением сохраняется первым (по самому низкому адресу).

Например, рассмотрим значение 133710, которое равно 0000 0000 0000 0000 0000 0000 0101 0011 10012 в двоичном формате или 0x00000539 в шестнадцатеричном. На рис. 2.4 показано, как эти значения будут храниться в памяти.

Address	1828	1829	1830	1831
Little Endian	0x39	0x05	0x00	0x00
Big Endian	0x00	0x00	0x05	0x39

Рисунок 2.4: Последовательность

Определение

В системе с младшим порядком байт наименьшего значения находится по наименьшему адресу. В системе с большим порядком байт наибольшего значения находится по наименьшему адресу.

Независимо от порядка следования в системе, адрес, связанный с переменной, является самым младшим используемым адресом или базовым адресом. Как в системе с малым, так и в системе с большим порядком следования в этом примере это будет адрес 1828.

В этой книге основное внимание уделяется архитектуре x86, которая имеет малозначительный порядок следования. В результате наименее значимый байт фрагмента данных будет расположен со смещением 0 от базового адреса. Вы заметите, что для людей это выглядит “задом наперед”, поскольку мы читаем и пишем с большой буквы.

Совет

x86 - это архитектура с меньшим порядком байтов, поэтому самый младший адрес содержит наименьший значащий бит.

Регистры

Регистры обеспечивают процессору высокоскоростной доступ к данным. Поскольку регистры физически расположены внутри центрального процессора, они имеют гораздо меньшую задержку, чем память, где запросы должны проходить через шины и мосты для доступа к данным.

В 32-разрядной архитектуре регистр содержит 32 бита данных и может обрабатываться как переменная в программе. Каждый регистр имеет уникальное имя, и данные внутри регистра могут быть изменены на основе вычислений или загрузки новых значений из памяти.

Основным ограничением регистров является то, что их количество ограничено и они должны

использоваться совместно всей программой. Если в программе заканчиваются регистры, ей необходимо начать сохранять информацию в памяти, что отрицательно сказывается на производительности. При ограниченном количестве регистров обычный цикл выполнения выглядит следующим образом:

- Извлекать данные из памяти и сохранять их в регистрах
- Работать с данными
- Сохранять данные обратно в память
- Повторять

Регистры в x86

Как мы упоминали ранее в обзоре архитектуры, регистры - это специальные имена и места в процессоре, которые позволяют выполнять очень быстрые операции. Все регистры можно разделить на две различные категории.

- Регистры общего назначения (GPRs): Используются для общего хранения данных, адресов и т.д. и доступны для непосредственного управления
- Регистры специального назначения (SPRs): Используются для хранения состояния программы

Архитектура x86 определяет множество регистров, которые показаны на рис. 2.5. Однако многие из них зарезервированы для использования самим процессором, что делает необходимым знать только их подмножество.

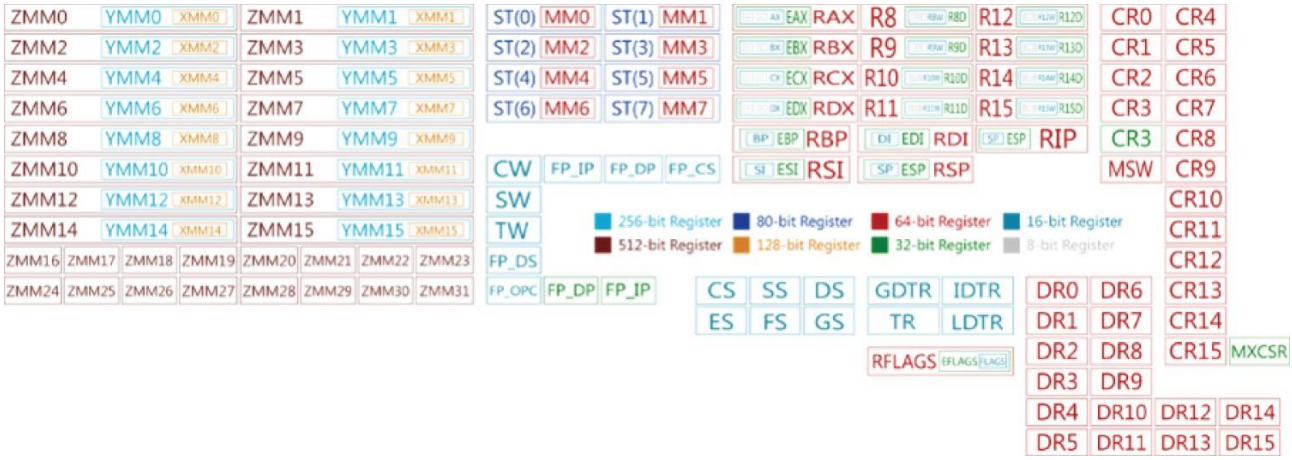


Рисунок 2.5: регистры x86

Источник: Liam McSherry/ wikimedia Commons /CC BY-SA 3.0.

Регистры общего назначения x86

GPRs выполняют большую часть тяжелой работы в приложении, сохраняя значения, извлеченные из памяти, выполняя манипуляции с данными и сохраняя результаты вычислений. Следующие GPRs являются наиболее значимыми в x86, и каждый из них может хранить 32 бита данных. У каждого аккумулятора есть роль, для которой он традиционно используется и в честь которой он назван; однако GPRs можно использовать для любых целей, и вы можете поместить счетчик в любой регистр, а не только в регистр ecx.

eax
eax - это “накопительный” регистр. Его название происходит от того факта, что он обычно используется для хранения результата арифметической операции. Например, программа может выполнять вычисление eax += ebx.

ebx
ebx - это “базовый” регистр. Обычно он используется для хранения базового адреса фрагмента памяти, используемого для хранения переменной. Например, выражение [ebx + 5] может использоваться для доступа к пятому элементу массива.

ecx
ecx является регистром “счетчика” и традиционно используется для подсчета. Например, ecx может использоваться для отслеживания текущей итерации цикла. В команде for (i=0; i<10; i++) переменная i, скорее всего, будет сохранена в регистре ecx.

edx
edx - это регистр “данных”. Его название происходит от того факта, что он обычно используется для хранения данных. Например, приложение может включать в себя инструкцию sub edx, 7.

esi
esi - это регистр “исходного индекса”. Он традиционно используется для хранения индекса в исходном массиве. Например, в команде array[i] = array[k] значение k, скорее всего, будет сохранено в esi.

edi
edi - это регистр “целевого индекса”. Он используется для хранения индекса в целевом массиве. Например, в команде array[i] = array[k] значение i, скорее всего, будет сохранено в edi.

ebp
ebp - это регистр “базового указателя”. Его назначение - хранить адрес базы текущего стекового фрейма. Концепции программного стека и стековых фреймов будут рассмотрены в последующих главах.

esp
esp - это регистр “указателя стека”. Он хранит адрес в верхней части текущего фрейма стека.

Регистры специального назначения
SPR предназначены для конкретных задач и не допускаются к прямому изменению. Например, команда mov eip, 1, которая использует SPR, не будет собираться, в то время как mov eax, 1, которая использует GPR, будет.

eip
eip - это регистр “указатель инструкции”. В нем хранится адрес следующей инструкции для выполнения.

eflags
eflags - это регистр “флагов”. Он хранит “флаги”, которые имеют значение true или false и содержат информацию о состоянии системы и результатах ранее выполненных инструкций.

Совет

GPRs доступны как для чтения, так и для записи, но SPR доступны только для чтения.

Работа с регистрами

В assembly GPRs можно рассматривать так же, как переменные, и обращаться к ним по имени. Например, команда mov eax, 1 сохраняет значение 1 в eax, в то время как add eax, ebx добавляет содержимое eax в ebx.

Обратите внимание, что все названия этих регистров начинаются с буквы e. Это связано с тем, что эти 32-разрядные регистры были “расширены” из исходных 16-разрядных регистров.

К нижней половине содержимого регистра можно получить доступ, удалив эту букву e из названия. Например, регистр ax содержит младшие 16 бит регистра eax.

Если имя регистра заканчивается на x (eax, ebx, ecx и edx), этот 16-разрядный регистр может быть дополнительно разделен на два 8-разрядных регистра, которые обозначаются l и h. al содержит 8 младших разрядов регистра ax, в то время как ah содержит 8 бит высокого порядка. Это проиллюстрировано на рисунке 2.6, где eax=0x01234567, ax=0x4567, ah=0x45 и al=0x67.

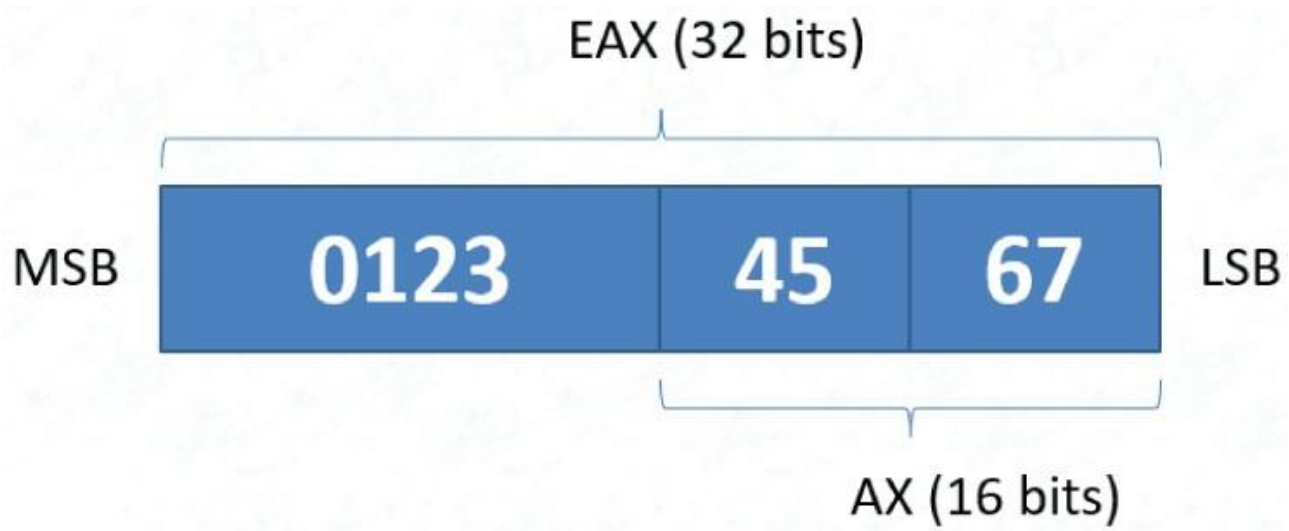


Рисунок 2.6: Фрагменты регистра eax

64-разрядные регистры

В 64-разрядной x86 все инструкции и поведение такие же, как и в 32-разрядной x86. Однако в 64-разрядных архитектурах регистров больше.

На рисунке 2.7 показаны обычно используемые регистры в 64-разрядной x86. В дополнение к 32-разрядным регистрам, описанным по-разному, 64-разрядные архитектуры включают в себя еще восемь регистров с маркировкой r8-r15.

Все 64-разрядные регистры также больше, чем их 32-разрядные аналоги. Для регистров, которые существуют в 32-разрядной x86, таких как eax, полный 64-разрядный аналог заменяет e на r, что делает регистр rax. Младшие 32 бита регистра затем доступны с использованием 32-разрядного имени, и использование таких имен, как ax, al и ah, остается неизменным.

Для новых регистров, таких как r8, 64-разрядный x86 обеспечивает доступ к младшим 32, 16 и 8 битам. Они обозначены как d (r8d), w (r8w) и b (r8b) соответственно, как показано на рисунке 2.8.

Доступ к памяти

32-разрядная (или 64-разрядная) система имеет только ограниченное количество доступных регистров. При игнорировании SPRs и GPRs, используемых для отслеживания стека (esp и ebp), у вас остается только шесть регистров, доступных для общих вычислений (eax, ebx, ecx, edx, esi и edi). Этого недостаточно, чтобы сделать многое, вот почему программа также должна иметь возможность считывать и записывать данные в память.

В синтаксисе сборки Intel x86 доступ к памяти обозначается с помощью обозначения []. Например, к данным, хранящимся по адресу 0x12345678, можно получить доступ с помощью [0x12345678]. Адреса памяти также могут храниться в регистрах, таких как инструкция [eax].

Указание длины данных

При обращении к данным из памяти необходимо знать не только адрес, по которому расположены данные, но и объем памяти, к которому требуется получить доступ. Например, инструкция [0x12345678] не указывает, требуется ли программе байт, слово, двойное слово или больше.

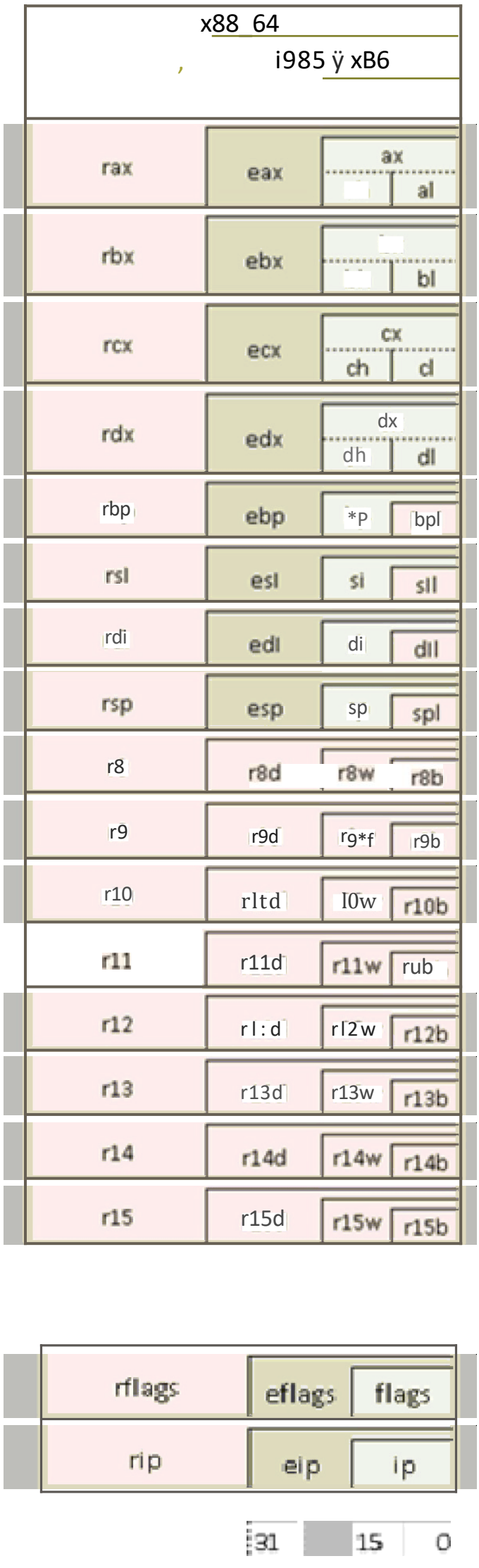

x88_64
i985 ÿ xB6
rax
eax
ax
al
rbx
ebx
bl
rcx
ecx
cx
ch
cl
rdx
edx
dx
dh
dl
rbp
ebp
*P
bpl
rsl
esl
si
sll
rdi
edl
di
dll
rsp
esp
sp
spl
r8
r8d
r8w
r8b
r9
r9d
r9*f
r9b
r10
rltd
l0w
r10b
r11
r11d
r11w
rub
r12
rl:d
rl2w
r12b
r13
r13d
r13w
r13b
r14
r14d
r14w
r14b
r15
r15d
r15w
r15b
rflags
eflags
flags
rip
eip
ip
31
15
0

Рисунок 2.7: Общие регистры x64

Источник: Bob Mon / Wikimedia Commons / CC BY-SA 4.0.

В некоторых случаях длина данных, к которым осуществляется доступ, может быть выведена из контекста. Например, в инструкции mov eax, [0x12345678] данные, извлекаемые из памяти, будут сохранены в eax. Поскольку eax является 32-разрядным регистром, программа должна запрашивать 32 бита данных.

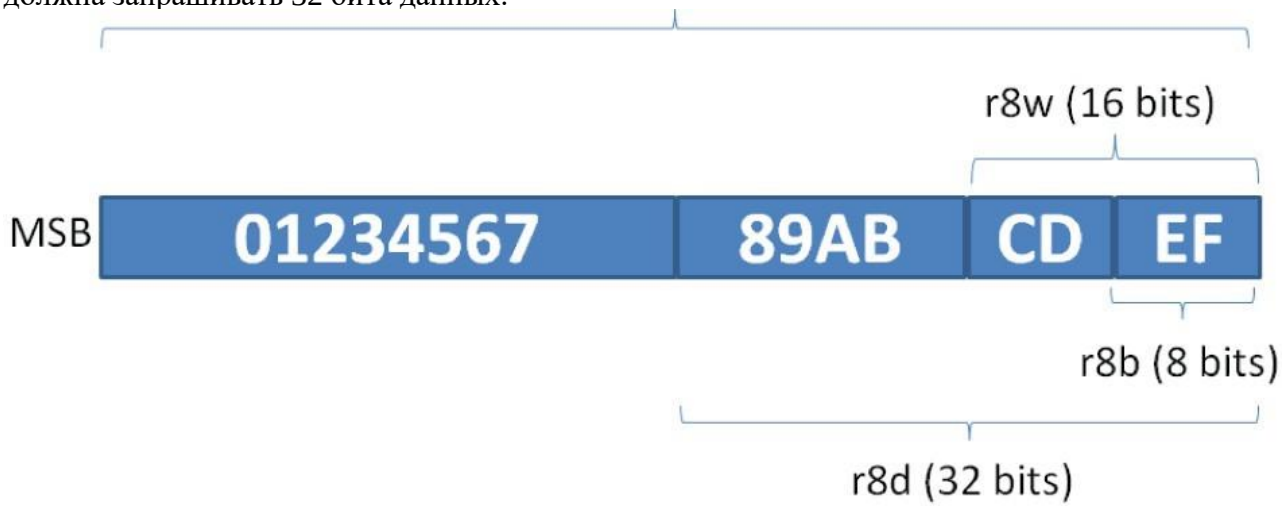


Рисунок 2.8: Фрагменты регистра r8

Это не всегда так. Например, рассмотрим команду movl [0x12345678], 1, которая помещает значение 1 по определенному адресу в памяти. Однако в этой инструкции не указывается длина устанавливаемого значения. Следует ли считать 1 байтом (0000 0001), словом (0000 0000 0000 0001) или двойным словом (0000 0000 0000 0000 0000 0000 0000 0001)? Начальные нули часто удаляются из значений для ясности и компактности, так что любая из них потенциально является допустимой интерпретацией перемещения 1.

Совет

Традиционно 32-разрядная версия x86 должна содержать 32-разрядные слова. Однако обратная совместимость с 16-разрядными архитектурами x86 означает, что слова имеют размер 16 бит, а двойное слово (dword) - 32 бита.

Когда размер доступа к памяти не подразумевается, он должен быть явно указан в инструкции. Например, байт инструкции [100] обращается к байту по адресу 100, word[ebx] обращается к слову, на которое указывает ebx, а dword[ax] обращается к двойному слову, на которое указывает ax. На рисунке 2.9 показана разница между следующими тремя инструкциями.

```
mov byte[100], 1
mov word[100], 1
mov dword[100], 1
```

Address	Value	Address	Value	Address	Value
98	0x00	98	0x00	98	0x00
99	0xff	99	0xff	99	0xff
100	0x01	100	0x01	100	0x01
101	0xfd	101	0x00	101	0x00
102	0x92	102	0x92	102	0x00
103	0xe8	103	0xe8	103	0x00
104	0x42	104	0x42	104	0x42
105	0x13	105	0x13	105	0x13
mov byte [100], 1		mov word [100], 1		mov dword [100], 1	

Рисунок 2.9: Сравнение инструкций mov разного размера

Режимы адресации
В синтаксисе Intel x86 адреса памяти обозначаются квадратными скобками. Например, [0x1234] указывает, что программа должна получить доступ к памяти, расположенной по адресу 0x1234.

Однако адресация памяти не ограничивается указанием адресов с непосредственными значениями, такими как 0x1234. Язык x86 поддерживает несколько различных режимов адресации. Различные режимы адресации используются для доступа к различным типам переменных.

Абсолютная адресация
Абсолютная адресация использует постоянное значение для указания адреса. Это постоянное значение может быть указано в любой базе данных, например [1] или [0x1234]. Это также может быть результатом арифметической операции [0x1337 + 0777] или обозначаться меткой [label].

Пример: Глобальные переменные
В C/C++ глобальные переменные предназначены для доступа из любого места в программе. Для достижения этой цели они хранятся в памяти по фиксированному адресу и не перемещаются при перемещении приложения по различным фреймам стека.

Это означает, что в assembly точный адрес переменной всегда будет известен. Следовательно, доступ к глобальным переменным будет осуществляться с использованием абсолютной адресации, такой как mov eax, [0x1000].

Косвенная адресация
Косвенная адресация использует регистры для указания адреса. Это включает как 16-разрядные GPRs, такие как [ax], так и 32-разрядные GPRs, такие как [eax]. Однако 8-разрядные GPRs (al, bh и т.д.) и SPRs не могут использоваться для адресации.

Пример: Указатели
Многие языки программирования используют концепцию указателей, некоторые более непосредственно, а другие скрыты за кулисами. Прямое использование и манипулирование указателями являются примером типа данных C/C++, который обычно использует косвенную

адресацию. Программа на C может содержать строку int x = 1; int* p = &x;, где указатель p установлен так, чтобы указывать на x. Если вы не знакомы с C, не волнуйтесь; просто знайте, что p содержит адрес того места, где x находится в памяти.

Однако значение p может быть изменено, чтобы указывать на другие объекты, поэтому адрес его назначения не фиксирован. Чтобы получить доступ к значению, указанному p в assembly, p сначала будет загружен в регистр, а затем этот регистр будет использоваться для поиска желаемого значения. Это показано в следующих инструкциях x86:

mov ebx, [p] ; Загрузите адрес, указанный p, в ebx
mov eax, [ebx] ; Переместите значение, указанное p, в eax

Адресация по базе + смещение

Некоторые переменные, такие как массивы, хранятся в памяти с использованием базового адреса и смещений. Доступ к отдельным значениям в массиве можно получить, используя базовый адрес и смещение.

Адресация по базе + смещение или адресация на основе данных использует комбинацию значения регистра и смещения для указания адреса. Этот тип режима часто используется для доступа к массивам. Таким образом, в языке, где у вас мог быть myList[8], вы получаете доступ к восьми элементам из базы myList. В assembly, например, [eax + 8] указывает восемь байт из базового адреса массива, который хранится в eax.

Индексированная адресация

Адресация по базе + смещение хорошо работает, если элементы в массиве всегда имеют длину в один байт. Для массивов с более крупными элементами смещение должно вычисляться вручную, что утомительно и подвержено ошибкам.

В этих случаях индексированная адресация может быть лучшим выбором. Индексированная адресация использует индексный регистр, масштабный коэффициент и смещение для указания адреса. Масштабный коэффициент всегда должен быть равен 1, 2, 4 или 8.

Пример: Массивы

Давайте определим массив целых чисел, int x[100];, который объявляет массив, содержащий 100 целых чисел. В памяти каждое значение в массиве хранится с определенным смещением от базового адреса. Это смещение определяется размером значений в массиве, например 32-разрядным или 4-байтовым int.

Предположим, что массив int был создан со смещением 0x1000. Следующая инструкция переместит n-й элемент массива в eax, если n хранится в ebx:

```
mov eax, [ ebx * 4 + 0x1000]
```

Адресация на основе индекса

Адресация на основе индекса сочетает в себе элементы индексированной адресации и адресации по базе + смещение. Для адреса используются базовый регистр, индексный регистр, масштабный коэффициент (1, 2, 4 или 8) и смещение.

Например, рассмотрим [ebx + edi * 4 + 0x1000]. Этот адрес имеет базу, хранящуюся в ebx, индекс, хранящийся в edi, и смещение 0x1000.

Пример: Структуры

Адресация на основе индекса идеально подходит для доступа к элементам вложенных типов

данных. Например, рассмотрим команду C struct { int i; short a[4]; } s;. Это создает структуру, содержащую несколько полей, включая массив.

Каждый элемент внутри этой структуры расположен с определенным смещением, что означает, что массив a имеет определенное смещение или базовый адрес. Однако элементы, содержащиеся внутри a, также расположены с разными смещениями от этого базового адреса.

Предположим, что базовый адрес структуры s хранится в ebx, а массив a хранится в 4 байтах от этого базового адреса. Следующая инструкция получит доступ к n-му элементу внутри a, если n хранится в ecx:

```
mov eax, [ebx + 2 * ecx + 4]
```

Не слишком расстраивайтесь, если более продвинутая адресация трудна для понимания. Ваша операционная система скрывала от вас память, поэтому естественно, что размышления о том, как хранятся массивы в памяти, - это новая территория. Сначала необходимо ознакомиться с этими режимами адресации в качестве теории, но их может быть трудно понять, прежде чем вы начнете использовать их позже в реальном ассемблерном коде. И не волнуйтесь, вы это сделаете.

Резюме

x86 - широко используемый язык ассемблера. Понимание того, как это работает, необходимо для того, чтобы стать успешным реверс-инженером программного обеспечения и взломщиком.

В этой главе рассмотрены некоторые ключевые концепции ассемблера x86. К ним относятся представление данных, синтаксис ассемблера и использование регистров и адресов памяти для доступа к данным и их хранения.

Глава 3
Сборка x86: Инструкции
Взлом и обратное проектирование включают чтение, запись и модификацию ассемблерного кода. В этой книге основное внимание уделяется языку ассемблера x86.

Необязательно разбираться во всех деталях ассемблера x86, чтобы быть реверс-инженером или даже писать программы на ассемблере. В этой главе рассматриваются основы x86 и основные инструкции, которые составляют более 90 процентов ассемблерного кода программного обеспечения.

Формат инструкций x86
Мнемоника используется в ассемблере x86 для создания удобочитаемого ассемблерного кода. Каждая из этих мнемонических инструкций собрана в машинный код, который управляет процессором. Таким образом, процессор не имеет понятия о мнемонике, только машинный код. Например, мнемоническая команда add присваивает машинному коду значение 0x04.

В x86 инструкции записываются в определенном формате. Примером простой инструкции x86 является:

```
add eax, 1
```

В этой инструкции add - это мнемоника, используемая для указания процессору, что делать. Эта инструкция также включает в себя пару операндов, которые указывают данные, которые будут использоваться в этой операции. В данном случае операндами являются регистр eax и значение 1. Инструкция x86 в обычных условиях может содержать до трех операндов, если они у нее вообще есть. Существуют специальные расширения языка, которые допускают расширение до четырех операций (префикс VEX), но мы не будем углубляться в этот аспект ассемблера.

Операндами инструкций x86 могут быть регистры, непосредственные значения или адреса памяти. Регистры обычно являются регистрами общего назначения (GPRs), а ячейки памяти задаются по адресу. Непосредственные значения - это числа или константы, такие как 12345.

Хотя инструкция x86 может включать в себя любую из них, она может содержать максимум одну ячейку памяти. Например, инструкции add eax, ebx и add eax, [0x12345678] действительны, поскольку они обращаются к двум регистрам и регистру и ячейке памяти соответственно. Однако команда add [0x12345678], [0x87654321] недопустима, поскольку она использует сразу два адреса памяти. Это связано с тем, что конвейер процессора представляет собой сложную конструкцию, которая может выполнять только одну выборку памяти за команду.

Инструкции x86
Язык ассемблера x86 включает сотни различных инструкций. Некоторые из наиболее часто используемых включают следующие:

- **Арифметика**
    - add
    - sub
    - mul
    - inc
    - dec

- **Манипулирование битами**
  - and
  - or
  - xor
  - not

- **Стек**
  - call
  - return
  - push
  - pop

- **Перемещение данных:**
  - mov

- **Поток выполнения**
  - jmp
  - Conditional jumps

- **Условные переходы**
  - test
  - cmp

- **Другие**
  - lea
  - nop

Хотя это может показаться многовато, рассмотрим распространенные операторы, используемые в языках программирования (+, −, *, /, %, &&, ||, &, |, ^, !, ~, <, >, >=, <=, ==, ., −> и т.д.) и основные ключевые слова (if, else, switch, while, do, case, break, continue, for и т.д.). Для достижения такого поведения в ассемблере требуется много возможностей.

По правде говоря, никто не знает всех инструкций x86 и не испытывает в этом необходимости (если только они действительно не хотят произвести впечатление на своих друзей). Полный список инструкций x86 можно найти по адресу http://ref.x86asm.net/coder32.html, и при необходимости можно просмотреть подробную информацию о любой инструкции.

Однако четкое понимание того, как работает наиболее распространенная сборка x86, необходимо для успеха в качестве реинжиниринга. Если вы понимаете это важное подмножество инструкций x86, вы сможете читать и понимать большинство программ x86.

mov

Как следует из названия, инструкция mov предназначена для перемещения данных из одного места в другое. Это включает в себя копирование данных между регистрами и ячейками памяти или немедленное размещение в определенном месте. Обратите внимание, что,

несмотря на свое название move, он копирует данные; он не перемещает их (это означает, что они не удаляются из источника; скорее, они копируются из источника в пункт назначения).

Синтаксис команды mov таков: mov destination, source. Например, команда mov eax, 5 помещает значение 5 в регистр eax. Аналогично, команда mov eax, [1] перемещает значение по адресу 0x1 в eax.

При работе с mov и подобными инструкциями важно помнить, что имена используемых переменных влияют на длину перемещаемого значения. Например, команда mov eax, [0x100] перемещает 32-разрядное значение в eax, в то время как команда mov dx, [0x100] перемещает 16-разрядное значение в dx.

ПРИМЕЧАНИЕ. Операнд x86 может использовать значение регистра для указания адреса памяти. Например, команда mov [eax], ebx перемещает значение, сохраненное в ebx, в ячейку памяти, адрес которой сохранен в eax. Итак, если eax имеет значение 0x7777, то адрес памяти 0x7777 - это место, где хранится значение ebx.

mov - чрезвычайно универсальный оператор и отличный пример преимущества мнемоники по сравнению с машинным кодом. mov можно использовать множеством различных способов, как показано на рисунке 3.1, и каждый из них преобразуется в другой машинный код в зависимости от того, какие два операнда используются. Однако все эти различные варианты представлены как mov на мнемоническом уровне. Задача ассемблера - преобразовать мнемонику в правильный машинный код.

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV r/m8, r8	MR	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8[1], r8[1]	MR	Valid	N.E.	Move r8 to r/m8.
89 /r	MOV r/m16, r16	MR	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32, r32	MR	Valid	Valid	Move r32 to r/m32.
REX.W + 89 /r	MOV r/m64, r64	MR	Valid	N.E.	Move r64 to r/m64.
8A /r	MOV r8, r/m8	RM	Valid	Valid	Move r/m8 to r8.
REX + 8A /r	MOV r8[1], r/m8[1]	RM	Valid	N.E.	Move r/m8 to r8.
8B /r	MOV r16, r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32, r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64, r/m64	RM	Valid	N.E.	Move r/m64 to r64.
8C /r	MOV r/m16, Sreg[2]	MR	Valid	Valid	Move segment register to r/m16.
8C /r	MOV r16/r32/m16, Sreg[2]	MR	Valid	Valid	Move zero extended 16-bit segment register to r16/r32/m16.
REX.W + 8C /r	MOV r64/m16, Sreg[2]	MR	Valid	Valid	Move zero extended 16-bit segment register to r64/m16.
8E /r	MOV Sreg, r/m16[2]	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg, r/m64[2]	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL, moffs8[3]	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL, moffs8[3]	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX, moffs16[3]	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX, moffs32[3]	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX, moffs64[3]	FD	Valid	N.E.	Move quadword at (offset) to RAX.
A2	MOV moffs8, AL	TD	Valid	Valid	Move AL to (seg:offset).
REX.W + A2	MOV moffs8[1], AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16[3], AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32[3], EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64[3], RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8, imm8	OI	Valid	Valid	Move imm8 to r8.
REX + B0+ rb ib	MOV r8[1], imm8	OI	Valid	N.E.	Move imm8 to r8.
B8+ rw iw	MOV r16, imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32, imm32	OI	Valid	Valid	Move imm32 to r32.
REX.W + B8+ rd io	MOV r64, imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /0 ib	MOV r/m8, imm8	MI	Valid	Valid	Move imm8 to r/m8.
REX + C6 /0 ib	MOV r/m8[1], imm8	MI	Valid	N.E.	Move imm8 to r/m8.
C7 /0 iw	MOV r/m16, imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /0 id	MOV r/m32, imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /0 id	MOV r/m64, imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

Рисунок 3.1: инструкции по перемещению

Практический пример

Как бы следующий псевдокод был записан в ассемблере? Предположим, что переменная i расположена по адресу 100, а j - по адресу 200.

int i = 42, j = i;

Эта единственная строка псевдокода может быть собрана в три инструкции x86.

```
mov [100], 42
mov eax, [100]
mov [200], eax
```
Обратите внимание, что регистр eax используется для хранения значения, копируемого с адреса памяти 100 на адрес памяти 200. Причина этого в том, что одна команда не может выполнить два разных обращения к памяти. Регистр, такой как eax, должен использоваться для временного хранения.

При взгляде на код может показаться, что было бы разумнее загрузить адрес 200 с немедленным значением 42, а не выполнять две операции для его загрузки из адреса памяти 100. Однако компилятор не будет и не должен этого делать.

Причиной этого является потенциал многопоточных приложений. Если в системе запущен другой поток, значение в местоположении 100, возможно, было обновлено между шагом, присваивающим ему значение 42, и шагом, присваивающим его значение местоположению 200. Копирование значения из местоположения 100 вместо немедленного использования помогает гарантировать, что переменная j в местоположении 200 получит самую последнюю версию значения, хранящегося в i.

inc, dec

Инструкции inc и dec x86 увеличивают или уменьшают указанное значение на 1 соответственно. Это эквивалент инструкций i++ или i-- в традиционном коде.

Эти инструкции принимают один операнд, который может быть регистром или адресом памяти. Например, команда inc eax увеличивает значение, хранящееся в eax, на 1, в то время как dec [0x12345678] уменьшает значение, хранящееся по адресу памяти 0x12345678, на 1.

add, sub

Инструкции add и sub вычитают значение из определенного значения соответственно. Эти инструкции принимают два операнда. Например, команда add будет указана как add destination, value.

Адресатом в инструкции add может быть регистр или ячейка памяти, в то время как значение может быть регистром, ячейкой памяти или немедленным. Операция принимает значение destination и сохраняет результат в destination. Это означает, что входящее значение получателя имеет отношение к математическому выражению, но перезаписывается для сохранения результата. Обратите внимание, что размер двух операндов должен быть одинаковым. Например, add eax, ebx является допустимой инструкцией (32-разрядная плюс 32-разрядная), в то время как sub eax, bx - нет (32-разрядная минус 16-разрядная).

При использовании add и sub важно учитывать размеры используемых значений. Например, команда sub ecx, [100] подразумевает 32-разрядное значение, используя ecx в качестве назначения. Однако для инструкции add dword [edx], 100 требуется спецификатор размера dword, поскольку 32-разрядное значение edx указывает, что адрес памяти имеет длину 32 бита, но не указывает размер данных, изменяемых в этом местоположении.

mul

Операция mul выполняет умножение целых чисел без знака. Однако это немного необычно, поскольку она принимает только один операнд, но неявно использует два дополнительных регистра. Синтаксис операции mul - это mul operand, где операндом может быть регистр или адрес памяти. Операция умножает значение, сохраненное в eax, на значение, указанное в операнде.

Результат операции mul сохраняется в edx:eax, причем edx содержит старшие 32 бита результата. Значения, сохраненные в edx и eax, всегда изменяются с помощью mul, даже если длина результата меньше 32 бит и edx не требуется. mul интересен тем, что вы можете получить 64-разрядный вывод (edx:eax) на основе 32-разрядной математики.

Примером операции mul является mul eax, которая возводит в квадрат 32-разрядное значение, хранящееся в eax. Когда операнд содержит адрес памяти, длина значения может варьироваться. Например, mul dword [0x555] умножает eax на 32-разрядное значение, сохраненное в 0x555, в то время как mul byte [0x123] использует при умножении 8-разрядное значение, сохраненное в 0x123.

div

Операция div выполняет деление без знака. Как и mul, она принимает один операнд и неявно изменяет регистры eax и edx. В этом случае частное сохраняется в eax, а остаток сохраняется в edx. Для тех, кому нужно краткое напоминание о математическом жаргоне (не смущайтесь, модные математические словечки тоже ранят мой мозг), 5, разделенное на 2, будет иметь частное 2, а остаток - 1.

Операция div использует как eax, так и edx для своих входных данных и форматирует их таким же образом, как и выходные данные mul, со старшими 32 битами, содержащимися в edx. Как и в случае с mul, выходные данные всегда изменяют eax и edx, даже если edx не требуется (т.е. остаток равен нулю).

Примером операции div является div eax. Это эквивалентно вычислению eax, edx = edx:eax / eax . В этом случае операнд представляет собой 32-разрядный регистр, но адреса памяти могут указывать и использовать делители разной длины.

Практический пример

Предположим, что вы хотели рассчитать остаток от 123/4. Это можно выполнить с помощью четырех инструкций по сборке.

```
mov eax, 123  ; Load the lower 32 bits of the dividend into eax
mov edx, 0    ; Clear the edx register, which holds the higher 32 bits of the dividend
mov ecx, 4    ; Load the divisor into ecx since div can't take an immediate operand
div ecx       ; Perform the division
```

В конце этого процесса частное сохраняется в eax, а остаток - в edx.

and, or, xor

Стандарт x86 включает поддержку нескольких различных логических операций. Все операции and, or и xor принимают два операнда. Таблицы истинности для этих трех операций показаны здесь. Параметры ввода показаны на верхнем и левом краях таблицы. Например, чтобы найти 1 И 1, мы находим пересечение 1 столбца и 1 строки, и результатом будет 1. Итак, 1 И 1 равно 1.

AND	1	0		OR	1	0		XOR	1	0
1	1	0		1	1	1		1	0	1

AND	1	0		OR	1	0		XOR	1	0
0	0	0		0	1	0		0	1	0

Все три операции используют один и тот же синтаксис: мнемонический адресат, источник. Например, синтаксис операции and - это и адресат, источник. Как и в операции add, адресатом должен быть регистр или адрес памяти, в то время как источником может быть регистр, адрес памяти или немедленный. И также, как и при операции добавления, назначение используется при вычислении, но также перезаписывается для сохранения результата.

Логические операции могут использоваться для множества различных целей. Например, операция или eax, 0xffffffff - это быстрый способ установить значение eax равным всем единицам. Операция и dword [0xdeadbeef], 0x1 маскируют все, кроме младшего бита 32-разрядного значения в местоположении 0xdeadbeef. Операция xor eax, eax - это распространенный метод очистки значения eax.
нет
Операция not - это логическая операция, которая вычисляет дополнение значения к единице. Для тех, кто не знаком с термином дополнение к единице, вы можете, по сути, представить это как взятие всех нулей и превращение их в 1 и наоборот. Это инвертирует число. Для этого требуется один операнд с синтаксисом not operand.

Оператор not может работать со значениями различной длины. Например, операция not ch вычисляет дополнение единицы к 8-разрядному регистру ch. Команда not dword [2020] вычисляет дополнение к 32-разрядному значению, расположенному по адресу 2020.

shr, shl
shr и shl - это две операции сдвига, доступные в x86, причем shr - это сдвиг вправо, а shl - сдвиг влево. Они принимают два операнда: местоположение значения, подлежащего сдвигу, и величину, на которую они должны быть сдвинуты. Примером операции сдвига является регистр shr, немедленный (?).

shr и shl являются логическими операторами сдвига. Это означает, что при сдвиге значения на указанное немедленное значение они увеличат значение до нуля влево или вправо. Таким образом, любые новые цифры, которые появятся в результате сдвига, будут автоматически равны 0.

Например, операция shr al, 3 сдвинет значение, сохраненное в al, вправо на три бита. Если al содержит значение 00010000, то результирующим значением будет 00000010.

Совет

Ноль - расширение значения, сдвинутого вправо, заполнит пустые биты нулями и называется логическим сдвигом. Знак -расширение значения, сдвинутого вправо, заполнит пустые биты тем же значением, что и старший бит, и называется арифметическим сдвигом.

sar, sal
sar и sal являются операторами арифметического сдвига. Их синтаксис идентичен синтаксису логических сдвигов, но они различаются по реализации. sar выполняет арифметический сдвиг вправо, а sal - арифметический сдвиг влево.

При выполнении сдвига влево sal работает так же, как shl, увеличивая значение до нуля.

Например, инструкции shl al, 3 и sal al, 3 со значением 00000100, сохраненным в al, обе выдадут значение 00100000. Все новые позиции, которые были открыты в number, были заполнены 0s.

Однако операция sar расширит значение по знаку, в то время как shl расширит его по нулю. Расширение по знаку означает, что он повторит любой бит, который был наиболее значимым. Например, если al содержит значение 10000000, то команда shr al, 3 выдаст значение 00010000, как показано здесь:

Начальное значение 10000000
01000000 1-разрядный сдвиг
00100000 2-разрядный сдвиг
00010000 3-разрядный сдвиг
Однако команда sar al, 3 приведет к 11110000. Поскольку наиболее значимым битом является 1, 1 реплицируется во всех новых позициях.

nop
Оператор nop расшифровывается как “нет операции”. Это однобайтовый оператор (0x90), который ничего не делает.

Хотя технически nop ничего не делает, он используется для множества законных целей, включая следующие:

- Синхронизация
- Выравнивание памяти
- Предотвращение опасностей
- Слот задержки перехода (архитектуры RISC)
- Заполнитель, который позже будет заменен будущим патчем
И в мире безопасности он используется для следующих целей:

- Взлом (nop sleds)
- Взлом (nop outs)
lea
Оператор lea обозначает эффективный адрес загрузки. Требуется два оператора, включая адрес назначения (адрес регистра или памяти) и источник, который должен быть адресом памяти. Инструкция lea вычисляет адрес указанного исходного операнда и помещает его в пункт назначения. Для тех, кто знаком с указателями, это похоже на оператор &.

Хотя lea предназначен для работы с адресами, он также обычно используется для простых математических операций. Например, операция lea eax, [ebx + ecx + 5] запрашивает, на какой адрес указывает ebx+ecx+5, а затем сохраняет это в eax. Это, по сути, вычисляет ebx + ecx + 5 и сохраняет результат в eax. Более стандартное использование оператора lea, lea eax, [100], поместило бы значение 100 в eax.

Хотя на первый взгляд это может показаться немного глупым или бессмысленным, lea - полезный оператор, поскольку он делает работу с массивами в assembly более эффективной. В массивах значения хранятся с определенным смещением от базового адреса. (Помните наши режимы адресации base + displacement?) С помощью lea можно эффективно вычислить адрес конкретного элемента в массиве. Например, предположим, что eax содержит базовый адрес массива символов. В этом случае команда lea ebx, [eax + 2] поместила бы адрес второго элемента массива в ebx. Эта отдельная инструкция более эффективна, чем последовательность инструкций mov ebx, eax и add ebx, 2, которые приводят к тому же

результату.

Практический пример
Как бы следующий псевдокод был записан на ассемблере? Предположим, что i находится по адресу 100, j - по адресу 200, а k - по адресу 300.

```
int i = 7;
char j = 5;
int k = i + j;
```

Этот псевдокод был бы собран в следующие инструкции x86:

```
mov dword [ 100 ], 7  ; set i
mov byte [ 200 ], 5   ; set j

mov eax, [ 100 ]      ; load i into eax
xor ebx, ebx          ; zero ebx
mov bl, [ 200 ]       ; load j into ebx

add eax, ebx          ; add ebx to eax, store in eax

mov [ 300 ], eax      ; save result to k
```

В этом примере обратите внимание на использование как ebx, так и bl. Значение, которое должно было храниться в этом регистре, помещается в bl. Однако при выполнении операции добавления используется весь регистр ebx. Это происходит из-за повышения класса, если вы добавляете значение в 1 байт к значению в 4 байта, значения в 1 байт повышаются до 4 байт, а дополнительные байты должны быть равны 0. Итак, в этом случае то, что было 0x05 в bl, в ebx повышено до 0x00000005. Операция XOR для очистки ebx была необходима для гарантии того, что предыдущее значение, сохраненное в регистре ebx, было полностью удалено и не повлияло на результат добавления.

Собираем все это воедино
До сих пор многие примеры представляли собой простые операции с использованием всего пары операторов x86. Теперь попробуйте написать ассемблерный код для следующего псевдокода, предполагая, что i находится по адресу 100, j - по адресу 200, а k - по адресу 300.

```
int i = 7;
char j = 5;
int k = i * i + j * j;
```

Этот псевдокод собирается в следующие инструкции x86:

```
mov dword [ 100 ], 7  ; set i
mov byte [ 200 ], 5   ; set j

mov ecx, [ 100 ]      ; load i into ecx
xor ebx, ebx          ; zero ebx
mov bl, [ 200 ]       ; load j into ebx

mov eax, ecx          ; copy ecx into eax (eax = ecx = i)
mul ecx               ; multiply ecx by eax, store result in eax
```

```
mov ecx, eax          ; save result back to ecx to free up eax

mov eax, ebx          ; copy ebx into eax (eax = ebx = j)
mul ebx               ; multiply ebx by eax, store result in eax

add eax, ecx          ; add ecx to eax, store result in eax
mov [ 300 ], eax      ; save final value to k
```

Распространенные ошибки в инструкциях x86

x86 - мощный язык ассемблера, и большинство инструкций следуют согласованному набору правил. Однако в нем есть свои несоответствия, которые могут сбить людей с толку.

Вот несколько примеров распространенных ошибок, которые люди совершают при попытке написать свой собственный x86, в результате чего код не собирается:

- mov [bl], 0xf: x86 поддерживает косвенную адресацию с использованием 16- и 32-разрядного GPRs. Поскольку длина bl составляет всего 8 бит, он не может быть использован для адресации.
- mov [0xabcd], 1337: В этой инструкции не указывается размер перемещаемого значения, поскольку 1337 может быть записано как 0x0539 или 0x00000539.
- mov word [0xabcd], eax: В этой инструкции указан неверный размер памяти, поскольку слово имеет размер 16 бит, но eax содержит 32 бита.
- mov byte [1], byte [2]: Две ячейки памяти не могут использоваться в одной и той же инструкции.
- mov sl, al: В то время как eax имеет регистр al, регистра sl не существует.
- mov 0x1234, eax: Значение 0x1234 является непосредственным, а не адресом памяти, и не может быть назначением команды.
- mov eax, dx: Эта инструкция имеет несоответствие размера между 16-разрядным исходным dx и 32-разрядным целевым eax.

Если сомневаетесь, посмотрите это в справочнике

Помните, никто (даже я, хотя мне трудно это признать) не знает все инструкции x86 наизусть. Независимо от того, пишете ли вы x86 или читаете его, если вы сталкиваетесь с чем-то, чего не понимаете, важно знать, как быстро это найти. У нас всегда открыта эта вкладка для быстрого поиска: http://ref.x86asm.net/coder32.html.

Резюме

x86 - сложный, мощный язык ассемблера. Однако необязательно понимать все до мелочей, чтобы быть эффективным программным взломщиком и реинжиниринговым специалистом.

В этой главе рассмотрены инструкции x86, составляющие подавляющее большинство ассемблерного кода. Изучение этих инструкций имеет важное значение и обеспечивает прочную основу для обратного проектирования.

Глава 4
Создание и запуск программ на ассемблере
Обратная разработка программного обеспечения заключается в том, чтобы взять скомпилированный исполняемый файл и превратить его в удобочитаемый код. Однако понимание того, как сделать обратное - создать и запустить программу на ассемблере, может оказаться неоценимым для понимания этого процесса.

В этой главе рассматриваются некоторые ключевые концепции, необходимые для понимания того, как создаются и запускаются программы на ассемблере. Сюда входит то, как эти программы взаимодействуют с внешним миром, как на самом деле их создавать и запускать, и как они управляют строками.

Анекдот

В колледже я обнаружил, что довольно забавно пойти в комиссионный магазин и купить кучу сломанной электроники, разобрать ее на части и собрать из кусочков что-то другое. Рисунок 4.1 - одна из первых вещей, которые я когда-либо создавал. Мне действительно нравится работать с непрактичными вещами, потому что думать, проектировать, создавать и учиться - это весело, но как только вы начинаете беспокоиться о реальных приложениях и удобстве использования, это лишает удовольствия. Итак, я изо всех сил стараюсь работать над непрактичными вещами.

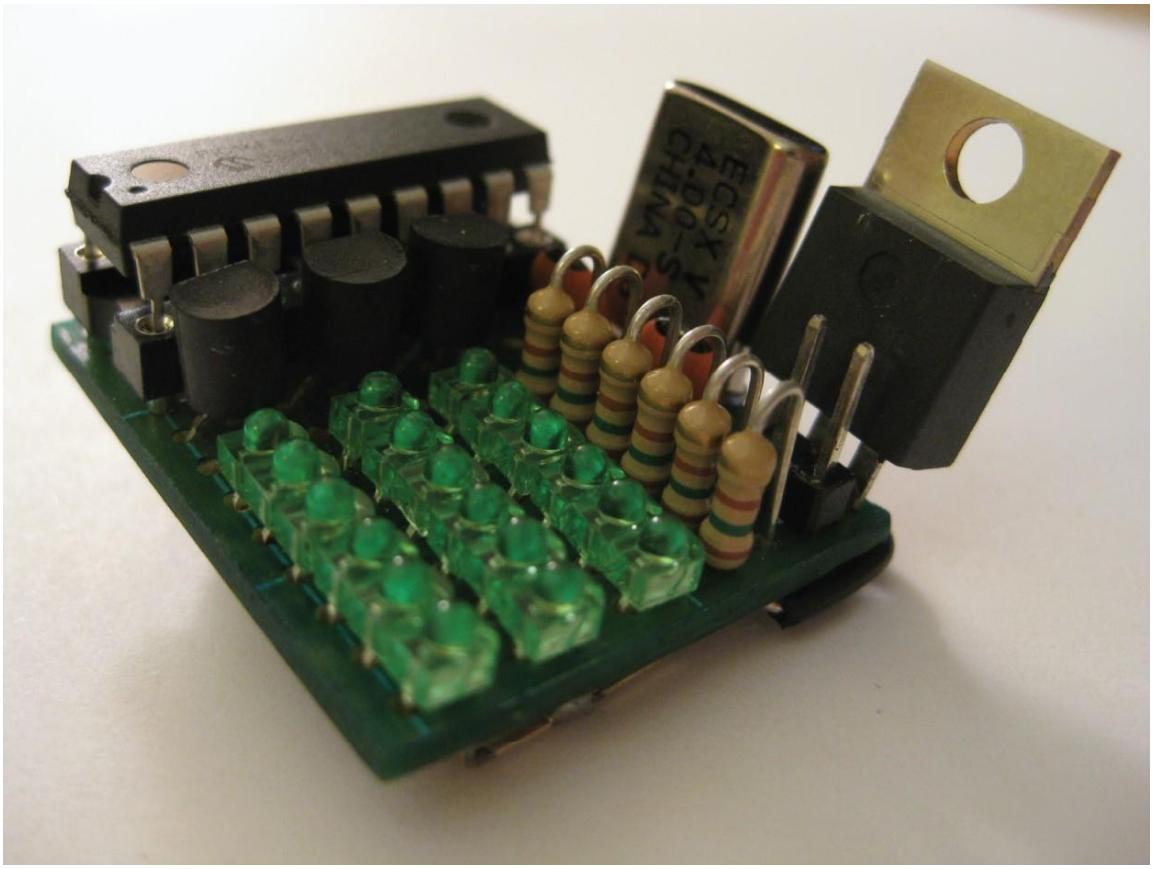


Рисунок 4.1: Бинарные наручные часы

Я пытался придумать самую непрактичную вещь, которую я мог бы здесь

смастерить. Это бинарные наручные часы, которые необходимо подключать к электрической розетке. Я так и не нашел для этого браслета, но я пытался придумать самую занудную вещь, которую я мог бы сделать, и это показалось мне довольно занудным.

Ассемблер и машинный код - это здорово, но в какой-то момент ваш код захочет взаимодействовать с внешним миром. Для этого нам нужен способ вывода информации.

Если вы когда-нибудь рассматривали процессор, они покрыты маленькими контактами. Контакты процессора позволяют процессору взаимодействовать с внешним миром. Используя ассемблер, можно управлять этими контактами, включая и выключая их, вызывая такие эффекты, как включение и выключение светодиодов. Современный процессор x86 имеет от 400 до 1000 контактов, что позволяет управлять многими вещами.

Контакты организованы в группы, называемые портами. С помощью портов вместо управления отдельными контактами, что было бы утомительно и отнимало много времени, можно управлять несколькими одновременно. Установка значения на выводах эквивалентна записи в порт, а получение значения из выводов равносильно считыванию из порта.

Для x86 определено несколько различных портов. В таблице 4.1 приведены некоторые примеры небольшого подмножества.

Таблица 4.1: Порты x86

PORT RANGE	SUMMARY
0x0000-0x001F	Первый устаревший контроллер DMA, часто используемый для передачи данных на дискеты
0x0020-0x0021	Первый программируемый контроллер прерываний
0x0022-0x0023	Доступ к регистрам, зависящим от модели процессоров Cyrix.
0x0040-0x0047	Программируемый интервальный таймер (PIT)
0x0060-0x0064	Контроллер PS/2 “8042” или его предшественники
0x0070-0x0071	Регистры CMOS и RTC
0x0080-0x008F	Регистры DMA (page registers)
0x0092	Расположение регистра fast A20 gate
0x00A0-0x00A1	Второй PIC
0x00C0-0x00DF	Второй контроллер DMA, часто используемый для sound blasters
0x00E9	Домен для взлома порта E9
0x0170-0x0177	Дополнительный контроллер жесткого диска ATA
0x01F0-0x01F7	Основной контроллер жесткого диска ATA
0x0278-0x027A	Параллельный порт
0x02F8-0x02FF	Второй последовательный по
0x03B0-0x03DF	Линейка, используемая для IBM VGA, ее прямых предшественников
0x03F0-0x03F7	Контроллер гибких дисков

PORT RANGE	SUMMARY
0x03F8-0x03FF	Первый последовательный порт

Управляющие контакты
В x86 контактами можно управлять с помощью инструкций ввода и вывода, которые принимают регистр и порт в качестве параметров.

Синтаксис команды in - in register, port. Например, в al 0x64 возвращает статус клавиатуры.

Команда out изменяет порядок параметров на противоположный, с синтаксисом out port, register. Например, из 0x3c0, eax устанавливает значение пикселя.

На самом деле, вы часто не подключены напрямую к портам назначения или источника, и все немного сложнее. Контакты подключены к общей шине, и работа по отправке операций чтения/записи в нужное место назначения переносится на отдельную плату или мост. Команды ввода и вывода обращаются к предопределенным адресам на шине, которые транслирует мост. Однако идея та же.

Скука
Давайте вернемся к этому понятию вывода: с помощью инструкций ввода и вывода можно устанавливать и отключать отдельные пиксели. Однако один экран дисплея может содержать тысячи или миллионы пикселей. Установка их по отдельности была бы утомительной и неэффективной.

Вместо этого эти детали абстрагируются. При отображении изображений графическая карта обрабатывает детали настройки каждого отдельного пикселя. Однако изучение того, как именно взаимодействовать с графической картой, может быть утомительным.

Именно здесь вступает в действие операционная система. Она может справиться со сложностями взаимодействия с видеокартой, которая устанавливает значения пикселей и отображает изображение. Для взаимодействия с операционной системой требуется системный вызов. Так что, если вы хотите играть на x86 в epic hard mode, вы можете пойти по пути прямого взаимодействия с видеокартой, но для целей этой книги нам нравится играть в режиме hard-with-help, и мы будем использовать ОС, чтобы выполнить эту тяжелую работу за нас.

Системные вызовы
Системные вызовы доступны в x86 для обеспечения ограниченной функциональности ввода-вывода, вызывающей поведение через операционную систему (OS). Наборы доступных системных вызовов различаются в зависимости от операционной системы.

Поскольку системные вызовы являются понятием операционной системы, они зависят от операционной системы; мы рассмотрим некоторые из наиболее полезных системных вызовов в Linux. Системные вызовы вызываются путем загрузки номера функции в регистр eax. В Linux системный вызов затем выполняется путем вызова прерывания с помощью инструкции int 0x80.

sys_write
На языке программирования более высокого уровня функция sys_write имела бы синтаксис ssize_t sys_write(unsigned int fd, const char * buf, size_t count). Эта функция вернет размер, указывающий объем записанных данных.

Функция sys_write принимает три аргумента. Первый, fd, является файловым дескриптором, который указывает, куда должны быть записаны данные. Значение 1 указывает на запись данных в консоль Linux. Аргумент buf содержит данные, которые будут записаны в качестве выходных данных, а count сообщает функции количество символов для печати.

В сборке x86 функции не могут быть вызваны с использованием этого описания функции. Вместо этого аргументы были бы загружены в регистры, как показано в таблице 4.2. После загрузки этих регистров системный вызов может быть выполнен с помощью инструкции int 0x80.

Таблица 4.2: sys_write

REGISTER	VALUE	DESCRIPTION
eax	4	идентификатор sys_write
ebx	1 (console out)	Файловый дескриптор
ecx	const char* buf	Строка для записи
edx	size_t count	Длина строки

Регистры, используемые в функции sys_write, должны быть загружены с помощью серии инструкций по сборке. В следующем примере показано, как можно использовать sys_write:

```
mov   edx,len        ; message length
mov   ecx,buff       ; message to write
mov   ebx,1          ; file descriptor (stdout)
mov   eax,4          ; system function(sys_write)
int   0x80           ; call kernel
```

sys_exit

Системный вызов sys_exit эквивалентен вашему main, возвращающему статус; в языках программирования более высокого уровня. Это приведет к завершению работы программы. Он принимает единственный аргумент, код состояния, который хранится в ebx, как показано в таблице 4.3.

Таблица 4.3: sys_exit

REGISTER	VALUE	DESCRIPTION
eax	1	идентификатор sys_write
ebx	int	Статус-код

Вызов sys_exit начинается с загрузки значений в регистры eax и ebx, как показано в следующем примере:

```
mov eax, 1           ; system function (sys_exit)
mov ebx, 0           ; return 0;
```

```
int 0x80            ; call kernel
```

Печать строки

Печать строки требует включения и выключения определенных выводов процессора. Выполняя системный вызов, программа сборки может переложить работу по определению того, какие выводы включать и выключать, на операционную систему. Операционная система сообщает об этом видеокарте, которая выбирает свои биты для включения и выключения. Это отправляет информацию на микроконтроллер монитора, заставляя его включать и выключать вращение, которое отображается на экране. Попутно в процесс могут быть вовлечены десятки других контроллеров.

Программа на ассемблере, которая печатает строку, а затем завершает работу, будет использовать системные вызовы sys_write и sys_exit. Мы рассмотрим общий синтаксис файла в следующем разделе. А пока, чтобы вы были взволнованы и сидели на краешке стула, читая следующий раздел, вот краткий обзор. В следующем примере на консоль выводится сообщение “Привет, мир!”:

```
global _start

section .text
_start:
    mov eax, 4 ; write
    mov ebx, 1 ; stdout
    mov ecx, msg
    mov edx, msg.len
    int 0x80

    mov eax, 1 ; exit
    mov ebx, 0
    int 0x80

section .data
msg:    db  "Hello, world!", 10
.len:   equ $ - msg
```

Построение и компоновка

Обратное проектирование и взлом - это понимание существующего чужого ассемблерного кода. Однако вы обнаружите, что если вы выполняете какие-либо исправления / взломы, написать свою собственную сборку и реверс-инжиниринг намного проще, если вы понимаете, как работает процесс в другом направлении, при написании, сборке и ассемблерном монтаже вашего собственного ассемблерного кода. Сборка и компоновка - важнейший шаг в процессе перехода от ассемблерного кода к функциональному приложению.

Сборка и компоновка в Linux

Процесс создания и компоновки ассемблерного кода зависит от операционной системы, поэтому этот раздел посвящен Linux. В среде Linux мы традиционно называем файлы ассемблера с расширением .asm, например program.asm. Затем программу можно собрать, связать и выполнить, используя следующие три команды:

```
nasm -f elf program.asm
```

```
ld -melf_i386 program.o -o program.out
./program.out
```

Первая из этих команд использует сетевой ассемблер nasm для сборки кода в объектный файл. Флаг -f указывает формат, которым является ELF, исполняемый файл Linux. Результатом будет объектный файл с именем program.o.

Следующим шагом в процессе является компоновка, для которой будет использоваться ld, компоновщик GNU. Параметр -melf_i386 определяет архитектуру, которая должна использоваться для компоновки, и указывает, что это должен быть двоичный файл ELF с использованием i386 (x86). Флаг -o указывает имя выходного файла, которым будет program.out.

После завершения компоновки файл program.out представляет собой полнофункциональный исполняемый файл Linux. Этот исполняемый файл можно запустить с помощью команды ./program.out.

Написание программы на ассемблере
Предыдущий пример демонстрирует, как создать и связать программу на ассемблере в Linux. Однако, прежде чем это произойдет, вам необходимо написать программу на ассемблере! В этом разделе рассматриваются основные концепции, необходимые для этого.

Разделы и статистика
В следующем примере показана общая структура файла сборки**:**

```
section .text ; section for code
global _start ; exports start method
_start:       ; execution starts here

; code here

section .data ; section for data

; variables here
```

Исходный файл ассемблера разбит на несколько основных разделов. Раздел .text содержит фактический код ассемблера. Этот раздел начнется с команды global _start, которая экспортирует метку _start, сообщающую внешним программам, с чего начать выполнение вашего кода. После этого появляется метка _start, которая указывает адрес памяти первой инструкции в программе. Оставшийся код будет следовать этой первой инструкции.

После раздела .text находится раздел .data, который содержит данные, необходимые программе сборки для запуска. Распространенным примером данных в программе сборки являются переменные, определенные в этой программе.

Этикетки
Метка _start жизненно важна для функционирования программы на ассемблере, но также возможно определить другие метки. Включая текстовую метку: в коде будет создана метка с именем label, которая является постоянным значением, синонимичным этому местоположению в памяти.

После определения метки ее можно использовать вместо традиционного адреса памяти. Метки можно использовать везде, где может использоваться постоянное или непосредственное значение. В следующих примерах показаны эквивалентные инструкции с метками и без них:

```
mov eax, [ label ]          ; access the dword stored at the label
mov eax, [ 0x1000 ]         ; if the label was on data at address
0x1000,
; this is equivalent to the previous instruction

jmp label2                  ; jump to the code at the label2
jmp 1337h                   ; if the label2 was on code at address
1337h,
; this is equivalent to the previous instruction
```

Метки используются только для того, чтобы облегчить чтение и запись ассемблерного кода. После того, как код будет собран, слово label будет заменено ассемблером и компоновщиком эквивалентным адресом памяти.

Константы

Константы упрощают работу с данными. Так, например, вместо напоминания о том, что максимальный размер буфера равен 1000, гораздо проще определить константу с именем MAX_SIZE со значением 1000.

Константы могут быть определены в сборке x86 с помощью директивы EQUU. Например, константа MAX_SIZE может быть определена со значением 1000 с помощью команды MAX_SIZE, равной 1000.

Глобальные данные

nasm позволяет объявлять пространство для глобальных данных различного размера. Некоторые команды включают следующее:

- db: Зарезервировать пространство для одного байта.
- dw: Зарезервировать пространство для одного слова (два байта).
- dd: Зарезервировать пространство для одного dword (четыре байта).
- dq: Зарезервируйте место для одного слова (восемь байт).

В следующем примере показаны некоторые инструкции, которые используют эти команды для выделения различных типов данных:

```
db 0x01       ; store the value "1" in a single byte
db 1, 2, 3    ; store the array 1, 2, 3 as 1 byte elements
db 'a'        ; store the ascii value of 'a' in one byte
db "hello", 0 ; store the nul terminated string "hello"
dw 0x1234     ; store 0x1234 as a two byte value
dd 0xdeadbeef ; store 0xdeadbeef as a four byte value
dq 1          ; store 1 as an 8 byte value
```

Хранение данных в памяти не дает никакой пользы, если к этим данным нельзя получить доступ позже. Как правило, при определении глобальных данных им также присваивается метка, позволяющая ссылаться на них в коде.

В следующем примере показана простая программа сборки, которая определяет пробел для dword со значением 0, помечает его как i и помещает в него значение 1.

```
section .text
mov dword [ i ], 1
section .data
i: dd 0
```

В этом примере важно отметить, что i - это не переменная, это символ, созданный с помощью метки. Использование i в коде аналогично использованию адреса памяти выделенных данных.

strings

Строка определяется в ассемблере как последовательность байтов, причем каждый символ занимает один байт. Например, слово “hello” может быть сохранено в памяти с помощью команды label: db “hello” и указано в коде как label.

При работе со строками в ассемблере важно отметить, что по умолчанию они не заканчиваются нулем. Чтобы явно завершить строку нулем, добавьте нулевой байт (0x0) в конец, как в label: db “hello”, 0. Нулевое завершение используется почти во всех языках программирования для хранения строк; однако компилятор сделал это за вас. Теперь, когда вы владеете возможностями сборки, вам нужно будет сделать это вручную, чтобы любые функции, использующие строки, выполнялись корректно. Без нулевого ограничителя в конце строки функции на основе строк будут продолжать захватывать память после предполагаемой строки и пытаться использовать или напечатать ее как символ. Это приводит к непредсказуемому результату, от безобидного, как печать некоторых непечатаемых символов, до серьезного, как сбой программы при попытке извлечь данные из памяти, на использование которой у нее нет разрешения.

times

Префикс times может использоваться для указания того, что конкретная инструкция или префикс должны повторяться определенное количество раз. Это может быть полезно для создания буферов фиксированной длины и других приложений, как показано в следующих примерах:

```
times 100 db 0   ; create 100 bytes, initialized to 0

times 64 db 0x55 ; create 64 bytes, each initialized to 55h

; pad "hello world" to a length of 64
buffer: db      'hello, world' times 64-$+buffer db ' '
```

$

$ - это сокращение для адреса текущей строки. Его можно использовать аналогично метке, как показано в следующих примерах:

```
jmp $                ; Infinite loop
string: db "hello"
length EQU $-string  ; Calculate length of string on previous line
```

В этом примере length является примером имени переменной. Значение length устанавливается равным текущему адресу ($) за вычетом адреса, указанного строкой метки. Поскольку length находится сразу после строки, это фактически берет адрес после “hello” и вычитает адрес в начале, давая вам длину строки “hello”.

ОБРАТИТЕ

внимание, что префиксы, такие как times и $, специфичны для nasm и не будут отображаться во встроенном коде. Разные ассемблеры могут иметь разные сочетания клавиш.

objdump

Такие инструменты, как nasm и ld, используются для создания исполняемого файла из ассемблерного кода. Object Dump (objdump) - это инструмент Linux, который обращает этот процесс вспять, беря исполняемый файл и сбрасывая его ассемблерный код. Хотя по мере продвижения по книге мы будем внедрять все более мощные инструменты, мы начинаем с objdump, потому что он есть в каждой системе на базе Linux и обеспечивает хорошую основу.

objdump может выгружать ассемблерный код любого приложения, работающего в Linux. В результате у него есть несколько возможных вариантов конфигурации. Но два наиболее важных при начале работы с обратным проектированием включают следующее:

-d: Дает objdump указание дизассемблировать содержимое всех разделов
-Mintel: Указывает, что выходные данные сборки должны отображаться в синтаксисе Intel (к сожалению, по умолчанию используется AT&T)
Принимая это во внимание, синтаксис для дизассемблирования программы с именем appname - objdump -d -Mintel appname. Некоторые примеры выходных данных для этого показаны здесь:

```
804a030 <test_key>:
804a030 55          push ebp
804a031 89 5e       mov  ebp, esp
804a033 53          push ebx
804a034 83 ec 14    sub  esp, 0x14
```

Выходные данные objdump организованы в три столбца. Первый столбец содержит адреса памяти, которые являются виртуальными адресами, по которым будут расположены инструкции при запуске кода. Второй содержит машинный код x86 в этом месте, а третий содержит эквиваленты ассемблерного кода x86 для этого машинного кода.

Основными исключениями из этого макета являются метки, как показано в верхней части предыдущей таблицы. Эта метка содержит имя. Обратите внимание, что адрес, связанный с меткой, совпадает с адресом первой инструкции в коде; метка не занимает никакого места в памяти.

Лабораторная работа: Hello World

Настало время для второго лабораторного упражнения. Пожалуйста, перейдите на GitHub книги по адресу https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-

ENGINEERING-CRACKING-AND-COUNTER-MEASURES и найдите лабораторную работу “hello world”.

Навыки

Эта лабораторная работа предоставляет возможность научиться писать и создавать приложения с кодом x86. Некоторые из ключевых навыков, которые будут проверяться, включают следующие:

- Регистры
- Память
- Инструкции
- Системные вызовы
- Создание и компоновка сборки x86

В этой лабораторной работе также предлагается практический опыт работы с несколькими различными инструментами, включая следующие:

- Makefiles
- nasm
- ld
- objdump

Выводы

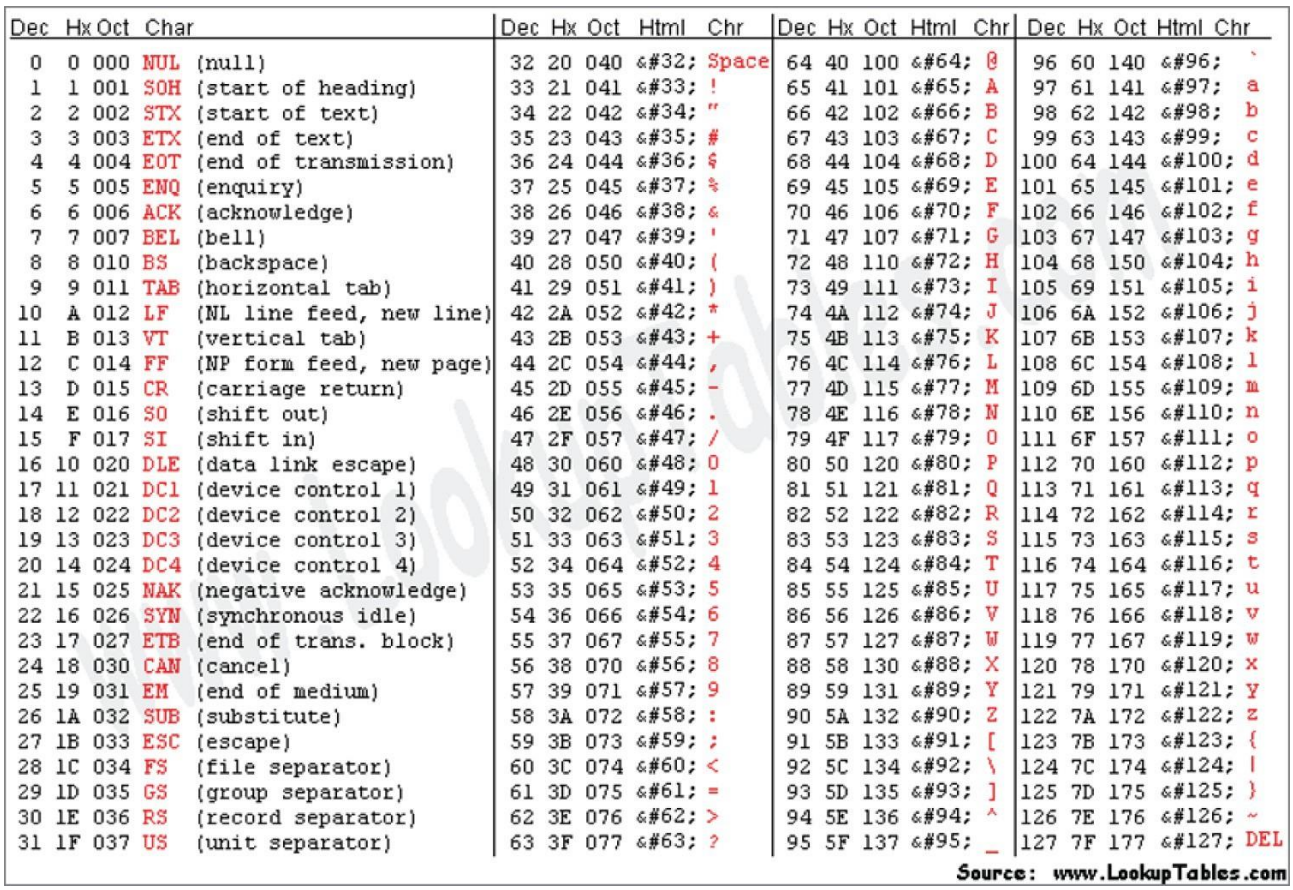
Приложения в основном состоят из некоторой формы инструкций по сборке. Обычно на ПК это x86, но иногда это может быть язык JIT или промежуточный язык (IL).

Понимание того, как создавать программы на этом низкоуровневом языке, также дает представление о том, как их разбирать на части. При взломе программ способность писать x86 может оказаться неоценимой для разработки исправлений, позволяющих обойти защиту программного обеспечения.

ASCII

Американский стандартный код для обмена информацией (ASCII) и формат преобразования Unicode (UTF) являются стандартами, которые определяют, как компьютеры представляют текст. На самом деле ASCII является подмножеством UTF-8.

ASCII был разработан в 1960 году и использует семь битов для представления каждого символа. На рисунке 4.2 показана полная таблица ASCII.



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Рисунок 4.2: Таблица ASCII

Стандарт ASCII может поддерживать следующие типы символов:

- Цифры (0-9)
- Строчные буквы (a–z)

Заглавные буквы (A–Z)
Общепринятая пунктуация
Понимание ASCII полезно для обратного проектирования, поскольку именно так строки, вероятно, будут представлены в ассемблерном коде и памяти. Например, строка “Hello, world” хранится в памяти как 0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x2C, 0x20,0x77, 0x6F, 0x72, 0x6C, 0x64.

Идентификация строк ASCII
Одной из проблем при обратном проектировании является определение того, является ли последовательность байтов строкой ASCII, числом или чем-то еще. Например, последовательность байтов 0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x2C, 0x20, 0x77, 0x6F, 0x72, 0x6C и 0x64 может быть строкой “Привет, мир”; значения 1,819,043,144; или любая из многих других возможностей.

Совет

Сложность идентификации строк ASCII заключается в том, что такие инструменты, как strings, часто возвращают много мусора. Они просто ищут последовательность байтов, которую можно интерпретировать как

строку символов, пригодных для печати.

Единственный способ узнать наверняка, что последовательность байтов является строкой, - это посмотреть, как она используется программой. Если байты передаются функции, которая интерпретирует их как строку, то, скорее всего, это строка.

Во многих языках полезным признаком строки является последовательность пригодных для печати байтов, заканчивающаяся нулевым символом. На самом деле, поскольку процессору нечего сообщить, где начинается или заканчивается строка, если строка забывает свой нулевой терминатор, процесс будет просто продолжать считывать символы до тех пор, пока не возникнет проблема!

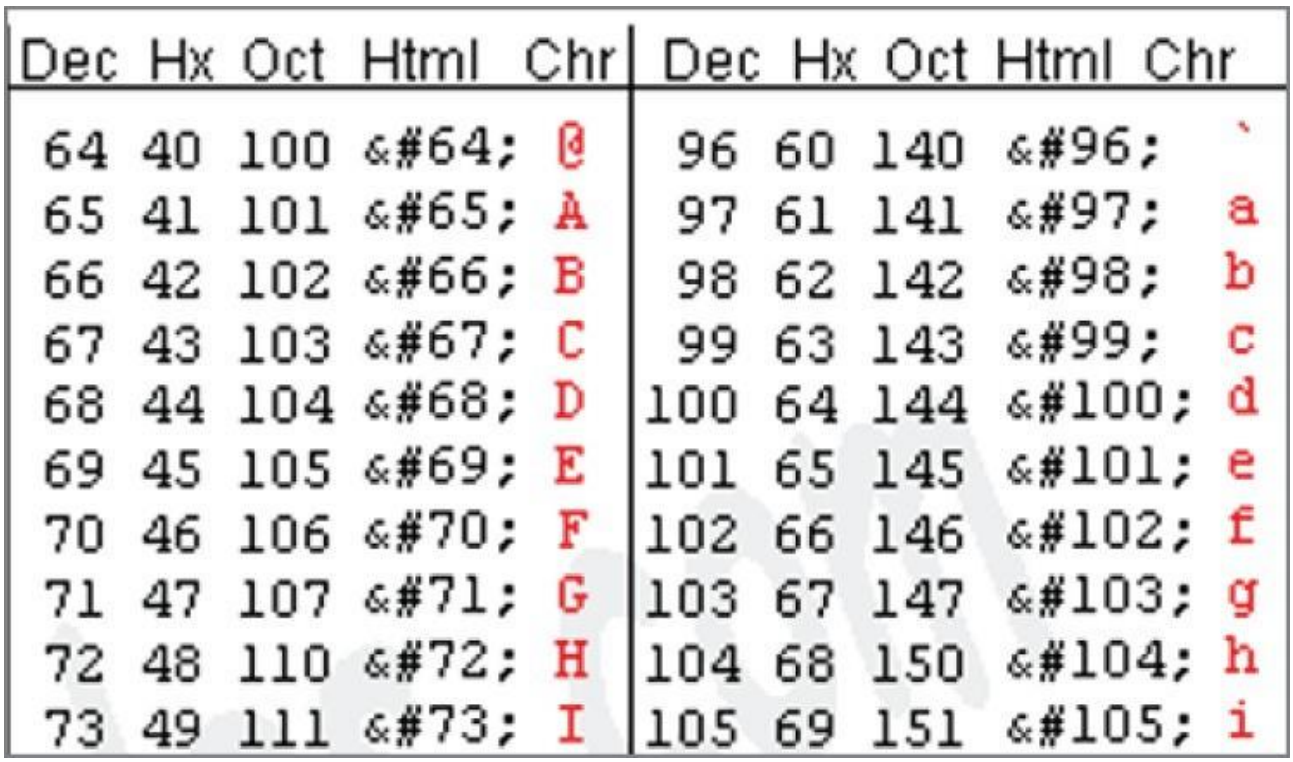
Например, следующая программа напечатает “hello”, за которым последуют 16 бит, а затем продолжит чтение и печать памяти до тех пор, пока не достигнет нулевого байта.

```
#include <stdio.h>
int main()
{
        char mybuffer[16];
        for (int i = 0; i < 16; i++)
        {
                mybuffer[i] = 'B';
        }
        Printf("hello %s\n", mybuffer);
}
```

На рисунке 4.3 показаны выходные данные этого кода.

Рисунок 4.3: Выходные данные программы

Подсказка по манипулированию ASCII
Стандарт ASCII разработан таким образом, что заглавные и строчные буквы всегда разделяются 0x20, как показано на рисунке 4.4. В языках программирования более высокого уровня функция ToUpper просто добавляет 0x20 к значению строчной буквы, а функция toLower просто вычитает 0x20.



Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
64	40	100	@	@	96	60	140	`	`
65	41	101	A	A	97	61	141	a	a
66	42	102	B	B	98	62	142	b	b
67	43	103	C	C	99	63	143	c	c
68	44	104	D	D	100	64	144	d	d
69	45	105	E	E	101	65	145	e	e
70	46	106	F	F	102	66	146	f	f
71	47	107	G	G	103	67	147	g	g
72	48	110	H	H	104	68	150	h	h
73	49	111	I	I	105	69	151	i	i

Рисунок 4.4: Значения ASCII в верхнем и нижнем регистре

Резюме

В этой главе исследуется, как программы на ассемблере объединяются в прямом направлении. Это включает в себя то, как они взаимодействуют с внешним миром, процесс их создания и связывания, а также как они управляют строками.

Понимание этих процессов в прямом направлении может оказаться бесценным для понимания того, как это работает в обратном направлении. Если вы знаете, как ассемблерный код переходит из кода в исполняемый файл, вы лучше понимаете, как его разобрать и снова собрать воедино.

Глава 5
Понимание кодов условий
Инструкции по сборке обычно включают регистры назначения, в которых будет сохранен результат операции. Однако некоторые инструкции могут иметь эффекты, выходящие за рамки тех, которые записаны в этом регистре назначения.

x86 использует коды условий для отслеживания этих эффектов. В этой главе рассматриваются эти коды условий и описываются основные из них, которые вам необходимо понять для эффективного обратного проектирования приложений x86.

Коды условий
Большинству архитектур, включая x86, необходимы средства отслеживания основных свойств предыдущих операций. Например, при вычислении оператора if программе необходимо оценить условие, а затем действовать в соответствии с его результатом. Возможность отслеживать информацию о состоянии в разных инструкциях необходима для возможности выполнения этой и подобных операций.

Для хранения этой информации о состоянии компьютер имеет регистр специального назначения (SPR), называемый flags. В 32-разрядной системе это называется регистром eflags, в то время как 16-разрядные и 64-разрядные версии называются flags и rflags, соответственно.

eflags
Регистр eflags состоит из набора флагов, каждый из которых представлен одним битом. Каждому биту может быть присвоено значение true (1) или false (0).

Регистр eflags разбит на три типа флагов.

- Флаги состояния: Флаги состояния представляют статус некоторой операции, например, равна ли предыдущая операция нулю.
- Флаги управления: Флаги управления влияют на работу процессора, например, на включение и отключение прерываний.
- Системные флаги: Системные флаги отражают состояние процессора, например, является ли система виртуализированной.

С 32 битами в регистре eflags в этих битах может храниться значительный объем информации о состоянии. Для обратного проектирования важны только некоторые флаги состояния.

Из флагов состояния четыре важны для обратного проектирования; к ним относятся флаги переноса, нуля, знака и переполнения.

Флаг переноса
Флаг переноса (CF) - это бит 0 регистра eflags. Он указывает, привела ли последняя арифметическая операция к переносу.

Перенос указывает на то, что сложение перенесло 1 из старшего бита или вычитание перенесло 1 в старший бит. Например, рассмотрим следующее вычисление, которое приведет к установке бита переноса:

```
                        unsigned     signed
  0011 0000               48           48
+ 1110 0000             + 224        + -32
1 0001 0000               16           16
```

Напомним, что в двоичном коде нет ничего, что указывало бы на значение; все дело в том, как оно используется или интерпретируется. Итак, двоичное представление в этом примере может быть интерпретировано как значение без знака или со знаком signed. Подписанный, как вы можете видеть, означает вариант отрицательный или положительный, в то время как беззнаковый означает всегда положительный.

В этом примере, если вы проследите сложение, то сможете увидеть самый левый столбец, в котором выполнено 1. Глядя на значения со знаком и без знака, вы можете видеть, что для unsigned флаг переноса представляет собой переполнение, что означает, что результат был слишком большим для сохранения в размере (в данном случае мы рассматриваем 1 байт). И именно так это традиционно используется для идентификации переполнений /недопотоков в математике без знака. Если условие переноса выполнено, то CF устанавливается равным 1.

Нулевой флаг
Нулевой флаг (ZF) является битом 6 регистра eflags и указывает, закончилась ли последняя арифметическая операция нулем. Например, следующее вычисление установило бы нулевой флаг:

```
  0100 0000          64
- 0100 0000        - 64
  0000 0000          00
```

О нулевом флаге рассуждать проще, поскольку ответом будут просто все нули. Нет разницы в интерпретации между знаковыми и неподписанными. Если результат равен 0, то нулевому флагу присваивается значение 1.

Знаковый флаг
Седьмой бит регистра eflags, знаковый флаг (SF), указывает, был ли установлен знаковый бит в результате предыдущей арифметической операции. В числах со знаком знаковый бит является старшим разрядом используемого регистра.

Например, в инструкции add ax, bx знаковым битом является бит 15 из ax. В инструкции sub bl, dl знаковым битом является бит 7 из bl. Если бит установлен, он считается отрицательным, а если нет, то положительным. В случае, если установлен знаковый бит, SF будет установлен равным 1. Если установлен верхний бит результата, мы знаем, что это отрицательное число, но это не так просто, как использовать наш обычный перевод в десятичное значение для оставшихся битов, чтобы получить значение. Если число отрицательное, оно сохраняется в формате “дополнение к двум”, который требует дополнительной обработки, чтобы вернуться к своему истинному значению.

Флаг переполнения
Флаг переполнения (OF) является одиннадцатым разрядом регистра eflags и указывает, привела ли предыдущая арифметическая операция к переполнению. Переполнение происходит, когда перенос в старший бит не соответствует выполнению. Точно так же, как флаг переноса полезен для математики без знака, флаг переполнения используется для математики со знаком, чтобы определить, когда что-то пошло не так.

Часто это указывает на один из двух случаев:

Положительный + Положительная = Отрицательная
Отрицательный + Отрицательная = Положительная
Для первого случая рассмотрим следующий расчет:

```
  0101 0000           80
+ 0101 0000         + 80
0 1010 0000          -96
```

При этом вычислении два положительных значения складываются вместе. Однако в результате устанавливается знаковый бит, указывающий на отрицательное число. Прослеживая этот столбец за столбцом в его двоичной форме, вы увидите, что 1 переносится в крайний левый столбец, но ничего не выполняется (т.е. выполняется 0). Это означает, что ввод не совпал с выводом, и, как вы можете видеть в десятичном формате, в результате мы получили неверное отрицательное значение.

Для примера второго случая рассмотрим следующее:

```
  1000 0000          -128
+ 1011 0000        + -80
1 0011 0000            48
```

В этом случае суммируются два отрицательных числа. Однако переполнение приводит к тому, что результат становится положительным значением. Опять же, отслеживая это на двоичном уровне, мы можем видеть, что в крайнем левом столбце нет переноса, но есть перенос, поэтому перенос не соответствует переносу. Рассматривая это в десятичной форме, мы видим, что получаем неправильное положительное значение.

Другие флаги статуса

Хотя эти четыре являются наиболее важными флагами статуса, они не единственные. Некоторые из других, менее важных флагов, о которых все же стоит знать, включают следующее:

- Регулирующий (AF): Указывает, что последняя арифметическая операция приводит к выполнению младших 4 бит (используется для арифметики BCD)
- Ловушка (TF): Включает одноступенчатый режим процессора, который используется для отладки
- Включение прерывания (IF): Позволяет процессору обрабатывать системные прерывания
- Направление (DF): Задает направление обработки строки справа налево
- Четность (PF): Указывает, что последняя арифметическая/логическая операция приводит к четной четности (четное число единиц в младшем байте).

Операции, влияющие на флаги состояния

На флаги состояния могут влиять различные операции. Четыре примера включают add, sub, cmp и test.

add

Команда add потенциально может изменять флаги переноса, обнуления, знака и переполнения. Например, команда add al, bl может запускать различные комбинации флагов в зависимости от значений, сохраненных в al и bl. На рисунке 5.1 показаны результаты пяти различных операций добавления.

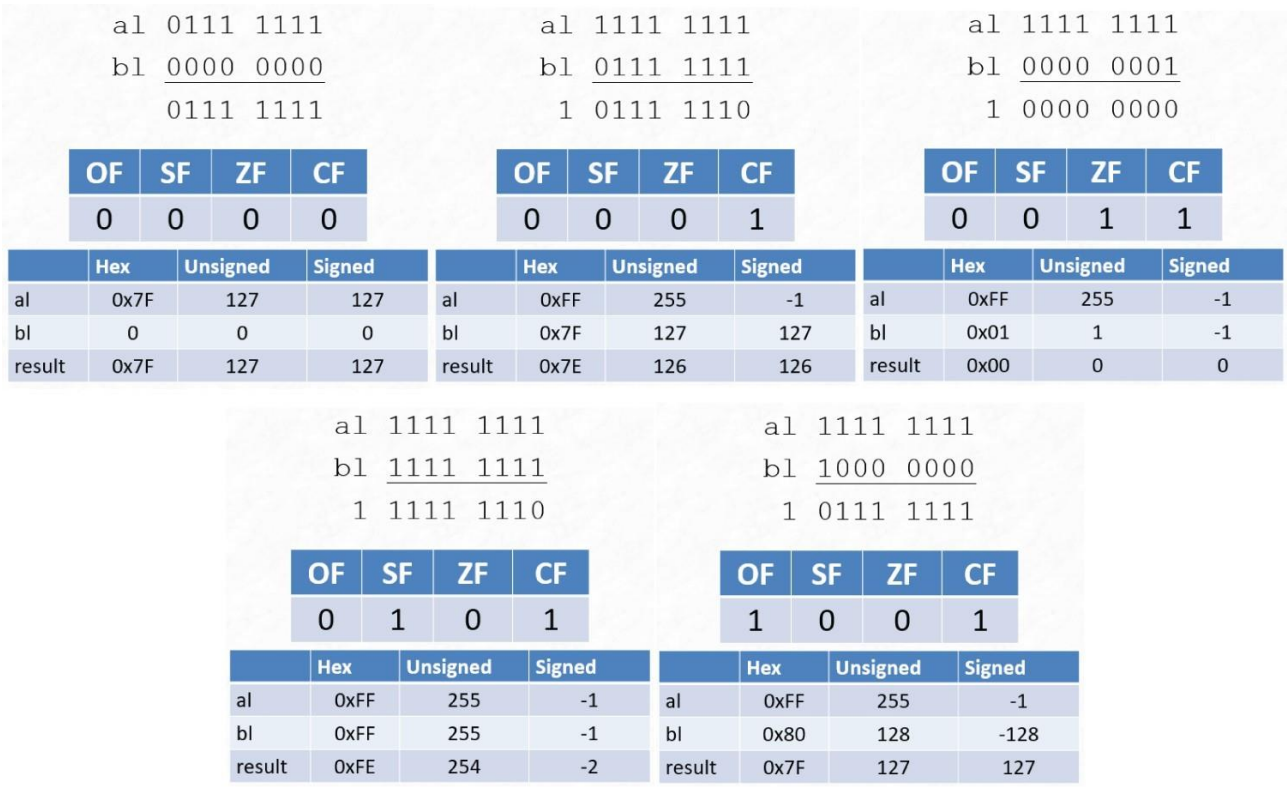


Рисунок 5.1: Эффекты добавления al,bl с различными входными данными

Обратите внимание на влияние, которое оказывает использование целых чисел со знаком или без знака на интерпретацию значений и их корректность. Например, вторая операция добавления дает правильный результат для значений со знаком, но неправильный для значений без знака.

sub

Дополнительная инструкция также потенциально может изменять те же флаги, что и add, то есть все четыре значимых флага состояния. На рисунке 5.2 показаны различные результаты sub al, bl с различными значениями al и bl.

Как и в случае с операцией добавления, корректность результата sub зависит от значений, сохраненных в al и bl. Например, обе версии первого примера верны, но только подписанная версия второго имеет правильный результат.

cmp
Инструкция cmp предназначена для сравнения двух значений, которые могут быть памятью, константами или регистром. Она работает путем вычитания второго операнда из первого операнда. Однако результат вычитания отбрасывается, но флаги настраиваются.

Цель cmp - определить, является ли одно значение больше, меньше или равно другому. Рассмотрим следующий пример, где eax < ebx:

```
mov eax, 0x100
mov ebx, 0x200
cmp eax, ebx ; вычисляет eax-ebx
```

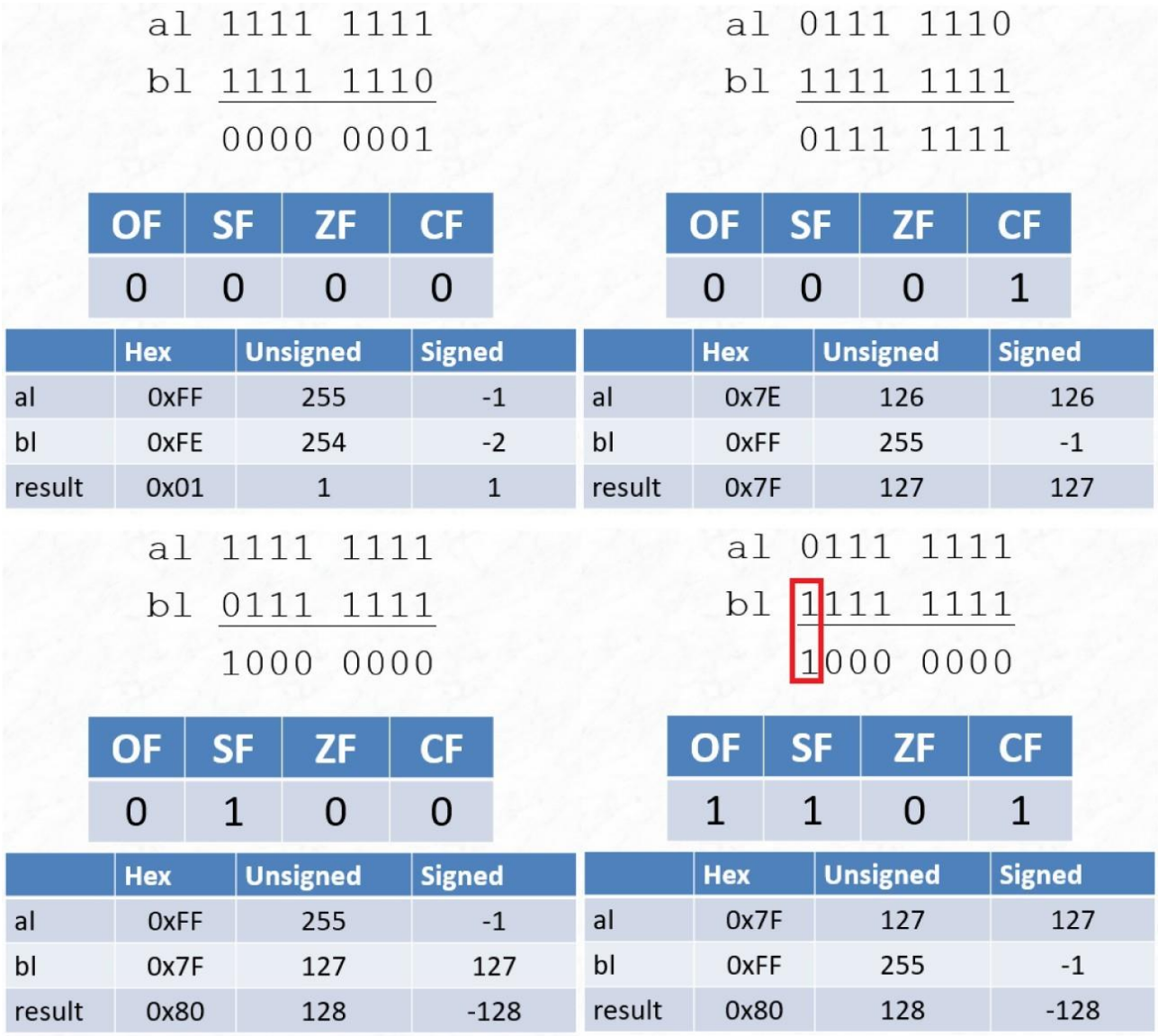


Рисунок 5.2: Эффекты sub al, bl с различными входными данными

Последней инструкцией здесь является вычитание, которое привело бы к отрицательному значению, поскольку ebx больше, чем eax. В результате флаг знака был бы установлен на 1 (что указывает на отрицательный результат), в то время как флаг нуля был бы установлен на 0 (т.е. результат не равен нулю).

В другом примере значение первого операнда, eax, может быть больше, чем значение второго операнда, ebx.

```
mov eax, 0x300
mov ebx, 0x200
cmp eax, ebx  ; evaluates eax-ebx
```

В этом случае результатом вычитания будет положительное значение. В результате флаги sign и zero будут равны нулю.

Последний потенциальный случай для cmp - это если два операнда равны, как показано здесь:

```
mov eax, 0x500
mov ebx, 0x500
cmp eax, ebx  ; evaluates eax-ebx
```

Если операнды равны, результат вычитания равен нулю, что установило бы флаг нуля, но не флаг знака. На рисунке 5.3 показана таблица истинности, суммирующая влияние операций cmp на флаги знака и нуля.

If...	SF	ZF
eax > ebx	0	0
eax = ebx	0	1
eax < ebx	1	0

Рисунок 5.3: таблица истинности cmp

test

Тестовая инструкция выполняет побитовое и между двумя операндами, которые могут быть памятью, константами или регистрами. Как и cmp, результат операции отбрасывается, но значения флагов корректируются.

Тестовая инструкция обычно используется для проверки того, установлен ли один или несколько определенных битов в пределах значения, путем установки флага нуля. Например, следующие инструкции проверяют, установлены ли биты 0 или 2:

```
mov ax, 0x1450
test ax, 0x05 ; check if bit 0 or 2 is set (0x5 is 0000 0101 in
binary)
```

Эти инструкции эквивалентны выполнению следующей математической операции:

```
  0001 0100 0101 0000      (0x1450)
& 0000 0000 0000 0101      (0x0005)
  0000 0000 0000 0000
```

Результат этой операции и равен нулю, что привело бы к установке нулевого флага. Это указывает на то, что ни бит 0, ни бит 2 не были установлены.

Следующие инструкции выполняют ту же проверку, когда значение 0x1451 помещается в ax:

```
mov ax, 0x1451
test ax, 0x05 ; check if bit 0 or 2 is set
```

Эти инструкции эквивалентны следующему расчету:

```
  0001 0100 0101 0001      (0x1451)
& 0000 0000 0000 0101      (0x0005)
  0000 0000 0000 0001
```

В этом случае результат операции and ненулевой, поэтому нулевой флаг не установлен. Это указывает на то, что был установлен по крайней мере один из двух битов.

Совет

Тестовая инструкция может быть использована для определения того, является ли число четным или нечетным.

Резюме

Коды условий используются для записи некоторых последствий операции, которые могут не отображаться в регистре назначения. Например, код условия может указывать, привела ли операция к нулю или вызвала переполнение. Отслеживание этих кодов условий важно для понимания текущего состояния приложения при его обратном проектировании.

Глава 6
Анализ и отладка ассемблерного кода
Предыдущие главы были посвящены теории и основам обратного проектирования. Изучение принципов работы x86 и распространенных форматов команд имеет важное значение для успеха.

В этой главе рассматривается практический подход к обратному проектированию и взлому программного обеспечения. В нем представлен gdb, мощный отладчик, и рассматриваются некоторые важные советы и рекомендации по обратному проектированию программного обеспечения и взлому.

Бинарный анализ
Анализ существующих исполняемых файлов во многом связан с обратным проектированием. Бинарный анализ может быть выполнен несколькими различными способами, включая статический и динамический анализ и отладку.

Статический и динамический анализ
Функциональность программы может быть проанализирована несколькими различными способами. Двумя основными методами являются статический и динамический анализ.

Статический анализ предполагает анализ исходного кода без его запуска. Статический анализ имеет несколько преимуществ, в том числе следующие:

- Хорошая отправная точка для дальнейшего анализа
- Безрисковый метод анализа потенциальных вредоносных программ
- Нет необходимости в доступе к специализированным архитектурам

Статический анализ имеет свои преимущества, одно из самых больших заключается в том, что он всегда доступен. Но он может отнимать много времени и не позволит уловить все. Всегда будут фрагменты кода, которые имеют смысл только во время выполнения. При анализе сложного кода, не наблюдая за его выполнением, может быть трудно или невозможно предугадать, куда может привести что-то вроде скачка. Кроме того, многие потоки кода продиктованы входными данными, вводимыми в программу, поэтому статического анализа недостаточно для определения того, куда пойдет выполнение кода, что затрудняет его анализ.

Динамический анализ - это дополнительный метод, который включает запуск программы и анализ ее поведения во время работы. Некоторые из преимуществ динамического анализа включают следующее:

- Более быстрый анализ
- Более широкое выявление потенциальных проблем

Динамический анализ может принимать множество различных форм. Что касается обратного проектирования, то одной из наиболее распространенных является отладка. Наблюдая за запущенным приложением, можно уточнить многие неизвестные во время статического анализа строки (например, куда код, скорее всего, перейдет). Однако динамический анализ означает выполнение рассматриваемого кода, и в зависимости от кода это не всегда может быть осуществимо. Это может быть отрывок из более крупного приложения, для него может потребоваться уникальная среда выполнения, к которой у вас нет доступа, или, в случае вредоносного ПО, оно может быть потенциально вредоносным при выполнении.

Отладка
Напомним, что целью реверс-инжиниринга и взлома программного обеспечения является понимание и модификация существующего программного обеспечения. Отладка - один из

самых быстрых и эффективных способов достижения этой цели. Динамически анализируя функциональность программы и изменяя ее поведение "на лету", можно собрать информацию, необходимую для взлома, и протестировать потенциальные уязвимости программного обеспечения.

Отладка обычно представляет собой многоступенчатый процесс. Типичный процесс отладки включает следующее:

1. Установите точки останова в точках интереса.
2. Запустите код.
3. Выполнение приостанавливается («прерывается») в точке останова.
4. Проверьте состояние программы.
5. При необходимости внесите изменения.
6. Повторять.

Точки останова

Точки останова дают указание процессору остановить выполнение программы в определенной точке. Точки останова бывают одной из двух форм:

Программное обеспечение: Программные точки останова устанавливаются в инструкциях по сборке, и их количество не ограничено.
Аппаратное обеспечение: Ограниченное количество аппаратных точек останова (четыре в x86) может быть установлено в инструкциях по сборке или при доступе к памяти.
В этой книге ранние лабораторные работы посвящены использованию программных точек останова, а аппаратные точки останова появятся в более поздних лабораторных работах. В этой книге будет продемонстрировано использование различных отладчиков, и настройка точек останова в каждом из них будет отличаться.

Программные точки останова

Программные точки останова являются параметром по умолчанию для большинства отладчиков. При установке программной точки останова на самом деле происходит то, что отладчик фактически изменяет инструкцию, заменяя ее инструкцией точки останова. В x86 это инструкция int3 (0xcc). Программная точка останова ограничена для выполнения, что означает, что для выполнения точки останова должна быть выполнена инструкция int3.

Когда процессор достигает команды точки останова, он останавливает выполнение и передает управление обратно отладчику. Это позволяет обратному инженеру проверить состояние программы и, возможно, внести изменения.

Основным ограничением программных точек останова является то, что они могут быть легко обнаружены программой, которая считывает свою собственную память. С помощью анти-отладки программа может удалить точку останова или предпринять другие защитные действия в ответ на нее.

Аппаратные точки останова

Большинство отладчиков поддерживают аппаратные точки останова. Однако, как правило, их необходимо выбирать и настраивать вручную.

Аппаратная точка останова не изменяет код программы, как это делает программная точка останова. Вместо этого адреса точек останова хранятся в аппаратных регистрах.

В x86 отладочные регистры DR0-7 используются для аппаратных точек останова. Регистры DR0-4 содержат адреса точек останова, в то время как DR6,7 хранят информацию о

конфигурации.

Аппаратные точки останова могут быть сконфигурированы для прерывания при выполнении, чтении или записи определенного адреса. Когда процессор обнаруживает условие, соответствующее регистрам точек останова, он передает управление отладчику.

Аппаратные точки останова полезны, поскольку они могут обнаруживать доступ к памяти. Например, аппаратная точка останова может использоваться для определения того, где в коде задан определенный байт или используется строка.

Аппаратные точки останова также полезны для обхода защиты программы от программных точек останова. Если программа сканирует свой собственный код в поисках инструкций int3, она пропустит аппаратные точки останова, которые не изменяют код. Это не защита от отладчика; обладая передовыми знаниями системы, приложение может копать достаточно глубоко, чтобы отслеживать аппаратные точки останова, но это значительно повышает планку по сравнению с программными точками останова.

gdb

Отладчик GNU (gdb) является стандартом де-факто для отладки в Linux. Он поставляется во многих дистрибутивах Linux и может быть установлен в любом из них. Некоторые из ключевых функций, которые предоставляет gdb, включают следующее:

- Отладчик командной строки (без графического интерфейса)
- С возможностью создания сценариев
- Поддержка удаленной отладки

gdb является настолько распространенным отладчиком, что многие системы и процессоры включают заглушку gdb для поддержки отладки gdb. В то время как многие отладчики ограничены несколькими архитектурами и платформами, gdb работает на сотнях. В этой книге будет рассмотрено множество различных отладчиков, а позже будут представлены отладчики на основе графического интерфейса пользователя (GUI). Однако важно фундаментально понимать, как использовать GDB, который представляет собой командную строку. Многие “красивые” отладчики используют GDB и его протокол под капотом; они просто обернули его красивым интерфейсом.

Отладка с помощью gdb

Как программой командной строки, gdb управляется путем ввода команд в командной строке, которая отображается как (gdb). Хотя интерфейс gdb может показаться архаичным, это чрезвычайно мощный и чрезвычайно популярный отладчик.

Одной из полезных функций gdb является то, что команды можно вводить в виде кратчайшей недвусмысленной формы команды. Например, run можно сократить до r, info registers - до info reg, а disassemble - до disas. С некоторыми из них вы познакомитесь по ходу работы.

Запуск gdb

gdb можно запустить с помощью команды gdb. Например, исполняемый файл print reg-shift.out можно запустить с помощью gdb print reg-shift.out, как показано на рисунке 6.1.

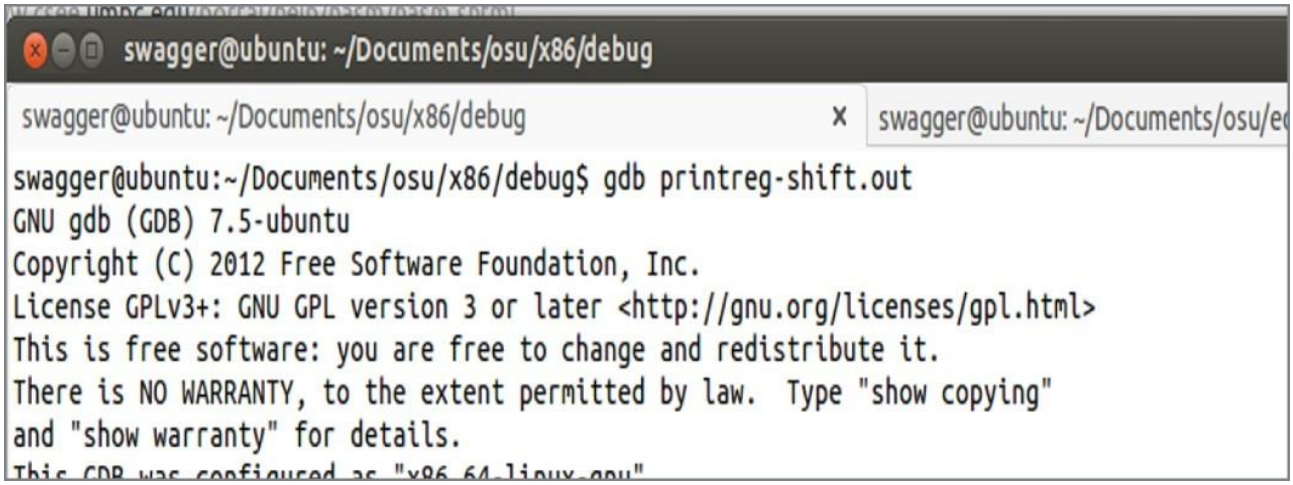


Рисунок 6.1: Команда gdb

Дизассемблирование с помощью gdb
Напомним, что x86 имеет несколько разных синтаксисов, включая Intel и AT&T. Инструкция для указания синтаксиса Intel в gdb - это (gdb) set disassembly-flavor intel.

После настройки режима дизассемблирования есть несколько различных вариантов запуска отладки, включая следующие:

- disassemble запускает дизассемблирование с указателя текущей инструкции.
- disassemble address запускает дизассемблирование по указанному адресу.
- disassemble label запускает дизассемблирование по указанной метке (loop, main и т.д.).

На рисунке 6.2 показан пример дизассемблирования с использованием метки, где нужный сегмент кода начинается с метки цикла. Изображение слева представляет исходный код сборки, а правая половина изображения показывает эквивалентную дизассемблирование в gdb.

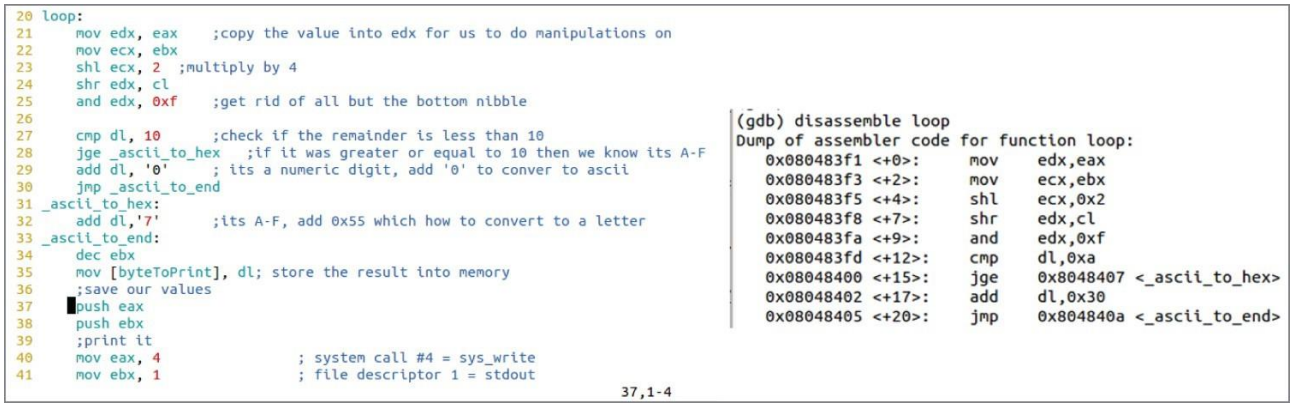


Рисунок 6.2: Дизассемблирование в gdb

После указания начальной точки можно указать определенное количество инструкций для дизассемблирования. Например, команда disassemble main +50 запустит дизассемблирование с метки main и напечатает 50 инструкций.

Запуск и остановка кода в gdb

Команда run используется для выполнения программы с самого начала. Это приведет к удалению любой информации о состоянии, полученной при запуске программы до этого момента.

Команда continue используется для возобновления выполнения после паузы. Например, для остановки выполнения для просмотра стека программы можно использовать точку останова, за которой следует команда continue для возобновления.

Выполнение программы может быть завершено командами quit и kill. kill завершает запущенный процесс, в то время как quit делает это и покидает gdb.

Точки останова gdb
Точки останова останавливают выполнение кода, позволяя проанализировать состояние программы. В gdb команда break address указывает точку останова по определенному адресу, в то время как break label использует метку для указания желаемого местоположения точки останова, как показано на рисунке 6.3.

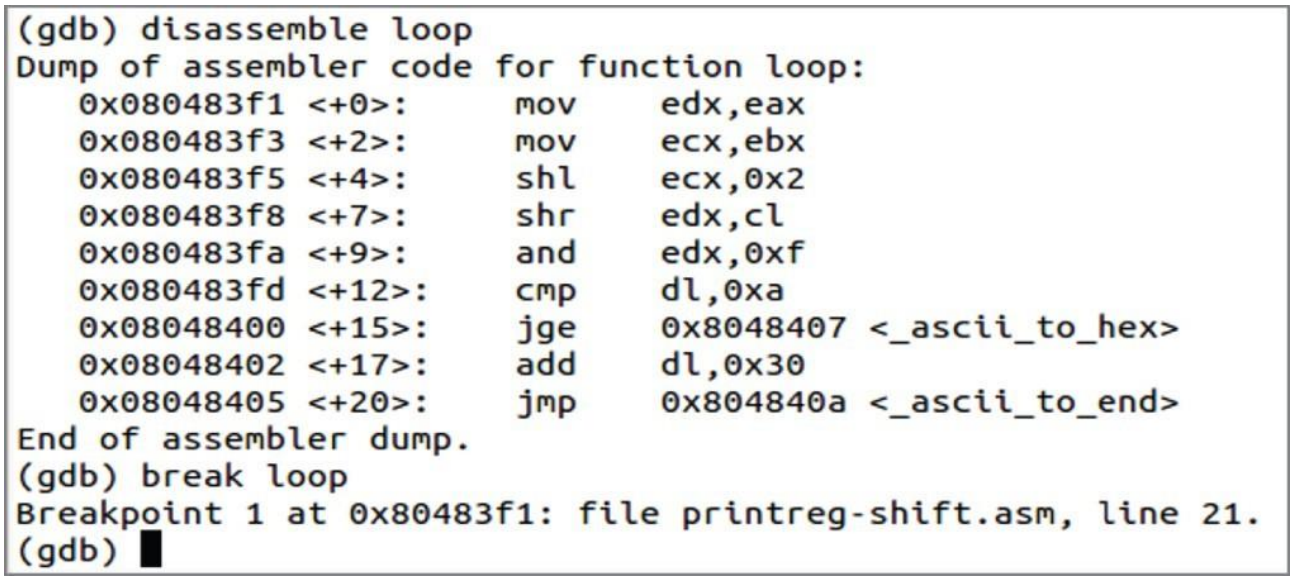


Рисунок 6.3: Установка точки останова в gdb

Информационные команды gdb
Команда info в gdb может использоваться для доступа к различным типам информации. Некоторые распространенные информационные команды включают следующее:

- информационные файлы: Показывает различные части разобранного файла. На рисунке 6.4 показан пример простого исполняемого файла с именем a.out.

```
(gdb) info files
Symbols from "/home/swagger/Documents/osu/x86/a.out".
Unix child process:
        Using the running image of child process 61165.
        While running this, GDB does not access memory from...
Local exec file:
        `/home/swagger/Documents/osu/x86/a.out', file type elf32-i386.
        Entry point: 0x80480d1
        0x08048080 - 0x080480dd is .text
        0x080490e0 - 0x080490e5 is .data
(gdb)
```

Рисунок 6.4: команда gdb info files

- информационные точки останова: Перечисляет текущие определенные точки останова для дизассемблированной программы.
- информационный регистр: Отображает текущие значения регистров x86, как показано на рисунке 6.5.
- информационные переменные: Отображает все определенные переменные в приложении, как показано на рисунке 6.6.

В дополнение к команде info register, также можно распечатать значения отдельных регистров с помощью команды print $reg, как показано здесь:

```
(gdb) print $esp
$1 = (void *) 0xffffd260
(gdb)
```

```
Starting program: /home/swagger/Documents/osu/x86/debug/printreg-shift.out

Breakpoint 1, loop () at printreg-shift.asm:21
21              mov edx, eax    ;copy the value into edx for us to do manipulations on
(gdb) info register
eax            0xabcdef12       -1412567278
ecx            0xffffd304       -11516
edx            0xffffd294       -11628
ebx            0x7      7
esp            0xffffd268       0xffffd268
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x80483f1        0x80483f1 <loop>
eflags         0x246    [ PF ZF IF ]   <-- Flags currently set
cs             0x23     35
ss             0x2b     43
ds             0x2b     43   <-- Decimal Value
es             0x2b     43
fs             0x0      0
gs             0x63     99   <-- Hex Value
(gdb)
```

Рисунок 6.5: команда регистрации информации gdb

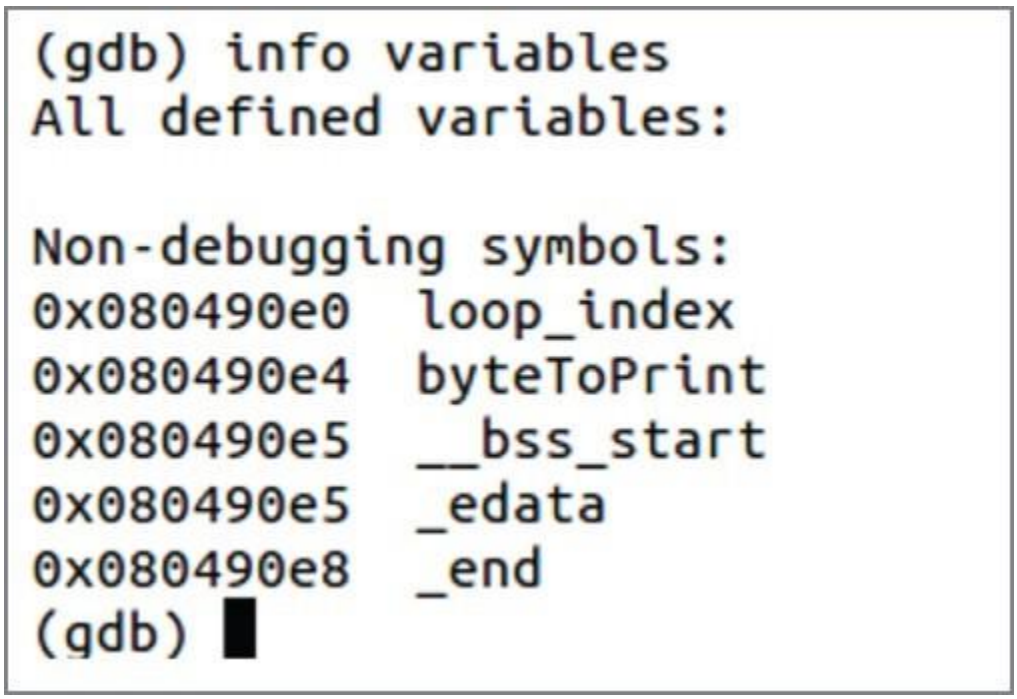


Рисунок 6.6: команда информационной переменной gdb

Пошаговое выполнение инструкций
Команды run и continue просто запускают программу снова, пока что-то не заставит выполнение остановиться, например точка останова. Это затрудняет наблюдение за тем, что делает программа или как изменяются переменные с течением времени.

Команда step выполняет по одной инструкции за раз, как показано на рисунке 6.7, что позволяет провести более глубокий анализ. Обратите внимание, что комментарии показаны, поскольку приложение было создано в режиме отладки.

```
(gdb) stepi
34              dec ebx
(gdb) stepi
35              mov [byteToPrint], dl; store the result into memory
(gdb) stepi
37              push eax
(gdb) stepi
38              push ebx
(gdb) stepi
40              mov eax, 4              ; system call #4 = sys_write
(gdb) stepi
41              mov ebx, 1              ; file descriptor 1 = stdout
(gdb)
```

Рисунок 6.7: команда gdb stepi

Примечание

Отладочная информация, такая как имена функций/переменных, комментарии и т.д., может быть включена при компиляции с

использованием флага отладки, который поддерживается большинством компиляторов. Например, в gcc/g++ это флаг -g. Однако это редко встречается при обратном проектировании коммерческого исполняемого файла.

Если следующей командой для выполнения является вызов функции, шаг i будет следовать за вызовом вызываемой функции. Во многих случаях это нежелательно, особенно если функция хорошо понятна. Например, знание того, что инструкция напечатает строку, является достаточной информацией, и нет необходимости проверять каждую инструкцию в функции printf.

Следующая команда позволяет перейти к вызову функции. Это выполнит вызов функции и перейдет к следующей видимой инструкции.

Проверка памяти
В gdb для проверки памяти может использоваться команда x (для “проверки памяти”). Синтаксис команды - x/nfu addr.

В этой команде n, f и u являются необязательными параметрами со следующими значениями:

- n: Указывает количество единиц (u) памяти, которые должны отображаться
- f: Указывает формат отображения
  - s: Строка, заканчивающаяся нулем
  - i: Машинная инструкция
  - x: шестнадцатеричный (по умолчанию)
- u: Указывает размер единицы
  - b: Байты
  - h: Полуслова (два байта)
  - w: Слова (по умолчанию)
  - g: Гигантские слова (восемь байт)

На рисунке 6.6 показана переменная с именем byte для печати, которая была расположена по адресу 0x080490e4. Чтобы отобразить 16 байт памяти в этом месте, команда должна быть x/16x 0x80490e4, как показано на рисунке 6.8. Выходные данные команды показывают, что интересующий байт имеет значение 0x41, которое является A в ASCII.

```
(gdb) x/16x 0x80490e4
0x80490e4 <byteToPrint>:        0x00000041      0x00000001      0x00210000
0x00000014
0x80490f4:      0x00000001      0x00000064      0x08048080      0x00000000
0x8049104:      0x000a0044      0x08048080      0x00000000      0x000b0044
0x8049114:      0x08048085      0x00000000      0x000c0044      0x0804808a
(gdb)
```

Рисунок 6.8: команда gdb x

В дополнение к использованию адресов памяти, команда x может также использовать

регистры для указания местоположения дампа. Например, на рисунке 6.9 показан пример сброса 10 байт в шестнадцатеричном формате в местоположение, указанное esp.

```
(gdb) x/10x $esp
0xffffd260:     0x00000006      0xabcdef12      0x0804843b      0xf7e324d3
0xffffd270:     0x00000001      0xffffd304      0xffffd30c      0xf7fda858
0xffffd280:     0x00000000      0xffffd31c
(gdb)
```

Рисунок 6.9: Печать 10 байт с помощью команды gdb x

Ошибки сегментации
Ошибки сегментации, или segfaults, возникают, когда центральный процессор пытается прочитать или записать недоступную область памяти. Это может произойти из-за того, что указанное местоположение не существует или у центрального процессора отсутствуют разрешения, необходимые для доступа к этому местоположению памяти или его изменения.

Например, команда mov eax, [0x000000000] всегда приводит к ошибке сегмента. Причина этого заключается в том, что адрес 0x0 обычно не отображается и не доступен приложению, поэтому чтение из памяти адреса 0x0. gdb приведет к сбою сегмента.

Сбои сегмента могут возникать по множеству различных причин. При использовании программного обеспечения переполнение буфера может вызвать сбой сегмента. При взломе программного обеспечения сбои сегмента могут возникать, если программа неправильно исправлена или допущены ошибки при изменении выполнения во время отладки. По мере того, как вы начнете писать ассемблерный код и манипулировать им, вы станете хорошими друзьями с segfault. Держать поблизости стресс-мяч, который вы используете каждый раз, когда видите segfault, может быть полезным при взломе. Нет худа без добра, при подключении к GDB, когда происходит сбой сегмента, gdb покажет строку, которая его вызвала, что полезно для отслеживания того, где код вышел из строя.

Лаборатория: Shark Sim 3000
Эта лабораторная работа представляет собой пробную задачу по обратному проектированию. Приложение намеренно разработано таким образом, чтобы его поведение было неочевидным. Опыт расшифровки и анализа подобных программ является основой взлома программного обеспечения.

Перейдите на страницу книги на GitHub (https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES) и найдите лабораторную Shark Sim 3000.

Навыки
Эта лабораторная работа предназначена для проверки базовых навыков в области реверс-инжиниринга, включая способность разбирать на части и понимать неизвестную программу. Некоторые из тестируемых навыков включают следующее:

- ASCII
- Коды условий
- Отладка скомпилированных программ
- Расшифровка неизвестных инструкций по сборке и программ

Выводы

Отладка - бесценный инструмент при реверс-инжиниринге приложения. Фундаментальной частью реверс—инжиниринга является расшифровка новых инструкций - не всегда хватает времени, чтобы разобраться во всем!

Один из секретов успеха в обратном проектировании заключается в том, чтобы не запутаться в неизвестных инструкциях по сборке. Постарайтесь быстро понять основы того, что они делают, чтобы вы могли продолжить. В следующих главах вы продолжите работать с gdb и другими более мощными отладчиками. Теперь, когда вы начали изменять сборку, вы можете перейти к выполнению этого вообще без какого-либо источника.

Отключение шума

Эффективное обратное проектирование и взлом программного обеспечения требуют владения несколькими различными наборами навыков. Однако одним из наиболее важных является способность отключаться от шума и концентрироваться на том, что имеет значение.

Даже небольшие программы содержат слишком много кода, чтобы проанализировать все. Подавляющее большинство инструкций не имеют отношения к основным функциям приложения и являются пустой тратой времени для обратного проектирования. Часто знание того, на чем следует сосредоточиться, менее важно, чем знание того, на чем не следует сосредотачиваться.

При определении того, на чем сосредотачиваться, а на чем нет, важно понимать основы. Некоторые чрезвычайно распространенные коды, которые должны быть сразу узнаваемы, включают следующее:

- Конструкции потока управления
- Расположение стека (локальные переменные, входящие параметры и исходящие параметры)
- Стандартные функции компилятора (прологи/эпилоги, "канарейки", распределение стека и управление регистрами)

Все это будет подробно рассмотрено в следующих главах, чтобы вы стали профессионалом в их быстром распознавании.

Существует порядок операций, которому следует следовать при определении приоритетности усилий по обратному проектированию: вызовы функций ➪ поток управления ➪ инструкции ➪ шаблон.

- Вызовы функций: Сосредоточьтесь на определении того, какие функции вызываются. Часто достаточно знать, что функция вызывает CreateDialog, чтобы понять ее назначение.
- Поток управления: При необходимости изучите поток управления, например, определите, что CreateDialog вызывается в цикле внутри этой функции.
- Отдельные инструкции: Если этого недостаточно, изучите отдельные инструкции.
- Шаблон компилятора: Изучение шаблона почти никогда не бывает полезным для обновления программного обеспечения или взлома. Однако важно понимать типичный шаблон, чтобы его можно было быстро идентифицировать и игнорировать.

Резюме

В этой главе представлен gdb, мощный и широко используемый отладчик для систем Linux. Знакомство и практический опыт работы с gdb важны для начинающего реинжиниринга или взломщика, поскольку этот инструмент можно использовать для анализа программного обеспечения для широкого спектра различных систем.

Глава 7
Функции и поток управления
Программа представляет собой последовательность инструкций, и приложение не может переходить линейно от одной инструкции к другой. При реверсировании и взломе приложения жизненно важно понимать потоки управления и различные факторы, которые могут на них влиять, такие как операторы if и циклы в языках более высокого уровня.

При реверсировании функции в x86 или языке более высокого уровня вы, скорее всего, также столкнетесь с функциями. В этой главе также рассматривается, как функции работают в x86 и их влияние на программный стек.

Поток управления
До сих пор ассемблерный код, рассмотренный в этой книге, следовал последовательному потоку инструкций. Выполнение просто продолжается сверху вниз. Однако большинство приложений не являются полностью последовательными. Рассмотрим следующий блок кода:

```
if (x) {
        // Do something
}
```

При выполнении этого кода процессор оценит условие x и определит, истинно ли оно. Если да, он переходит к инструкциям в блоке if.

Однако, если условие x не истинно, то инструкции в блоке if пропускаются. Для этого требуется возможность указывать процессору выполнять одни инструкции, а не другие, изменяя поток выполнения.

Указатель инструкции
Регистр eip известен как указатель команды и содержит адрес следующей команды для выполнения. Процессор автоматически увеличивает значение, сохраненное в eip, после выполнения команды.

Разрешение увеличивать eip после каждой инструкции позволяет выполнять последовательную серию инструкций. Однако в некоторых случаях мы хотим выполнить код условно. Для этого требуется другое обновление eip. Однако eip нельзя манипулировать напрямую (напомним, что это регистр специального назначения [SPR]). Вместо этого для настройки eip используются инструкции потока управления.

Инструкции потока управления
Наиболее распространенные отклонения от нормального потока выполнения, которые приводят к изменениям в eip, известны как переходы или ветви. Например, следующий блок кода имеет ветвление в инструкции if:

```
int x = 1;
int y = 2*x;
if (!y) { // branch!
    x = 2;
}
```

При сборке кода высокого уровня, подобного этому, инструкции перехода используются для указания того, на что должен быть установлен регистр eip. Инструкция перехода имеет

синтаксис jmp op, где op может быть адресом памяти или меткой.

Инструкция jmp - это не условный переход, который выполняется всегда (условные переходы рассматриваются далее в этой главе).

jmp
Инструкция jmp имеет синтаксис jmp op. Его цель - перенести поток управления программой в ячейку памяти op, установив eip на значение, хранящееся в op .

Некоторые примеры инструкций jmp включают следующее:

```
jmp eax        ; Copies eax into eip (branches to eax)
jmp label      ; Branches to the instruction at label
jmp $          ; An infinite loop in nasm(valuable
               ;debugging tool in assembly)
```

инструкции jmp могут использоваться для реализации различных функций. Например, следующие инструкции ведут отсчет от нуля в бесконечном цикле:

```
mov eax, 0
loop: add eax, 1
      jmp loop
```

Условные переходы
Условные переходы - это способ привязать выполнение перехода к условию true или false. Это определяет, следует ли выполнять переход на основе значений, сохраненных во флагах состояния. Например, рассмотрим следующие инструкции:

cmp eax, ebx
Команда jle (перейти меньше или равно) выполнит переход к указанному адресу или метке, если регистр флагов указывает, что предыдущее сравнение привело к значению меньше или равно. В этом случае команда непосредственно перед ней (cmp) используется для сравнения eax и ebx и установки флагов. Напомним, что cmp принимает операнд 1 минус операнд 2 и отбрасывает результат. Итак, чтобы получить условие меньше или равно, eax должно быть меньше или равно ebx . В этом случае процессор перейдет к метке готово . В противном случае переход будет пропущен, и выполнение продолжится до следующей инструкции после jle.

В языке x86 существует множество инструкций условного перехода. В таблице 7.1 перечислены эти инструкции и условия, которые определяют, будет ли выполнен переход.

Таблица 7.1: инструкции по условному переходу x86

Инструкция	Значение	Условия
`je`	Переход, если равно.	ZF = 1
`jz`	Переход, если последний результат равен нулю.	ZF = 1
`jne`	Переход, если не равно.	ZF = 0
`jge`	Переход, если больше или равно.	SF = OF

Инструкция	Значение	Условия
`jl`	Переход, если меньше.	SF != OF
`jle`	Переход, если меньше или равно.	ZF = 1 OR SF != OF
`jg`	Переход, если больше.	ZF = 0 AND SF == OF

Взглянув на таблицу, вы могли бы заметить, что некоторые инструкции имеют идентичные условия. Например, jz и je будут выполнять переход, если нулевой флаг (ZF) установлен равным единице. Это означает, что переход выполняется, если два указанных значения равны. Но логически они рассматриваются как разные вещи. “Перейти, если предыдущий результат равен нулю” может использоваться после вычитания двух чисел, тогда как “перейти равным”, скорее всего, используется после сравнения.

Например, рассмотрим случай, когда eax = ebx = 0x10. Команда cmp eax, ebx выполняет вычитание и при выполнении установит нулевой флаг. Обе команды jz и je выполнят переход, если они будут следовать этой инструкции.

Эти инструкции могут использоваться взаимозаменяемо, но обычно они выбираются на основе инструкции, используемой для установки флагов, определяющих переход. Например, если для выполнения условия используется команда sub eax, ebx, то, скорее всего, будет использоваться jz, поскольку вы смотрите на нулевой результат математической операции. Если используется команда cmp eax, ebx, то будет использоваться je, поскольку сравнение проверяет эквивалентность.

Помните, что cmp выполняет вычитание за кулисами, поэтому оно оказывает такое же влияние на флаги, как и операция sub. jz и je - синонимичные инструкции, разработанные исключительно для того, чтобы сделать ассемблерный код более читабельным.

Подводные камни условных переходов
Условные переходы используют флаги состояния, чтобы определить, следует ли выполнять переход. Но каждая инструкция работает изолированно, и условные переходы не знают, по какому сравнению или математическому выражению вы хотите выполнить условный переход. Это может вызвать проблемы, если флаги состояния изменяются между условной инструкцией и переходом. Хотя компилятор не допустил бы такой путаницы, если вы пишете ассемблер, вы будете вынуждены задуматься об этом.

Например, рассмотрим следующий набор инструкций:

```
cmp eax, ebx
cmp edx, ecx
jle done
```

В этом случае, возможно, вы хотите, чтобы команда cmp eax, ebx определяла, выполняется ли переход. Однако cmp edx, ecx устанавливает флаги последними перед переходом, перезаписывая предыдущие настройки. Следовательно, переход выполняется на основе результата второго сравнения, а не первого.

При использовании нескольких команд cmp подряд может быть очевидно, что последняя

команда cmp устанавливает флаги для перехода. Однако при использовании других инструкций это может быть менее очевидно, как в следующих инструкциях:

```
cmp eax, ebx
add ecx, 1
je done
```

В этой серии инструкций, возможно, предполагалось использовать инструкцию cmp для установки флагов для перехода. Однако инструкция add также обновляет флаги состояния и перезаписывает предыдущие настройки. Вместо перехода, если eax = ebx , переход выполняется, если команда add устанавливает нулевой флаг (т.е. ecx + 1 = 0).

Пример
Инструкции перехода обычно используются для реализации операторов if и циклов. Следующий ассемблерный код суммирует числа 0-4 с помощью цикла:

```
mov   eax, 0    ; initialize eax (accumulator) to 0
       mov   ecx, 0    ; initialize ecx (counter) to 0

loop:
       add   eax, ecx  ; add current iteration
       add   ecx, 1    ; increment counter
       cmp   ecx, 5    ; at 5 iterations yet?
       jne   loop      ; loop if not yet 5

done:
```

Итератор в этом цикле хранится в регистре ecx и инициализируется перед циклом. Регистр eax является регистром накопления и содержит текущую сумму.

Цикл начинается с добавления текущего счетчика цикла в накопитель, а затем увеличения счетчика цикла. Это реализует желаемую логику суммирования значений 0-4 по мере выполнения цикла.

Цикл начинается с добавления текущего счетчика цикла в накопитель, а затем увеличения счетчика цикла. Это реализует желаемую логику суммирования значений 0-4 по мере повторения цикла.

Ветвление происходит в инструкции jne (переход не равен) в предпоследней строке. Предыдущая инструкция - это cmp, которая проверяет, равен ли счетчик цикла 5, и соответствующим образом устанавливает флаги состояния. Если счетчик циклов не равен 5, запускается переход, и регистр eip устанавливается на адрес, указанный loop , начиная другую итерацию. Если счетчик циклов равен 5, переход не выполняется, и процессор продолжает работу с меткой done.

Логические конструкции в x86
C/C++ и аналогичные языки высокого уровня содержат множество логических конструкций, которые приводят к непоследовательному выполнению кода. Некоторые примеры включают следующее:

```
if (…) { … }
if (…) { … } else { … }
```

```
if (…) { … } else if (…) { … } else { … }
while (…) { … }
do { … } while (…);
for (…; …; …) { … }
switch (…) { … }
```

В ассемблере эти логические конструкции записываются с использованием комбинации инструкций сравнения (cmp) и перехода (jmp, je, jne, jl, jle, jg и jge). Когда код скомпилирован, компилятор автоматически выполнит преобразование в ассемблерный код.

При написании ассемблерного кода необходимо вручную выполнить преобразование из высокоуровневых концепций в ассемблер. Или при рассуждении о коде других людей важно уметь понимать, как эти структуры выглядят в ассемблере. Чтобы развить это распознавание, сосредоточьтесь на том, как бы вы взяли эти языковые концепции более высокого уровня и перевели их в assembly. Выполнение этого процесса состоит из двух этапов:

Удалите блоки кода: перепишите код, заменив логические конструкции операторами goto. Assemble: Перепишите программу на ассемблере.

if (…) {…}

Оператор if является одной из простейших логических конструкций высокого уровня. С блоками кода это выглядит следующим образом:

```
if (condition)
{
        code_if_true;
}
```

Первым шагом является удаление блоков кода. Блоки кода - это код, вложенный в фигурные скобки: {}. При удалении этих блоков кода используйте инструкции goto, которые сообщают коду, куда перейти для выполнения. Не все языки более высокого уровня имеют концепцию goto, но сосредоточьтесь на этом как на псевдокоде и используйте goto. Следующий код - это тот же оператор if, написанный без блоков кода:

```
if (!condition)
        goto skip_block;

code_if_true;

skip_block:
```

Обратите внимание, что в этой версии условие инвертировано. Это связано с тем, что переход мимо блока if происходит только в том случае, если условие ложно, в то время как блок if инструкции if определяет, что произойдет, если условие истинно. Удаление блоков кода всегда будет включать инвертирование условия.

Преобразование этого из псевдокода в функциональное приложение требует замены condition и code_if_true фактическим кодом.

WITH BLOCKS	WITHOUT BLOCKS

if (x==5)
{
x++;
y=x;
}
	if (x!=5)
	goto skip_block;
	x++;
	y=x;
	skip_block:

После удаления блоков кода преобразование кода в ассемблер становится намного проще. Затем это может быть напрямую сопоставлено с их эквивалентами x86.

CODE	X86 ASSEMBLY

if (x!=5)
goto skip_block;
x++;
y=x;
skip_block:
	cmp dword [x], 5
	jne skip_block
	inc dword [x]
	mov eax, [x]
	mov [y], eax
	skip_block:

if (…) { … } else { … }

Добавление оператора else к конструкции if увеличивает сложность и требуемое количество переходов. В дополнение к пропуску блока if, если условие оценивается как ложное, конструкция if (...) { … } else { … } перепрыгивает через блок else после выполнения кода в блоке if.

Следующие примеры показывают, как эта логическая конструкция выглядит с блоками и без них:

WITH BLOCKS	WITHOUT BLOCKS

if (condition)
{
code_if_true;
}
else
{
code_if_false;
	if (!condition)
	goto false_block;
	code_if_true;
	goto skip_block;
	false_block:

WITH BLOCKS	WITHOUT BLOCKS
}	code_if_false; skip_block:

Обратите внимание, что код использует две разные метки в своих инструкциях goto. Метка false_block используется для пропуска блока if, если условие ложно, в то время как метка skip_block используется для пропуска блока else после выполнения блока if. Как и раньше, инвертируйте условный оператор при написании кода без блоков.

Замена псевдокода реальным кодом приводит к следующему результату с блоками кода и без блоков кода:

WITH BLOCKS	WITHOUT BLOCKS

if (x)
{
    x++;
}
else
{
    x--;
}
	if (!x)
	    goto false_block;
	
	x++;
	goto skip_block;
	
	false_block:
	x--;
	
	skip_block:

Как и прежде, удаление блоков упрощает преобразование высокоуровневого кода в ассемблерный.

CODE	X86 ASSEMBLY

if (!x)
    goto false_block;
 	
x++;
goto skip_block;
 	
false_block:
x--;
 	
skip_block:
	cmp dword [x], 0
	je false_block
	
	inc dword [x]
	jmp skip_block
	
	false_block:
	dec dword [x]
	
	skip_block:

if (…) { … } else if { … } else { … }

операторы if можно сделать более сложными и вычислять несколько разных условий. Ниже демонстрируется оператор if с else if и else с блоками и без них. Но процесс все равно остается тем же. Инвертируйте условие и добавьте gotos.

WITH BLOCKS	WITHOUT BLOCKS
if (condition_1)	if (!condition_1)

WITH BLOCKS	WITHOUT BLOCKS

{
    code_if_1;
}
else if (condition_2)
{
    code_if_2;
}
else
{
    code_if_false;
}
	goto test_2;
	code_if_1;
	goto skip_block;
	
	test_2:
	if (!condition_2)
	    goto false_block;
	code_if_2;
	goto skip_block;
	
	false_block:
	code_if_false;
	
	skip_block:

В этой версии кода необходимо несколько меток и переходов, чтобы преобразовать код в версию без блоков. Рассмотрим реальный пример, в котором реализована система оценки с чрезвычайно сложной кривой.

WITH BLOCKS	WITHOUT BLOCKS

if (score>70)
{
    grade='a';
}
else if (score>50)
{
    grade='b';
}
else
{
    grade='c';
}
	if (score<=70)
	    goto test_2;
	grade='a';
	goto skip_block;
	
	test_2:
	if (score<=50)
	    goto false_block;
	grade='b';
	goto skip_block;
	
	false_block:
	grade='c';
	
	skip_block:

Обратите внимание, что, опять же, для преобразования в версию без блоков требуется изменить условия. Операторы strictly less than становятся больше или равны 1. Следующий пример показывает, как этот код затем легко переводится в assembly:

CODE	X86 ASSEMBLY

if (score<=70)
    goto test_2;
grade='a';
goto skip_block;
 	
test_2:
if (score<=50)
    goto false_block;
	cmp dword [score], 70
	jle test_2
	mov byte [grade], 'a'
	jmp skip_block
	
	test_2:
	cmp dword [score], 50
	jle false_block

CODE	X86 ASSEMBLY
grade='b'; goto skip_block; false_block: grade='c'; skip_block:	mov byte [grade], 'b' jmp skip_block false_block: mov byte [grade], 'c' skip_block:

do { ... } while (...);

Языки программирования более высокого уровня имеют ряд различных структур циклов, каждая из которых работает немного по-разному. Цикл do...while гарантированно выполнит по крайней мере одну итерацию перед вычислением условия, которое завершит цикл. Ниже приведен пример цикла do...while с блоками:

```
do
{
    code;
}
while (condition);
```

В отличие от операторов if, цикл do...while оценивает свое условие в конце, поэтому дальнейшие итерации цикла требуют перехода назад. Как и ранее, этот код необходимо переписать без блоков кода. Ниже показан тот же цикл do...while, использующий операторы goto вместо блоков:

```
loop:

code;

if (condition)
    goto loop;
```

В отличие от оператора if, цикл do..while не инвертирует проверяемое условие. Это связано с тем, что обратный переход выполняется только в том случае, если условие истинно и требуется еще одна итерация цикла.

Теперь взгляните на версию кода, использующую реальные условия и логику.

WITH BLOCKS	WITHOUT BLOCKS

do
{
    y*=x;
    x--;
}
while (x);
	loop:
	y*=x;
	x--;
	
	if (x)
	    goto loop;

Преобразовать этот код так, чтобы он не использовал блоки, относительно просто, поскольку поток инструкций в основном одинаков. Основное отличие заключается в том, что оператор while заменяется if и goto.

В отличие от предыдущих примеров, большая часть сложности преобразования этого в ассемблер заключается в сложности примера кода, а не ветвей.

CODE	X86 ASSEMBLY

loop:
 	
y*=x;
x--;
 	
if (x)
goto loop;
	loop:
	
	mov eax, [y]
	mul dword [x]
	mov [y], eax
	
	sub dword [x], 1
	
	cmp dword [x], 0
	jne loop

while (...) { ... }

Цикл do...while гарантирует, что перед вычислением условия будет выполнена одна итерация цикла. Цикл while немедленно вычисляет условие, поэтому код внутри цикла может вообще не выполняться. Следующий код демонстрирует цикл while с блоками кода и без них. Цикл while может быть разбит на оператор if и, таким образом, следует обычному шаблону преобразования оператора if.

WITH BLOCKS	WITHOUT BLOCKS

while (condition)
{
code;
}
	loop:
	if (!condition)
	goto done;
	
	code;
	goto loop;
	
	done:

Обратите внимание, что поскольку условие вычисляется в начале, оно инвертируется, как в операторах if. Ниже показано, как это может выглядеть при преобразовании из псевдокода в реальный код:

WITH BLOCKS	WITHOUT BLOCKS

while (tired)
{
sleep();
}
	loop:
	if (!tired)
	goto done;
	
	sleep();
	goto loop;
	
	done:

После преобразования кода для удаления блоков его можно перевести в сборку x86, как показано ниже:

CODE	X86 ASSEMBLY

loop:
if (!tired)
    goto done;
 	
sleep();
goto loop;
 	
done:
	loop:
	cmp dword [tired], 0
	je done
	
	call sleep
	jmp loop
	
	done:

for (...; ...; ...) { ... }

Цикл for работает иначе, чем цикл while или do… while. Вместо того, чтобы запускать переменную количество раз на основе условия, цикл for включает условие цикла плюс инициализирует значение и обновляет значение.

Оператор for включает в себя три выражения. Первый из них инициализирует счетчик цикла. Второй определяет условие для завершения выполнения цикла, а третий определяет, как счетчик цикла будет изменяться между итерациями. Ниже показано это в псевдокоде с блоками кода и без них:

WITH BLOCKS	WITHOUT BLOCKS

for (expr_1; expr_2; expr_3)
{
    code;
}
	expr_1;
	
	loop:
	if (!expr_2)
	    goto done;
	code;
	expr_3;
	goto loop;
	
	done:

При преобразовании цикла for, чтобы не использовать блоки кода, три выражения в инструкции for разделяются по всему коду. Первое выражение является предварительным условием, которое выполняется только один раз перед циклом, а второе запускает цикл. Конечное условие, которое изменяет значение счетчика цикла, выполняется в конце каждой итерации цикла.

Эти три выражения в инструкции for легче понять при просмотре реального кода. Например, следующий код определяет счетчик цикла i и инициализирует его равным нулю. Этот счетчик циклов будет увеличиваться на 1 при каждой итерации цикла (i++), и цикл прекратит выполнение, когда i достигнет 100.

WITH BLOCKS	WITHOUT BLOCKS
for (i=0; i<100; i++) {	i=0;

WITH BLOCKS	WITHOUT BLOCKS
sum+=i; }	loop: if (i>=100)

	goto done;
	
	sum+=i;
	i++;
	goto loop;
	
	done:

Обратите внимание, что, подобно циклу while или оператору if, цикл for инвертирует условие. Еще раз, это связано с тем, что условие вычисляется в начале цикла, а не в конце, как в цикле do..while.

Ниже показано, как выглядит цикл for в сборке x86:

CODE	X86 ASSEMBLY

i=0;
 	
loop:
if (i>=100)
goto done;
 	
sum+=i;
i++;
goto loop;
 	
done:
	mov dword [i], 0
	
	loop:
	cmp dword [i], 100
	jge done
	
	mov eax,[i]
	add [sum],eax
	inc dword [i]
	
	jmp loop
	
	done:

switch (...) { ... }

Оператор switch - это логическая структура, существующая в некоторых языках программирования для упрощения условной логики. Целью оператора switch является выполнение одной из нескольких различных операций на основе значения определенной переменной. Следующий оператор switch вычисляет значение, хранящееся в op, и выводит символ, представляющий эту операцию:

```
typedef enum {ADD, SUB, MUL, DIV, MOD} op_t;

switch (op) {
    case ADD:
        c='+'; break;
    case SUB:
        c='-'; break;
    case MUL:
        c='*'; break;
```

```
    case DIV:
        c='/'; break;
    case MOD:
        c='%'; break;
    default:
        c='?'; break;
}
```

Любой оператор switch может быть написан с использованием серии операторов if и else-if. Однако это может быстро стать сложным для написания и неэффективным для выполнения. Потому что, если вы подходите для самого последнего случая, вам пришлось выполнить каждое предыдущее сравнение, чтобы определить это. Следующее является эквивалентом предыдущего оператора switch, использующего операторы if и else-if:

```
if (op==ADD)
    c='+';
else if (op==SUB)
    c='-';
else if (op==MUL)
    c='*';
else if (op==DIV)
    c='/';
else if (op==MOD)
    c='%';
else
    c='?';
```

Построение таблицы переходов

При оценке этого списка операторов if и else-if процессору необходимо выполнить пять проверок, чтобы выяснить, что делать с MOD, что очень неэффективно. Представьте сценарий, в котором были сотни вариантов... или тысячи. Невероятно неэффективно для выполнения. Чтобы оптимизировать этот процесс, компилятор может вместо этого создать таблицу переходов.

Таблица переходов - это структура данных сборки, которая предоставляет список целевых адресов для инструкции switch, как показано на рисунке 7.1. Как только инструкция switch определила, какой регистр является правильным, она может использовать этот номер регистра в качестве индекса в массиве адресов, что позволяет ей перейти непосредственно к нужному блок кода.

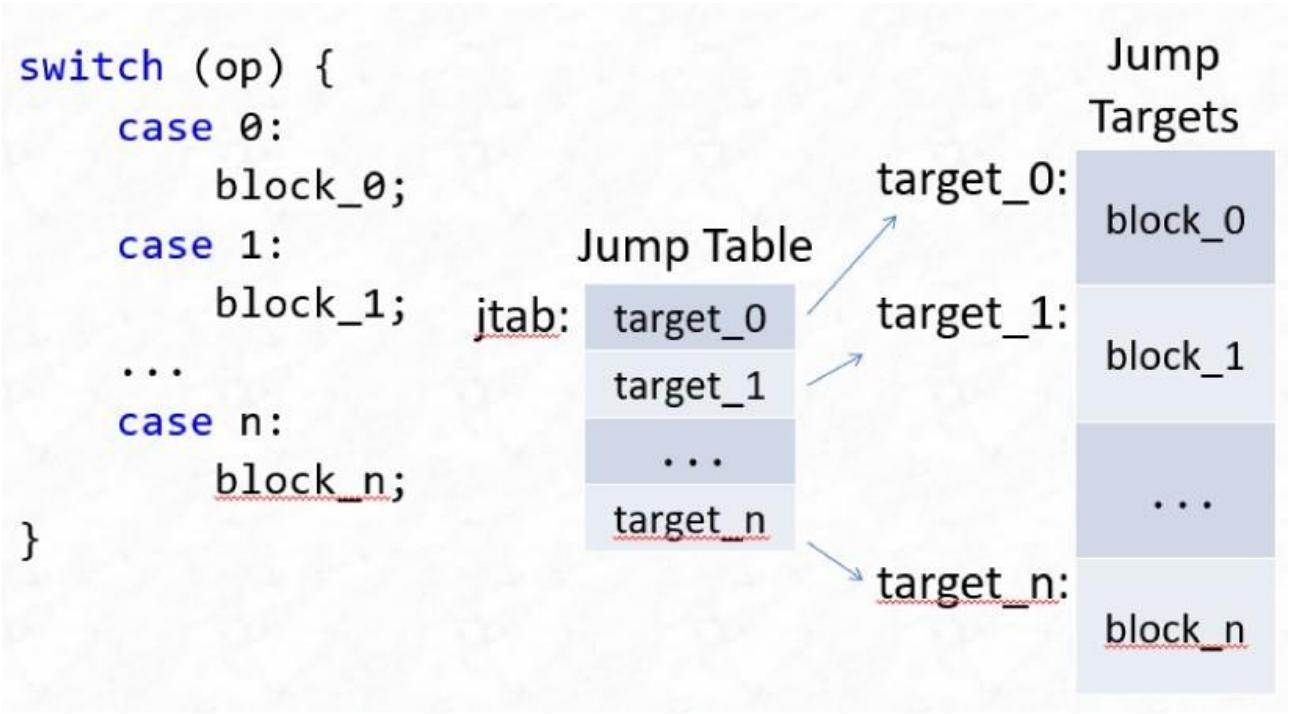


Рисунок 7.1: Пример таблицы переходов

Следующий код иллюстрирует, как может выглядеть программа, использующая таблицу переходов в assembly:

```
.section data
table:
dd target_0
dd target_1
dd target_2
dd target_3
dd target_4

.section text
mov eax, [op]
cmp eax, 5
jge default
jmp [table+eax*4]

target_0:
mov byte [c], '+'
jmp done

target_1:
mov byte [c], '-'
jmp done

target_2:
mov byte [c], '*'
jmp done
```

```
target_3:
mov byte [c], '/'
jmp done

target_4:
mov byte [c], '%'
 jmp done

default:
mov byte [c], '?'
jmp done

done:
```

Он начинается с таблицы переходов, которая содержит адреса различных блоков кода в памяти. Каждый из этих блоков кода, помеченный как target_x, перемещает соответствующий символ в байт [c], а затем переходит к метке done.

Между таблицей и блоками целевого кода находится код, который фактически реализует оператор switch под заголовком .section text. Этот код начинается с перемещения op в eax . Затем он проверяет, больше ли значение op или равно 5. Если это так, он переходит к регистру по умолчанию.

Если значение op меньше 5, оно сопоставляется с одним из целевых объектов. Используя его в качестве поиска в таблице переходов, процессор может получить адрес соответствующего блока кода и перейти к этому местоположению для выполнения кода.

В этом случае, используя таблицу переходов, время выполнения одинаково независимо от того, какой это регистр. Последний регистр не требует больше инструкций или сравнений, чем первый.

Пропущенные регистры
Таблица переходов предполагает, что варианты охватывают непрерывный диапазон значений. Например, в следующем примере кода оператор switch содержит варианты 1, 2, 4 и 5. В этом сценарии пропущенные 3 могут быть проблемой.

Что-то должно занять третье место в таблице прыжков.

```
switch (x) {
    case 0:
        …
    case 1:
        …
    case 2:
        …
    case 4:
        …
    default:
        …
}
```

Пропущенные значения в таблице переходов можно заполнить адресом по умолчанию или выполнить, если значения по умолчанию отсутствуют, как показано далее. Это приведет к тому, что процессор выполнит переход к нужному местоположению, когда значение op равно 3, и попытается перейти к соответствующему местоположению в таблице переходов.

Таблица переходов

target_0
target_1
target_2
default
target_4

Ненулевые базы
Таблица переходов предназначена для набора регистров, начинающихся с 0. Однако оператор switch может иметь ненулевые значения регистра, как показано в следующем примере кода:

```
switch (x) {
    case 'a':
        …
    case 'b':
        …
    case 'c':
        …
    case 'd':
        …
}
```

В этом случае необходимо найти способ обнулить регистры. С помощью ASCII необходимо найти наименьший регистр в таблице переходов, который в данном примере имеет значение a или 97. При реализации таблицы переходов для этого оператора switch код может использовать смещение, получая доступ к значениям в виде таблицы[x-97]. В следующей таблице переходов target_a будет указывать на регистр a оператора switch.

Таблица переходов

target_a
target_b
target_c
target_d

Непрактичные таблицы переходов
Компилятор будет использовать эти приемы для повышения эффективности кода, и вы тоже можете это сделать при написании кода вручную. Однако иногда таблица переходов просто не будет работать. Для примера рассмотрим следующий код:

```
switch (x) {
    case 1:
```

```
         printf("this is the beginning."); break;
    case 1000:
        printf("this is the end."); break;
}
```

Таблица переходов для этого оператора switch должна содержать 1000 записей, и 998 из них будут указывать на метку done. В этом случае оператор if/else является более эффективным вариантом.

Это не единственный случай, когда это непрактично; большая таблица переходов с сотнями обращений или таблица, использующая индекс, который нелегко обнулить, или имеющая слишком много пробелов в таблице, могут оказаться непрактичными. Но поскольку вы одновременно пишете и анализируете ассемблерный код, важно уметь понимать эти структуры и то, как они используются.

Продолжение
Некоторые языки более высокого уровня включают ключевое слово continue. continue используется внутри цикла и указывает процессору перейти к следующей итерации цикла, пропуская любые последующие инструкции.

Например, в следующем примере кода второй раздел с надписью code будет недоступен. Каждый раз, когда вычисляется оператор continue, процессор переходит непосредственно к оператору while.

```
do
{
    code;
    continue;
    code;
}
while (condition);
```

Цикл с оператором continue выглядит аналогично обычному циклу такого типа, если он написан без блоков кода. Как показано далее, continue может быть реализован с помощью goto, который переходит к метке, расположенной непосредственно перед условием цикла.

```
loop:

code;
goto check_condition;
code;

check_condition:
if (condition)
    goto loop;
```

В следующих примерах показан пример цикла с оператором continue, использующим реальный код:

WITH BLOCKS	WITHOUT BLOCKS
do	loop:

WITH BLOCKS	WITHOUT BLOCKS

`{`
`  x--;`
`  continue;`
`  x++;`
`}`
`while (x);`
	`x--;`
	`goto check_condition;`
	`x++;`
	
	`check_condition:`
	`if (x)`
	`  goto loop;`

В этом коде инструкция x++; никогда не будет выполнена, потому что continue всегда приводит к ее пропуску. Как правило, continue будет находиться внутри инструкции if, потому что в противном случае код, следующий за ней, бессмыслен. Этот надуманный пример предназначен для демонстрации того, как работает continue без сложных инструкций if, вложенных в цикл.

Как только код был преобразован для удаления блоков кода, его можно перевести в assembly. Как и ранее, переход может быть реализован с использованием условного перехода.

CODE	X86 ASSEMBLY

`loop:`
 	
`x--;`
`goto check_condition;`
`x++;`
 	
`check_condition:`
`if (x)`
`  goto loop;`
	`loop:`
	
	`sub dword [x], 1`
	`jmp check_condition`
	`add dword [x], 1`
	
	`check_condition:`
	`cmp dword [x], 0`
	`jne loop`

break

Ключевое слово break также существует в некоторых языках программирования, и оно предписывает процессору выйти из текущего цикла. Как и в случае с continue, второй фрагмент кода в следующем примере никогда не будет выполнен, поскольку break обычно находится внутри условного оператора, но это пример просто для демонстрации механизма break.

```
do
{
    code;
    break;
    code;
}
while (condition);
```

Следующий пример демонстрирует, как этот код был бы реализован без блоков кода. Ключевое слово break также заменено на goto, но это ключевое слово переходит к точке вне цикла, а не перед условным обозначением.

```
loop:
```

```
code;
goto break;
code;

if (condition)
    goto loop;

break:
```

В следующем примере оператор break завершит цикл после того, как операция x-- будет вычислена один раз. В этом случае оператор x++ и условие цикла никогда не будут выполнены, но, опять же, это пример кода. Это показывает, как вы могли бы легко преобразовать break в условие.

WITH BLOCKS	WITHOUT BLOCKS

`do`
`{`
`  x--;`
`  break;`
`  x++;`
`}`
`while (x);`
	`loop:`
	
	`x--;`
	`goto break;`
	`x++;`
	
	`if (x)`
	`  goto loop;`
	
	`break:`

В следующем примере показано, как код может быть преобразован в ассемблер:

CODE	X86 ASSEMBLY

`loop:`
 	
`x--;`
`goto break;`
`x++;`
 	
`if (x)`
`  goto loop;`
 	
`break:`
	`loop:`
	
	`sub dword [x], 1`
	`jmp break`
	`add dword [x], 1`
	
	`cmp dword [x], 0`
	`jne loop`
	
	`break:`

&&

В языках более высокого уровня условное выражение в операторе if или цикле потенциально может вычислять множество различных условий, таких как логическое значение И (&&). В следующем примере блок if будет выполнен только в том случае, если оба условия_1 и условия_2 имеют значение true.

```
if ( condition_1 && condition_2 ) {
```

```
    code;
}
```

При преобразовании этого кода для удаления блоков кода необходимо разбить составной оператор if на два разных оператора if. Каждый из них отменяет одно из условий, которые были включены в исходный оператор if.

```
if (!condition_1) goto skip_block;
if (!condition_2) goto skip_block;
true:
code;
skip_block:
```

После переписывания подобным образом (без блоков кода) следуйте той же формуле, чтобы перевести это в assembly.

||

Другим вариантом в операторах if является объединение нескольких условий с логическим значением OR (||). Пример этого показан в следующем псевдокоде:

```
if ( condition_1 || condition_2 ) {
    code;
}
```

Логическое значение OR также разбивается на два оператора if при удалении блоков кода. Однако эти операторы if выглядят иначе, чем с логическим значением AND.

```
if (condition_1) goto true;
if (!condition_2) goto skip_block;
true:
code;
skip_block:
```

Логическое выражение OR истинно, если истинно любое из двух условий. В предыдущем примере первое выражение if использует исходное условие_1 и переходит к метке true, поскольку, если оно истинно, нет необходимости вычислять второе условие.

Однако, если это условие ложно, код продолжает вычислять условие_2 . Это условие инвертируется, и переход переходит к концу инструкции if. Если условие_2 истинно, код переходит к блоку true. В противном случае он пропускает оператор if.

Стек
В ассемблере стек используется для хранения нескольких различных типов данных, включая следующие:

- Локальные переменные
- Пустое пространство
- Параметры и вызов функций

Стек получил свое название из-за того, что он представляет собой структуру last-in-first-out

(LIFO), подобную стопке бумаги. Это концептуально соответствует потоку управления программированием, и стеки чрезвычайно распространены в самых разных архитектурах.

Стек будет иметь указатель стека, который указывает на вершину стека. В этом разделе будут подробно рассмотрены несколько инструкций, специально посвященных манипулированию стеком. Нажатие сохраняет новое значение в текущей верхней части стека и обновляет указатель стека, чтобы указать новую верхнюю часть стека (думайте об этом как о добавлении нового листа бумаги в вашу стопку). Всплывающее окно переместит значение в верхней части стека в регистр или адрес памяти и обновит указатель стека, чтобы указать значение под ним, которое является новой вершиной стека (вынимая верхний лист бумаги из стопки).

Как работает стек

Программный стек увеличивается с базового адреса при вызове новых функций и уменьшается по мере возврата функций. По мере увеличения размера стека адреса уменьшаются, а по мере его сжатия адреса увеличиваются.

Стек x86

Стек не является принципиально отдельным объектом или пространством памяти на процессоре. Вместо этого это область памяти, которая была выделена и назначена для использования в качестве стека. Он существует в памяти вместе с остальной частью программы, а данные - это просто пространство, выделяемое приложением. В качестве примера, следующий код выделит 128 байт, которые вы могли бы использовать в качестве пространства стека:

```
section .data
times 128 db 0
stack equ $-4
```

В x86 есть два регистра, которые используются для управления стеком:

- esp: Указатель стека содержит адрес вершины стека.
- ebp: Базовый указатель содержит адрес основания фрейма стека.

Логически, стек должен расти вверх, а в x86 это означает уменьшение адреса, как показано на рисунке 7.2. Представьте это как перевернутый термометр, где 0 находится вверху, а наибольшее число — внизу.

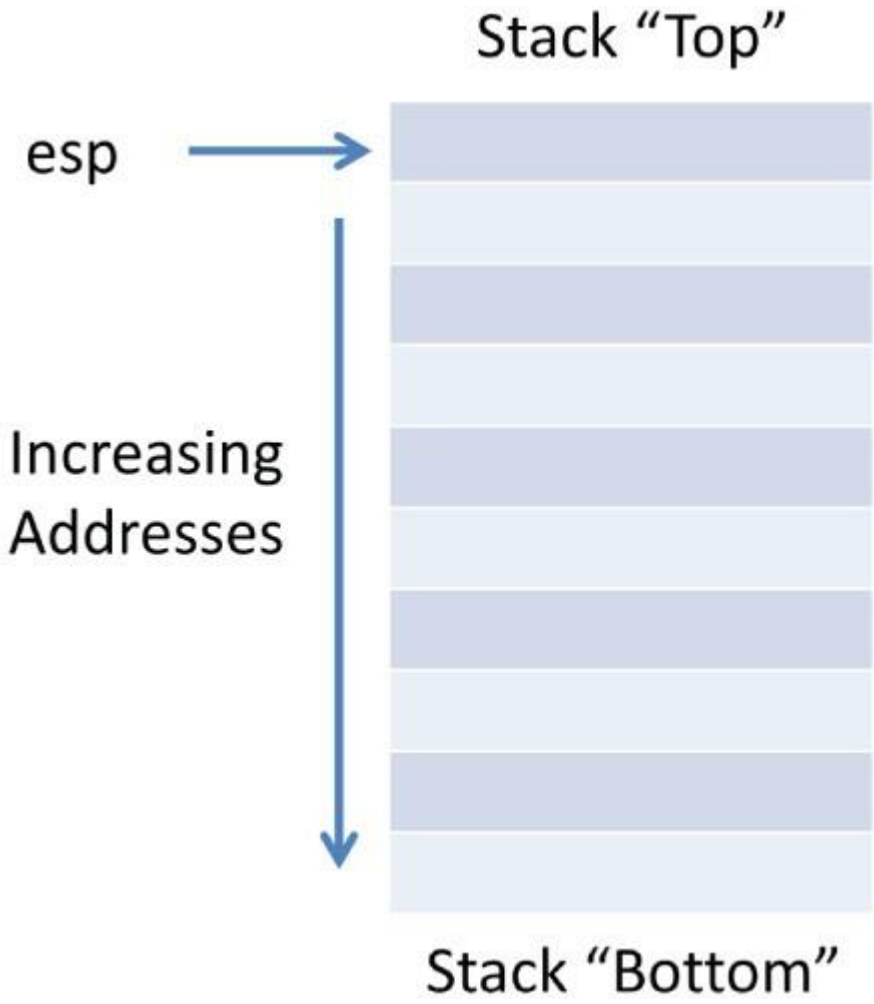


Рисунок 7.2: Увеличение адреса стека

Рассмотрим следующий фрагмент файла сборки:

```
mov esp, stack
...
section .data
times 128 db 0
stack equ $-4
```

Команда times 128 db 0 выделяет пространство для 128-байтового стека. Затем стек команд, равный $-4, определяет постоянный стек, установленный на начальное значение текущего местоположения ($) минус 4 (т.е. на 4 байта в конец стека), который содержит адрес конечного dword (4-байтового фрагмента) этого стека.

Первая инструкция в этом примере кода инициализирует указатель стека, особенно, этим постоянным значением. Затем стек может увеличиваться по мере добавления новых данных с помощью инструкций push.

push и pop
Инструкции push и pop используются для добавления и удаления данных из стека.

push
Команда push добавит 4 байта или 1 dword в начало стека. Она принимает единственный аргумент, который может быть именем регистра, адресом памяти или константой.

```
push <register>
push <memory>
push <constant>
```

Когда выполняется push, указатель стека, esp, автоматически уменьшается на 4, чтобы указать новую вершину стека. Затем значение, помещаемое в стек, помещается в это местоположение.

Таблица 7.2 иллюстрирует, как работает команда push. В этом случае указатель стека, esp, начинается со значения 0x120, как показано в левой части таблицы 7.2. Затем выполняются следующие инструкции:

```
; esp = 0x120
mov eax, 0xFEDA8712
push eax
; esp = 0x11C
```

Эти инструкции поместят в стек 1 dword или 4 байта. Для достижения этой цели указатель стека будет уменьшен, чтобы указывать на адрес 0x11C. Затем значение 0xFEDA8712 будет сохранено в стеке, как показано в таблице справа от таблицы 7.2.

Таблица 7.2: Перемещение переменной в стек

ADDRESS	VALUE		ADDRESS	VALUE
0x11B			0x11B	
0x11C		0x11C (esp)	0x12	
0x11D		0x11D	0x87	
0x11E		0x11E	0xDA	
0x11F		0x11F	0xFE	
0x120 (esp)	0x11	0x120	0x11	
0x121	0x22	0x121	0x22	
0x122	0x33	0x122	0x33	

Инструкция single push объединяет пару шагов в одну инструкцию. Следующий пример кода эквивалентен предыдущему:

```
;esp = 0x120
mov eax, 0xFEDA8712
sub esp, 4
mov [esp], eax
; esp = 0x11C
```

В этом примере значение указателя стека явно уменьшается, чтобы указывать на новую вершину стека. Затем значение, сохраненное в eax, перемещается в это местоположение. Эти два подхода взаимозаменяемы, но использование push требует меньше инструкций.

pop
В x86 команда pop является обратной инструкции push и удаляет одно dword из стека. Она имеет синтаксис pop dst, где dst может быть регистром или адресом памяти.

Команда pop отменяет операции, выполняемые push. Она начинается с перемещения значения, сохраненного в [esp], в указанный регистр или ячейку памяти. Затем она автоматически увеличивает esp на 4, чтобы указать на новую вершину стека.

Таблица 7.3 иллюстрирует, как можно использовать pop для отмены нажатия из предыдущего примера. В конце этого примера стек напоминал столбцы слева от этой таблицы.

```
;esp = 0x11C
pop eax
;esp = 0x120
;eax = 0xfeda8712
```

Таблица 7.3: Извлечение переменной из стека

ADDRESS	VALUE		ADDRESS	VALUE
0x11B			0x11B	
0x11C(esp)	0x12	0x11C		
0x11D	0x87	0x11D		
0x11E	0xDA	0x11E		
0x11F	0xFE	0x11F		
0x120	0x11	0x120(esp)	0x11	
0x121	0x22	0x121	0x22	
0x122	0x33	0x122	0x33	

Выполнение команды pop eax обновило бы стек, чтобы он напоминал изображение справа. В рамках этого процесса регистр eax был бы обновлен, чтобы содержать значение 0xFEDA8712. Тогда указатель стека, esp, был бы увеличен на 4 до значения 0x120.

Как и в случае с инструкцией push, pop объединяет двухэтапный процесс в одну инструкцию. Следующий код эквивалентен, но явно выполняет каждый шаг:

```
;esp = 0x11C
mov eax, [esp]
add esp, 4
;esp = 0x120
;eax = 0xfeda8712
```

Стек в виде блокнота
x86 имеет ограниченное количество регистров, и его легко исчерпать. Часто для временного хранения информации, которую вы еще не закончили использовать, требуется своего рода область подкачки/зачистки. Стек обеспечивает удобное расположение для временного хранения значений.

Например, рассмотрим следующий код:

```
mov eax, 0xcafed00d
```

```
mov ebx, 0x00c0ffee
add eax, ebx
push eax      ; save to free up eax
...           ; do other things
pop eax       ; retrieve saved eax
```

В этом примере инструкции push и pop используются для временного сохранения содержимого eax в стеке. Это освобождает регистр для использования в других вычислениях. Когда сохраненное значение понадобится снова, можно использовать pop для возврата его в eax .

Осторожно используйте pop

Данные на компьютере удаляются редко. Например, когда файл удаляется в файловой системе или программа освобождает переменную, связанная с ним память просто помечается как доступная для других целей. Сохраненные данные по-прежнему присутствуют на диске.

То же самое верно и для стека в x86. Когда значение извлекается из стека, оно копируется в регистр или ячейку памяти, но значение остается в стеке. После настройки указателя стека извлекаемое значение находится за пределами допустимого диапазона стека.

После ввода значения оно должно считаться освобожденным и более небезопасным для использования. Любая попытка доступа к данным за пределами допустимого диапазона стека опасна. Например, ни одна законная инструкция по сборке не должна включать [esp-...].

Рассмотрим следующий пример, в котором показаны трассировки стека из различных местоположений; соответствующая трассировка стека указана в комментариях (например, ; (1)). В каждом местоположении комментария отображается стек после выполнения этой строки, как показано в таблице 7.4.

```
;(1) esp = 0x10C
push 0xbadc0de ; (2)
pop eax  ;(3)  eax = 0xbadc0de
push 0xc0ffee ;(4)
```

Таблица 7.4: Примеры трассировки стека

(1)		(2)		
ADDRESS	**VALUE**	**ADDRESS**	**VALUE**	
0x1000	??		0x1000	??
0x1004	??	0x1004	??	
0x1008	??	0x1008 (esp)	0xbadc0de	
0x100c (esp)	0x11223344	0x100c	0x11223344	
0x1010	0x55667788	0x1010	0x55667788	
(3)	(4)			
ADDRESS	VALUE	ADDRESS	VALUE	
0x1000	??	0x1000	??	
0x1004	??	0x1004	??	

(1)		(2)	
ADDRESS	**VALUE**	**ADDRESS**	**VALUE**
0x1008	0xbadc0de	0x1008 (esp)	0xc0ffee
0x100c (esp)	0x11223344	0x100c	0x11223344
0x1010	0x55667788	0x1010	0x55667788

Обратите внимание в stack trace 3, что 0xbadc0de был удален из стека, но он все еще там, пока не появится что-то другое, чтобы перезаписать его, как показано в stack state 4. Опять же, при законных условиях вы не хотите полагаться / использовать что-либо выше выделенного в данный момент стека ([особенно-..]), но эти знания на самом деле могут быть полезными для других, менее законных целей. В последующих примерах либо больше не будет отображаться содержимое, которое является “нераспределенным” в стеке, либо будет вычеркнуто (пример) значение, чтобы указать, что оно нераспределено.

Вызовы функций и фреймы стека
Языки программирования более высокого уровня имеют концепцию функций, которые представляют собой фрагменты кода, которые могут быть вызваны из других функций. в x86 также есть концепция функций.

При вызове функции вносятся изменения в состояние стека. Понимание этих изменений важно для понимания того, как работает приложение.

Функции в x86
инструкции call и ret в x86 предоставляют возможность создавать функции, аналогичные языкам программирования более высокого уровня.

call
Команда вызова имеет синтаксис call op, где op указывает адрес вызываемой функции. Аргументом op может быть регистр, метка или адрес памяти.

call eax ; branch to eax
call label ; branch to label
call 0x1000 ; branch to 0x1000
Подобно push и pop, вызов фактически объединяет несколько шагов в одну операцию. Сначала он создает обратный адрес, помещая адрес следующей инструкции в стек. Затем он выполняет безусловный переход к местоположению кода, указанному op .

ret
Инструкция ret не принимает аргументов. Ее цель - вернуть выполнение вызывающей функции.

Это достигается двухэтапным процессом. Сначала ret извлекает из стека адрес возврата, сохраненный при вызове. Затем он выполняет безусловный переход к этому адресу.

Как работают функции x86

С помощью call и ret можно создавать функции непосредственно в x86 или переводить их с других языков Рассмотрим следующий код:

```
void a() {
```

```
}

void b() {
    a();
}
```

Этот код определяет две функции, a и b, где b вызывает a. Этот код эквивалентен следующему в x86:

```
b:
    call a
    ret

a:
    ret
```

Таблица 7.5 иллюстрирует, как выполнение этого кода повлияет на стек. Предположим, что эти инструкции хранятся в следующих местах памяти с соответствующими трассировками стека, как указано в комментариях. Напомним, что стек показывает состояние после выполнения этой инструкции.

```
b:
0x10000 call a  ;(1)
0x10003 ret     ;(2)

a:
0x20012 ret     ;(3)
```

Крайнее левое изображение в таблице 7.5 показывает начальное состояние стека. В этот момент esp указывает на адрес 0x9010, а eip имеет значение 0x1000.

Среднее изображение иллюстрирует, что происходит, когда выполняется вызов a. На этом этапе значение eip, которое теперь равно 0x10003, помещается в стек. Теперь eip указывает на первую строку кода в a, которая имеет адрес 0x20012.

Как только a возвращается, исходное значение eip извлекается из стека, в результате чего оно указывает на инструкцию ret в bat 0x10003. Указатель стека также обновляется, чтобы указывать на адрес 0x9010. Обратите внимание, что значение 0x10003 остается в памяти, но теперь находится за пределами стека и его больше не следует использовать или доверять ему.

Объединяя это понятие стека и функций, взгляните на пример. Рассмотрим следующий набор определений функций:

```
void a() { int x; b(); }
void b() { int x; c(); }
void c() { int x; }
```

Таблица 7.5: Вызовы функций и стек

(1)		(2)		(3)			
ADDRESS	**VALUE**	**ADDRESS**	**VALUE**	**ADDRESS**	**VALUE**		
0x9000	??		0x9000	??		0x9000	??
0x9004	??	0x9004	??	0x9004	??		
0x9008	??	0x9008	??	0x9008	??		
0x900c	??	0x900c	0x10003	0x900c	0x10003		
0x9010	??	0x9010	??	0x9010	??		
REGISTER	VALUE	REGISTER	VALUE	REGISTER	VALUE		
esp	0x9010	esp	0x900c	esp	0x9010		
eip	0x10000	eip	0x20012	eip	0x10003		

Это начинается с функции a(), которая имеет единственную локальную переменную x и вызывает функцию b. Таблица 7.6 иллюстрирует структуру стека после вызова a. Обратите внимание, что по мере того, как локальным данным выделяется пространство, они добавляются в стек.

Таблица 7.6: Стек программы после вызова

STACK
a's local data

Когда выполняется a, он объявляет x и затем вызывает b. Это означает, что поток выполнения переключится на выполнение кода, содержащегося в b, прежде чем вернуться в a.

Чтобы гарантировать, что он вернется в правильное местоположение в a, процессор сохраняет адрес возврата в стеке. Этот адрес возврата является адресом инструкции, следующей за вызовом b в a.

После того, как адрес возврата помещен в стек, процессор сохраняет там локальные данные b. В таблице 7.7 показано состояние стека до того, как процессор выполнит первую инструкцию в b.

Таблица 7.7: Стек программы после вызова b

STACK

b'local data
a's return address
a's local data

Как и a, b объявит свои локальные переменные, а затем выполнит вызов другой функции, c. При выполнении этого вызова адрес возврата для b будет помещен в стек так же, как и локальные переменные вызываемой функции. Как только эта настройка для c будет завершена, стек будет похож на таблицу 7.8.

Таблица 7.8: Стек программы после вызова c

STACK
c' local data
b'sreturn address
b's local data
a's return address
a's local data

В c объявляется локальная переменная x, а затем функция завершается. Когда процессор завершает работу в c, поток кода должен вернуться к вызывающей функции b.

На этом этапе локальные данные c находятся в верхней части стека, но больше не нужны. Процессор может извлечь эти данные из стека, изменив указатель стека таким образом, чтобы он указывал обратный адрес b.

Затем процессор может извлечь этот обратный адрес из стека, сохранив его в eip и обновив указатель стека с помощью команды ret. Это позволяет программе вернуться к b и возобновить выполнение любого кода, следующего за ее вызовом в c. На этом этапе стек возвращается в состояние, показанное в таблице 7.7.

Вызов c является последней инструкцией в b, поэтому он также будет немедленно возвращен. Как и возврат из c, это включает в себя удаление локальных переменных из стека и обновление eip путем извлечения адреса возврата a из стека в eip (через ret). Как только это будет завершено, стек будет похож на таблицу 7.6.

После этого возврата из b также вернется a. локальные данные a будут извлечены из стека. Затем регистр eip будет обновлен на основе обратного адреса вызывающей функции, который произошел до нашего анализа и не показан в таблице 7.6, и выполнение возобновится в рамках этой функции.

Анализ стека
Поскольку функции вызываются и возвращаются из вызовов, они оказывают влияние на

программный стек. Например, рассмотрим следующий код:

```
void a() { }
void b() { }
void c() { a(); b(); }
```

Этот код определяет три функции, a, b и c, и c вызывает обе другие две.

Когда функция запущена или находится в стеке вызовов выполняемой функции, ее адрес возврата находится в стеке. Например, когда выполняется a, адреса возврата для a и c находятся в стеке. Аналогично, когда выполняется b, адреса возврата для b и c находятся в стеке.

Проверка адресов возврата, хранящихся в стеке, позволяет увидеть, как была достигнута определенная точка в программе. Каждая функция в стеке вызовов будет иметь свой локальный адрес возврата, переменные и начальные данные, видимые в стеке.

Эта практика называется раскручиванием стека. В gdb команда info stack покажет текущее состояние стека.

Соглашения о вызовах
Инструкции call и ret позволяют создавать функции в assembly. Однако, имея только call и ret, эти функции должны быть автономными без возможности передачи данных между функциями.

В языках программирования более высокого уровня функции обычно имеют параметры или аргументы, которые являются переменными, передаваемыми функции вызывающей функцией. Однако в машинном коде нет понятия параметров; есть только регистры и память.

в x86 есть все инструменты, необходимые для создания параметров. Языки программирования более высокого уровня, использующие параметры, переводятся на ассемблер. Программист или компилятор несет ответственность за выбор способа использования этих инструментов.

Почему необходимы соглашения
С помощью регистров, стека и даже ячеек памяти x86 имеет возможность передавать значения из одной функции в другую. Параметры могут храниться в регистрах или передаваться и извлекаться из стека.

Однако связь или соглашения между функциями необходимы, если они планируют использовать параметры, регистры или стек согласованным образом. Если вызывающая функция использует определенные местоположения для передачи параметров, вызываемый объект должен знать, какие местоположения используются для каких значений. То же самое верно, если вызываемый объект возвращает данные вызывающему абоненту.

void caller() { ... вызываемый объект()} //определение номенклатуры
Кроме того, если вызывающий объект использует регистр для хранения своих внутренних значений, вызываемый объект должен знать, что не следует перезаписывать эти значения. Это особенно важно, если вызываемый объект использует такие операции, как mul, которые изменяют регистры, но которые легко пропустить.

В рамках небольшой программы разработчик мог бы спроектировать свой код таким образом, чтобы он содержал эти знания. Если функция a принимает три параметра, разработчик мог бы создать схему, которая передает их через регистры или стек. Аналогично, структура, необходимая функции b, может быть передана путем выделения определенного места в памяти.

Однако, хотя этот подход может работать в небольших масштабах, он не поддается масштабированию и подвержен ошибкам. Оплошность может привести к случайному удалению важных данных в результате операции mul. Кроме того, такая схема ad hoc затрудняет работу с командами разработчиков.

Введение в соглашения о вызовах
Соглашения о вызовах предназначены для упрощения передачи данных между функциями путем определения правил взаимодействия между функциями. Они являются частью двоичного интерфейса приложения (ABI), который является определением самого низкого уровня взаимодействия фрагментов кода.

Соглашение о вызове должно определять несколько правил, включая следующие:

- Расположение параметров: Где параметры будут передаваться от вызывающего объекта вызываемому объекту (стек или регистры)?
- Порядок параметров: Как будут организованы параметры, в стеке или в регистрах?
- Очистка стека: Если используется стек, какая функция отвечает за удаление значений из стека (очистка вызывающей стороны или вызываемого объекта)?
- Доступ к регистрам: Какие регистры может использовать вызываемый объект без необходимости создавать резервные копии своих предыдущих значений и восстанавливать их перед возвратом?
- Возвращаемые значения: Где и как будут возвращены значения от вызываемого объекта вызывающей стороне?
Соглашения о вызовах могут варьироваться в зависимости от различных факторов, включая следующие:

- Архитектура (x86 или ARM)
- Операционная система (UN*X или Windows)
- Язык программирования (C или Java)
- Даже компилятор (GCC против Microsoft)
На заре программирования стандартов на самом деле не существовало. В результате программы не могли работать вместе, если их разработчики не договаривались о соглашении о вызовах.

Изначально существовало множество различных компаний, каждая со своими собственными соглашениями. Со временем они были сведены к нескольким популярным стандартам, включая следующие:

- cdecl
- syscall
- optlink
- pascall
- register
- stdcall
- fastcall
- safecall

- thiscall
- cdecl

cdecl (“смотрите deckel”) - сокращение от “C declaration” и является одним из наиболее распространенных соглашений о вызовах в архитектуре x86. Хотя cdecl появился на C, он используется для множества различных языков программирования и архитектур. Это также полезный стандарт при написании ассемблера вручную.

cdecl определяет следующие правила:

- Параметры на основе стека: Аргументы помещаются справа налево в стек для передачи вызываемому объекту.
- Очистка вызывающего объекта: Вызывающая функция отвечает за удаление аргументов из стека после возврата вызываемого объекта.
- Возвращаемое значение: Регистр eax используется для хранения возвращаемого значения функции.
- Доступные регистры: Вызываемый объект может свободно изменять eax, ecx и edx . Вызывающий объект должен сохранить все необходимые значения в этих регистрах перед выполнением вызова. Вызываемый объект должен сохранить значения других регистров перед их использованием и восстановить их перед возвратом.

Рассмотрим инструкцию int s = add(1,2). Используя стандарт cdecl, это привело бы к следующему ассемблерному коду x86:

```
; Save regs we need to keep according to cdecl.
; Optional if we don't intend to modify these registers.
push    eax
push    ecx
push    edx
; Push parameters from right to left. The original
; code was add(1,2), so left to right is 2, then 1
push    2
push    1

; Call add.
call    add

; Remove parameters from the stack. We pushed 2x 4-byte values
; we can either do 2 pops, or add 8 back to the stack
add esp, 8

; Save the return value into eax (where cdecl says return values
go)
mov [s], eax

; Restore the saved registers, remember its last
; in first out, so we pushed edx last, meaning it is the first to
pop
pop edx
pop ecx
pop eax
```

Сохранение регистров

В cdecl функции могут свободно изменять eax, ecx и edx без сохранения их значений. Следовательно, следующая функция, f, действительна в соответствии со стандартом.

```
f:  mov ecx, 0xd15ea5e
    mov edx, 0xfee1dead
    lea eax, [ecx + edx]
    ret
```

Однако значения любых других регистров, используемых вызываемым пользователем, должны быть сохранены до их изменения, а исходные данные восстановлены перед возвратом.

```
f:  push   ebx
    push   ebp
    push   esi
    mov    ebp, 0xd15ea5e
    mov    ebx, 0xfee1dead
    lea    esi, [ebp + ebx]
    pop    esi
    pop    ebp
    pop    ebx
    ret
```

Эта функция использует ebx, ebp и esi, поэтому она помещает их предыдущие значения в стек перед использованием регистров и помещает эти значения обратно в регистры перед возвратом.

С помощью cdecl вызывающая функция знает, значениям каких регистров она может доверять после вызова другой функции. Вызываемому объекту разрешено изменять значения eax , ecx и edx по своему желанию, поэтому вызывающий объект должен сохранить значения этих регистров, если он захочет использовать их позже. Однако вызываемый объект должен сохранять значения всех остальных регистров, поэтому нет необходимости сохранять их перед выполнением вызова.

Например, рассмотрим следующий блок кода:

```
g:
    mov   ebx, 0xd15ea5e
    mov   ecx, 0xfee1dead
    call  f
```

После вызова f функция g может полагаться на то, что ebx сохранит значение 0xd15ea5e. Однако она не может предполагать, что ecx по-прежнему будет иметь значение 0xfee1dead.

Возвращаемые значения

В языках программирования более высокого уровня функции обычно используют возвращаемые значения для передачи информации своим вызывающим объектам. Например, функция может быть спроектирована так, чтобы возвращать 0 при успешном завершении или код ошибки, если что-то пошло не так. Например, следующая функция возвращает значение 1 по завершении:

```
int f()
{
    return 1;
}
```

При использовании соглашения о вызове cdecl это возвращаемое значение сохраняется в регистре eax. Следующий код x86 эквивалентен предыдущей функции f:

```
f:
    mov  eax, 1
    ret
```

Функции могут быть разных типов и иметь разные возвращаемые значения, соответствующие этим типам. Например, следующая функция предназначена для возврата указателя char*, а по умолчанию используется нулевой указатель:

```
char* f()
{
    return NULL;
}
```

В x86 регистр может использоваться в качестве указателя. Следующий код x86 использует значение 0 для представления эквивалента нулевого указателя; eax также может использоваться для указания местоположения массива символов в памяти.

```
f:
    mov  eax, 0
    ret
```

Доступ к параметрам

Стандарт cdecl использует стек для передачи параметров функции. Некоторые вещи, которые следует иметь в виду при попытке доступа к параметрам, включают следующее:

- Вершина стека (последнее переданное значение) - [esp].
- Стек растет вниз (в направлении более низких адресов).
- Команда вызова помещает адрес возврата в стек.
- Вызывающий объект перемещает аргументы справа налево.
- Возвращаемое значение вызываемого объекта должно храниться в eax .

Принимая во внимание эти факторы, рассмотрим, как вызов следующей функции был бы реализован в x86:

```
int add (int x, int y)
{
        return x+y;
}
```

В x86 эквивалентом этого было бы следующее:

```
f:  push  1   ; y
    push  2   ; x
    call  add
    mov   [s], eax  ;save the return value to memory
    pop   eax
    pop   eax
    ret
```

```
; int add(int x, int y) { return x+y; }
add:
    mov eax, [esp+4]    ; retrieve x from stack
    mov edx, [esp+8]    ; retrieve y from stack
    add eax, edx
    ret
```

Когда вызывается функция, это влияет на текущий программный стек. В таблице 7.9 показано состояние стека в функции добавления.

Таблица 7.9: Стек в функции добавления

ADDRESS	VALUE
0xeff0	
0xeff4	
0xeff8 (esp)	(Return address)
0xeffc	2
0xf000	1

Хотя доступ к параметрам возможен из [esp], при таком подходе могут возникнуть проблемы. Рассмотрим, как следующие инструкции в add влияют на значение esp:

```
; int f(int x);
f:
    mov  eax, [esp+4]    ; x is at [esp+4]
    push ebx             ; save ebx
    mov  ebx, [esp+8]    ; x is now at [esp+8]
    ...
```

Поскольку вызываемый объект извлекает параметры из стека, местоположение вершины стека изменяется. В результате расположение параметров относительно esp также изменяется.

Кадры стека

Значение esp изменяется слишком часто, чтобы быть полезной системой отсчета для расположения переменных в стеке. Каждый раз, когда значение передается или извлекается из стека, значение esp и относительное расположение других переменных стека меняются.

Именно здесь вступает в действие другой регистр стека ebp (также известный как базовый указатель или указатель фрейма). Регистр ebp указывает на нижнюю часть текущего фрейма стека, который является нижней частью раздела памяти в стеке, используемого конкретной функцией.

Прологи и эпилоги

функции x86 обычно начинаются и заканчиваются фрагментами шаблонного кода. Целью этого кода является настройка и удаление фрейма стека функции.

Настройка фрейма стека

Пролог функции или преамбула устанавливает фрейм стека и находится в самом начале функции. Пролог выполняет две функции.

- Сохраните предыдущий фрейм стека с помощью push ebp.
- Установите новый фрейм стека с помощью mov ebp, esp.

Эти инструкции отображаются в самом начале функции. В таблице 7.10 показано влияние каждой из них на стек.

```
;(1)
push ebp ;(2)
move ebp, esp ; (3)
push 0x11223344 ;(4)
```

Таблица 7.10: Влияние пролога функции на стек

(1)		(2)		
ADDRESS	**VALUE**	**ADDRESS**	**VALUE**	
0xefe8	??		0xefe8	??
0xeff0	??	0xeff0	??	
0xeff4	??	0xeff4 (esp)	Old ebp	
0xeff8 (esp)	(return address)	0xeff8	(return address)	
0xeffc	2	0xeffc	2	
0xf000	1	0xf000	1	
(3)	(4)			
ADDRESS	VALUE	ADDRESS	VALUE	
0xefe8	??	0xefe8	??	
0xeff0	??	0xeff0 (esp)	0x11223344	
0xeff4 (esp, ebp)	Old ebp	0xeff4 (ebp)	Old ebp	
0xeff8	(return address)	0xeff8	(return address)	
0xeffc	2	0xeffc	2	
0xf000	1	0xf000	1	

В первой таблице показан стек перед запуском функции. На этом этапе параметры функции помещаются в стек (справа налево), а также обратный адрес вызывающей стороны.

Когда выполняется команда push ebp, базовый указатель предыдущей функции сохраняется в стеке. Результирующий стек показан во второй таблице.

В третьей таблице показан стек после выполнения инструкции mov ebp, esp. Хотя сам стек не обновляется, новый ebp указывает на обратный адрес вызывающей функции, такой же, как esp.

После выполнения этих инструкций вызываемый объект может поместить локальные переменные в стек. Хотя это изменит значение esp, значение ebp останется постоянным (таблица четвертая). Это позволяет получить доступ к параметрам и локальным переменным относительно фиксированной точки, ebp, а не к более изменяемой esp.

Удаление фрейма стека

Создание нового фрейма стека для текущей функции означает, что вы потеряли фрейм стека вызывающей функции. Прежде чем функция вернется, ей необходимо отменить внесенные ею изменения и восстановить фрейм стека вызывающей стороны.

Эпилог функции появляется в конце функции и завершает этот процесс. Он состоит из следующих трех инструкций:

```
mov esp, ebp
pop ebp
ret
```

Первым шагом этого процесса является удаление любых данных, которые были добавлены в стек. Поскольку данные фактически не удаляются из памяти, это просто включает изменение указателя стека с помощью команды mov esp, ebp. Эта операция восстановила бы состояние стека, как показано в таблице 7.11.

```
; function body (1)
mov esp, ebp ;(2)
pop ebp ; (3)
```

Далее значение базового указателя должно быть восстановлено до значения вызывающей функции. Напомним, что в преамбуле это было помещено в стек, поэтому это выполняется с помощью инструкции pop ebp, которая восстанавливает стек в исходное состояние. На данный момент стек находится в надлежащем состоянии для возврата к вызывающей функции.

Хотя это можно выполнить с помощью этих двух инструкций, x86 предлагает альтернативный вариант. Команда leave эквивалентна следующим двум инструкциям:

```
mov esp, ebp
pop ebp
```

Таблица 7.11: Влияние эпилога функции на стек

(1)		(2)		
ADDRESS	**VALUE**	**ADDRESS**	**VALUE**	
0xefe8 (esp)	0x3325d321		0xefe8	0x3325d321
0xeff0	0x11223344	0xeff0	0x11223344	
0xeff4 (ebp)	Old ebp	0xeff4 (ebp, esp)	Old ebp	
0xeff8	(return address)	0xeff8	(return address)	
0xeffc	2	0xeffc	2	
0xf000	1	0xf000	1	

(3)	
ADDRESS	VALUE
0xefe8	0x3325d321
0xeff0	0x11223344

(1)		(2)	
ADDRESS	**VALUE**	**ADDRESS**	**VALUE**
0xeff4 (esp)	Old ebp		
0xeff8	(return address)		
0xeffc	2		
0xf000	1		

Доступ к параметрам

Фреймы стека предназначены для упрощения доступа к параметрам и другим значениям в стеке из функции. Использование статического ebp в качестве ссылки упрощает процесс определения того, где находится конкретное значение в стеке. Например, в таблице 7.12 показано расположение определенных значений в стеке для любой функции, использующей соглашение cdecl. Поскольку ebp не перемещается во время выполнения функции, эти соотношения и смещения всегда будут одинаковыми. Это означает, что если вызывающий объект передал переменную, то первая всегда будет иметь значение ebp+8, вторая всегда будет иметь значение ebp+12 и т. д.

Таблица 7.12: Расположение стека для общих значений

LOCATION	VALUE
[ebp + 0]	Previous frame pointer
[ebp + 4]	Function return address
[ebp + 8]	First parameter
[ebp + 12]	Second parameter
[ebp + 16]	Third parameter
...	...

В следующем примере показано построение и использование параметров функции. Как и во всех этих примерах стека, помните, что каждый стек отображается после выполнения инструкции.

```
f:
    push  1   ; y
    push  2   ; x   ;(1)
    call  add
    mov   [s], eax
    pop   eax
    pop   eax
    ret

; int add(int x, int y) { return x+y; }
add:
       ; (2)
    push ebp
    mov  ebp, esp   ;(3)
    mov  eax, [ebp+8]     ; retrieve x from stack
```

```
    mov  edx, [ebp+12]    ; retrieve y from stack
    add  eax, edx
    mov  esp, ebp
    pop  ebp
    ret
```

В таблице 7.13 показано содержимое стека для пунктов (1), (2) и (3) в предыдущем коде.

Таблица 7.13: Содержимое стека в точках 1, 2 и 3 программы

(1)		(2)		(3)			
ADDRESS	VALUE	ADDRESS	VALUE	ADDRESS	VALUE		
0xeff4	??		0xeff4			0xeff4 (esp, ebp)	Old ebp
0xeff8	??	0xeff8 (esp)	return address	0xeff8	return address		
0xeffc (esp)	2	0xeffc	2	0xeffc	2		
0xf000	1	0xf000	1	0xf000	1		

Как только будет достигнуто местоположение 3, основываясь на ваших знаниях о cdecl и фрейме стека, вы с уверенностью знаете, что первый параметр, x, будет находиться в местоположении ebp+8 и иметь значение 2. Второй параметр, y, будет находиться в местоположении ebp+12 и иметь значение 1.

Локальные переменные

Локальные переменные функции хранятся в стеке над указателем предыдущего фрейма (по более низким адресам). После настройки фрейма стека можно выделить место для локальных переменных, просто вычитая требуемый объем пространства из esp . Это распределение будет автоматически отменено в эпилоге функции, когда указатель стека будет сброшен на основе базового указателя.

Например, рассмотрим следующую функцию:

```
void one_up(int x)
{
       int y = x + 1;
}
```

В дополнение к входящему аргументу x, она также определяет локальную переменную y, которая будет сохранена в стеке. Следующий код показывает, как эта функция будет выглядеть после перевода на x86:

```
one_up:
    push ebp
    mov  ebp, esp
    sub  esp, 4         ; allocate space for local y (4 bytes)
    mov  eax, [ebp+8]   ; load parameter x
    inc  eax            ; x + 1
    mov  [ebp-4], eax   ; save local y
      ;stack shown here
```

```
mov  esp, ebp
pop  ebp
ret
```

Кадр стека этой программы показан в таблице 7.14. Обратите внимание, что, хотя esp теперь указывает на 0xeff4, значение ebp остается прежним (указывая на сохраненный ebp вызывающей программы) после размещения локальных переменных в стеке. Как параметры, так и локальные переменные могут быть легко доступны относительно ebp.

Таблица 7.14: Кадр стека программы one_up

ADDRESS	VALUE
0xeff4 (esp)	y
0xeff8 (ebp)	Old ebp
0xeffc	Ret address
0xf000	x

Как и в случае с параметрами, cdecl гарантирует, что локальные переменные хранятся в согласованных местоположениях в различных функциях. В таблице 7.15 показано расположение локальных переменных относительно ebp. При обратном проектировании знание ebp становится невероятно мощным.

Совет

Понимание того, что ebp минус что-либо ссылается на локальную переменную, что-то выделенное внутри функции, в то время как доступ к ebp плюс что-либо - это доступ к информации, предоставляемой функции вызывающим пользователем, может помочь вам быстро обнаружить интересные фрагменты кода или определить критическую функциональность, которой можно манипулировать, скажем, путем ввода в программу.

Таблица 7.15: Расположение стека для локальных переменных

LOCATION	VALUE
[ebp - 4]	First local variable
[ebp - 8]	Second local variable
[ebp - 12]	Third local variable
...	...

Кратчайший путь

Можно по отдельности поместить каждый параметр или локальную переменную в стек; каждое нажатие обновляет указатель стека и перемещает значение на место.

Вместо этого обычный маршрут, который используют компиляторы, заключается в выделении пространства для всех сразу, прежде чем перемещать значения на место. Например, следующие инструкции реже встречаются в скомпилированном коде:

```
push 1
push 2
```

Вместо этого эти инструкции, скорее всего, будут сведены к следующему:

```
sub esp, 8     ; allocate 8 bytes on the stack
mov dword [esp+4], 1 ; put 1 on the stack
mov dword [esp], 2   ; put 2 on the stack
```

Выравнивание стека

Некоторые компиляторы обеспечивают выравнивание стека по 32 байтам при вводе функции. Это означает, что адрес памяти указателя стека должен быть равномерно разделен на 32. Исторически для систем было более эффективно извлекать память по границам, выровненным по 32 байтам. Этого повышения эффективности, возможно, все еще нет, но вы все равно увидите, что компиляторы время от времени что-то делают для поддержания выравнивания стека. Как вы можете видеть в этом манифесте, при выделении места для локальных переменных они могут выделять избыточное пространство для поддержания этого выравнивания.

Это означает, что неиспользуемое пространство обычно существует во фрейме стека функции. При реверсировании не зацикливайтесь на избыточном пространстве, потому что это совершенно нормально. При написании собственного кода это не то, что вам нужно делать вручную, но цель этой книги - научить вас распознавать это в коде и понимать, что это то, что вы в основном можете игнорировать.

Общая картина

Когда вызывается функция, она вносит несколько изменений в программный стек. Чтобы увидеть все эти изменения в одном месте, воспользуйтесь следующей программой:

```
void hack(...)
{
    ...
}

void drink(...)
{
    ...
    hack(...);
    ...
}
```

Каждая из двух функций в этой программе может иметь ноль или более параметров и локальных переменных. На рисунке 7.3 показана структура стековых фреймов для каждой функции.

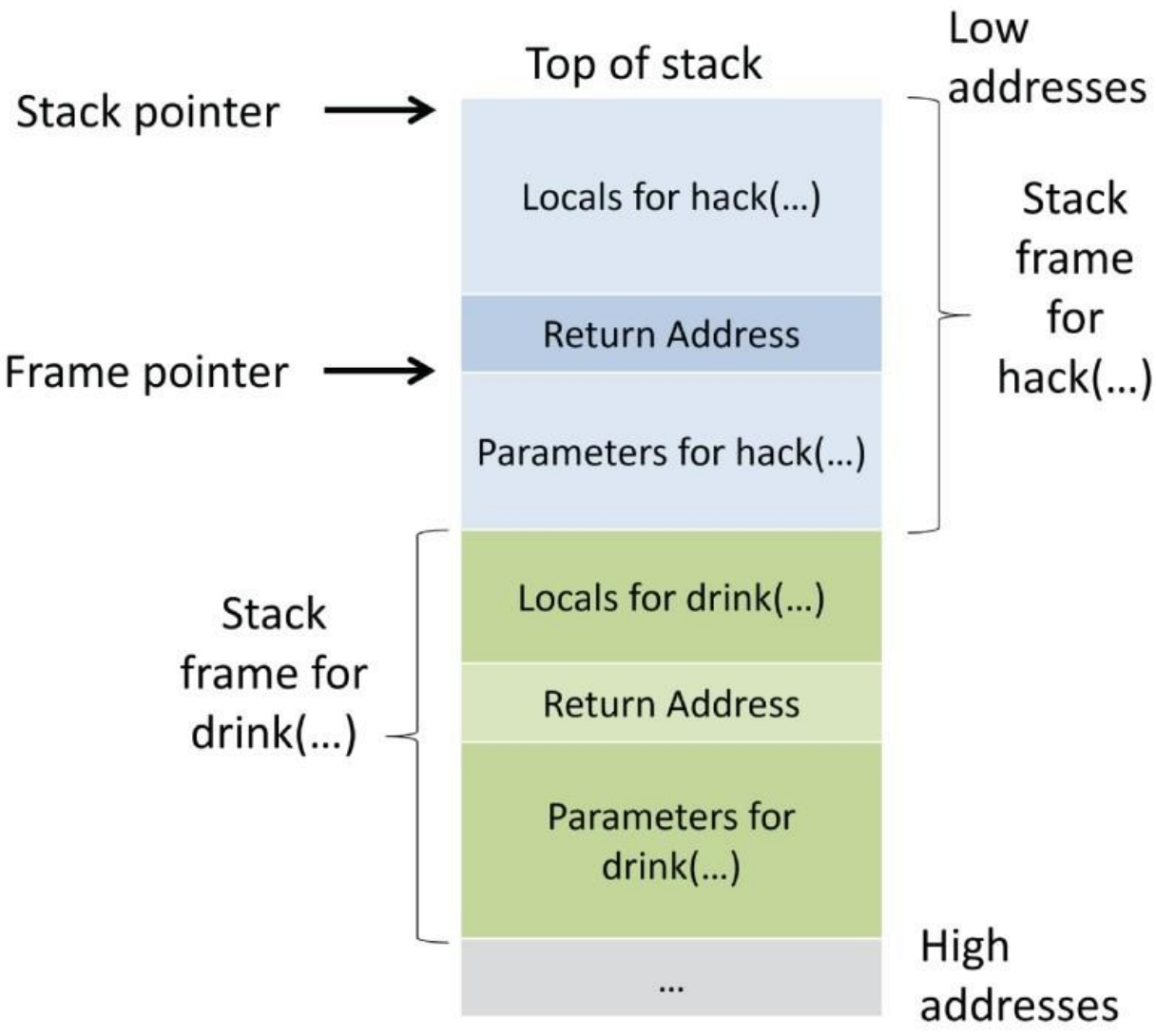


Рисунок 7.3: Стековые фреймы для функций hack и drink

Что нужно запомнить
ассемблерные программы x86 могут быть сложными. Чтобы быть эффективными в реверс-инжиниринге x86, жизненно важно запомнить определенные вещи.

Первая из них - структура стекового фрейма функции. В таблице 7.16 показан полный фрейм стека, включая параметры, адреса возврата, локальные переменные и начальное пространство.

Таблица 7.16: Полный фрейм стека функций

STACK	
	...
[ebp—12] or [ebp-0xC]	Third local variable
[ebp-8]	Second local variable
[ebp-4]	First local variable

STACK	
	...
[ebp]	Previous frame pointer
[ebp+4]	Function return address
[ebp+8]	First parameter
[ebp+12] or [ebp+0xC]	Second parameter
[ebp+16] or [ebp+0xF]	Third parameter
	...

Еще одна важная вещь, которую следует запомнить, - это разница между шаблонными и полными функциональными прологами и эпилогами. В таблице 7.17 показано, чем шаблонный пролог отличается от того, который включает выделение стека для локальных переменных.

Таблица 7.17: Два типа прологов

BOILERPLATE PROLOGUE	COMPLETE PROLOGUE

push ebp ; save stack frame
mov ebp, esp ; start new frame
	push ebp ; save stack frame
	mov ebp, esp ; start new frame
	
	sub esp, 20 ; allocate 5 4 byte locals
	
	push ebx ; save modified regs
	push esi
	(etc)

Таблица 7.18: Два типа послесловий

BOILERPLATE EPILOGUE	COMPLETE EPILOGUE
mov esp, ebp ; discard locals pop ebp ; restore frame ret ; return	(etc) pop esi ; restore modified regs pop ebx mov esp, ebp ; discard locals pop ebp ; restore frame ret ; return

Функциональный эпилог отменяет действие функционального пролога. В таблице 7.18 показаны эквивалентные эпилоги для каждого из этих прологов.

Резюме

В этой главе рассматриваются жизненно важные концепции для реверсирования и взлома приложений. Прежде чем двигаться дальше, убедитесь, что у вас есть четкое представление о том, как поток управления может работать в приложениях, а также о входах и выходах функций и их фреймворков стека.

Глава 8
Компиляторы и оптимизаторы

Для многих языков программирования более высокого уровня компиляция является жизненно важной частью процесса преобразования приложения из исходного кода в машиночитаемый двоичный код. Во время этого процесса компилятор может вносить незначительные изменения в код, чтобы сделать его максимально быстрым и эффективным.

Процесс компиляции и оптимизации приложения может затруднить его обратное проектирование. В этой главе описывается, как найти отправную точку для реверсирования приложения, и некоторые из распространенных действий, выполняемых компиляторами, которые могут усложнить реверс-инжиниринг.

Поиск исходного кода

Когда код компилируется, компилятор вводит большое количество шаблонных данных, которые выполняются до того, как будет вызван фактический код приложения. При реверс-инжиниринге одна из форм искусства, которой вам нужно овладеть, заключается в том, как пропустить это и сосредоточиться на целевом коде, а не на шаблонной настройке. Однако определение точки входа в целевой код может быть сложным.

При реверсировании чужого кода маловероятно, что код будет скомпилирован с использованием отладочных символов. Это означает, что имена функций и переменных и другая информация, которая могла бы дать подсказку относительно фактической точки входа в код, были удалены из приложения. На рисунке 8.1 показано, как выглядит открытие файла без отладочных символов в gdb.

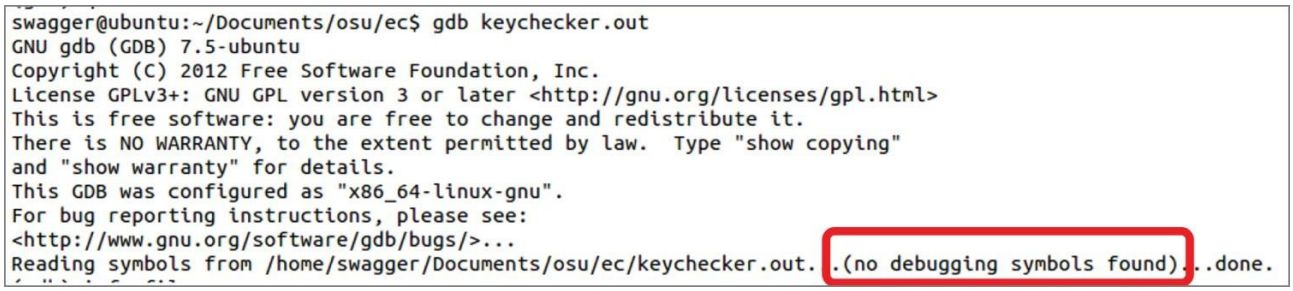


Рисунок 8.1: Приложение без символов отладки в gdb

Отсутствие символов отладки создает серьезную проблему, поскольку приложения, написанные на более высоких языках, а не на чистом ассемблере, содержат гораздо больше служебных данных и символов, генерируемых компилятором. Ниже приведен пример вывода команды info files в gdb, показывающий количество различных разделов, существующих в простом исполняемом файле:

```
Entry point: 0x80483a0
0x08048154 - 0x08048167 is .interp
0x08048168 - 0x08048188 is .note.ABI-tag
0x08048188 - 0x080481ac is .note.gnu.build-id
0x080481ac - 0x080481cc is .gnu.hash
0x080481cc - 0x0804823c is .dynsym
0x0804823c - 0x080482a6 is .dynstr
0x080482a6 - 0x080482b4 is .gnu.version
0x080482b4 - 0x080482e4 is .gnu.version_r
0x080482e4 - 0x080482ec is .rel.dyn
0x080482ec - 0x08048314 is .rel.plt
```

```
0x08048314 - 0x08048338 is .init
0x08048340 - 0x080483a0 is .plt
0x080483a0 - 0x08048648 is .text
0x08048648 - 0x0804865d is .fini
0x08048660 - 0x080486a9 is .rodata
0x080486ac - 0x080486f0 is .eh_frame_hdr
0x080486f0 - 0x080487f4 is .eh_frame
0x08049f08 - 0x08049f0c is .init_array
0x08049f0c - 0x08049f10 is .fini_array
0x08049f10 - 0x08049f14 is .jcr
0x08049f14 - 0x08049ffc is .dynamic
0x08049ffc - 0x0804a000 is .got
0x0804a000 - 0x0804a020 is .got.plt
0x0804a020 - 0x0804a028 is .data
0x0804a028 - 0x0804a02c is .bss
0xf7fdc114 - 0xf7fdc138 is .note.gnu.build-id in /lib/ld-
linux.so.2
0xf7fdc138 - 0xf7fdc1f4 is .hash in /lib/ld-linux.so.2
0xf7fdc1f4 - 0xf7fdc2d4 is .gnu.hash in /lib/ld-linux.so.2
0xf7fdc2d4 - 0xf7fdc494 is .dynsym in /lib/ld-linux.so.2
0xf7fdc494 - 0xf7fdc612 is .dynstr in /lib/ld-linux.so.2
0xf7fdc612 - 0xf7fdc64a is .gnu.version in /lib/ld-linux.so.2
0xf7fdc64c - 0xf7fdc714 is .gnu.version_d in /lib/ld-linux.so.2
0xf7fdc714 - 0xf7fdc77c is .rel.dyn in /lib/ld-linux.so.2
0xf7fdc77c - 0xf7fdc7ac is .rel.plt in /lib/ld-linux.so.2
0xf7fdc7b0 - 0xf7fdc820 is .plt in /lib/ld-linux.so.2
0xf7fdc820 - 0xf7ff4baf is .text in /lib/ld-linux.so.2
0xf7ff4bc0 - 0xf7ff8a60 is .rodata in /lib/ld-linux.so.2
0xf7ff8a60 - 0xf7ff90ec is .eh_frame_hdr in /lib/ld-linux.so.2
0xf7ff90ec - 0xf7ffb654 is .eh_frame in /lib/ld-linux.so.2

0xf7ffccc0 - 0xf7ffcf3c is .data.rel.ro in /lib/ld-linux.so.2

0xf7ffcf3c - 0xf7ffcff4 is .dynamic in /lib/ld-linux.so.2
```

Этот список может стать еще длиннее в более сложных двоичных файлах с многочисленными зависимостями и библиотеками. Глядя на этот вывод, вы знаете, что раздел .text исполняемого файла расположен по адресу 0x080483a0. Дизассемблирование кода в этом месте может дать подсказку к точке ввода целевого кода. На рисунке 8.2 показан результат дизассемблирования кода в этом месте в gdb.

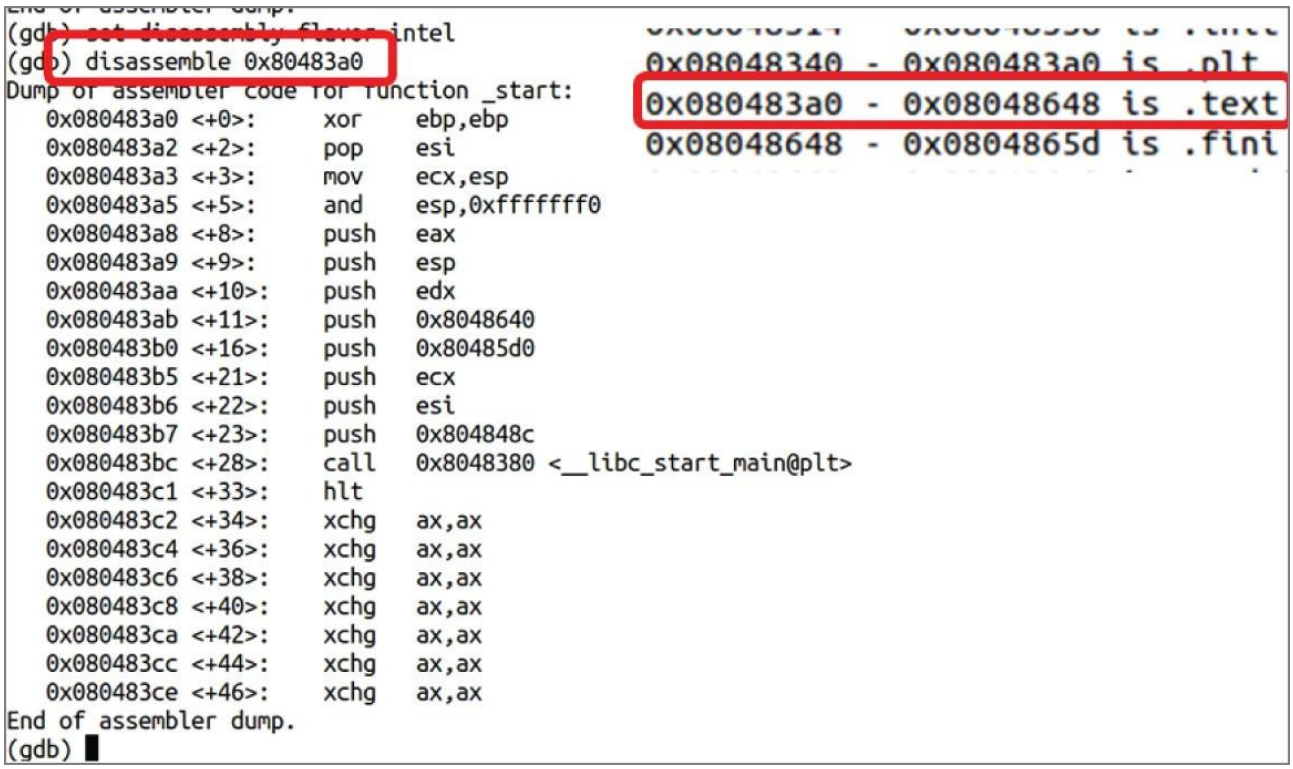


Рисунок 8.2:дизассемблирование текста в gdb

При поиске точки входа в целевой код это может зависеть от точного компилятора и языка, используемого для сборки. Вы увидите пример поиска исходного кода в приложении на C/C++, поскольку это все еще один из наиболее распространенных языков, используемых сегодня. Для начала найдите вызов __libc_start_main. Адрес целевого кода будет передан в качестве параметра этой функции, и, учитывая то, что вы знаете о соглашениях о вызовах, вы знаете, что это означает, что мы ищем то, что помещается в стек перед вызовом.

На рисунке 8.2 адрес 0x804848c помещается в стек непосредственно перед вызовом __libc_start_main, что делает его параметром функции. Следовательно, целевой код начинается с этого адреса. На рисунке 8.3 показана разборка основной функции, включая вызовы libc.

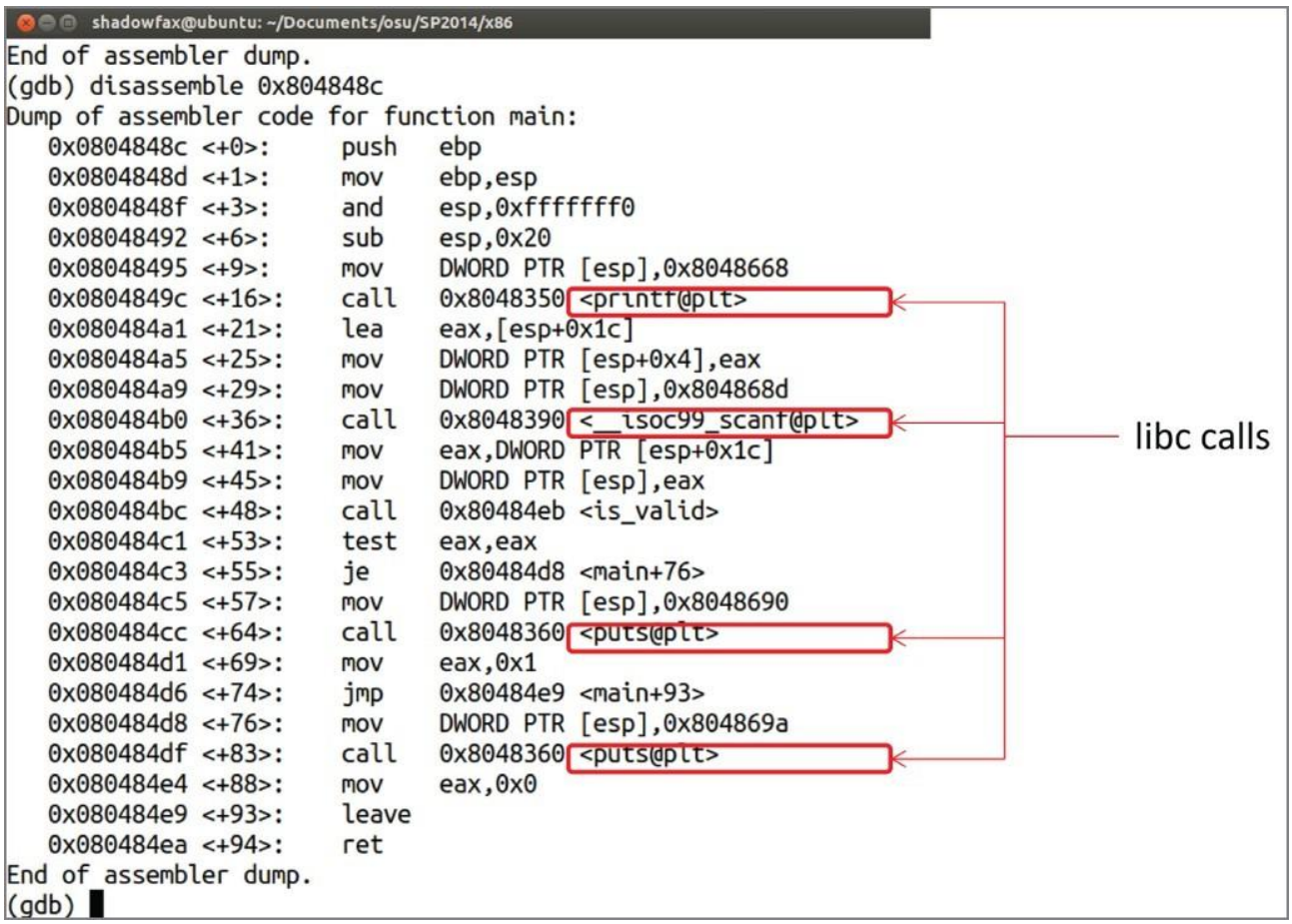


Рисунок 8.3: Дизассемблирование основной функции в gdb

Компиляторы

Компиляторы берут код и преобразуют его в машинный код, который может прочитать процессор. Существуют различные действия, которые компиляторы могут делать, чтобы повлиять на обратное проектирование, как намеренно, так и непреднамеренно. Этот раздел посвящен непреднамеренным изменениям; преднамеренные методы, такие как запутывание, будут рассмотрены в главе 12 “Защита”.

Оптимизация

Компиляторы могут быть настроены на оптимизацию кода на основе различных показателей, включая скорость и размер диска, или не оптимизироваться вообще. Код может выглядеть очень по-разному в зависимости от того, применялась ли оптимизация.

Рассмотрим следующий пример кода. В этом коде реализован простой оператор if с двумя условиями.

```
int main(int argc, char* argv[])
{
        if (argc>= 3 && argc <= 8)
        {
              printf("valid number of args\n");
        }
}
```

На рисунке 8.4 показано, как выглядит код в дизассемблере (подробнее об этом в главе 11 “Исправление и расширенный инструментарий”, не волнуйтесь) при компиляции без оптимизаций. Обратите внимание, что проверки для двух условий, сравнивающих значения 2 и 8, четко видны в коде.

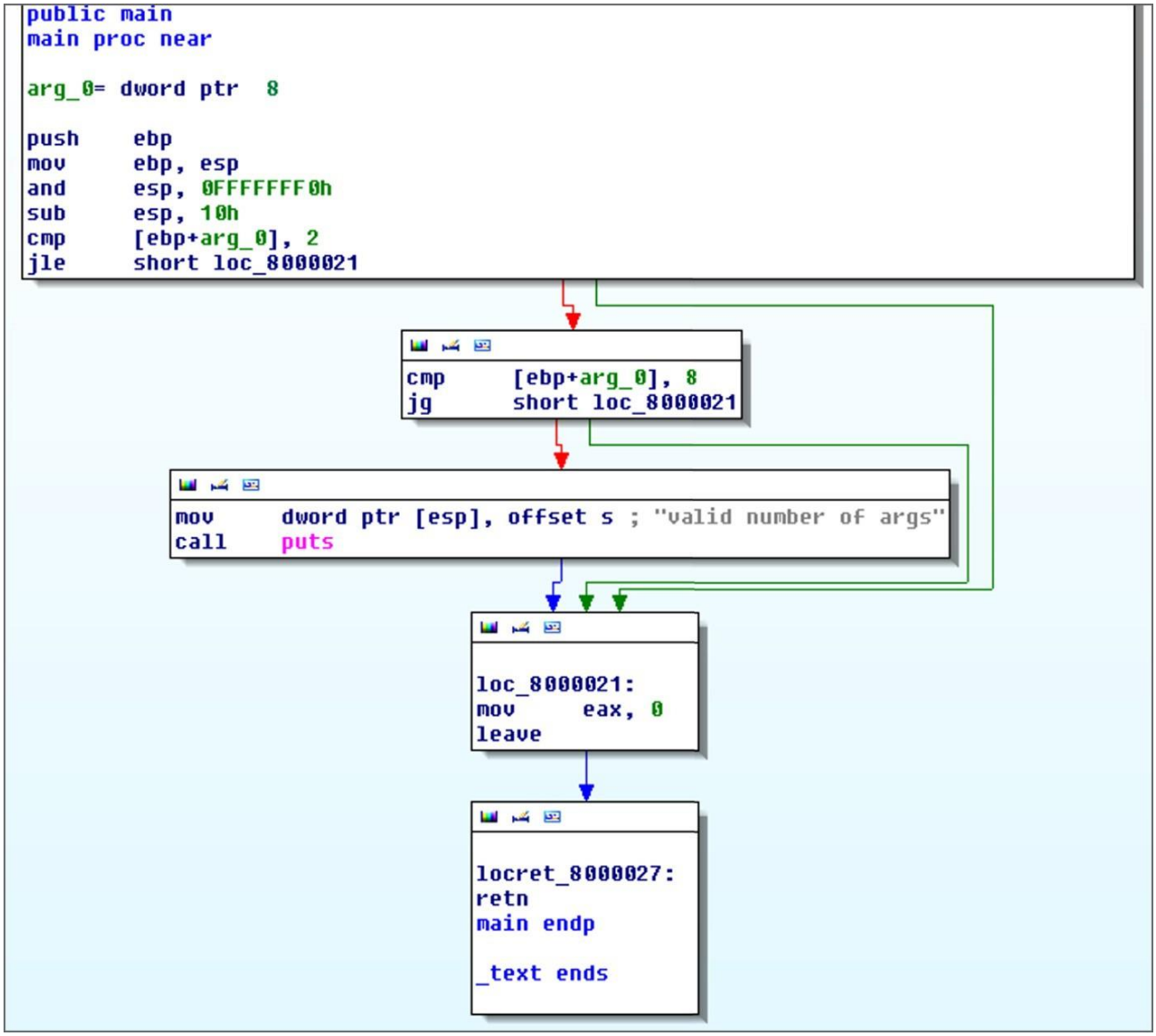


Рисунок 8.4: Неоптимизированный код в дизассемблере

На рисунке 8.5 показан тот же код, оптимизированный по скорости и занимаемому пространству. Сравнения со значениями 2 и 8 больше не видны в коде, и код больше не выглядит как оператор if с двумя условиями.

На рисунке 8.6 показан код, оптимизированный исключительно на основе дискового пространства. Опять же, два сравнения отсутствуют.

Если вы изучите код, вы увидите, что код проверяет, если (argc-3) > 5. Если argc < 3, то вычитание 3 вызовет недостаточный поток и приведет к тому, что значение в eax будет большим положительным числом. Если argc > 8, то argc-3 > 5. В обоих этих случаях результат будет больше 5, поэтому оптимизированный оператор эквивалентен исходному

тесту. Оптимизация компилятора приводит к эквивалентной логике, но может значительно усложнить чтение кода и его осмысление.

В большинстве компиляторов есть опции для настройки уровня оптимизации. Во время обучения, если у вас возникли трудности с реверсированием написанного вами приложения, попробуйте отключить оптимизацию при компиляции. С другой стороны, если вы хотите усложнить реинжиниринг вашего кода, оптимизация компилятора - простой и выгодный способ сделать это.

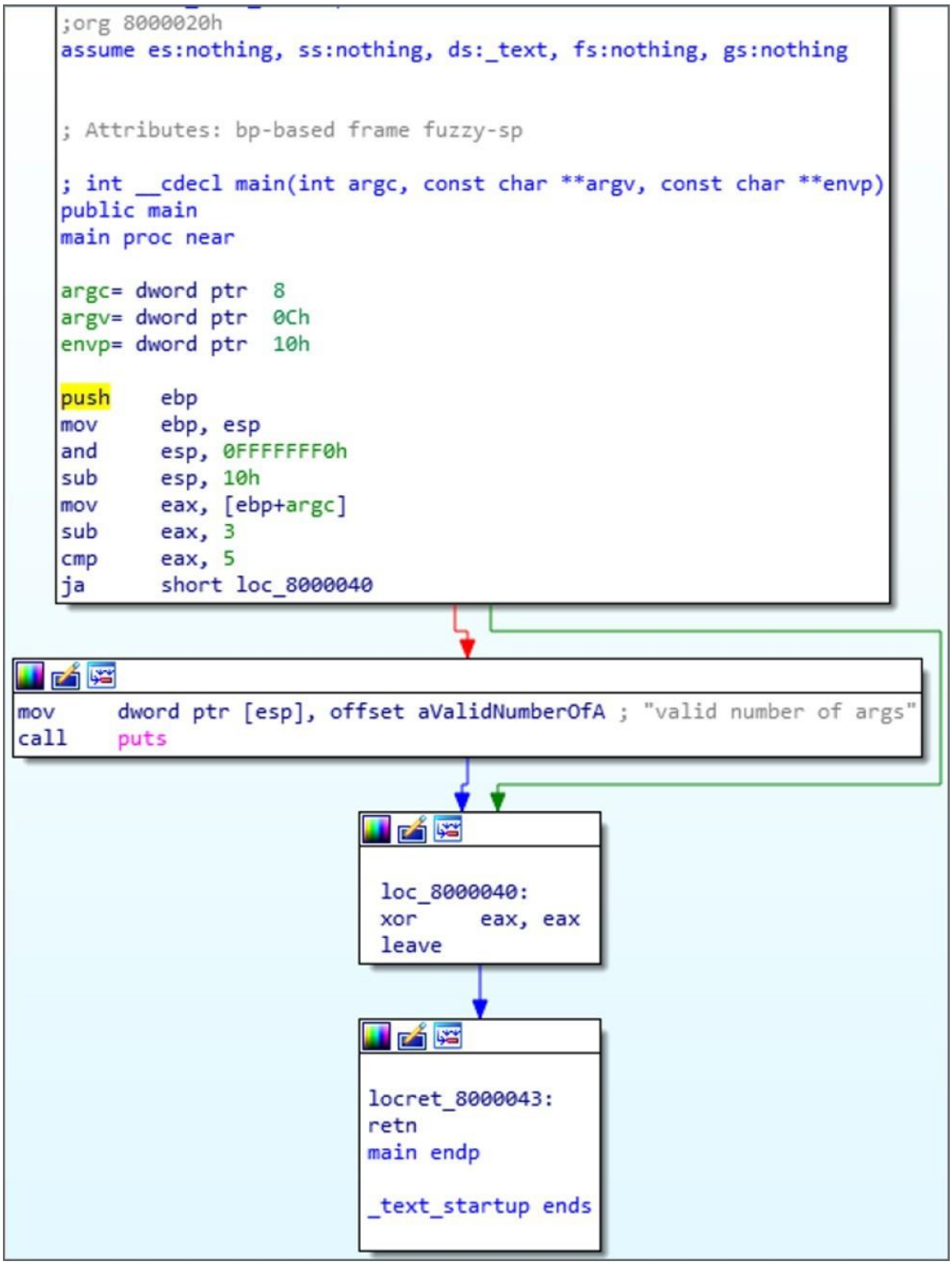


Рисунок 8.5: Оптимизированный по скорости и пространству код в дизассемблере

Удаление

Удаление двоичного файла означает удаление всей информации, которая не является

необходимой для выполнения кода, включая таблицу символов. Нерасчлененный двоичный файл сохраняет свою таблицу символов, в то время как разделенный - нет.

Символы могут быть чрезвычайно полезны для отладки приложения. Например, рассмотрим следующий код:

```
// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
	double sum = 0.0;

	// Sum all the values bar(1) to bar(count)
	for (int i = 1; i <= count; i++)
		sum += bar((double) i);
	return sum;
}
```

```
public main
main proc near

var 4= duord pts -4
argc= dword pts g
argv= dword ptr 0Ch
endp= dword ptr 10h

lea ecx, [esp+4]
and esp, 0FFFFFFFOh
push dword ptr [ecx-4]
push ebp
‹sov ebp, esp
push ecx
push eax
‹Nov eax, [ecx]
sub eax, 3
cmp eax, 5
ja short loc 8€XMX}47
```

```
sub esp, BCh
push offset aValidNumberOfA ; 'valid numbs' cf a gs”
call puts
add esp, 16h
```

```
loc_8000047:
‹»ov ecx, [ebp+var 4]
xor eax, eax
leave
tea esp, [ecx-4j
```

```
1ocret 8eeeese:
retn
aaln endp

_text_startup ends
```

Рисунок 8.6: Оптимизированный по пространству код в дизассемблере

Если этот код будет проанализирован компилятором, он, по крайней мере, будет содержать записи таблицы символов, показанные на рисунке 8.7. Символы настолько полезны при отладке, что Microsoft позволяет загружать символы для своих приложений на случай, если вам потребуется устранить неполадки! Эта дополнительная информация может оказаться бесценной для понимания цели, стоящей за приложением.

Symbol name	Type	Scope
bar	function, double	extern
x	double	function parameter
foo	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

Рисунок 8.7: Символы отладки приложения

Если файл удален, при открытии в gdb будет показано, что символы отладки не найдены, как показано на рисунке 8.1. Эти файлы гораздо сложнее реконструировать.

Символы можно удалить из приложения несколькими различными способами. Один из вариантов – использовать флаги компилятора, такие как gcc -fno–rtti -s. Другой вариант - использовать инструменты удаления после сборки, такие как strip в Linux.

Символы облегчают злоумышленнику реинжиниринг приложения, поскольку они могут помочь определить интересующие области и понять намерения, стоящие за определенными переменными. Однако существуют законные причины оставить приложение незарегистрированным. Например, символы помогают создавать отчеты о сбоях и журналы ошибок и поддерживают законную отладку для исправления ошибок клиента. Во время обучения, если вы пишете свой собственный код и компилируете его для практики, начните с того, что убедитесь, что вы создаете с оставленными символами. По мере того как вы будете совершенствоваться в своих навыках, удаляйте символы. При обратном проектировании чужого кода крайне маловероятно, что вы обнаружите, что в нем были оставлены символы, но такое случается!

Связывание

Приложения теперь редко пишутся изолированно. Что более распространено, так это включение библиотек, которые предоставляют основные возможности (такие как связь, ведение журнала, рисование и т.д.). При компиляции приложения, использующего библиотеки, есть два варианта их создания. Эти библиотеки могут быть статически или динамически связаны с приложением. Каждая из них имеет свои преимущества и недостатки с точки зрения взлома программного обеспечения.

Статическое связывание
При статическом связывании библиотеки встраиваются в само приложение. Это повышает скорость выполнения, поскольку целевые адреса любых обращений к библиотеке встраиваются в нее во время компиляции. Кроме того, статически связанные приложения более переносимы, поскольку у них меньше зависимостей от среды.

Однако статическая компоновка также имеет свои недостатки. Приложения со статической компоновкой имеют больший размер, поскольку вся библиотека встроена в исполняемый файл, даже если вы используете только одну функцию из большой библиотеки. Кроме того, любые обновления библиотеки требуют перекомпиляции приложений, использующих их.

Увеличение размера файла, вызванное статическим связыванием, может быть значительным для программ. Например, как показано на рисунке 8.8, даже простая однострочная программа “hello world” свяжет десятки библиотек.

Динамическое связывание
Динамическая компоновка - это другой вариант, используемый по умолчанию для многих компиляторов. При динамической компоновке необходимые библиотеки находятся в системе во время выполнения. Если библиотека еще не загружена в системную память, она должна быть найдена в системе и загружена в общую библиотечную память; однако общие библиотеки, скорее всего, уже загружены и доступны для использования.

```
0xf7fdc114 - 0xf7fdc138 is .note.gnu.build-id in /lib/ld-linux.so.2
0xf7fdc138 - 0xf7fdc1f4 is .hash in /lib/ld-linux.so.2
0xf7fdc1f4 - 0xf7fdc2d4 is .gnu.hash in /lib/ld-linux.so.2
0xf7fdc2d4 - 0xf7fdc494 is .dynsym in /lib/ld-linux.so.2
0xf7fdc494 - 0xf7fdc612 is .dynstr in /lib/ld-linux.so.2
0xf7fdc612 - 0xf7fdc64a is .gnu.version in /lib/ld-linux.so.2
0xf7fdc64c - 0xf7fdc714 is .gnu.version_d in /lib/ld-linux.so.2
0xf7fdc714 - 0xf7fdc77c is .rel.dyn in /lib/ld-linux.so.2
0xf7fdc77c - 0xf7fdc7ac is .rel.plt in /lib/ld-linux.so.2
0xf7fdc7b0 - 0xf7fdc820 is .plt in /lib/ld-linux.so.2
0xf7fdc820 - 0xf7ff4baf is .text in /lib/ld-linux.so.2  96KB!
0xf7ff4bc0 - 0xf7ff8a60 is .rodata in /lib/ld-linux.so.2
0xf7ff8a60 - 0xf7ff90ec is .eh_frame_hdr in /lib/ld-linux.so.2
0xf7ff90ec - 0xf7ffb654 is .eh_frame in /lib/ld-linux.so.2
0xf7ffccc0 - 0xf7ffcf3c is .data.rel.ro in /lib/ld-linux.so.2
0xf7ffcf3c - 0xf7ffcff4 is .dynamic in /lib/ld-linux.so.2

..(many more)...
```

Рисунок 8.8: Связанные библиотеки в программе “hello world”

Динамическая компоновка уменьшает размер приложения и устраняет необходимость перекомпиляции приложения после обновления библиотеки, если обновление обратно совместимо. Кроме того, динамически связанные приложения могут загружаться быстрее, если используемые ими библиотеки уже загружены в память.

Однако динамически связанные приложения зависят от библиотек, которые им необходимы для установки в системе, и могут работать медленнее, чем статически связанные приложения (если зависимости еще не загружены и их необходимо найти и загрузить). В дополнение к необходимости загружать любые библиотеки, которых еще нет в памяти, динамически подключаемым приложениям необходимо находить адреса вызываемых функций во время выполнения. Это включает поиск библиотеки в общем пространстве памяти и может потребовать большого объема подкачки памяти.

Влияние компоновки на безопасность

Выбор того, использовать статическую или динамическую компоновку, зависит от разработчика или компилятора. Но, надевая шляпы для взлома программного обеспечения, оба варианта имеют свои последствия для безопасности.

Реверс-инженеры обычно предпочитают, чтобы приложение было статически связано. Статическое связывание упрощает определение точного адреса загрузки функций общей библиотеки, что полезно при разработке эксплойтов. Это означает, что вы можете использовать код в разделяемых библиотеках для выполнения своей эксплуатации, и этот код будет находиться в предсказуемом месте внутри вашего двоичного файла во время выполнения. Использование библиотеки, которая подключается динамически, возможно, но это гораздо сложнее из-за необходимости искать нужную библиотеку в общей библиотечной памяти и каждый раз определять ее адрес, поскольку она будет перемещаться и быть непредсказуемой.

Взломщики, с другой стороны, как правило, предпочитают динамически связанные библиотеки. Динамическое связывание приводит к гораздо меньшему количеству кода для просеивания, и взломщики заинтересованы исключительно в пользовательском коде приложения, а не в коде общей библиотеки.

Резюме

Процесс компиляции и оптимизации приложения может значительно затруднить обратное проектирование, даже если компилятор намеренно не запутывает его. Однако, как и любая защита от обратного копирования, это может только замедлить процесс, поскольку ни одно программное обеспечение не поддается взлому.

Глава 9
Реверс-инжиниринг: инструменты и стратегии
До этого момента основное внимание в этой книге уделялось пониманию того, как работают внутренности компьютеров. Это важно для эффективного взломщика программного обеспечения.

Теперь, когда у вас есть основы, акцент смещается на искусство взлома программного обеспечения. Чтобы поэкспериментировать и попрактиковаться во взломе, вы будете работать с различными целями:

- Реальное программное обеспечение: Программное обеспечение, взятое из реального мира. - При анализе реального программного обеспечения вы должны учитывать закон об авторском праве, чтобы гарантировать отсутствие нарушений авторских прав.
- Готовые примеры: Приложения, написанные для этой книги, чтобы проиллюстрировать конкретные концепции.
crackmes: Небольшие программы, которые можно взломать, написанные другими взломщиками программного обеспечения, чтобы продемонстрировать идею и бросить вызов другим.
crackmes, подобные тем, которые используются в этом курсе, - это готовые примеры, которые предоставляют несколько преимуществ начинающему взломщику. В общем, они разработаны таким образом, чтобы быть разрешимыми, легальными для взлома и безопасными для запуска в отладчике.

взломщики также часто помечаются в зависимости от их направленности, уровня знаний и т.д. В результате вы можете специально искать сложные задачи, соответствующие вашим интересам и уровню квалификации (например, продвинутый взломщик C или начинающий взломщик Java).

Лаборатория: RE Bingo
Эта лабораторная работа дает практический опыт обращения кода, который был создан (и обфускирован) компилятором.

Лабораторные работы и все связанные с ними инструкции можно найти в соответствующей папке здесь:

https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES

Для этой лаборатории, пожалуйста, найдите Lab RE Bingo и следуйте предоставленным инструкциям.

Навыки
В этой лабораторной работе objdump используется для отработки идентификации конструкций потока управления и настроек компилятора при реверсировании. Некоторые из тестируемых ключевых навыков включают следующее:

- Обратное проектирование x86
- Конструкции потока управления
- Влияние настроек компилятора

Выводы
Быстрое определение конструкций потока управления может значительно ускорить обратное

проектирование. Они дают представление о логике приложения и делают его более читабельным и понятным.

Однако конфигурация компилятора оказывает значительное влияние на скорость обратного проектирования. Например, удаление и оптимизация, как правило, замедляют работу.

В более крупных и сложных программах часто необходима автоматизация некоторого обратного проектирования. Обычно пишутся пользовательские инструменты для конкретной цели. Распаковка, деобфускация и обход анти-отладочных проверок являются обычными задачами для автоматизации.

Базовая разведка
Как взломщик программного обеспечения, это наиболее распространенная ситуация, с которой вы столкнетесь:

- Вы хотите взломать программу.
- У вас нет исходного кода.
- У вас есть исполняемый файл.

В этой ситуации вам нужны средства быстрой оценки целевого исполняемого файла и поиска отправной точки для вашего анализа. Одними из наиболее часто используемых начальных инструментов для реверс-инжиниринга являются objdump, strace, ltrace и strings. По мере чтения книги вы познакомитесь с более продвинутыми инструментами, но, поскольку это одни из самых фундаментальных, они являются хорошей отправной точкой.

objdump
Object Dump (objdump) - это инструмент на базе Linux для сброса дизассемблирования любой программы. Как показано на рисунке 9.1, он имеет множество опций. Наиболее важными из них для быстрого обратного проектирования являются следующие:

- -d: Указывает objdump разобрать содержимое всех разделов
- -Mintel: Указывает objdump отображать сборку в синтаксисе Intel (в отличие от AT&T)

Например, чтобы разобрать приложение с именем appname, используйте команду objdump –d –Mintel appname.

```
shadowfax@ubuntu:~/Desktop/blackhat$ objdump
Usage: objdump <option(s)> <file(s)>
 Display information from object <file(s)>.
 At least one of the following switches must be given:
  -a, --archive-headers    Display archive header information
  -f, --file-headers       Display the contents of the overall file header
  -p, --private-headers    Display object format specific file header contents
  -P, --private=OPT,OPT... Display object format specific contents
  -h, --[section-]headers  Display the contents of the section headers
  -x, --all-headers        Display the contents of all headers
  -d, --disassemble        Display assembler contents of executable sections
```

Рисунок 9.1: параметры objdump

На рисунке 9.2 показаны выходные данные при запуске objdump в примере приложения. Обратите внимание, что objdump отобразит ячейки памяти, имена функций, машинный код x86 и сборку x86.

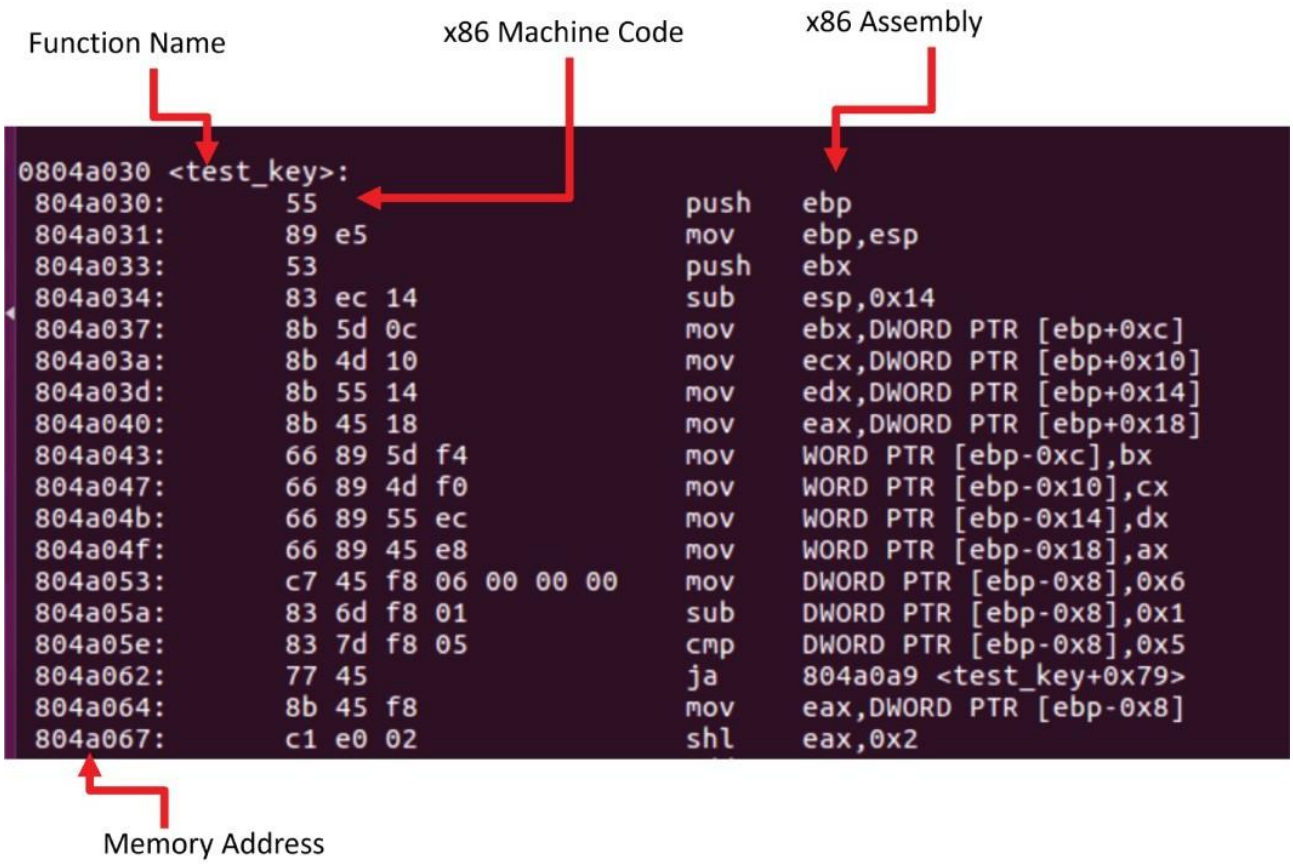


Рисунок 9.2: Пример вывода objdump

strace и ltrace
strace и ltrace предоставляют возможность отслеживать библиотечные (ltrace) и системные (strace) вызовы. Они позволяют отслеживать работу программы и получать представление о том, что делают другие программы.

Если какая-либо программа на любом языке хочет сделать что-либо полезное, ей придется выполнять системные вызовы. Характеристика того, какие библиотеки и внешние функциональные возможности она использует, может быть чрезвычайно полезна при проведении разведки системы. Вы заметите, что с помощью этих инструментов вы можете не только увидеть, что оно использует, но и кто им пользуется (т.е. какой адрес в вызываемом приложении). Таким образом, это также может помочь вам сосредоточиться на полезных функциях. Например, вы можете увидеть, какой фрагмент кода чаще всего обращается к криптографическим библиотекам; это, вероятно, интересно с точки зрения взлома.

ltrace
ltrace (трассировка библиотеки) - это утилита командной строки Linux, которая отслеживает вызовы библиотек. Библиотечные вызовы - это вызовы вашим приложением динамически связанных библиотек. Синтаксис команды - ltrace <команда>.

Например, если вы #include <stdio.h>, эта библиотека динамически подключается при загрузке вашей программы. Когда вы вызываете printf или fopen, это вызывает стандартную библиотеку C. Эта конструкция справедлива для всех языков программирования, которые все включают понятие включения внешних библиотек.

strace

strace (system trace) - утилита командной строки Linux, которая отслеживает системные вызовы. Синтаксис команды strace <команда>.

Системные вызовы - это вызовы вашего приложения операционной системе, которая управляет такими вещами, как файлы и окно вашей консоли. Такие функции, как fopen и printf, в конечном счете, в своей внутренней работе должны выполнять вызовы операционной системы. Как и в случае с ltrace, это справедливо для всех языков программирования; редко существует приложение, которое не использует функциональность на уровне операционной системы.

пример strace: echo
Мониторинг системных вызовов предоставляет простой способ отслеживания работы программы. Предположим, вы написали утилиту echo и хотели посмотреть, как она работает.

echo - это команда Linux, которая преобразует входные данные в выходные. Например, команда echo hello! выводит "привет!" на терминал.

Но что он на самом деле делает? Запуск strace echo hello! приведет к выводу, аналогичному рисунку 9.3.

```
steph@steph-XPS-9320:~$ strace echo hello!
execve("/usr/bin/echo", ["echo", "hello!"], 0x7ffec41908e8 /* 44 vars */) = 0
brk(NULL)                               = 0x55976ad13000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc703b2180) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f814463f000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=78555, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 78555, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f814462b000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\244;\374\204(\337f#\315I\214\234\f\256\271\32"..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2216304, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2260560, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8144400000
mmap(0x7f8144428000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f8144428000
mmap(0x7f81445bd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f81445bd000
mmap(0x7f8144615000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x214000) = 0x7f8144615000
mmap(0x7f814461b000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f814461b000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8144628000
arch_prctl(ARCH_SET_FS, 0x7f8144628740) = 0
set_tid_address(0x7f8144628a10)         = 142481
set_robust_list(0x7f8144628a20, 24)     = 0
rseq(0x7f81446290e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7f8144615000, 16384, PROT_READ) = 0
mprotect(0x559768f6d000, 4096, PROT_READ) = 0
mprotect(0x7f8144679000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f814462b000, 78555)           = 0
getrandom("\xa1\x77\x9f\xec\xe6\x33\xaa\xf1", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55976ad13000
brk(0x55976ad34000)                     = 0x55976ad34000
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=5712208, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 5712208, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8143e00000
close(3)                                = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
write(1, "hello!\n", 7hello!
)               = 7
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Рисунок 9.3: вывод strace для echo hello!

Это изображение сложное, и его может быть сложно расшифровать. Просматривая результат, вы можете увидеть некоторые стандартные системные вызовы в начале, которые используются для запуска программы echo.

Следующие строки являются интересными выводами, которые находятся в самом конце:

```
write(1, "hello!\n", 7hello!
)                       = 7
close(1)                = 0
```

Это говорит о том, что echo записало строку в поток 1, который, помните, является стандартным выводом. Команда write имела возвращаемое значение 7, потому что было записано семь символов. Наконец, echo закрыла поток 1, который вернул 0 для успешного выполнения. Хотя это кажется простым, представьте, что вы используете это для отслеживания того, куда приложение записало часть данных конфигурации. Допустим, вы изменили настройку и хотите посмотреть, как она сохраняет это в системе.

пример strace: Вредоносные котята
Comet Cursor был ранним примером шпионского ПО в ОС Windows. Оно позволяло пользователям изменять внешний вид курсора мыши, а веб-сайтам использовать настраиваемые курсоры. Приложение устанавливалось само по себе без разрешения пользователя и тайно отслеживало пользователей.

Как показано на рисунке 9.4, в дикой природе существует множество приложений для курсора kitten. В этом примере используется пример приложения для курсора, которое тайно обращается к российскому IP-адресу.

Запуск кода не показывает признаков вредоносной функциональности, как показано здесь:

```
deltaop@deltaleph-ubuntu:~$ ./kittens
Registering kitten cursor!
Done!  Enjoy the kitties!
deltaop@deltaleph-ubuntu:~$
```

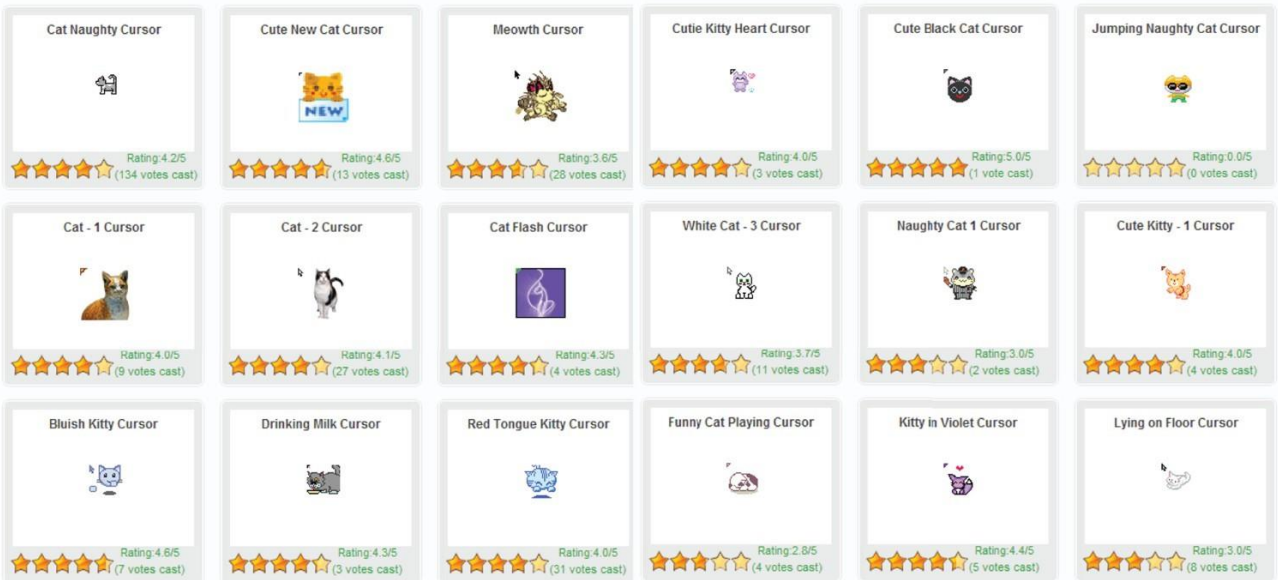


Рисунок 9.4: Приложения с курсором Kitten

Однако анализ кода в strace рассказывает другую историю, как показано здесь:

```
deltaop@deltaleph-ubuntu:~$ strace ./kittens
...
poll([{fd=3, events=POLLOUT}], 1, 0)     = 1 ([{fd=3,
revents=POLLOUT}])
send(3, "!$\1\0\0\1\0\0\0\0\0\0\7kremlin\2ru\0\0\34\0\1",
      28, MSG_NOSIGNAL) = 28
poll([{fd=3, events=POLLIN}], 1, 5000)   = 1 ([{fd=3,
revents=POLLIN}])
ioctl(3, FIONREAD, [28])                 = 0
recvfrom(3, "!$\201\200\0\1\0\0\0\0\0\0\7kremlin\2ru\0\0\34\0\1",
1024,
       0, {sa_family=AF_INET, sin_port=htons(53),sin_addr=inet_
          addr("192.168.1.1")}, [16]) = 28
close(3)                                 = 0
socket(PF_INET, SOCK_DGRAM|SOCK_NONBLOCK, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(53),
       sin_addr=inet_addr("192.168.1.1")}, 16) = 0


...
```

Этот пример выходных данных strace показывает несколько событий. Чтобы сосредоточиться на интересующих событиях, используйте grep (который ограничивает результаты строками, соответствующими вашей строке поиска, в данном случае connect).

```
deltaop@deltaleph-ubuntu:~$ strace -f ./kittens 2>&1 | grep
connect

connect(3, {sa_family=AF_FILE, path="/var/run/nscd/socket"},
       110) = -1 ENOENT
connect(3, {sa_family=AF_FILE, path="/var/run/nscd/socket"},
       110) = -1 ENOENT
 connect(3, {sa_family=AF_INET, sin_port=htons(53),
        sin_addr=inet_addr("192.168.1.1")}, 16) = 0
 connect(3, {sa_family=AF_INET, sin_port=htons(53),
        sin_addr=inet_addr("192.168.1.1")}, 16) = 0
 connect(3, {sa_family=AF_INET, sin_port=htons(53),
        sin_addr=inet_addr("192.168.1.1")}, 16) = 0
 connect(3, {sa_family=AF_INET, sin_port=htons(80),
       sin_addr=inet_addr("195.208.24.91")}, 16) = 0
write(2, "connected.\n", 11) = 11
```

Предыдущий пример вывода ищет события со словом connect в них. Это включает в себя несколько подключений к Интернету, в том числе одно к 195.208.24.91, что подозрительно, поскольку это внешний IP-адрес, и зачем вашему курсору это делать?

strings

strings - это утилита Linux, предназначенная для извлечения строк для печати, используемых

приложением. Она ищет серию символов ASCII для печати с (настраиваемой) минимальной длиной и печатает все, что находит.

строки могут быть очень полезны при реверс-инжиниринге, поскольку они обеспечивают высокоуровневое понимание того, что может делать программа. Кроме того, как только вы найдете интересующие вас строки, позже вы увидите, как можно использовать эти строки, чтобы легко найти связанный фрагмент кода. Например, строка с надписью "неверный пароль" может быть использована для быстрого отслеживания того, где находится код для обработки пароля. Например, следующие строки содержат ценные подсказки о приложении:

- "Введите пароль:"
- "open_socket"
- "ВАШИ ФАЙЛЫ БЫЛИ ЗАШИФРОВАНЫ!"

Синтаксис команды - strings program. Хотя обычно она используется без параметров, при обратном проектировании иногда полезны следующие флаги:

- -a: Показывать все строки в файле, а не только те, которые находятся в загруженных разделах объектных файлов. Это часто полезно при работе с запутанными, вложенными или иными необычными двоичными файлами.
- -n: Укажите минимальную длину последовательных символов, пригодных для печати, чтобы последовательность байтов считалась строкой. Значение по умолчанию равно 4. Часто бывает полезно расширить или ограничить количество строк, найденных инструментом.

Средство просмотра зависимостей

Средство просмотра зависимостей - это метод, используемый для быстрого понимания импорта и экспорта приложения. Средство просмотра зависимостей является одним из примеров такого инструмента. (Ссылки смотрите в разделе “Инструменты” нашего репозитория.)

Обход зависимостей обеспечивает ценное высокоуровневое представление о том, какие действия будет выполнять программа, и часто является полезным первым шагом при взломе. Большинство приложений не реализуют все свои собственные функции; они будут использовать функции из операционной системы или внешних библиотек. Каждый раз, когда приложение выходит за пределы своего кода, это будет отображаться как импортированная функция. Кроме того, часто приложения будут совместно использовать функциональность с другими приложениями, и всякий раз, когда функция является чем-то “доступным для совместного использования”, она будет отображаться как экспорт приложения.

Загрузка программы в программу, подобную Dependency Walker, показывает библиотеки DLL, которые она использует, и ожидаемые вызовы API. На рисунке 9.5 показано, что программа создаст несколько разделов реестра.

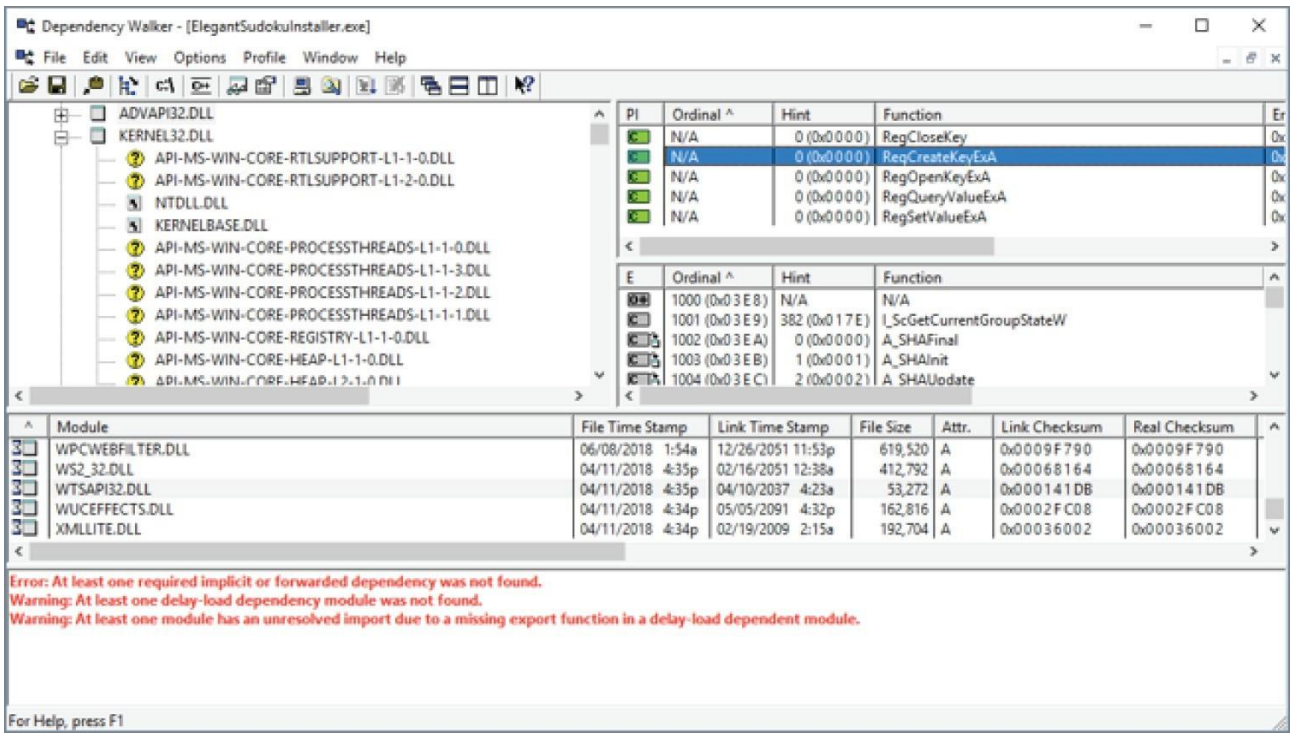


Рисунок 9.5: Анализ изменений реестра в Dependency Walker

Стратегия обратного проектирования

Обратное проектирование по-прежнему является скорее искусством, чем наукой. Хотя доступны отличные инструменты и методы, которые могут помочь, эффективное реверсирование в конечном итоге в значительной степени зависит от интуиции и опыта.

Таким образом, невозможно дать предписывающее решение. Однако существуют общие подходы и лучшие практики, которые могут помочь.

Найдите области, представляющие интерес

Приложения содержат большие объемы кода, и большая его часть не имеет отношения к делу или не нужна для обратного проектирования. Важным первым шагом при реверсировании приложения является поиск нужной области программы.

Вы продолжите изучать множество интересных методов, позволяющих сузить область поиска, но некоторые из них теперь доступны вам, основываясь только на знании предыдущих инструментов:

- Интересные строки: Найдите строки программы, которые вас интересуют (например, “Неправильный ключ”), и найдите, где используются эти строки (например, определите вызовы printf с использованием этих строк).
- Пользовательский ввод: Найдите, где принимаются входные данные от пользователя (например, scanf, диалоговые окна и т.д.), и найдите, где эти входные данные обрабатываются.
- Системный ввод: Найдите, где считываются входные данные из системы, такие как файлы конфигурации и параметры реестра.
- Код аутентификации: Если возможно, используйте отладчик, чтобы приостановить работу программы после ввода имени пользователя/ключа. Затем просканируйте память в поисках введенных значений, установите точки останова HW в этих местах и повторно запустите приложение, чтобы найти, где считываются или записываются значения.
- Итеративное комментирование кода

Даже после того, как вы определите интересующий код, его может быть трудно понять. Один из подходов к пониманию сложного кода заключается в выполнении нескольких проходов, добавляя информацию (например, комментарии) во время каждого прохода.

Для этого комментируйте целевую область до тех пор, пока не поймете, как она работает, используя следующий процесс:

- Идентифицируйте и помечайте локальные переменные: используйте правила соглашения о вызове для идентификации локальных переменных (например, [ebp-4] в cdecl). Сначала их можно пометить, используя что-то неопределенное (например, local1).
- Идентифицируйте и помечайте параметры функции: Используйте правила соглашения о вызове для идентификации параметров (например, [ebp+12] в cdecl). Сначала их также можно пометить, используя что-то неопределенное (например, arg1).
- Идентифицируйте вызовы API (например, atoi): Используйте знание параметров API для дальнейшего аннотирования локальных переменных. Например, в документации API указано, что atoi передается строка, которая будет преобразована в целое число, поэтому мы можем переименовать наш параметр integer_string.
- Добавляйте комментарии к сложным потокам управления: например, “этот код умножает число”.
- Уточняйте описания на основе наблюдаемых потоков данных: например, local1 может стать loop_counter, если вы видите, что он используется в качестве счетчика в цикле for.

Большая часть эффективного реверс-инжиниринга выполняется быстро. Даже в небольших программах слишком много кода, чтобы проанализировать все.

Подавляющее большинство кода приложения не будет иметь отношения к тому, что вам нужно. Знание того, на чем следует сосредоточиться, часто менее важно, чем знание того, на чем не следует сосредотачиваться. Изучение того, где совершать прыжки, требует времени.

Резюме

В этой главе представлены некоторые из основных инструментов и техник, которые вы будете использовать в качестве реинжиниринга программного обеспечения и взломщика. Прежде чем двигаться дальше, уделите некоторое время практике и приобретите некоторый практический опыт использования инструментов. Это время практики будет бесценным позже, когда вы перейдете к более сложному программному обеспечению и более продвинутым методам восстановления и взлома.

Глава 10
Взлом: инструменты и стратегии
Взлом - это искусство изменения программного обеспечения для обхода средств защиты или других нежелательных функций. В этой главе рассматриваются некоторые ключевые инструменты и стратегии, используемые для взлома программного обеспечения, включая использование генераторов ключей и исправлений для устранения средств проверки ключей.

Средства проверки ключей
Одной из наиболее распространенных практик лицензирования программного обеспечения является использование лицензионных ключей. В целях борьбы с пиратством для завершения установки каждого программного обеспечения требуется уникальный ключ. В случае программного обеспечения с несколькими уровнями функций некоторые функции могут быть всегда доступны в свободном доступе, в то время как другие находятся за лицензионной стеной, или программное обеспечение может вообще не работать без лицензионного ключа.

Лицензионные ключи являются распространенным решением для борьбы с пиратством, и у них есть свои преимущества. Это два наиболее важных параметра:

Лицензионные ключи легко генерировать и проверять.
Соотношение действительных и недействительных ключей настолько мало, что случайное угадывание вряд ли сгенерирует действительный ключ (при условии разумной длины ключа). Однако, как и все средства защиты, если они реализованы некачественно, они могут быть очень подвержены взлому, и, как и все средства безопасности, они не являются полностью безошибочными. Достаточно осведомленный и мотивированный взломщик может в конечном итоге победить их или обойти. Тем не менее, они по-прежнему являются одной из самых надежных форм защиты; это просто напоминание о том, что не существует такого понятия, как 100-процентно безопасное программное обеспечение.

В те времена, когда автономные системы были более распространены, проверка лицензий и валидация часто выполнялись полностью автономно, что означало, что вся логика проверки ключа была резидентной в системе. Сейчас, благодаря широкому подключению, мы часто видим проверки лицензионных ключей, которые состоят как из автономного, так и из онлайн-компонента, где они обращаются к серверу лицензий для дополнительной проверки. Существует несколько различных способов реализации проверок ключей с разным уровнем эффективности.

Плохой способ
В прошлом очень популярные компьютерные игры StarCraft и Half-life использовали контрольную сумму в качестве лицензионного ключа. Напомним, контрольные суммы часто представляют собой очень простые математические выражения, выполняемые в двоичном двоичном объекте, некоторые из которых так же просты, как сложение всех чисел вместе. В контрольной сумме, используемой в этих играх, 13-я цифра подтверждала первые 12.

Это означало, что пользователь мог ввести все, что он хотел, для первых 12 цифр, а затем вычислить 13-ю, чтобы создать действительную контрольную сумму. Это нарушение безопасности привело к появлению печально известного ключа 1234-56789-1234, который был действителен для этих игр и широко использовался для их пиратства.

Одной из самых больших проблем в этих случаях было то, что алгоритм, используемый для вычисления контрольной суммы, был слишком простым.

```
x = 3;
```

```
for(int i = 0; i < 12; i++)
{
    x += (2 * x) ^ digit[i];
}
lastDigit = x % 10;
```

Есть два способа взломать это. Первый заключается в том, что вы запускаете алгоритм и вычисляете допустимое значение последней цифры, как показано ранее.

Другой - атака методом перебора. Учитывая, что вам нужно было вычислить только одну цифру, есть только 10 вариантов для последней цифры [0-9]. Вы можете случайным образом выбрать набор из 12 цифр, а затем просто угадать и проверить 10 вариантов для последнего, пока не добьетесь успеха. Печально известный ключ 1234-56789-1234 был так знаменит, потому что его было легко запомнить, но, следуя любому из этих двух подходов (вычисление или грубая сила), вы могли бы сгенерировать любое количество новых ключей.

Разумный способ
Атака методом перебора на лицензионный ключ гарантированно сработает... в конечном счете. Лучшее, что может сделать лицензионный ключ, - это отнять у взломщика столько времени, что атака методом перебора становится неосуществимой или невозможной вообще.

Итак, как защититься от атак методом перебора? Одним из распространенных вариантов в других контекстах является криптографический хэш. Например, лицензионный ключ может быть реализован с использованием одного из следующих вариантов:

Имя пользователя: SHA(username)
Случайное значение: WXYZ-SHA(WXYZ)
Использование хэш-функции значительно усложняет атаку методом перебора. Однако взломщику тривиально легко определить, как работает алгоритм, после просмотра кода. В зависимости от вашего склада ума, если вы злоумышленник, это означает использование навыков реверс-инжиниринга, которым вы научились к этому моменту, чтобы найти алгоритм и разгадать его, а если вы защитник, это означает, что это ключевой фрагмент кода, который вам необходимо защитить.

Альтернативой является использование пользовательского сложного хэша, а не стандартного. Хотя обычно это ужасная идея с точки зрения безопасности, это не неслыханный выбор для данного приложения. Цель не в том, чтобы обеспечить абсолютную защиту, а просто замедлить обратное проектирование. Для тех, кто работает в сфере безопасности, у кого поджимаются пальцы при предложении сделать свой собственный хэш, просто обратите внимание, что это предложение сопровождается оговоркой, что вы в состоянии приготовить прилично хороший хэш. Как защитник, имейте в виду, что существует множество инструментов для выполнения распространенных методов хэширования, так что это будут все первые действия, которые злоумышленник попытается развернуть ваш ключ.

Кроме того, найдите способы добавить уникальную сложность, чтобы ключ можно было использовать только в уникальной настройке, а не распространять. Такие схемы, как объединение названия продукта, версии и имени компьютера в хэшированном значении, повышают уровень сложности. Таким образом, взломанный действительный ключ для одной установки не разблокирует другие версии.

Лучший способ
Хэши лучше, и, при правильной реализации, они могут быть достойными. Но есть еще лучшие варианты. Отличным примером этого является подход, который Microsoft использует при генерации лицензионных ключей для своего программного обеспечения.

Вместо алгоритмов хэширования Windows использует криптографию с открытым ключом. С помощью криптографии с открытым ключом цифровая подпись может быть сгенерирована с использованием закрытого ключа и проверена с помощью открытого. Это означает, что лицензионный ключ с цифровой подписью может быть проверен приложением без предоставления конфиденциальных ключей.

При генерации лицензионных ключей Windows использует много информации о программном обеспечении, включая, но не ограничиваясь этим:

- Разрядность (32, 64)
- Тип (домашний, профессиональный, корпоративный)
- Идентификатор продукта
- Аппаратные характеристики

Включение всей этой информации помогает привязать ключ продукта к конкретной установке программного обеспечения. Если вас интересует дополнительная информация о протоколе, в Интернете есть множество ресурсов, посвященных генерации ключей Microsoft.

Ключи с цифровой подписью
Цифровые подписи на лицензионных ключах, подобные тем, которые используются Windows, значительно затрудняют создание поддельных действительных ключей. Действительная подпись должна быть сгенерирована с использованием закрытого ключа, но может быть проверена с помощью нечувствительного открытого ключа.

Цифровые подписи препятствуют прямому генерированию лицензионных ключей и предоставляют злоумышленникам два варианта действий. Первый - утечка легитимного ключа, который можно отследить до конкретного пользователя. В качестве альтернативы злоумышленник может модифицировать программу, чтобы удалить код для проверки ключа, что увеличивает время и сложность пиратского использования программного обеспечения.

Наилучший способ
Представленные до сих пор примеры были сосредоточены преимущественно на автономной проверке лицензионных ключей, что означает, что весь код для проверки и разблокировки программного обеспечения находится в системе. Однако, учитывая широкую взаимосвязанность систем в наши дни, способ повысить их эффективность - добавить онлайн-компонент.

Это может принимать различные формы, но одна из них, которую вы видите сегодня, заключается в том, что каждая часть программного обеспечения может распространяться с лицензионным ключом в виде большого случайного числа, распространяемого вместе с программным обеспечением. Когда продукт установлен и зарегистрирован, это значение отправляется на сервер лицензий, который проверяет, что оно действительно и еще не использовалось. В наши дни при распространении цифрового программного обеспечения отправленный вам ключ недействителен даже после покупки программного обеспечения, а это означает, что если бы вы угадали этот ключ за 10 минут до покупки программного обеспечения, он бы не сработал.

Или вы можете использовать гибридные подходы, при которых большая часть алгоритма

проверки с помощью хэширования или криптографии с открытым ключом является резидентной в системе, но затем также выполняется шаг, на котором сервер лицензий проверяется, использовался ли этот ключ ранее или ключ был отозван.

Другие предложения
Представленные методы соответствуют лучшим отраслевым практикам и наиболее часто используемым методам. Но не существует универсального подхода к обеспечению безопасности, и некоторые из приведенных ниже методов вы можете встретить в сценарии взлома, или вы можете счесть их полезными в сценарии защиты, если у вас есть уникальные ограничения.

Предпочитаете автономную активацию
Хотя добавление онлайн-серверов ключей звучит мощно с точки зрения безопасности, и так оно и есть, стоит признать, что этот метод сопряжен с огромными трудностями в управлении и инфраструктуре. Управление серверами ключей - немалый подвиг, и они становятся маяком для кибератак. Таким образом, вы по-прежнему часто будете сталкиваться с тем, что многие компании не могут или не желают мириться с таким уровнем хаоса, поэтому они по-прежнему будут предпочитать более строгую автономную проверку. Поддержка автономной проверки ключей устраняет сложности управления сервером ключей и доступна пользователям без доступа в Интернет.

Выполните частичную проверку ключа
В автономном режиме у вас нет способа выполнить отзыв и нет возможности заставить некоторые ключи больше не работать. Чтобы предотвратить работу одного утекшего ключа во всех будущих версиях вашего программного обеспечения, проверьте только часть ключа. Упрощенным примером было бы проверить только первый символ каждой группы в лицензионном ключе, таком как X, 9, B и B в X4Z-951-B41-BR0.

Если кто-то выпустит генератор ключей для вашего приложения, выпустите новую версию, которая проверяет часть оставшегося ключа. Например, переключитесь на проверку второго символа каждой группы (4, 5, 4 и R). Это ограничивает потенциальный ущерб, наносимый одним генератором ключей.

Закодируйте полезные данные в ключе
Кодирование полезных данных в ключе может помочь ограничить его применимость. Например, ключ может указывать максимальную версию приложения, к которому он применяется, ограничивая влияние скомпрометированного ключа.

Генераторы ключей
Если часть программного обеспечения использует ключ для активации, взломщики захотят создать для нее генератор ключей. Это верно независимо от того, какой тип активации ключа вы выполнили. Генераторы ключей затем распространяются среди людей для создания “бесплатного” ключа для программного обеспечения.

Позже вы увидите, как исправить программное обеспечение, чтобы просто удалить проверку ключа, поэтому сейчас сосредоточьтесь на создании генератора ключей и предположите, что вы не можете просто обойти проверку ключа. Генераторы ключей обычно требуют более глубокого анализа программы и более глубокого понимания алгоритма ключа.

Зачем создавать генераторы ключей?
Если генераторы ключей сложнее создавать, зачем утруждать себя их сборкой? Есть несколько разных причин.

Программное обеспечение может иметь различные средства защиты, которые могут затруднить внесение исправлений, например следующие:

- Защита от несанкционированного доступа
- Динамические проверки
- Защита от отладки
- Защита программного обеспечения

Кроме того, для исправления может потребоваться выпустить измененную копию целевого программного обеспечения, на которую могут быть нанесены водяные знаки. Нанесение водяных знаков - это метод отслеживания программного обеспечения до того, кто его первоначально приобрел. Эти водяные знаки могут быть использованы для отслеживания взломанного программного обеспечения вплоть до взломщика, чего, очевидно, они не хотят.

Программное обеспечение могло бы осуществлять онлайн-проверки для поиска исправленных/модифицированных версий программ. Альтернативно, некоторое программное обеспечение может расшифровать само себя на основе введенного ключа (распаковка, которая будет рассмотрена в главе 13, “Продвинутые методы защиты”), и полное удаление проверки ключа означает, что оно не сможет расшифровать.

Генераторы ключей также более надежны в будущем, чем исправление. Разработчик приложения не может легко отозвать действительные ключи.

Наконец, взломщики могут выбирать генераторы ключей, потому что они сложнее. Исправление в некоторых случаях легко, в то время как создание успешного кейгена - задача, которая несет в себе определенную долю престижа.

Философия генерации ключей

При взломе средств проверки ключей полезно представлять средство проверки ключей в виде f(u) == g(k), где:

u - имя пользователя, введенное пользователем.
f - функция преобразования имени пользователя.
k - ключ, введенный пользователем.
g - это функция преобразования ключа.

В этой модели проверка ключа - это подтверждение того, что f(u) == g(k). На нематематическом языке это означает, что для имени пользователя выполняется некоторое преобразование/мутация, а затем сравнивается с некоторым типом преобразования, выполненным для ключа. В этом примере (и в следующих примерах) входными данными является имя пользователя, но имейте в виду, что это может быть любая комбинация параметров; они могут использовать номер версии, имя компьютера и т.д. Но идея в том, что что-то преобразуется, чтобы получить результат. И этот результат сравнивается с входным ключом, который также подвергся некоторому преобразованию (обратите внимание, что это преобразование может быть ничем, что означает, что результатом является просто ключ, или это может быть больше хэширования или мутации). Учитывая эту модель, существует несколько потенциальных вариантов проверки ключей.

Возвращаясь к первоначальному примеру StarCraft/Half Life, u на самом деле будет первыми 12 цифрами ключа, а k - последней цифрой. В этой настройке имя пользователя не вводится; скорее, часть ключа используется для проверки другой части.

Другой вариант заключается в том, что u, и, следовательно, f(u), является константой (т.е. жестко закодированными ключами). В этой настройке не вводится имя пользователя; скорее, ключ преобразуется и сверяется с фиксированным значением. Например, “сумма всех цифр в ключе равна 1337.”

Взлом различных типов проверок ключей
Рассуждая о средствах проверки ключей в формуле f(u) == g(k), вы можете начать разрабатывать методы для взлома различных перестановок.

Тип проверки ключей I: Преобразуйте только имя пользователя
В этом случае имя пользователя преобразуется с помощью некоторой функции, а затем сравнивается с введенным ключом. Итак, в этом случае вы можете считать, что g() не вызывает мутации ключа. Это позволяет нам упростить нашу проверку ключа до простого f(u) == k. В этой настройке программа преобразует имя пользователя и проверяет, соответствует ли преобразованное значение ключу, введенному пользователем.

Чтобы взломать этот тип, найдите и извлеките функцию преобразования f в приложение для генерации ключей. Например, умножьте порядковые номера символов в username вместе и сопоставьте с ключом. Генератор ключей запросит у пользователя имя пользователя, которое он желает использовать, а затем выполнит f(u) и распечатает действительный ключ.

Проверка ключа II типа: Преобразуйте оба
Для типа II у вас все еще есть преобразование имени пользователя, но, кроме того, g выполняет мутацию. Самый математический способ взглянуть на это заключается в том, что g имеет обратную величину. То есть, g-1 существует, и g-1(g(k)) == k). Простой способ подумать об этом заключается в том, что g выполнит мутацию, и у каждой мутации есть способ отменить ее (т.е. сделать прямо противоположное).

В этой настройке программа преобразует имя пользователя, преобразует введенный ключ и проверяет, что они дают одинаковые результаты. Однако функция g может быть инвертирована (“отменена” или “отменено резервное копирование”).

Чтобы взломать этот тип проверки ключа, перепроектируйте g и выведите g-1. Часто это так же просто, как “отменить” каждое преобразование в g в обратном порядке. Затем сгенерируйте ключ с g-1(f(u)).

Например, предположим, что g(k) = k * 2 + 1000. Если это так, то g-1(h) = (h – 1000) / 2.

В этом случае генератор ключей запросит желаемое имя пользователя (как в случае с типом I) и выполнит f(u), но тогда результатом теперь будет измененный ключ, поэтому вам придется выполнить развертывание с помощью g-1(h). Этот конечный результат и есть действительный ключ.

Проверка ключа III типа: С применением грубой силы
В типе III коллизия на f(u) может быть грубо обработана через g(k). Это жизнеспособный подход, если пространство для ключей очень мало или у вас большая вычислительная мощность.

В этой настройке программа преобразует имя пользователя, преобразует введенный ключ и проверяет, что оба они дают одинаковые результаты (такие же, как тип II). Но вместо этого вы ищете решение для f (u) == g (k) путем многократного тестирования случайных или псевдослучайных ks.

Чтобы взломать этот тип, определите формат k. Затем извлеките g в автономный генератор ключей. Наконец, генерируйте случайные k, пока не будет найдено решение для f(u) == g(k).

Например, рассмотрим случай, когда g(k) = CRC32(k). Если для мутации ключа используется что-то настолько маленькое, например алгоритм CRC32, то грубая сила становится довольно тривиальной на стандартном компьютере. Поскольку CRC32 имеет такой небольшой диапазон возможных значений, можно использовать грубую силу.

Защита от кейгенов
Проверки ключей могут быть комбинацией этих типов. Например, преобразование ключа g может быть как принудительным, так и обратимым.

Проверки ключей, как правило, должны относиться к одной из этих категорий. В противном случае не было бы возможности генерировать ключи в первую очередь.

Проверка ключа I типа является самой слабой. Взломщику нужно только извлечь алгоритм из средства проверки ключей, без необходимости фактически изменять алгоритм.

Проверка ключа III типа лучше. Это требует, чтобы злоумышленник извлек оба алгоритма и определил способ грубой силы преобразования ключа, что не всегда очевидно.

Тип проверки ключа II, вероятно, является лучшим, но также и самым сложным в разработке. Для взлома этого требуется, чтобы злоумышленник получил функцию, обратную функции преобразования ключа. Это может потребовать глубокого анализа алгоритма преобразования, что замедлит атаку.

Как всегда, нет худа без добра. Любое средство проверки ключей в конечном итоге может быть взломано, и лучшее, что может сделать защитник, - это замедлить атакующего.

Лабораторная работа: Вводный кейген
В этой лабораторной работе дается опыт создания кейгена для простой программы.

Лабораторные работы и все связанные с ними инструкции можно найти в соответствующей папке здесь:

https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES

Для этой лабораторной работы, пожалуйста, найдите вводный кейген и следуйте предоставленным инструкциям.

Навыки
В этой лабораторной практикуется использование objdump и утилиты strings для генерации кейгена. Некоторые из ключевых навыков, которые тестируются, включают следующее:

- Первоначальная разведка
- Обратное проектирование x86
- Генерация ключей

Блюда на вынос
Помимо модификации программы, часто можно взломать программу, просто наблюдая за тем, как она работает. Правильный подход часто определяется ограничениями программы, и

выбор того, что использовать, является важным навыком.

Procmon
При реверс-инжиниринге вы хотите узнать как можно больше о том, как работает программа. Прежде чем переходить к сверхсложной отладке, начните с простого - просто понаблюдайте за поведением программного обеспечения.

Procmon - это инструмент, распространяемый как часть набора инструментов Sysinternals (доступен по адресу http://technet.microsoft.com/en-us/sysinternals/bb842062). Этот репозиторий содержит около 60 утилит для Windows, созданных и свободно распространяемых корпорацией Майкрософт. Обратите внимание, что эти инструменты работают только в ОС Windows.

Пример: Notepad.exe
Попробуйте взглянуть на то, что делает notepad.exe, когда вы создаете новый файл, меняете шрифт, а затем сохраняете часть содержимого. Для этого выполните следующие действия:

- Откройте Procmon.exe.
- Запустите блокнот.
- Введите некоторый текст в документ блокнота.
- Выберите меню Формат, а затем пункт меню Шрифт.
- В окне Шрифт измените шрифт на Webdings.
- В окне Шрифт измените размер на 20.
- Нажмите кнопку ОК.
- Сохраните документ блокнота как Example1.txt.
- Закройте блокнот.

Остановите отслеживание активности Process Monitor, нажав кнопку захвата, как показано на рисунке 10.1. Теперь значок должен отображать крестик над увеличительным стеклом. На данный момент Process Monitor зафиксировал все события файлов, реестра и процессов/потоков.

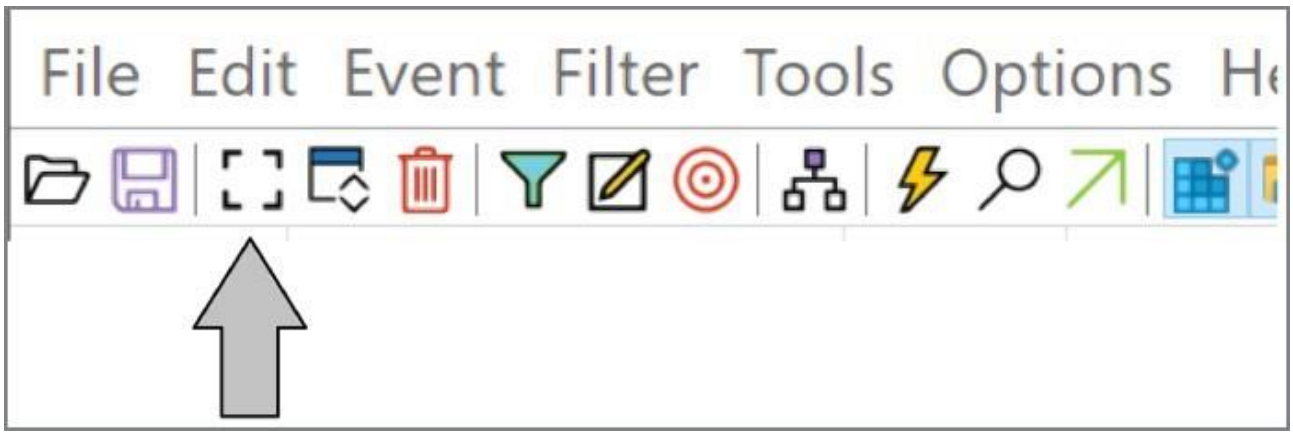


Рисунок 10.1: Остановка Process Monitor

Process Monitor фиксирует тысячи событий в секунду, что приводит к слишком большому количеству записей для просмотра вручную. Необходимо отфильтровать результаты до интересующих событий. Для этого откройте меню Фильтра, щелкнув значок воронки, как показано на рисунке 10.2.

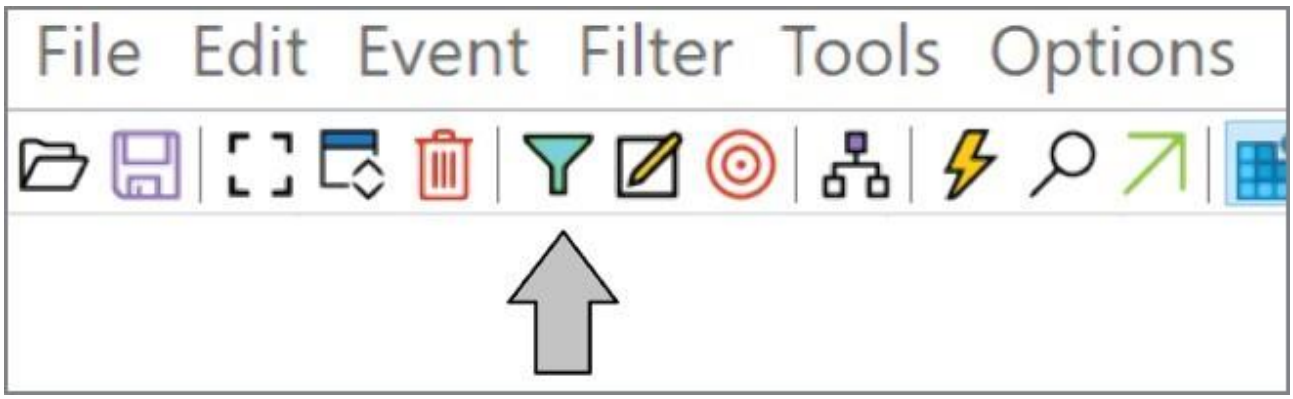


Рисунок 10.2: Фильтрация событий в Procmon

Чтобы видеть только события, связанные с процессом Notepad.exe, задайте фильтр, указывающий, что имя процесса равно Notepad.exe, как показано на рисунке 10.3. Вы можете выполнить это с помощью следующих шагов:

- Выберите Имя процесса в поле списка столбцов.
- Выберите is в поле список отношений.
- Введите Notepad.exe в текстовом поле Значение.
- Выберите Включить в поле список действий.
- Нажмите кнопку Добавить.
- Нажмите Применить и ОК.

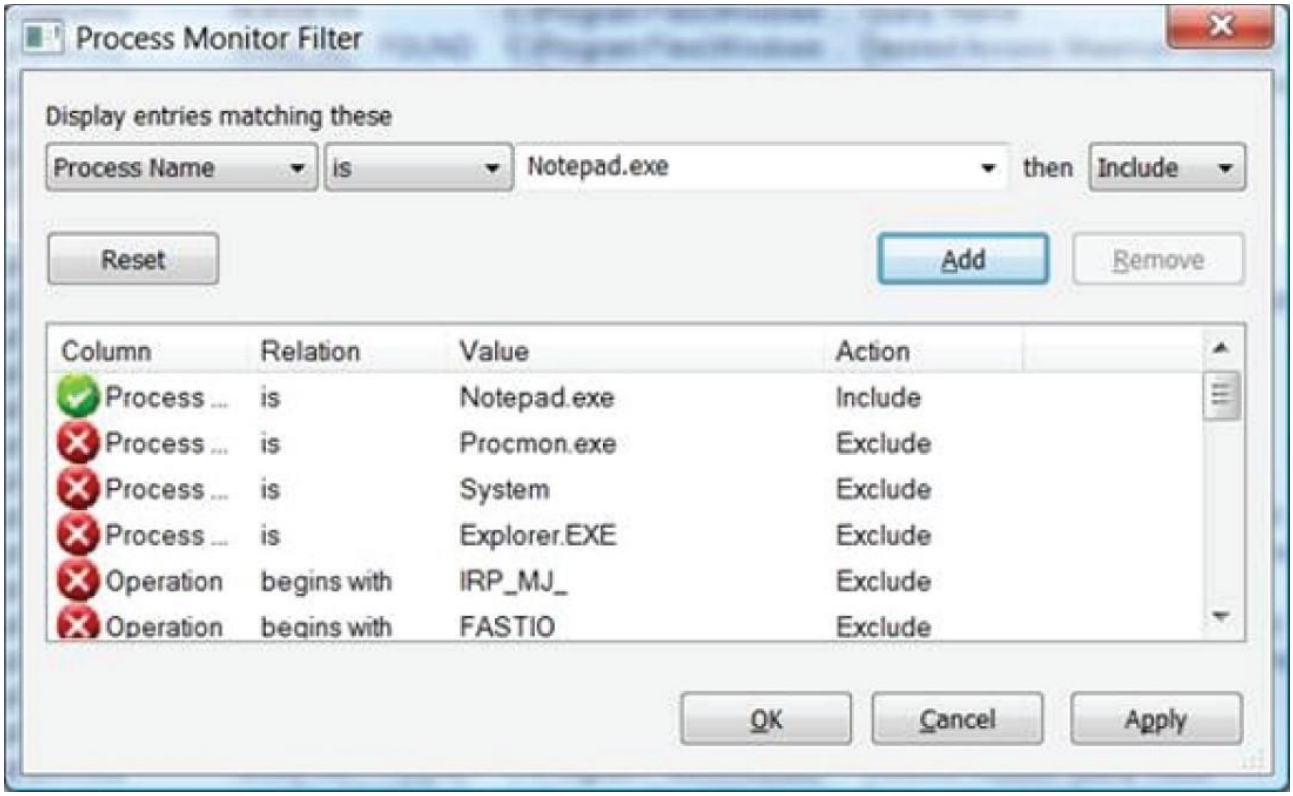


Рисунок 10.3: Определение фильтра в Procmon

Фильтрация на основе имени процесса значительно уменьшает количество событий. Однако этого все равно недостаточно.

Чтобы найти интересующие события, вам необходимо определить дополнительные фильтры. В Procmon есть несколько категорий событий, которые вы можете отфильтровать, включая следующие:

- Реестр
- Файл
- Сеть
- Поток процесса

Для начала попробуйте сосредоточиться на значениях реестра, которые изменяет Notepad. В Process Monitor есть удобная кнопка для этого, как показано на рисунке 10.4.

Рисунок 10.4: Фильтрация по событиям реестра в Procmon

Если Блокнот сохранил значения в реестре, он создаст запись о событии типа 'Operation' 'RegSetValue'. Щелкнув правой кнопкой мыши записи в журнале Procmon, вы можете включить или исключить определенные типы событий, как показано на рисунке 10.5. Это позволяет дополнительно уточнить результаты и сосредоточиться на интересующих событиях.

На рисунке 10.6 показана запись Procmon, которая, по-видимому, связана с изменениями шрифта в Блокноте. Чтобы просмотреть дополнительную информацию, щелкните запись правой кнопкой мыши и выберите "Свойства".

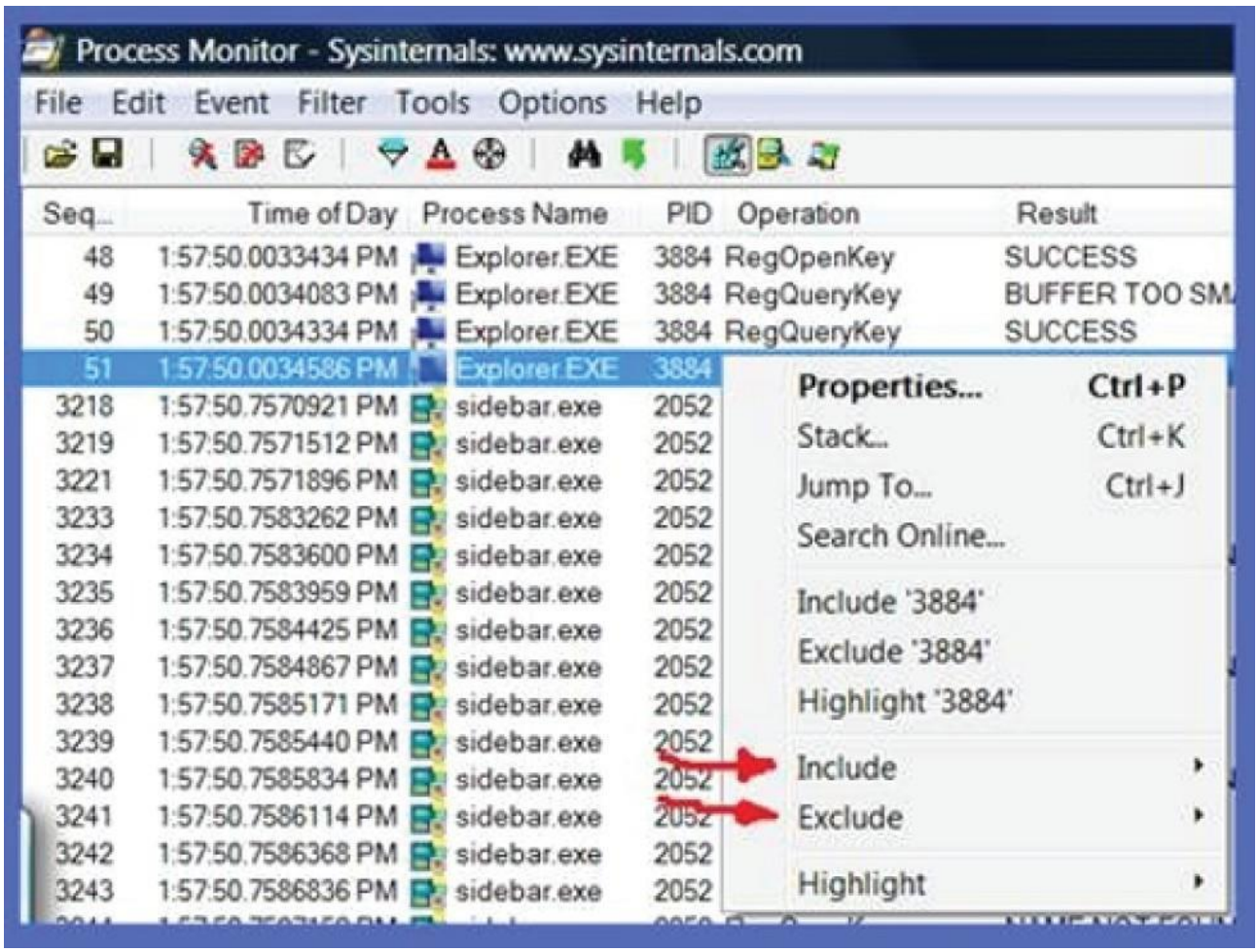


Рисунок 10.5: Включение и исключение категорий событий в Procmon

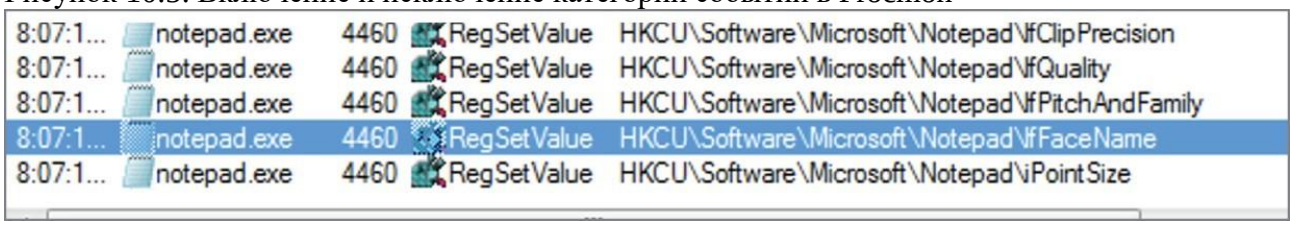


Рисунок 10.6: Событие реестра изменения шрифта блокнота

На рисунке 10.7 показаны свойства события. В поле данных вы можете увидеть текст “Webdings”, указывающий, что это событие вызвано изменением шрифта блокнота на Webdings.

Как Procmon помогает в восстановлении и взломе.
Procmon позволил увидеть изменения в реестре, внесенные Notepad. Однако это не все, что он может сделать. Дальнейшее изучение инструмента открывает много полезной информации.

Стеки вызовов
В окне свойств события есть несколько разных вкладок. Переход на вкладку Stack показывает последовательность вызовов, использованных для достижения этой точки, как показано на рисунке 10.8.

Просматривая трассировку стека дальше, можно увидеть точку, в которой программа

завершила работу notepad.exe, как показано на рисунке 10.9. Эта точка перехода от приложения к библиотекам может быть хорошей отправной точкой для обращения вспять.

Файловые операции

Procmon также записывает события для файловых операций, таких как открытие, закрытие и редактирование файлов. На рисунке 10.10 показан пример этого.

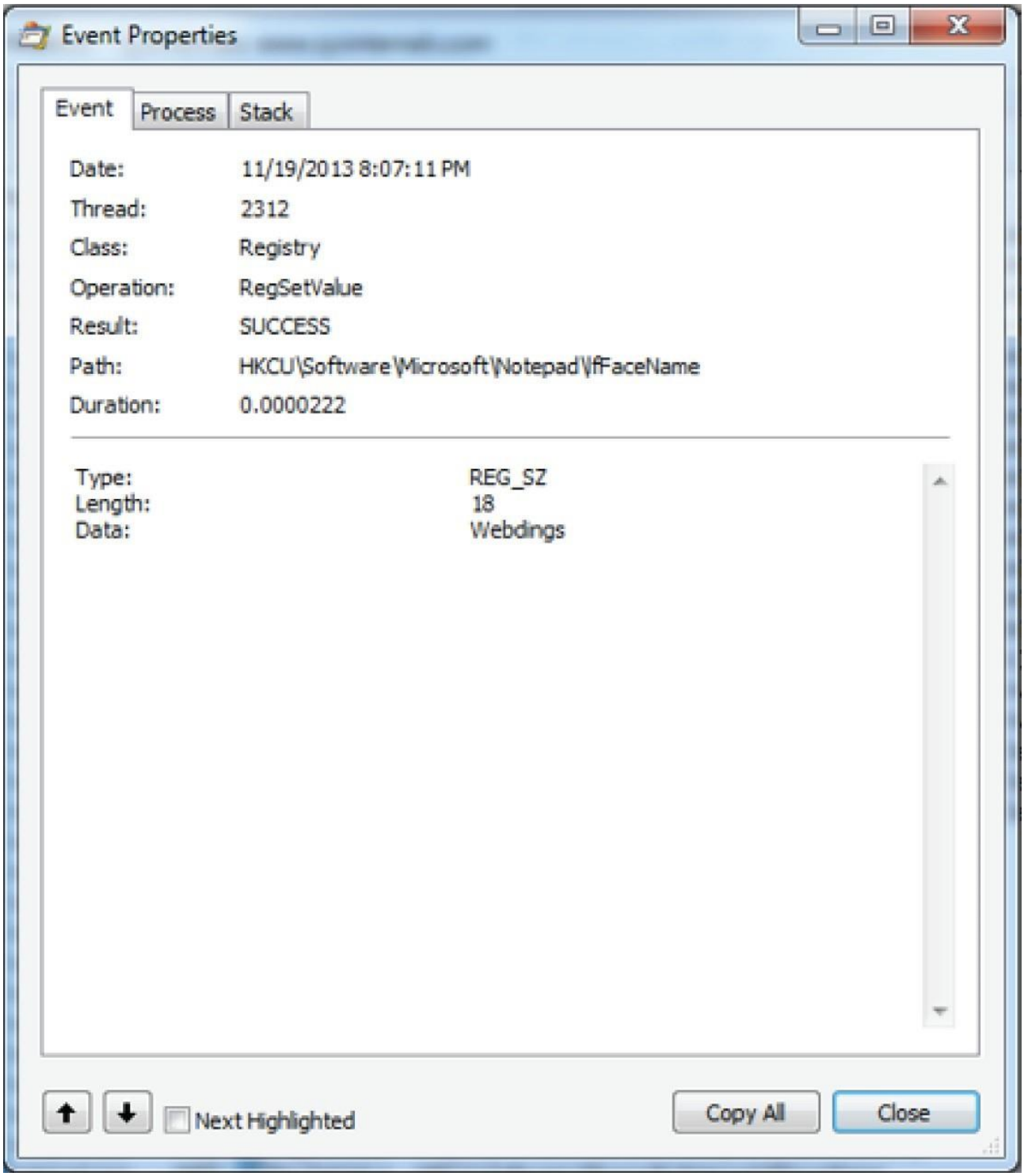


Рисунок 10.7: Свойства события в Procmon

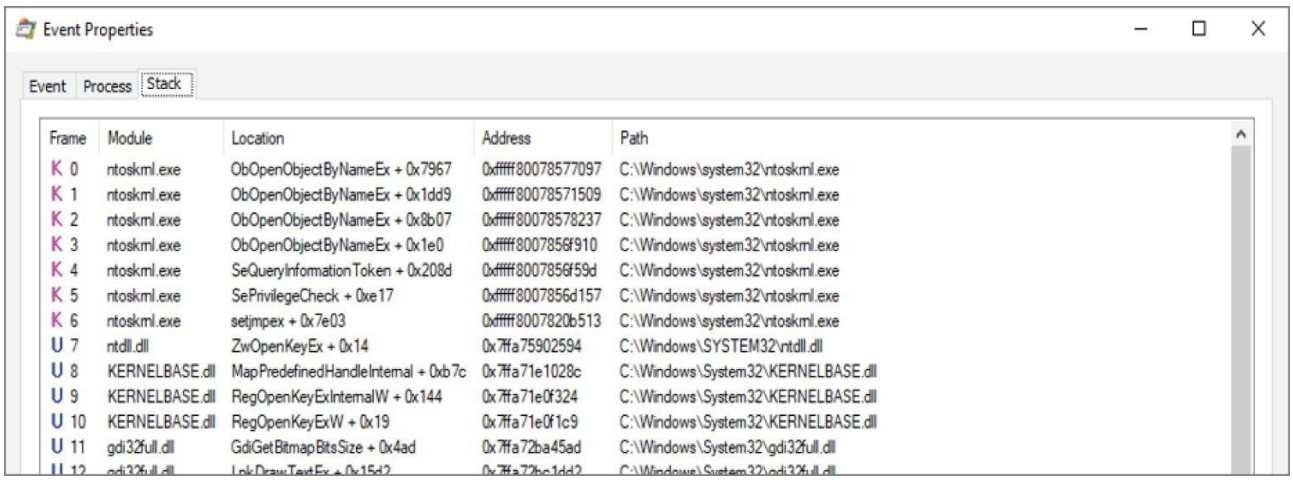


Рисунок 10.8: Представление стека в окне свойств Procmon

Эти файловые события могут предоставить полезную информацию для реверсирования. Например, они могут помочь в идентификации и анализе файлов конфигурации, функций экспорта и проприетарных форматов файлов.

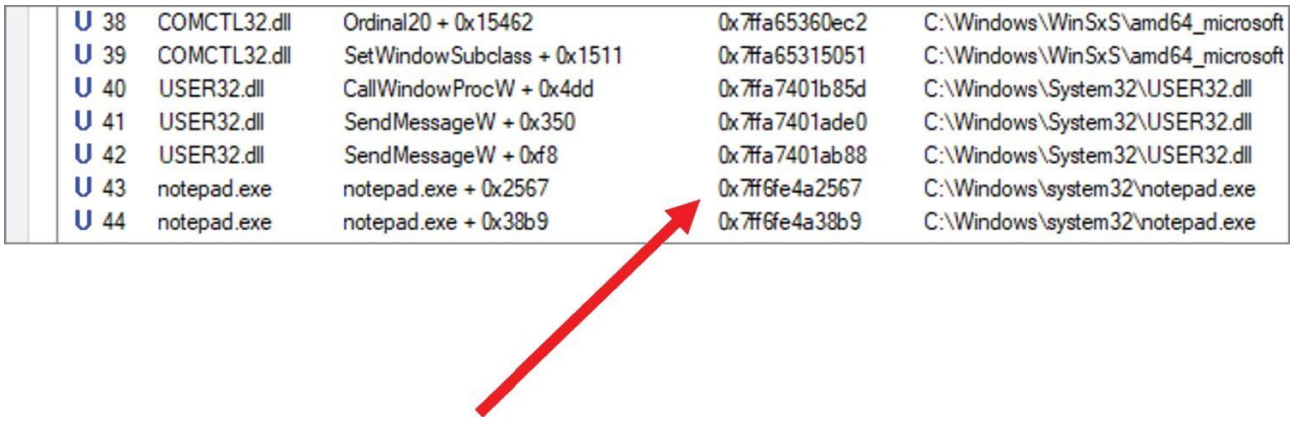


Рисунок 10.9: Трассировка стека для notepad.exe

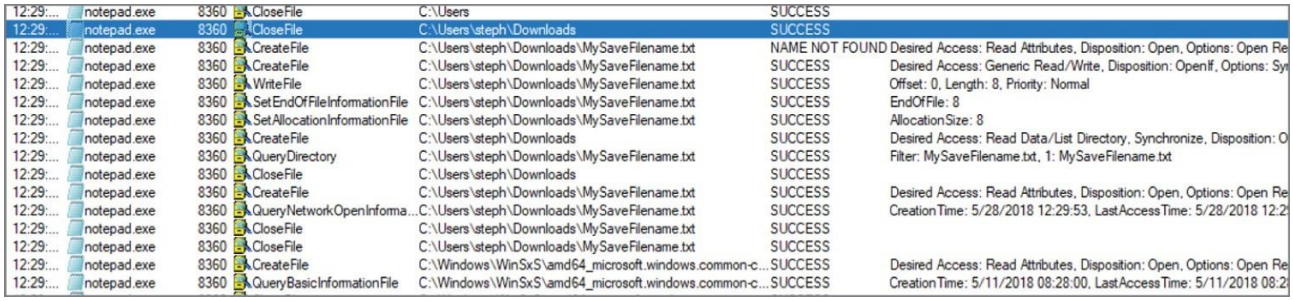


Рисунок 10.10: Файловые операции в Procmon

Запросы реестра

В примере Notepad.exe показано, как найти операцию реестра для изменения шрифта в Блокноте. Однако это не единственное возможное применение запросов реестра.

Например, на рисунке 10.11 показано, что Notepad искал два ключа со словом “Безопасность”

в них, но не смог их найти. Вы могли бы добавить эти ключи в свой реестр и поместить в них пользовательские значения, чтобы изменить работу Notepad.

Рисунок 10.11: Запросы к реестру безопасности в Procmon

Resource Hacker
Resource Hacker (также известный как ResHacker или перерешать) - это бесплатная утилита извлечения или компилятор ресурсов для Windows. Resource Hacker можно использовать для добавления, изменения или замены большинства ресурсов в двоичных файлах Windows, включая строки, изображения, диалоговые окна, меню, а также ресурсы VersionInfo и Manifest. (Ссылки на инструменты можно найти в разделе инструменты нашего сайта GitHub по адресу https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES.)

Resource Hacker может быть полезным инструментом для изучения структуры двоичного файла перед процессом взлома. Его можно использовать для поиска и понимания структуры экранов nag, экранов ввода клавиш, меню справки и многого другого.

Resource Hacker также можно использовать для добавления функциональности в программу до или после взлома. Например, в существующее приложение можно добавить новые значки, меню и скины.

Для начала откройте exe-файл в ResHack, чтобы изучить его строки, изображения, диалоговые окна, меню и т.д., как показано на рисунке 10.12. Затем щелкните элемент в ResHack (слева), чтобы показать, как этот элемент будет выглядеть в приложении (справа).

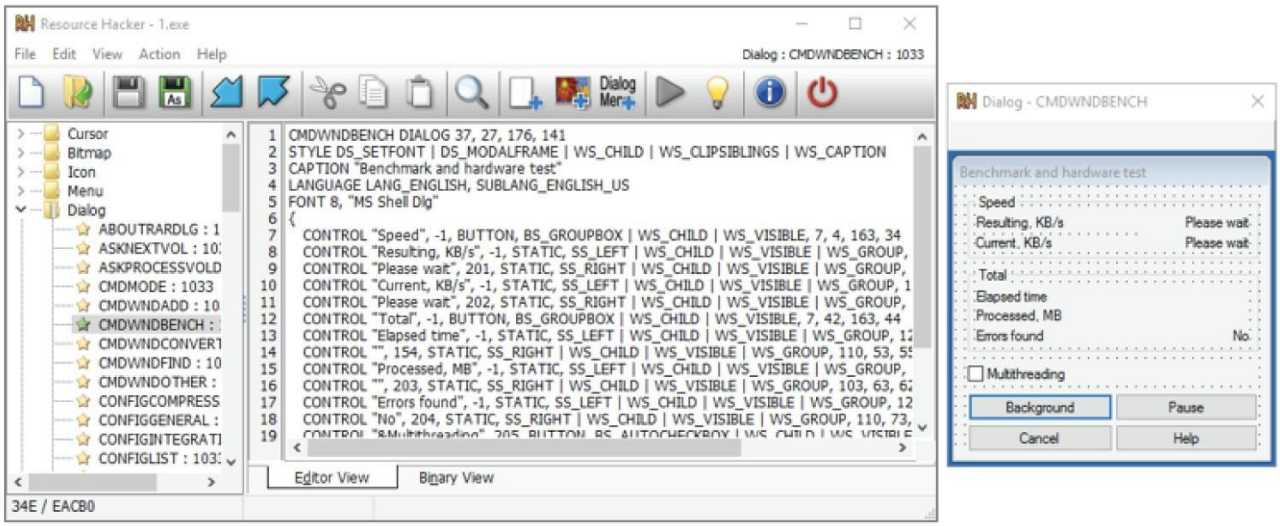


Рисунок 10.12: Пример приложения в Resource Hacker

Пример
Предположим, вы видите окно, показанное на рисунке 10.13 в программе. Как взломщик, вы хотите понять, как это окно будет использоваться программой.

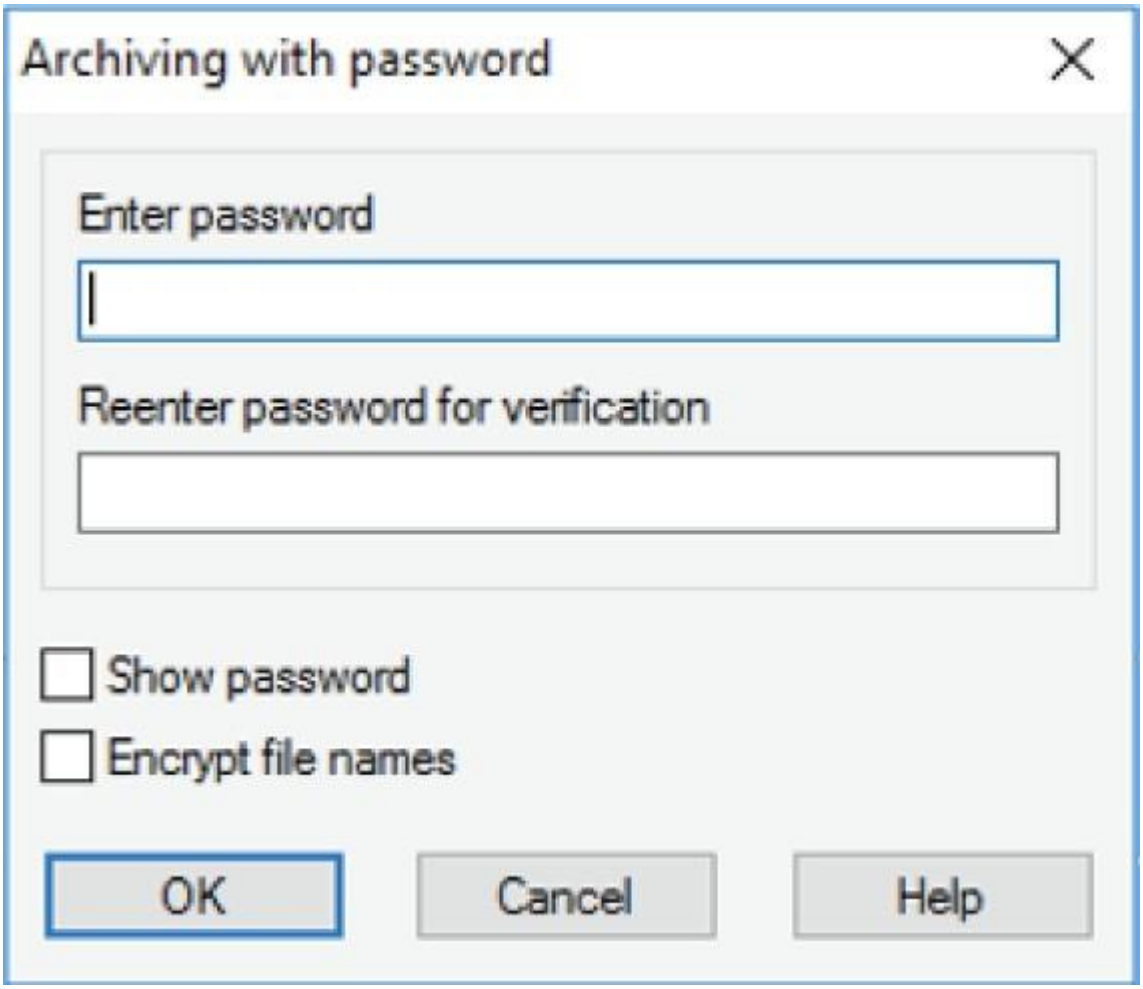


Рисунок 10.13: Окно ввода пароля

Чтобы узнать это, откройте программу в ResHack. Затем используйте Ctrl+F для поиска одной из строк, используемых в диалоговом окне, как показано на рисунке 10.14.

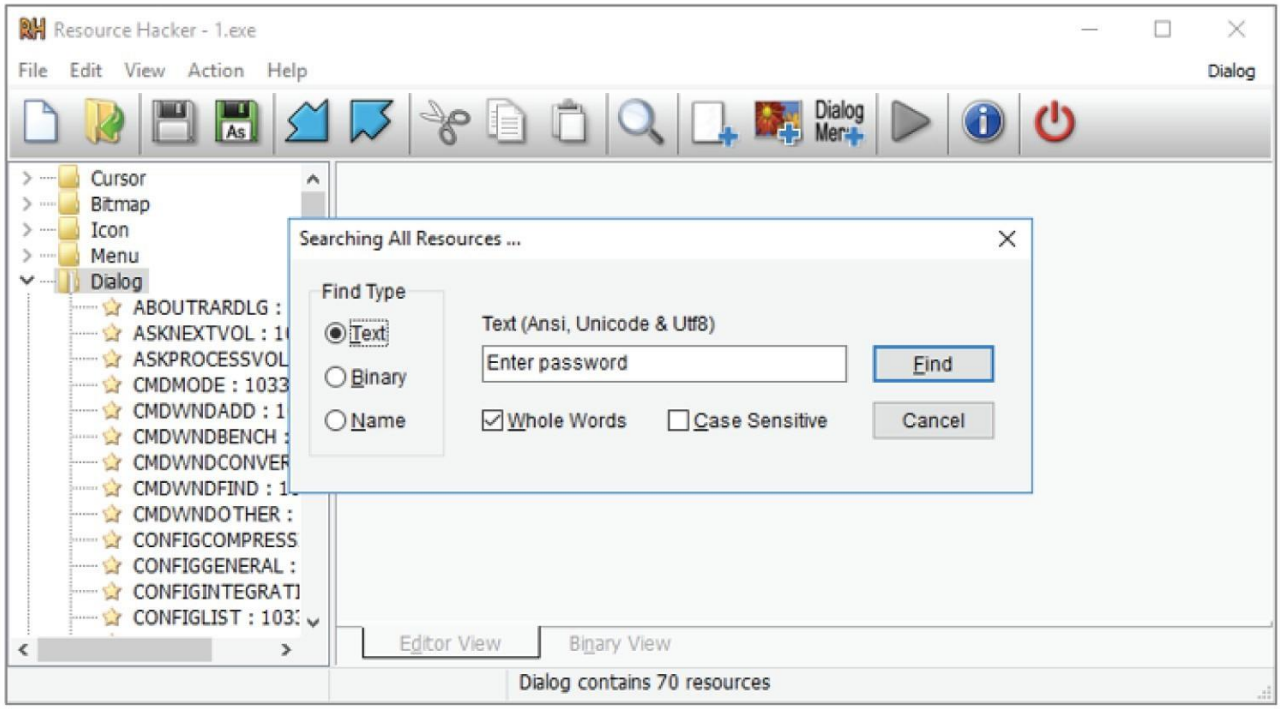


Рисунок 10.14: Поиск по строкам в Resource Hacker

Resource Hacker идентифицирует это диалоговое окно как диалоговое окно “GETPASSWORD2”, как показано на рисунке 10.15. Знание этого может помочь управлять процессом реверсирования программы.

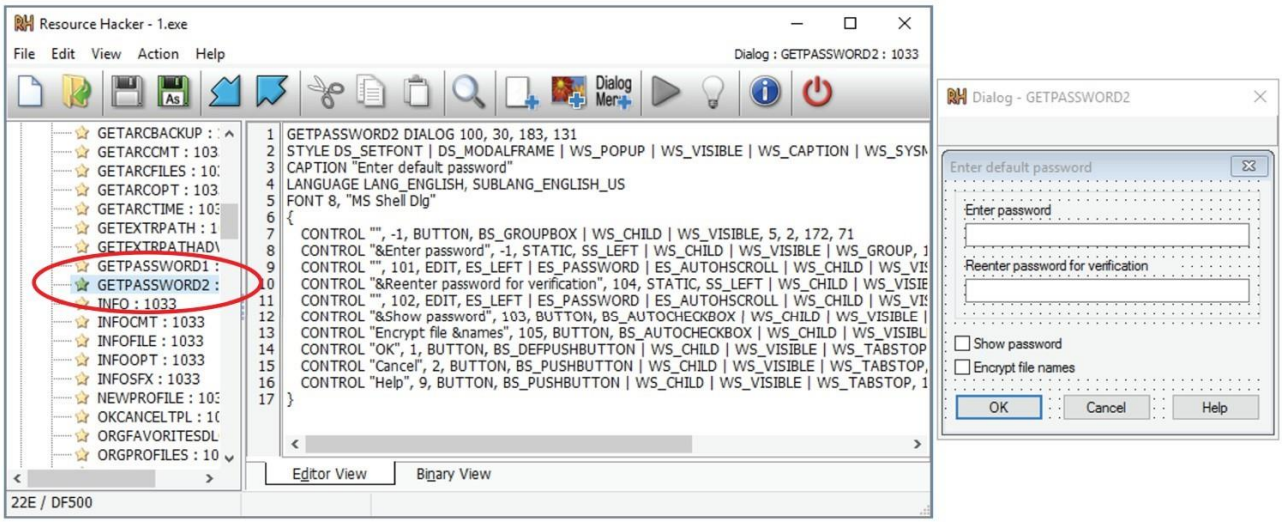


Рисунок 10.15: Идентификация диалогового окна в Resource Hacker

Мини-лаборатория: Калькулятор Windows
Чтобы попробовать свои силы в использовании Resource Hacker, попробуйте переименовать калькулятор Microsoft. Как показано на рисунке 10.16, окно калькулятора называется Calculator. Попробуйте изменить это значение на что-нибудь другое.

Для начала откройте calc.exe исполняемый файл в Resource Hacker. Затем найдите слово Calculator, как показано на рисунке 10.17.

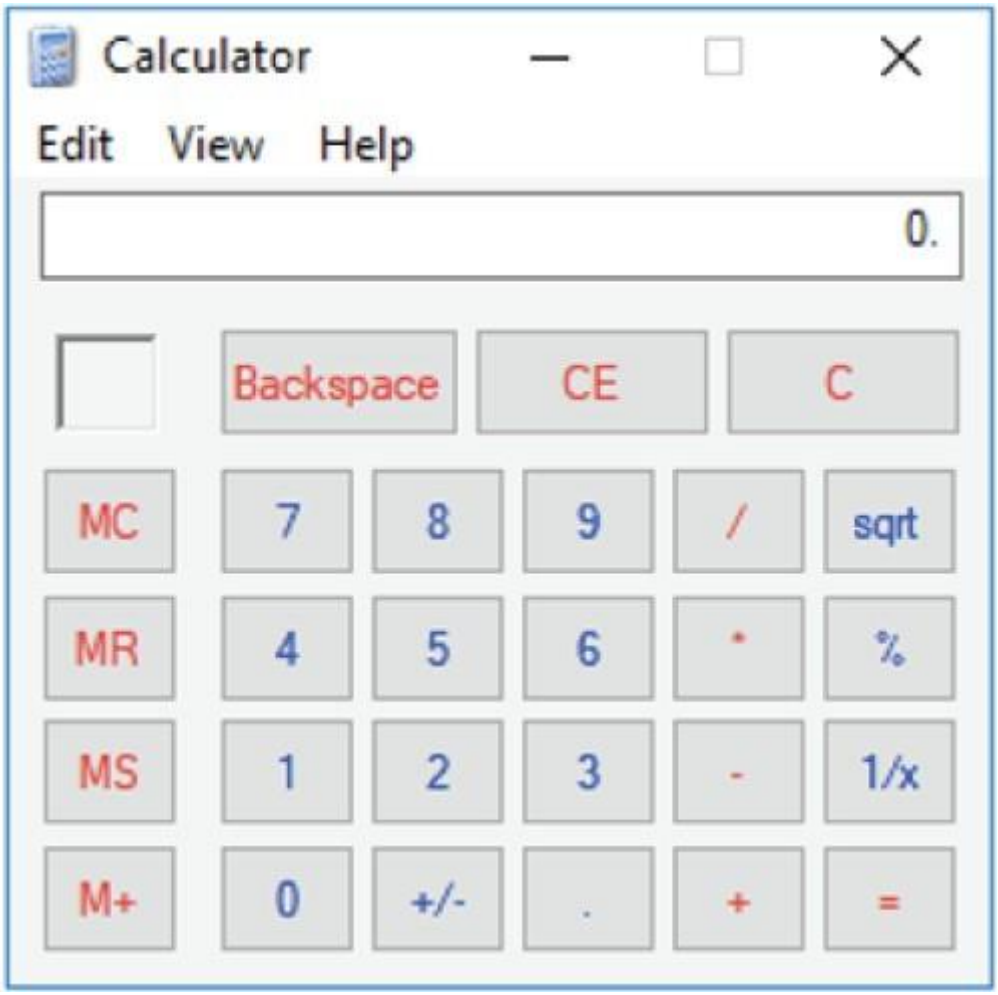


Рисунок 10.16: Калькулятор Microsoft

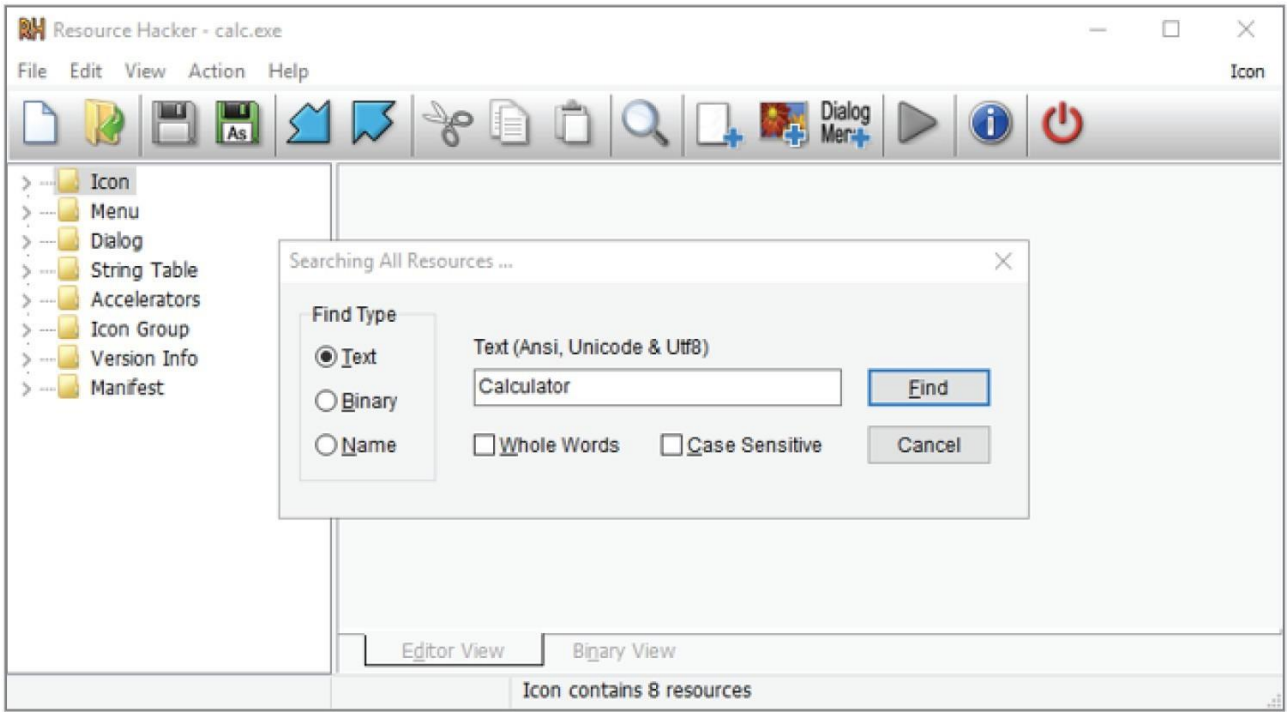


Рисунок 10.17: Поиск калькулятора в ResHack

Главное окно калькулятора может быть не первым результатом. Продолжайте поиск, пока не найдете код, определяющий диалоговое окно калькулятора, как показано на рисунке 10.18.

На рисунке 10.18 строка ЗАГОЛОВКА определяет заголовок в окне приложения. Измените эту строку, чтобы переименовать приложение как свое собственное.

После изменения ЗАГОЛОВКА нажмите зеленую кнопку со стрелкой, показанную на рисунке 10.19. Это скомпилирует измененное приложение калькулятора.

После того, как приложение будет скомпилировано, обновленная версия окна должна быть показана в окне предварительного просмотра. Это должно включать измененный заголовок, как показано на рисунке 10.20.

Компиляция приложения автоматически не сохраняет измененную версию. Для этого выберите Файл ➪ Сохранить, как показано на рисунке 10.21.

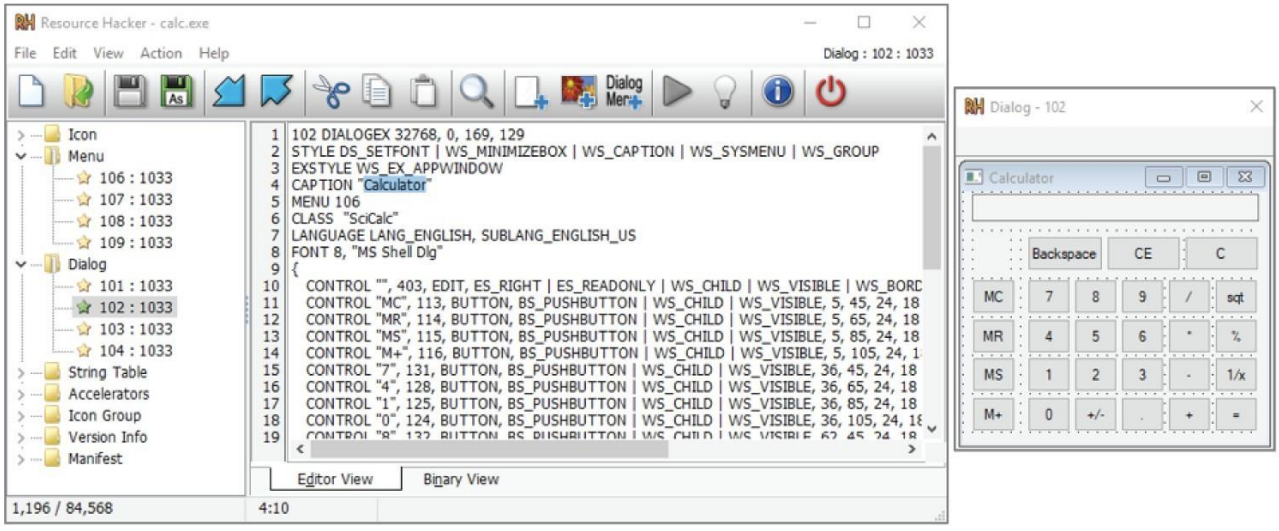


Рисунок 10.18: Окно калькулятора в Resource Hacker

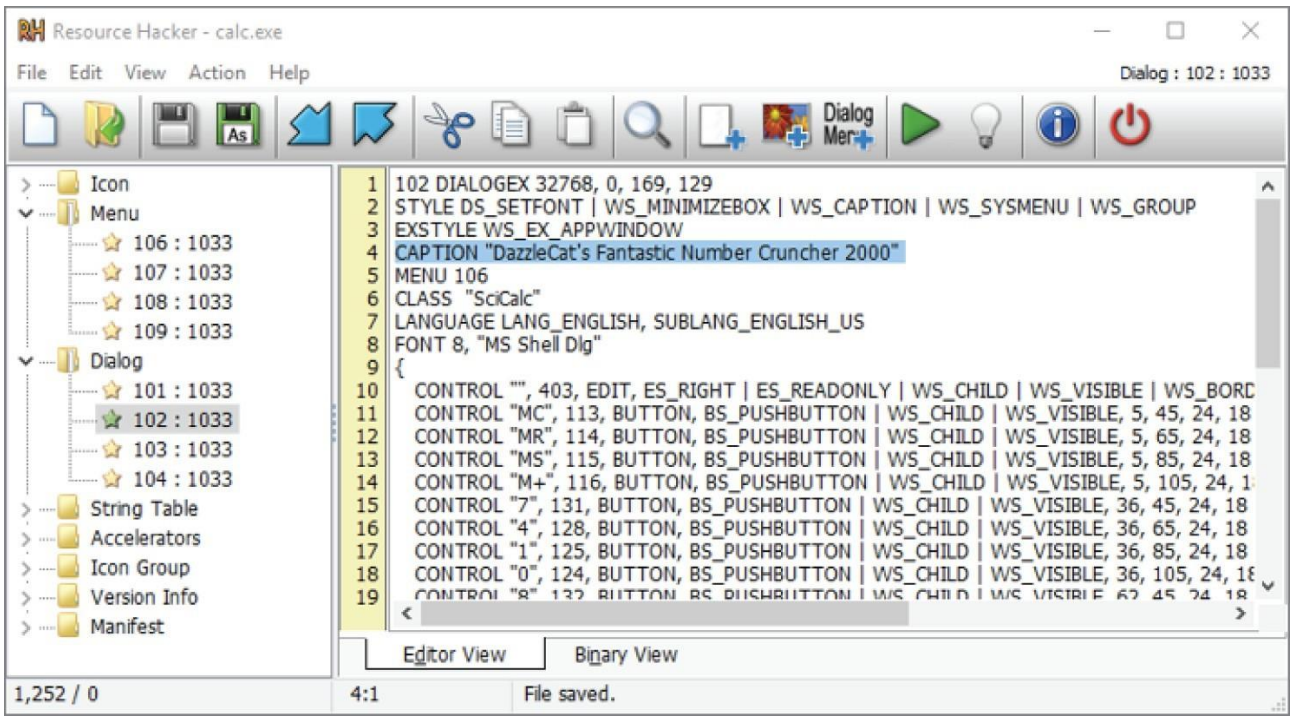


Рисунок 10.19: Компиляция измененного приложения

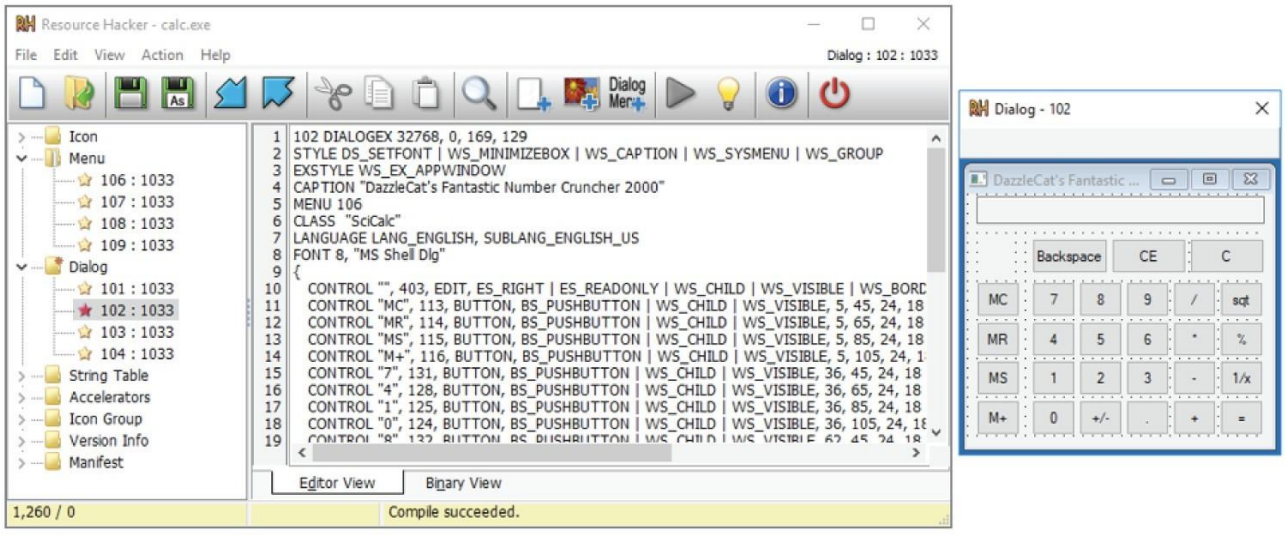


Рисунок 10.20: Измененное окно в Resource Hacker

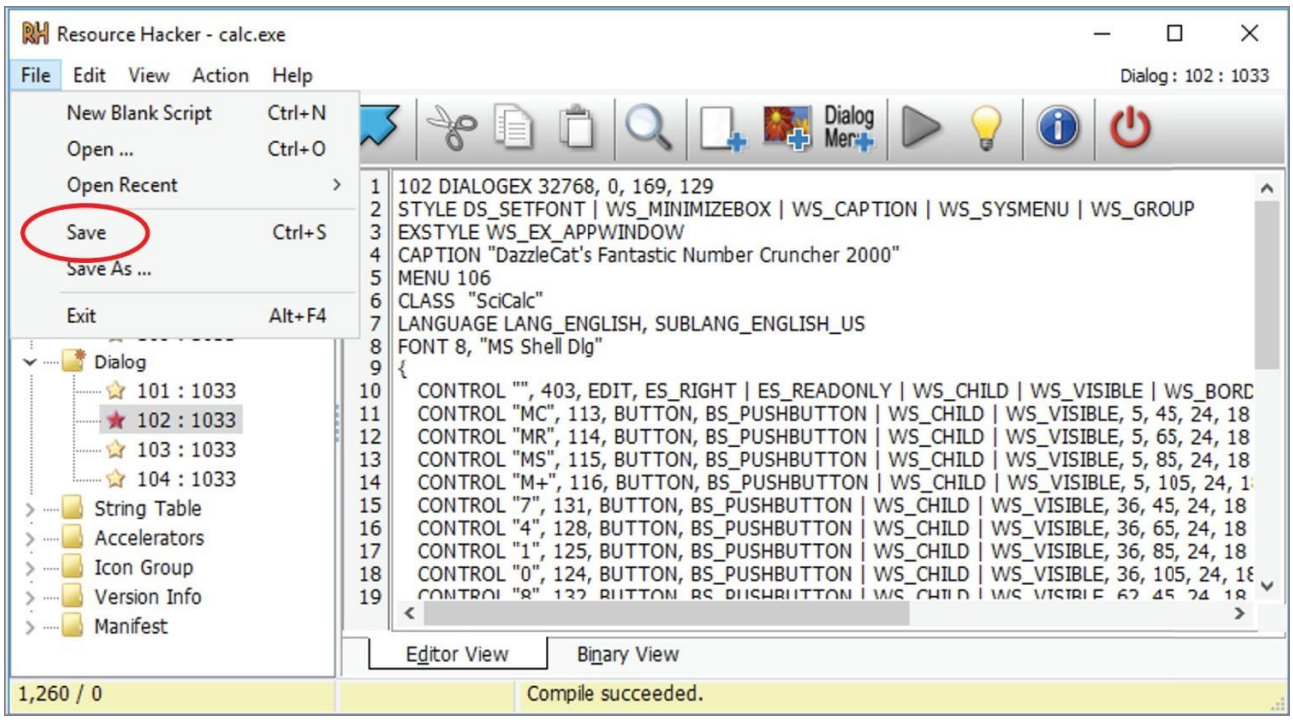


Рисунок 10.21: Сохранение измененного приложения в ResHack

На данный момент вы успешно переименовали Windows Calculator. Для более сложной задачи попробуйте следующее:

- Используйте Resource Hacker, чтобы изменить размер окна в соответствии с вашим новым именем.
- Измените доступные кнопки.
- Измените фоновый режим калькулятора.
- Открывайте и редактируйте другие программы в своей виртуальной машине.

Патчинг

Патчинг включает в себя изменение скомпилированного двоичного файла для изменения кода, влияющего на его выполнение. В зависимости от ситуации иногда проще всего пропатчить приложение, чтобы обойти его защиту.

Патчинг против Генерации ключей

В некоторых случаях расширенные проверки целостности или запутывание могут затруднить внесение исправлений. Например:

- Патчить зашифрованную/упакованную программы на диске невозможно.

- Внесение исправлений вокруг динамических проверок целостности (например, постоянно проверяемых контрольных сумм) может оказаться слишком громоздким.
- Логистика распространения исправленного исполняемого файла может оказаться нежелательной.

В таких ситуациях вы можете вместо этого прибегнуть к генераторам ключей. В противном случае, исправление программы для удаления ее проверок ключей (или любой другой логики, которой вы хотите избежать) часто является более простым подходом, когда это возможно.

Где патчить

Патчинг может выполняться в двух разных местах: в памяти или на диске.

Внесение исправлений в память изменяет машинный код в памяти. Это полезно при попытках обратного проектирования, потому что вам, возможно, придется попробовать десятки (или сотни... или больше... ахнуть) вещей, прежде чем что-то заработает. Внесение исправлений в памяти влияет только на текущее выполнение приложения. Каждый раз, когда вы перезапускаете приложение, все исправления, внесенные в память, будут потеряны.

Исправление на диске изменяет машинный код в скомпилированном двоичном файле. Это полезно, когда вы знаете, что работает и влияет на все будущие выполнения приложения. Это делает изменения постоянными и они будут присутствовать при каждом запуске приложения.

NOPs
Напомним инструкцию nop. Это однобайтовая инструкция (0x90), которая ничего не делает.

При исправлении приложений важно не перемещать код. Фактически, изменение размера или простое удаление кода приведет к аварийному завершению работы приложения. Чтобы удалить разделы кода, сохранив при этом прежний размер, заполните пробел символами nops.

Для тех из вас, кому интересно, почему простое удаление кода не работает, на это есть много факторов, но наиболее важным является то, что часть кода x86 является относительной, а часть - абсолютными ссылками. Сначала рассмотрим относительный регистр: это означает, что некоторый код преобразуется в относительные вещи, такие как “перейти вперед на 40 байт с того места, где я сейчас нахожусь”. В подобных случаях, если вы удаляете код между переходом и его пунктом назначения на расстоянии 40 байт, вы испортили переход. Он продолжит перескакивать на 40 байт вперед, за исключением того, что теперь он может приземлиться в середине кода операции или пропустить важные инструкции, что затем приведет к сбою. Если код, который вы удаляете, находится за пределами этого 40-байтового пузырька, а переход вперед на 40 байт по-прежнему происходит в том же месте, то это не будет иметь никакого эффекта.

Теперь рассмотрим абсолютные ссылки. Эти типы ссылок будут выглядеть как “использовать значение данных по адресу 0x1234567”. Если вы удалите код в любом месте двоичного файла перед этим адресом, вы заставите все измениться. Таким образом, когда любая абсолютная ссылка переходит к захвату своих значений или выполняет абсолютный переход, все местоположения будут неверными, даже если все, что вы сделали, это удалили 1 байт из двоичного файла.

Это означает, что относительные ссылки затрагиваются только добавлением/удалением байтов, если они встречаются между местом создания ссылки и пунктом назначения. Однако все абсолютные ссылки уничтожаются, если вы сдвигаете приложение даже на 1 байт. Вот почему при исправлении очень важно поддерживать размер (если, конечно, ваша цель не состоит в том, чтобы привести все к сбою, и в этом случае разбейте его вдребезги!).

Возвращаясь к nop, если вы хотите удалить фрагмент кода, например, заставить программное обеспечение пропустить проверку ключей, вместо удаления кода вы просто заменяете все это на nops. Это поддерживает выравнивание байтов приложения, но ничего не приводит к тому, что при достижении нежелательного кода ничего не происходит.

Другие отладчики
Для обратного проектирования с динамическим анализом в Windows существует множество популярных вариантов. Вот несколько:

- OllyDbg
- Immunity
- x64dbg
- WinDbg

Какой из них использовать, зависит от ситуации и предпочтений пользователя. Все они имеют схожие функции, и навыки, приобретенные в одном из них, обычно распространяются и на другие. На протяжении всей книги вы познакомитесь с несколькими различными программами; цель состоит в том, чтобы дать вам представление о многих из них, чтобы вы могли почувствовать, когда каждое из них полезно.

OllyDbg

OllyDbg - чрезвычайно популярный и мощный отладчик. В то время как большинство отладчиков сосредоточены на отладке, Olly обладает расширенными функциями, включая следующие:

- Расширяемость, плагины, написание сценариев
- Система отслеживания выполнения
- Функции исправления кода
- Автоматическое описание параметров для большинства функций Windows
- Упор на анализ двоичного кода (т.е. не основанный на отладке исходного кода)
- Небольшой размер и портативность

Эти функции делают OllyDbg отличным решением для следующего:

- Написания эксплойтов
- Анализа вредоносных программ
- Обратного проектирования

Однако, хотя OllyDbg является мощным и популярным инструментом, у него есть свои ограничения. Одним из них является то, что он работает только с 32-разрядными исполняемыми файлами, которые, по общему признанию, являются вымирающим видом, но еще не умерли.

Другая проблема заключается в том, что к интерфейсу OllyDbg часто требуется некоторое привыкание, и поначалу он не кажется надежным или интуитивно понятным. Тем не менее, вам определенно следует придерживаться его, поскольку это мощный инструмент динамического анализа.

Immunity

Immunity является форком OllyDby, что означает, что он обладает многими из тех же возможностей. Он также предоставляет множество дополнительных функций, которые делают его популярным среди разработчиков эксплойтов, таких как поддержка сценариев на Python.

Однако, как и OllyDbg, Immunity можно использовать только для отладки 32-разрядных исполняемых файлов. Кроме того, он наследует неинтуитивный пользовательский интерфейс OllyDbg.

x86dbg

x86dbg - это замена OllyDbg, которая поддерживает как 32-разрядные (x86dbg), так и 64-разрядные (x64dbg) приложения. Такая широкая поддержка означает, что это обычно предпочтительный инструмент при реверсировании или отладке 64-разрядных приложений.

WinDbg
WinDbg - это отладчик, который универсально применим, имеет мощную поддержку и предлагает отличную поддержку символов отладки (но которая менее полезна при RE). Однако он ориентирован на отладку и лишен некоторых функций переориентированных инструментов.

Отладка с Immunity
Из-за ограничений по времени и пространству изучение всех этих отладчиков в этой книге невозможно. Immunity был выбран из-за его популярности для обратного проектирования и разработки эксплойтов. Однако важно помнить, что все эти отладчики обладают схожими функциями, и навыки, приобретенные в одном из них, часто передаются другим.

На рисунке 10.22 показано, как выглядит Immunity в Windows. В верхнем левом углу, двигаясь по часовой стрелке, четыре окна отображают разборку программы, регистры, стек и память.

Immunity: Сборка
На рисунке 10.23 показана разборка программы в Immunity. Обратите внимание, что здесь показаны адрес памяти, машинный код и сборка x86.

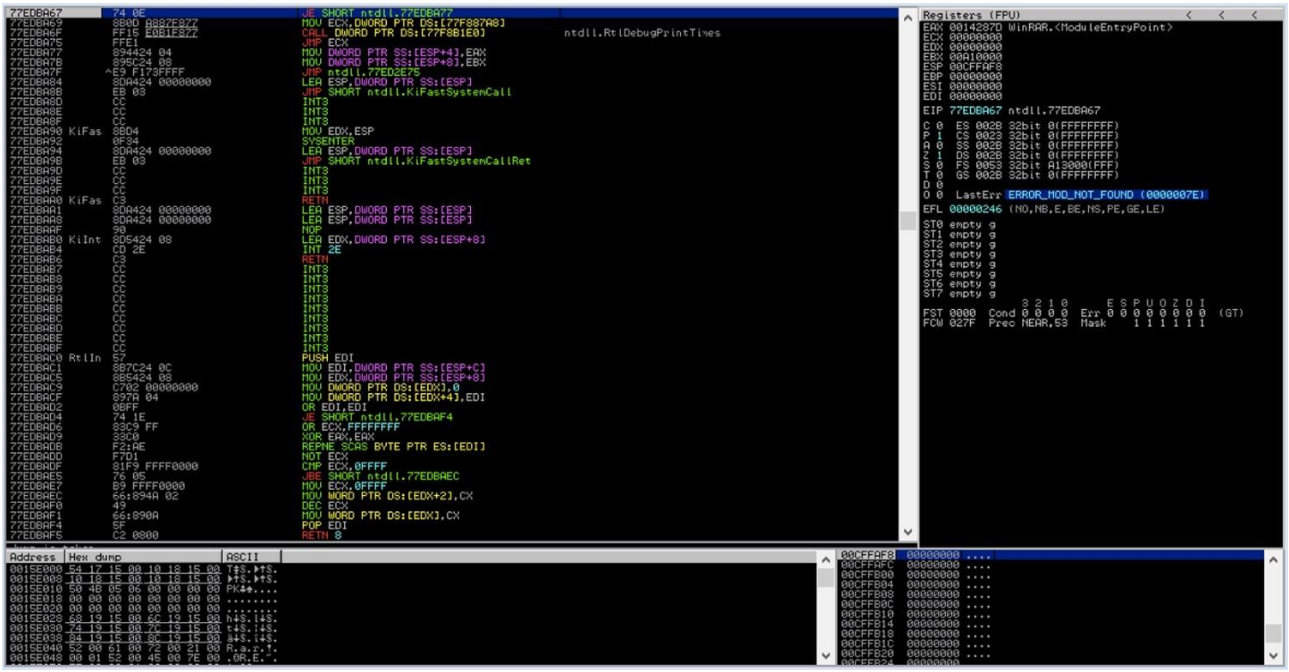


Рисунок 10.22: Окно отладчика невосприимчивости

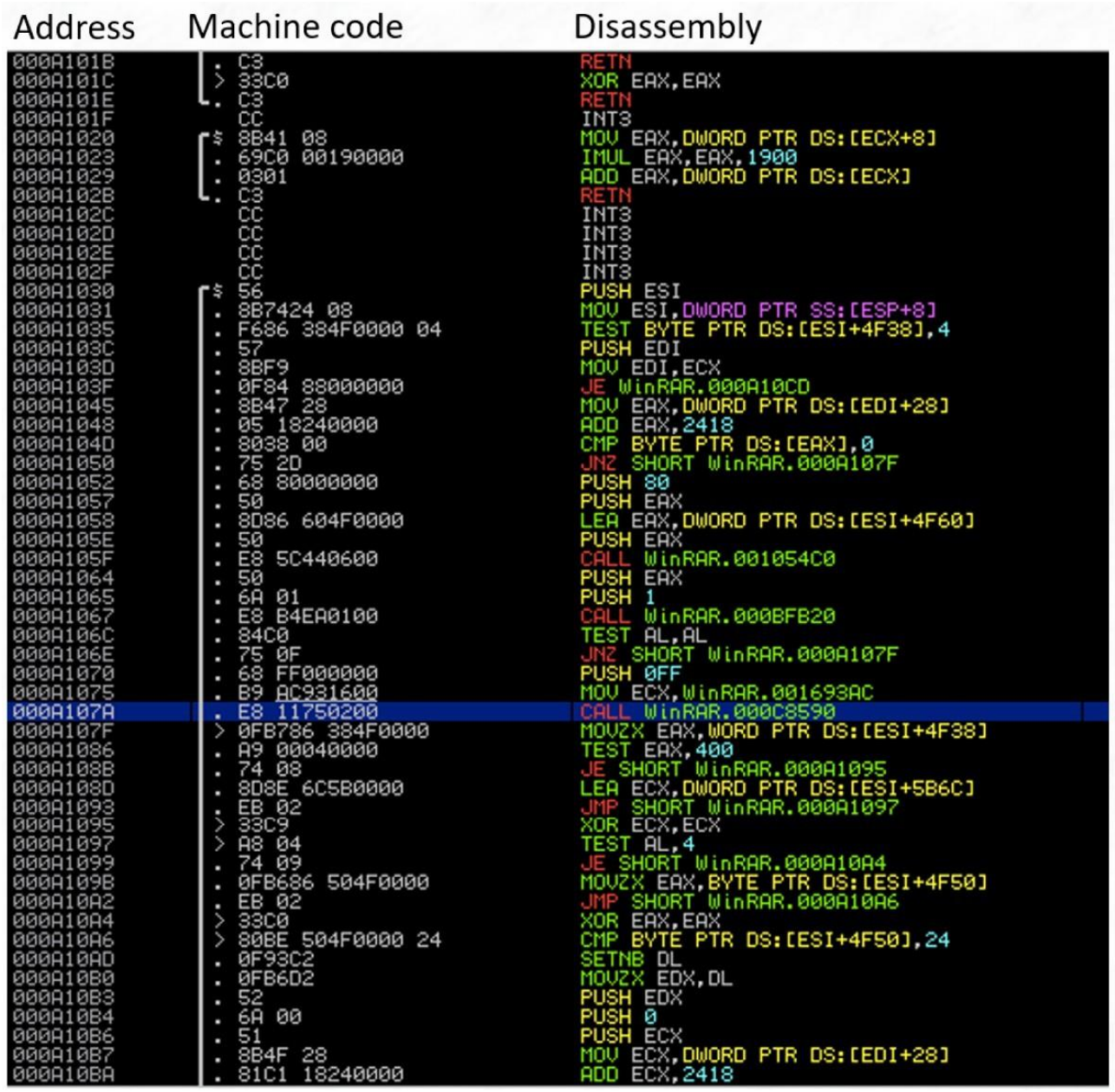


Рисунок 10.23: Ассемблерный код в отладчике Immunity

Чтобы выбрать строку кода, щелкните по ней. Как только строка выбрана, Immunity предлагает различные сочетания клавиш, включая следующие:

- ;: Добавьте комментарий к выбранной строке. Это наиболее важная часть обратного проектирования; она помогает вам отслеживать вашу работу.
- ctrl-a: Автоматический анализ программы. Immunity может довольно хорошо выполнять добавление комментариев и угадывание параметров функции.
- <enter>: Переход к выбранной функции. Например, если вы видите, что сборка вызывает 0x1234, и хотите узнать, что делает функция в 0x1234.
- -: Вернитесь к предыдущему местоположению. Например, после того, как вы проанализировали функцию 0x1234 и хотите вернуться туда, где вы были.
- +: Перейдите к следующему местоположению (после нажатия -). Например, если вы вернулись к вызывающей функции с помощью -, но затем хотите вернуться к функции

0x1234.
-ctrl-r: Найдите перекрестные ссылки на выбранную строку. Например, если у вас есть строка, выбранная в окне дампа памяти, и вы хотите знать, кто использует эту строку; или если у вас есть верхняя часть функции, выбранная в дизассемблировании, и вы хотите узнать, кто вызывает эту функцию.
Дважды щелкните адрес: установите точку останова отладки по этому адресу.

Immunity: Модули

В Immunity вы можете загрузить список исполняемых модулей, нажав кнопку e. Здесь показан весь код, включая динамически загружаемые библиотеки, который вы можете отлаживать, как показано на рисунке 10.24. После открытия списка вы можете дважды щелкнуть модуль, чтобы перейти к этому коду.

Когда вы запустите Immunity, посмотрите, какой модуль вы просматриваете в данный момент, проверив регистр eip. Почти в каждом случае вам захочется начать с отладки основного исполняемого файла, а не разделяемой библиотеки, такой как ntdll. Вы можете использовать окно модулей для переключения на основной исполняемый файл.

Immunity: Строки

Часто бывает полезно узнать, какой код использует определенную строку в исполняемом файле. Чтобы найти все строки, используемые программой, щелкните правой кнопкой мыши и выберите Поиск ➪ Всех текстовых строк, на которые даны ссылки, как показано на рисунке 10.25.

В окне строки щелкните правой кнопкой мыши и выберите Поиск текста, чтобы найти определенную строку, как показано на рисунке 10.26. Затем снова щелкните правой кнопкой мыши и выберите Поиск далее, чтобы найти следующую ссылку на эту строку. Вы можете дважды щелкнуть адрес строки, чтобы перейти к местоположению, где она используется при разборке.

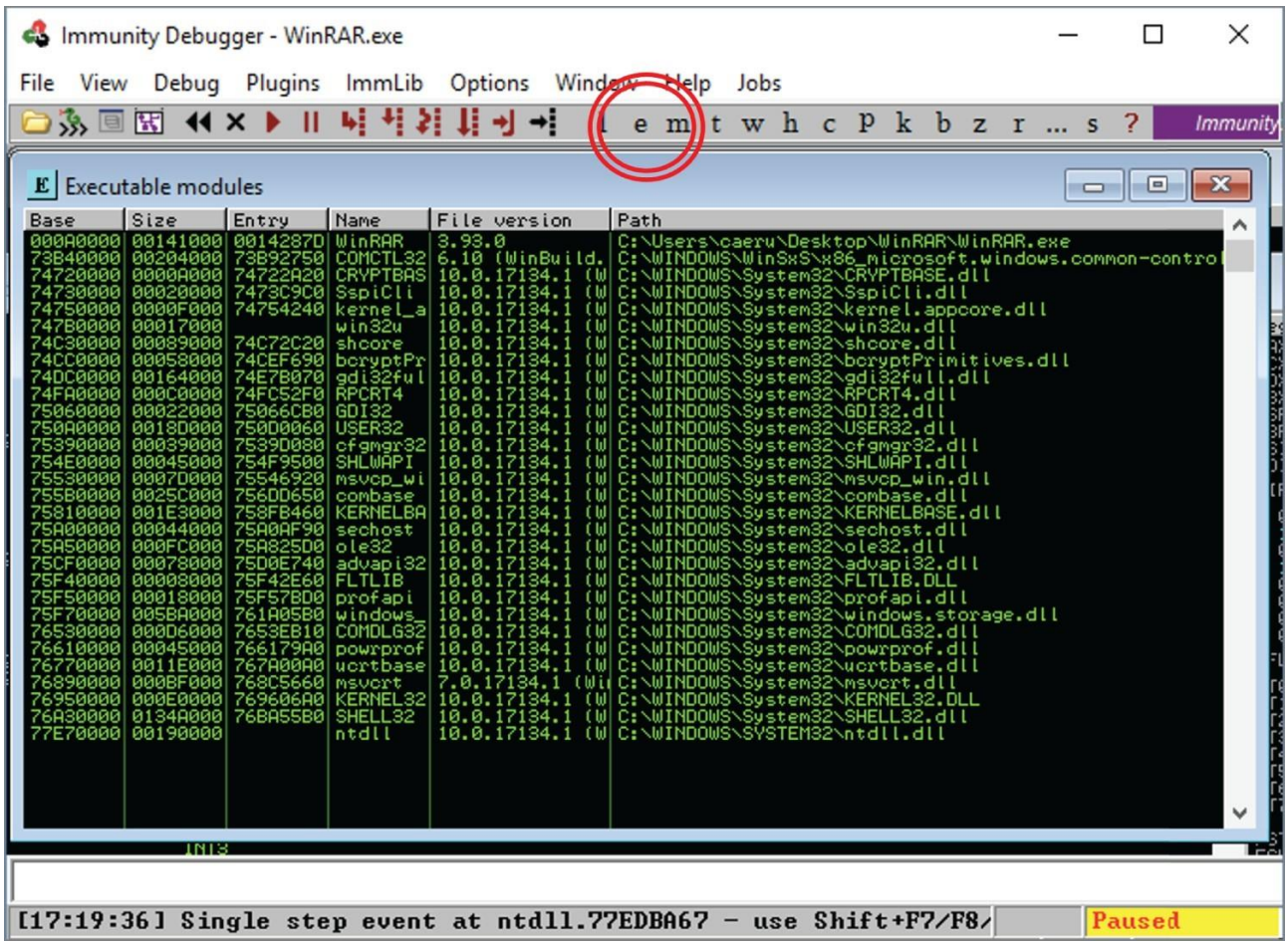

Immunity Debugger - WinRAR.exe
File View Debug Plugins ImmLib Options Window Help Jobs
Executable modules
Base Size Entry Name File version Path
000A0000 00141000 0014287D WinRAR 3.93.0 C:\Users\caeru\Desktop\WinRAR\WinRAR.exe
73B40000 00204000 73B92750 COMCTL32 6.10 (WinBuild. C:\WINDOWS\WinSxS\x86_microsoft.windows.common-contro
74720000 0000A000 74722A20 CRYPTBAS 10.0.17134.1 (W C:\WINDOWS\System32\CRYPTBASE.dll
74730000 00020000 7473C9C0 SspiCli 10.0.17134.1 (W C:\WINDOWS\System32\SspiCli.dll
74750000 0000F000 74754240 kernel_a 10.0.17134.1 (W C:\WINDOWS\System32\kernel.appcore.dll
747B0000 00017000 win32u 10.0.17134.1 (W C:\WINDOWS\System32\win32u.dll
74C30000 00089000 74C72C20 shcore 10.0.17134.1 (W C:\WINDOWS\System32\shcore.dll
74CC0000 00058000 74CEF690 bcryptPr 10.0.17134.1 (W C:\WINDOWS\System32\bcryptPrimitives.dll
74DC0000 00164000 74E7B070 gdi32ful 10.0.17134.1 (W C:\WINDOWS\System32\gdi32full.dll
74FA0000 000C0000 74FC52F0 RPCRT4 10.0.17134.1 (W C:\WINDOWS\System32\RPCRT4.dll
75060000 00022000 75066CB0 GDI32 10.0.17134.1 (W C:\WINDOWS\System32\GDI32.dll
750A0000 0018D000 750D0060 USER32 10.0.17134.1 (W C:\WINDOWS\System32\USER32.dll
75390000 00039000 7539D080 cfgmgr32 10.0.17134.1 (W C:\WINDOWS\System32\cfgmgr32.dll
754E0000 00045000 754F9500 SHLWAPI 10.0.17134.1 (W C:\WINDOWS\System32\SHLWAPI.dll
75530000 0007D000 75546920 msvcp_wi 10.0.17134.1 (W C:\WINDOWS\System32\msvcp_win.dll
755B0000 0025C000 756DD650 combase 10.0.17134.1 (W C:\WINDOWS\System32\combase.dll
75810000 001E3000 758FB460 KERNELBA 10.0.17134.1 (W C:\WINDOWS\System32\KERNELBASE.dll
75A00000 00044000 75A0AF90 sechost 10.0.17134.1 (W C:\WINDOWS\System32\sechost.dll
75A50000 000FC000 75A825D0 ole32 10.0.17134.1 (W C:\WINDOWS\System32\ole32.dll
75CF0000 00078000 75D0E740 advapi32 10.0.17134.1 (W C:\WINDOWS\System32\advapi32.dll
75F40000 00008000 75F42E60 FLTLIB 10.0.17134.1 (W C:\WINDOWS\System32\FLTLIB.DLL
75F50000 00018000 75F57BD0 profapi 10.0.17134.1 (W C:\WINDOWS\System32\profapi.dll
75F70000 005BA000 761A05B0 windows_ 10.0.17134.1 (W C:\WINDOWS\System32\windows.storage.dll
76530000 000D6000 7653EB10 COMDLG32 10.0.17134.1 (W C:\WINDOWS\System32\COMDLG32.dll
76610000 00045000 766179A0 powrprof 10.0.17134.1 (W C:\WINDOWS\System32\powrprof.dll
76770000 0011E000 767A00A0 ucrtbase 10.0.17134.1 (W C:\WINDOWS\System32\ucrtbase.dll
76890000 000BF000 768C5660 msvcrt 7.0.17134.1 (Wi C:\WINDOWS\System32\msvcrt.dll
76950000 000E0000 769606A0 KERNEL32 10.0.17134.1 (W C:\WINDOWS\System32\KERNEL32.DLL
76A30000 0134A000 76BA55B0 SHELL32 10.0.17134.1 (W C:\WINDOWS\System32\SHELL32.dll
77E70000 00190000 ntdll 10.0.17134.1 (W C:\WINDOWS\SYSTEM32\ntdll.dll
[17:19:36] Single step event at ntdll.77EDBA67 - use Shift+F7/F8/
Paused

Рисунок 10.24: Исполняемые модули в отладчике иммунитета

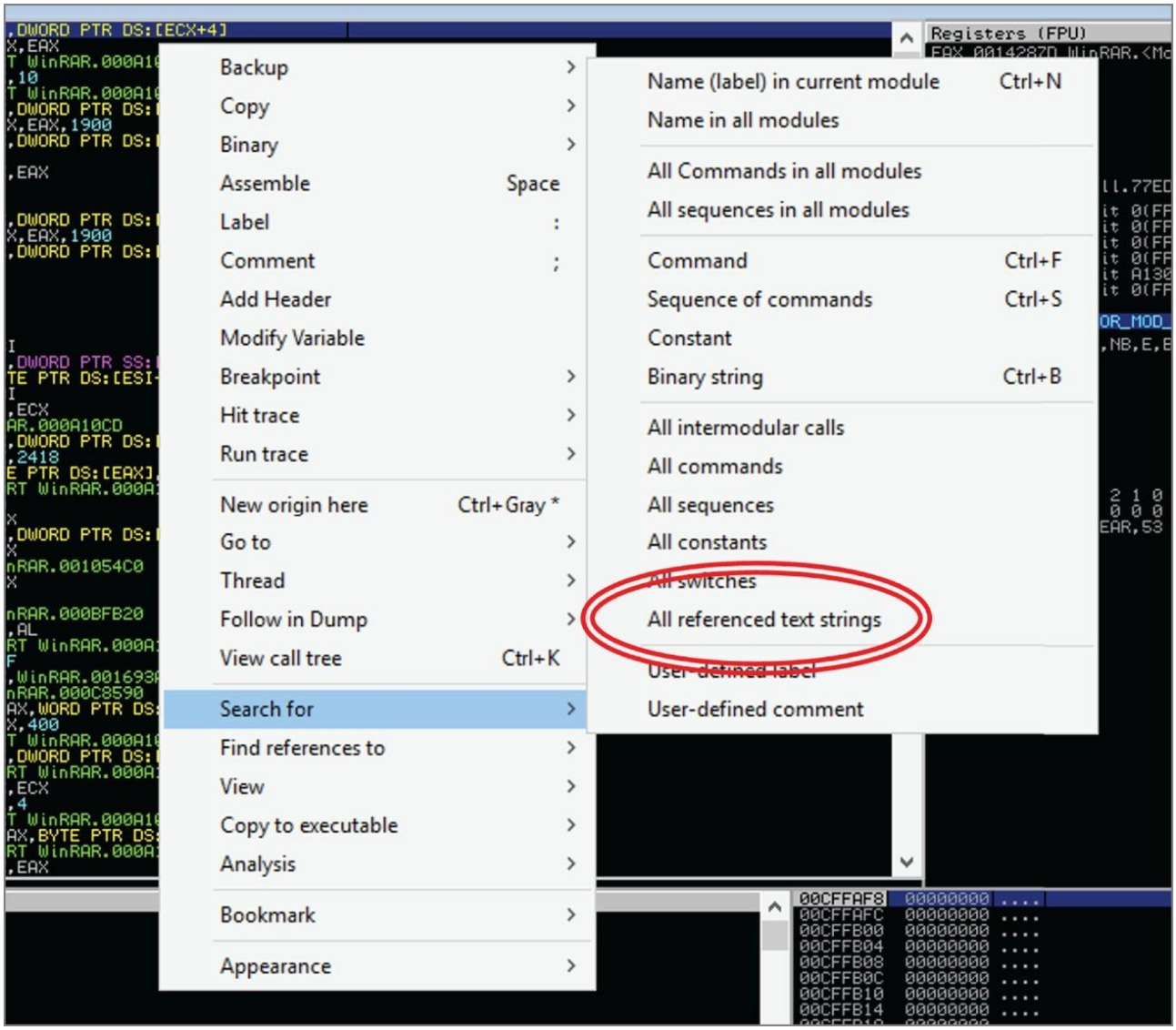


Рисунок 10.25: Строки в отладчике иммунитета

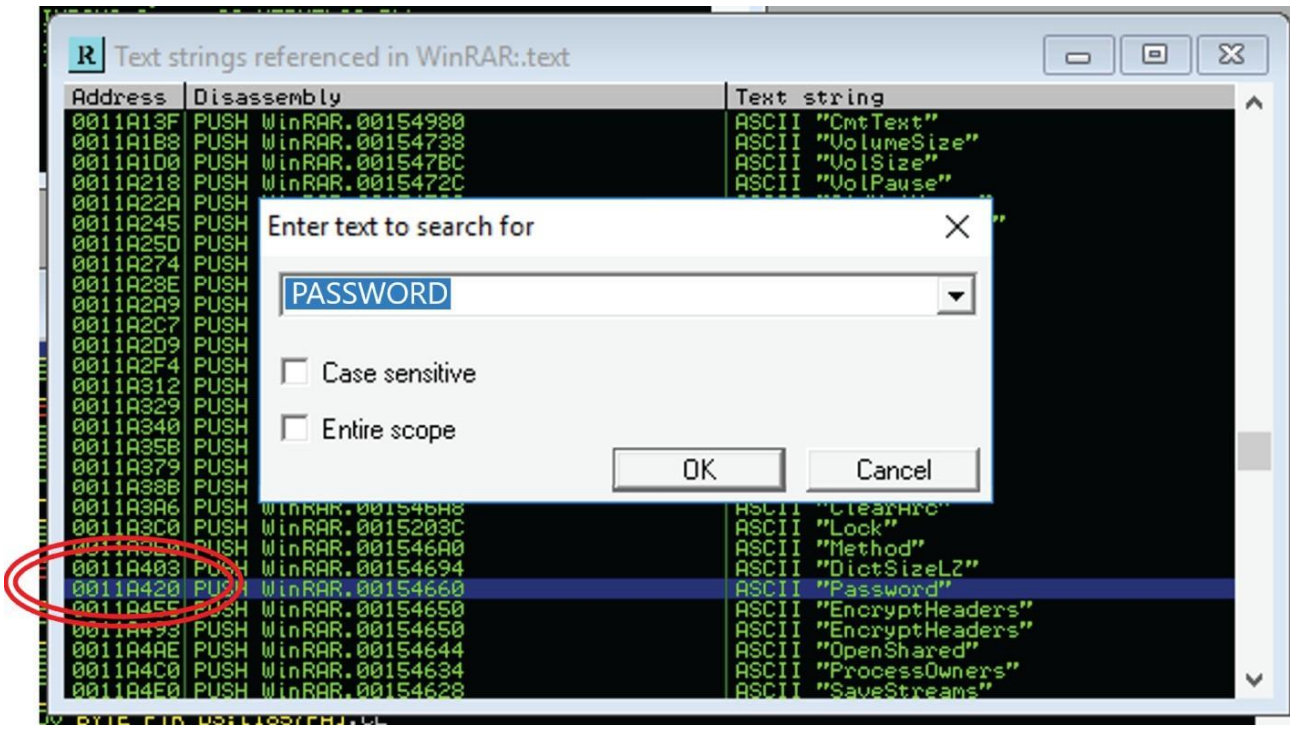


Рисунок 10.26: Ссылки на строки в отладчике Immunity

Immunity: Запуск программы

Щелкните стрелку воспроизведения, чтобы запустить исполняемый файл в отладчике, как показано на рисунке 10.27. Выполнение можно остановить, нажав крестик слева от стрелки воспроизведения, или приостановить, нажав кнопку паузы справа от нее. Выполнение можно перезапустить с помощью кнопки с двумя стрелками влево.

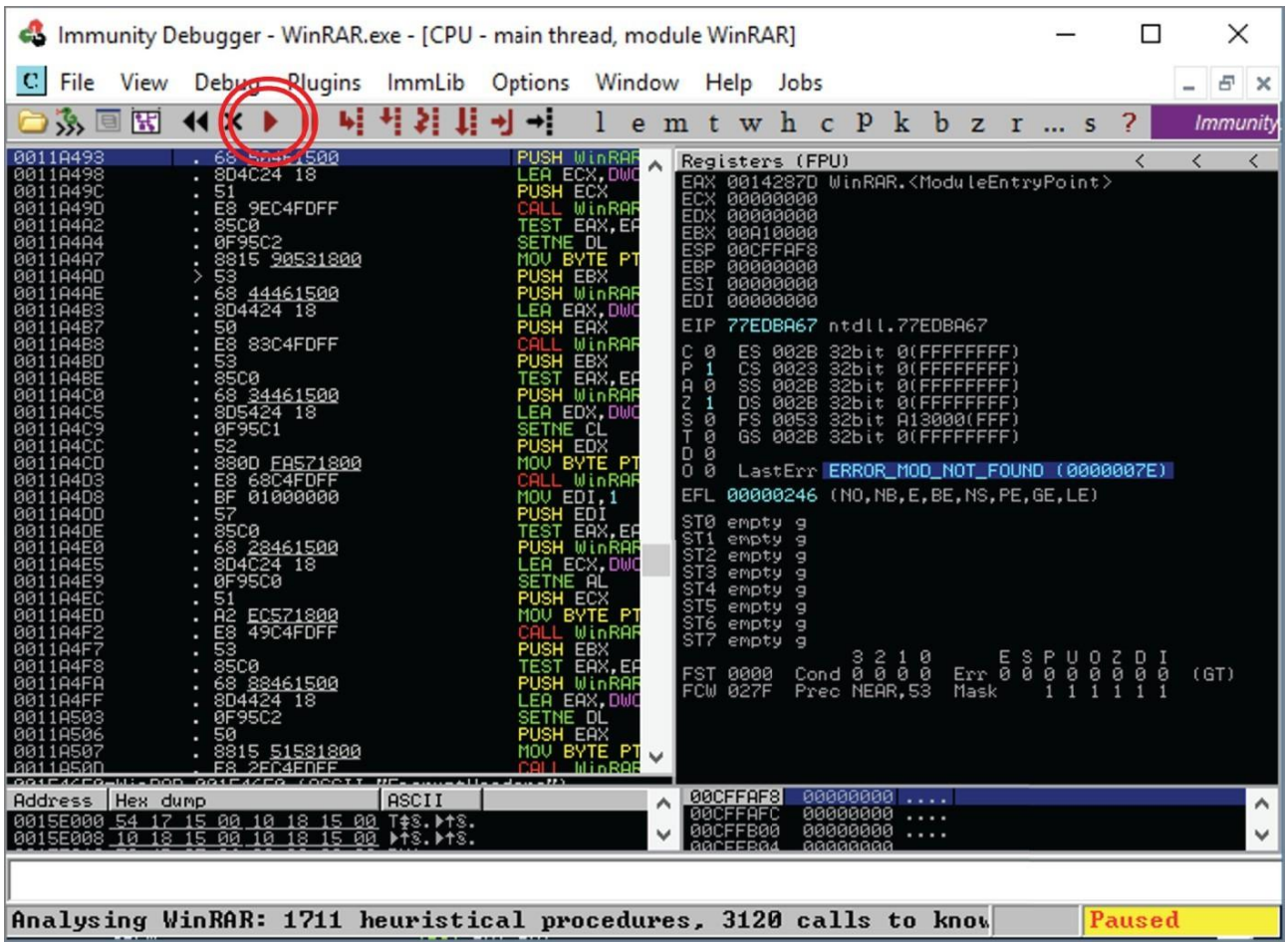


Рисунок 10.27: Запуск исполняемого файла в Immunity debugger

После того, как выполнение было остановлено точкой останова или кнопкой паузы, вы можете нажать Step Into, чтобы выполнить одну инструкцию программы, как показано на рисунке 10.28. В качестве альтернативы, если вы остановились при вызове функции, но уже знаете или вам безразлично, что делает функция, нажмите overstep Over, как показано на рисунке 10.29, чтобы продолжить отладку после возврата функции.

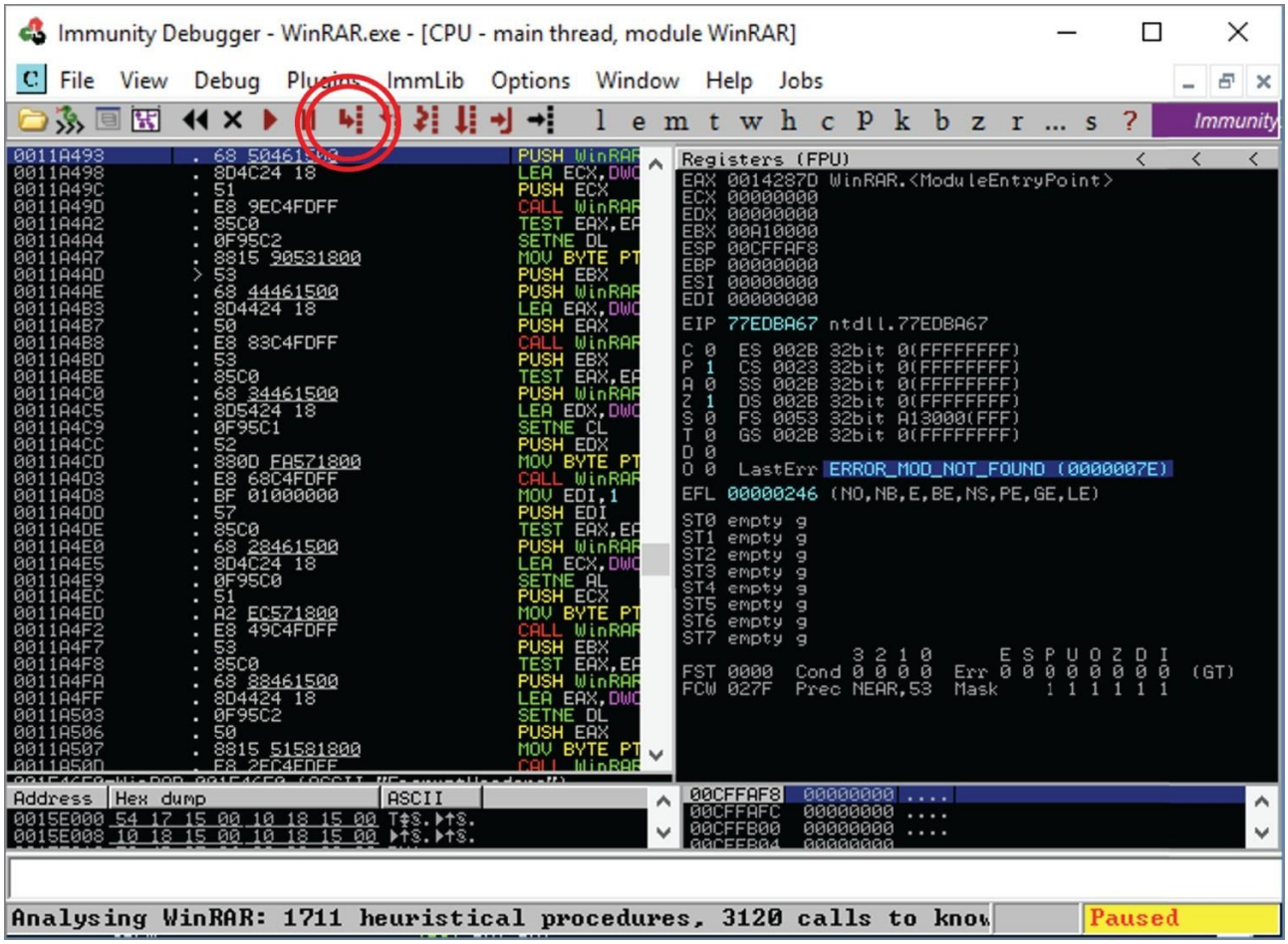


Рисунок 10.28: Одноступенчатый отладчик Immunity

Immunity: Исключения

Многие приложения генерируют исключения как часть обычного выполнения. Например, блок try {} except {} сгенерирует исключение, если что-то пойдет не так в блоке try. Как отладчик, инструменты динамического анализа, такие как Immunity, обычно сначала перехватывают исключение, чтобы увидеть, хотите ли вы что-либо с ним сделать.

Но при обратном проектировании вы, как правило, не хотите вмешиваться в нормальное выполнение. Вместо этого вы хотите позволить приложению обрабатывать исключение так, как это было бы обычно. Это означает, что вы почти всегда хотите передать исключение из отладчика в приложение.

Как показано на рисунке 10.30, сообщения об исключениях отображаются в нижней части окна Immunity, но каждый отладчик немного отличается. В Immunity нажмите Shift+F9, чтобы передать исключение и продолжить выполнение.

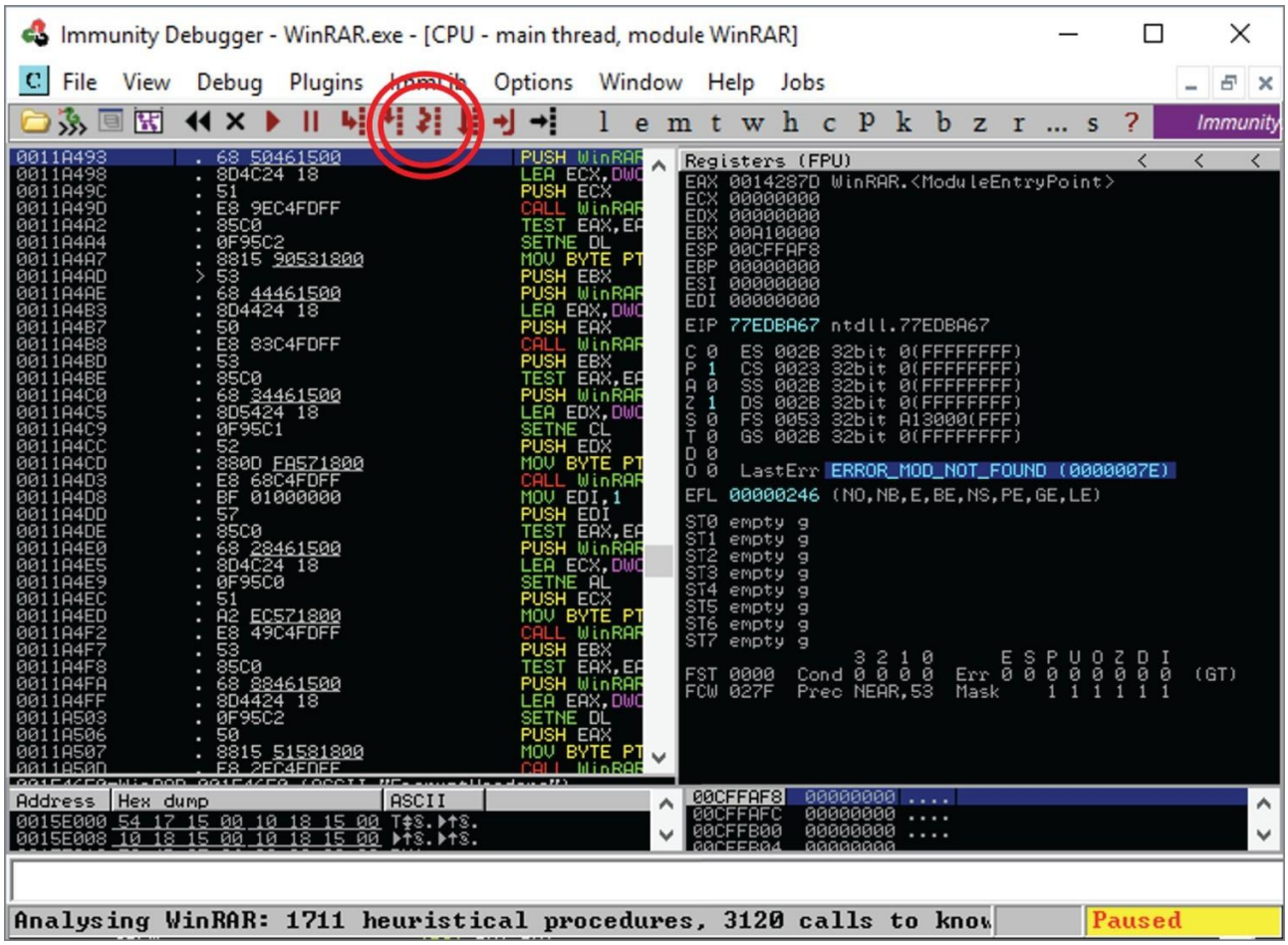


Рисунок 10.29: Пошаговое выполнение инструкций в отладчике иммунитета

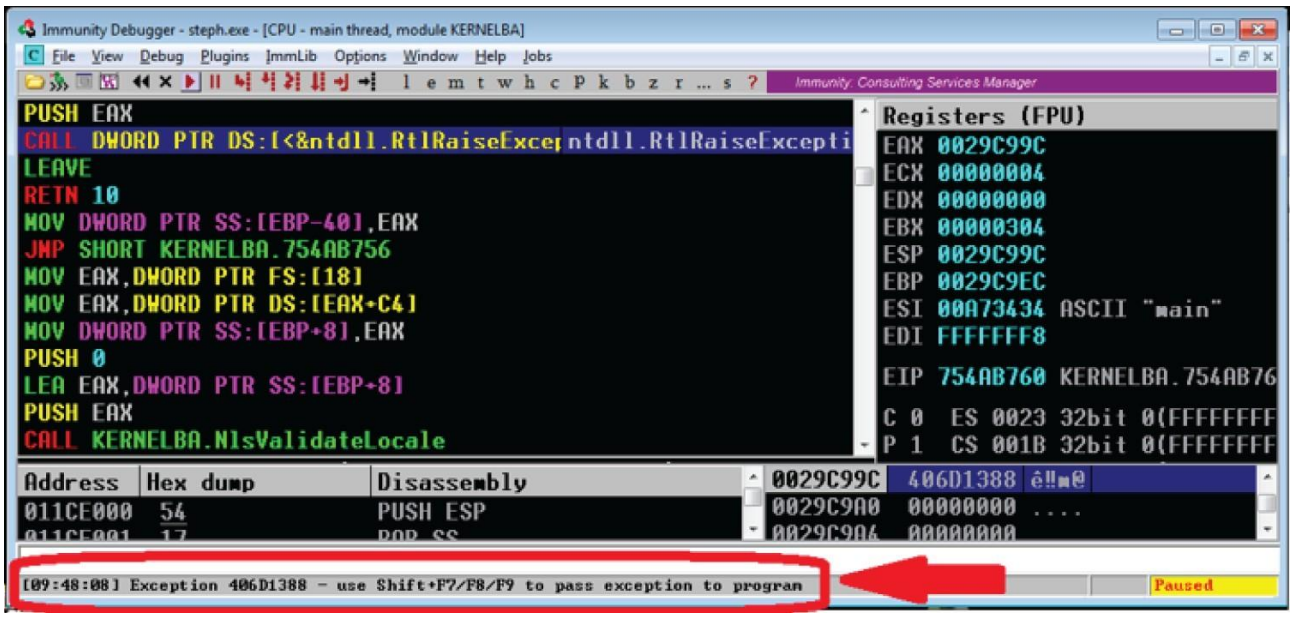


Рисунок 10.30: Исключения в отладчике Immunity

Immunity: Перезапись программы
У Immunity есть множество функций, помогающих в разработке исправлений для изменения поведения программного обеспечения. Для целей взлома программного обеспечения это включает внесение изменений в программу для удаления проверок ключей, экранов проверки ошибок и т.д.

В своих первых взломах вы будете использовать процесс “удаления” кода, чтобы удалить его из программы. Это предполагает замену инструкций программы инструкциями nop.

Чтобы сделать это в Immunity, сначала выберите инструкции, которые вы хотите удалить. Затем щелкните правой кнопкой мыши и выберите Binary ➪ Заполнить NOPs, как показано на рисунке 10.31.

Это заменит выбранную команду (команды) серией nops, как показано на рисунке 10.32.

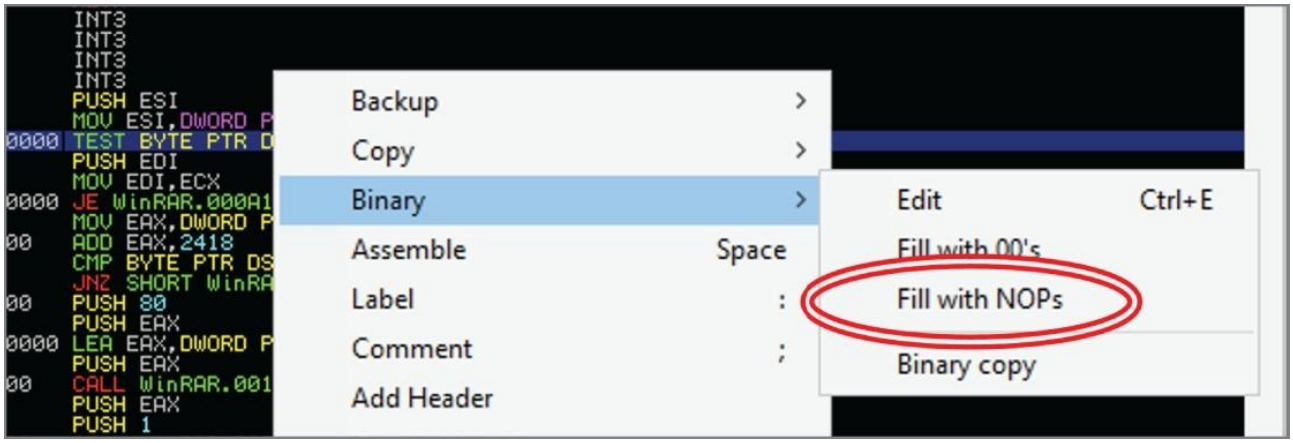


Рисунок 10.31: удаление кода в отладчике Immunity

Рисунок 10.32: исправленный код в отладчике Immunity

После модификации программы протестируйте исправление, повторно запустив программу. Если вы исправили правильную часть кода, вы должны обнаружить, что экран проверки (проверка ключа и т.д.) исчез.

Однако, если при установке исправления произошел сбой или не удалось удалить вашу цель, вы можете легко отменить свои изменения и повторить попытку. Для этого нажмите кнопку исправления, чтобы открыть окно исправлений. Затем щелкните правой кнопкой мыши на вашем патче и выберите Восстановить исходный код, как показано на рисунке 10.33, чтобы восстановить ваш патч и повторить попытку.

Как только вы определили работающий патч, сохраните изменения в исполняемом файле, чтобы сделать его постоянным. Как показано на рисунке 10.34, щелкните правой кнопкой мыши и выберите Копировать в исполняемый файл ➪ Все изменения. Когда появится окно подтверждения, выберите Копировать все.

Должно появиться окно измененного исполняемого файла с вашими изменениями. Закройте окно и выберите Да, чтобы сохранить файл. Дайте файлу новое имя, например cracked.exe.

Если вы уверены в своих изменениях, вы можете запустить cracked.exe напрямую. Если вы хотите продолжить отладку с этими новыми изменениями, вам нужно перезагрузить cracked.exe в Immunity.

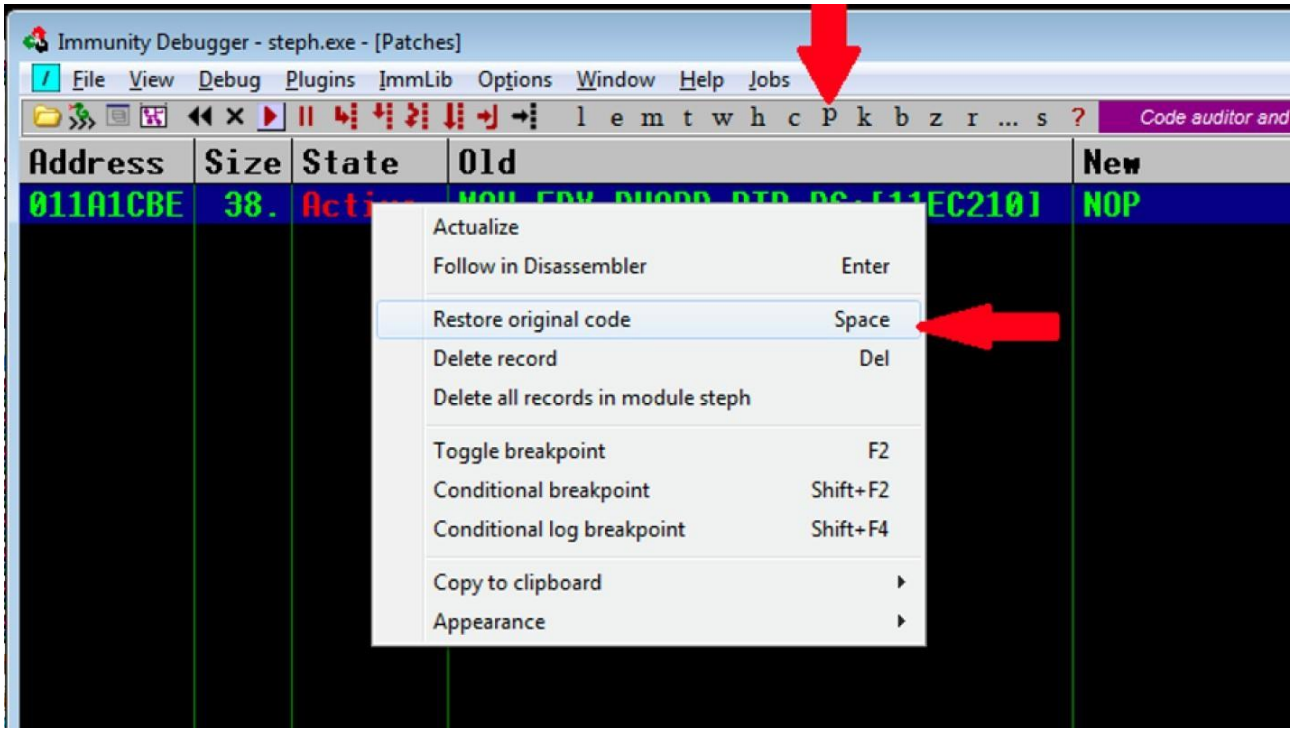


Рисунок 10.33: Возврат измененного кода в отладчике иммунитета

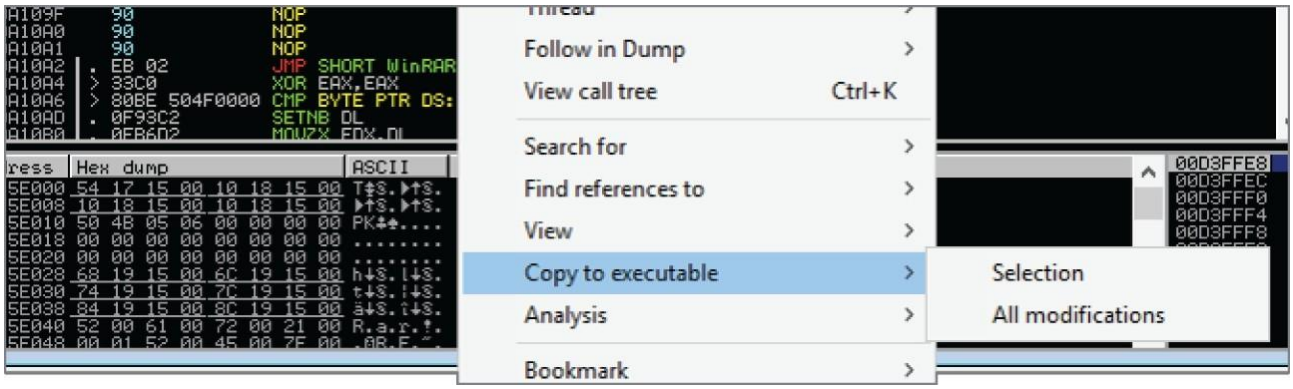


Рисунок 10.34: Сохранение измененного файла в отладчике Immunity

Лабораторная работа: Взлом с помощью Immunity
В этой лабораторной работе дается практический опыт взлома программ с использованием отладчика. Лабораторные работы и все связанные с ними инструкции можно найти в соответствующей папке здесь:

https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES

Для этой лабораторной работы, пожалуйста, найдите Lab Cracking with Immunity и следуйте предоставленным инструкциям.

Навыки
В этой лаборатории практикуется обратное проектирование, внесение исправлений и обход программных защит с использованием Immunity и Resource Hacker. Некоторые из ключевых навыков, которые тестировались, включают следующее:

- Обратное проектирование x86
- Внесение исправлений
- Статический и динамический анализ

Выводы
Программное обеспечение можно легко модифицировать для добавления, изменения или удаления функциональных возможностей. Эти же методы можно использовать для обхода любых средств защиты, от тривиальных до продвинутых, при условии, что вы понимаете, как работает программное обеспечение.

Резюме
Средства проверки ключей предназначены для защиты от распространения и использования нелицензионных и взломанных копий программного обеспечения, но ни одна защита не идеальна. Такие инструменты, как Procmon, Resource Hacker и debuggers, могут быть использованы для понимания этих защит и устранения их с помощью генераторов ключей или исправлений.

Глава 11
Внесение исправлений и расширенный инструментарий
В предыдущей главе описывались взлом программного обеспечения и внесение исправлений. В этой главе более подробно рассматривается внесение исправлений и некоторые из более продвинутых инструментов, которые можно использовать для исправления ошибок.

Внесение исправлений в редакторе 010
Часто бывает полезно иметь возможность просматривать и редактировать шестнадцатеричный код файла. Если вы когда-либо пытались открыть двоичный файл в текстовом редакторе, вы видели много странных символов и пустых мест. Это связано с тем, что текстовый редактор пытается интерпретировать все в файле как ASCII, чем это не является. Вместо этого нам нужен редактор, который будет отображать как hex, а не ASCII. Существует множество различных шестнадцатеричных редакторов, способных это делать. Одним из наших любимых является редактор 010. (Найдите ссылки в разделе Инструменты на нашем сайте GitHub по адресу https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES).

Откройте любой файл (исполняемый файл, файл данных, изображение, музыку и т.д.), чтобы просмотреть его шестнадцатеричный формат. На рисунке 11.1 показан пример исполняемого файла в редакторе 010.

На рисунке 11.2 показана панель инспектора. Это показывает различные возможные интерпретации данных, отображаемых вашим курсором.

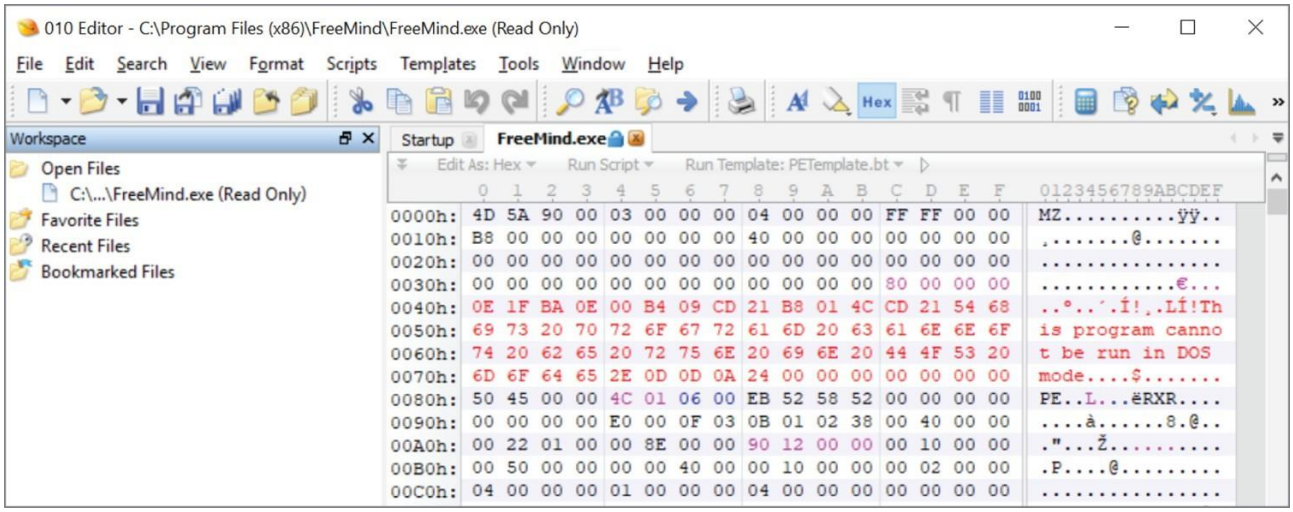


Рисунок 11.1: Просмотр файла в редакторе 010

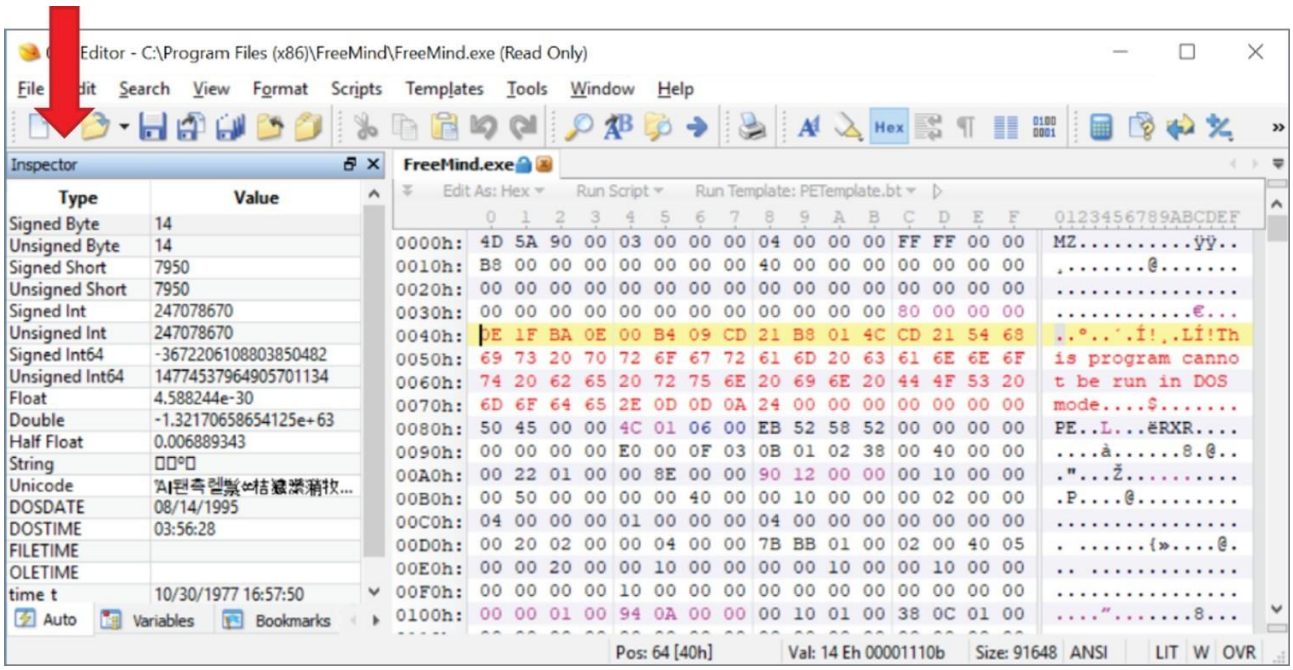


Рисунок 11.2: Панель инспектора в редакторе 010

Если вы знаете, что ищете, вы можете выполнить поиск, как показано на рисунке 11.3. Вы можете выполнять поиск по множеству различных типов данных, включая следующие:

- Текст
- Шестнадцатеричные байты
- Строка ASCII
- Строка Unicode
- Строка EBCDIC
- Байт со знаком/без знака
- Короткий байт со знаком/без знака
- Подписанный/без знака int

- Подписанный/без знака int64
- Поплавок
- Двойной
- Имя переменной
- Значение переменной

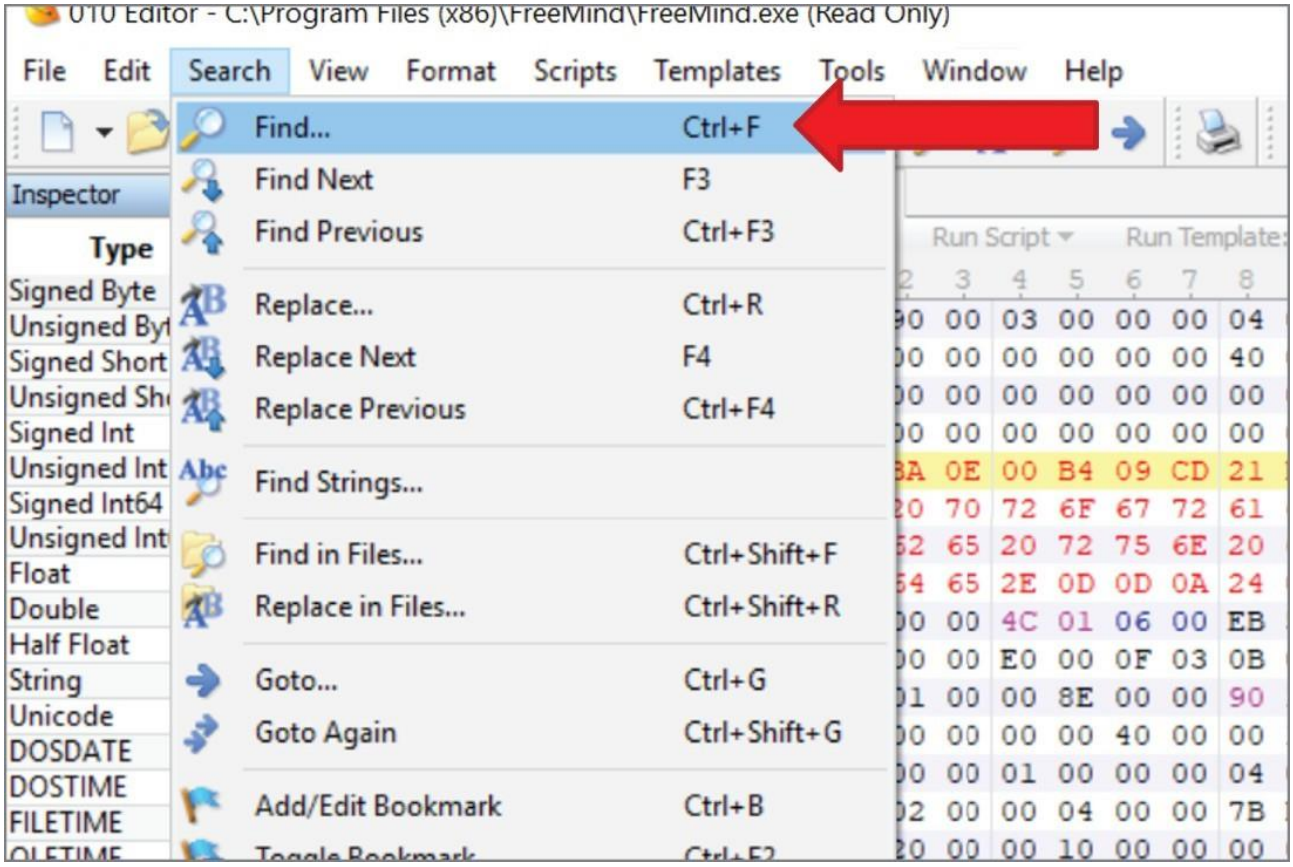


Рисунок 11.3: Поиск в редакторе 010

Вы можете перейти к определенному адресу, если знаете, куда вам нужно перейти, как показано на рисунке 11.4. Это местоположение “куда идти” может быть указано в виде байта, номера строки, сектора или краткости.

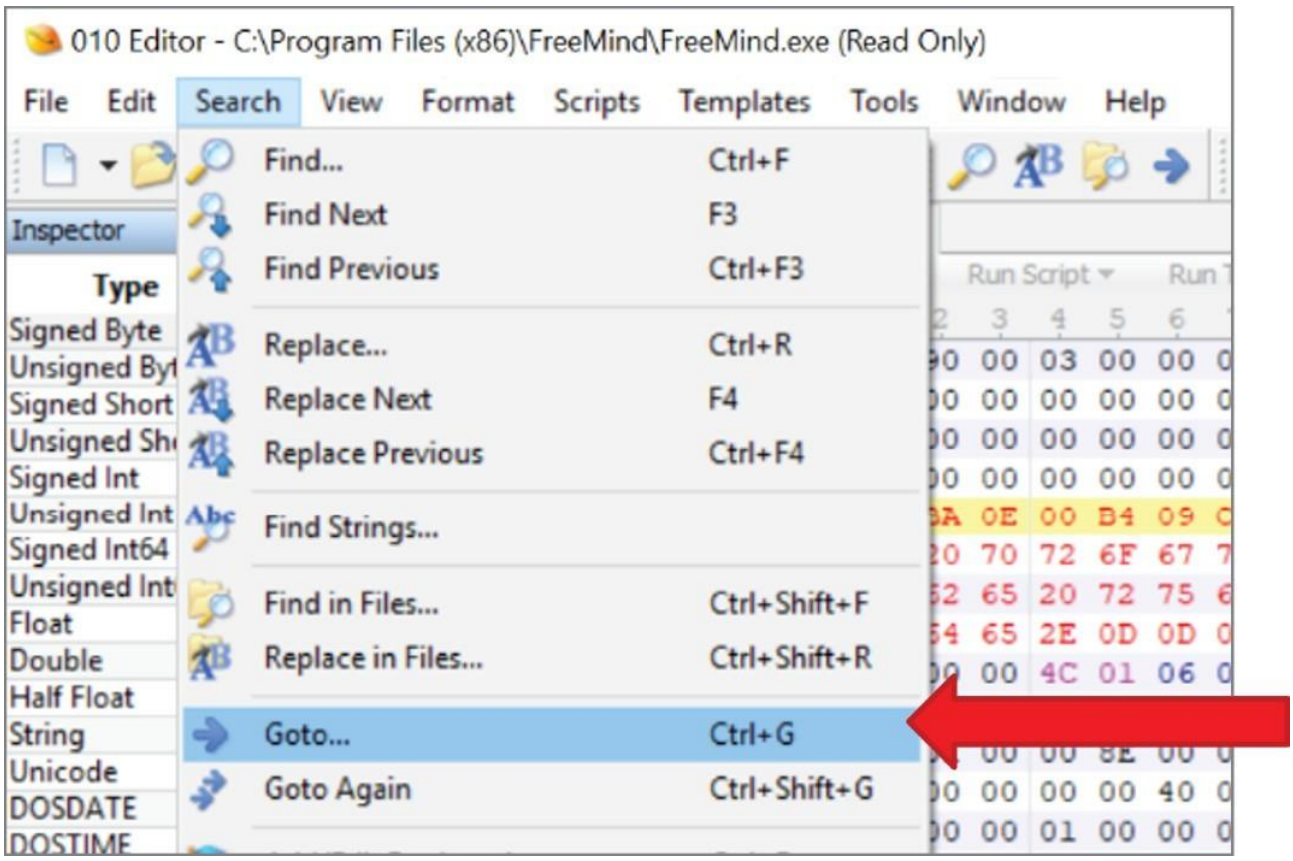


Рисунок 11.4: Переход к адресу в редакторе 010

В редакторе 010 вы можете напрямую изменить шестнадцатеричный код. Просто наведите курсор и начните печатать, чтобы перезаписать.

Однако редактор 010 понимает, насколько важно поддерживать размер файла. Когда вы вводите значения, редактор 010 перезаписывает существующие значения в этом месте. Он не вставляет их, что увеличило бы размер файла.

Исправление Codefusion

После того, как исследователь выясняет, как взломать программу, следующим шагом часто является создание утилиты для исправления/ взломщика. Это позволит другим взломать ту же программу.

Coldefusion - популярный генератор исправлений. Он создает автономный исполняемый файл, который можно использовать для взлома определенного приложения. (Найдите ссылки в разделе инструментов нашего сайта GitHub здесь:

https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES).

Чтобы начать создание патчера, запустите Coldfusion и настройте информацию, которая будет отображаться при запуске патчера. Эта информация показана на рисунке 11.5 и включает заголовок программы, название программы, комментарии, значок и т.д. Это может быть все, что вы захотите.

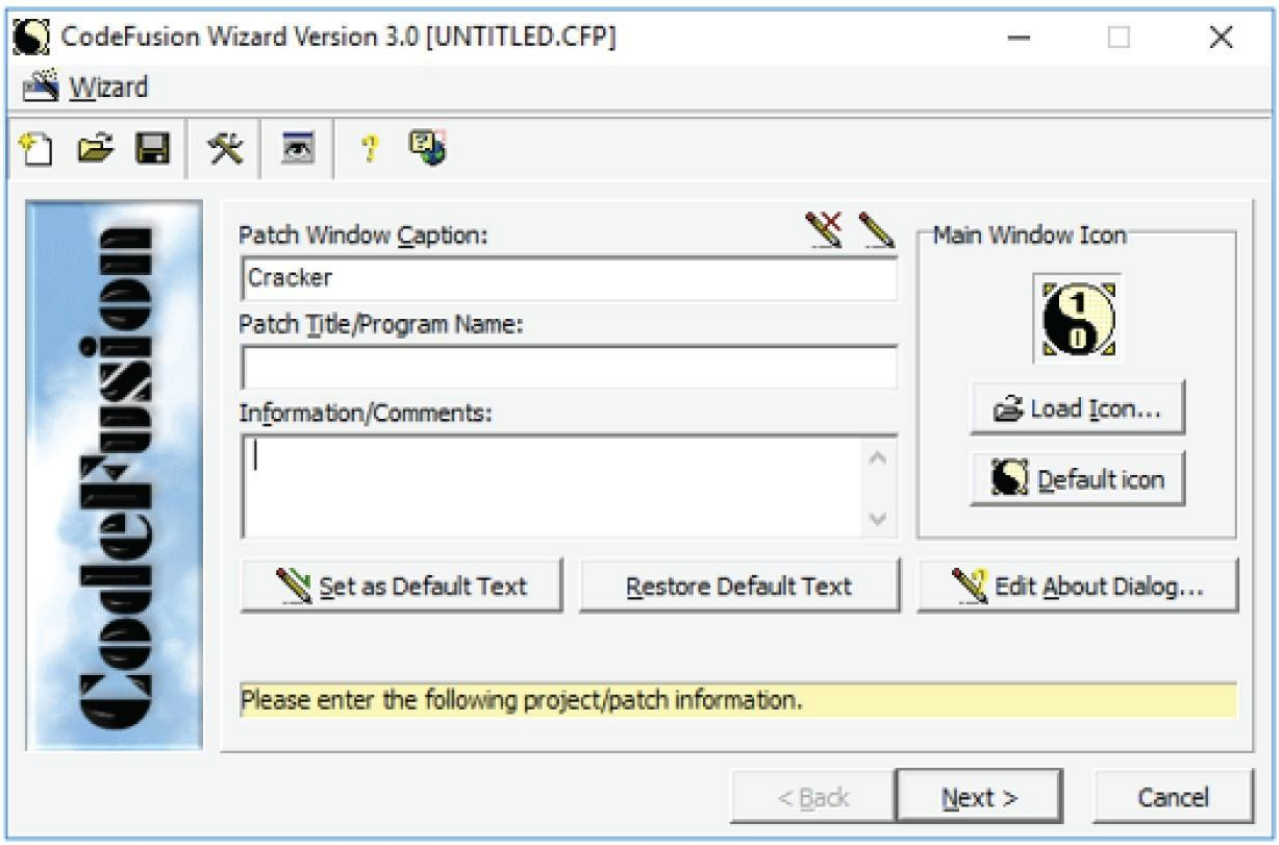


Рисунок 11.5: Начальный экран CodeFusion

На следующем экране добавьте файлы для исправления, как показано на рисунке 11.6. Это исполняемый файл, который вы хотите взломать.

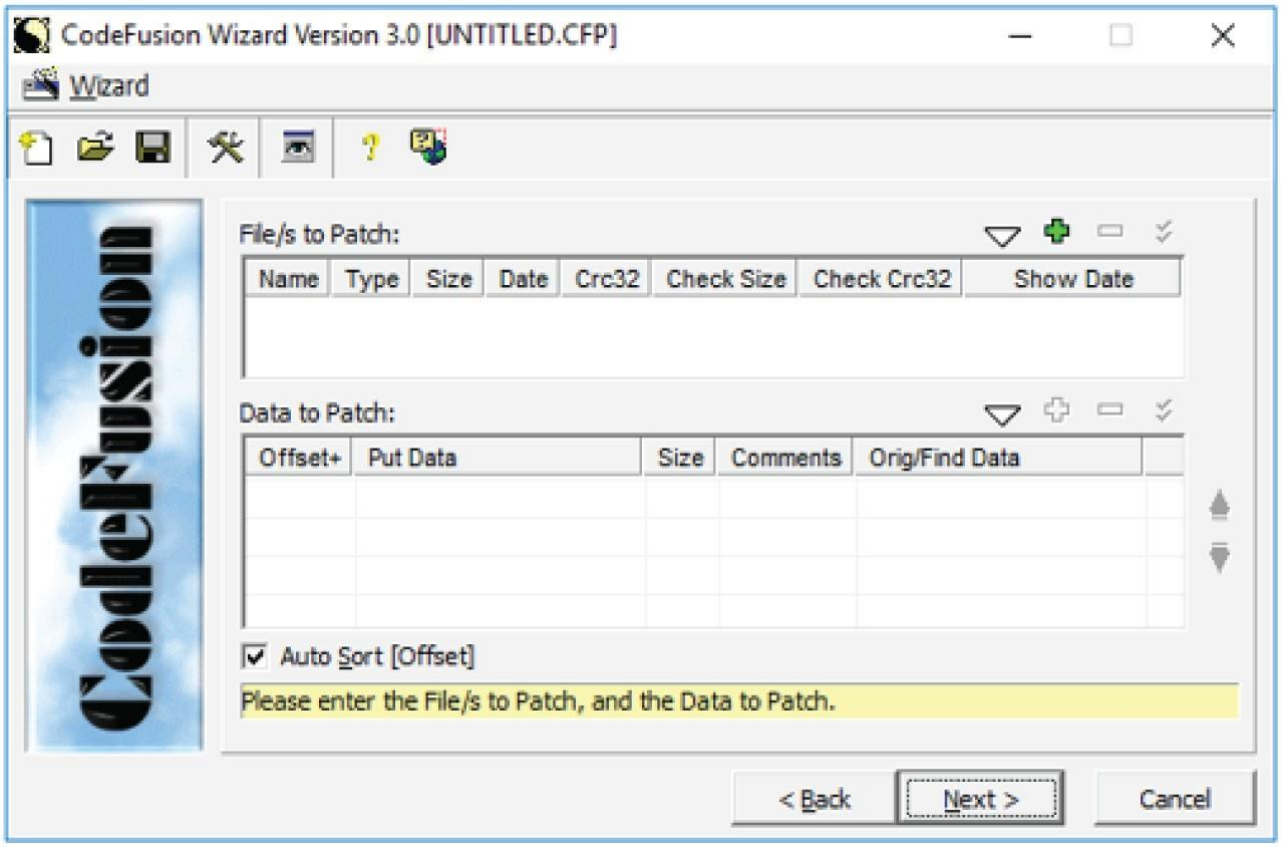


Рисунок 11.6: Загрузка файла в CodeFusion

Далее добавьте информацию о патче, щелкнув значок +, показанный на рисунке 11.7. Обычно это информация, которую вы почерпнули из Immunity, Cheat Engine, IDA и т.д. Обычно он включает в себя смещение для исправления и байты для замены. Часто байты для исправления равны 0x90 (nops). На следующей странице нажмите "Создать исполняемый файл Win32", чтобы создать EXE-файл для исправления целевого приложения.

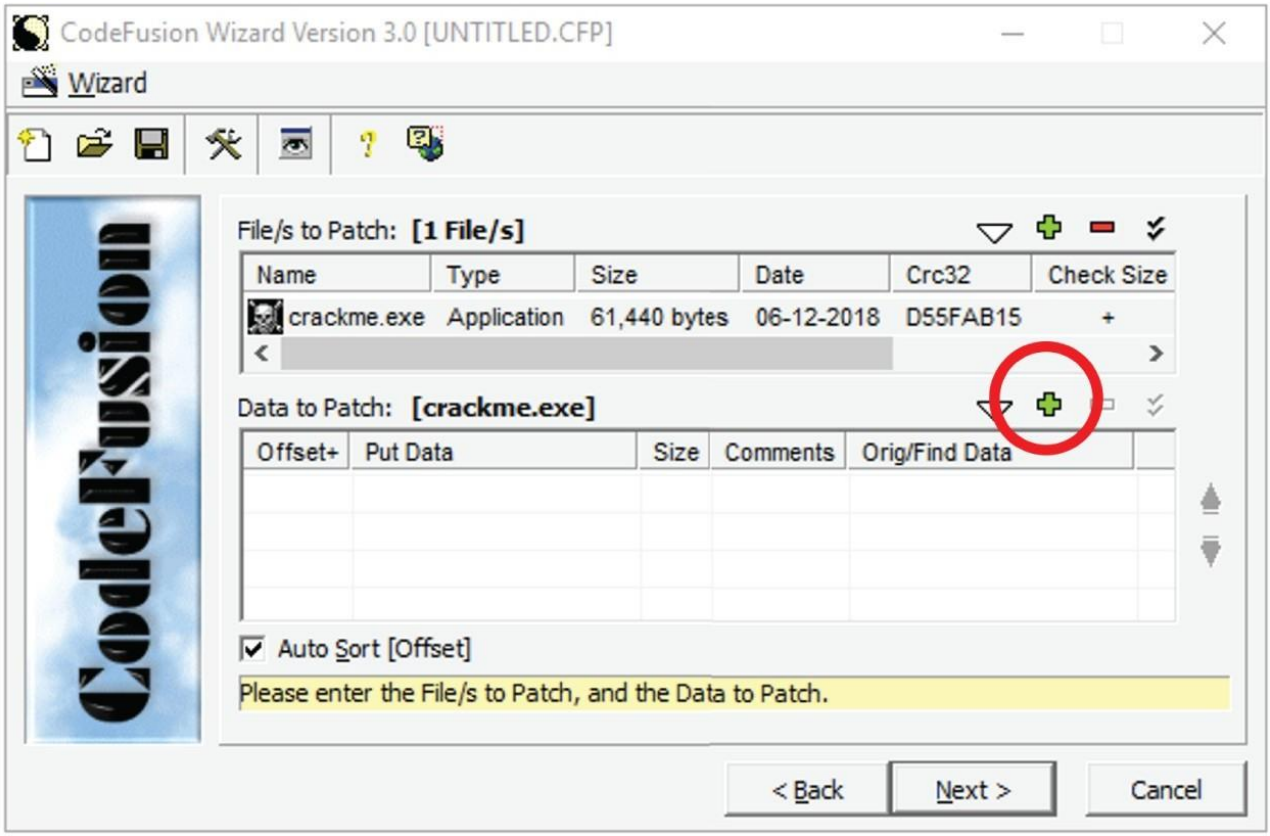


Рисунок 11.7: Добавление информации о патче в CodeFusion

CodeFusion добавит новый исполняемый файл вместе с целевым приложением. Как показано на рисунке 11.8, запустите этот исполняемый файл, выберите целевой файл и нажмите "Пуск", чтобы применить патч и взломать приложение.

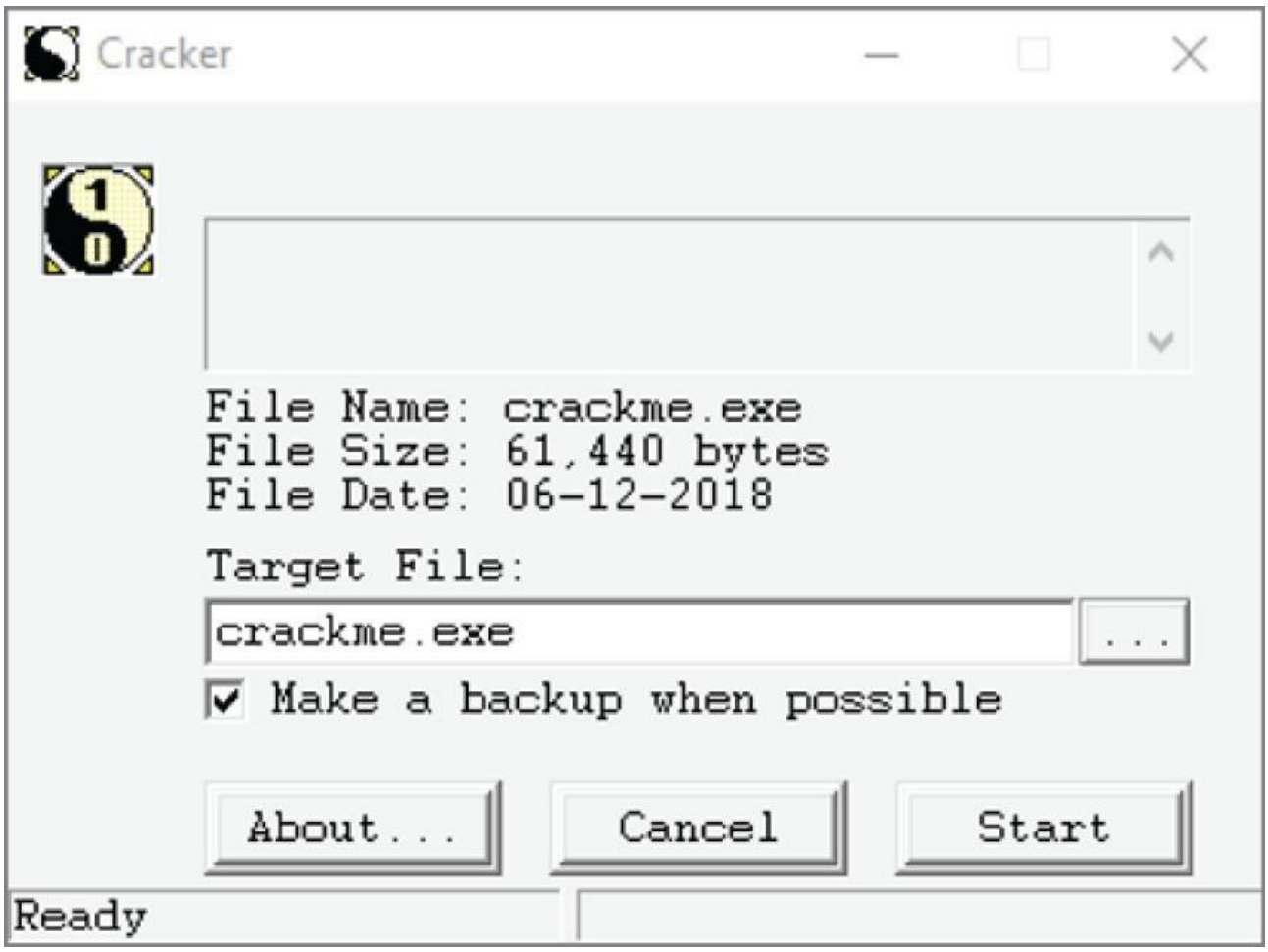


Рисунок 11.8: Запуск исправленного исполняемого файла в CodeFusion

Этот взломанный исполняемый файл - это то, что часто распространяет группа взломщиков. Он намного меньше и более переносим, чем полноценное взломанное приложение. Кому-то просто нужно установить приложение, загрузить ваш небольшой патчер и запустить его, и он выполнит исправление подлинного исполняемого файла.

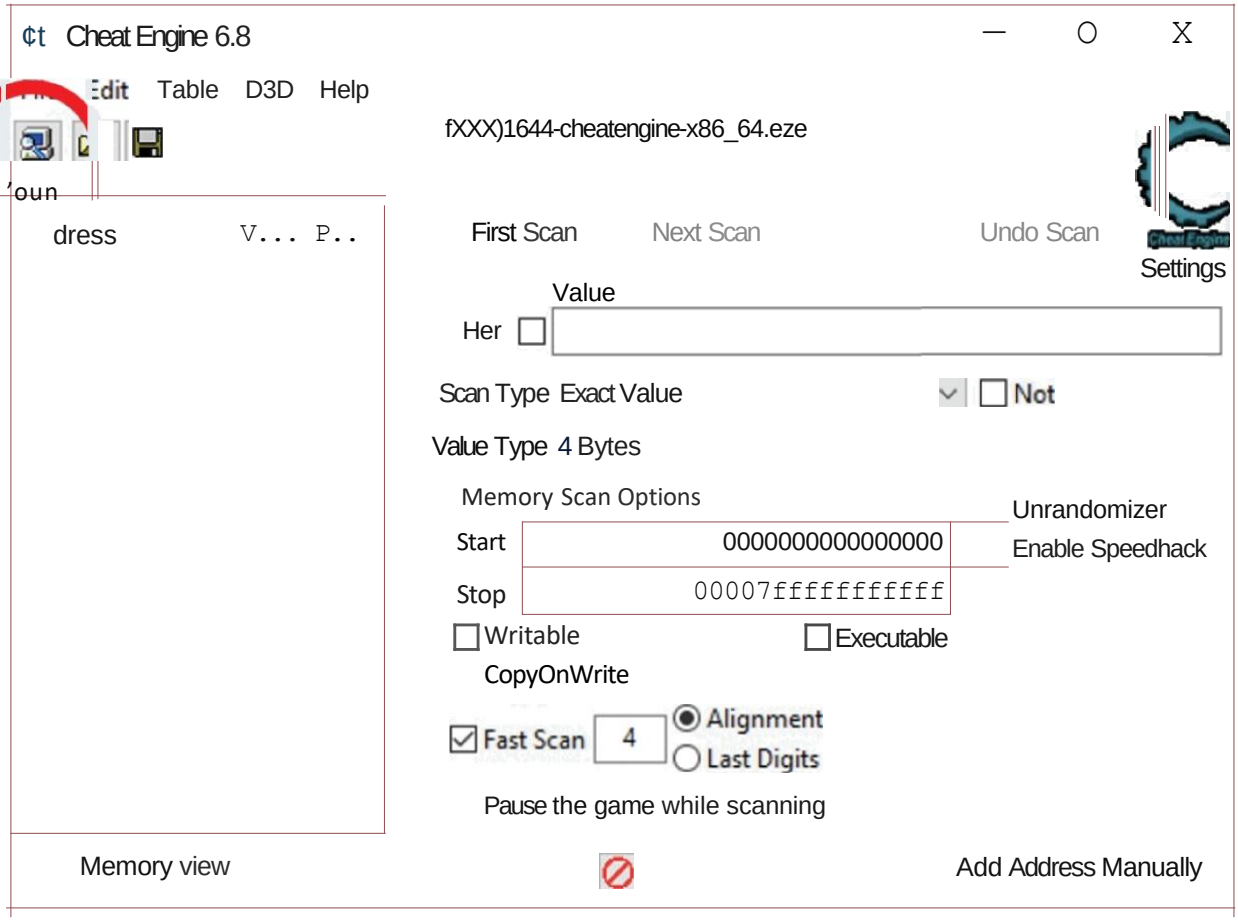
Cheat Engine
Cheat Engine - популярный и мощный сканер памяти с открытым исходным кодом, шестнадцатеричный редактор и отладчик. Хотя этот инструмент в основном используется для мошенничества в компьютерных играх, он также часто может быть полезен для быстрого динамического анализа при взломе программного обеспечения. (Найдите ссылки в разделе "Инструменты" на нашем GitHub здесь: https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES).

Cheat Engine позволяет выполнять поиск значений, введенных пользователем, с широким спектром опций. Они позволяют пользователю находить и сортировать данные в памяти компьютера.

Cheat Engine: Открытие процесс
В отличие от других инструментов, reversing with Cheat Engine не начинается с открытия исполняемого файла. Вместо этого вы выбираете запущенный процесс для редактирования.

Сначала запустите программу, которую вы хотите взломать. Затем запустите Cheat Engine и

нажмите "Выбрать процесс для открытия", как показано на рисунке 11.9. Появится окно списка процессов, в котором вы можете выбрать процесс для взлома и нажать "Открыть".


Cheat Engine 6.8
Edit
Table
D3D
Help
fXXX)1644-cheatengine-x86_64.eze
dress
V... P..
First Scan
Next Scan
Undo Scan
Settings
Value
Her
Scan Type Exact Value
Not
Value Type 4 Bytes
Memory Scan Options
Unrandomizer
Start
0000000000000000
Enable Speedhack
Stop
00007fffffffffff
Writable
Executable
CopyOnWrite
Fast Scan
4
Alignment
Last Digits
Pause the game while scanning
Memory view
Add Address Manually

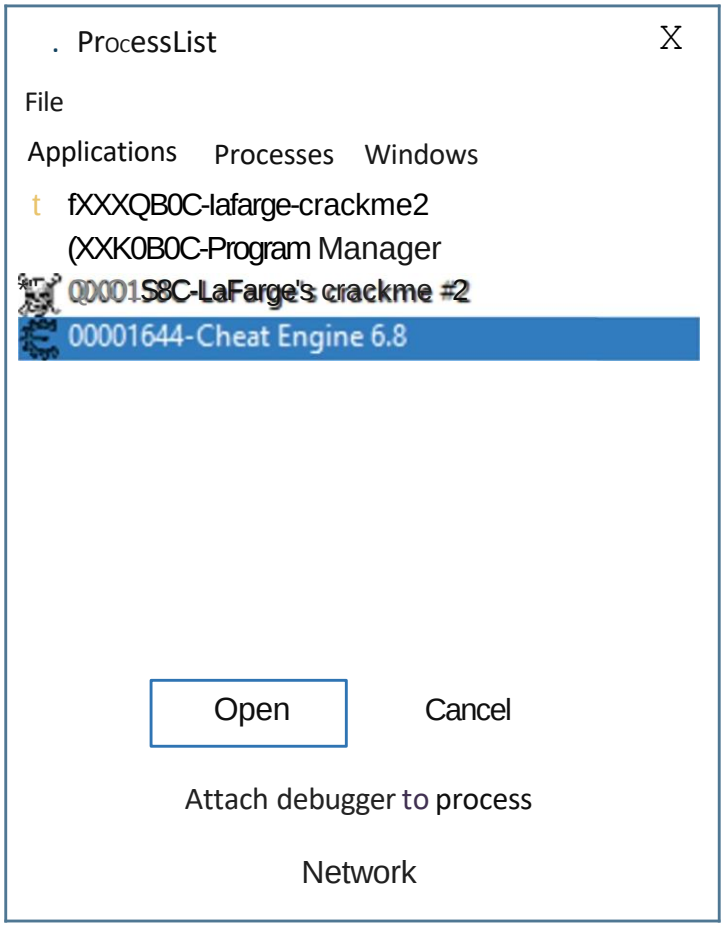

ProcessList
File
Applications
Processes
Windows
fXXXQB0C-lafarge-crackme2
(XXK0B0C-Program Manager
S8C-LaFarge's crackme #2
00001644-Cheat Engine 6.8
Open
Cancel
Attach debugger to process
Network

Рисунок 11.9: Открытие процесса в Cheat Engine

Cheat Engine: Просмотр памяти
Cheat Engine в значительной степени основан на идее сканирования памяти.

Главное окно Cheat Engine в основном используется для сканирования памяти. Однако сейчас сосредоточьтесь на более простой функциональности: просмотре памяти. Как показано на рис. 11.10, нажмите кнопку Просмотр памяти, чтобы просмотреть память процесса.

Представление памяти предоставляет простой и мощный способ просмотра, сканирования и модификации памяти процесса. Как показано на рисунке 11.11, представление памяти включает в себя разборку в верхней части экрана, ссылку на инструкцию в середине и шестнадцатеричный дамп внизу.

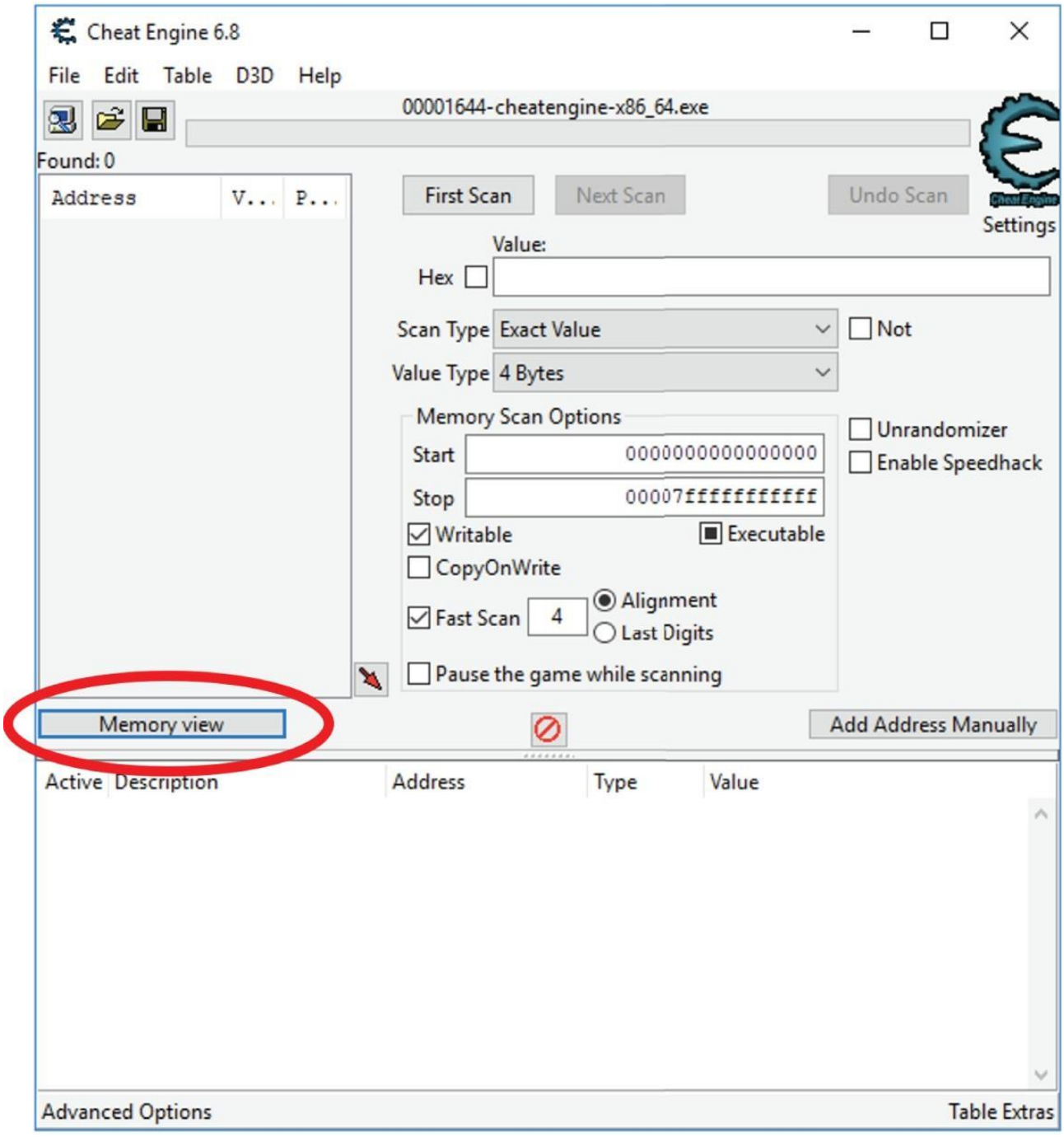


Рисунок 11.10: Просмотр памяти в Cheat Engine

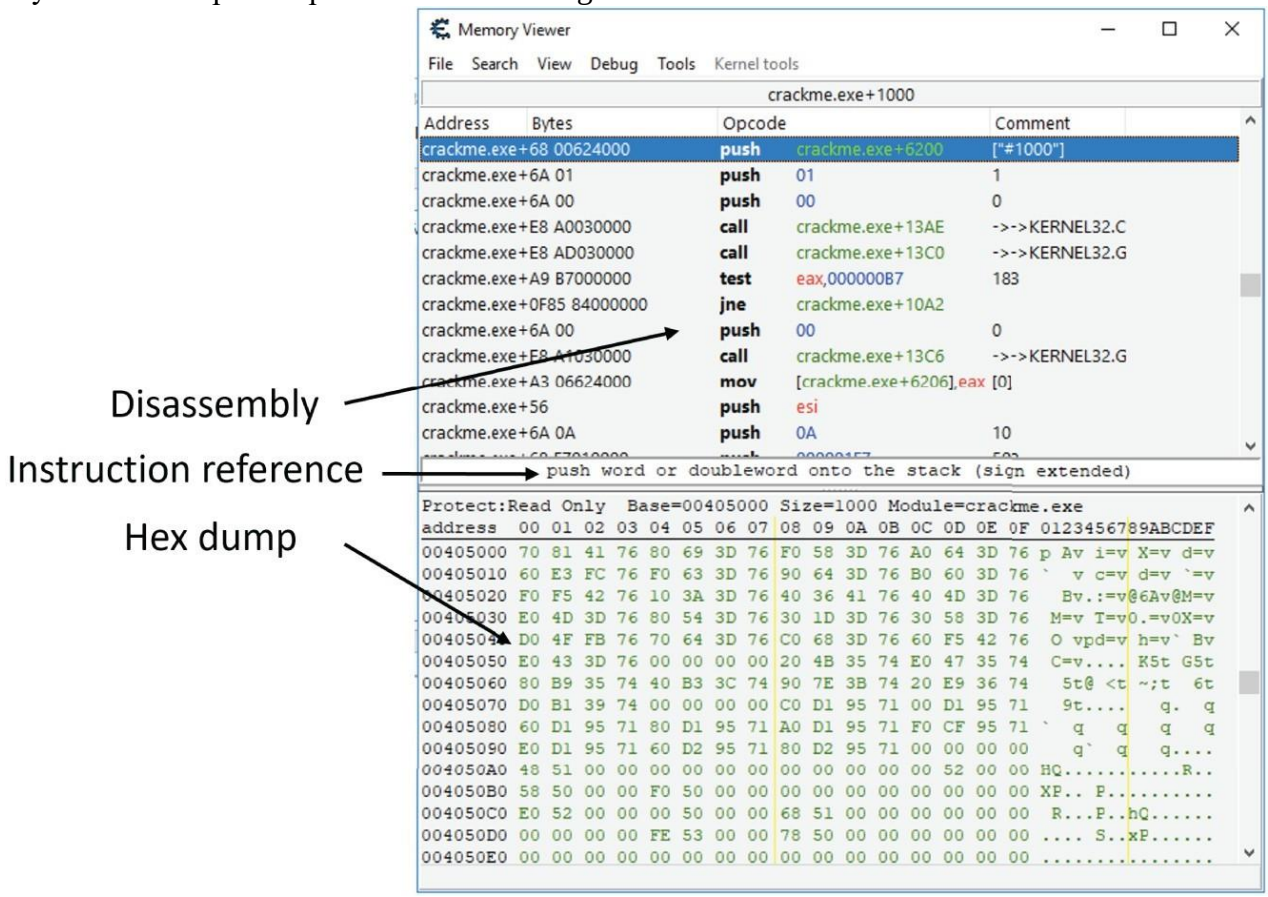


Рисунок 11.11: Панель просмотра памяти в Cheat Engine

Cheat Engine: Ссылки на строки

Как уже обсуждалось, изучение строк в исполняемом файле может дать бесценные подсказки относительно его функциональности. Чтобы просмотреть строки в Cheat Engine, выберите Просмотр ➪ Строки с ссылками, чтобы получить список всех строк, используемых программой.

На рисунке 11.12 показано всплывающее окно, в котором вы можете щелкнуть по строке, чтобы просмотреть перекрестные ссылки на нее. Дважды щелкните по адресу перекрестной ссылки, чтобы перейти к тому месту, где строка используется при дизассемблировании.

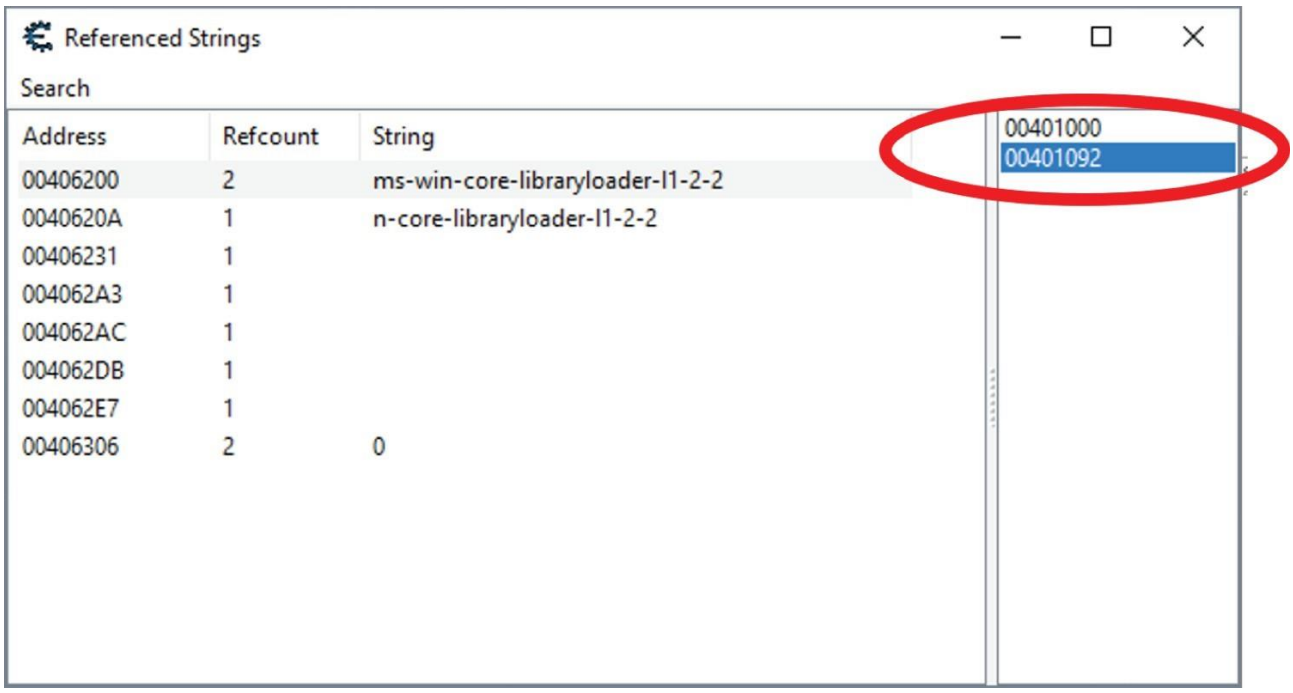


Рисунок 11.12: Ссылки на строки в Cheat Engine

Cheat Engine: переписывание программ

Напомним, что удаление фрагмента кода - самый безопасный и простой способ удалить его, не затрагивая остальную часть программы. Cheat Engine упрощает это. Чтобы обойти инструкцию (например, последний условный переход при проверке ключа), щелкните инструкцию правой кнопкой мыши и выберите Заменить кодом, который ничего не делает, как показано на рисунке 11.13.

Cheat Engine очень интерактивен. Вы можете немедленно попробовать свои изменения в запущенной программе! Если ваши изменения не сработали или вы хотите отменить их, щелкните правой кнопкой мыши измененный код и выберите Восстановить с помощью исходного кода, как показано на рисунке 11.14.

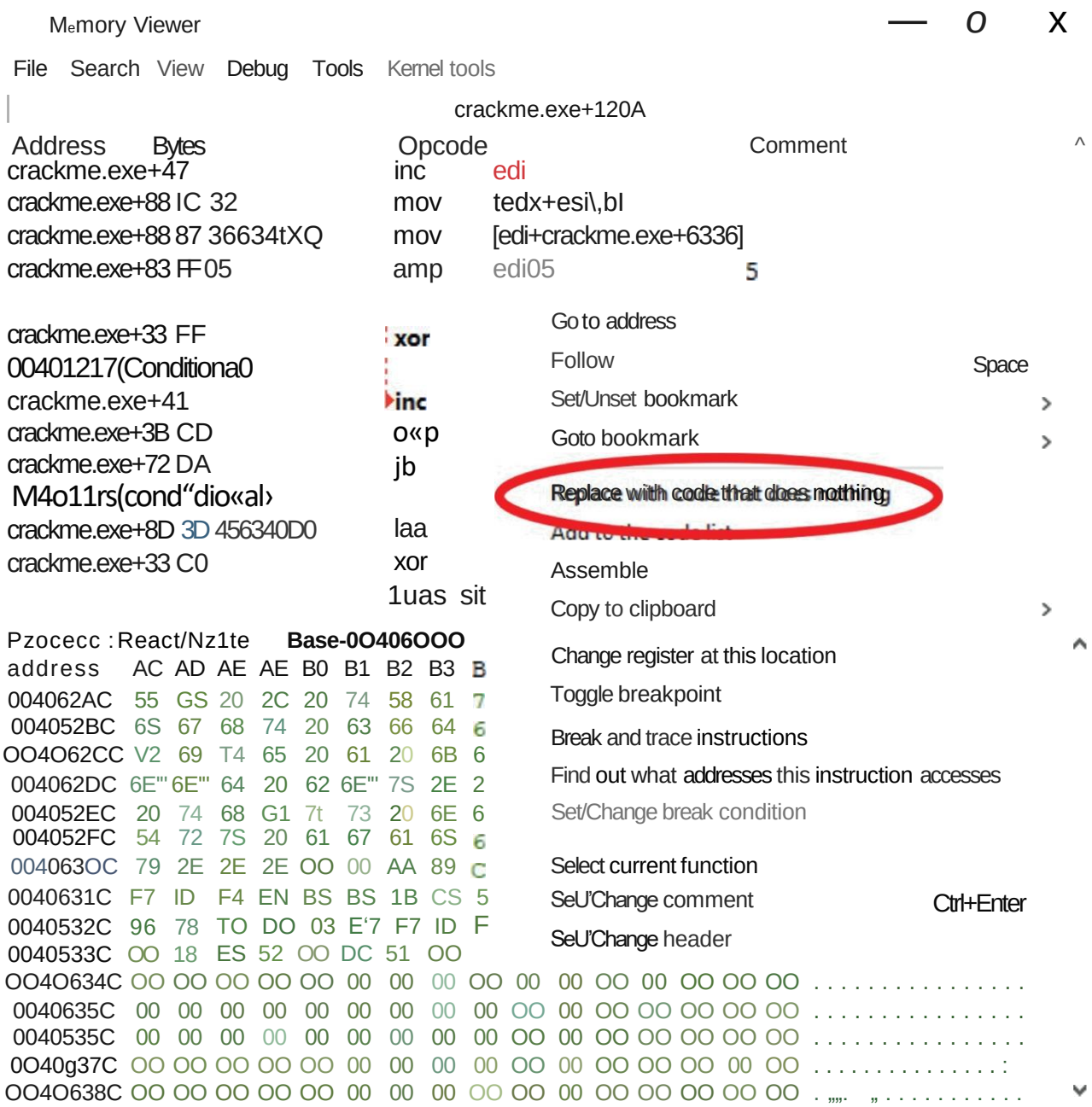

Memory Viewer
File Search View Debug Tools Kernel tools
crackme.exe+120A
Address Bytes Opcode Comment
crackme.exe+47 inc edi
crackme.exe+88 IC 32 mov tedx+esi\,bI
crackme.exe+88 87 36634tXQ mov [edi+crackme.exe+6336]
crackme.exe+83 FF05 amp edi05 5
crackme.exe+33 FF xor
00401217(Conditiona0
crackme.exe+41 inc
crackme.exe+3B CD o«p
crackme.exe+72 DA jb
M4o11rs(cond“dio«al›
crackme.exe+8D 3D 456340D0 laa
crackme.exe+33 C0 xor
1uas sit
Go to address
Follow Space
Set/Unset bookmark
Goto bookmark
Replace with code that does nothing
Assemble
Copy to clipboard
Change register at this location
Toggle breakpoint
Break and trace instructions
Find out what addresses this instruction accesses
Set/Change break condition
Select current function
SeU'Change comment Ctrl+Enter
SeU'Change header
Pzocecc : React/Nz1te Base-0O406OOO
address AC AD AE AE B0 B1 B2 B3
004062AC 55 GS 20 2C 20 74 58 61
004052BC 6S 67 68 74 20 63 66 64
OO4O62CC V2 69 T4 65 20 61 20 6B
004062DC 6E"' 6E"' 64 20 62 6E"' 7S 2E
004052EC 20 74 68 G1 7t 73 20 6E
004052FC 54 72 7S 20 61 67 61 6S
004063OC 79 2E 2E 2E OO 00 AA 89
0040631C F7 ID F4 EN BS BS 1B CS
0040532C 96 78 TO DO 03 E‘7 F7 ID
0040533C OO 18 ES 52 OO DC 51 OO
OO4O634C OO OO OO OO OO 00 00 00 OO 00 00 OO 00 OO OO OO
0040635C 00 00 00 00 00 00 00 00 00 OO 00 OO OO OO OO OO
0040535C 00 00 00 00 00 00 00 00 00 OO 00 OO OO OO OO OO
0O40g37C OO OO OO OO OO 00 00 00 00 OO 00 OO OO OO 00 OO
OO4O638C OO OO OO OO OO 00 00 00 OO OO 00 OO OO OO OO OO

Рисунок 11.13 удаление инструкций в Cheat Engine

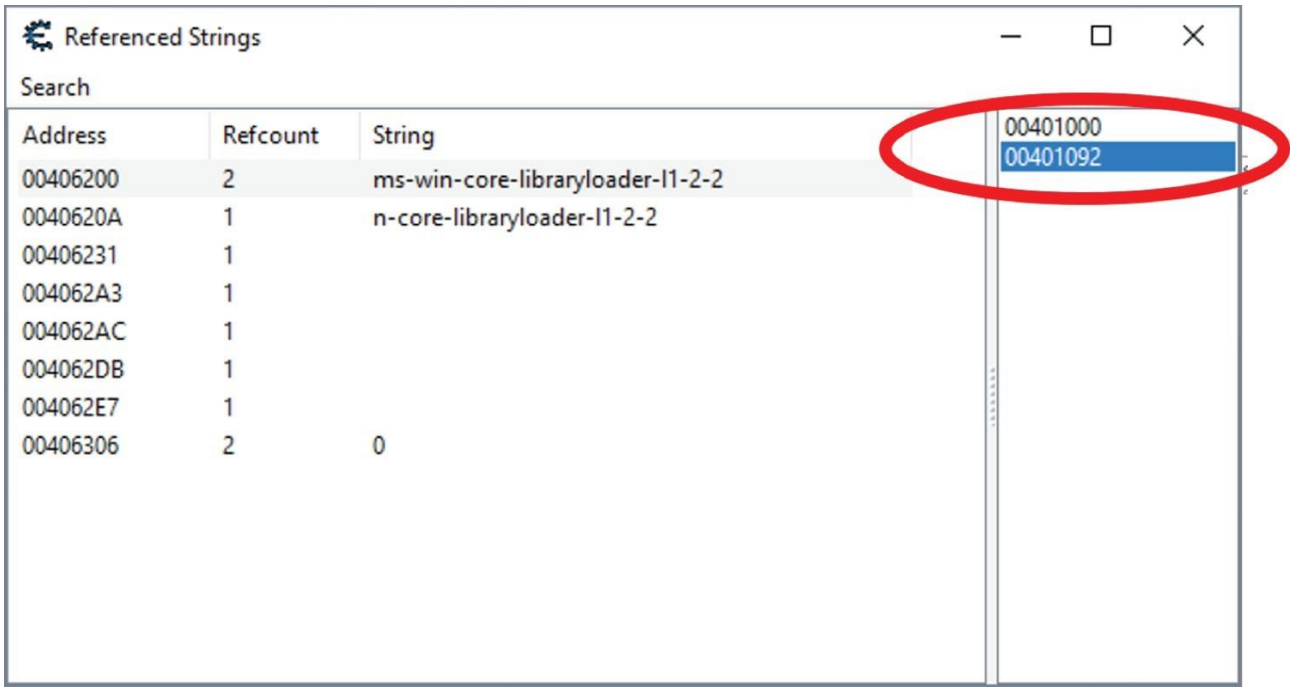


Рисунок 11.12: Ссылки на строки в Cheat Engine

Cheat Engine: переписывание программ
Напомним, что удаление фрагмента кода - самый безопасный и простой способ удалить его, не затрагивая остальную часть программы. Cheat Engine упрощает это. Чтобы обойти инструкцию (например, последний условный переход при проверке ключа), щелкните инструкцию правой кнопкой мыши и выберите Заменить кодом, который ничего не делает, как показано на рисунке 11.13.

Cheat Engine очень интерактивен. Вы можете немедленно попробовать свои изменения в запущенной программе! Если ваши изменения не сработали или вы хотите отменить их, щелкните правой кнопкой мыши измененный код и выберите Восстановить с помощью исходного кода, как показано на рисунке 11.14.

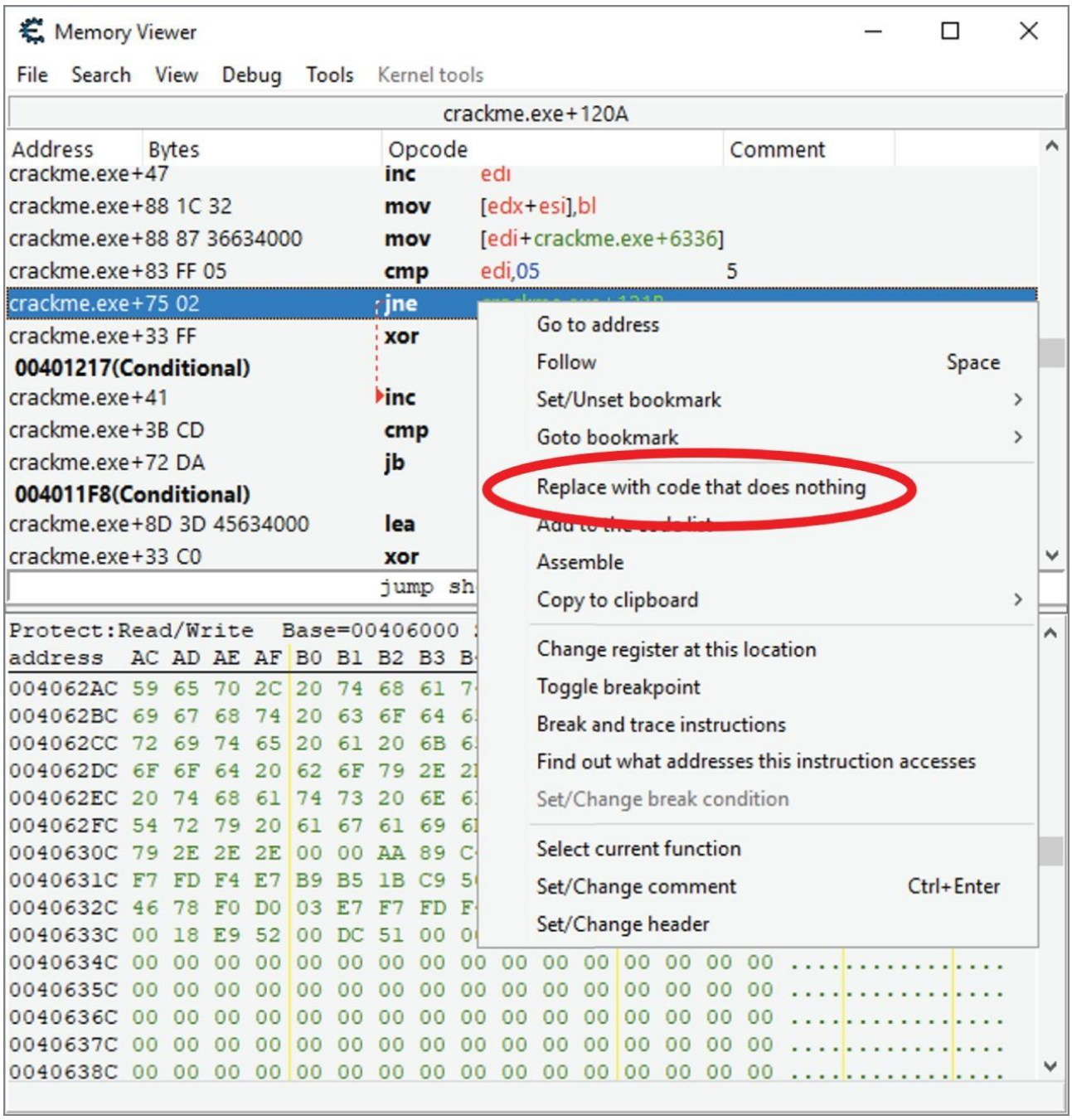


Рисунок 11.13 удаление инструкций в Cheat Engine

Cheat Engine: Копирование байтов

Как только вы нашли работающий патч, следующий шаг - скопировать этот патч в исполняемый файл, а не в запущенный процесс. Как показано на рисунке 11.15, вы можете щелкнуть правой кнопкой мыши местоположение исправления и выбрать Копировать в буфер обмена ➪ Только байты, чтобы скопировать эти байты для использования другими инструментами.

Механизм обмана: Получение адресов

Чтобы создать исправление, вам нужно знать, где в файле находятся данные для исправления. Cheat Engine полностью основан на анализе во время выполнения, поэтому он не знает, где в файле находятся данные.

Чтобы найти адрес, используйте редактор 010 для выполнения поиска машинного кода, который вы заменяете. Этот адрес является смещением файла для исправления для использования в Coldfusion или других исправлениях.

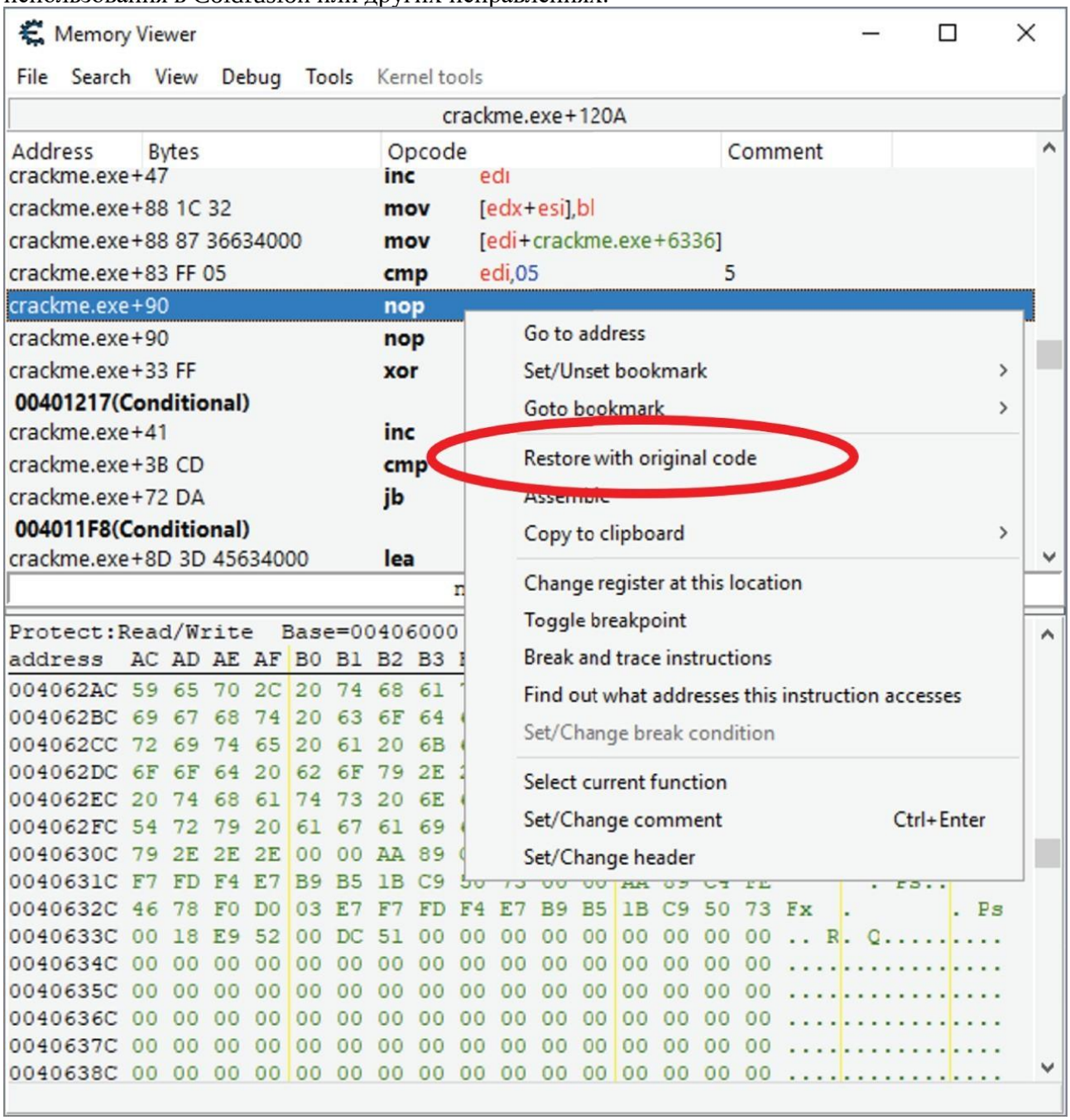


Рисунок 11.14: Отмена изменений в Cheat Engine

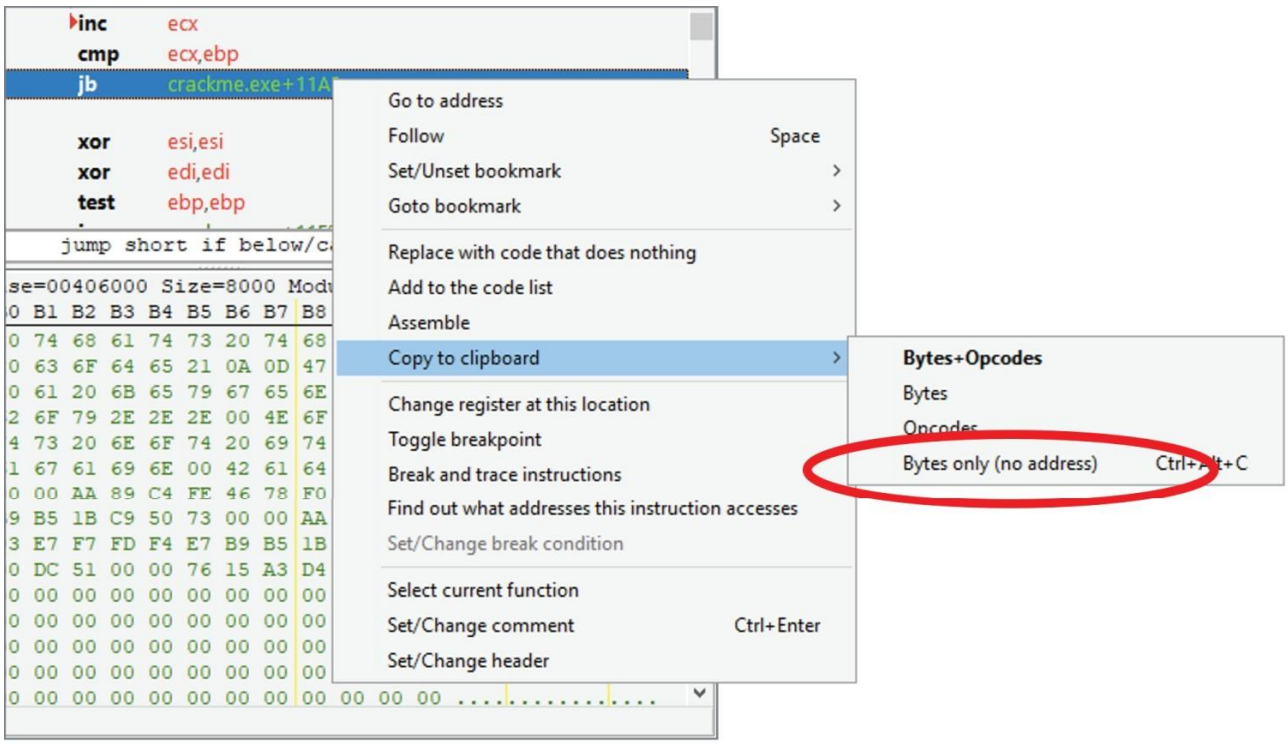


Рисунок 11.15: Копирование байтов в Cheat Engine

Лабораторная работа: Взлом LaFarge
В этой лабораторной работе практикуется использование этих инструментов для исправления программ. Лабораторные работы и все связанные с ними инструкции можно найти в соответствующей папке здесь:

https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES

Что касается этой лаборатории, пожалуйста, найдите лабораторию LaFarge и следуйте предоставленным инструкциям.

Навыки
В этой лабораторной работе вы получите опыт использования CodeFusion и Cheat Engine для отработки следующих навыков:

- Реверс-инжиниринг x86
- Исправление и патчеры
- Выводы

Для реверс-инжиниринга и взлома доступно множество инструментов; выбор “правильного” из них зависит от конкретной задачи и личных предпочтений. Взломщики - это (как правило) безопасный, всегда легальный и невероятно захватывающий способ практиковать свои навыки взлома.

Введение в IDA
Если вы когда-нибудь гуглили инструменты реверс-инжиниринга, IDA гарантированно появится. Это "Кадиллак" инструментов реверс-инжиниринга.

IDA, он же интерактивный дизассемблер, позволяет визуализировать дизассемблирование в двоичном формате. Он доступен по модели freemium, где ограниченные функции доступны бесплатно, в то время как для некоторых более мощных функций (или более малоизвестных архитектур) требуется платная лицензия.

На рисунке 11.16 показан процесс загрузки нового файла в IDA. IDA автоматически распознает многие распространенные форматы файлов, но если программа распознает их неправильно, вы можете выбрать общий двоичный файл. IDA также предлагает раскрывающееся меню Типа процессора для изменения архитектуры.

Одной из самых сильных сторон IDA является ее графический вид, который показывает визуальное представление сборки x86 исполняемого файла и потоков управления. На рисунке 11.17 показано это представление и некоторые из наиболее полезных его компонентов, включая карту памяти исполняемого файла, список функций, представление логического блока и окно графика.

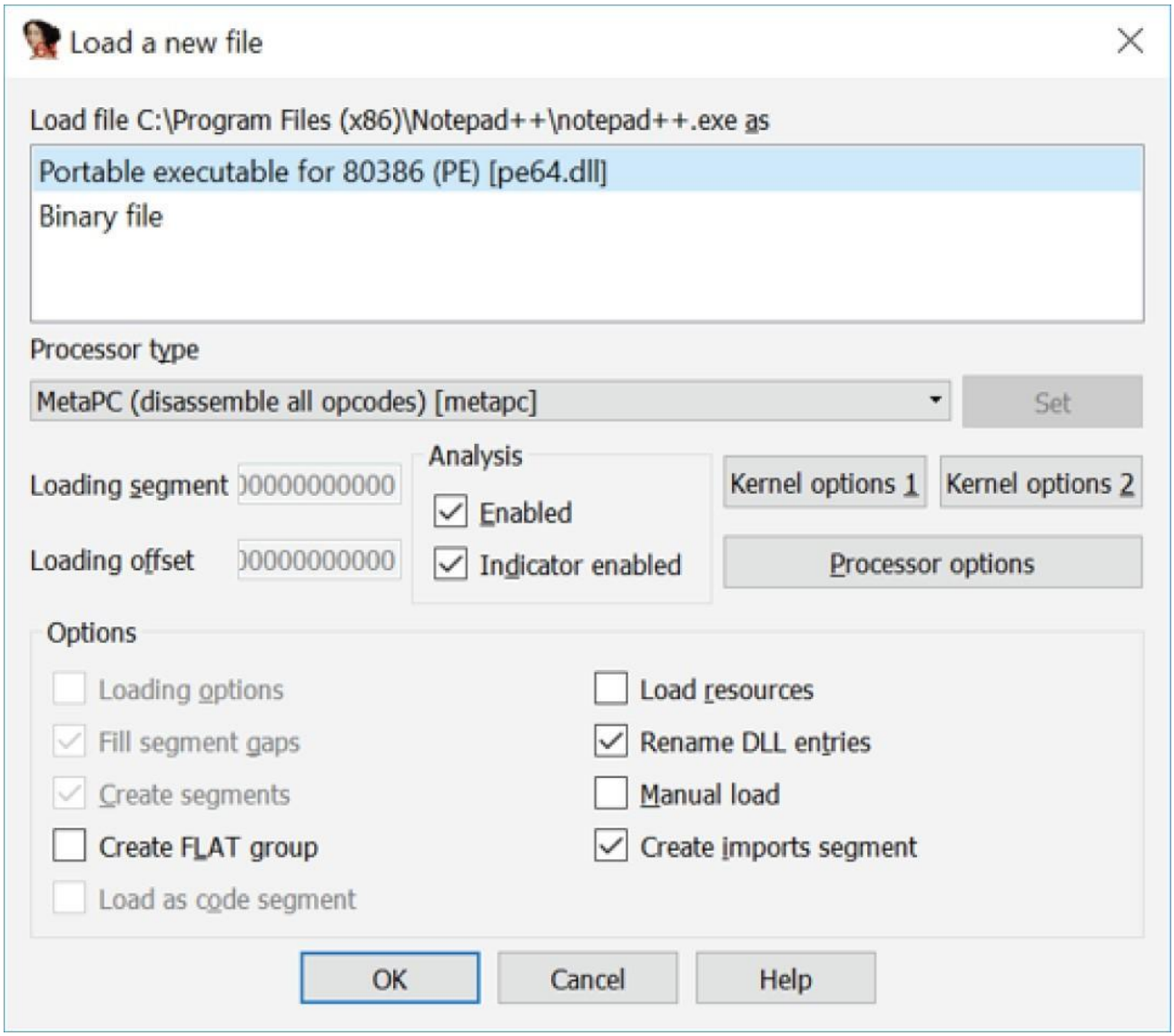


Рисунок 11.16: Загрузка файла в IDA

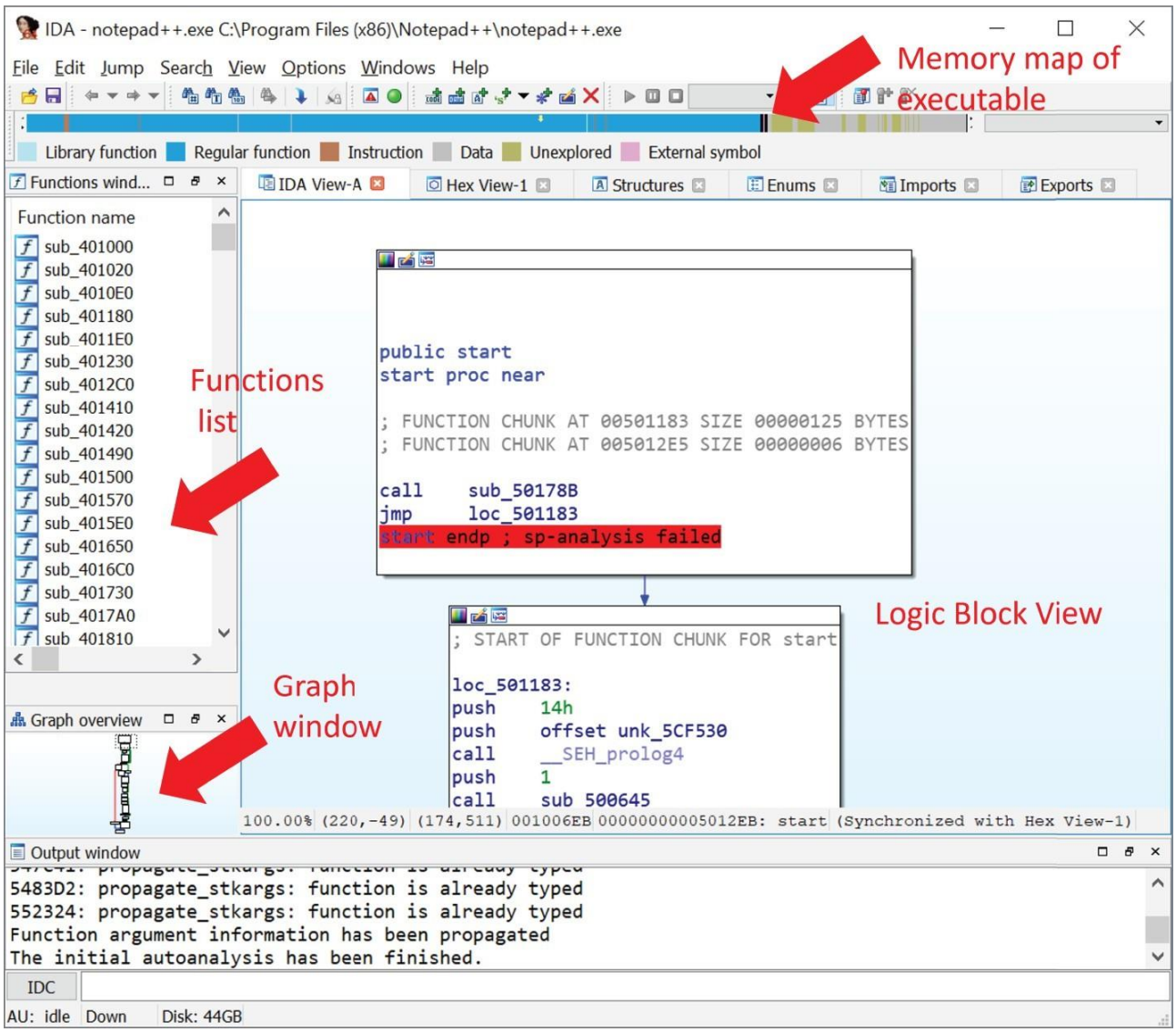


Рисунок 11.17: Представление графика IDA

IDA: Строки

Как всегда, строки являются хорошей отправной точкой при анализе нового исполняемого файла. Однако IDA не отображает их по умолчанию. На рисунке 11.18 показано, как получить доступ к представлению строк, нажав Просмотр ➪ Открыть вложенные представления ➪ Строки.

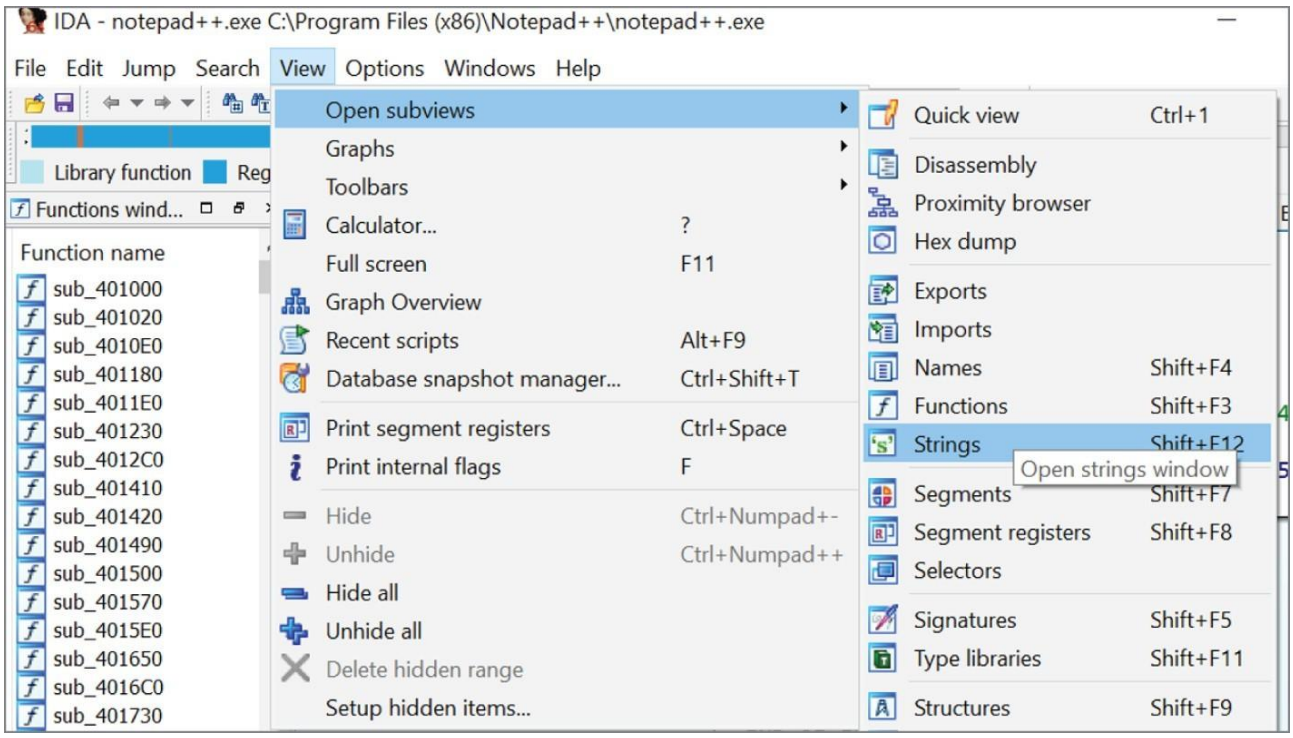


Рисунок 11.18: Открытие представления строк в IDA

На рисунке 11.19 показан полный список строк в IDA. IDA показывает текст самой строки, ее адрес и прогнозируемую длину.

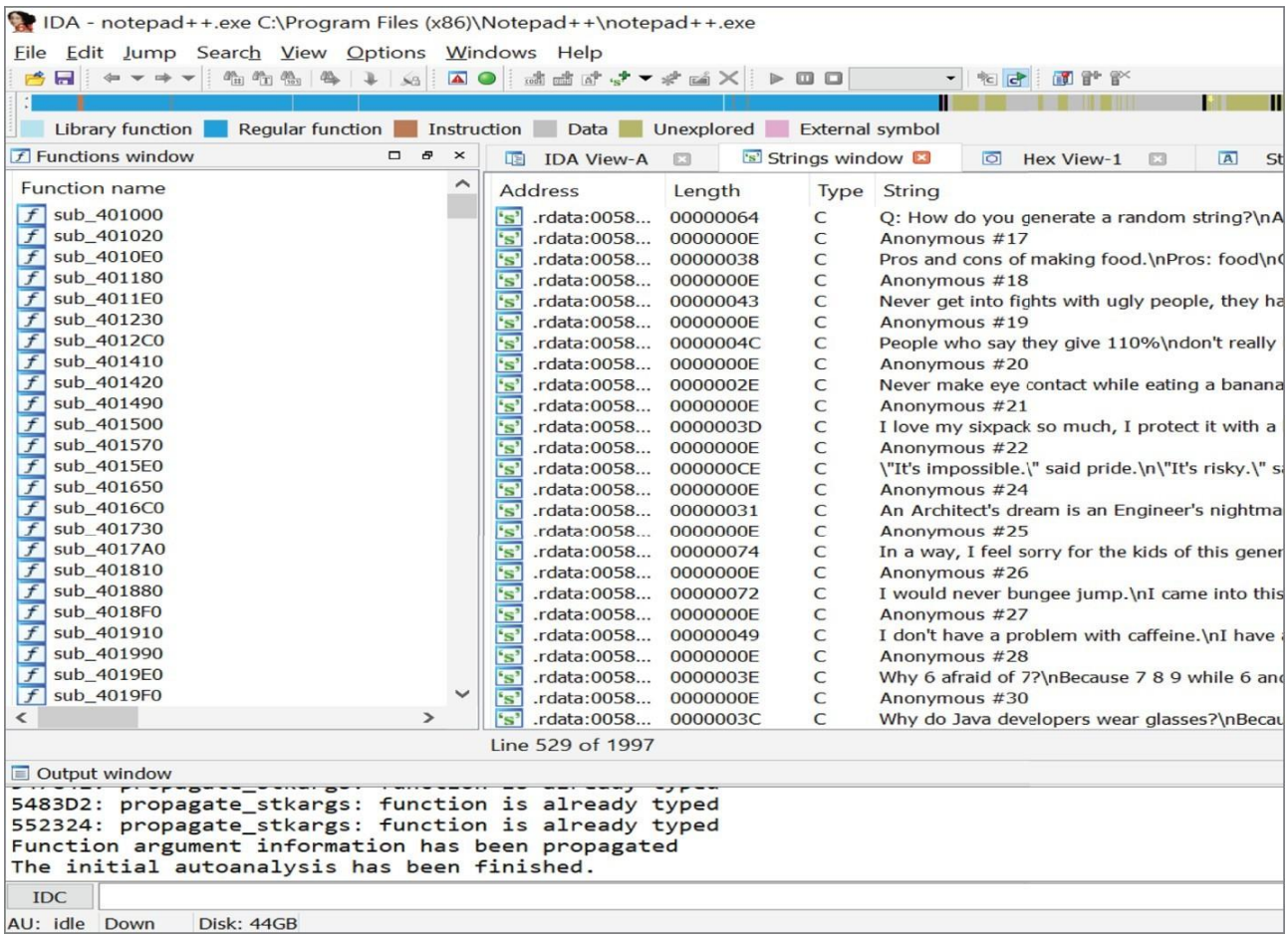


Рисунок 11.19: Просмотр строк в IDA

Щелкните строку, чтобы выделить ее. Затем нажмите X или щелкните правой кнопкой мыши и выберите Перейти к внешней ссылке на операнд. Откроется окно, показывающее все места, где строка используется в программе, как показано на рис. 11.20.

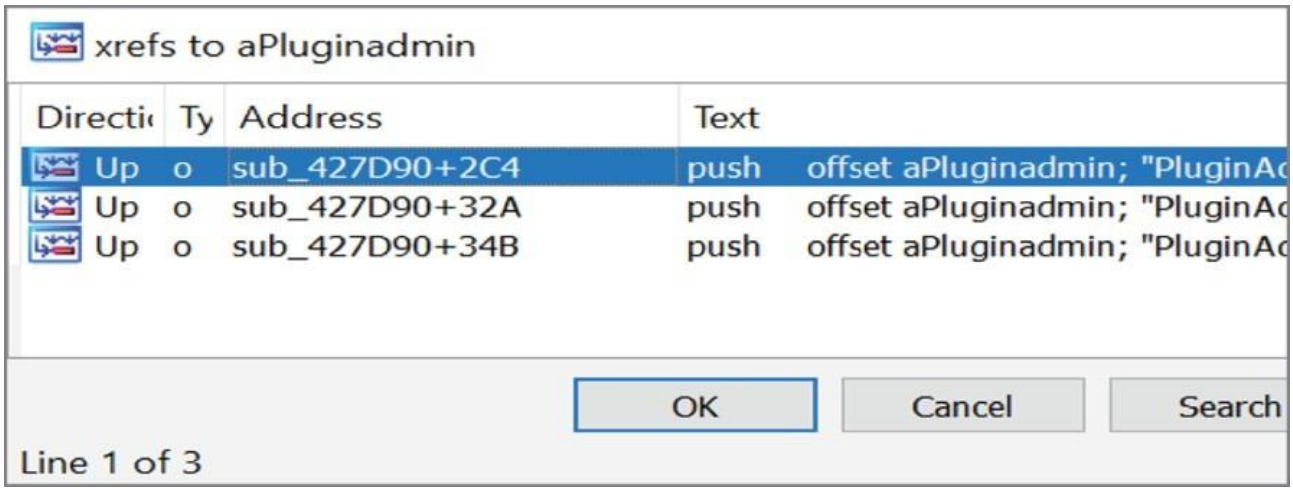


Рисунок 11.20: Перекрестные ссылки на строки в IDA

После одной из этих перекрестных ссылок будет показана разборка, в которой используется

строка. Как показано на рисунке 11.21, IDA понимает, как работают ссылки на строки. Когда он видит один из них, он показывает строку в качестве комментария.

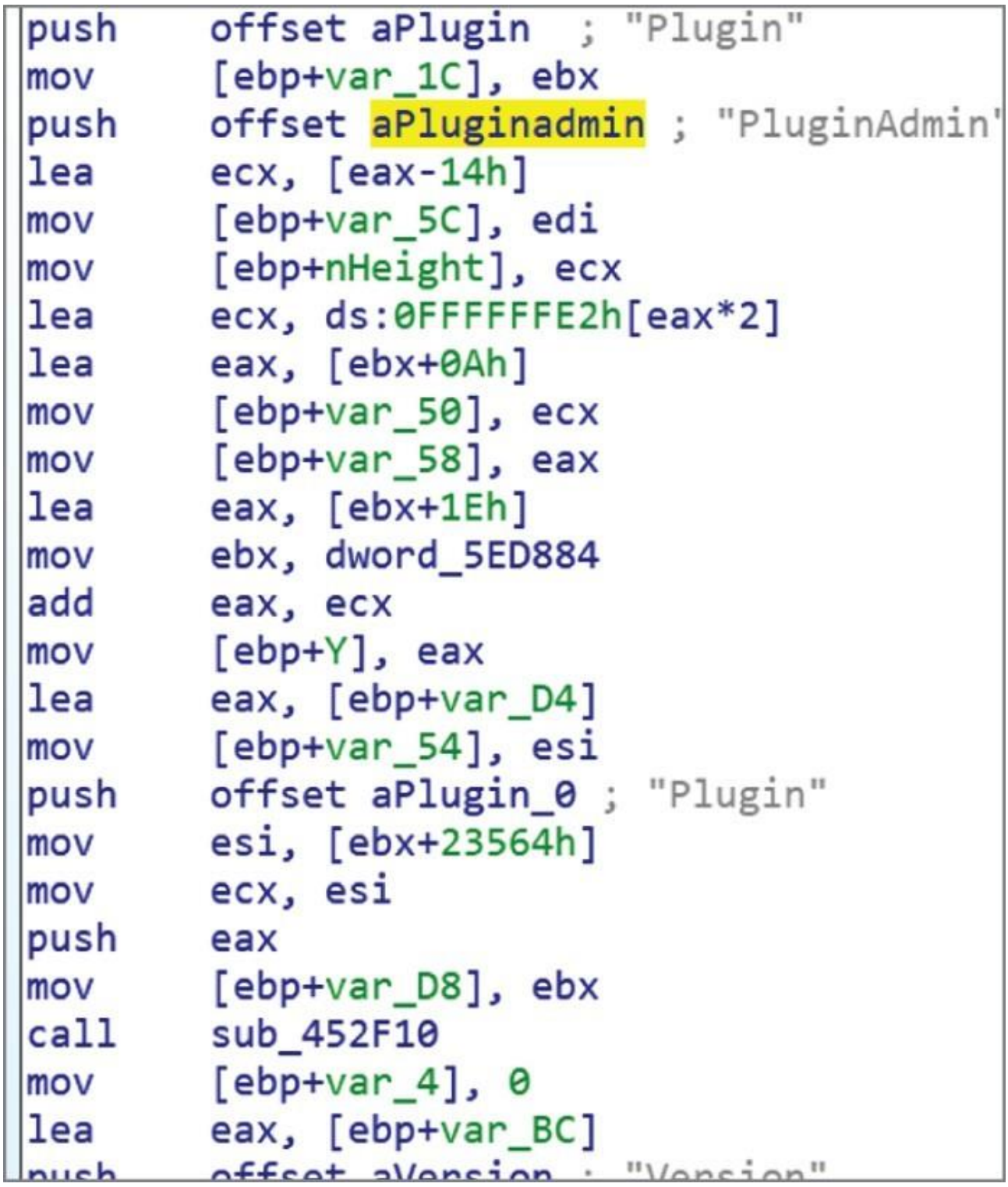


Рисунок 11.21: Строки в представлении кода IDA

IDA: Базовые блоки
Графический вид IDA показывает код в базовых блоках. Базовый блок - это непрерывная последовательность инструкций, не прерываемая инструкцией ветвления или ссылкой на ветвление.

Рассмотрим следующую простую программу в псевдокоде. На рисунке 11.22 показано, как выглядит эта программа при дизассемблировании в IDA.

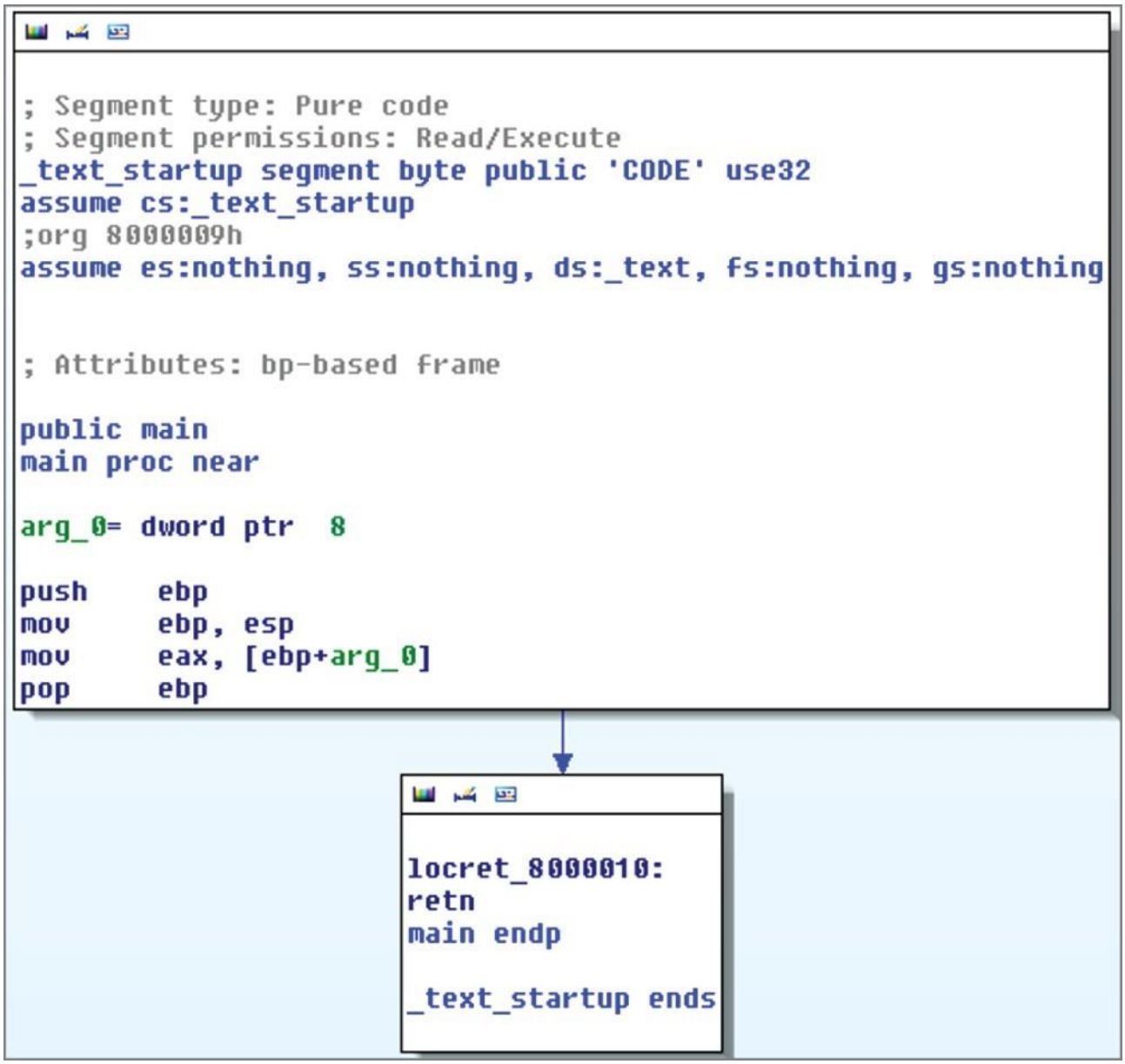


Рисунок 11.22: Основные блоки в IDA

```
int main(int argc, char* argv[])
 {
       return argc;
 }
```

IDA: Функции и переменные

IDA понимает многие соглашения о вызовах, включая cdecl. Он распознает cdecl и знает, что первый аргумент всегда начинается с ebp+8. IDA переименовывает это смещение в arg_0, чтобы упростить чтение. Он выполнит это переименование со всеми входными переменными (arg_X), как показано на рисунке 11.23.

Это понимание также распространяется на то, как обрабатываются локальные переменные в

стеке. Например, как показано на рисунке 11.24, IDA переименует локальные переменные в var_X.

Знание того, как IDA помечает аргументы и переменные, может значительно помочь в анализе функций. Например, с функцией, показанной на рисунке 11.25, мы можем очень быстро определить, что у нее есть одна локальная переменная и шесть входных переменных, потому что мы знаем, как IDA использует свои соглашения об именовании.

Часто IDA не располагает информацией о назначении или контексте, в котором используются эти переменные, поэтому помечает их последовательно. Когда вы узнаете об аргументе, переменной или функции, вы можете переименовать их, нажав N или щелкнув правой кнопкой мыши метку переменной и выбрав Переименовать.

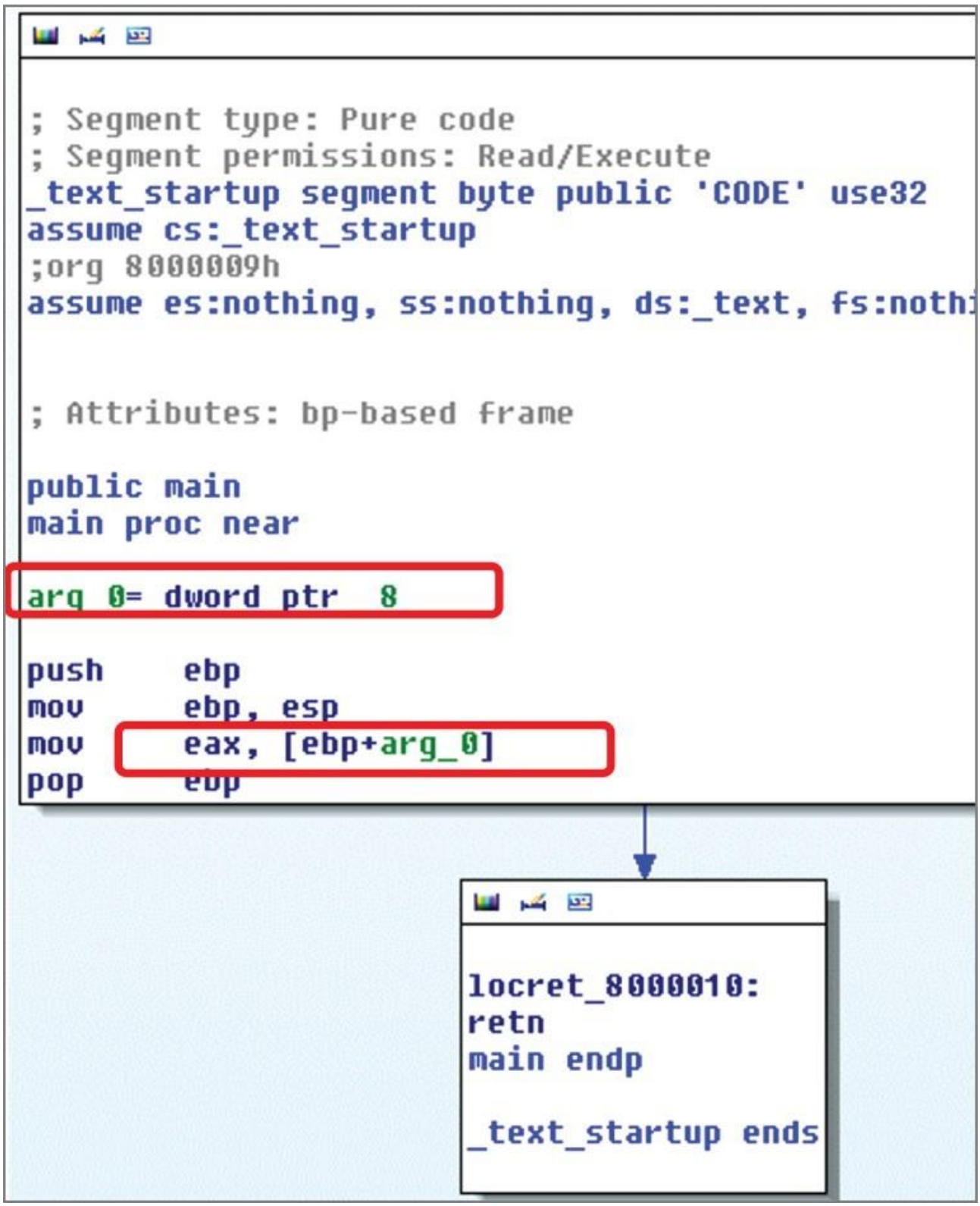


Рисунок 11.23: Аргументы функции в IDA

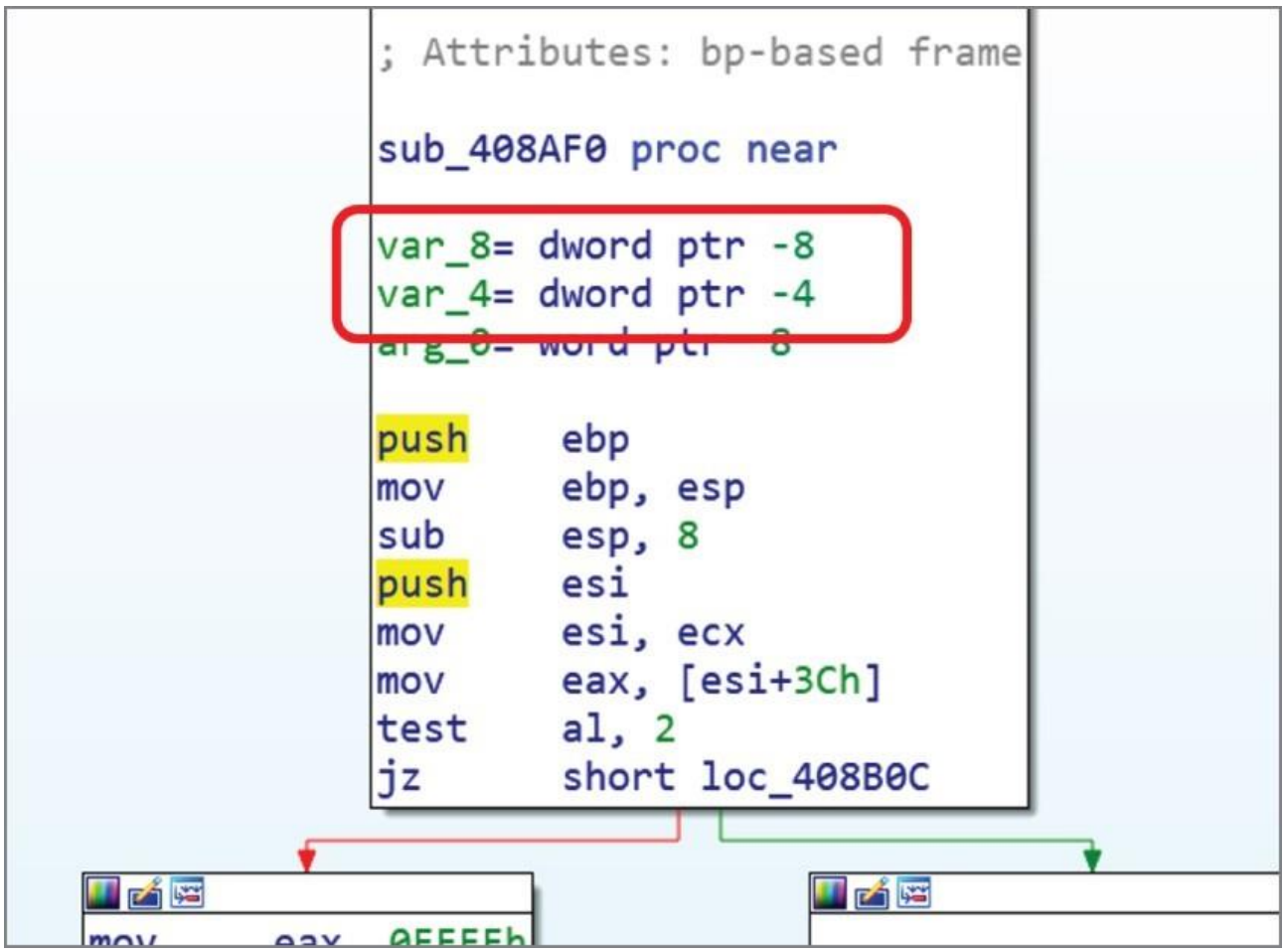


Рисунок 11.24: Локальные переменные в IDA

```
sub_408FB0 proc near

var_4= dword ptr -4
arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= dword ptr  14h
arg_10= dword ptr  18h
arg_1C= dword ptr  24h

push    ebp
```

Рисунок 11.25: Локальные переменные и аргументы функции в IDA

IDA: Комментарии
При реверсировании приложения важно иметь возможность отслеживать, что вы выяснили и сделали на данный момент. В IDA нажатие ; открывает окно для ввода комментариев, как показано на рисунке 11.26.

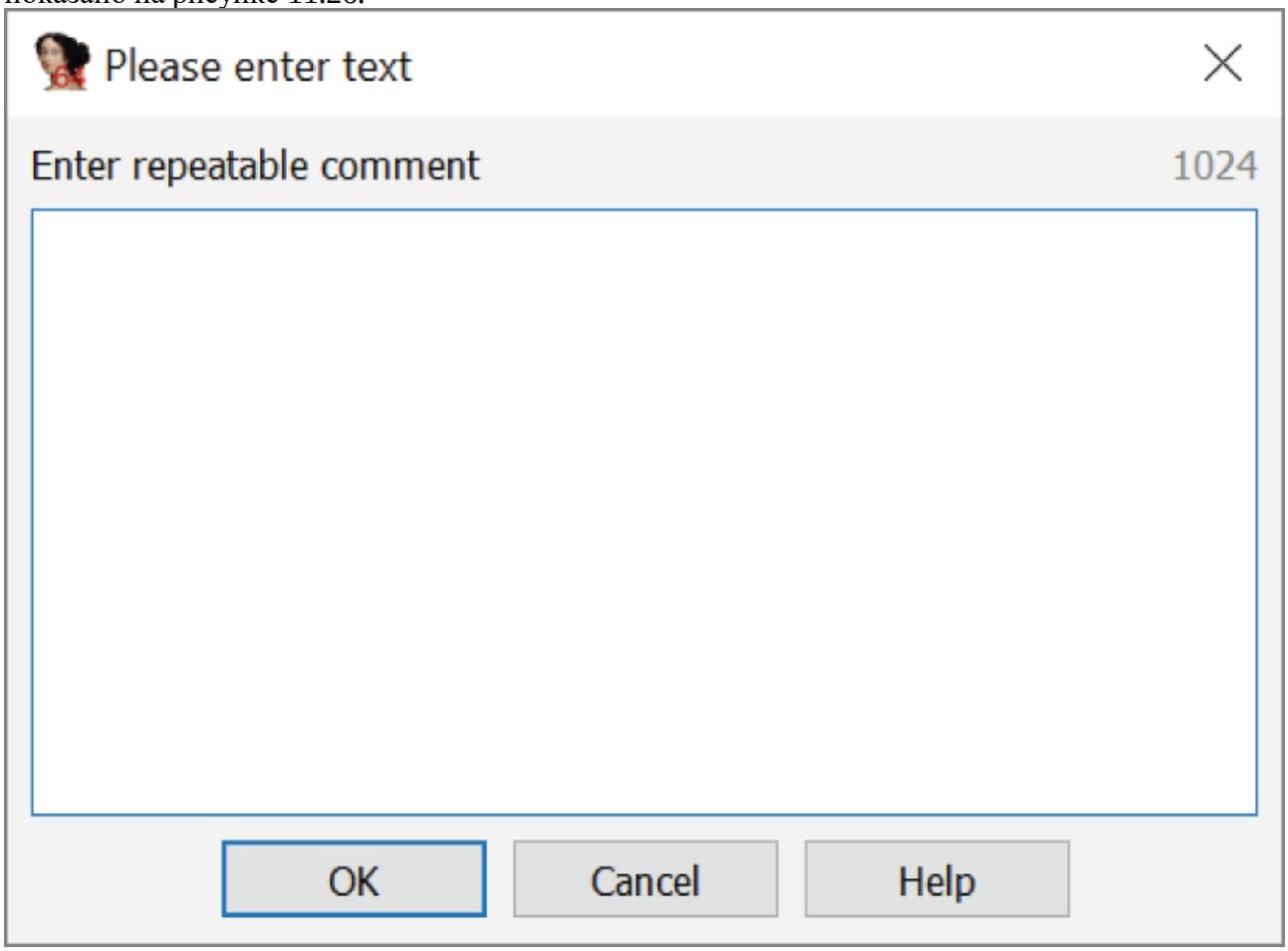


Рисунок 11.26: Окно комментариев IDA

Один из советов заключается в том, чтобы указывать идентификатор, подобный “_x”, во всех ваших комментариях. Это дает вам что-то для поиска, чтобы найти все комментарии.

Чтобы начать поиск комментариев, выберите Поиск по тексту, как показано на рисунке 11.27. Затем выполните поиск по “_x”, выбрав "Найти все вхождения", чтобы найти все комментарии, которые вы разместили в программе.

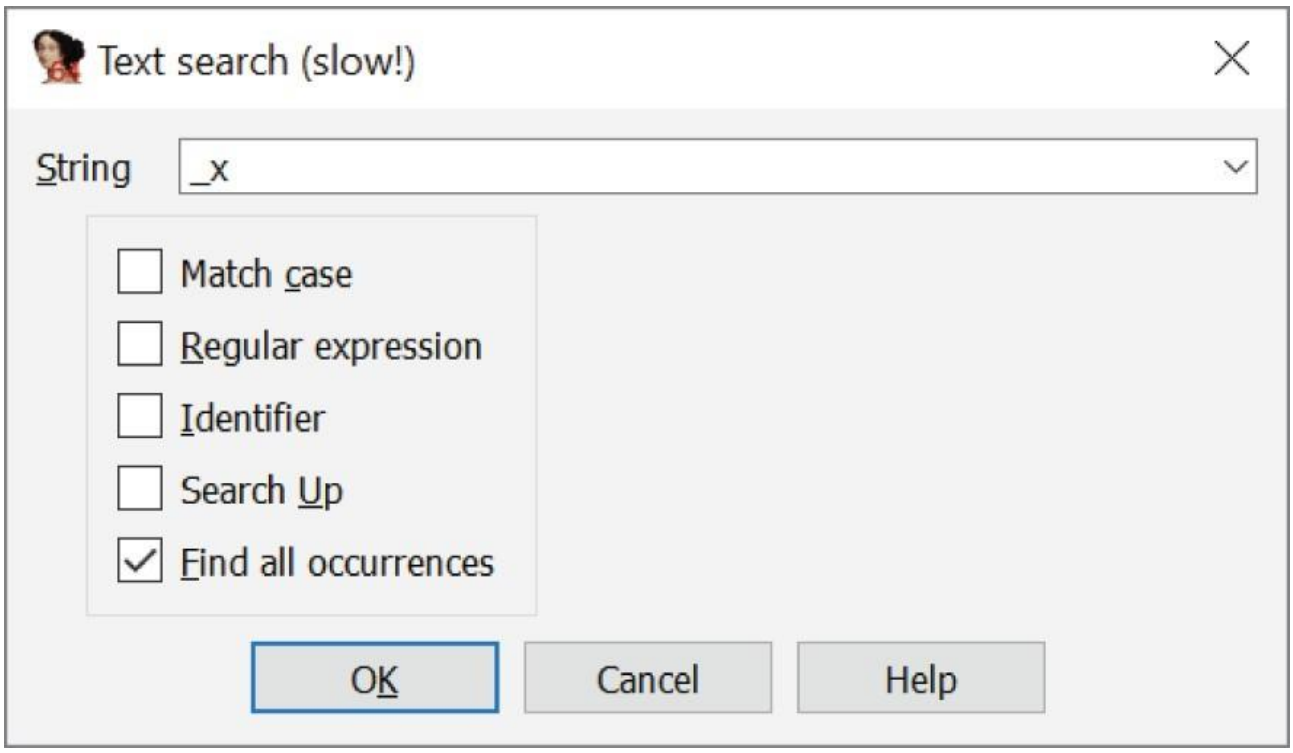


Рисунок 11.27: Поиск комментариев в IDA

Используя согласованный стиль комментирования и поиск комментариев, легко найти места в коде, которые вы уже изучили. Например, как показано на рис. 11.28, вы можете быстро определить местоположения, которые были помечены как “TODO” для последующего анализа.

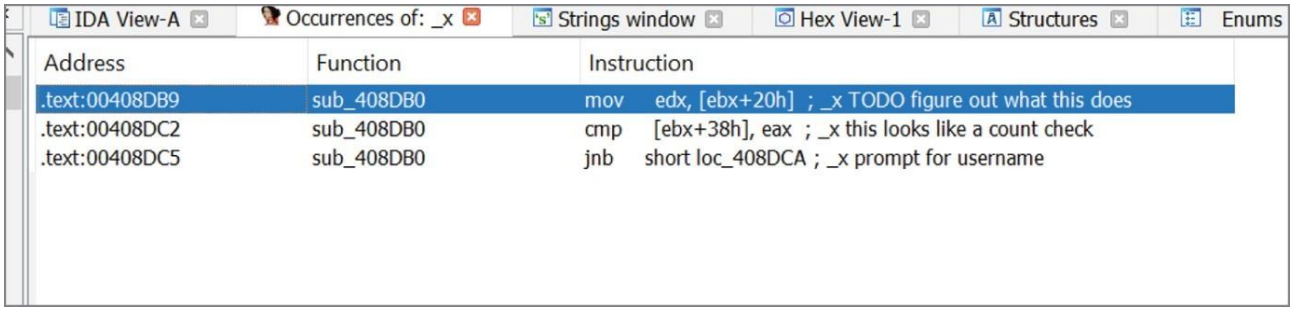


Рисунок 11.28: Результаты поиска в IDA

IDA: Пути
IDA показывает три типа путей между базовыми блоками:

- Красный: Путь, выбранный, если не выполняется условный переход
- Зеленый: Путь, выбранный, если выполняется условный переход
- Синий: Гарантированный путь (без условий)

Для примера рассмотрим следующий пример кода, содержащий простую инструкцию if:

```
int main(int argc, char* argv[])
{
        if (argc> 1)
                return 0;
```

```
        return argc;
}
```

На рисунке 11.29 показано, как этот код выглядел бы в IDA. После условного блока пути расходятся. Цвета в этой книге не показаны, но левый путь, который в IDA красный, показывает, что произойдет, если переход не будет выполнен. Правильный путь, который в IDA обозначен зеленым цветом, выполняется, если условие принимает значение false.

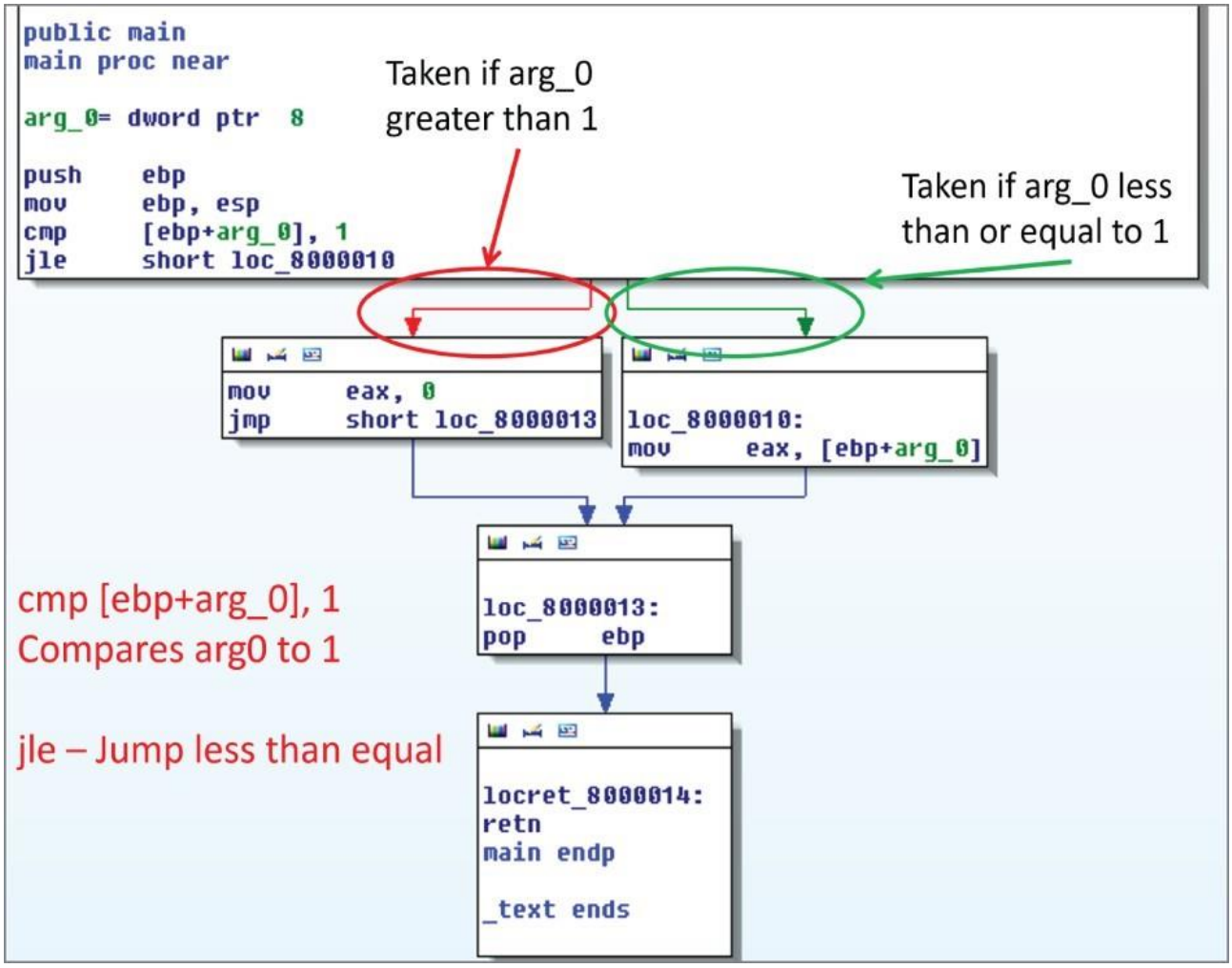


Рисунок 11.29: Пути к коду в IDA

Ниже этой точки еще несколько стрелок указывают переходы между базовыми блоками. Поскольку ни один из них не содержит условных обозначений, все они будут синими в IDA.

Исправление IDA

IDA - это еще один инструмент, который можно использовать для исправления исполняемых файлов. В качестве примера рассмотрим следующий код:

```
printf("please enter the password\n");
scanf("%s", user_entered_password);
if (strcmp(user_entered_password, correct_password) == 0)
{
        printf("SUCCESS\n");
}
```

```
else
{
        printf("Failure\n");
}
```

Этот код реализует простую систему аутентификации. Он просит пользователя ввести пароль и проверяет ответ. Если ответ правильный, выводится сообщение об успешном завершении; в противном случае выводится сообщение об ошибке. Хотя это упрощенный пример, имейте в виду, что процесс проверки пароля и перехода в одну сторону, если он неверный, и в другую, если он правильный, очень распространен. В IDA вы можете исправить приложение, чтобы отменить эту проверку пароля.

По умолчанию IDA не отображает машинный код в виде графика. Если вы не вносите исправления, это не имеет особого смысла. Но когда вы начнете желать исправления, вы захотите его увидеть. Чтобы отобразить машинный код, выберите Опции ➪ Общие, чтобы открыть окно, показанное на рисунке 11.30. Затем укажите количество байт кода операции, которое будет отображаться в виде графика (большинство кодов операций не превышают 8 байт, поэтому рекомендуется установить его равным 8).

На рисунке 11.31 показана логика проверки паролей приложения в IDA. Как показано, используется левый (красный) путь, если пароли совпадают, в то время как в противном случае используется правый (зеленый) путь.

Инструкция, которая решает, какой прыжок выполнить, называется jnz. Напомним, что jnz расшифровывается как “прыжок не нулевой”.

Эту проверку пароля можно обойти несколькими различными способами. Один из вариантов - попытаться выяснить, что должно быть “не нулевым”. Это означает выяснение того, какие два значения он сравнивает, чтобы вы потенциально могли создать действительный ключ или взломщик.

Более простой альтернативой является использование ваших знаний о x86 для исправления приложения. Как есть, приложение оценивает условие и выполняет jnz (0x75), если пароль неверен. Но что, если вы сделали прямо противоположное? Изменение этого jnz на jz (0x74) изменит логику, в результате чего приложение будет принимать только неправильные пароли. При перевернутой логике неправильный пароль приведет к успеху, а правильный - к сбою.

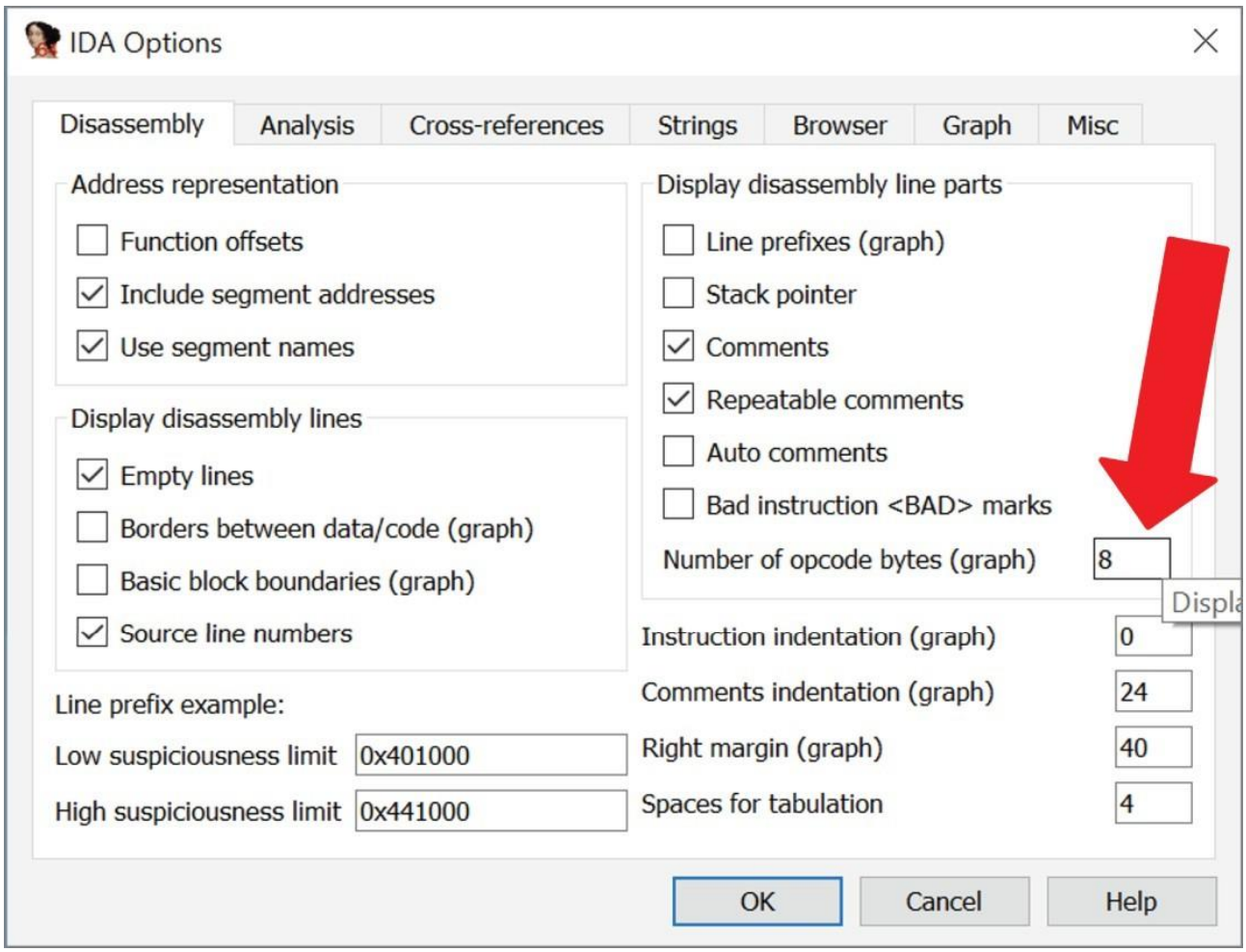


Рисунок 11.30: Отображение байтов кода операции в IDA

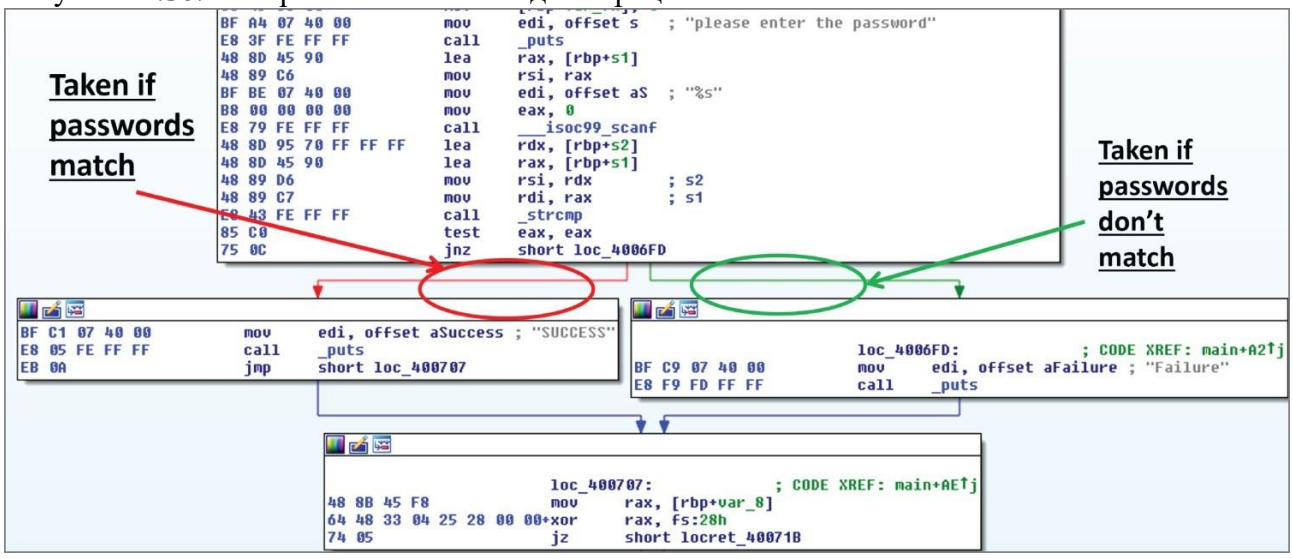


Рисунок 11.31: Код для проверки пароля в IDA

Чтобы изменить инструкцию, выделите ее и нажмите Редактировать ➪ Программа исправления ➪ Изменить байт. Затем в окне "Байты исправления", показанном на рисунке 11.32, измените первое значение с 74 на 75.

На рисунке 11.33 показано, как будет выглядеть приложение после применения исправления. Единственный бит, который был изменен, будет выделен в IDA, а значение двух путей после перехода изменится на противоположное. Теперь приложение будет работать для чего угодно, кроме правильного пароля.

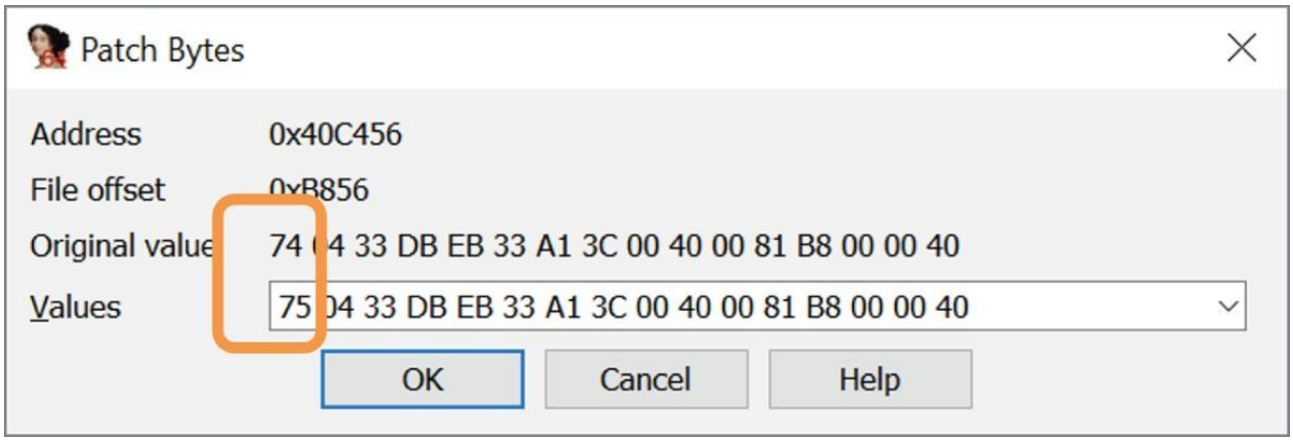


Рисунок 11.32: Окно байтов исправления IDA

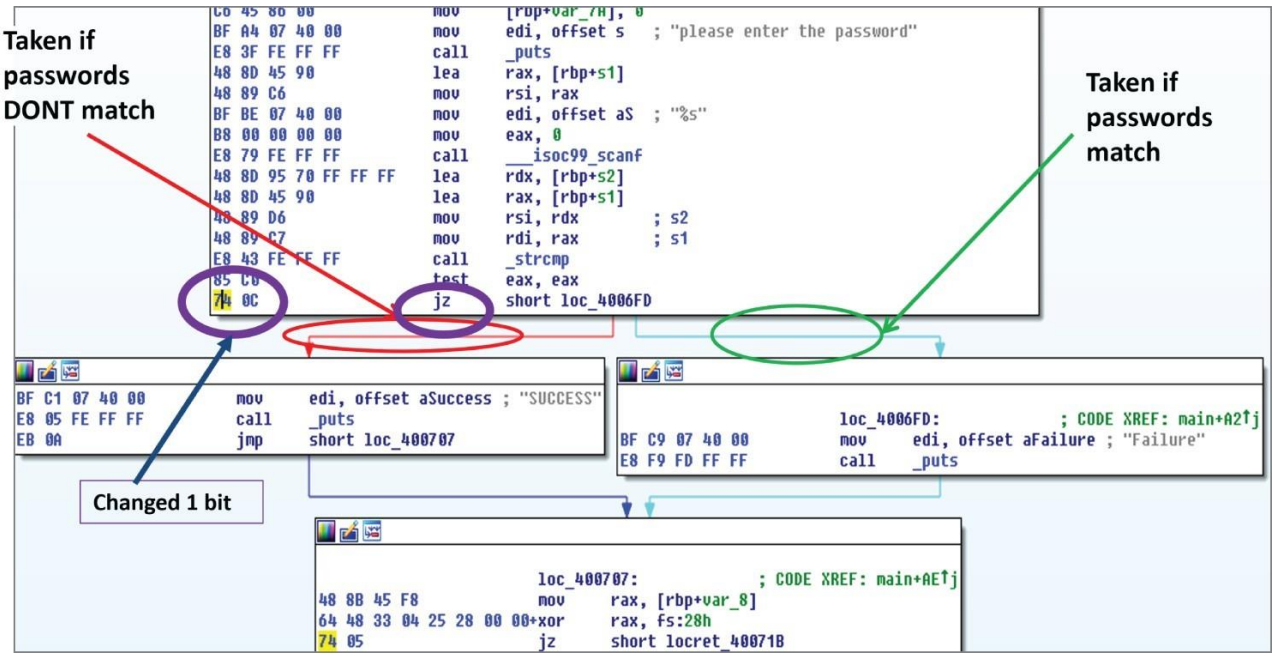


Рисунок 11.33: Логика проверки пароля в IDA после исправления

Лабораторная работа: Логические потоки IDA
В этой лабораторной работе дается введение в использование IDA для реверсирования. Файлы лабораторной работы находятся на виртуальной машине Windows в папке ida_logic на рабочем столе. Внутри этой папки будет несколько двоичных файлов. Узнайте, какой из них является:

- if
- Составной if (т.е. if(cont1 && cond2))
- цикл while

- цикл for
- do while loop

Навыки
В этой лабораторной работе предлагается практика использования IDA для обратного проектирования графиков потоков управления. Цель состоит в том, чтобы научиться быстро идентифицировать высокоуровневые конструкции кодирования на основе их шаблонов потока управления.

Выводы
Анализ потока управления программой может облегчить быстрое понимание того, что происходит внутри кода. Умение быстро распознавать эти потоки может значительно улучшить ваши способности к обратному проектированию.

Ghidra
Ghidra - это инструмент статического анализа, выпущенный в 2019 году АНБ. Он имеет много общего с IDA, но, в отличие от IDA, является бесплатным и с открытым исходным кодом. Во многих ситуациях Ghidra является адекватной заменой IDA.

IDA имеет гораздо более давнюю репутацию в этой области, но Ghidra также чрезвычайно мощна и во многих случаях обладает множеством тех же функций. Этот пример демонстрирует IDA с учетом ее долгой истории в области обратного проектирования, но все показанное также может быть выполнено в Ghidra. Инструменты достаточно схожи, чтобы навыки, приобретенные в одном из них, часто передавались другим. Попробуйте Ghidra для некоторых из последующих открытых лабораторных работ в этой книге и на собственной практике.

Лабораторная: Взлом с помощью IDA
В этой лабораторной работе рассматривается более сложное приложение в IDA. Лабораторные работы и все связанные с ними инструкции можно найти в соответствующей папке здесь:

https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES

Для этой лабораторной работы, пожалуйста, найдите Lab Cracking с помощью IDA и следуйте предоставленным инструкциям.

Навыки
В этой лабораторной практикуется использование IDA для взлома больших реальных приложений. Цель состоит в том, чтобы научиться быстро определять точки интереса и расставлять приоритеты в нескольких подходах к взлому.

Выводы
Программы реального мира слишком велики для единообразного, мелкозернистого анализа. Сортировка имеет решающее значение для поиска интересующих точек.

Взломщику обычно доступно множество возможностей. Выбор того, что использовать, может сэкономить (или стоить) значительное время.

Резюме
В этой главе рассмотрены некоторые из наиболее широко используемых инструментов для

реверсирования и взлома. Найдите время, чтобы ознакомиться с ними. В долгосрочной перспективе это окупится!

Глава 12
Защита
Как вы защищаетесь от взлома? Для начала важно иметь хороший дизайн проверки ключей (не используйте Starcraft / Half-Life). Оттуда вы можете реализовать дополнительные защитные опции.

Однако важно помнить, что не существует такого понятия, как программное обеспечение, которое невозможно взломать. Ваша задача как защитника - замедлить работу злоумышленников в критических частях вашего программного обеспечения и расстроить их настолько, что они перейдут к другой цели.

Как и во многих вещах в сфере кибербезопасности, вы просто не хотите быть низко висящим фруктом. “Плавая в воде, кишащей акулами, вам не обязательно быть самым быстрым... просто быстрее, чем парень рядом с вами”.

Запутывание
Запутывание - это практика сокрытия предполагаемого значения кода путем целенаправленного придания логике двусмысленности и неясности. Для замедления обратного проектирования может быть полезно выполнить следующее:

- Медленный взлом
- Медленное вмешательство
- Защита интеллектуальной собственности

При правильном выполнении обфускация может сделать код практически нечитаемым. Например, следующий код на C (доступен по ссылке [www.ioccc.org/1988/phillipps.c)](www.ioccc.org/1988/phillipps.c) при компиляции и запуске выводит текст всей песни "12 дней Рождества". Это был один из победителей IOCCC, который является соревнованием по ручному запутыванию кода. При взгляде на это у меня болит голова, и я не могу предположить, сколько времени мне пришлось бы перепроектировать код, прежде чем я понял, что он делает.

```
#include <stdio.h>
main(t,_,a)
char
*
a;
{
       return!
 0<t?
t<3?
 main(-79,-13,a+
main(-87,1-_,
main(-86, 0, a+1 )
 +a)):
 1,
t<_?
main(t+1, _, a )
:3,
 main ( -94, -27+t, a )
&&t == 2 ?_
<13 ?
 main ( 2, _+1, "%s %d %d\n" )
 :9:16:
t<0?
```

```
t<-72?
main( _, t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,
       /n{n+,/+#n+,/#;\
#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!
      /+k#;\
q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw'
      i;# ){nl]!/n{n#'; \
r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c
;;\
{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:
t<-50?
_==*a ?
putchar(31[a]):
 main(-65,_,a+1)
:
main((*a == '/') + t, _, a + 1 )
:
 0<t?
 main ( 2, 2, "%s")
:*a=='/'||
 main(0,
 main(-61,*a, "!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,
       fxntdCeghiry")
,a+1);}
```

Концепция запутывания также проникла в массовую культуру. Следующие цитаты взяты из сцены в одном из фильмов о Джеймсе Бонде "Скайфолл", когда Q пытается проникнуть в ноутбук Сильвы.

“Существуют алгоритмы, шифрования и асимметрия!”
“Похоже на запутанный код, скрывающий свое истинное назначение. Безопасность за счет скрытности!”
Обфускации могут применяться вручную или автоматически к программе на различных этапах ее жизненного цикла, включая следующие:

- Исходный код
- Байт-код
- Объектный код
- Двоичный исполняемый код
- Вычисление запутывания

При оценке вариантов обфускации следует учитывать несколько различных факторов:

- Эффективность: Насколько сильно обфускация применяется к программе
- Устойчивость: Насколько хорошо обфускированный код выдерживает атаки с помощью инструментов обратного проектирования
- Скрытность: Насколько хорошо запутанный код сочетается с остальной частью программы
- Стоимость: Снижение производительности запутанного приложения

В целом, эти факторы, как правило, работают друг против друга. Например, чем сильнее

запутывание, тем менее скрытным оно обычно является.

На практике стоимость производительности часто является ограничивающим фактором. Однако почти все обфускации допускают некоторую степень масштабирования/настройки в зависимости от требований.

Автоматическая обфускация
Обфускацию можно выполнить вручную. Однако почти всегда лучше использовать инструменты для запутывания кода. Некоторые из распространенных методов запутывания включают следующее:

- Искажение имен
- Шифрование строк
- Запутывание потока управления
  - Сглаживание потока управления
  - Непрозрачные предикаты
- Замена инструкций

Искажение имен
Искажение имен включает в себя запутывание имен функций и переменных. Это можно сделать несколькими различными способами, включая следующий:

- Заменить на тарабарщину (get_key -> aVJ230AM)
- Заменить вводящим в заблуждение именем (get_key -> draw_screen)
- Заменить не описательным именем (get_key -> a)

После изменения назначение функций и переменных больше не сразу становится очевидным. Для примера рассмотрим следующий пример кода:

```
public static void SelectionSort <T> (T[] data, int size)
	where T: IComparable
{
	for (int num1 = size – 1; num1>= 1; num1--)
	{
		T local1 = data[0];
		int num2 = 0;
		for (int num3 = 1; num3 <= num1; num3++)
		{
			if (data[num3].CompareTo(local1)> 0)
			{
				local1 = data[num3];
				num2 = num3;
			}
		}
		T local2 = data[num2];
		data[num2] = data[num1];
		data[num1] = local2;
	}
}
```

После искажения это может выглядеть примерно так:

```
public static void a <a> (a[] A_0, int A_1) where a:IComparable
{
	int num1 = A_1 – 1;
```

```
Label_004D:
        if (num1 < 1)
        {
              return;
        }
        a local1 A_0[0];
        int num2 = 0;
        int num3 = 1;
        while(true)
        {
              if (num3 <= num1)
              {
                    if (A_0[num3].CompareTo(local1)> 0)
                    {
                          local1 = A_0[num3];
                          num2 = num3;
                    }
              }
              else
              {
                    a local2 = A_0[num2];
                    A_0[num2] = A_0[num1];
                    A_0[num1] = local2;
                    num1--;
                    goto Label_004D;
              }
              num3++;
        }
}
```

В оригинале относительно легко определить, что код является алгоритмом сортировки, даже без имени функции. Однако сделать это после искажения намного сложнее.

Надежное шифрование

Другой метод обфускации заключается в том, что обфускатор шифрует строки при сборке исполняемого файла. Функция decrypt в коде затем расшифровывает отдельные строки по мере необходимости во время выполнения. Это делает такие инструменты, как представление строк IDA, непригодными для использования.

Надежное шифрование может оказать существенное влияние на читаемость кода. Рассмотрим следующий код:

```
public a() {

        this.a = "Hi, my name is Paul."
}
public static void a() {
        a a1 = new a();
        Console.WriteLine("Enter password: ");
        string text1 = Console.ReadLine();
        if (!text1.Equals(a1.a))
```

```
        {
              Console.WriteLine("Incorrect password.");
        }
        else
        {
              Console.WriteLine("Correct password.");
        }
        Console.ReadLine();
}
```

После надежного шифрования этот код может выглядеть следующим образом:

```
pubic a() {
        int num1 = 5;
        this.a =
a("\ue6ad\u9eb1\u94b3\uc1b7\u9ab9\ud2bb\uadbf\ua7c1\ue4c
      3\uafc5\ubbc7\ueac9\u9ccb\uafcd\ua5cf\ubed1\ufad3", num1;
}

public static void a()
{
        int num1 = 13;
        a a1 = new a();
        Console.WriteLine(a("\uf3b5\ud6b7\uceb9\uccbd\ue0bf\ub2c1\
ua5c3\u
              b5c5\ubbc7\ubdc9\ua3cb\ubccd\ub4cf\ue8d1\uf4d3,
num1));
        string text1 = Console.ReadLine();
        if (1text1.Equals(a1.a)) {
                 Console.WriteLine(a(\uffb5\ud8b7\ud3bb\uccbd\
ub2bf\ua7c1
                     \ua7c3\ub2c5\ue8c7\ubac9\uadcb\ubdcd\ua3cf\
ua5d1\ubb
                     d3\ua4d5\ubcd7\uf4d9", num1));
        }
        else
        {
                Console.WriteLine(a("\uf5b5\ud7b7\uc8b9\ucebb\
ua3bf\ub6c1
                     \ue4c3\ub6c5\ua9c7\ub9c9\ubfcb\ub9cd\ubfcf\
ua0d1\ub0
                     d3\uf8d5", num1));
        }
        Console.ReadLine();
}
```

В оригинале строки позволяют легко определить, что это код аутентификации (что часто очень интересно злоумышленникам). Без этих строк логику кода гораздо сложнее понять. Имейте в виду, что одна из самых больших трудностей при взломе приложения - это поиск соответствующего кода. В двоичном коде с сотнями тысяч строк кода только пять могут быть связаны с проверкой ключей, и использование таких инструментов, как strings, является мощным способом быстро разобраться с этими пятью строками. Удаление строк довольно болезненно для реинжиниринга.

Выравнивание потока управления
С помощью этого метода запутывания поток управления каждой функцией “сглаживается”. Это включает в себя следующие шаги:

- Функция сворачивается в инструкцию switch в бесконечном цикле.
- Каждому базовому блоку исходного потока присваивается номер состояния.
- Оператор switch выполняет выбор между базовыми блоками, распределяя их в правильном порядке.

На рисунке 12.1 показано, как процесс выравнивания преобразует приложение в IDA. Хотя логика та же, поток управления гораздо сложнее проанализировать.

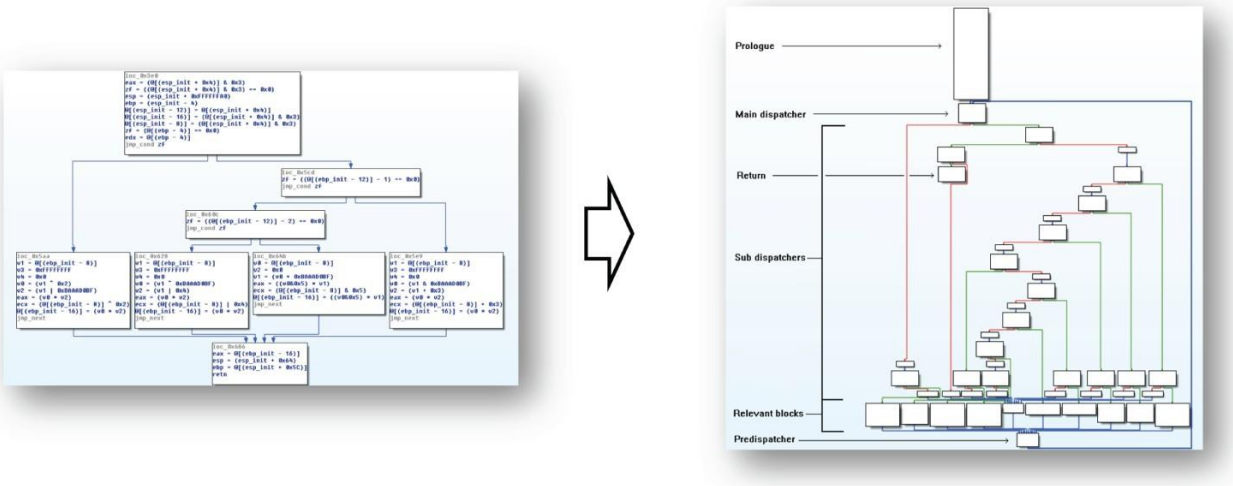


Рисунок 12.1: Сглаживание потока управления в IDA

Непрозрачные предикаты
Непрозрачные предикаты добавляют ненужный код, чередующийся с реальным кодом. Ненужный код никогда не выполняется, в то время как реальный код выполняется всегда. Однако для реверс-инженера это хороший способ отвлечь его бесполезным кодом, заставляя тратить часы на реверс-инжиниринг ненужного кода, который по сути не имеет значения. На рисунке 12.2 показан пример этого в IDA.

Путь определяется оператором if, который всегда принимает одно и то же значение. Однако для идентификации может потребоваться время (“непрозрачный предикат”), что замедляет анализ.

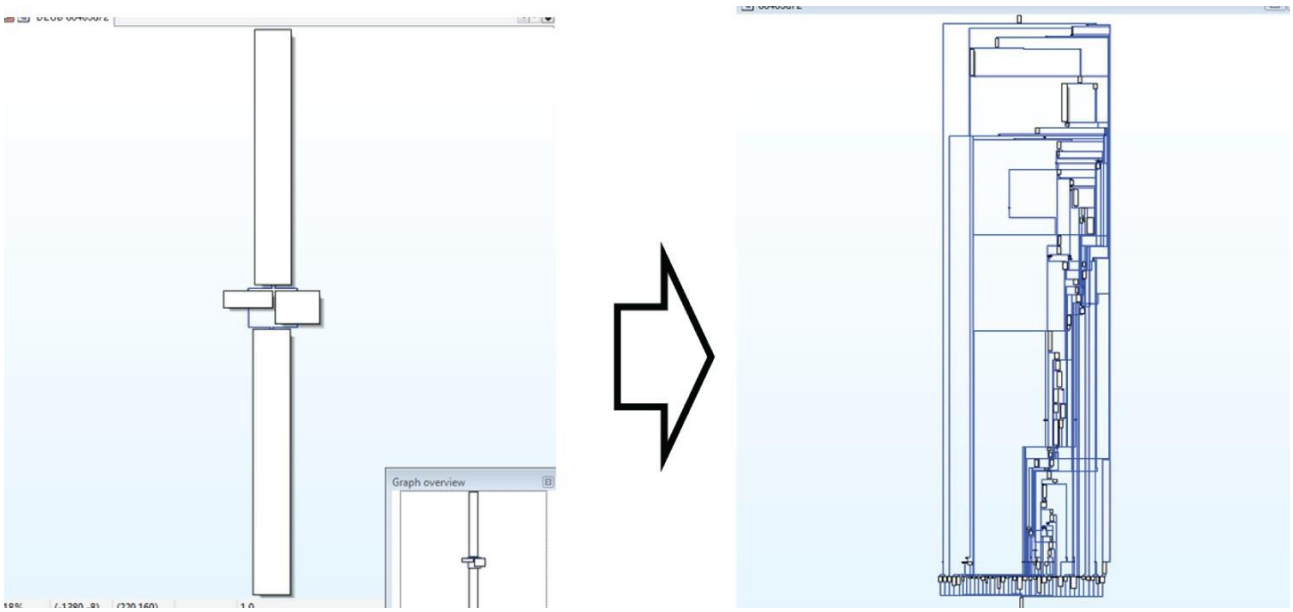


Рисунок 12.2: Непрозрачные предикаты в IDA

Рассмотрим следующее утверждение

```
if ( (a<<1)%2 ) { b = a * b + a; } else { a = a + b; }
```

Где здесь мусорный код?

Замена инструкций
Замена инструкций предполагает замену легко идентифицируемых инструкций сложными, которые выполняют одно и то же действие. Для примера рассмотрим следующий код:

```
sub edx, 0x192A6C72
neg ecx
sub edx, ecx
add edx, 0x192A6C72
```

Какова была первоначальная операция?

Обфускаторы
Обфускаторы обычно предоставляют “регуляторы”, которые позволяют разработчику настраивать уровень обфускации. Причина этого в том, что большее количество обфускации не всегда лучше. В целом, увеличение обфускации снижает скорость выполнения и увеличивает размер файла. Кроме того, резкое увеличение обфускации существенно не увеличивает сложность обратного проектирования. Баланс между удобством использования и безопасностью требует нахождения золотой середины.

Если вам удастся это сделать, обфускация может стать ценным инструментом, особенно для кода, который в остальном тривиален для декомпиляции (например, языки JIT, рассмотренные ранее, например .NET и т.д.). Однако также важно убедиться, что используемый вами инструмент также не обеспечивает простой доступный де-обфускатор.

Для обфускации общего назначения OLLVM может быть хорошей отправной точкой. Этот

инструмент имеет несколько преимуществ, включая тот факт, что он работает с промежуточным представлением LLVM (IR) и поддерживает все интерфейсы LLVM (gcc, clang) и многие исходные языки (C, C++, C#, Lisp, Fortran, Haskell, Python, Ruby и т.д.).

Использование OLLVM не рекомендуется для производственного кода. Тем не менее, это может быть хорошей основой для пользовательских обфускаторов или просто обучения / игры с обфускацией.

В дополнение к OLLVM существует множество инструментов и приемов для обфускации, специфичных для конкретного языка. Некоторые примеры включают Dotfuscator для C# и Proguard для Java.

Для программ на JavaScript для запутывания можно использовать такие инструменты, как YUICompressor и UglifyJS. В общем, минимизаторы, просто как побочный продукт, вводят некоторый разумный уровень запутывания.

Код Python может быть скомпилирован в байт-код, чтобы удалить некоторые имена переменных и комментарии. Затем байт-код может быть запутан и выпущен с помощью пользовательского интерпретатора. Некоторые обфускаторы Python включают Tigress, BitBoost и Opy, но они менее популярны, чем упомянутые ранее.

Устранение обфускаторов
Обфускаторы предназначены для защиты от обратного проектирования, делая его выполнение более сложным и отнимающим много времени. Однако запутывание не идеально, и, как неоднократно говорилось ранее, мотивированные взломщики могут в конечном итоге победить его.

Некоторые из способов, с помощью которых реверс-инженер может ускорить процесс анализа запутанного двоичного файла, включают следующее:

- Запуск трассировок для идентификации реального кода от поддельного
- Использование символьного анализа для упрощения сложности
- Написание пользовательских сценариев для устранения запутываний

Лабораторная работа: Запутывание
В этой лабораторной работе рассматриваются методы запутывания. T Labs и все связанные с ними инструкции можно найти в соответствующей папке здесь:

https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES

Для этой лабораторной работы, пожалуйста, найдите Lab Obfuscation и следуйте предоставленным инструкциям.

Навыки
Эта лабораторная работа дает опыт обхода методов запутывания с использованием objdump. Цель состоит в том, чтобы понять влияние распространенных методов защиты кода.

Выводы
Методы обфускации замедлят, но не уничтожат взлом. Однако помните, что иногда достаточно замедления. У продвинутых реверс-инженеров часто есть инструменты для автоматического обхода распространенных обфускаций.

Защита от отладки

Отладка часто является самым быстрым способом обратного проектирования исполняемого файла. Анти-отладка - это серия методов, направленных на то, чтобы попытаться лишить кого-либо возможности динамически анализировать ваше приложение с помощью отладчика. В этой области существует множество методов, но большинство из них направлены на попытку проверить наличие отладчика. Несколько распространенных проверок защиты от отладки включают следующее:

- Проверки памяти
- Проверки процессора
- Проверки синхронизации
- Проверки исключений
- Проверки среды

Как и в случае с большинством средств контроля безопасности, существуют компромиссы между удобством использования и защитой от отладки, размер кода и производительность являются двумя наиболее болезненными побочными эффектами. Из-за этого функциональность защиты от отладки часто добавляется только выборочно, оставляя за собой право ее использования для кода, наиболее подверженного атаке (средства проверки ключей, конфиденциальные IP-адреса и т.д.). Но, как и во всякой безопасности, есть свои плюсы и минусы; если вы создаете кучу анти-отладочных проверок вокруг своего конфиденциального кода, вы также попадаете в яблочко, сообщая злоумышленнику, где именно находится интересный материал. Таким образом, хотя они, возможно, и не смогут его отладить, теперь они точно знают, на чем следует сосредоточиться с помощью методов статического анализа. Но это не значит, что этого не стоит делать; статический анализ может занять в 100 раз больше времени, чем динамический, поэтому, даже если вы рисуете стрелки в своем конфиденциальном коде, принуждение их выполнять это статически все равно может быть мощным инструментом.

Основная цель анти-отладки - определить, когда подключен отладчик, и предпринять действие. Наиболее часто используемые действия включают следующее:

- Принудительное отключение отладчика
- Выход из программы
- Выполнение отвлекающего кода для потери времени злоумышленника

IsDebuggerPresent()

IsDebuggerPresent - это проверка памяти для отладчика. Функция IsDebuggerPresent, которая находится в Windows.h, возвращает значение true, если программа запускается под управлением отладчика. Следующий код показывает пример того, как он используется для выхода из приложения, если подключен отладчик:

```
if (IsDebuggerPresent())
      exit(1);
```

Проверка с использованием IsDebuggerPresent может быть отменена путем установки точки останова в инструкции сразу после возврата функции. Когда срабатывает точка останова, установите значение eax равным 0, что сообщает программе, что отладчик не подключен. Помните, что eax содержит возвращаемое значение. Возвращать 1 нежелательно, потому что это означает, что он обнаружил отладчик, поэтому вместо этого заставьте его возвращать 0. Хотя это кажется тривиальным, имейте в виду, что игра заключается в том, чтобы просто усложнить задачу. Если в вашем коде есть 100 таких проверок, злоумышленники, желающие

выполнить отладку, должны отслеживать каждую из них и либо вручную устанавливать точку останова и каждый раз изменять возвращаемое значение, либо запускать пользовательские скрипты, которые будут выполнять это изменение за них. Это раздражает злоумышленника? Ага.

Регистры отладки

Приложение также может использовать регистры отладки центрального процессора для выполнения проверки наличия отладчика. Напомним, что в разделе "отладка" обсуждались программные и аппаратные точки останова. Аппаратная точка останова использует аппаратные регистры центрального процессора для установки самой себя.

Эти аппаратные точки останова используют регистры отладки (в x86: DR0, 1, 2, 3, 6, 7) вместо модификаций памяти. Можно обнаружить отладку, проверив эти регистры.

Для примера рассмотрим следующий пример кода. Он проверяет, установлен ли какой-либо из регистров отладки, указывая на аппаратную точку останова.

```
if (GetThreadContext(hThread, &ctx))
       if ((ctx.Dr0 != 0x00) || … || (ctx.Dr7 != 0x00))
              exit(1);
```

Вызов функции GetThreadContext() имеет решающее значение для этого метода защиты от отладки. Для тех, кто хочет обойти этот метод, поместите точку останова после этого вызова и измените структуру контекста, установив наблюдаемые значения всех регистров отладки равными 0x0. Опять же, возможно ли это обойти? Да. Раздражает ли злоумышленника необходимость продолжать вносить эти изменения? Ага. Раздраженный атакующий равен успеху защитника! Также напомним, что мы обсуждали, что IDA 6.3 и выше поддерживают аппаратные точки останова. Эти точки останова не используют регистры отладки и вместо этого используют разрешения страницы. Другими словами, этот тип анти-отладочной проверки не выявит аппаратную точку останова.

RDTSC

RDTSC расшифровывается как счетчик временных меток чтения команд x86. Этот счетчик можно использовать для считывания временных меток с центрального процессора. У этого есть много интересных применений, но одно из них заключается в выполнении временной проверки для отладчика.

При запуске приложения (без отладчика) процессор работает очень быстро, но когда подключен отладчик, это не так. Даже если вы не выполняете пошаговые действия и просто позволяете коду выполняться, это на порядки медленнее, чем просто отключение процессора. И это происходит еще медленнее, если вы выполняете что-то вроде пошагового выполнения кода. С помощью RDTSC приложение может получать временные метки до и после блока кода и измерять, сколько времени потребовалось для выполнения кода. Если дельта велика, вполне вероятно, что код достиг точки останова или был обработан вручную с помощью отладчика.

Следующий псевдокод показывает, как RDTSC можно использовать для обнаружения отладчика:

```
a = __rdtsc();
keycheck();
b = __rdtsc();
```

```
if (b - a> 0x10000)
      exit(1);
```

Чтобы обойти этот тип анти-отладочной проверки, вы могли бы прервать второй вызов RDTSC. Затем вы могли бы изменить значение либо a, чтобы оно было ближе к b, либо b, чтобы оно было ближе к a. По сути, сделайте разницу между ними очень маленькой, чтобы предполагать, что выполнение прошло по плану. Обходимо? Да. Раздражает необходимость вносить исправления каждый раз при отладке? Да!

Недопустимый CloseHandle()

Использование недопустимого вызова CloseHandle является примером проверки исключения для отладчика. Функция CloseHandle Windows генерирует исключение, если вызывается с недопустимым дескриптором во время работы в отладчике (и никак иначе). Приложение может использовать эти знания для вызова CloseHandle для недопустимого дескриптора, чтобы обнаружить присутствие отладчика.

Следующий код демонстрирует, как CloseHandle можно использовать для обнаружения отладчика:

```
HANDLE hInvalid = (HANDLE)0xDEADBEEF;
__try { CloseHandle(hInvalid); }
__except (EXCEPTION_EXECUTE_HANDLER) { exit(1); }
```

Чтобы отменить эту проверку, установите точку останова в CloseHandle. Когда точка останова сработает, измените аргумент на INVALID_HANDLE_VALUE.

Сканирование каталогов

Сканирование каталогов - это проверка среды на наличие отладчика. Оно включает в себя сканирование файловой системы на предмет установки распространенных отладчиков и средств взлома. Если эти средства найдены, приложение может выбрать выход.

Однако это беспорядочный поиск, и эти инструменты могут не выполнять активную отладку приложения. В результате это вредит законным пользователям этих инструментов.

Чтобы отменить эту проверку, установите точку останова при обходе каталога. Затем замаскируйте каталоги инструментов, чтобы приложение не видело их и не выполняло в них поиск.

Активная защита от отладки

Методы защиты от отладки не обязательно должны заключаться в пассивном обнаружении отладчиков. Существует множество подходов “активной защиты”, включая следующий:

- NtUserBlockInput: Блокировать ввод с клавиатуры подключенного отладчика.
- NtUserFindWindowEx: Получить доступ к окну отладчика.
- Атаки, специфичные для отладчика: Например, версии IDA старше 7.0 приводят к сбою примерно при выполнении 10 000 инструкций без перехода.

Существует гораздо больше вариантов. Для защиты от агрессивной отладки сначала вам нужно распознать наличие отладчика, а затем предпринять какие-либо оскорбительные действия. В помощь доступны плагины с открытым исходным кодом, в том числе некоторые, используемые в следующей лабораторной работе.

Для защитной защиты от отладки важно помнить, что вам не нужно изобретать велосипед. Доступны готовые решения, в том числе бесплатные проверки Windows с открытым

исходным кодом с помощью отладчика.

Устранение защиты от отладки
Как и другие средства защиты программного обеспечения, код защиты от отладки может быть побежден (хотя, если все сделано правильно, это болезненно). Первый шаг - найти и перепроектировать проверку защиты от отладки. Часто это достигается путем обратной работы с того места, где вы были пойманы с помощью отладчика.

Как только вы определили код защиты от отладки, у вас есть несколько различных вариантов его устранения, включая следующие:

- Удаление проверки с помощью nops
- Установка точки останова для проверки и изменение памяти/ регистров для маскировки отладчика
- Использование встроенных плагинов или скриптов отладчика

В общем, скрытнее маскировать отладчик сразу же при анти-отладочной проверке. Например, если приложение использует IsDebuggerPresent, измените возвращаемое значение IsDebuggerPresent вместо того, чтобы возиться с инструкцией if или кодом завершения, предназначенным для использования этого значения.

Лабораторная работа: Защита от отладки
В этой лабораторной работе представлены практические приемы устранения методов защиты от отладки. Лабораторные работы и все связанные с ними инструкции можно найти в соответствующей папке здесь:

https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES

Для этой лабораторной работы, пожалуйста, найдите Lab Anti-Debugging и следуйте предоставленным инструкциям.

Навыки
В этой лабораторной работе используется x64dbg для обхода методов защиты от отладки. Цель состоит в том, чтобы понять влияние распространенных методов защитного кодирования.

Выводы
Опять же, часто бывает достаточно замедлить реверсивный инжиниринг; защита не обязательно должна быть идеальной. Однако у опытных реверсоров будут инструменты для преодоления распространенных защитных приемов.

Резюме
Разработчики хотят защитить себя и свой код от реверсоров и взломщиков. В этой главе рассмотрены некоторые распространенные методы достижения этой цели, включая запутывание и защиту от отладки.

Глава 13
Расширенные методы защиты
В предыдущей главе были представлены некоторые базовые методы защиты приложения от обратного проектирования и взлома. В этой главе демонстрируются некоторые более продвинутые методы, которые сложнее обойти, включая защиту от несанкционированного доступа, упаковку, виртуализацию и использование шифровальщиков.

Защита от несанкционированного доступа
Одним из мощных методов взлома, которые мы рассмотрели, является исправление, как для долговременного взлома, так и для помощи в обратном проектировании. Защита от несанкционированного доступа - это серия методов, направленных на то, чтобы сделать программное обеспечение более трудным для модификации злоумышленником. Некоторые распространенные подходы включают следующее:

- Хэширование
- Подпись
- Водяной знак

Защита программного обеспечения
У всех следующих техник есть способы потерпеть поражение, но (и я не могу не подчеркнуть это достаточно сильно) только потому, что у них есть способы потерпеть поражение, не означает, что их не стоит применять. Каждый из них обеспечивает глубокий уровень защиты, и даже если метод их устранения укладывается в несколько предложений, это не значит, что на практике это легко.

Хэширование
Приложение может использовать хэш-функции для реализации защиты от несанкционированного доступа с помощью следующих шагов:

- Вычислите хэш программного обеспечения.
- Встроите хэш в программное обеспечение.
- Попросите программное обеспечение проверить свой собственный хэш перед выполнением.
- Любые модификации программного обеспечения изменяют хэш.
- Защита полагается на тот факт, что изменения в приложении приведут к сбою проверки хэша. Чтобы обойти это, злоумышленнику необходимо будет внести свои изменения, а затем повторно вычислить хэш после внесения изменений и изменения проверяемого значения или полного удаления проверки хэша.

Подписи
Цифровые подписи могут обеспечить надежную защиту целостности данных и подлинности. Они используют криптографию с открытым ключом, при которой генерируется пара открытого и закрытого ключей. Чтобы использовать их для защиты от несанкционированного доступа, выполните следующие действия:

- Подпишите программное обеспечение закрытым ключом, создав подпись.
- Встроите подпись в программное обеспечение.
- Попросите программное обеспечение проверить свою подпись с помощью вашего открытого ключа перед выполнением.
- Любые изменения в программном обеспечении делают подпись недействительной.

Одним из ключевых преимуществ цифровых подписей является то, что фактически невозможно сгенерировать действительную подпись без знания закрытого ключа. Чтобы обойти этот тип защиты, злоумышленнику пришлось бы полностью удалить проверку

подписи или завладеть закрытым ключом, чтобы они могли восстановить действительную подпись.

Водяной знак
Для внедрения водяных знаков каждый покупатель вашего программного обеспечения получает уникальную версию исполняемого файла, в которую вносятся изменения в следующем порядке:

- Порядок команд
- Имена функций
- Порядок параметров
- Замена команд

И т.д.
Конкретные изменения “водяного знака” этого экземпляра, позволяющие отследить его до владельца, а также обнаружить модификации. Кроме того, любые модификации программного обеспечения портят водяной знак, делая их очевидными.

Чтобы злоумышленник смог обойти эту защиту, ему необходимо идентифицировать разделы с водяными знаками. Затем замените их альтернативным знаком, чтобы скрыть источник измененного программного обеспечения.

Стража
С помощью защиты код внутри программы проверяет уязвимые области на предмет модификации. Например, код может специально проверять критический прыжок, чтобы убедиться, что он по-прежнему перемещается в намеченное местоположение. Общие области для мониторинга с помощью охранников включают проверку ключей, инструкции по переходу, других охранников и т.д.

Любые изменения в этих разделах отслеживаются охранниками. Затем охранники изменят поведение программного обеспечения (выйдут, изменят пути, отменят изменения и т.д.).

Эта защита основана на том факте, что охранник присутствует и может модифицировать программное обеспечение по мере необходимости. Если злоумышленник захочет обойти этот метод, ему нужно будет удалить защитный код программного обеспечения.

Упаковка
Упаковка - это широкий термин, относящийся к методам, обычно используемым в исполняемых файлах для сжатия и запутывания их содержимого. Некоторые распространенные методы упаковки включают следующее:

- Сжатие/шифрование разделов данных
- Шифрование разделов кода
- Сжатие/шифрование разделов кода

Защита от обратного проектирования
Одним из основных преимуществ упаковки является то, что она усложняет обратное проектирование. Например, упаковщик может включать функции, которые устраняют многие распространенные угрозы обратного проектирования, включая следующие:

- Защита от отладки: Упаковщики могут скрывать использование IsDebuggerPresent, что затрудняет его обнаружение.
- Защита от виртуализации: Упаковщики могут обнаруживать, когда приложение виртуализируется на платформе, такой как VMware, и скрывать код обнаружения.

- Защита от несанкционированного доступа: Упаковщики могут стирать заголовки в памяти, затрудняя сброс памяти.
- Защита от несанкционированного доступа: Защита от несанкционированного доступа может быть реализована с помощью контрольных сумм. Это включает в себя как распространенные (скользящая контрольная сумма, CRC32, MD5 и SHA-1), так и другие (Tiger, Whirlpool, MD4, Adler).

Упаковщики могут использовать шифрование, чтобы скрыть свой код. Часто для этого используются простые алгоритмы, такие как побитовые операторы (XOR/ROL/...), LCG, RC4 и Tea. Однако также могут быть использованы более продвинутые алгоритмы шифрования (DES, AES, Blowfish, Trivium, IDEA, ElGamal и т.д.). Если приложение было упаковано таким образом, что его код и разделы данных зашифрованы, то если бы вы загрузили его в один из дизассемблеров или шестнадцатеричных редакторов, вы бы увидели только небольшой фрагмент кода и много бессмысленного мусора. Крошечный фрагмент кода, который доступен, - это распаковщик. Чтобы код запустился, ему нужно будет распаковать себя в памяти во время выполнения, но это означает, что статический анализ не сможет увидеть остальную часть кода.

Упаковщики также могут использовать мутаторы (запутывание), которые изменяют код, сохраняя тот же набор команд и архитектуру. Некоторые мутации, которые могут быть использованы, включают перепрофилирование и олигоморфизм или другие методы запутывания, обсуждаемые в главе 12 “Защита”.

Как работают упаковщики

Упаковщик (автономный инструмент) упаковывает исполняемый файл (сжимает, запутывает и т.д.). Затем упаковщик добавляет распаковщик в начало исполняемого файла. При запуске исполняемого файла распаковщик будет первым запущенным кодом, и он распакует исходный код и данные в память (и только в память).

На рисунке 13.1 показано, как будет выглядеть упакованный исполняемый файл в IDA. IDA

```
.text:0805C050
.text:0805C050                  public start
.text:0805C050 start            proc near
.text:0805C050                  jmp     start_0
.text:0805C050 start            endp
.text:0805C050
.text:0805C050 ; ---------------------------------------------------------------------------
.text:0805C055                  db 0B3h, 58h, 12h
.text:0805C058                  dd 7A2EF958h, 73DE6856h, 297708BCh, 41272C37h, 492E3AE1h
.text:0805C058                  dd 0BB91C39Fh, 17072049h, 0B6572788h, 26CD82A6h, 0B6B2FBB8h
.text:0805C058                  dd 0D7D87B3h, 0EEA8772Ch, 0A5E71B1Bh, 0E5E1B170h, 1AA084B1h
.text:0805C058                  dd 0F67F9497h, 27E22E54h, 0D2EF54E5h, 5FC8B0FCh, 107274EDh
.text:0805C058                  dd 33DF407Ch, 3321B16Eh, 9054A231h, 25097B7Ah, 0C959BE98h
.text:0805C058                  dd 0FA156FA4h, 3A6C5D35h, 482937C8h, 4BD4AEBh, 51CE4A8h
.text:0805C058                  dd 0AFC3DC5h, 28340E28h, 0F8732F8h, 0F67A534h, 8C997059h
.text:0805C058                  dd 7C092F51h, 0F9C12512h, 41D3B406h, 413DA6AEh, 48A47C67h
.text:0805C058                  dd 9DCCFC00h, 472B12C6h, 4C19D2A2h, 472E7E3Eh, 509B602h
```

Рисунок 13.1: Упакованный код в IDA

Является ли это надежной защитой?

В следующих разделах мы поговорим о некоторых методах защиты и зададим вопрос о том, являются ли они надежной защитой. Оценки предназначены для того, чтобы на очень высоком уровне определить, на какие области каждая защита оказывает наиболее сильное воздействие. В центре внимания нашей книги в основном наступление, но мы сочли важным кратко остановиться на некоторых способах защиты. В каждом разделе для оценки эффективности защиты от взлома мы будем использовать так называемую триаду ЦРУ (CIA расшифровывается как конфиденциальность, целостность и доступность). Для тех, кто не

знаком с этим, это распространенный способ думать о средствах контроля безопасности, поскольку не все средства контроля безопасности охватывают все три части триады, поэтому важно знать, что полезно в каждом компоненте. Целостность - это подлинность чего-либо. Соответствует ли это первоначальному замыслу или было изменено? Конфиденциальность - это способность чего-либо быть доступным только уполномоченным сторонам. Доступность - это уровень, до которого что-либо доступно для выполнения своей предполагаемой функции. Эти три вместе обычно известны как триада CIA. Сравнение упаковщиков с триадой CIA:

- Конфиденциальность: Да, за исключением части кода для распаковки, остальная часть находится в формате, недоступном для чтения.
- Целостность: Да, изменения в двоичном файле могут привести к повреждению упакованных разделов, что, вероятно, приведет к сбою приложения.
- Доступность: Упаковщики могут оказывать негативное влияние на производительность, что может повлиять на доступность. Однако при тщательной настройке этот эффект можно свести к минимуму.

Победа над упаковкой

Итак, как можно победить упаковщиков? Отладьте программу и проследите за расшифровкой программы в памяти. Как только программа будет распакована в памяти, вы сможете проанализировать ее, но любое выполненное исправление будет применимо только к распакованному двоичному файлу. Исправление не может быть сохранено обратно в упакованный двоичный файл.

Одна естественная мысль, которая приходит людям в голову, заключается в том, что после того, как файл распакован в памяти, не могу ли я просто выгрузить его из памяти в новый распакованный двоичный файл? Технически это возможно, но сложно сделать. Приложения содержат много кода для запуска, и его загрузка в нужное место в памяти, настройка стека и т.д. Естественным образом не достигается путем сброса памяти и простого вызова исполняемого файла.

Другой вариант - посмотреть, сможете ли вы распаковать программу. У некоторых распространенных упаковщиков есть инструменты для распаковки, которые можно использовать для отмены установленных мер защиты; некоторые примеры включают UPX, MEW и ASPack.

Однако автономного распаковщика может и не быть, а код распаковки существует только в упакованном исполняемом файле. Однако это не значит, что мы застряли! Существует ряд отличных плагинов и инструментов, созданных специально для этой цели, таких как OllyDumpEx и ImpRec, которые предназначены для восстановления таблицы импорта. Это сложный, но выполнимый процесс, но он не является предметом нашей книги. Однако, если это представляет интерес, в Интернете можно найти несколько отличных блогов, посвященных реконструкции импорта.

PEiD

Часто при обращении к файлу бывает трудно понять, какие манипуляции с ним производились. Если вы каким-то образом знаете, что он был упакован с помощью определенного инструмента, то легко начать с этого пути. Но к взлому обычно не прилагается удобная инструкция, в которой рассказывается, какие средства защиты установлены. PEiD - это инструмент для обнаружения наиболее распространенных упаковщиков, шифровальщиков и компиляторов для переносимых исполняемых файлов (например, приложений). Он может обнаруживать сигнатуры более чем 470 различных

инструментов обфускации. Еще один более современный инструмент в этой области - Detect it Easy.

Как мы уже упоминали, многие средства защиты, такие как упаковщики и шифровальщики, также имеют распаковщики и дешифраторы. Идентификация используемого может сократить время анализа на порядок, позволяя снять многие средства защиты приложения.

На рисунке 13.2 показан пример использования PEiD. Для начала выберите файл для проверки. Затем PEiD покажет подробности его упаковки, шифрования и компиляции.

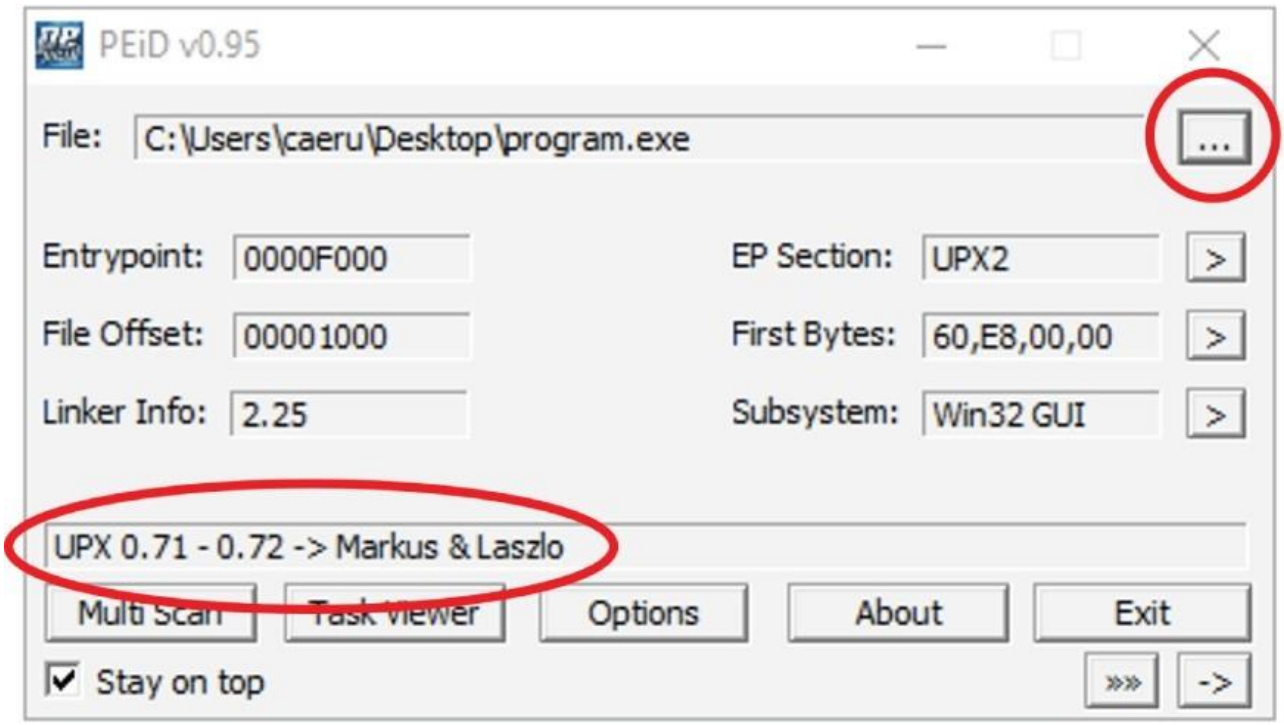


Рисунок 13.2: Идентификация упаковщиков с помощью PEiD

Лабораторная работа: Обнаружение и распаковка
В этой лабораторной работе рассматривается, как обнаружить и предотвратить использование обычного упаковщика. Лабораторные работы и все связанные с ними инструкции можно найти в соответствующей папке здесь:

https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES

Для этой лаборатории, пожалуйста, найдите лабораторию обнаружения и распаковки и следуйте предоставленным инструкциям.

Навыки
Упаковщики - распространенная защита от реверсирования. В этой лабораторной работе рассматривается использование IDA, Cheat Engine и Paid для проверки следующих навыков:

- Обнаружение присутствия упаковщиков
- Распаковка программ с помощью существующих инструментов

- Распаковка программ с расширенной отладкой
Рекомендации на вынос
Готовые распаковщики доступны для многих упаковщиков (не изобретайте велосипед). Когда распаковщики недоступны, распакованную исходную программу все равно можно восстановить вручную из памяти.

Виртуализация
Виртуализация обеспечивает некоторую форму запутывания и упаковки. Она переводит программу на пользовательский машинный язык и генерирует виртуальную среду/машину (ВМ) для ее интерпретации. Виртуальная машина встраивается в приложение и запускается при выполнении приложения. Обратите внимание, что в данном случае мы не говорим о типичных больших виртуальных машинах, таких как Windows или Linux, работающих в гипервизоре. Виртуализация в этом случае может просто означать добавление уровня абстракции/интерпретации между хостом (x86) и кодом.

Например, рассмотрим следующую простую программу “hello world”:

```
#include <stdio.h>
int main(void)
{
       printf(“hello, world!\n”);
       return 0;
}
```

Затем эту программу можно было бы скомпилировать на произвольный машинный язык. Например, вот как это выглядит в Brain$#@!:

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---
      .+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.#
```

Затем приложение упаковывается с помощью интерпретатора, написанного в целевой архитектуре (т.е. x86).

```
#include <stdio.h>

char data[30000];
char program[30000];
int ip=0; /* instruction pointer */
int dp=0; /* data pointer */

char read_byte(void) { return getchar(); }
void write_byte(char b) { putchar(b); }

int main(void) {
  int i=0; char b;

  do {
    b=read_byte();
    program[i]=b;
    i++;
  } while (b!='#');
  while (1) {
```

```
    b=program[ip];
    if (b==0) {
      break;
    } else if (b=='>') {
      dp++;
    } else if (b=='<') {
      dp--;
    } else if (b=='+') {
      data[dp]++;
    } else if (b=='-') {
      data[dp]--;
    } else if (b=='.') {
      write_byte(data[dp]);
    } else if (b==',') {
      data[dp]=read_byte();
    } else if (b=='[') {
      if (!data[dp]) {
        int c=1;
        do {
          ip++;
          if (program[ip]=='[`) { c++; }
          else if (program[ip]==']`) { c--; }
        } while (c);
      }
    } else if (b==']') {
      if (data[dp]) {
        int c=1;
        do {
          ip--;
          if (program[ip]=='[`) { c--; }
          else if (program[ip]==']`) { c++; }
        } while (c);
      }
    } else {
      /* do nothing */
    }
    ip++;
  }
  return 0;
}
```

Это добавляет уровень абстракции, через который должен пройти взломщик или реинжиниринг. Сначала выполните реинжиниринг промежуточного языка виртуальной машины. Для тех, кто знаком с языком программирования Java, Java запускается внутри виртуальной машины, называемой виртуальной машиной Java (JVM). Хотя это было сделано для обеспечения переносимости, а не безопасности, это добавляет уровень сложности. Существуют другие языки, которые выполняются внутри виртуальной машины, но вы также можете создать свой собственный (как в примере).

Как работает виртуализация кода

В отличие от упрощенного примера, хороший виртуализатор создаст уникальный произвольный машинный язык "на лету", в отличие от использования статического или известного языка. Это затрудняет разработку инструментов девиртуализации.

В этом случае программная логика преобразуется в пользовательский набор команд. В результате инструменты реверсирования не применяются немедленно, поскольку они не могут восстановить/проанализировать логику программы. Затем виртуальная машина компилируется в собственную архитектуру (т.е. x86).

Для реверсирования приложения требуются оба этих параметра:

- Реверсирование виртуальной машины для расшифровки пользовательского набора команд
- Изменение логики приложения в новом наборе команд

Этот процесс сложный и утомительный, поскольку ваш доступ к отладке ограничен. Вы не можете отлаживать логику целевой программы напрямую, только виртуальную машину. Некоторые инструменты, которые помогают в этом, включают Themida и VMProtect.

Многоуровневая виртуализация

Защита виртуализации может быть многоуровневой, как показано в следующем процессе:

- Виртуальная машина VM0 реализует пользовательский набор команд IS0.
- IS0 запускает виртуальную машину VM1, которая реализует пользовательский набор команд IS1.
- IS1 запускает исходное приложение.

Примером многоуровневой виртуализации может быть следующее:

- Скомпилируйте исходный код C на пользовательский язык, такой как DazzleZ.
- Напишите интерпретатор DazzleZ на пользовательском языке, таком как CatCat.
- Напишите интерпретатор CatCat на x86.
- Запустите программу на обычной платформе x86.

Реверсирование требует резервного копирования всех уровней виртуализации.

Проблемы с виртуализацией

Виртуализация может быть эффективным инструментом для замедления реверсирования и взлома. Однако у нее есть свои недостатки, в том числе следующие:

- Обнаружение AV: Часто вредоносные программы используют виртуализацию для маскировки, поэтому многие антивирусные программы автоматически помечают приложения, использующие ее.
- Раздувание файлов: Приложение, использующее виртуализацию, должно иметь встроенную виртуальную машину, что увеличивает размеры файлов.
- Замедленное выполнение: Виртуализированным приложениям необходимо запускать как виртуальную машину, так и виртуализированный код, что замедляет выполнение приложения.

Многоуровневое использование нескольких виртуальных машин увеличивает эти проблемы с размером и скоростью в геометрической прогрессии.

Является ли это надежной защитой?

Оценка виртуализации с учетом триады CIA дает следующие результаты:

Конфиденциальность: Да, исходный код абстрагируется с помощью уровней виртуализации.
Целостность: Да, модификации любого из уровней, скорее всего, вызовут волновой эффект сбоев, что затруднит исправление.
Доступность: Каждый уровень, добавленный в эту настройку, влияет на производительность. Слишком много уровней может существенно повлиять на скорость и доступность кода, выборки данных и т.д.

Отказ от виртуализации
Виртуализация может быть эффективной защитой, поскольку отказ от нее требует много времени и сложен. В целом, для отказа от виртуализации можно использовать следующий процесс:

- Обратная схема отправки кода: виртуальные машины обычно следуют знакомому циклу выборки-декодирования-выполнения центрального процессора, что позволяет понять, как отправляется код.
- Уменьшите сложность: Используйте сопоставление с образцом, символьный анализ и аналогичные методы для устранения ненужной сложности.
- “Девиртуализируйте” программу: попытайтесь восстановить представление исходного кода.
- Однако это не всегда простой “обратный процесс” для сложных виртуальных машин, и может оказаться невозможным восстановить исходный код, что вынудит вас провести реинжиниринг виртуализированного кода.
- Восстановите восстановленный код: Используйте традиционные инструменты для восстановления восстановленного кода. Возможно, вам придется полагаться на статический анализ, если работоспособная программа не может быть восстановлена.

Виртуализацию можно обойти путем обратного проектирования виртуальной машины и последующего преобразования приложения обратно в машинный код x86 для анализа. Некоторые инструменты, которые помогают в этом, включают Themida, VMProtect и Tigress.

Шифровальщики/дешифраторы
Шифровальщики шифруют разделы кода приложения (подмножество методов, рассмотренных в предыдущем разделе о пакерах), часто для того, чтобы избежать обнаружения вредоносных программ. Многие средства защиты от вредоносных программ анализируют часть программного обеспечения перед запуском и блокируют программное обеспечение на основе вызовов API к подозрительным функциям операционной системы. Зашифровывая раздел кода, вредоносная программа не позволяет антивирусным программам проверять содержимое приложения перед выполнением.

Как правило, зашифрованное программное обеспечение должно расшифровать само себя перед выполнением. Как правило, это означает, что ключ дешифрования находится где-то внутри программного обеспечения. Следовательно, обратное проектирование должно быть в состоянии найти ключ и расшифровать программное обеспечение.

Однако из этого есть некоторые исключения. Например, программное обеспечение, заблокированное узлом, может получить ключ из конкретной системы, в которой оно находится. В качестве альтернативы вредоносное ПО может подключаться к серверу для получения ключа дешифрования на лету.

Является ли это полезной защитой?
Преимущества, которые предоставляют криптографы, включают следующее:

- Конфиденциальность: Да, шифрование всегда добавляет уровень конфиденциальности. Расшифровка выполняется только при определенных обстоятельствах.
- Целостность: Некоторые, большинство алгоритмов шифрования добавляют здесь уровень защиты целостности, потому что изменение зашифрованных данных приводит к повреждению, а перевод - к модификации конечного кода.
- Доступность: Отсутствует; это не имеет никакого эффекта.

Устранение шифровальщиков
Большинство шифраторов имеют поддерживающие дешифраторы, которые являются

инструментами, способными автоматически восстанавливать исходное программное обеспечение. Часто эти дешифраторы - это просто сам шифратор с другим флагом ввода

Если вы реверсируете зашифрованное приложение, расшифровка вернет вас к исходному двоичному файлу. Поскольку это будет намного проще для анализа, посмотрите, есть ли доступный дешифратор, прежде чем приступать к обратному проектированию. Некоторые распространенные крипторы для проверки включают Криптор Йоды, морфин и PGMP.

Резюме
При рассмотрении вариантов защиты нет "серебряной пули". У большинства методов противодействия обращению вспять также есть недостатки.

Запутывание влияет на производительность и усложняет законную отладку. Однако на разумных уровнях это может быть хорошим вариантом для замедления RE, особенно для декомпилируемых языков.

Защита от отладки оказывает относительно незначительное влияние на время повторного использования (многие отладчики имеют плагины, позволяющие обойти все распространенные приемы защиты от отладки) и усложняет законную отладку. Однако этого может быть достаточно, чтобы помешать начинающим взломщикам. Упаковщики снова поднимают планку сложности для реверс-инжиниринга и упаковки, но, тем не менее, остерегайтесь коммерческих готовых упаковщиков (COTS), у которых обычно есть соответствующие общедоступные распаковщики.

Шифраторы и дешифраторы значительно усложняют RE. Это делает их полезными для защиты программного обеспечения. Однако, при неосторожном использовании они могут помечать обычные AV, поскольку при злонамеренном использовании они также могут помочь защитить вредоносное ПО.

При рассмотрении вопроса о том, следует ли / когда использовать средства защиты от обратного хода, вам следует взвесить такие компромиссы, как эффективность, устойчивость, скрытность и стоимость. Подумайте о своем противнике и его целях:

- Конкурирующая компания (кража IP)
- Случайный взломщик (низко висящий фрукт)
- Профессиональный взломщик (крупные/высокоценные цели)

Кроме того, подумайте, что нужно защищать:

Проверка ключа? Вся программа? Большинство защит могут быть применены к конкретным функциям.
Учтите, что добавление защит может привлечь внимание к цели.
Распространенная ложь заключается в том, что “все поддается взлому и может быть подвергнуто обратному проектированию, если кто-то достаточно постарается, поэтому нам не следует утруждать себя [защитой /запутыванием / шифрованием / и т.д.]”. Это грубое непонимание того, чего должна достичь защита. Часто взломщик может отказаться от взлома, реверсирования, взламывания или поломки продукта только потому, что это перестало приносить удовольствие.

Если вы сможете достаточно замедлить реверс-инжиниринг, вы выполнили свою работу. Часто надежный дизайн с умеренными настройками готового коммерческого обфускатора (COTS) является наилучшим доступным вариантом. Вам нужно будет взвесить каждый подход, исходя из потребностей проекта.

Нет худа без добра. Не позволяйте совершенству быть врагом хорошего. Как только будет выбран разумный подход, в ваш DevOps могут быть встроены обфускаторы, средства защиты от отладки, упаковщики и т. д.

Глава 14
Обнаружение и предотвращение
Разработчики приложений используют различные механизмы для обнаружения и защиты от обращения вспять и взлома. Однако некоторые из этих методов более эффективны, чем другие. В этой главе рассматриваются некоторые из наиболее распространенных методов, их относительные сильные и слабые стороны, а также способы их устранения.

CRC
Циклическая проверка избыточности (CRC) - это математический расчет, выполняемый для байтов данных, подлежащих защите. Результат сохраняется в виде CRC, который часто добавляется к данным (т.е. data CRC). Чтобы проверить данные, пересчитайте и сравните.

Алгоритмы CRC имеют свои преимущества, в том числе следующие:

- Быстрые и компактные
- Легко ускоряются с помощью аппаратных средств
- Быстро вычисляются и сравниваются
- Доступно множество опций (IEEE802.3, CRC-32 и т.д.)
В целом, CRC отлично подходят для обнаружения случайных ошибок или модификаций, таких как ошибки передачи.

Однако они являются слабой защитой от преднамеренных ошибок или модификаций. CRC могут быть легко пересчитаны и обновлены злоумышленником. Например, упрощенный CRC может сложить все байты вместе и сохранить результат. Если бы в файле где-то в данных произошло повреждение, то новая сумма не соответствовала бы, и можно было бы предпринять действие. Если повреждение произошло в CRC-части файла, то сумма не соответствовала бы поврежденному CRC, и можно было бы предпринять действие. Это отлично подходит, например, для определения того, был ли случайно перевернут бит во время загрузки.

Но поскольку пересчет CRC настолько тривиален, злоумышленнику несложно внести свои изменения и просто обновить CRC, включив в него новые значения.

Является ли это надежной защитой?
Сравнение CRC с триадой CIA дает неутешительные результаты:

Конфиденциальность: Отсутствует
Целостность: Очень низкая (злоумышленнику слишком легко пересчитать и поместить новый CRC в файл)
Доступность: Отсутствует
Эту защиту можно легко обойти, создав новый действительный CRC. В качестве альтернативы вы можете просто исправить проверку CRC. CRC эффективны для обнаружения случайного повреждения, но бесполезны для преднамеренного повреждения.

Подпись кода
Многие организации подписывают свой код цифровой подписью перед его выпуском. Это связано с тем, что подписание кода обеспечивает два основных преимущества:

- Аутентичность: Цифровая подпись может быть сгенерирована только с использованием правильного закрытого ключа. Это доказывает, что программное обеспечение получено от его предполагаемого создателя.
- Целостность: Изменение данных с цифровой подписью делает недействительной их

подпись. Подпись кода доказывает, что программное обеспечение не было изменено после выпуска.
Подпись кода защищает от широкого спектра потенциальных атак. Однако, с точки зрения взломщика, наиболее существенное влияние заключается в том, что он может предотвратить исправление, если программа проверяет свою подпись перед выполнением.

Как подписать код
Подписание кода работает с использованием открытого ключа или асимметричной криптографии. Эти криптографические алгоритмы используют пару открытого и закрытого ключей. Чтобы подписать код, вам сначала нужно сгенерировать пару открытых/закрытых ключей.

Цифровые подписи проверяются с использованием вашего открытого ключа; однако вам нужен способ доказать, что конкретный открытый ключ принадлежит вам. Здесь на первый план выходит инфраструктура открытых ключей (PKI). Используя сгенерированный открытый ключ, вы подаете заявку на получение сертификата в центре сертификации подписи кода (CA). Центр сертификации подтвердит вашу личность и выдаст цифровой сертификат, который содержит ваш открытый ключ и подтверждает ваше право собственности на него.

С помощью этого сертификата теперь вы можете генерировать цифровые подписи. Для этого вам нужно сгенерировать хэш исполняемого файла и зашифровать этот хэш с помощью закрытого ключа. Затем, когда вы распространяете исполняемый файл, вы связываете полученную подпись и свой цифровой сертификат с исполняемым файлом.

Хотя вы можете выполнить этот процесс вручную, многие инструменты сборки сделают это за вас. Вам все равно потребуется купить сертификат и загрузить его в свой инструмент сборки, но затем вы можете попросить инструмент сборки подписать приложение. Если вы впервые сталкиваетесь с PKI, знайте, что это намеренно лишь поверхностное описание; существует множество книг, посвященных именно этой концепции.

Как проверить подписанное приложение
Подпись кода - это, по сути, зашифрованный хэш исполняемого файла. После проверки того, что открытый ключ действителен, используя связанный сертификат, вы можете расшифровать хэш исполняемого файла. Затем вы самостоятельно вычисляете хэш приложения, используя ту же хэш-функцию, что и разработчик приложения. Если вы сравниваете два хэша и они совпадают, приложение является подлинным и неизмененным. Если они отличаются, приложение является поддельным или было подделано.

Большинство операционных систем проверят подписи кода за вас. Операционная система также выдаст предупреждение, если открытый ключ, используемый для генерации подписи кода, не проверен, как показано на рисунке 14.1. Однако большинство пользователей все равно нажмут кнопку Выполнить.

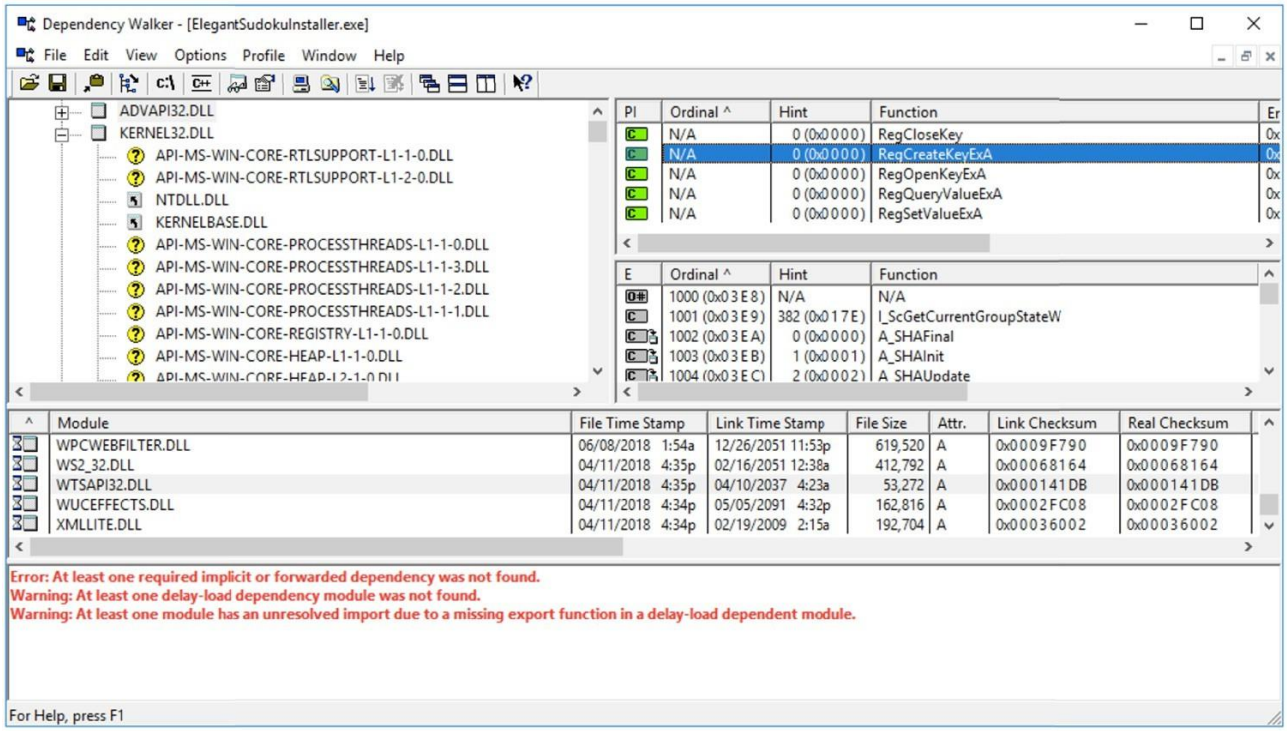


Рисунок 14.1: Предупреждение Windows о непроверенной программе

Эффективна ли подпись кода?
Останавливает ли подпись кода все атаки с исправлениями? Нет.

Причина этого в том, что должен существовать фрагмент неподписанного кода, который выполняет проверку подписи. Это включает в себя выполнение нескольких действий:

1. Вычислите хэш кода.
2. Проверьте, так ли это должно быть.
    2a. Если ответ правильный, запустите код.
    2b. Если ответ неверный, не запускайте код.

Этот код проверки подписи не может быть включен в подпись кода, поскольку он должен содержать хэш-значение для сравнения (или иметь доступ к нему). Невозможно предсказать, каким будет это значение без хэширования приложения. Если вы хэшировали приложение (которое включает это значение) и включили хэш в приложение, то измененное приложение будет иметь новый хэш.

Поскольку код подтверждения подписи не может быть подписан, существует другое местоположение, которое можно было бы исправить, чтобы обойти подписание кода. Однако подписание кода является одним из лучших методов защиты целостности программного обеспечения как от случайных, так и от преднамеренных модификаций.

Подписание кода против CRC
CRC обычно используются для обнаружения битовых ошибок в данных, передаваемых по сети. Однако они обеспечивают защиту только от случайных изменений, а не от преднамеренных. CRC легко пересчитываются злоумышленником.

Подпись кода так же надежна, как и защита вашего закрытого ключа. Без закрытого ключа

злоумышленник не сможет восстановить действительный подписанный хэш.

Является ли это надежной защитой?
Подписание кода работает намного лучше, чем CRC, по сравнению с триадой CIA.

- Конфиденциальность: Отсутствует
- Целостность: Да! Фантастический
- Доступность: Отсутствует

Более сложный подход к отказу от подписи кода заключается в краже закрытых ключей подписи и использовании их для цифровой подписи измененной версии приложения.

RASP

Самозащита приложения во время выполнения (RASP) обеспечивает безопасность запущенного приложения. Это достигается путем перехвата системных вызовов и проверки того, что они исходят из ожидаемого источника. Он также перехватывает манипуляции с данными и проверяет, что они поступают из авторизованных источников.

RASP - это реактивная защита. Его можно настроить на “остановку” атак в реальном времени. Например, RASP может выполнять следующее:

- Отбрасывать/удалять вызов, который он считает вредоносным, например подозрительный SQL-вызов в приложении.
- Завершать сеанс пользователя.
- Останавливать выполнение.

Функция подсоединения

Одним из методов, который использует RASP, является подключение функции. Это включает в себя перезапись первых нескольких байтов кода функции с переходом к коду RASP.

Код RASP будет включать проверки, чтобы убедиться, что вызов является законным. Это может включать следующее:

- Проверка параметров и контекста вызова
- Проверка того, что код не был изменен (возможно, сравнение хэша функции с известным хорошим хэшем)
- В конце кода RASP затем будет выполнен перезаписанный код, прежде чем вернуться к исходной функции.

Риски RASP

Если RASP обнаруживает атаку, он может остановить выполнение. Однако это может быть неприемлемо в зависимости от варианта использования вашего программного обеспечения. Например, в больницах, на производстве, в критически важных объектах инфраструктуры, автомобилях и аналогичных средах внезапная остановка приложения может представлять значительный риск для здоровья и безопасности.

RASP также может иметь свои недостатки даже при отсутствии атаки. Некоторые эффекты включают следующее:

- Скорость: Из-за подключения функции RASP оказывает нетривиальное влияние на скорость.
- Размер: Привязка функций и таблицы поиска помогают обеспечить безопасность; однако они также приводят к раздуванию двоичных файлов.

Является ли это надежной защитой?

RASP дает неоднозначные результаты при сравнении с эталонным тестом ЦРУ:

Конфиденциальность: Нет
Целостность: Да (для разделов, которые защищает RASP, проверка контекста во время выполнения является очень эффективной проверкой)
Доступность: Нет (фактически может быть отрицательной)
Если RASP настроен правильно, его может быть трудно обойти. Вы не сможете легко исправить приложение, если включена подпись кода, и вы не сможете легко отменить работу приложения, если включена защита от отладки.

Однако RASP открывает потенциальный путь для атаки на доступность. Если вы сможете идентифицировать входные данные, которые он воспринимает как “атаку”, вы потенциально можете заставить его отключиться.

Разрешить внесение в список
Список разрешений, иногда называемый белым списком, предоставляет среде выполнения список “полезных” вещей. Например, компьютер может разрешать запуск только приложений, включенных в список разрешений.

Существует множество коммерческих продуктов, которые предоставляют списки разрешений. Например, операционная система Windows имеет встроенные политики ограничения программного обеспечения.

С точки зрения взлома, список разрешений может предотвратить использование инструментов для взлома и обратного проектирования. Например, такие инструменты, как Procmon, отладчики, Cheat Engine, ResourceHacker, Dependency Walker и другие приложения для реверсирования и взлома, вряд ли будут включены в список разрешений.

Список разрешений сложно составить правильно. Может быть сложно узнать все различные библиотеки, которые нужны вашему приложению. При создании белого списка необходимо выполнить большое количество тестов, чтобы убедиться, что все необходимые приложения и библиотеки включены в список разрешений.

Как работает Allowlisting
Существует два основных подхода к allowlisting. Список может основываться на имени процесса или на его хэше. Эти списки применяются только при первом запуске приложения.

Разбивка списков разрешений на основе имен
Списки разрешений отслеживают имена процессов или приложений, разрешенных к выполнению. Чтобы обойти этот тип списка, присвоите своему вредоносному приложению имя, одобренное для включения в белый список. Например, вы определяете, что solitare.exe включено в список разрешенных, поэтому вы даете своему вредоносному приложению имя solitare.exe.

Списки разрешений на основе ломаных имен и хэшей
Если в списке разрешений используются как имена приложений, так и хэши, его нельзя обойти путем переименования приложения. Хэш вредоносного приложения не будет совпадать с хэшем законного.

Однако эти списки разрешений можно обойти с помощью внедрения процесса. Как только приложение, включенное в список разрешений, запущено, если вы можете запустить выполнение кода, вы можете внедрить вредоносные библиотеки. И хотя это легко сказать,

получение выполнения кода часто не является тривиальным. Итак, это один из тех случаев, когда это звучит просто, потому что это можно сказать в предложении, но на самом деле наличие обязательного условия выполнения кода в процессе, внесенном в белый список, может стать настоящим препятствием для взломщика.

Если вы получили неуловимое выполнение кода в приложении, включенном в список разрешенных, существует множество способов загрузки в процесс. В Windows вы можете использовать LoadLibrary() или SetWindowsHookEx(). В Linux вы можете использовать ptrace()/PTRACE_POKEDATA/opcodes для системного вызова uselib().

Хэш приложения проверяется перед запуском приложения. Изменения в приложении после его запуска не будут обнаружены списком разрешений.

Пример: Metasploit
Metasploit - популярный инструмент для взлома. Его основная цель - использовать приложение и внедрить meterpreter, который предоставляет злоумышленнику удаленный доступ к зараженному компьютеру. (Ссылки смотрите в разделе “Инструменты” нашего репозитория.)

При использовании Metasploit новые приложения не запускаются; во взломанный процесс внедряется meterpreter. Оттуда он может “развернуться” в любое другое запущенное приложение.

Является ли это надежной защитой?
Список разрешений обеспечивает ограниченную защиту:

- Конфиденциальность: Нет
- Целостность: Да (в сочетании с именем и хэшем; однако проверка целостности обычно выполняется только во время запуска приложения)
- Доступность: Нет

Список разрешений можно обойти несколькими способами. Вредоносная программа может выдавать себя за законное приложение, чтобы обойти белый список на основе имен, или использовать внедрение кода для обхода списка разрешений, который использует как имена, так и хэши.

Внесение в черный список
Внесение в черный список, иногда называемый черным списком, является полной противоположностью внесению в разрешенный список. Вместо указания всего, что разрешено, это список всего, что запрещено. Список блокировок может основываться на именах, ключах или хэшах.

Списки блокировок легко создавать, но сложно поддерживать. Например, рассмотрим список блокировок, включающий вредоносный исполняемый файл virus1.exe. Что произойдет на следующей неделе, когда выйдет virus2.exe?

С точки зрения взлома, вы можете заблокировать ключи, которые, как вам известно, являются плохими (т.е. взломанными). В зависимости от того, как работает ваша генерация ключей, может оказаться возможным заблокировать целые подмножества ключей.

Альтернативно, программа также может отказаться запускаться, если видны определенные другие приложения. Например, приложение может не запускаться, если установлен отладчик.

Многие антивирусы используют этот подход для выявления и блокировки известных вредоносных программ. Они включают список “сигнатур” известных вредоносных приложений. Если что-то совпадает с подписью, оно помечается как плохое.

Это надежная защита?
Блок-лист обеспечивает меньшую защиту, чем белый список:

- Конфиденциальность: Отсутствует
- Целостность: Некоторая (в сочетании с хэшами или ключами)
- Доступность: Отсутствует

Способы обхода блок-листа зависят от информации, которую он использует для идентификации вредоносных приложений. Если оно основано на имени, измените имя. Если оно хранит хэши заведомо вредоносных программ, измените его, внеся небольшие изменения в код приложения или данные, чтобы изменить его хэш.

Удаленная аутентификация
Для большинства стратегий защиты от реверсирования и взлома у злоумышленника есть все элементы, необходимые для преодоления защиты. Имея достаточно времени, он может провести реинжиниринг и/или исправление приложения.

Удаленная аутентификация требует, чтобы приложение извлекало что-либо удаленно для работы. Например, он может получить ключ с удаленного сервера, который он использует для расшифровки какого-либо важного кода.

Большинство злоумышленников перепроектируют систему “в автономном режиме”. Они не хотят, чтобы она подключалась к вашим серверам, потому что не хотят, чтобы у вас был их IP-адрес или чтобы вы знали, что они запускают ваше программное обеспечение. Имейте в виду, что при попытке взломать часть программного обеспечения вы, скорее всего, часто запускаете программу автозагрузки и проверяете код. В то время как законный пользователь, скорее всего, запускал бы приложение максимум несколько раз в день. Такое поведение действительно легко обнаружить на удаленном сервере аутентификации. Пользователь, который проходит аутентификацию 100 раз в день, скорее всего, совершает что-то гнусное.

Построение архитектуры приложения таким образом, чтобы оно не могло запускаться без информации с сервера, помогает предотвратить обратное проектирование. Злоумышленнику нужно будет либо отменить его “онлайн”, либо отказаться от него.

Пример удаленной аутентификации
Один из возможных подходов к реализации удаленной аутентификации заключается в шифровании каждой части приложения, за исключением загрузчиков. Загрузчик отправляет системную информацию, хэш программного обеспечения и ключ активации на сервер.

Сервер проверит ожидаемый хэш и ключ активации. Если они подтвердятся, он использует алгоритм для получения ключа дешифрования, который он отправит обратно приложению. Затем загрузчик может расшифровать приложение, позволяя ему запускаться.

Злоумышленник не сможет “имитировать” ваш удаленный сервер и алгоритм без доступа к серверному коду. Единственный способ изучить программное обеспечение - активировать его онлайн. Код расшифровки приложения может находиться только в памяти. Таким образом, каждый запуск требует взаимодействия с сервером.

Основная проблема этого подхода заключается в том, что внедрение решений для криптографии и управления корпоративными ключами не является тривиальным. Ошибка может позволить взломщику обойти код проверки и сгенерировать свои собственные ключи дешифрования. Как обсуждалось в случае с упакованными приложениями, как только они распакованы в памяти, вы можете извлечь из них дамп памяти для будущего статического анализа. Однако этот дамп памяти нелегко (а иногда и вовсе невозможно) превратить в расшифрованное приложение, способное к запуску. Дамп памяти не будет полезен для исправления или тестирования модификаций, но никогда не стоит недооценивать ценность статического анализа.

Является ли это надежной защитой?
Удаленная аутентификация дает неоднозначные отзывы по сравнению с триадой ЦРУ:

- Конфиденциальность: Некоторая (приложение в конечном итоге будет расшифровано в памяти, но двоичный файл в состоянии покоя ограничен)
- Целостность: Да (сервер должен выполнить некоторую проверку целостности перед отправкой ответа)
- Доступность: Возможно, отрицательная

Одной из возможных атак на удаленный сервер является установка поддельного сервера. Для начала активируйте приложение онлайн и перехватите все сообщения между приложением и сервером.

Затем запустите поддельный сервер с соответствующими ответами. Код приложения будет расшифрован и может быть сохранен на диске.

При таком подходе для получения расшифрованного кода требуется одно онлайн-приложение. Однако это позволяет создать полный расшифрованный двоичный файл, что делает ненужной дальнейшую онлайн-аутентификацию. Но обратите внимание, что этот подход не всегда может работать, если приложение проявляет должную осмотрительность, требуя определенных сертификатов от сервера, или если запрос/ответ от сервера не всегда одинаков (т.е. меняется со временем или датой).

Лабораторная работа: ProcMon
Эта лабораторная работа показывает, что существует более одного возможного способа взлома программы. Вернитесь на страницу книги на GitHub (https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES) и найдите ProcMon lab.Skills.

В этой лабораторной работе используются ProcMon и IDA, чтобы понять возможности альтернативных решений для взлома. Некоторые ключевые навыки, которые тестируются, включают следующее:

- Динамический анализ поведения программы
- Выявление косвенных подходов к обходу программных защит

Выводы
Наблюдать за тем, что делает процесс извне, может быть быстрее / проще, чем наблюдать за ним изнутри (то есть отладка не всегда является лучшим подходом).

Обычно существует много способов взлома программы; поиск наилучшего требует практики.

Резюме
В этой главе представлены различные методы защиты от взлома программного обеспечения

и его реверсирования. Некоторые методы, как правило, неэффективны, в то время как другие могут работать, но также имеют некоторые недостатки.

Важно помнить, что практически любую защиту можно победить, если приложить достаточно времени и усилий. Цель состоит в том, чтобы замедлить атакующего и, в идеале, расстроить его настолько, чтобы он сдался.

Глава 15

Законность
Во “Введении” на высоком уровне были рассмотрены юридические последствия и соображения, связанные с обратным проектированием и взломом. В этом разделе приводится более подробное обсуждение соответствующих законов США, их последствий и интерпретаций.

ПРЕДУПРЕЖДЕНИЕ

В качестве оговорки, мы не юристы, и это не юридическая консультация. Если вам нужна юридическая консультация, пожалуйста, свяжитесь с любым уважаемым юристом или Фондом Electronic Frontier по адресу www.eff.org, который имеет глубокую специализацию в области безопасности.

Законы США, влияющие на реверс-инжиниринг
Законы, касающиеся авторских прав, взлома и т.д., различаются в зависимости от юрисдикции. В этом разделе рассматриваются некоторые применимые законы Соединенных Штатов. Если вы находитесь в другом месте, ознакомьтесь с местными законами и ограничениями.

Закон об авторском праве в цифровую эпоху
Управление цифровыми правами (DRM) - это решение, разработанное для защиты интеллектуальной собственности. Решения DRM позволяют отслеживать защищенный контент и контролировать его после того, как он попадает на рынок.

Закон об авторском праве в цифровую эпоху (DMCA) был принят Конгрессом в 1998 году. Это привело Соединенные Штаты к соблюдению международных соглашений об авторском праве.

Закон о компьютерном мошенничестве и злоупотреблениях
Закон о компьютерном мошенничестве и злоупотреблениях (CFAA) был принят в 1984 году. Это федеральный закон о борьбе со взломом, который запрещает несанкционированный доступ к компьютерам и сетям.

Законодатели написали закон настолько плохо, что с тех пор им злоупотребляют творческие прокуроры. Однако в последние годы были предприняты усилия по защите исследователей в области безопасности от судебного преследования в соответствии с ним. Wired.com об этом говорилось в статье CFAA за 2014 год:

Громкий случай, связанный со злоупотреблением статутом, произошел в 2008 году, когда трем студентам Массачусетского технологического института было запрещено выступать с презентацией на хакерской конференции Def Con. Студенты обнаружили недостатки в системе электронных билетов, используемой транспортным управлением Массачусетского залива, которые позволили бы любому получить бесплатные поездки. MBTA запросило и получило временный судебный запрет, чтобы запретить студентам говорить о недостатках. При вынесении временного постановления о неразглашении судья сослался на CFAA, заявив, что информация, которую студенты планировали предоставить, предоставит другим средства для взлома системы. Слова судьи подразумевали, что просто говорить о взломе - это то же самое, что и сам взлом. Однако это решение подверглось публичной критике как неконституционное предварительное ограничение свободы слова, и

когда MBTA впоследствии добилась судебного приказа о том, чтобы сделать запретительный приказ постоянным, другой судья отклонил запрос, постановив частично, что CFAA не распространяется на свободу слова и, следовательно, не имеет отношения к делу.

www.wired.com/2014/11/hacker-lexicon-computer-fraud-abuse-act

Второе громкое и очень печальное злоупотребление CFAA привело к громкому самоубийству. Это произошло после того, как прокурор США использовал CFAA для возбуждения жесткого судебного преследования против интернет-активиста Аарона Шварца за то, что многие сочли незначительным нарушением. Шварцу, который помогал разрабатывать стандарт RSS и был соучредителем правозащитной группы Demand Progress, было предъявлено обвинение после того, как он получил доступ к шкафу в Массачусетском технологическом институте и якобы подключил ноутбук к университетской сети для загрузки миллионов научных статей, которые распространялись службой подписки JSTOR. Шварца обвинили в том, что он неоднократно подделывал MAC-адрес своего компьютера, чтобы обойти блокировку, которую MIT наложил на адрес, который он использовал.

Хотя JSTOR не подала жалобу, Министерство юстиции продолжило судебное преследование Шварца. Прокурор США Кармен Ортис настаивала на том, что “кража есть кража” и что власти просто соблюдали закон. Шварц, отчаявшись из-за предстоящего судебного разбирательства и перспективы осуждения за уголовное преступление, покончил с собой в 2013 году. В ответ на трагедию два законодателя предложили давно назревшую поправку к закону, которая помогла бы предотвратить чрезмерное использование прокурорами этого закона. Поправка, называемая законом Аарона, была внесена через несколько месяцев после смерти Шварца представителем. Зои Лофгрен (штат Калифорния) и сенатор. Рон Уайден (штат Орегон). Поправка исключит из закона нарушения условий предоставления услуг и пользовательских соглашений, а также сузит определение несанкционированного доступа, чтобы провести четкое различие между преступной хакерской деятельностью и простыми действиями, которые превышают разрешенный доступ на незначительном уровне. Вместо этого поправка предлагает определить несанкционированный доступ как “обход одной или нескольких технологических мер, которые исключают или препятствуют неавторизованным лицам получать или изменять” информацию на защищенном компьютере. Поправка также разъяснит, что акт обхода не будет включать в себя простое изменение пользователем своего MAC или IP-адреса для получения доступа к системе.

Закон об авторском праве

Согласно Закону об авторском праве 1976 года, авторское право на компьютерную программу возникает, когда исходный код компьютерной программы пишется программистом. Программа не обязательно должна быть полной или даже функциональной, чтобы возникла защита авторских прав. Прецедентное право в области авторского права рассматривает авторские права на исходный код и объектный код как эквивалентные.

Если вы не являетесь владельцем авторских прав, как правило, выполнение любого из следующих действий без разрешения является незаконным:

- Копирование программы или частей программ для передачи или продажи кому-либо другому
- Предварительная загрузка программы на жесткий диск продаваемого компьютера
- Распространение программы через Интернет
- Обход средств контроля, препятствующих доступу к материалам, защищенным авторским правом

Однако из этого есть много исключений и нюансов. Первое авторское право на программное обеспечение было зарегистрировано в 1964 году. Основанием для того, чтобы начать предоставлять защиту программному обеспечению, было то, что теперь они рассматривали компьютерную программу как “учебное пособие”. Закон об авторском праве 1976 года официально относит программное обеспечение к объектам авторского права.

Итак, когда часть программного обеспечения защищена авторским правом, что именно защищено авторским правом? Авторское право защищает выражение идеи, а не саму идею. Например, если вы разрабатываете концепцию игры lemonade stand, вы можете защитить авторским правом свою реализацию, но не идею игры lemonade stand. Во-вторых, защита защищает объектную (исполняемую) программу, а не исходный код. Наконец, он защищает экранные изображения, создаваемые программой во время ее выполнения.

Исходный код программного обеспечения, как правило, хранится в качестве коммерческой тайны и не публикуется в соответствии с авторским правом.

Важные судебные дела
В дополнение к законам, судебный прецедент важен для определения того, что является законным, а что нет в Соединенных Штатах. Пара важных судебных дел включает следующее:

Apple Computer против Franklin Computer: Установлено, что объектные программы защищены авторским правом.
В начале 1980-х годов Franklin Computer corporation начала выпускать компьютер Franklin Ace, чтобы конкурировать с Apple II. Franklin ACE был совместим с программами Apple 2. Для этого Franklin Ace скопировал некоторые функции операционной системы непосредственно из ПЗУ Apple II. Apple подала в суд на Franklin Computers за нарушение авторских прав, поскольку они скопировали их объектный код. Apple выиграла.
Sega против. Accolate: Установлено, что дизассемблирование объектного кода для определения технических характеристик является добросовестным использованием.
Производитель видеоигр Accolate хотел создать несколько своих игр для Sega Genesis. Однако Sega не поделилась техническими характеристиками системы, поэтому Accolade разобрала объектный код игры Sega, чтобы определить, как она работает. Sega подала в суд на Accolade за нарушение их авторских прав. Однако на этот раз суд вынес решение в пользу Accolade, поскольку действия Accolade представляли собой добросовестное использование программного обеспечения.
Из этих двух судебных дел стало ясно, что обратное проектирование допустимо до тех пор, пока вы не нарушаете авторские права. Напомним, что Franklin Computer нарушила авторские права, скопировав часть кода Apple. Там, где Accolate не нарушала авторские права, они не копировали никаких копирайтных материалов; они просто извлекали из них уроки.

Один из способов убедиться, что вы попадаете в ситуацию "правильного использования", как Accolate, а не в ситуацию копирования, как Franklin Computers, - это использовать так называемую стратегию программного обеспечения для чистых помещений. Это заключается в том, что две команды разделены; каждая выполняет разные части работы. Первая команда изучила бы систему или программу конкурента и написала технические спецификации того, как она работает. Вторая команда использовала бы эти спецификации для разработки новой системы.

Если бы Franklin Computer использовала этот подход, некоторые члены команды выяснили бы, как работает система Apple, и описали функциональность. Затем, если бы вторая команда

реализовала это самостоятельно, никогда не видя реализации Apple, это потенциально могло бы изменить исход мероприятия. Если бы они подошли к ситуации таким образом, то, скорее всего, не пострадали бы от судебных исков, потому что они не использовали бы какой-либо код Apple.

Главное - избежать неосознанного копирования кода. Если бы команда, исследовавшая Apple II, также была командой, реализовывавшей спецификацию, они, вероятно, страдали бы от предрасположенности к использованию подобного кода в системе Apple, потому что это то, что они уже видели.

Добросовестное использование
Иногда законно воспроизводить защищенную авторским правом работу без разрешения. В целом суды учитывают четыре фактора при оценке того, подпадает ли что-либо под исключения добросовестного использования:

Назначение и способ использования: Если целью является критика, комментарии, новостные репортажи, обучение или исследование, то это, скорее всего, допустимо. Однако коммерческое использование, скорее всего, нет.
Как насчет характера использования? Наиболее важным фактором является то, насколько произведение было изменено по сравнению с оригиналом. Если новый автор добавил новые выражения или смысл, то это потенциально кандидат на добросовестное использование.

- Характер работы: Добросовестное использование более благоприятно относится к документальным произведениям, чем к художественным.
- Объем копируемой работы: Краткий отрывок, скорее всего, подойдет, чем копирование всей книги или целой главы.
- Влияние на рынок произведений, защищенных авторским правом: Например, копирование материалов, вышедших из печати, не имеет такого же материального эффекта, как копирование недавно написанного и напечатанного произведения.
Согласно Закону об авторском праве, 17 U.S.C. § 107, обратное проектирование подпадает под “добросовестное использование”, когда выполняется для “... таких целей, как критика, комментарии, новостные репортажи, преподавание (включая многократные копии для использования в классе), стипендии или исследования....” Однако это сопоставляется с “влиянием использования на потенциальный рынок или стоимость защищенной авторским правом работы”.

Исключение из исследования DMCA
В октябре 2016 года DMCA добавила в закон исключение для добросовестных исследований в области безопасности. В нем говорится, что “доступ к компьютерной программе осуществляется исключительно в целях добросовестного тестирования... когда такая деятельность осуществляется в контролируемой среде, предназначенной для предотвращения какого-либо вреда отдельным лицам или общественности... и не используется или не поддерживается способом, способствующим нарушению авторских прав”.

Это также может относиться к обратному проектированию и взлому. В нем говорится: “...исследователи могут обходить средства контроля цифрового доступа, осуществлять обратное проектирование, получать доступ, копировать и манипулировать цифровым контентом, защищенным авторским правом, не опасаясь судебного преследования — в разумных пределах”.

Это не карточка, дающая право на освобождение из тюрьмы, и не пустое разрешение взламывать все подряд. Это свидетельствует об эволюции отрасли, о признании того, что

исследования в области безопасности, проводимые по правильным причинам, - это хорошо, и что закон теперь защитит тех, кто проводит добросовестные исследования.

Законность
Закон об авторском праве в отношении обратного проектирования и модификации кода в значительной степени подчеркивает намерения и последствия. Если вы действуете самостоятельно, проконсультируйтесь с юристом... или держите это при себе. Это не имеет в виду ничего подлого, но помните, что частью добросовестного использования является эффект, который ваша работа оказывает на рынок. Если вы работаете в сфере образования или научных исследований, и ваши результаты остаются с вами, они на самом деле не влияют на потенциальный рынок сбыта вашей работы. Это ключевой фактор добросовестного использования. В ту секунду, когда вы используете свои знания для создания кейгена, который размещаете в Сети и из-за которого поставщик теряет деньги, это больше не считается добросовестным использованием. Но если вы держите все это при себе таким образом, чтобы это не влияло на рынок или других людей, то вы прошли долгий путь, чтобы подпадать под действие закона о добросовестном использовании.

ПРЕДУПРЕЖДАЮ
еще раз, мы не юристы, это не юридическая консультация, и это наша интерпретация и понимание нормативно-правовой базы в Соединенных Штатах, которая влияет на реверс-инжиниринг.

Резюме
В этой главе рассмотрены некоторые юридические аспекты обратного проектирования и взлома, но мы не юристы. За юридической консультацией мы рекомендуем обращаться в EFF.

Глава 16
Продвинутые методы
До этого момента в этой книге рассматривались основные инструменты и навыки, используемые для обратного проектирования и взлома. Однако это развивающаяся область, и разрабатываются новые методы, позволяющие сделать это быстрее и проще. В этом разделе на высоком уровне описаны некоторые передовые методы и инструменты на переднем крае реверс-инжиниринга. Наша цель в этой главе состоит в том, что если на данный момент вам все еще нравится взламывать программное обеспечение и вы хотите перейти на следующий уровень, мы хотим представить вам множество кроличьих нор, в которые можно спуститься. В зависимости от того, что вас интересует, мы надеемся, что нижеследующее укажет вам правильные направления для углубления.

Вневременная отладка
Вневременная отладка также известна как обратная отладка. Основная идея такова: “что, если бы мы могли вернуться назад при отладке?”

Рассмотрим случай, когда что-то пошло не так во время отладки. Возможно, произошел сбой исправления, вы пропустили анти-отладочную проверку, вы не знаете, как вы сюда попали и т.д.

Существует несколько различных инструментов, предназначенных для вневременной отладки, включая следующие:

- Visual Studio Ultimate (.NET)
- rr
- gdb

Для начала ознакомьтесь с докладом Джорджа Хотца @ Enigma о USENIX Enigma в 2016 году по адресу www.youtube.com/watch?v=eGl6kpSajag.

Двоичный инструментарий
Двоичный инструментарий - это когда вы вводите код для наблюдения или модификации процесса по мере его выполнения. Это может быть полезно для поиска утечек памяти, отслеживания проверок ключей, выполнения анти-анти-отладки и т.д.

Некоторые инструменты для бинарного инструментирования включают следующее:

- PIN
- DynamoRIO
- Frida
- Valgrind
- QBDI

Для ознакомления с бинарным инструментарием ознакомьтесь с докладом Blackhat USA 2015 года “Расширение статического анализа с помощью Pintool: удаление” по адресу www.youtube.com/watch?v=wHIlNRK_HiQ.

Промежуточные представления
Обычно для реверсирования и взлома необходимо изучать и писать инструменты для каждой новой архитектуры. Идея промежуточных представлений заключается в переводе всего ассемблерного кода для всех архитектур на один и тот же язык. Таким образом, вы можете изучать и писать инструменты только для этого языка.

Существует несколько различных инструментов, которые можно использовать для работы с

промежуточными представлениями, включая следующие:

- Binary Ninja
- REIL
- VEX
- BNIL
- Ghidra PCode
- IDA microcode
- LLVM IR

Чтобы начать работу с промежуточными представлениями, ознакомьтесь с “Поиском ошибок с помощью Binary Ninja” Джордана Винса из LevelUp 0x03 по адресу www.youtube.com/watch?v=55gClG-sjWc.

Декомпиляция

Идея декомпиляции заключается в восстановлении исходного кода на основе расширенного автоматизированного анализа ассемблерного кода. Некоторые инструменты, которые предлагают декомпиляцию, включают следующие:

- IDA's Hex-Rays
- Ghidra
- Binary Ninja
- Snowman Decompiler

Чтобы узнать больше о декомпиляции, ознакомьтесь с разделом “Декомпиляция вируса с помощью IDA Pro” по адресу www.youtube.com/watch?v=gYkDcUO9otQ.

Автоматическое восстановление структуры

Автоматическое восстановление структуры включает автоматический поиск шаблонов и ссылок в памяти, чтобы сделать выводы об используемых типах данных. Некоторые инструменты для этого включают следующее:

- dynStruct
- Cheat Engine

Чтобы узнать больше об автоматическом восстановлении структуры, ознакомьтесь с идеей и статьей dynStruct по адресу https://github.com/ampotos/dynStruct.

Визуализация

Списки кода и текст могут быть сложными для восприятия. Визуализация может быть использована для углубления вашего понимания файловой структуры и выполнения.

Некоторые инструменты реверсирования, которые предлагают полезную визуализацию, включают следующее:

- BinWalk
- Hopper
- IDA plugins
- Veles
- ..cantor.dust..
- Cheat Engine

Хорошей отправной точкой для понимания того, как визуализацию можно использовать для реверсирования, является выступление Кристофера Домаса на Derbycon “Динамическая бинарная визуализация” на www.youtube.com/watch?v=4bM3Gut1hIk.

Деобфускация
Обфускация предназначена для замедления реверсирования в попытке заставить взломщика сдаться. Идея заключается в использовании инструментов для автоматического удаления обфускаций из программ с использованием таких инструментов, как Tigress Protection.

Ознакомьтесь с “Позволяет устранить обфускацию современного двоичного кода” в www.youtube.com/watch?v=TDnAkm6ZTYw.

Проверяющие теоремы
Проверяющие теоремы используют математику для анализа кода, включая сокращение, деобфускацию, границы, входные данные и т.д. Некоторые инструменты доказательства теорем для реверсирования включают следующее:

- Z3
- STP
- Boolector
- Yices

Чтобы увидеть, как можно использовать средства проверки теорем, смотрите “Использование z3 для поиска пароля и обращения к запутанному JavaScript” по адресу www.youtube.com/watch?v=TpdDq56KH1I.

Также ознакомьтесь с ежегодным SMT-COMP!, в котором есть несколько действительно интересных тестов для многих уникальных решателей по адресу https://smt-comp.github.io/2023.

Символьный анализ
Идея символьного анализа заключается в попытке найти входные данные, которые приводят к интересным результатам. Например, какие входные данные могут вызвать сбой, пройти проверку ключа, открыть секрет и т.д.

Инструменты символьного анализа будут отслеживать пользовательский ввод через программу. В каждой ветви они спрашивают проверяющего теорему, какой пользовательский ввод прошел бы по выбранному пути. Какой пользовательский ввод прошел бы по неиспользованному пути?

Для примера рассмотрим следующий код:

```
if (strlen(username)> 10)
    if (key_1^sum(username)==key_2)
        printf("key passed");
```

Инструмент символьного анализа автоматически определит комбинацию username, key_1 и key_2, которая пройдет проверку и достигнет инструкции печати “ключ передан”.

Некоторые инструменты символьного анализа включают следующее:

- Angr
- Mayhem
- KLEE
- Triton
- S2E

Чтобы увидеть пример символьного анализа с помощью Angr, ознакомьтесь с докладом

Шошитаишвили и Вана на DEF CON 23 “Злой хакинг: следующее поколение бинарного анализа” по адресу www.youtube.com/watch?v=oznsT-ptAbk.

Резюме

На данный момент лучший способ улучшить свои навыки реверсирования и взлома - это больше практиковаться. На виртуальной машине Windows папка allthethings на рабочем столе содержит множество различных взломов, отсортированных по уровню сложности.

Глава 17

Дополнительные темы

В последней главе этой книги рассказывается о реверсировании программного обеспечения и взломе. В первую очередь она посвящена пониманию того, как работает программа, и обходу или модификации нежелательных функций (например, средств проверки ключей).

В этой главе эти знания используются для взлома в реальном мире. Разбиение стека и шеллкодинг используют понимание того, как работает программа и стек, для запуска вредоносного кода внутри программы.

Разбиение стека

Разрушение стека, также известное как переполнение буфера на основе стека, является одной из самых классических атак на программное обеспечение. Это использует тот факт, что языки, не защищенные от памяти, такие как C/C++, не имеют встроенной защиты, которая предотвращает доступ приложения к данным в других частях памяти или их перезапись. Например, C/C++ автоматически не проверяет, что данные, записанные в массив, вписываются в границы этого массива. Если вы не знаете C, не волнуйтесь. Если вы знаете какой-либо язык программирования, вы должны быть в состоянии следовать за ним.

Поскольку разбиение стека существует уже так давно, существует множество компиляторов, которые имеют встроенные автоматические средства защиты, которые вводятся в скомпилированный код для предотвращения этого. Хотя атака не так проста, как раньше, каждый должен полностью понимать, как работает атака, потому что:

- Некоторые аспекты этого по-прежнему работают.
- Это основа других типов атак.
- Не каждое приложение имеет стековую защиту.

Для любого из следующих примеров кода на C, если вы создаете их с помощью gcc, вы должны использовать флаг -fno-stack-protector, чтобы отключить эти средства защиты. Создание полной командной строки для использования gcc для сборки в Linux: gcc myfile.c-fno-stack-protector.

Для примера рассмотрим следующую простую программу на C:

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

После того, как это приложение было скомпилировано и объект был выгружен из памяти, результатом является следующий ассемблерный код:

```
function:
      push ebp
      mov esp, ebp
      sub ebp, 20  (*stack shown here)
      leave
      ret
```

```
main:
      push ebp
      mov ebp, esp
      push 3
      push 2
      push 1
      call function
      add esp 0xc
      leave
      ret
```

После выполнения первых трех инструкций в разделе function, включая sub ebp, 20, стек будет выглядеть как следующая таблица с адресами, увеличивающимися от верхней части таблицы вниз:

NAME	SIZE
buffer2	10
buffer1	5
ebp	4
ret	4
a	4
b	4
c	4
ebp	4

Теперь рассмотрим следующий пример кода:

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str); //Copies incoming str to buffer
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';  //creates a string of 255 'A's

  function(large_string);

}
```

В этом коде функция main создает строку, состоящую из 255 As. Затем она передает указатель на этот буфер функции, и функция выделяет 16 байт для локального буфера, но затем копирует (используя strcpy) входной буфер вслепую без проверки длины. Это означает, что входной буфер, который был равен 255 байтам, переполнит локальный буфер, которому было

выделено всего 16 байт.

Если вы запустите код, результатом будет ошибка сегментации (сброс ядра). Ошибка сегментации возникает, когда приложение пытается прочитать, записать или выполнить недопустимый адрес памяти. Давайте копнем глубже, чтобы выяснить, что произошло.

После сборки код преобразуется в следующий ассемблерный код:

```
0804840c <function>:
 804840c:    55                          push   ebp
 804840d:    89 e5                       mov    ebp,esp
 804840f:    83 ec 28                    sub    esp,0x28
 8048412:    8b 45 08                    mov    eax,DWORD PTR
[ebp+0x8]
 8048415:    89 44 24 04                 mov    DWORD PTR
[esp+0x4],eax
 8048419:    8d 45 e8                    lea    eax,[ebp-0x18]  ;[1]
 804841c:    89 04 24                    mov    DWORD PTR [esp],eax
 804841f:    e8 cc fe ff ff              call   80482f0 <strcpy@plt>
 8048424:    c9                          leave
 8048425:    c3                          ret
```

Глядя на это, вы можете видеть, что ebp-0x18 - это адрес в начале буфера (помеченный как [1] в предыдущем коде). Взглянув на преамбулу функции с настройкой стека, вы можете увидеть, что для стека было выделено 0x28 байт. Напомним, что ebp указывает на нижнюю часть стека, а esp - на верхнюю. Следовательно, ebp = esp+0x28.

Итак, во время настройки функции начало массива, в терминах относительно esp, начинается с esp+0x10. Хотя это кажется сложным, все, что это означает, - это то, что буфер находится на расстоянии 0x10 байт от конца выделенного стека функции, что имеет смысл. Напомним, что 0x10 равно 16 в базе 10, а функции выделено 16 байт.

Чтобы увидеть эффекты разбиения стека в действии, запустите приложение в gdb и установите точку останова непосредственно перед операцией strcpy. В точке останова при печати памяти по указателю стека должно отображаться нечто похожее на рисунок 17.1.

```
(gdb) x/16x $esp
0xffffd120:     0x00000000      0xf7fdab18      0x00000000      0x00000000
0xffffd130:     0x00000000      0x00000003      0xf63d4e2e      0x000003f3
0xffffd140:     0x00000000      0xf7e25938      0xffffd278      0x08048470
0xffffd150:     0xffffd16c      0x00000000      0x00000026      0xf7e4a95d
```

Рисунок 17.1: Кадр стека функции перед strcpy

На этом изображении выделенный буфер занимает строку, указанную адресом 0xffffd130, а 0x10 байт после этого - это конец кадра стека функции. Затем следует сохраненное значение предыдущего стека ebp и, наконец, адрес возврата. Значение сохраненного регистра ebp (фрейм стека предыдущих функций) равно 0xffffd278, а адрес возврата равен 0x08048470.

После выполнения операции strcpy та же область памяти будет выглядеть как показано на рисунке 17.2. Операция strcpy перезаписывает буфер, сохраненный регистр ebp и адрес возврата с помощью 0x41 (A).

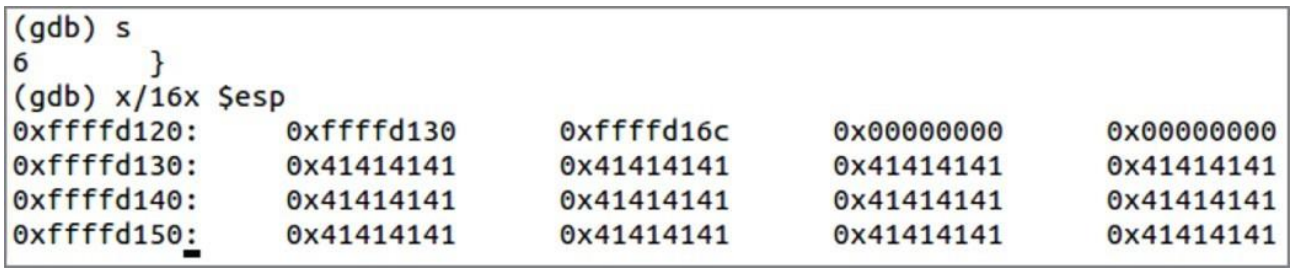


Рисунок 17.2: Стек функций после strcpy

Когда приложение достигает удаленной операции, оно извлекает обратный адрес из стека и пытается продолжить выполнение в этом месте. Однако, поскольку 0x41414141 является недопустимым адресом, сегмент процессора выполняется по умолчанию.

Этот пример приводит к аварийному завершению работы приложения, но это не единственный возможный эффект. На высоком уровне у вас есть возможность управлять обратным адресом и кадром стека предыдущей функции. Хотя манипулирование стековым фреймом имеет свои преимущества, гораздо чаще используется манипулирование обратным адресом, поэтому мы сосредоточимся на этом. В первом случае обратный адрес был перезаписан мусором, но что, если бы мы были более тактичны в отношении того, что мы перезаписываем обратным адресом? Следующий пример кода предназначен для изменения адреса возврата для управления выполнением кода. Цель состоит в том, чтобы пропустить инструкцию x=1 в следующем коде:

```
#include <stdio.h>
void function(int a, int b, int c) {
//do something so we skip x=1 after a return
}
void main() {
     int x;
     x = 0;
     function(1,2,3);
     x = 1;
     printf("%d\n",x);
 }
```

В этом коде функция main устанавливает локальную переменную x и присваивает ей начальное значение 0. Затем она вызывает функцию с некоторыми фиксированными значениями. Внутри функции пока нет кода. Следующий шаг - выяснить, какой код там нужен для достижения цели переписывания обратного адреса.

После возврата из функции функция main обновляет значение x равным 1, а затем переходит к печати значения x. Можем ли мы использовать наши знания о cdecl и настройке стека, чтобы сделать так, чтобы код никогда не запускал x=1 и вместо этого печатал x=0? Да! Задача состоит в том, чтобы записать содержимое функции таким образом, чтобы инструкция x=1 внутри основной функции была пропущена.

Для этого кода стек внутри функции выглядел бы следующим образом:

NAME	ADDR
ebp	ebp
return address	ebp+4
a	ebp+8
b	ebp+12
c	ebp+16

Это ваша обычная стандартная настройка стека cdecl. Вы знаете, что вам понадобится буфер, поскольку в этой главе все о переполнении буфера, поэтому добавьте буфер в функцию. Вам также понадобится способ манипулировать определенными значениями в буфере, поэтому добавьте указатель. Вы также могли бы использовать синтаксис, подобный buffer[z], но указатель помогает более явно указать смещения памяти, что полезно для обучения.

```
#include <stdio.h>
void function(int a, int b, int c) {
char buffer[16];
int *r;
r = 0x99;  //this is here so r is not optimized out
buffer[0] = 0x88; //this is here so buffer is not optimized out
}
void main() {
     int x;
     x = 0;
     function(1,2,3);
     x = 1;
     printf("%d\n",x);
 }
```

При сборке это преобразуется в следующий ассемблерный код:

```
0804840c <function>:
 804840c:    55                          push   ebp
 804840d:    89 e5                       mov    ebp,esp
 804840f:    83 ec 20                    sub    esp,0x20
 8048412:    c7 45 fc 99 00 00 00        mov    DWORD PTR [ebp-
0x4],0x99
 8048419:    c6 45 ec 88                 mov    BYTE PTR [ebp-
0x14],0x88
 804841d:    c9                          leave
 804841e:    c3                          ret
```

Теперь в стеке появились новые объекты - указатель и буфер.

NAME	ADDR
buffer	ebp-0x14
r	ebp-4
ebp	ebp
return address	ebp +4

NAME	ADDR
a	ebp +8
b	ebp+12
c	ebp+16

В этом кадре стека адрес возврата находится в буфере+0x18. Следующим шагом является обновление кода функции, чтобы указатель указывал на этот адрес в памяти.

Для тех, кто не знаком с C, & - это “адрес чего-либо”, поэтому следующий код устанавливает ret так, чтобы он указывал на адрес в памяти, где находится buffer+0x18. Вытянув стек, вы можете увидеть, что это сохраненный обратный адрес. На данный момент обратный адрес не был изменен, но у нас есть указатель на него. Следующий шаг - выяснить, на что его изменить, чтобы пропустить x=1.

```
#include <stdio.h>
void function(int a, int b, int c) {
       char buffer[16];
       int *ret;

       //now we have the return value, what do we do with it?
       ret = (unsigned int)&buffer+0x18;
       buffer[0] = 0x88; //this is here so buffer is not optimized
out
}
void main() {
     int x;
     x = 0;
     function(1,2,3);
     x = 1;
     printf("%d\n",x);
 }
```

Чтобы выяснить, как манипулировать обратным адресом, взгляните на собранный код для main:

```
0804841f <main>:
 804841f:    55                          push   ebp
 8048420:    89 e5                       mov    ebp,esp
 8048422:    83 e4 f0                    and    esp,0xfffffff0
 8048425:    83 ec 20                    sub    esp,0x20
 8048428:    c7 44 24 1c 00 00 00        mov    DWORD PTR
[esp+0x1c],0x0
 8048430:    c7 44 24 08 03 00 00        mov    DWORD PTR
[esp+0x8],0x3
 8048438:    c7 44 24 04 02 00 00        mov    DWORD PTR
[esp+0x4],0x2
 8048440:    c7 04 24 01 00 00 00        mov    DWORD PTR [esp],0x1
 8048447:    e8 c0 ff ff ff              call   804840c <function>
 804844c:    c7 44 24 1c 01 00 00        mov    DWORD PTR
[esp+0x1c],0x1;x=1
```

```
 8048454:    8b 44 24 1c              mov    eax,DWORD PTR
[esp+0x1c]
 8048458:    89 44 24 04              mov    DWORD PTR
[esp+0x4],eax
 804845c:    c7 04 24 08 85 04 08     mov    DWORD PTR
[esp],0x8048508
 8048463:    e8 88 fe ff ff           call   80482f0 <printf@plt>
 8048468:    c9                       leave
 8048469:    c3                       ret
```

Обычно адрес возврата функции был бы 0x804844C, и, глядя на эту инструкцию, это x=1, которого мы хотим избежать! После этой строки следующая инструкция начинается с 0x8048454.

Теперь есть два варианта изменения обратного адреса. Один из них - использовать указатель на обратный адрес, чтобы изменить его на жестко закодированный 0x8048454. Проблема с этим подходом заключается в том, что адрес - это виртуальный адрес, выбранный компилятором во время сборки, и каждый раз, когда вы его запускаете, он будет одним и тем же, пока вы не перекомпилируете. При перекомпиляции есть шанс, что вы получите новые виртуальные адреса. Вам нужно было бы перекомпилировать, чтобы проверить эту теорию, так что этот подход немного жесткий.

Вместо этого лучший подход - отметить, что команда x=1 имеет длину 8 байт. Это всегда будет согласовано, поэтому более сильный подход - добавить 8 байт к текущему адресу возврата.

ПРИМЕЧАНИЕ. При распечатке сборки gdb часто отключает шестнадцатеричное отображение, поэтому, если вы посмотрите на распечатку, вы увидите только 7 байт в строке x=1. Это просто потому, что она была отключена. Всегда выполняйте математические вычисления с адресами, чтобы убедиться, что у вас правильное количество байтов.

Чтобы пропустить инструкцию x=1, обратный адрес должен быть обновлен путем добавления 8 байт. Добавление этого в код приводит к следующему:

```
#include <stdio.h>
void function(int a, int b, int c) {
char buffer[16];
int *ret;

ret = (unsigned int)buffer+0x18; //get the return value
*ret +=0x8; //increment the return value by 8
buffer[0] = 0x88; //this is here so buffer is not optimized out
}
void main() {
     int x;
     x = 0;
     function(1,2,3);
     x = 1;
     printf("%d\n",x);
 }
```

Выполнение этого кода (с флагом компиляции -fno-stack-protector) должно привести к тому, что программа выведет значение 0. Это указывает на то, что обратный адрес был успешно изменен, и программа пропускает инструкцию x=1. Победа!

Шеллкод

Возможность изменять адреса возврата обеспечивает контроль над выполнением кода, что является мощным средством. Одним из распространенных применений этого является “запуск оболочки”, обеспечивающей возможность выполнения более мощных команд.

Чтобы запустить оболочку, вы должны иметь возможность запускать свой собственный произвольный код внутри приложения. Для этого вам нужно поместить шеллкод в буфер, который переполняется, и изменить адрес возврата, чтобы он указывал на начало этого кода. Шеллкод буквально означает код, который запускает командную строку (shell). Шеллкод может быть введен до или после адреса возврата в зависимости от объема доступного вам буферного пространства. Цель состоит в том, чтобы поместить ваш шеллкод куда-нибудь в буфер, а затем изменить обратный адрес, чтобы он указывал на него.

Следующий код показывает очень простой шелл-код. Он использует системный вызов execve Linux для выполнения /bin/sh, который является обычным приложением оболочки. execve просит ядро Linux что-то сделать. В этом случае при передаче в оболочку приложения Linux запрашивает запуск оболочки.

```
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
      exit(0);
}
```

Этот простой шелл-код собирается в следующий ассемблерный код:

```
0804843c <main>:
 804843c:    55                          push   ebp
 804843d:    89 e5                       mov    ebp,esp
 804843f:    83 e4 f0                    and    esp,0xfffffff0
 8048442:    83 ec 20                    sub    esp,0x20
8048445:    c7 44 24 18 18 85 04     mov    DWORD PTR [esp+0x18],
0x8048518
 804844c:    08
 804844d:    c7 44 24 1c 00 00 00     mov    DWORD PTR
[esp+0x1c],0x0
 8048454:    00
 8048455:    8b 44 24 18              mov    eax,DWORD PTR
[esp+0x18]
 8048459:    c7 44 24 08 00 00 00     mov    DWORD PTR
[esp+0x8],0x0
 8048460:    00
 8048461:    8d 54 24 18              lea    edx,[esp+0x18]
```

```
 8048465:    89 54 24 04               mov    DWORD PTR
[esp+0x4],edx
 8048469:    89 04 24                  mov    DWORD PTR [esp],eax
 804846c:    e8 cf fe ff ff            call   8048340 <execve@plt>
 8048471:    c7 04 24 00 00 00 00      mov    DWORD PTR [esp],0x0
 8048478:    e8 a3 fe ff ff            call   8048320 <exit@plt>
```

Этот код полагается на стандартные методы C для execve и exit, которые будут перемещаться по памяти, что затрудняет прогнозирование их адресов и встраивание их в код. Это означает, что если вы взяли этот ассемблерный код как есть, поместили коды операций в буфер и обновили адрес возврата, чтобы указать на него, то, когда код достигнет команды call execve, скорее всего, произойдет сбой сегмента. Это связано с тем, что адрес, скомпилированный в шеллкод, - это адрес, по которому был загружен execve для этого приложения (0x8048340), но это не универсальный адрес. Вам нужно было бы знать, где загружается execve для целевого приложения (если в нем вообще есть execve). Это делает необходимым поиск альтернативного способа запуска оболочки, который не использует библиотеки C.

Если вы разберете методы execve и exit, вы сможете увидеть базовую реализацию, как показано в следующем примере кода:

```
mov   eax, 0xb
mov   ebx, string_addr
lea   ecx, string_addr
lea   edx, null_string
int   0x80   ;sys call for exec
mov   eax, 0x1
mov   ebx, 0x0
int   0x80   ;sys call for exit
“:/bin/sh”\0
```

Таким образом, это решает часть проблем, и вызовы библиотеки C преобразуются в системные вызовы int 0x80, рассмотренные ранее в книге. Но теперь возникает другая проблема: значения string_addr и null_string неизвестны, поскольку вы не можете предсказать, где они будут загружены в память. Опять же, собранный шеллкод поместил их в это пространство локальной памяти (в этом примере 0x8048518 - это скомпилированный адрес для /bin/sh), но когда шеллкод будет помещен в целевой буфер, эти адреса будут неправильными.

Чтобы заставить шеллкод работать, требуется найти другой способ поиска адреса, который является относительным, а не жестко закодированным. Один из способов узнать это значение - воспользоваться преимуществами обратных адресов в вызовах функций; опять же, примените свои обширные знания о соглашениях о вызовах и стеке! Если вызов функции помещен непосредственно перед строкой, то адрес строки будет находиться в верхней части стека внутри этой функции (поскольку строка находится по адресу возврата функции).

Для начала добавьте несколько заменителей в существующий шеллкод.

```
jmp    ??
pop   esi
mov    [esi+0x8],esi
mov    [esi+0x7],0x0
mov    [esi+0xc],0x0
mov   eax, 0xb
```

```
mov   ebx, esi
lea   ecx, [esi+0x8]
lea   edx, [0xc+esi]
int    0x80
mov   eax, 0x1
mov   ebx, 0
int    0x80
call   ??
.string \"/bin/sh\"
```

В этом примере кода берется исходный шеллкод и добавляются две инструкции спереди и две в конце. Следующим шагом является определение адреса строки, которая находится в конце блока сборки. В идеале начальная инструкция jmp должна переходить к новому вызову внизу.

Затем этот вызов должен вызвать новую строку pop esi. Почему? При использовании вызова (вместо перехода) для возврата к началу кода обратный адрес (следующий адрес после вызова) будет помещен в стек. У нас нет намерения выполнять обычную настройку стека cdecl; это злоупотребление знаниями x86 для совершения непослушных действий.

После обратного вызова pop esi в верхней части стека будет указан обратный адрес, который в данном случае является строкой оболочки. Этот адрес может быть извлечен из стека в esi и использован в предыдущем шелл-коде.

Это звучит потрясающе, но в настоящее время существуют заполнители для перехода и вызова. Чтобы выяснить, куда они будут переходить, мы должны подсчитать наши байты. Здесь мы подсчитываем скомпилированные байты, чтобы определить правильные смещения для jmp и вызова:

```
jmp    0x26                    # 2 bytes
pop   esi                      # 1 byte
mov   [esi+0x8],esi            # 3 bytes
mov   [esi+0x7],0x0            # 4 bytes
mov   [esi+0xc],0x0            # 7 bytes
mov   eax, 0xb                 # 5 bytes
mov   ebx, esi                 # 2 bytes
lea   ecx, [esi+0x8]           # 3 bytes
lea   edx, [0xc+esi]           # 3 bytes
int    0x80                    # 2 bytes
mov   eax, 0x1                 # 5 bytes
mov   ebx, 0                   # 5 bytes
int    0x80                    # 2 bytes
call   -0x2b                   # 5 bytes
.string \"/bin/sh\"
```

Этот модифицированный код решает проблему поиска строки в памяти, делая все это относительным (без жестко закодированных адресов) и используя фундаментальные принципы работы x86, чтобы помочь. Последней задачей является запуск кода, для чего требуется поместить двоичное представление кода в стек посредством переполнения буфера.

Разбиение стека и защита стека

Как упоминалось, по умолчанию многие компиляторы теперь встраивают средства защиты стека для предотвращения элементарных атак стеком. В качестве примера, gcc и g++ после

gcc 4.1 имеют некоторую встроенную защиту стека. Чтобы попрактиковаться в разбиении стека, необходимо создавать исполняемые файлы, используя флаг -fno-stack-protector. Итак, как выглядит защита стека? Давайте создадим пример и посмотрим, что он добавляет.

В следующем примере кода показана программа, созданная с включенной защитой стека:

```
0804845c <function>:
 804845c:    55                          push   ebp
 804845d:    89 e5                       mov    ebp,esp
 804845f:    83 ec 48                    sub    esp,0x48
 8048462:    8b 45 08                    mov    eax,DWORD PTR
[ebp+0x8]
 8048465:    89 45 d4                    mov    DWORD PTR [ebp-
0x2c],eax
 8048468:    65 a1 14 00 00 00           mov    eax,gs:0x14
 804846e:    89 45 f4                    mov    DWORD PTR [ebp-
0xc],eax
 8048471:    31 c0                       xor    eax,eax
 8048473:    8b 45 d4                    mov    eax,DWORD PTR [ebp-
0x2c]
 8048476:    89 44 24 04                 mov    DWORD PTR
[esp+0x4],eax
 804847a:    8d 45 e4                    lea    eax,[ebp-0x1c]
 804847d:    89 04 24                    mov    DWORD PTR [esp],eax
 8048480:    e8 bb fe ff ff              call   8048340 <strcpy@plt>
 8048485:    8b 45 f4                    mov    eax,DWORD PTR [ebp-
0xc]
 8048488:    65 33 05 14 00 00 00        xor    eax,DWORD PTR gs:0x14
 804848f:    74 05                       je     8048496
<function+0x3a>
8048491:    e8 9a fe ff ff              call   8048330 <__stack_chk
_fail@plt>
 8048496:    c9                          leave
 8048497:    c3                          ret
```

Выделенные жирным шрифтом строки иллюстрируют то, что добавлено компилятором для защиты стека. Компилятор добавил код, который сохранит адрес возврата при вводе функции и проверит, что он не изменился после операции strcpy. Компилятор знает, что вызовы типа strcpy могут быть опасны; это предотвращает перезапись обратного адреса strcpy.

Существует несколько вариантов защиты от разбиения стека, включая встроенные средства защиты стека gcc, использование языков, безопасных для памяти, с проверкой границ и предотвращение выполнения данных (DEP). Однако в некоторых случаях переполнение буфера по-прежнему представляет угрозу, поскольку не все компиляторы будут поддерживать защиту стека или DEP, и, как вы можете видеть, есть нюанс в том, как это защищает, а не добавляет защиту стека для каждого отдельного вызова. Тем не менее, средства защиты направлены против конкретных вещей, таких как strcpy. И многие компиляторы довольно умны в определении того, какие из них наиболее опасны и нуждаются в защите.

Соединение C и x86

Любая программа, которая может быть написана на C (или любом другом языке), также может быть написана на ассемблере. Фактически, языки более высокого уровня компилируются в ассемблер перед их запуском процессором. Однако в некоторых случаях

может оказаться полезным смешать C и ассемблерный код. Если вы пишете свои собственные эксплойты /кряки, это мощная комбинация. Некоторые вещи настолько тонки, что вам нужен контроль на уровне ассемблера, а некоторые вещи - это просто код, который должен выполняться, и быстрее писать его на C, так что не стесняйтесь смешивать их!

Чтобы вызвать функцию, написанную на другом языке, необходимо знать, где эта функция находится в памяти. Компоновщик может предоставить эту информацию автоматически.

Также необходимо знать, как передать информацию этой функции, т.е. соглашение о ее вызове. В этом случае мы будем предполагать, что наши функции C используют cdecl. Напомним следующее, с помощью cdecl:

- Аргументы передаются в стеке справа налево.
- Вызывающий отвечает за очистку стека после возврата вызова.
- Возвращаемое функцией значение хранится в eax.
- Вызываемому объекту доступны регистры eax, ecx и edx.

Вызывающий объект должен сохранить значения этих регистров, если это необходимо, а вызываемый объект должен сохранить и восстановить значения любых других регистров, которые ему нужны.
Если вы следуете правильному соглашению о вызове, вы можете вызывать функции C из своего ассемблерного кода.

Использование функций C в коде x86
Для того, чтобы код x86 использовал функции C, ассемблерный код должен знать, что функция C определена в другом месте. Это делается с помощью директивы extern в ассемблерном коде. Например, чтобы вызвать функцию C, x(), в x86, используйте следующие инструкции:

```
extern x
call x
```

Первым шагом к использованию функций C в assembly является включение директивы extern function_name в начало файла assembly. Это сообщает ассемблеру, что вы собираетесь использовать эту функцию, но вы еще не знаете ее местоположение (адрес). Когда вы пишете call function_name в ассемблерном коде, он изначально будет собран как call 0x????????. Однако программа не сможет запуститься, пока вы не пропустите ее через компоновщик, который заполнит соответствующий адрес.

Следующим шагом является вызов нужной функции с использованием соглашения о вызове cdecl. Например, при вызове функции C int add(int x, int y) вы должны использовать следующий ассемблерный код. Помните, что аргументы передаются справа налево, и вам нужно очистить стек после вызова и поместить возвращаемое значение в eax.

```
push [y]
push [x]
call add
add esp, 8
mov [sum], eax
```

После написания ассемблерного кода следующим шагом является его сборка с помощью nasm. Вот пример: nasm example.asm –o example.o.

На этом этапе все будет в ассемблере, за исключением этих заполнителей. Если бы у вас не было внешних функций, ваш код был бы готов к запуску, но поскольку он есть, вам нужна помощь компоновщика. Последний шаг - связать ваш ассемблерный код с функцией C. Если вы используете gcc и вызываете функции из библиотеки C, gcc может обработать это автоматически. Например, gcc example.o -o example будет использовать компоновщик для заполнения любых известных ему адресов, преобразуя вызов 0x???????? для вызова 0x08048320.

Для примера рассмотрим следующий пример, в котором выполняется printf hello world 42:

```
extern printf
global main

section .text
main:

push 42
push world
push hello
call printf
add esp, 12

mov eax, 1
mov ebx, 0
int 0x80

section .data
hello: db "hello %s %d", 0xa, 0
world: db "world"
```

Этот ассемблерный код может быть собран с использованием nasm –f elf example.asm и связан с gcc –m32 example.o –o example.

Может быть очень полезно и мощно иметь возможность вызывать простые вещи, такие как printf, из вашего ассемблерного кода, пока вы тестируете свои идеи взлома / исправления.

Использование функций x86 в коде C

Также возможно вызывать функции ассемблера из кода C. Программа на C должна иметь прототип для функций x86, которые она хочет использовать. Например, в C, чтобы использовать ассемблерную функцию f, вам нужен прототип int f(void);. Прототип - это причудливый способ сказать, что вам нужно объявить, как выглядело бы это определение функции, если бы оно было на языке более высокого уровня (как оно называется, какие аргументы оно принимает и что оно возвращает).

Чтобы использовать функции x86 в вашем коде C, их необходимо экспортировать из вашего ассемблерного кода, чтобы компоновщик мог их найти. Чтобы экспортировать функцию x86 в ваш файл сборки, пометьте ее глобальной директивой, как показано в следующем примере:

```
global f
f:
  mov eax, 0xdabbad00
  ret
```

Затем соберите свой ассемблерный код с помощью nasm, скомпилируйте и свяжите всю программу с помощью gcc.

Для примера рассмотрим следующую программу на C:

```
// x.c

#include <stdio.h>

int add(int,int);

int main(void)
{
    int x=add(1,2);
    printf("%d\n",x);
    return 0;
}
```

Эта программа использует функцию add, которая определена в следующем ассемблерном коде:

```
; y.asm

add:
  push ebp
  mov ebp, esp

  mov eax, [ebp+8]
  add eax, [ebp+12]

  leave
  ret
```

Чтобы связать и собрать эту программу, выполните следующие команды:

```
nasm -f elf y.asm # produces y.o object
gcc -m32 -c x.c # produces x.o object
gcc x.o y.o -o adder # produces executable adder
# run with ./adder
```

_start против main()
программы на ассемблере x86 обычно начинаются с метки с именем _start. Программы на C, с другой стороны, начинаются с функции main(). В чем разница?

Выполнение программы (независимо от того, написана ли она на C, ассемблере или любом другом языке) на самом деле начинается не с main. Для примера рассмотрим простейшую возможную функцию C, как показано здесь:

```
int main()
{
     return 0;
```

```
}
```

Компиляция этого с помощью gcc simple.c –o simple переводит вашу программу в ассемблер. В рамках этого процесса компилятор добавляет функцию с именем _start, а _start вызывает main.

Результирующая скомпилированная функция main имеет следующую сборку:

```
80483b4:     55                    push   ebp
 80483b5:    89 e5                 mov    ebp,esp
 80483b7:    5d                    pop    ebp
 80483b8:    c3                    ret
```

Функция запуска выглядит следующим образом:

```
8048300:     31 ed                 xor    ebp,ebp
 8048302:    5e                    pop    esi
 8048303:    89 e1                 mov    ecx,esp
 8048305:    83 e4 f0              and    esp,0xfffffff0
 8048308:    50                    push   eax
 8048309:    54                    push   esp
 804830a:    52                    push   edx
 804830b:    68 30 84 04 08        push   0x8048430
 8048310:    68 c0 83 04 08        push   0x80483c0
 8048315:    51                    push   ecx
 8048316:    56                    push   esi
 8048317:    68 b4 83 04 08        push   0x80483b4
 804831c:    e8 cf ff ff ff        call   80482f0
<__libc_start_main@plt>
 8048321:    f4                    hlt
```

Функция start отвечает за несколько различных задач, включая следующие:

- Инициализация указателя фрейма
- Настройка стека
- Настройка стандартных аргументов (параметров для main())
- Вызов libc_start_main, который выполняет проверки безопасности, потоковой подсистемы, init, вызывает вашу основную функцию и, наконец, вызывает exit()

При написании чистого ассемблерного кода вы пишете все сами. Вам не нужны все настройки, которые выполняет C, и вы можете написать свою собственную функцию _start.

При объединении assembly и C вам нужен gcc для участия. Часто gcc хочет предоставить свою собственную функцию _start и ожидает, что вы предоставите функцию main().

При написании программы на ассемблере, которая будет связана со стандартной библиотекой C, выполните следующее:

1. Используйте main вместо _start (libc_start_main вызовет main() для вас).
2. Настройте только фрейм стека, а не весь стек (_start уже настроил ваш стек).
3. Завершите с помощью ret, а не int 0x80 (ret вернется к libc_start_main, который вызовет функцию C exit, которая вызовет int 0x80 для вас).
4. Установите возвращаемое значение в eax перед повторным вводом (обычно 0).
Для примера рассмотрим следующую автономную программу сборки, которая определяет свой собственный _start:

```
global _start

section .text
_start:
    mov  esp, stack
    mov  ebp, esp

    …

    mov  esp, ebp

    mov  eax, 1
    mov  ebx, 0
    int  0x80

section .data
times 128 db 0
stack equ $-4
```

При подключении к libc программа должна вместо этого использовать main.

```
global main

section .text
main:
    push ebp
    mov  ebp, esp

    …

    mov  eax, 0
    leave
    ret
```

Стандартные аргументы
В C аргументы могут быть считаны из командной ссылки со стандартами. Например, main() обычно определяется как int main(int argc, char **argv), что обеспечивает доступ к этим аргументам командной строки. Напомним, что argc - это количество переданных аргументов, а argv - массив, содержащий эти аргументы.

Также возможно получить доступ к аргументам командной строки при написании функции main в assembly. Ваша версия main для сборки будет автоматически вызвана с помощью cdecl.

Напомним, что следующее:

- Аргументы передаются в стек, перемещаясь справа налево.
- Аргументы находятся в [ebp+8], [ebp+12] и т.д.
- argc будет последним аргументом и находится в верхней части списка аргументов в стеке, в [ebp+8].
- argv является первым аргументом, помещенным в стек, и будет иметь значение [ebp+12].

Например, следующая программа на ассемблере напечатает первый аргумент командной строки:

```
extern printf
global main

main:
   push  ebp
   mov   ebp, esp

   mov   eax, [ebp+12]  ; load argv into eax
   push  dword [eax+4]  ; push argv[1]
   call  printf         ; print argv[1]
   add   esp, 4         ; clean up stack
   mov   eax, 0
   leave
   ret
```

Смешивание C и Assembly

В C можно легко переключаться между C и ассемблерным кодом. Это называется встроенной сборкой, названной так из-за того, что сборка встроена в ваш исходный код.

Встроенная сборка не является частью спецификации C, но большинство компиляторов будут поддерживать ее через расширение. Однако синтаксис уникален для каждого компилятора. В gcc это синтаксис AT&T x86.

Базовая форма этого - __asm___(“Здесь код ассемблера”);. При компиляции gcc компилирует код C в assembly и вставляет код ассемблера из директивы __asm__.

Например, рассмотрим следующую программу на языке Си:

```
int main(void)
{
    // set keyboard control register

    __asm___ ("mov  $0x10010001, %eax");
    __asm___ ("out  %eax, $0x64");

    return 0;
}
```

Расширенная форма встроенной сборки позволяет вам устанавливать расширенные “ограничения”. Эти ограничения могут включать следующее:

- Входные переменные: переменные C, которыми вы хотите управлять с помощью assembly.
- Выходные переменные: Значения, полученные во встроенном коде assembly, которые вы хотите использовать в коде C.
- Перегруженные регистры: gcc преобразует C в assembly и определяет, какие регистры использовать. Этот список гарантирует, что регистры, используемые кодом C и assembly, не будут конфликтовать.
- Расширенная сборка может быть указана следующим образом:

```
__asm__(
             "assembly"
             : input constraints
             : output constraints
             : clobber list
             );
```

В следующем примере кода показан пример использования расширенной сборки на C:

```
int main(void)
{
    // getting the return address for the current function

    int x;

    __asm__("\
            movl 0x4(%%ebp), %%eax  \n\
            movl %%eax, %0          \n\
            "
            :"=r"(x)
            :
            :"%eax"
           );


    printf("%08x\n", x);

    return 0;
}
```

Встроенная сборка широко используется в C для следующих целей:

- Ядро операционной системы (ознакомьтесь с исходным кодом ядра Linux).
- Встроенные системы.
- Любой код, который должен работать с аппаратным обеспечением.
- Любой код, который должен быть очень быстрым.
- Вы будете видеть это время от времени, если когда-нибудь будете работать с C, и вам, возможно, придется использовать это самостоятельно.

Помните, что при использовании встроенной сборки вам нужно будет добавить новый флаг в gcc. Например, команда gcc –masm=intel myFile.c сообщает gcc, что вы записали некоторую сборку Intel в свой файл C.

Резюме
В этой главе продемонстрировано, как использовать понимание x86 и стека для взлома. Разбивая стек и вставляя шелл-код, реверсор может обманом заставить программу запустить код злоумышленника.

Вывод
Вау, это было настоящее путешествие! Мы рассмотрели от нападения до защиты; языки высокого уровня вплоть до ассемблера; регистры, поток управления, реверс-инжиниринг; исправления, инструменты, техники и мышление. Если вы зашли так далеко, у вас есть потрясающий багаж знаний, на основе которого вы можете продолжать двигаться вперед.

И по мере того, как вы продвигаетесь вперед, вы всегда будете сталкиваться с чем-то новым. Сначала это будут инструкции по сборке, которых вы не знаете, затем средства защиты, которых вы никогда не видели, затем архитектуры, о которых вы никогда не слышали, и, конечно, новейший, величайший инструмент дня или защита года. Но теперь, когда у вас есть основы, вы обнаружите, что быстро осваивать новые вещи становится все проще.

Теперь, когда вы знаете mov, вы можете легко разобраться в строковой версии movs. Вы работали с битовыми манипуляциями, такими как not, поэтому отрицание с помощью neg довольно быстро обретает смысл. Вы освоили такие сравнения, как cmp, так что cmps - это не большая натяжка, а дальше как насчет cmpxchg или cmpxchg16b или блокировки cmpxchg8b? Суть такова: теперь, когда у вас есть основы, становится все легче понимать новые инструкции; будь то ud (неопределенная инструкция) или gf2p8affineinvqb (обратное аффинное преобразование поля Галуа), основы, как правило, в основном одинаковы для всего.

Но, конечно, на этом обучение не заканчивается. Новые инструкции - это здорово, но если вы будете продолжать в том же духе, то вскоре столкнетесь с совершенно новыми архитектурами. Хорошей новостью является то, что они также, как правило, следуют одним и тем же базовым шаблонам, и теперь, когда вы освоили один из них, вы сможете быстро понять новые. x64 (64-разрядный x86) теперь, когда вы освоили x86, это просто — просто расширьте регистры до 64 бит (rax вместо eax, rsp вместо esp) и следуйте некоторым другим соглашениям о вызовах (AMD64 ABI в дополнение к cdecl), и вы сможете применять все те же инструменты и методы к 64-разрядному коду. Оттуда Arm приходит довольно легко — опять же, это просто новые регистры (r0 вместо rax), инструкции (b вместо jmp) и соглашения о вызовах (Arm вместо cdecl). Базовые шаблоны, как правило, в основном одинаковы, поэтому, какой бы ни была ваша цель — PowerPC, MIPS, RISC-V, MIL-STD-1750A и т.д., — вы обычно можете освоить основы за несколько часов. Переход на новые архитектуры также позволит вам применить свои навыки к новым устройствам. Будь то телефоны, маршрутизаторы, автомобили или спутники, основы довольно единообразны.

Естественно, по мере продвижения вы столкнетесь не только с новыми архитектурами, но и с новыми инструментами. Хорошей новостью здесь также является то, что они, как правило, основаны на одном и том же базовом наборе концепций. Мы проработали множество дизассемблеров, шестнадцатеричных редакторов, отладчиков и декомпиляторов. Теперь пришло время начать изучать новые опции, чтобы посмотреть, что вам подходит. Ghidra,

Binary Ninja и Cutter/radare2 - это популярные следующие шаги, которые будут основываться на вашем опыте работы с IDA и предложат еще больше способов анализа и понимания программы. По мере расширения вашего арсенала инструментов вы постепенно будете создавать свои собственные сценарии, рабочие процессы и стратегии, чтобы становиться все более опытными в работе со все более сложными целями.

И, конечно, если вы будете продолжать в том же духе, вы начнете сталкиваться с новыми средствами защиты. Будь то новейший античит в онлайн-играх, новая система запутывания непрозрачных предикатов от академических кругов или креативное новое хэширование в эзотерическом кейчекере, следование последним тенденциям поможет вам оставаться начеку, независимо от того, является ли ваша страсть нападением или защитой. Фантастическими ресурсами здесь могут быть как академические журналы, так и форумы по взлому.

Но каковы бы ни были ваши конечные цели при использовании этого набора навыков, единственным ключом к продвижению вперед является практика. Попробуйте написать свой собственный кейчекер, а затем посмотрите, сможете ли вы его взломать — игра за обе стороны одновременно может дать интересное представление о проблемах и ограничениях противника. Crackme предлагают фантастический, увлекательный и (в основном) безопасный способ получить опыт в реверс-инжиниринге и модификации программного обеспечения на самых разных языках и архитектурах. Как только у вас появится несколько минут, возьмите crackme, который соответствует вашему опыту и уровню квалификации, и посмотрите, сможете ли вы победить его; если у вас есть несколько часов, найдите программу, использующую язык, которого вы не знаете, или средства защиты, которых вы никогда не видели. Помимо взлома, модификация простых программ может быстро предложить новые идеи и расширить ваш набор навыков. Загрузите свою любимую видеоигру 90-х годов в IDA и посмотрите, сможете ли вы добавить бесконечные жизни; попробуйте Ghidra в вашем любимом текстовом редакторе и посмотрите, сможете ли вы добавить секретное меню. Кроме того, соревнования по захвату флага могут стать захватывающим способом довести свои навыки реверс-инжиниринга до предела, одновременно осваивая новые области, такие как использование двоичных файлов и компьютерная криминалистика.

Как бы вы ни продвигались, оставайтесь настойчивыми, продолжайте практиковаться и расширяйте свои возможности в новых областях. Мы надеемся, что эта книга помогла вам приобрести широкий набор навыков и что вы будете использовать их, чтобы еще глубже погрузиться в этот удивительный аспект безопасности.