Memory Dump Analysis Anthology

Volume 2

Dmitry Vostokov

OpenTask

Published by OpenTask, Republic of Ireland

OpenTask books are available through booksellers and distributors worldwide. For further information or comments send requests to press@opentask.com.

A CIP catalogue record for this book is available from the British Library.

ISBN-13: 978-0-9558328-7-1 (Paperback)

ISBN-13: 978-1-906717-22-3 (Hardback)

First printing, 2008

Revision 1 (October, 2008)

Exception “is what we see at a glance”.

Blaise Pascal

Crash dump analysis “does not consist merely in” **peeking** “the memory and enlightening the understanding. Its main business should be to direct the” **Customer**.

Joseph Joubert

SUMMARY OF CONTENTS

CONTENTS

PREFACE

This is a revised, edited, cross-referenced and thematically organized volume of selected DumpAnalysis.org blog posts written in January - September 2008. It is intended to be used as a reference and will be cited in my future books.

I hope these articles will be useful for:

- Software engineers developing and maintaining products on Windows platforms.
- Technical support and escalation engineers dealing with complex software issues.
- Quality assurance engineers testing software on Windows platforms.
- Some articles will be of interest to a general Windows user.

If you encounter any error please contact me using this form

http://www.dumpanalysis.org/contact

or send me a personal message using this contact e-mail:

dmitry.vostokov@dumpanalysis.org

ACKNOWLEDGEMENTS

Thousands of people reviewed DumpAnalysis.org blog content and I would like to thank all of them including the following individuals for providing their comments, suggestions and encouragement during the period of January - September 2008 (I apologize if I missed someone):

Dave Agans
Gene Allen
Francisco Alves
Wanderley Caloni
Paul Carrington
Erdenebat Chuluun
Jeff Curless
Jeff Dailey
Shantanu Deo
Soren Dreijer
Volker von Einem
Michelle Griffin
Da-Chang Guan
Roy Gunter
Roberto Farah
Mario Hewardt
Yanic Inghelbrecht
Bill Joy
Koby Kahane
Aleksei Kazbayev
Heejune Kim
Ray Kinsella
Martin Kulov
Taehwa Lee
Chae Lim
Ramzy Mansour
Thomas Monahan
Mikhail Naganov
Kapildev Ramlal
Jerome Reid
Deepesh Reja
Carlos Pizano
Brad Rutkowski
Jói Sigurðsson
Marc Sherman
Jeffrey Tan
Alan Tang
Konstantin Tchebotarev
Suresh Thotakura
Serhat Toktamisoglu
Kobi Ben Tzvi
Brian Quinn
Nicholas Vasile
Narasimha Vedala

Thanks to Tony Donegan, for the front cover design.

PART 1: CRASH DUMPS FOR BEGINNERS

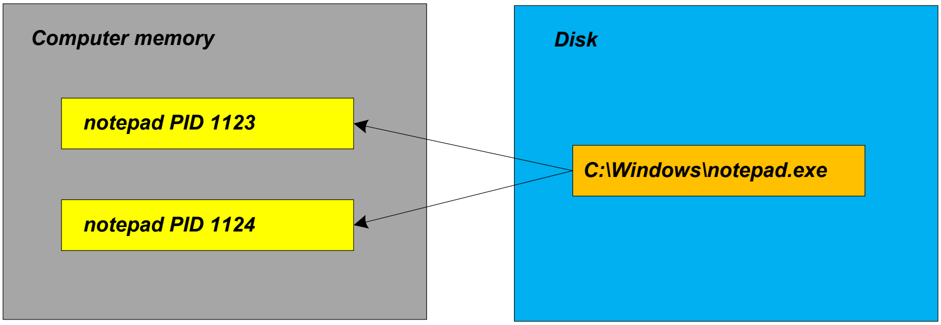
THE TIME OF THE CRASH

When we have a crash dump WinDbg tells us the time of the crash and the time interval since the start of OS:

```
1: kd> vertarget
Windows Vista Kernel Version 6000 MP (2 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 6000.16575.amd64fre.vista_gdr.071009-1548
Kernel base = 0xfffff800`01c00000 PsLoadedModuleList = 0xfffff800`01d9af70
Debug session time: Tue Jan 29 11:03:52.572 2008 (GMT+0)
System Uptime: 0 days 0:12:06.648
```

STACK TRACE

Here I uncover the mystery of the following phrase used so often in analysis reports: **“stack trace of the problem thread“**. First, what is a thread actually? It is defined as a unit of execution or processing. Imagine an Internet browser running on our computer. It was launched by clicking on browser.exe file which we call an application executable file. When this file is loaded and running it occupies some memory regions and we call all these memory regions an Internet browser **process**. Let click on browser.exe again. A new instance is launched and again, a new running instance is also called a process, the second one. We see that one application file corresponds to several simultaneously running processes where each one has its own unique number, process id (PID). This is illustrated on the following picture for the case of notepad.exe:



Some processes do several tasks in parallel. We call them **threads** of execution and each one is assigned a unique number, thread id (TID). Consider an Internet browser again. TID 112 is processing keyboard / mouse events (scrolling the page), TID 114 is downloading a graphics file and TID 212 is drawing an animated GIF picture. Every thread does its task in steps, splitting its steps into sub-tasks, dividing every sub-tasks into even smaller units and so on. For example downloading a graphics file can be split into locating a file name on an HTML page, establishing a TCP/IP connection, receiving data and then closing down that connection. The sub-task of locating a file name on an HTML page can be subdivided into finding an IMG tag, then finding its SRC attribute, then parsing a file name, an so on. Abstractly this can be depicted as the nested hierarchy of sequential actions (called functions):

- A
    - a …
    - b …
    - c …
- B
    - d
        - α …
        - β …
        - γ …
    - e
- C
    - f …
    - g …

Suppose during the step γ we have a problem. At that time a crash dump is saved. Then in the crash dump we would see the sequence B -> d -> γ. It is called a **stack trace** (or a **backtrace**) of the problem thread that was supposed to successfully finish the sequence A -> B -> C but was interrupted at the sub-sub-step γ. In the debugger this stack trace would be shown as:

```
application!γ
application!d
application!B
```

The most recent action is on the top. Let’s go back to our Internet browser example and the thread locating a file name on an HTML page. Suppose the code for parsing file names contains a defect and Internet browser crashes. Then the crash dump would reveal the following stack trace:

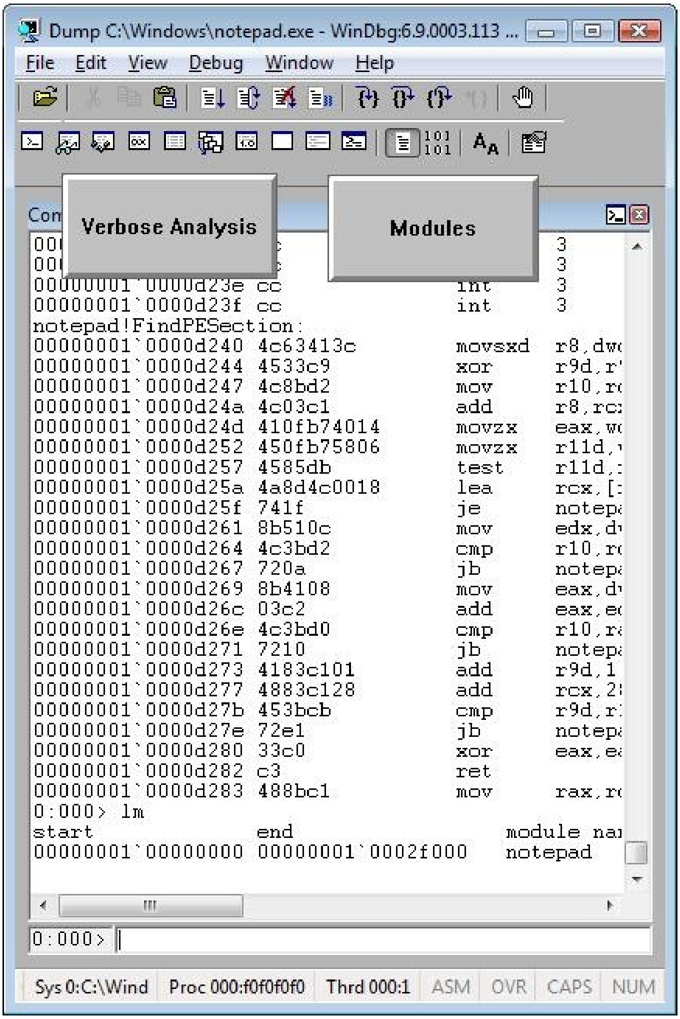
```
browser!ParseFileName
browser!FindSRCAttrubute
browser!FindIMGTag
browser!LocateFileName
browser!DownloadFile
```

It is often possible to interpret stack traces and guess their meanings by reading the names of corresponding actions.

EASYDBG

This is a high-level interface to WinDbg (can be any GUI debugger actually). The basic idea revolves around floating buttons (list box and task bar icons, optionally) that dynamically change with every new window or application. The number of buttons can be unlimited, they have tooltips and can be repositioned to any corner of the screen, they can play sounds, show video and pictures. On click they execute elaborated macro commands, including keystrokes and mouse movements, written in a special scripting language. For example, we can create buttons for CDA checklist (Volume 1, page 251).

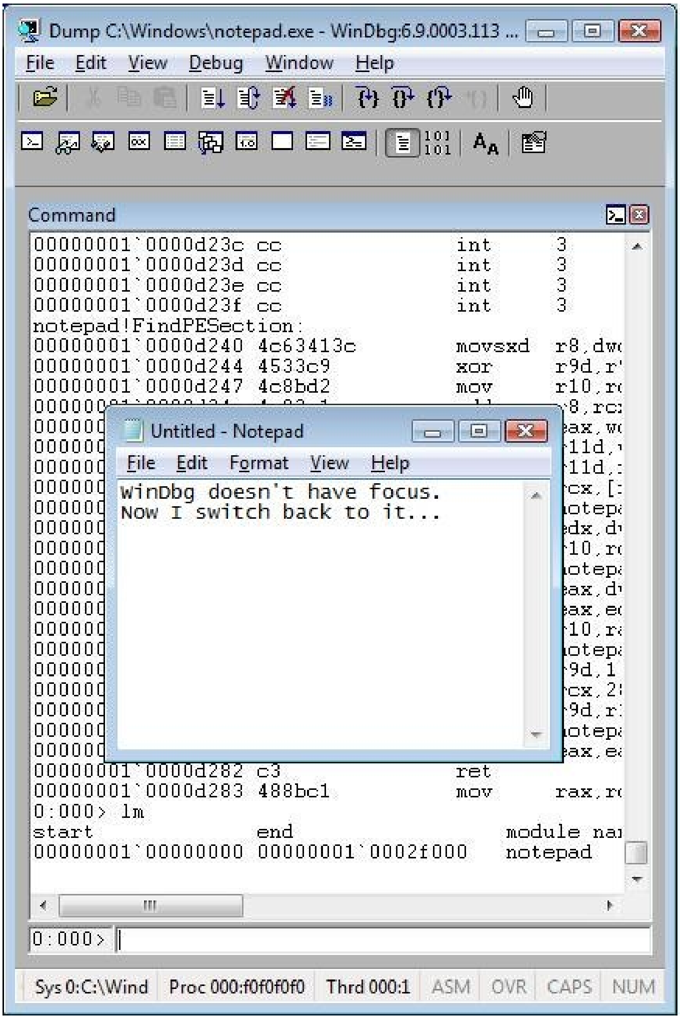
For example, we create 2 buttons for WinDbg window:


Dump C:\Windows\notepad.exe - WinDbg:6.9.0003.113 ...
File Edit View Debug Window Help
Verbose Analysis
Modules
Com
Sys 0:C:\Wind Proc 000:f0f0f0f0 Thrd 000:1 ASM OVR CAPS NUM

```
00(                                        3
00(                                        3
00000001`0000d23e cc              int     3
00000001`0000d23f cc              int     3
notepad!FindPESection:
00000001`0000d240 4c63413c        movsxd  r8,dw
00000001`0000d244 4533c9          xor     r9d,r'
00000001`0000d247 4c8bd2          mov     r10,r
00000001`0000d24a 4c03c1          add     r8,rc
00000001`0000d24d 410fb74014      movzx   eax,w
00000001`0000d252 450fb75806      movzx   r11d,
00000001`0000d257 4585db          test    r11d,
00000001`0000d25a 4a8d4c0018      lea     rcx,[
00000001`0000d25f 741f            je      notepa
00000001`0000d261 8b510c          mov     edx,d
00000001`0000d264 4c3bd2          cmp     r10,r
00000001`0000d267 720a            jb      notepa
00000001`0000d269 8b4108          mov     eax,d
00000001`0000d26c 03c2            add     eax,e
00000001`0000d26e 4c3bd0          cmp     r10,r
00000001`0000d271 7210            jb      notepa
00000001`0000d273 4183c101        add     r9d,1
00000001`0000d277 4883c128        add     rcx,2
00000001`0000d27b 453bcb          cmp     r9d,r
00000001`0000d27e 72e1            jb      notepa
00000001`0000d280 33c0            xor     eax,e
00000001`0000d282 c3              ret
00000001`0000d283 488bc1          mov     rax,r
0:000> lm
start             end                   module na
00000001`00000000 00000001`0002f000     notepad
```

```
0:000>
```

When we switch from WinDbg to another application they disappear:

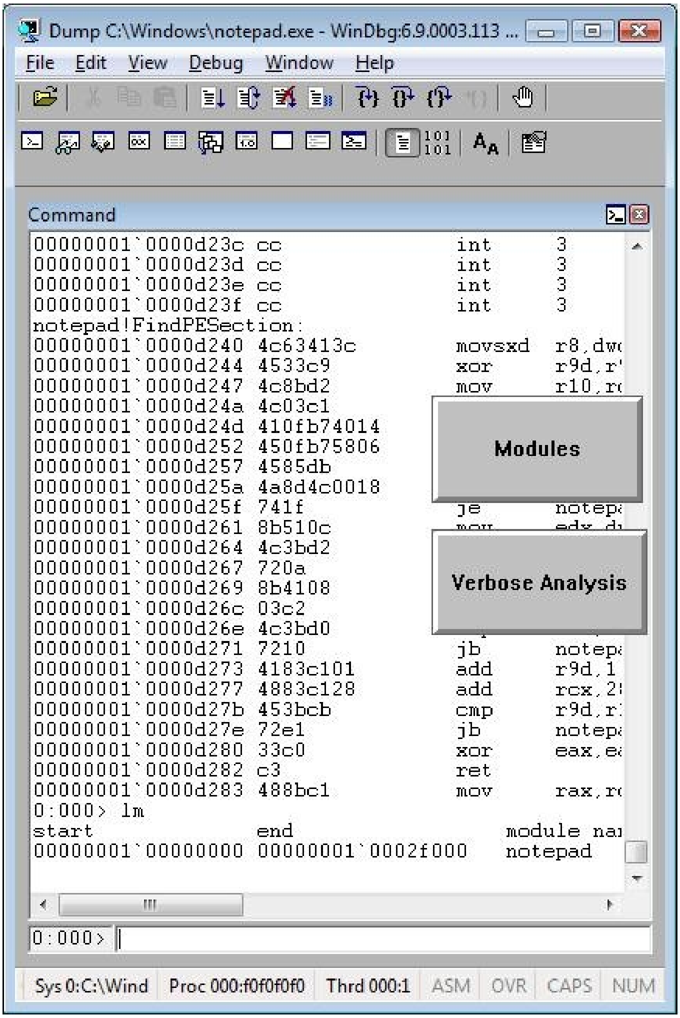

Dump C:\Windows\notepad.exe - WinDbg:6.9.0003.113 ...
File Edit View Debug Window Help
Command
Untitled - Notepad
File Edit Format View Help
Sys 0:C:\Wind
Proc 000:f0f0f0f0
Thrd 000:1
ASM
OVR
CAPS
NUM

```
00000001`0000d23c cc               int     3
00000001`0000d23d cc               int     3
00000001`0000d23e cc               int     3
00000001`0000d23f cc               int     3
notepad!FindPESection:
00000001`0000d240 4c63413c         movsxd  r8,dw
00000001`0000d244 4533c9           xor     r9d,r'
00000001`0000d247 4c8bd2           mov     r10,r
0000000                                    8,rc
000000                                     ax,w
000000                                     11d,
000000                                     11d,
000000                                     cx,[
000000                                     otep
000000                                     dx,d
000000                                     10,r
000000                                     otep
000000                                     ax,d
000000                                     ax,e
000000                                     10,r
000000                                     otep
000000                                     9d,1
000000                                     cx,2
000000                                     9d,r
000000                                     otep
000000                                     ax,e
00000001`0000d282 c3               ret
00000001`0000d283 488bc1           mov     rax,r
0:000> lm
start             end                 module na
00000001`00000000 00000001`0002f000   notepad
```

```
WinDbg doesn't have focus.
Now I switch back to it...
```

```
0:000>
```

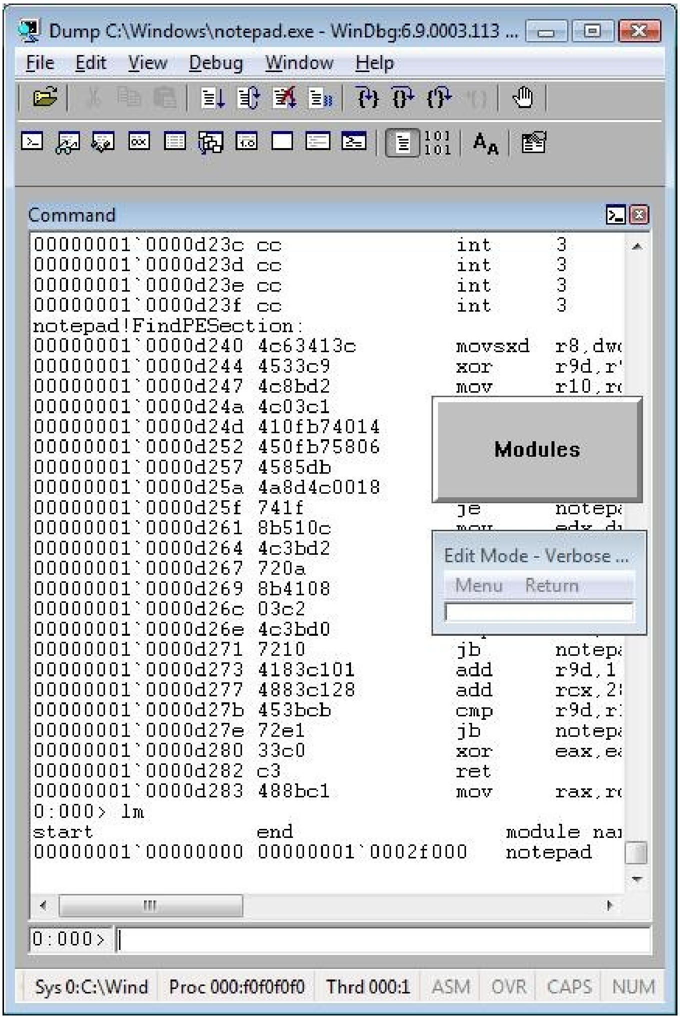
We switch back to WinDbg and they reappear. We can move them around the screen:


Dump C:\Windows\notepad.exe - WinDbg:6.9.0003.113 ...
File Edit View Debug Window Help
Command
Modules
Verbose Analysis
Sys 0:C:\Wind
Proc 000:f0f0f0f0
Thrd 000:1
ASM
OVR
CAPS
NUM

```
00000001`0000d23c cc                    int     3
00000001`0000d23d cc                    int     3
00000001`0000d23e cc                    int     3
00000001`0000d23f cc                    int     3
notepad!FindPESection:
00000001`0000d240 4c63413c              movsxd  r8,dw
00000001`0000d244 4533c9                xor     r9d,r'
00000001`0000d247 4c8bd2                mov     r10,r
00000001`0000d24a 4c03c1
00000001`0000d24d 410fb74014
00000001`0000d252 450fb75806
00000001`0000d257 4585db
00000001`0000d25a 4a8d4c0018
00000001`0000d25f 741f                  je      notepa
00000001`0000d261 8b510c
00000001`0000d264 4c3bd2
00000001`0000d267 720a
00000001`0000d269 8b4108
00000001`0000d26c 03c2
00000001`0000d26e 4c3bd0
00000001`0000d271 7210                  jb      notepa
00000001`0000d273 4183c101              add     r9d,1
00000001`0000d277 4883c128              add     rcx,2
00000001`0000d27b 453bcb                cmp     r9d,r
00000001`0000d27e 72e1                  jb      notepa
00000001`0000d280 33c0                  xor     eax,e
00000001`0000d282 c3                    ret
00000001`0000d283 488bc1                mov     rax,r
0:000> lm
start             end                   module na
00000001`00000000 00000001`0002f000     notepad
```

```
0:000>
```

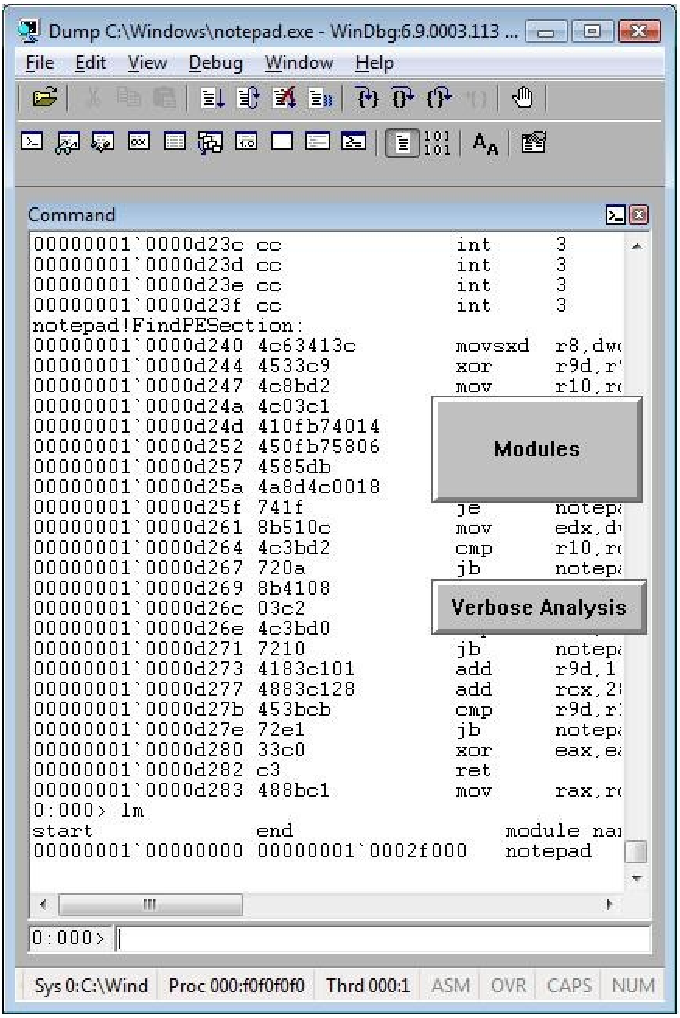
We can edit them by right click:



```
00000001`0000d23c cc              int     3
00000001`0000d23d cc              int     3
00000001`0000d23e cc              int     3
00000001`0000d23f cc              int     3
notepad!FindPESection:
00000001`0000d240 4c63413c        movsxd  r8,dw
00000001`0000d244 4533c9          xor     r9d,r'
00000001`0000d247 4c8bd2          mov     r10,r
00000001`0000d24a 4c03c1
00000001`0000d24d 410fb74014
00000001`0000d252 450fb75806
00000001`0000d257 4585db
00000001`0000d25a 4a8d4c0018
00000001`0000d25f 741f            je      notepa
00000001`0000d261 8b510c
00000001`0000d264 4c3bd2
00000001`0000d267 720a
00000001`0000d269 8b4108
00000001`0000d26c 03c2
00000001`0000d26e 4c3bd0
00000001`0000d271 7210            jb      notepa
00000001`0000d273 4183c101        add     r9d,1
00000001`0000d277 4883c128        add     rcx,2
00000001`0000d27b 453bcb          cmp     r9d,r
00000001`0000d27e 72e1            jb      notepa
00000001`0000d280 33c0            xor     eax,e
00000001`0000d282 c3              ret
00000001`0000d283 488bc1          mov     rax,r
0:000> lm
start             end                 module na
00000001`00000000 00000001`0002f000   notepad
```

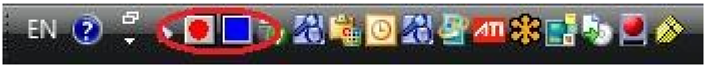
```
0:000>
```

and change their shape:


Dump C:\Windows\notepad.exe - WinDbg:6.9.0003.113 ...
File Edit View Debug Window Help
Command
Modules
Verbose Analysis
Sys 0:C:\Wind
Proc 000:f0f0f0f0
Thrd 000:1
ASM
OVR
CAPS
NUM

```
00000001`0000d23c cc              int     3
00000001`0000d23d cc              int     3
00000001`0000d23e cc              int     3
00000001`0000d23f cc              int     3
notepad!FindPESection:
00000001`0000d240 4c63413c        movsxd  r8,dw
00000001`0000d244 4533c9          xor     r9d,r
00000001`0000d247 4c8bd2          mov     r10,r
00000001`0000d24a 4c03c1
00000001`0000d24d 410fb74014
00000001`0000d252 450fb75806
00000001`0000d257 4585db
00000001`0000d25a 4a8d4c0018
00000001`0000d25f 741f            je      notep
00000001`0000d261 8b510c          mov     edx,d
00000001`0000d264 4c3bd2          cmp     r10,r
00000001`0000d267 720a            jb      notep
00000001`0000d269 8b4108
00000001`0000d26c 03c2
00000001`0000d26e 4c3bd0
00000001`0000d271 7210            jb      notep
00000001`0000d273 4183c101        add     r9d,1
00000001`0000d277 4883c128        add     rcx,2
00000001`0000d27b 453bcb          cmp     r9d,r
00000001`0000d27e 72e1            jb      notep
00000001`0000d280 33c0            xor     eax,e
00000001`0000d282 c3              ret
00000001`0000d283 488bc1          mov     rax,r
0:000> lm
start             end                   module na
00000001`00000000 00000001`0002f000     notepad

0:000>
```

The set of buttons can be saved as an executable file. When we run it on another PC it recreates all buttons when WinDbg window appears. Written in C and using only Win32 API EasyDbg process consumes minimum resources. It sits on task bar for easy access:



There is a dedicated website for this future product: www.easydbg.com

CITRIX SYMBOL SERVER

It is **srv*<*your local folder*>*http://ctxsym.citrix.com/symbols**

Please refer to the following article for details:

How to Use the Citrix Symbol Server to Obtain Debug Symbols
http://support.citrix.com/article/CTX118622

PART 2: PROFESSIONAL CRASH DUMP ANALYSIS

WINDBG SCRIPTS

INTRODUCTION FOR C/C++ USERS

All debuggers from Debugging Tools for Windows package use the same engine dbgeng.dll. It contains a script interpreter for a special language we call WinDbg scripting language for convenience and we use WDS file extension for WinDbg script files. Below is the call stack of a WinDbg thread caught while parsing one of the scripts from this chapter:

```
0:000> ~1kL 100
ChildEBP RetAddr
037cd084 6dd28cdc dbgeng!TypedData::ForceU64+0x3
037cd0ec 6dcbd08c dbgeng!GetPseudoOrRegVal+0x11c
037cd134 6dcbceff dbgeng!MasmEvalExpression::GetTerm+0x12c
037cd198 6dcbca23 dbgeng!MasmEvalExpression::GetMterm+0x36f
037cd1d4 6dcbc873 dbgeng!MasmEvalExpression::GetAterm+0x13
037cd220 6dcbc783 dbgeng!MasmEvalExpression::GetShiftTerm+0x13
037cd254 6dcbc523 dbgeng!MasmEvalExpression::GetLterm+0x13
037cd2c0 6dcbc443 dbgeng!MasmEvalExpression::GetLRterm+0x13
037cd2f4 6dcbc424 dbgeng!MasmEvalExpression::StartExpr+0x13
037cd308 6dcbbc2f dbgeng!MasmEvalExpression::GetCommonExpression+0xc4
037cd31c 6dccdca3 dbgeng!MasmEvalExpression::Evaluate+0x4f
037cd390 6dccd83d dbgeng!EvalExpression::EvalNum+0x63
037cd3d0 6dd293cc dbgeng!GetExpression+0x5d
037cd458 6dd2a7e2 dbgeng!ScanRegVal+0xfc
037cd4ec 6dd17502 dbgeng!ParseRegCmd+0x422
037cd52c 6dd194e8 dbgeng!WrapParseRegCmd+0x92
037cd608 6dc8ed19 dbgeng!ProcessCommands+0x1278
037cd644 6dc962af dbgeng!DotFor+0x1d9
037cd658 6dd1872e dbgeng!DotCommand+0x3f
037cd738 6dd19b49 dbgeng!ProcessCommands+0x4be
037cd77c 6dc5c879 dbgeng!ProcessCommandsAndCatch+0x49
037cdc14 6dd19cc3 dbgeng!Execute+0x2b9
037cdc64 6dc89db0 dbgeng!ProcessCurBraceBlock+0xa3
037cdc74 6dc962af dbgeng!DotBlock+0x10
037cdc88 6dd1872e dbgeng!DotCommand+0x3f
037cdd68 6dd19b49 dbgeng!ProcessCommands+0x4be
037cddac 6dc5c879 dbgeng!ProcessCommandsAndCatch+0x49
037ce244 6dd173ca dbgeng!Execute+0x2b9
037ce2c4 6dd1863c dbgeng!ParseDollar+0x29a
037ce3a0 6dd19b49 dbgeng!ProcessCommands+0x3cc
037ce3e4 6dc5c879 dbgeng!ProcessCommandsAndCatch+0x49
037ce87c 6dc5cada dbgeng!Execute+0x2b9
037ce8ac 00318693 dbgeng!DebugClient::ExecuteWide+0x6a
037ce954 00318b83 windbg!ProcessCommand+0x143
```

```
037cf968 0031ae46 windbg!ProcessEngineCommands+0xa3
037cf97c 76fa19f1 windbg!EngineLoop+0x366
037cf988 77c8d109 kernel32!BaseThreadInitThunk+0xe
037cf9c8 00000000 ntdll!_RtlUserThreadStart+0x23
```

Here I assume that you already know C or C++ language or any C-style language like Java or C#. Therefore I omit explanation for language elements that appear to have similar syntax and semantics when we look and compare equivalent C/C++ and WinDbg script code.

HELLO WORLD

Let's write our first script that prints the famous message.

```
$$ HelloWorld.wds - Hello World script
.block
{
  .printf "Hello World!\n"
}
```

This script is multiline and it has to be executed using either **$>**< or **$$><** command:

```
0:000> $$><c:\scripts\HelloWorld.wds
Hello World!
```

One line scripts can be executed when we type them in WinDbg command window or you load them from a file using **$<** or **$$<** commands:

```
$$ Hello World script; .block { .printf "Hello World!\n" }
```

We can see that in one line scripts comments and commands must be ended with a semicolon unless the command or comment is final. Semicolons are not required for multiline scripts if commands are on separate lines.

```
0:000> $$<c:\scripts\HelloWorld2.wds

0:000> $$ Hello World script; .block { .printf "Hello World!\n" }
Hello World!
```

From now on we will use only multiline scripts because of their readability. You might have noticed that I deliberately made the first script more complex than necessary by enclosing **.printf** in **.block { }** to show the resemblance to C-style function:

```
// Hello World function
void helloWorld ()
{
  printf ("Hello World!\n");
}
```

SIMPLE ARITHMETIC

Consider the simple C-style function that prints the sum of 2 numbers and uses local variables:

```
void sum ()
{
    unsigned long t1 = 2;
    unsigned long t2 = 3;
    unsigned long t0 = t1 + t2;
    printf("Sum(%x,%x) = %x\n", t1, t2, t0);
}
```

In WinDbg scripts we can use 20 different user-defined variables called pseudo-registers. Their names are **$t0** - **$t19**. If you want to obtain the pseudo-register value then use **@** symbol, for example, **@$t0**. We can use **%p** type field character in **.printf** to interpret the value as a pointer. This is the equivalent WinDbg script and its output:

```
$$ Arithmetic1.wds - Calculate the sum of two predefined variables
.block
{
   r $t1 = 2
   r $t2 = 3
   r $t0 = @$t1 + @$t2
   .printf "Sum(%p, %p) = %p\n", @$t1, @$t2, @$t0
}
```

```
0:000> $$>a<c:\scripts\Arithmetic1.wds

Sum(0000000000000002, 0000000000000003) = 0000000000000005
```

Using hardcoded values is not useful. Let's rewrite the same function to use parameters. The equivalent to function arguments in WinDbg scripts are **$arg1** - **$argN** aliases to character strings. To obtain the alias value enclose it into ${…}, for example, **${$arg1}**. However we don't need to enclose it if you use it in some expression and the type of the argument can be inferred from other participating operands.

```
$$ Arithmetic2.wds - Calculate the sum of two function arguments
.block
{
   r $t0 = $arg1 + $arg2
   .printf "Sum(%p, %p) = %p\n", ${$arg1}, ${$arg2}, @$t0
}
```

Now we can call scripts and specify arguments:

```
0:000> $$>a<c:\scripts\Arithmetic2.wds 2 3

Sum(0000000000000002, 0000000000000003) = 0000000000000005
```

If some arguments are missing we get an error:

```
0:000> $$>a<c:\scripts\Arithmetic2.wds

Couldn't resolve error at '${$arg1} + ${$arg2};   .printf "Sum(%p, %p) =
%p\n", ${$arg1}, ${$arg2}, @$t0;'
```

WinDbg allows us to check whether arguments are defined or not. This can be done via a special form of the alias evaluator **${/d:…}**:

```
$$ Arithmetic3.wds - Calculate the sum of two optional function arguments
.block
{
   r $t1 = 0
   .if (${/d:$arg1})
   {
      r $t1 = $arg1
   }
   r $t2 = 0
   .if (${/d:$arg2})
   {
      r $t2 = $arg2
   }
   .printf "Sum(%p, %p) = %p\n", @$t1, @$t2, @$t1+@$t2
}
```

Here is the script output for some arguments:

```
0:000> $$>a<c:\scripts\Arithmetic3.wds

Sum(0000000000000000, 0000000000000000) = 0000000000000000

0:000> $$>a<c:\scripts\Arithmetic3.wds 2

Sum(0000000000000002, 0000000000000000) = 0000000000000002

0:000> $$>a<c:\scripts\Arithmetic3.wds 2 3

Sum(0000000000000002, 0000000000000003) = 0000000000000005
```

FACTORIAL

Let's write more complicated script that computes the factorial of the given number. Recall the following definition of the factorial function:

n! = 1*2*3*4*…*(n-2)*(n-1)*n

This function can be computed recursively using this code:

```
// C-style factorial function using recursion
unsigned long factorial (unsigned long n)
{
   unsigned long f = 0;
   if (n > 1)
   {
     f = n*factorial(n-1);
   }
   else
   {
     f = 1;
   }
   return f;
}
```

Alternatively it can be computed using **while** or **for** loops:

```
// C-style factorial function using a "while" loop
unsigned long factorial (unsigned long n)
{
  unsigned long k=1;
  while (n-1)
  {
    k = k * n;
    --n;
  }
  return k;
}
// C-style factorial function using a "for" loop
unsigned long factorial2 (unsigned long n)
{
  unsigned long k=1;
  for (; n-1; --n)
  {
    k = k * n;
  }
  return k;
}
```

WinDbg scripts can be called recursively too. We can map C-style code to WinDbg script where **$t0** pseudo register is used to simulate the function return value:

```
$$ FactorialR.wds - Calculate factorial using recursion
.block
{
  .if (${$arg1} > 1)
  {
    $$>a<c:\scripts\FactorialR.wds ${$arg1}-1
    r $t1 = $arg1
    r $t0 = @$t1 * @$t0
  }
  .else
  {
    r $t0 = 1
  }
  .printf "Factorial(%p) = %p\n", ${$arg1}, @$t0
}
```

The output of the script for some arguments:

```
0:000> $$>a<c:\scripts\FactorialR.wds 1
Factorial(0000000000000001) = 0000000000000001
0:000> $$>a<c:\scripts\FactorialR.wds 2
Factorial(0000000000000001) = 0000000000000001
Factorial(0000000000000002) = 0000000000000002
0:000> $$>a<c:\scripts\FactorialR.wds 3
Factorial(0000000000000001) = 0000000000000001
Factorial(0000000000000002) = 0000000000000002
Factorial(0000000000000003) = 0000000000000006
0:000> $$>a<c:\scripts\FactorialR.wds 4
Factorial(0000000000000001) = 0000000000000001
Factorial(0000000000000002) = 0000000000000002
Factorial(0000000000000003) = 0000000000000006
Factorial(0000000000000004) = 0000000000000018
0:000> $$>a<c:\scripts\FactorialR.wds 10
Factorial(0000000000000001) = 0000000000000001
Factorial(0000000000000002) = 0000000000000002
Factorial(0000000000000003) = 0000000000000006
Factorial(0000000000000004) = 0000000000000018
Factorial(0000000000000005) = 0000000000000078
Factorial(0000000000000006) = 00000000000002d0
Factorial(0000000000000007) = 00000000000013b0
Factorial(0000000000000008) = 0000000000009d80
Factorial(0000000000000009) = 0000000000058980
Factorial(000000000000000a) = 0000000000375f00
Factorial(000000000000000b) = 0000000002611500
Factorial(000000000000000c) = 000000001c8cfc00
Factorial(000000000000000d) = 000000017328cc00
Factorial(000000000000000e) = 000000144c3b2800
Factorial(000000000000000f) = 0000013077775800
Factorial(0000000000000010) = 0000130777758000
```

Now we are ready to rewrite our script using a **while** loop.

```
 $$ FactorialL.wds - Calculate factorial using a "while" loop
.block
{
    r $t0 = 1
    r $t1 = $arg1
    .while (@$t1-1)
    {
        r $t0 = @$t0 * @$t1
        r $t1 = @$t1 - 1
    }
    .printf "Factorial(%p) = %p\n", ${$arg1}, @$t0
}
```

The output of the script for some arguments:

```
0:000> $$>a<c:\scripts\FactorialL.wds 1
Factorial(0000000000000001) = 0000000000000001
0:000> $$>a<c:\scripts\FactorialL.wds 2
Factorial(0000000000000002) = 0000000000000002
0:000> $$>a<c:\scripts\FactorialL.wds 3
Factorial(0000000000000003) = 0000000000000006
0:000> $$>a<c:\scripts\FactorialL.wds 4
Factorial(0000000000000004) = 0000000000000018
0:000> $$>a<c:\scripts\FactorialL.wds 10
Factorial(0000000000000010) = 0000130777758000
```

We can simplify the script using **.for** loop token:

```
$$ FactorialL2.wds - Calculate factorial using a "for" loop
.block
{
    .for (r $t0 = 1, $t1 = $arg1; @$t1-1; r $t1 = @$t1 - 1)
    {
       r $t0 = @$t0 * @$t1
    }
    .printf "Factorial(%p) = %p\n", ${$arg1}, @$t0
}
```

Its output is the same:

```
0:000> $$>a<c:\scripts\FactorialL2.wds 4

Factorial(0000000000000004) = 0000000000000018
```

GENERATING FILE NAME FOR .DUMP COMMAND

.dump WinDbg command doesn’t have an option to include the process name although we can specify PID, date and time using **/u** option. To generate names we can use aliases:

```
      as /c CrashApp [get a module name here]

      .dump /ma /u c:\UserDumps\${CrashApp}.dmp
```

Unfortunately an attempt to use **lm** command fails due to a line break in the output:

```
0:001> lmM *.exe 1m
notepad

0:001> as /c CrashApp lmM *.exe 1m

0:001> .dump /ma /u c:\UserDumps\${CrashApp}.dmp
Unable to create file 'c:\UserDumps\notepad
_06ec_2008-08-13_14-39-30-218_06cc.dmp‘ - Win32 error 0n123
    “The filename, directory name, or volume label syntax is incorrect.”
```

We recall that **.printf** command doesn’t output line breaks. Also the module name can be extracted from _PEB structure if it is accessible. **$peb** pseudo-register can be used to get PEB address automatically. Therefore we can use the following alias:

```
as /c CrashFirstModule .printf "%mu",
@@c++((*(ntdll!_LDR_DATA_TABLE_ENTRY**)&@$peb->Ldr->
InLoadOrderModuleList.Flink)->BaseDllName.Buffer)

0:001> as /c CrashFirstModule .printf "%mu",
@@c++((*(ntdll!_LDR_DATA_TABLE_ENTRY**)&@$peb->Ldr->
InLoadOrderModuleList.Flink)->BaseDllName.Buffer)

0:001> .dump /ma /u c:\UserDumps\${CrashFirstModule}.dmp
Creating c:\UserDumps\notepad.exe_06ec_2008-08-13_14-44-51-702_06cc.dmp -
mini user dump
Dump successfully written
```

These commands can be included in a script for a postmortem debugger, for example, CDB.

ALL AT ONCE: POSTMORTEM LOGS AND DUMP FILES

To partially resolve security issues we use logs generated from memory dump files (Volume 1, page 230). In the case of process dumps the obvious step is to save logs by a postmortem debugger at the moment of the crash. Here WinDbg scripts come to the rescue. Suppose that CDB is set as a postmortem debugger (Volume 1, page 618) and AeDebug \ Debugger registry key value is set to:

```
"C:\Program Files\Debugging Tools for Windows\cdb.exe" -p %ld -e %ld -g -y
SRV*c:\mss*http://msdl.microsoft.com/download/symbols -c
"$$><c:\WinDbgScripts\LogsAndDumps.txt;q"
```

Here we specify MS symbols server and the script file. The symbol path is absolutely necessary to have correct stack traces. The script file has the following contents:

```
.logopen /t c:\UserDumps\process.log
.kframes 100
!analyze -v
~*kv
lmv
.logclose
.dump /m /u c:\UserDumps\mini_process
.dump /ma /u c:\UserDumps\full_process
.dump /mrR /u c:\UserDumps\secure_mini_process
.dump /marR /u c:\UserDumps\secure_full_process
```

.kframes WinDbg meta-command is necessary to avoid the common pitfall of looking at cut off stack traces (see page 39). In addition to logging the output of any command we want, the script writes 4 memory dumps of the same process:

- minidump
- full dump
- secure minidump
- secure full dump

The article **WinDbg is Privacy-Aware** (Volume 1, page 600) explains secure dumps in detail. If we need to tailor dump file names and logs to include real process name might need to try the following or similar technique explained on the page 37.

COMMON MISTAKES

NOT LOOKING AT FULL STACK TRACES

By default WinDbg cuts off stack traces after 20th line and an analyst misses essential information when looking at **Stack Trace** (Volume 1, page 395) or **Stack Trace Collection** (Volume 1, page 409). Consider the following thread stack trace taken from a user process dump where runaway information (Volume 1, page 305) was not saved but customers reported CPU spikes:

```
0:000> ~3kvL
ChildEBP RetAddr
0290f864 773976f2 user32!_SEH_prolog+0xb
0290f86c 0047f9ec user32!EnableMenuItem+0xf
0290f884 00488f6d Application!Close+0x142c
0290f8a4 0047a9c6 Application!EnableMenu+0x5d
0290f8b8 0048890d Application!EnableWindows+0x106
0290f8d0 0048cc2b Application!SetHourGlass+0xbd
0290f8fc 0046327a Application!WriteDataStream+0x24b
0290f924 0048d8f9 Application!WriteDataStream+0x21a
0290fa68 00479811 Application!WriteDataStream+0xcb9
0290fadc 5b5e976c Application!OnWrite+0x3c1
0290fb70 5b60e0b0 mfc71!CWnd::OnWndMsg+0x4f2
0290fb90 5b60e14f mfc71!CWnd::WindowProc+0x22
0290fbf0 5b60e1b8 mfc71!AfxCallWndProc+0x91
0290fc10 00516454 mfc71!AfxWndProc+0x46
0290fc3c 7739c3b7 Application!ExitCheck+0x28f34
0290fc68 7739c484 user32!InternalCallWinProc+0x28
0290fce0 77395563 user32!UserCallWinProcCheckWow+0x151
0290fd10 773ad03f user32!CallWindowProcAorW+0x98
0290fd30 0047a59a user32!CallWindowProcA+0x1b
```

We can see that it uses MFC libraries and window messaging API but was it caught accidentally? Is it a typical message loop like idle message loops in **Passive Thread** pattern (Volume 1, page 430) using GetMessage or it is an active GUI message pump using PeekMessage? If we expand stack trace we would see that the thread is actually MFC GUI thread that spins according to MFC source code:

```
int CWinThread::Run()
{
  for (;;)
  {
    while (bIdle &&
      !::PeekMessage(&(pState->m_msgCur),
         NULL, NULL, NULL, PM_NOREMOVE))
     {
```

```
0:000> ~3kvL 100
ChildEBP RetAddr
0290f864 773976f2 user32!_SEH_prolog+0xb
0290f86c 0047f9ec user32!EnableMenuItem+0xf
0290f884 00488f6d Application!Close+0x142c
0290f8a4 0047a9c6 Application!EnableMenu+0x5d
0290f8b8 0048890d Application!EnableWindows+0x106
0290f8d0 0048cc2b Application!SetHourGlass+0xbd
0290f8fc 0046327a Application!WriteDataStream+0x24b
0290f924 0048d8f9 Application!WriteDataStream+0x21a
0290fa68 00479811 Application!WriteDataStream+0xcb9
0290fadc 5b5e976c Application!OnWrite+0x3c1
0290fb70 5b60e0b0 mfc71!CWnd::OnWndMsg+0x4f2
0290fb90 5b60e14f mfc71!CWnd::WindowProc+0x22
0290fbf0 5b60e1b8 mfc71!AfxCallWndProc+0x91
0290fc10 00516454 mfc71!AfxWndProc+0x46
0290fc3c 7739c3b7 Application!ExitCheck+0x28f34
0290fc68 7739c484 user32!InternalCallWinProc+0x28
0290fce0 77395563 user32!UserCallWinProcCheckWow+0x151
0290fd10 773ad03f user32!CallWindowProcAorW+0x98
0290fd30 0047a59a user32!CallWindowProcA+0x1b
0290fdb0 7739c3b7 Application!OnOK+0x77a
0290fddc 7739c484 user32!InternalCallWinProc+0x28
0290fe54 7739c73c user32!UserCallWinProcCheckWow+0x151
0290febc 7738e406 user32!DispatchMessageWorker+0x327
0290fecc 5b609076 user32!DispatchMessageA+0xf
0290fedc 5b60913e mfc71!AfxInternalPumpMessage+0x3e
0290fef8 004ba7cf mfc71!CWinThread::Run+0×54
0290ff04 5b61b30c Application!CMyThread::Run+0xf
0290ff84 5b869565 mfc71!_AfxThreadEntry+0×100
0290ffb8 77e66063 msvcr71!_endthreadex+0xa0
0290ffec 00000000 kernel32!BaseThreadStart+0×34
```

There is also WinDbg **.kframes** meta-command that can change default stack trace depth:

```
2: kd> .kframes 0n100
Default stack trace depth is 0n100 frames
```

This command is highly recommended to run before dumping all stack traces from kernel and complete memory dump files, for example before **!process 0 ff** command.

NOT SEEING SEMANTIC AND PRAGMATIC INCONSISTENCIES

Why would FreeHeap need a file name? See **Incorrect Stack Trace** pattern case study (Volume 1, page 288) for semantic inconsistency. Why is this function on the stack trace

```
       dll!exit+0x10,834
```

67,636 bytes long (0×10,834 in decimal)?

The latter is an example of pragmatic inconsistency and the answer is that we don’t have symbols and the name appears from the DLL export table. The code on the stack has nothing to do with exit action when proper symbols are applied.

Another example. The memory dump of a hanging process has only one thread and it is waiting for an event. Is this the problem in ThreadProc and application logic or in the fact that _endthreadex was called when the thread was created?

```
STACK_TEXT:
0379fa50 7642dcea ntdll!NtWaitForMultipleObjects+0x15
0379faec 75e08f76 kernel32!WaitForMultipleObjectsEx+0x11d
0379fb40 75e08fbf user32!RealMsgWaitForMultipleObjectsEx+0x14d
0379fb5c 00f6b45d user32!MsgWaitForMultipleObjects+0x1f
0379fba8 752e29bb application!ThreadProc+0xad
0379fbe0 752e2a47 msvcr80!_endthreadex+0×3b
0379fbe8 7649e3f3 msvcr80!_endthreadex+0xc7
0379fbf4 7773cfed kernel32!BaseThreadInitThunk+0xe
0379fc34 7773d1ff ntdll!__RtlUserThreadStart+0×23
0379fc4c 00000000 ntdll!_RtlUserThreadStart+0×1b
```

The latter assumption is wrong. The presence of _endthreadex stems from the fact that its address was pushed to let a user thread procedure to automatically call it upon the normal function return:

```
0:000> u 752e29bb
msvcr80!_endthreadex+0x3b:
752e29bb 50               push    eax
752e29bc e8bfffffff       call    msvcr80!_endthreadex (752e2980)
752e29c1 8b45ec           mov     eax,dword ptr [ebp-14h]
752e29c4 8b08             mov     ecx,dword ptr [eax]
752e29c6 8b09             mov     ecx,dword ptr [ecx]
752e29c8 894de4           mov     dword ptr [ebp-1Ch],ecx
752e29cb 50               push    eax
752e29cc 51               push    ecx
```

A thread procedure passed to thread creation API call can be any C function. How would a C/C++ compiler understand that it needs to generate a call to thread exit API especially if ThreadProc is named FooBar and resides in a different compilation unit or a library? It seems logical that the runtime environment provides such an automatic return address dynamically. Also why and how _endthreadex knows about our custom ThreadProc to call it? Looks like inconsistency. The ability to see and reason about them is the very important skill in memory dump analysis and debugging. The lack of sufficient unmanaged code programming experience might partly explain many analysis mistakes.

PATTERN INTERACTION

HEURISTIC STACK TRACE

Here is another 64-bit example of **Hidden Exception** pattern (Volume 1, page 271) where looking at raw stack data helps in problem identification. Opening the dump in 6.8.0004.0 version of WinDbg shows this meaningless stack trace:

```
00000000`00000000 ??              ???

0:035> kL
Child-SP          RetAddr           : Call Site
00000000`00000000 00000000`00000000 : 0x0
```

Analysis command doesn’t help too:

```
FAULTING_IP:
ntdll!DbgBreakPoint+0
00000000`77ef2aa0 cc              int     3

EXCEPTION_RECORD:  ffffffffffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 0000000077ef2aa0 (ntdll!DbgBreakPoint)
   ExceptionCode: 80000003 (Break instruction exception)
  ExceptionFlags: 00000000
NumberParameters: 1
   Parameter[0]: 0000000000000000

FAULTING_THREAD:  0000000000000e50

DEFAULT_BUCKET_ID:  STATUS_BREAKPOINT

PROCESS_NAME:  application.exe

ERROR_CODE: (NTSTATUS) 0x80000003 - {EXCEPTION}  Breakpoint  A breakpoint
has been reached.

NTGLOBALFLAG:  2000000

APPLICATION_VERIFIER_FLAGS:  0

LAST_CONTROL_TRANSFER:  from 0000000000000000 to 0000000000000000

STACK_TEXT:
00000000`00000000 00000000`00000000 : 00000000`00000000 00000000`00000000
00000000`00000000 00000000`00000000 : 0x0

STACK_COMMAND:  kb

PRIMARY_PROBLEM_CLASS:  STATUS_BREAKPOINT
```

```
BUGCHECK_STR:  APPLICATION_FAULT_STATUS_BREAKPOINT_STACK_CORRUPTION

FOLLOWUP_IP:
ntdll!DbgBreakPoint+0
00000000`77ef2aa0 cc              int     3

SYMBOL_NAME:  ntdll!DbgBreakPoint+0

FOLLOWUP_NAME:  MachineOwner

MODULE_NAME: ntdll

IMAGE_NAME:  ntdll.dll

DEBUG_FLR_IMAGE_TIMESTAMP:  45d6cc72

FAILURE_BUCKET_ID:  ntdll.dll!DbgBreakPoint_80000003_STATUS_BREAKPOINT

BUCKET_ID:  X64_APPLICATION_FAULT_STATUS_BREAKPOINT_STACK_CORRUPTION_ntdll
!DbgBreakPoint+0

Followup: MachineOwner
---------
```

However, looking at thread raw stack data allows us to get the problem stack trace showing that the full page heap enabled process detected heap corruption (Volume 1, page 257) during **free** operation (stack trace shown in smaller font for visual clarity):

```
0:035> !teb
TEB at 000007fffff72000
    ExceptionList:        0000000000000000
    StackBase:            00000000080e0000
    StackLimit:           00000000080d8000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 000007fffff72000
    EnvironmentPointer:   0000000000000000
    ClientId:             0000000000000918 . 0000000000000e50
    RpcHandle:            0000000000000000
    Tls Storage:          0000000000000000
    PEB Address:          000007fffffd8000
    LastErrorValue:       0
    LastStatusValue:      c0000034
    Count Owned Locks:    0
    HardErrorMode:        0

0:035> dds 00000000080d8000 00000000080e0000
...
00000000`080dd7b8  00000000`77ef3202 ntdll!KiUserExceptionDispatcher+0×52
00000000`080dd7c0  00000000`0178070a
```

```
00000000`080dd7c8  00000000`080dd7c0 ; exception context
00000000`080dd7d0  00000000`08599d30
00000000`080dd7d8  00000000`00000020
00000000`080dd7e0  00000000`00000000
00000000`080dd7e8  00000000`00000000
00000000`080dd7f0  00001fa0`0010001f
00000000`080dd7f8  0053002b`002b0033
00000000`080dd800  00000202`002b002b
00000000`080dd808  00000000`00000000
...

0:035> .cxr 00000000`080dd7c0
rax=0000000000000001 rbx=0000000008599d30 rcx=000077fad8cd0000
rdx=00000000ffff0165 rsi=0000000077ec0000 rdi=0000000000000000
rip=0000000077ef2aa0 rsp=00000000080ddd58 rbp=0000000000000020
 r8=00000000ffffffff  r9=0000000000000000 r10=0000000000000007
r11=0000000000000000 r12=00000000080dde70 r13=0000000077f5d300
r14=0000000077f5d2f0 r15=0000000077f86bc0
iopl=0  nv up ei pl nz na pe nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000202
ntdll!DbgBreakPoint:
00000000`77ef2aa0 cc              int     3
```

```
0:035> kL 100
Child-SP          RetAddr           Call Site
00000000`080ddd58 00000000`77f5c78d ntdll!DbgBreakPoint
00000000`080ddd60 00000000`77f5da05 ntdll!RtlpDphReportCorruptedBlock+0×86d
00000000`080dde50 00000000`77f5a0f3 ntdll!RtlpDphNormalHeapFree+0×45
00000000`080ddfb0 00000000`77f60d5b ntdll!RtlpDebugPageHeapFree+0×203
00000000`080de0e0 00000000`77f3bcc8 ntdll!RtlDebugFreeHeap+0×3b
00000000`080de170 00000000`77edc095 ntdll!RtlFreeHeapSlowly+0×4e
00000000`080de2e0 000007ff`7fc2daab ntdll!RtlFreeHeap+0×15e
00000000`080de3f0 00000000`67fa288f msvcrt!free+0×1b
00000000`080de420 00000000`1000d3e9 dll!FreeMem+0xf
...
00000000`080df180 000007ff`7fe96cc9 RPCRT4!Invoke+0×65
00000000`080df1e0 000007ff`7fe9758d RPCRT4!NdrStubCall2+0×54d
00000000`080df7a0 000007ff`7fd697b4 RPCRT4!NdrServerCall2+0×1d
00000000`080df7d0 000007ff`7fde06b6 RPCRT4!DispatchToStubInCNoAvrf+0×14
00000000`080df800 000007ff`7fd6990d RPCRT4!DispatchToStubInCAvrf+0×16
00000000`080df830 000007ff`7fd69766 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0×50d
00000000`080df9a0 000007ff`7fd6b214 RPCRT4!RPC_INTERFACE::DispatchToStub+0×2ec
00000000`080dfa20 000007ff`7fd6b9e3 RPCRT4!LRPC_SCALL::DealWithRequestMessage+0×63b
00000000`080dfae0 000007ff`7fd7007c RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0×3bf
00000000`080dfba0 000007ff`7fd45369 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0×710
00000000`080dfeb0 000007ff`7fd65996 RPCRT4!RecvLotsaCallsWrapper+0×9
00000000`080dfee0 000007ff`7fd65d51 RPCRT4!BaseCachedThreadRoutine+0xde
00000000`080dff50 00000000`77d6b6da RPCRT4!ThreadStartRoutine+0×21
00000000`080dff80 00000000`00000000 kernel32!BaseThreadStart+0×3a
```

Opening the dump in 6.9.3.113 version of WinDbg and running verbose analysis command shows “heuristic” stack trace (all symbols from raw stack) similar to old W2K extension **stack** command (see **Guessing Stack Trace**, Volume 1, page 193) where exception processing hints is highlighted in bold italics:

```
0:035> !analyze -v
...
STACK_TEXT:
00000000`77f15fb3 ntdll!RtlLookupFunctionTable
000007ff`5738e62c ole32!_pfnDliNotifyHook2 <PERF> (ole32+0x24e62c)
000007ff`57140000 ole32!_imp_TraceMessage <PERF> (ole32+0x0)
000007ff`57178356 ole32!ICoCreateInstanceEx
00000000`77ec0000 ntdll!_real <PERF> (ntdll+0x0)
00000000`77ef31dd ntdll!KiUserExceptionDispatcher
00000000`77f3ad68 ntdll!RtlAllocateHeapSlowly
00000000`77f967b8 ntdll!$$VProc_ImageExportDirectory
00000000`77f416ce ntdll!RtlpLookupFunctionEntryForStackWalks
00000000`77ef2aa0 ntdll!DbgBreakPoint
00000000`77ee5a36 ntdll!RtlVirtualUnwind
00000000`77f41c13 ntdll!RtlpWalkFrameChain
00000000`77f5d300 ntdll!`string'
00000000`77f5d2f0 ntdll!`string'
00000000`77f86bc0 ntdll!`string'
00000000`77ee455d ntdll!RtlpExecuteHandlerForException
00000000`77edc095 ntdll!RtlFreeHeap
00000000`77f979e4 ntdll!$$VProc_ImageExportDirectory
00000000`77ed609a ntdll!RtlCreateProcessParameters
00000000`77d5c71f kernel32!BasePushProcessParameters
00000000`77dc059b kernel32!UnhandledExceptionFilter
00000000`77ee6097 ntdll!RtlDispatchException
00000000`77f51285 ntdll!RtlpCaptureStackTraceForLogging
00000000`77f60270 ntdll!RtlDebugAllocateHeap
00000000`77f511a3 ntdll!RtlpExtendStackTraceDataBase
00000000`77d6ec00 kernel32!BasepComputeProcessPath
00000000`77d5c5b2 kernel32!BasePushProcessParameters
00000000`77d59c71 kernel32!CreateProcessInternalW
00000000`77dbc2df kernel32!BaseThreadStart
00000000`77ee6583 ntdll!_C_specific_handler
00000000`77f51432 ntdll!RtlpLogCapturedStackTrace
00000000`77f5e572 ntdll!RtlpDphLogStackTrace
00000000`77d5c4b2 kernel32!BasePushProcessParameters
00000000`77f4bb56 ntdll!DeleteNodeFromTree
00000000`77f4bf24 ntdll!RtlDeleteElementGenericTableAvl
00000000`77f574e1 ntdll!RtlpDphRemoveFromBusyList
00000000`77f5a0dd ntdll!RtlpDebugPageHeapFree
00000000`77f41799 ntdll!RtlCaptureStackBackTrace
00000000`67fa288f dll!FreeMem
00000000`77f5e559 ntdll!RtlpDphLogStackTrace
00000000`77f5a09f ntdll!RtlpDebugPageHeapFree
00000000`77f60d5b ntdll!RtlDebugFreeHeap
00000000`77f3bcc8 ntdll!RtlFreeHeapSlowly
00000000`77d4f7bc kernel32!CreateProcessInternalW
00000000`77f513bd ntdll!RtlpLogCapturedStackTrace
00000000`77ed495f ntdll!RtlDestroyProcessParameters
00000000`77d5c7c2 kernel32!BasePushProcessParameters
00000000`77dc0730 kernel32!`string’
00000000`77d813f0 kernel32!`string’
00000001`000000e0 application!_imp_RegQueryValueExW <PERF>
(application+0xe0)
```

```
00000000`77ef9971 ntdll!RtlImageNtHeader
00000000`77d6b302 kernel32!BaseCreateStack
00000000`77d5c8a1 kernel32!BaseInitializeContext
00000000`77ef5a81 ntdll!CsrClientCallServer
00000000`77d5c829 kernel32!CreateProcessInternalW
00000001`00000001 application!_imp_RegQueryValueExW <PERF>
(application+0×1)
00000001`00000000 application!_imp_RegQueryValueExW <PERF>
(application+0×0)
000007ff`57178717 ole32!CProcessActivator::CCICallback
000007ff`571921bf ole32!CoCreateInstance
00000000`77d59620 kernel32!BaseProcessStart
00000000`77dc05d4 kernel32!UnhandledExceptionFilter
00000000`77e346e0 kernel32!__PchSym_ <PERF> (kernel32+0xf46e0)
00000000`77d6b6da kernel32!BaseThreadStart
000007ff`7fe7a934 RPCRT4!Ndr64pSizing
00000000`77f41c93 ntdll!RtlpWalkFrameChain
00000000`77edca76 ntdll!RtlAllocateHeap
00000000`77d40000 kernel32!_imp_memcpy <PERF> (kernel32+0×0)
00000000`77fa0100 ntdll!RtlpStaticDebugInfo
00000000`77ed08b3 ntdll!vsnwprintf
00000000`77dbf42c kernel32!StringCchPrintfW
00000000`77d6e314 kernel32!CloseHandle
00000000`77dc06d8 kernel32!UnhandledExceptionFilter
00000000`77e0a958 kernel32!`string'
00000000`77e29080 kernel32!CfgmgrDllString
000007ff`7fd697b4 RPCRT4!DispatchToStubInCNoAvrf
00000000`77efc2d9 ntdll!bsearch
00000000`77efc791 ntdll!RtlpFindUnicodeStringInSection
00000000`77e23454 kernel32!__PchSym_ <PERF> (kernel32+0xe3454)
00000000`77e1d324 kernel32!g_hModW03A2409
00000000`77e1d330 kernel32!g_hModW03A2409
00000000`77f39fce ntdll!RtlLookupFunctionEntry
00000000`77f39231 ntdll!RtlDispatchException
00000000`77fa3c70 ntdll!RtlpCallbackEntryList
00000000`77d92290 kernel32!_C_specific_handler
00000000`77e30033 kernel32!__PchSym_ <PERF> (kernel32+0xf0033)
000007ff`7fd65d51 RPCRT4!ThreadStartRoutine
00000000`77efc437 ntdll!RtlpLocateActivationContextSection
00000000`77ef8708 ntdll!RtlFindActivationContextSectionString
000007ff`7fc2dab0 msvcrt!free
00000000`77fc5f08 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0×105f08)
00000000`77fc5fe0 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0×105fe0)
00000000`77fc5dd0 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0×105dd0)
00000000`77fc6250 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0×106250)
00000000`77fc2614 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0×102614)
00000000`77fb2e28 ntdll!CsrPortMemoryRemoteDelta <PERF> (ntdll+0xf2e28)
000007ff`7fc00000 msvcrt!_imp_MultiByteToWideChar <PERF> (msvcrt+0×0)
000007ff`7fc7fb78 msvcrt!bufin <PERF> (msvcrt+0×7fb78)
00000000`77ef3202 ntdll!KiUserExceptionDispatcher
00000000`77f86220 ntdll!`string'
00000000`77f5c78d ntdll!RtlpDphReportCorruptedBlock
00000000`77f5d1b0 ntdll!`string'
00000000`77f5d1e0 ntdll!`string'
```

```
00000000`77f5d200 ntdll!`string’
00000000`77f5da05 ntdll!RtlpDphNormalHeapFree
000007ff`7fde06b6 RPCRT4!DispatchToStubInCAvrf
000007ff`7fd7007c RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls
000007ff`7fd45369 RPCRT4!RecvLotsaCallsWrapper
000007ff`7fd65996 RPCRT4!BaseCachedThreadRoutine
00000000`77f57370 ntdll!RtlpDphFindBusyMemory
00000000`77f5a0f3 ntdll!RtlpDebugPageHeapFree
000007ff`57197e5b ole32!CRetailMalloc_Free
00000000`77c30000 USER32!InternalCreateDialog
000007ff`5719a21a ole32!COleStaticMutexSem::Request
00000000`77d6d6e1 kernel32!FreeLibrary
000007ff`7ebc0000 OLEAUT32!_imp_RegFlushKey <PERF> (OLEAUT32+0×0)
000007ff`56db3024 msxml3!ModelInit::~ModelInit
00000000`77d6e76c kernel32!LocalAlloc
000007ff`7fc2daab msvcrt!free
000007ff`7fd70000 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls
000007ff`7ff0b397 ADVAPI32!LocalBaseRegOpenKey
000007ff`7ff0b977 ADVAPI32!RegQueryValueExW
000007ff`7ff0b20e ADVAPI32!LocalBaseRegCloseKey
000007ff`7ff0b19f ADVAPI32!RegCloseKey
00000000`77ef7b33 ntdll!RtlNtStatusToDosError
000007ff`7fd66238 RPCRT4!LRPC_SCALL::ImpersonateClient
00000000`77efbcdf ntdll!RtlEqualSid
000007ff`7fd662a6 RPCRT4!LRPC_SCALL::RevertToSelf
000007ff`7ff0c6d4 ADVAPI32!GetTokenInformation
000007ff`7fd5cb7b RPCRT4!RpcRevertToSelf
000007ff`7fd666b2 RPCRT4!SCALL::AddToActiveContextHandles
000007ff`7fd37f76 RPCRT4!NDRSContextUnmarshall2
00000000`77f5a001 ntdll!RtlpDebugPageHeapFree
000007ff`7fd6f32b RPCRT4!DCE_BINDING::`scalar deleting destructor’
000007ff`7fd604c3 RPCRT4!RpcStringBindingParseW
000007ff`7fd30000 RPCRT4!_imp_GetSecurityDescriptorDacl <PERF>
(RPCRT4+0×0)
000007ff`7fd66374 RPCRT4!NdrServerContextNewUnmarshall
000007ff`7fd605e5 RPCRT4!RpcStringFreeA
000007ff`7fd69e00 RPCRT4!NdrServerInitialize
000007ff`7fd65e81 RPCRT4!RPC_INTERFACE::CheckSecurityIfNecessary
000007ff`7fd6b9e3 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest
000007ff`7fd66c1e RPCRT4!NdrUnmarshallHandle
000007ff`7fd69a75 RPCRT4!Invoke
000007ff`7fe96cc9 RPCRT4!NdrStubCall2
00000000`77f5e500 ntdll!RtlpDphFreeDelayedBlocksFromHeap
000007ff`7fd608b4 RPCRT4!SVR_BINDING_HANDLE::SVR_BINDING_HANDLE
00000000`77ef7dbb ntdll!RtlInitializeCriticalSectionAndSpinCount
000007ff`7fd60100 RPCRT4!DCE_BINDING::StringBindingCompose
000007ff`7fe9758d RPCRT4!NdrServerCall2
000007ff`7fd5cffd RPCRT4!ParseAndCopyEndpointField
000007ff`7fd604ce RPCRT4!RpcStringBindingParseW
000007ff`7fd6990d RPCRT4!RPC_INTERFACE::DispatchToStubWorker
000007ff`7fc40b0f msvcrt!getptd
000007ff`7fd37eaf RPCRT4!RpcServerInqCallAttributesW
000007ff`7fd65e9c RPCRT4!RPC_INTERFACE::CheckSecurityIfNecessary
000007ff`7fd69766 RPCRT4!RPC_INTERFACE::DispatchToStub
```

```
000007ff`7fd6b214 RPCRT4!LRPC_SCALL::DealWithRequestMessage
000007ff`7fd70466 RPCRT4!LRPC_ADDRESS::DereferenceAssociation
000007ff`7fd6ee28 RPCRT4!LRPC_SASSOCIATION::DealWithCopyMessage
000007ff`7fd65d30 RPCRT4!ThreadStartRoutine
00000000`77d6b6a0 kernel32!BaseThreadStart
```

MULTIPLE PATTERNS

Let’s apply the knowledge of crash dump analysis patterns to the real memory dump coming from the hanging system. **!locks** WinDbg command is the most frequently used start command for such symptoms and it reveals deep **Wait Chain** pattern for executive resources (page 147):

```
1: kd> !locks
...
Resource @ 0x8b10c168    Exclusively owned
    Contention Count = 1950624
    NumberOfSharedWaiters = 2
    NumberOfExclusiveWaiters = 154
     Threads: 88256430-01<*> 882aedb0-01    8a348580-01
     Threads Waiting On Exclusive Access:
              891fd368       894fd240       88382280       8921b9c0
              8aa18db0       89790328       882b8818       88a70bf8
              884a2780       88999818       8a13b020       8846a7a0
              8a0b3020       8812e568       897b6db0       88a16440
              8922c5f8       88bfe3b8       88264ac0       89ff2b40
              8a9da020       881cf020       8807adb0       89d64598
              887811d0       8822a850       88264820       88194738
              8801f7a8       88284020       88628db0       8a071db0
              884a84b0       88be46c0       89755b18       89700020
              89ca4580       881cddb0       882f7020       88bcf9a0
              8921b020       8826fdb0       88a73db0       88211020
              8868a1c8       89121280       89e01020       895cedb0
              88d03790       883941f8       8910f820       891ebc80
              89862db0       88154af8       8821e7d0       881cedb0
              8822b020       88094818       8a00b020       89e69020
              880bbdb0       8945f690       8954c1c0       88d2cb90
              881cd020       8921c448       89550540       8a5a5870
              8a159228       893976c0       882847b8       89306578
              880eb9a8       8978e020       882f72c0       8966c380
              8a12f4b8       8815adb0       881a5020       897c7db0
              8873ebf8       88674530       8831b468       88e999a0
              88287020       8966e600       88541db0       8826d7a0
              88119b10       8a226338       882f7810       888ba348
              884b89a8       88d03db0       8826db00       8910adb0
              881d8368       89288238       8a00adb0       89125db0
              88eb50a0       88dbbdb0       880ed020       895cd5b0
              881d4b00       88565db0       886e7780       884b86d8
              8a603598       89383020       8826f370       886d2248
              88cd1360       881d1888       88bef670       88117db0
              890d63f0       894d0368       8826f850       89123020
              88209020       8826fac0       88f9bdb0       89027478
              894b8d18       882a7338       899b9020       897c3db0
              8a13fc50       88b33d50       88b54b68       88652360
              8a199020       8910dc98       8833a020       8a194a70
              8a5af640       89b717a0       89464db0       8a152878
              884773d8       88afe020       88debaf8       88bef2a0
              88bd6948       89abddb0       8a133db0       88e0ebe8
              88287398       889622e0       8836aab8       88daec80
              88c5c450       88225718

...

Resource @ 0x8a316c98    Exclusively owned
    Contention Count = 40315
    NumberOfExclusiveWaiters = 2
     Threads: 893bd498-01<*>
     Threads Waiting On Exclusive Access:
              8846f9a8       88256430

...

Resource @ 0x8a2b3800    Exclusively owned
    Contention Count = 17735
    NumberOfExclusiveWaiters = 1
     Threads: 8a30ec80-01<*>
     Threads Waiting On Exclusive Access:
              893bd498

...

14606 total locks, 14 locks currently held
```

We have this chain: 154 threads -> **88256430** -> ***893bd498*** -> **8a30ec80**

The same conclusion comes from **!analyze -v -hang** command:

```
1: kd> !analyze -v -hang

...

Scanning for threads blocked on locks ...

CURRENT_IRQL:  2

BLOCKING_THREAD:  8a30ec80

LOCK_ADDRESS:  8a2b3800 -- (!locks 8a2b3800)

Resource @ 0x8a2b3800    Exclusively owned
    Contention Count = 17735
    NumberOfExclusiveWaiters = 1
     Threads: 8a30ec80-01<*>
     Threads Waiting On Exclusive Access:
              893bd498

BUGCHECK_STR:  LOCK_HELD

...
```

Let’s examine the thread 8a30ec80 that holds so many other threads:

```
1: kd> !thread 8a30ec80 1f
THREAD 8a30ec80  Cid 3ca0.20f0  Teb: 00000000 Win32Thread: 00000000
RUNNING on processor 2
Not impersonating
DeviceMap                 e1000930
Owning Process            8a254128       Image:         processA.exe
Wait Start TickCount      2024978        Ticks: 2291 (0:00:00:35.796)
Context Switch Count      339739
UserTime                  00:00:00.000
KernelTime                01:08:29.484
Start Address driverA!WorkerRoutine (0xbfa4b850)
Stack Init b7409000 Current b7407e74 Base b7409000 Limit b7406000 Call 0
Priority 13 BasePriority 13 PriorityDecrement 0
ChildEBP RetAddr
f77353a0 bfa4b880 driverA!DoProcessing+0×256
f77353a4 ff180010 driverA!WorkerRoutine+0×30
...
```

We see that this thread is running on processor 2 and the time spent in kernel is more than one hour. Seems we have the case of **Spiking Thread** pattern here (Volume 1, page 305). Also the thread is not waiting and seems to be moving some data:

```
1: kd> .thread 8a30ec80
Implicit thread is now 8a30ec80

1: kd> r
Last set context:
eax=baec2950 ebx=00000000 ecx=00001b4a edx=00002275 esi=bae8c010
edi=ba01a018
eip=bfa3b68c esp=f77353a4 ebp=b7407f44 iopl=0 nv up ei pl nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00000202
driverA!DoProcessing+0×256:
bfa3b68c f3a5 rep movs dword ptr es:[edi],dword ptr [esi]
```

This **driverA** was also known to have pool allocation problems resulting in various strange system behaviour so let’s inspect the output of **!vm** command to see whether we have an instance of **Insufficient Memory** pattern (Volume 1, page 440):

```
1: kd> !vm

*** Virtual Memory Usage ***
 Physical Memory:     1048242 (   4192968 Kb)
 Page File: \??\C:\pagefile.sys
   Current:   4456448 Kb  Free Space:   3483860 Kb
   Minimum:   4456448 Kb  Maximum:      4456448 Kb
 Available Pages:      409879 (   1639516 Kb)
 ResAvail Pages:       942398 (   3769592 Kb)
 Locked IO Pages:         124 (       496 Kb)
 Free System PTEs:     182782 (    731128 Kb)
 Free NP PTEs:          32766 (    131064 Kb)
 Free Special NP:           0 (         0 Kb)
 Modified Pages:           29 (       116 Kb)
 Modified PF Pages:        29 (       116 Kb)
 NonPagedPool Usage:    13567 (     54268 Kb)
 NonPagedPool Max:      65279 (    261116 Kb)
 PagedPool 0 Usage:     12350 (     49400 Kb)
 PagedPool 1 Usage:      2442 (      9768 Kb)
 PagedPool 2 Usage:      2457 (      9828 Kb)
 PagedPool 3 Usage:      2395 (      9580 Kb)
 PagedPool 4 Usage:      2465 (      9860 Kb)
 PagedPool Usage:       22109 (     88436 Kb)
 PagedPool Maximum:     64512 (    258048 Kb)

 ********** 3 pool allocations have failed **********

 Shared Commit:        133470 (    533880 Kb)
 Special Pool:              0 (         0 Kb)
 Shared Process:        24993 (     99972 Kb)
 PagedPool Commit:      22173 (     88692 Kb)
 Driver Commit:          2056 (      8224 Kb)
 Committed pages:      938909 (   3755636 Kb)
 Commit limit:        2119027 (   8476108 Kb)

...
```

We have signs here but the current size of paged pool and nonpaged pool seems to be very far from their maximum. Perhaps there were failures in session pool allocations? Let's look at session pool:

```
1: kd> !vm 4
...
 Session ID 5 @ f79f3000:
 Paged Pool Usage:       35640K

 *** 6 Pool Allocation Failures ***

 Commit Usage:           36900K
```

We see the problem in session 5 and let's see in which session our processA.exe was running:

```
1: kd> !process 8a254128 0
PROCESS 8a254128  SessionId: 15  Cid: 3ca0    Peb: 7ffd7000  ParentCid:
01ac
    DirBase: bff47800  ObjectTable: e779f5c8  HandleCount: 161.
    Image: processA.exe
```

It was session 15 and therefore we might conclude that previous problems with driverA are not connected to this new one. The identified problem is CPU spike. Perhaps the code contains a bug that causes this driver to loop indefinitely.

EXCEPTION AND DEADLOCK

When a process experienced an unhandled exception what were the possible reasons for a postmortem debugger not saving a crash dump? One of them will be illustrated here. The process AppA was hanging and causing another process AppB to hang too (see **Coupled Processes** pattern, Volume 1, page 419). If we look at AppA locked critical sections we would see a loader deadlock (Volume 1, page 276, stack trace below is shown in smaller font for visual clarity):

```
0:000> !locks

CritSec ntdll!LdrpLoaderLock+0 at 7c889d94
WaiterWoken        No
LockCount          8
RecursionCount     2
OwningThread       4534
EntryCount         0
ContentionCount    a3e
*** Locked

CritSec ntdll!FastPebLock+0 at 7c889d40
WaiterWoken        No
LockCount          2
RecursionCount     1
OwningThread       3eb4
EntryCount         0
ContentionCount    15
*** Locked
```

```
   3  Id: 30b0.3eb4 Suspend: 1 Teb: 7ffdb000 Unfrozen
ChildEBP RetAddr  Args to Child
00b3e11c 7c822124 7c83970f 0000004c 00000000 ntdll!KiFastSystemCallRet
00b3e120 7c83970f 0000004c 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
00b3e15c 7c839620 00000000 00000004 00000001 ntdll!RtlpWaitOnCriticalSection+0×19c
00b3e17c 7c832ad0 7c889d94 09150000 7c889e0c ntdll!RtlEnterCriticalSection+0xa8
00b3e1b0 77e68de4 00000001 00000000 00b3e1ec ntdll!LdrLockLoaderLock+0xe4
00b3e210 77e98fae 09150000 00b3e5b0 00000104 kernel32!GetModuleFileNameW+0×77
00b3e24c 77e76d5f 091518b8 00b3e5a4 00000000 kernel32!FillUEFInfo+0×51
00b3e8d4 77e84269 00b3e8fc 77e6b831 00b3e904 kernel32!UnhandledExceptionFilter+0xec
00b3e8dc 77e6b831 00b3e904 00000000 00b3e904 kernel32!BaseThreadStart+0×4a
00b3e904 7c82eeb2 00b3e9e8 00b3ffdc 00b3ea04 kernel32!_except_handler3+0×61
00b3e928 7c82ee84 00b3e9e8 00b3ffdc 00b3ea04 ntdll!ExecuteHandler2+0×26
00b3e9d0 7c82ecc6 00b38000 00b3ea04 00b3e9e8 ntdll!ExecuteHandler+0×24
00b3e9d0 7c832335 00b38000 00b3ea04 00b3e9e8 ntdll!KiUserExceptionDispatcher+0xe
(CONTEXT @ 00b3ea04)
00b3eeec 77e67319 00090000 00000000 0000056a ntdll!RtlAllocateHeap+0×9e3
00b3ef50 77e67690 77e676bc 00020a50 00000000 kernel32!BasepComputeProcessPath+0xc2
00b3ef90 77e41b95 00000000 00000000 00b3f05c kernel32!BaseComputeProcessDllPath+0xe3
00b3eff0 77e67b77 74065a30 00000000 00000000 kernel32!LoadLibraryExW+0×14e
00b3f004 7406615c 74065a30 007e1ab0 00000000 kernel32!LoadLibraryW+0×11
00b3f020 74066263 007e1ab0 00000000 534c4354 AppA!ProcessItem+0×2d
...

  21  Id: 30b0.4534 Suspend: 1 Teb: 7ffa3000 Unfrozen
ChildEBP RetAddr  Args to Child
031eeef8 7c822124 7c83970f 00000350 00000000 ntdll!KiFastSystemCallRet
031eeefc 7c83970f 00000350 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
031eef38 7c839620 00000000 00000004 031ef278 ntdll!RtlpWaitOnCriticalSection+0×19c
031eef58 7c830c25 7c889d40 031ef328 00000000 ntdll!RtlEnterCriticalSection+0xa8
031ef200 7c831016 00000001 031ef2ac 031ef288
ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0×97
031ef21c 7c830f85 031ef2ac 031ef288 00000000
ntdll!RtlDosPathNameToRelativeNtPathName_Ustr+0×18
031ef290 7c833ce9 031ef2ac 00000001 031ef414 ntdll!RtlDoesFileExists_UstrEx+0×1c
031ef2c0 7c83350e 031ef328 031ef434 031ef33c ntdll!LdrpSearchPath+0×76
031ef3c4 7c833637 0325fc78 031ef414 00000000 ntdll!LdrpCheckForLoadedDll+0xdc
031ef668 7c833ee5 00000000 0325fc78 031ef968 ntdll!LdrpLoadDll+0×1b3
031ef8e4 77e41bcc 0325fc78 031ef968 031ef948 ntdll!LdrLoadDll+0×198
031ef984 5e00b8d7 00115b00 00000000 00000008 kernel32!LoadLibraryExW+0×1b2
031ef9b0 5dff64aa 000cee40 7406275b 0000059c AppA!LoadPluginModule+0×42
...
```

Looking at TID#***3eb4*** we see that the cause for the deadlock was an exception while loading a DLL. Applying exception context WinDbg command **.cxr** reveals **Heap Corruption** pattern (Volume 1, page 257):

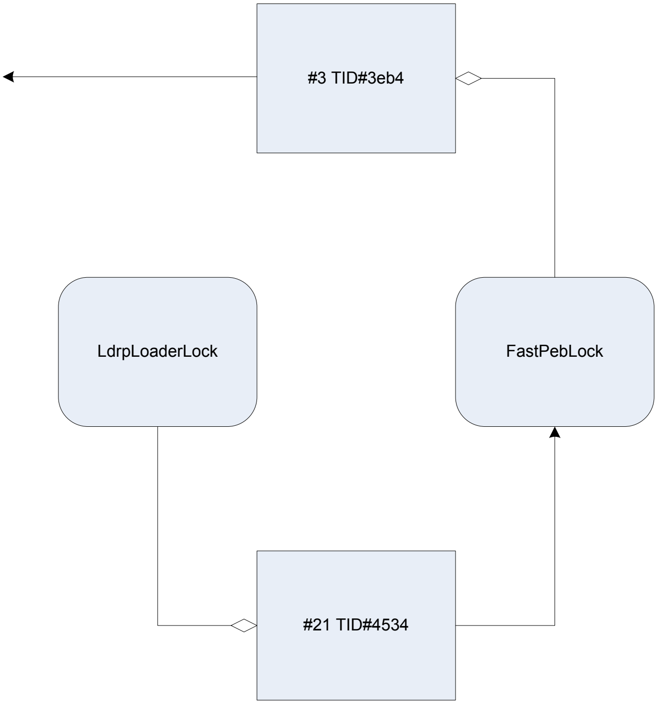
```
0:003> .cxr 00b3ea04
eax=0325f1f0 ebx=00000051 ecx=00090000 edx=00090400 esi=0008019d
edi=0325f1e8
eip=7c832335 esp=00b3ecd0 ebp=00b3eeec iopl=0  nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000  efl=00010283
ntdll!RtlAllocateHeap+0×9e3:
7c832335 8906 mov   dword ptr [esi],eax   ds:0023:0008019d={(AppA+0×2)
(01000002)}
```

```
0:003> kL
  *** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
00b3eeec 77e67319 ntdll!RtlAllocateHeap+0×9e3
00b3ef50 77e67690 kernel32!BasepComputeProcessPath+0xc2
00b3ef90 77e41b95 kernel32!BaseComputeProcessDllPath+0xe3
00b3eff0 77e67b77 kernel32!LoadLibraryExW+0×14e
00b3f004 7406615c kernel32!LoadLibraryW+0×11
00b3f020 74066263 AppA!ProcessItem+0×2d
...
```

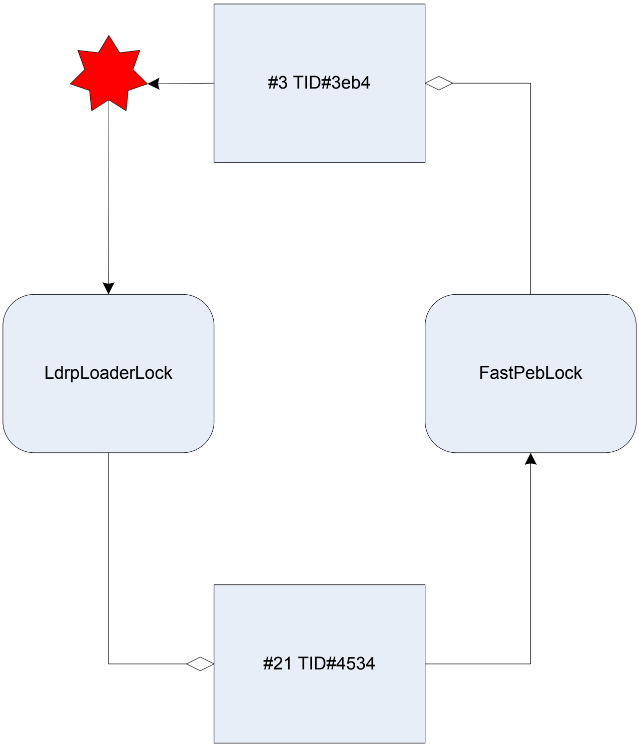
Seems the address pointed by ESI is valid: [0008019d] = 01000002. However this is a write MOV instruction but the accessed memory page is read-only:

```
0:003> !address 01000002
    01000000 : 01000000 - 00001000
                    Type     01000000 MEM_IMAGE
                    Protect  00000002 PAGE_READONLY
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageImage
                    FullPath ...AppA.exe
```

Without heap corruption the normal course of action can be depicted on the following diagram (**Wait Chain** pattern applied to critical sections, Volume 1, page 490):



However the exception changes the picture. The course of execution is deflected to the loader again and the loop is closed. We have a classical deadlock:



HEAP AND SPIKE

This is a case study showing how different patterns interact. It was reported that the service was spiking CPU (see **Spiking Thread** pattern, Volume 1, page 305) and other processes couldn't communicate with it (see **Coupled Processes** pattern, Volume 1, page 419). The dump was saved using userdump.exe and **!runaway** command showed that the thread #18 was consuming CPU mostly in user mode:

```
0:018> !runaway 0y11
 User Mode Time
  Thread       Time
  18:6080      0 days 20:46:57.156
   3:2838      0 days 0:00:00.203
   5:53a0      0 days 0:00:00.093
  16:44a8      0 days 0:00:00.078
  17:2ad8      0 days 0:00:00.046
  19:5834      0 days 0:00:00.015
...
   0:57d4      0 days 0:00:00.000
 Kernel Mode Time
  Thread       Time
   5:53a0      0 days 0:00:03.328
   3:2838      0 days 0:00:01.046
  16:44a8      0 days 0:00:00.937
  18:6080      0 days 0:00:00.765
  17:2ad8      0 days 0:00:00.531
  19:5834      0 days 0:00:00.171
  20:3174      0 days 0:00:00.140
...
```

Its thread stack revealed runtime heap manipulation function on top of it:

```
0:018> ~18kL
ChildEBP RetAddr
0207f748 77e673ca ntdll!RtlFreeHeap+0×4a5
0207f7b0 77e67690 kernel32!BasepComputeProcessPath+0×395
0207f7b0 77e67690 kernel32!BaseComputeProcessDllPath+0xe3
0207f7f0 77e41b95 kernel32!BaseComputeProcessDllPath+0xe3
0207f850 710f43d9 kernel32!LoadLibraryExW+0×14e
...
```

The parameters to RtlFreeHeap call are described in the following MSDN article:

http://msdn.microsoft.com/en-us/library/ms796710.aspx

We see that the code was freeing the heap block with the address 0×02f88278 and it was located in the first heap 0×00090000:

```
0:018> ~18kv
ChildEBP RetAddr  Args to Child
0207f748 77e673ca 00090000 00000000 02f88278 ntdll!RtlFreeHeap+0×4a5
...

0:018> !heap
Index   Address  Name      Debugging options enabled
  1:   00090000
  2:   00190000
  3:   003b0000
  4:   00340000
  5:   00380000
  6:   007d0000
  7:   008e0000
  8:   00a40000
  9:   00ae0000
 10:   00c60000
 11:   00cf0000
 12:   00d30000
 13:   00d40000
 14:   00ed0000
 15:   01010000
 16:   01090000
 17:   00a20000
 18:   00c90000
 19:   00fd0000
 20:   01690000
 21:   01b30000
 22:   01b50000
 23:   01c20000
 24:   01e50000
 25:   01f20000
 26:   01f30000
 27:   01f40000
```

We can ask a question: could it be the case that the heap was corrupt? We can validate it:

```
0:018> !heap -s 00090000
Walking the heap 00090000 .HEAP 00090000 (Seg 00090640) At 00184998 Error:
invalid block size
```

```
.List corrupted: (Blink->Flink = 00000000) != (Block = 02f88278)
HEAP 00090000 (Seg 02f80000) At 02f88270 Error: block list entry corrupted

...
```

The output shows our block 0×02f88278 too but this could also be the effect of false positive corruption, the fact that the heap control structures had not been updated yet: the thread is inside heap manipulation function.

When looking at critical sections and other threads we also see several wait chains (see **Wait Chain** pattern, Volume 1, page 490):

```
0:000> !locks

CritSec ntdll!LdrpLoaderLock+0 at 7c889d94
WaiterWoken        No
LockCount          18
RecursionCount     1
OwningThread       2300
EntryCount         0
ContentionCount    141
*** Locked

CritSec ntdll!FastPebLock+0 at 7c889d40
WaiterWoken        No
LockCount          1
RecursionCount     1
OwningThread       2904
EntryCount         0
ContentionCount    2
*** Locked

CritSec +90608 at 00090608
WaiterWoken        No
LockCount          7
RecursionCount     1
OwningThread       6080
EntryCount         0
ContentionCount    7
*** Locked

CritSec serviceA!ServiceSection+0 at 7617c340
WaiterWoken        No
LockCount          5
RecursionCount     1
OwningThread       2838
EntryCount         0
ContentionCount    1e3
*** Locked
```

Corresponding stack traces (shown in smaller font for visual clarity):

```
0:000> ~*kv

...

3 Id: da4.2838 Suspend: 1 Teb: 7ffdc000 Unfrozen
ChildEBP RetAddr Args to Child
00c4f164 7c822124 7c83970f 00000688 00000000 ntdll!KiFastSystemCallRet
00c4f168 7c83970f 00000688 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
00c4f1a4 7c839620 00000000 00000004 00090000 ntdll!RtlpWaitOnCriticalSection+0x19c
00c4f1c4 7c82fe72 00090608 000995b8 00000008 ntdll!RtlEnterCriticalSection+0xa8
00c4f3ec 7c81f4d5 00090000 00800000 000000f8 ntdll!RtlAllocateHeap+0×313
00c4f434 7c81f073 000995b8 00000000 00090000 ntdll!RtlpAllocateUserBlock+0×91
00c4f4f8 7c81ff83 00000bbe 00000000 00000001 ntdll!RtlpLowFragHeapAlloc+0×862
00c4f71c 7c820b23 00090000 00000000 0000000c ntdll!RtlAllocateHeap+0×80
00c4f734 77f6599e 00c4f7b8 00000001 00000002 ntdll!RtlAllocateAndInitializeSid+0×35
00c4f768 74062bfd 00c4f7b8 00000001 00000002 advapi32!AllocateAndInitializeSid+0×2c
...

...

18 Id: da4.6080 Suspend: 1 Teb: 7ffa8000 Unfrozen
[This is our spiking thread]
...

...

32 Id: da4.2904 Suspend: 1 Teb: 7ff9a000 Unfrozen
ChildEBP RetAddr Args to Child
0356fbfc 7c822124 7c83970f 00000688 00000000 ntdll!KiFastSystemCallRet
0356fc00 7c83970f 00000688 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
0356fc3c 7c839620 00000000 00000004 00090000 ntdll!RtlpWaitOnCriticalSection+0x19c
0356fc5c 7c82fe72 00090608 7c8329a4 0000008e ntdll!RtlEnterCriticalSection+0xa8
0356fe84 77e67319 00090000 00000000 0000056a ntdll!RtlAllocateHeap+0×313
0356fee8 77e67690 77e676bc 00020a50 00000000 kernel32!BasepComputeProcessPath+0xc2
0356ff28 77e41b95 00000000 00000000 7c82f337 kernel32!BaseComputeProcessDllPath+0xe3
0356ff88 77e67b77 74066908 00000000 00000000 kernel32!LoadLibraryExW+0×14e
0356ff9c 74071b74 74066908 74075320 74071c1c kernel32!LoadLibraryW+0×11
...

  33  Id: da4.2300 Suspend: 1 Teb: 7ff99000 Unfrozen
ChildEBP RetAddr  Args to Child
035afb40 7c822124 7c83970f 00000164 00000000 ntdll!KiFastSystemCallRet
035afb44 7c83970f 00000164 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
035afb80 7c839620 00000000 00000004 00000000 ntdll!RtlpWaitOnCriticalSection+0x19c
035afba0 7c83288c 7c889d40 77e630b8 00000000 ntdll!RtlEnterCriticalSection+0xa8
035afba8 77e630b8 00000000 007e5860 00000001 ntdll!RtlAcquirePebLock+0×11
...
```

```
18 threads -> TID#2300 -> TID#2904 -> TID#6080

5 threads -> TID#2838 -> TID#6080
```

In conclusion it looks like our spiking thread was the main problem and preventing other threads from running.

HOOKSWARE

This word describes applications heavily dependent on various hooks that are either injected by normal Windows hooking mechanism, registry or via more elaborate tricks like remote threads or code patching.

There are various patterns in memory dumps that help in detection, troubleshooting and debugging of hooksware:

- **Hooked Functions** (Volume 1, page 468)

This is the primary detection mechanism for hooks that patch code.

- **Changed Environment** (Volume 1, page 283)

Loaded hooks shift other modules by changing their load address and therefore might expose dormant bugs.

- **Insufficient Memory** (virtual memory, page **Error! Bookmark not defined.**)

Hooks loaded in the middle of address space limit the maximum amount of memory that can be allocated at once. For example, various virtual machines, like Java, reserve the big chunk of memory at start up.

- **No Component Symbols** (Volume 1, page 298)

We can get an approximate picture of what a 3rd-party hook module does by looking at its import table or in the case of patching by looking at the list of deviations returned by **.chkimg** command.

- **Unknown Component** (Volume 1, page 367)

This pattern might give an idea about the author of the hook.

- **Coincidental Symbolic Information** (Volume 1, page 390)

Sometimes hooks are loaded at round addresses like 0×10000000 and these values are very frequently used as flags or constants too.

- **Wild Code** (page 219)

When hooking goes wrong the execution path goes into the wild territory.

- **Execution Residue** (page 239)

Here we can find various hooks that use normal Windows hooking mechanism. Sometimes the search for “hook” word in symbolic raw stack output of **dds** command reveals them but beware of coincidental symbolic information. See also how to dump raw stack from process dump files (Volume 1, page 231) and complete memory dumps (Volume 1, page 236).

- **Hidden Module** (page 339)

Some hooks may hide themselves.

HEAP AND EARLY CRASH DUMP

The following error was reported when launching an application and no configured default postmortem debugger was able to save a crash dump:

```
      The application failed to initialize properly (0x06d007e). Click on
OK to terminate the application.
```

The process memory dump captured manually using userdump.exe when the error message box was displayed didn’t show anything helpful on stack traces:

```
0:000> ~*kL

.  0  Id: 310.1ab8 Suspend: 1 Teb: 7ffdf000 Unfrozen
ChildEBP RetAddr
0012fd14 7c8284c5 ntdll!_LdrpInitialize+0x184
00000000 00000000 ntdll!KiUserApcDispatcher+0x25

   1  Id: 310.1ec0 Suspend: 1 Teb: 7ffde000 Unfrozen
ChildEBP RetAddr
0820fcb0 7c826f4b ntdll!KiFastSystemCallRet
0820fcb4 7c813b90 ntdll!NtDelayExecution+0xc
0820fd14 7c8284c5 ntdll!_LdrpInitialize+0x19b
00000000 00000000 ntdll!KiUserApcDispatcher+0x25
```

However, one of last error values was access violation (**Last Error Collection** pattern, page 337):

```
0:000> !gle -all
Last error for thread 0:
LastErrorValue: (Win32) 0x3e6 (998) - Invalid access to memory location.
LastStatusValue: (NTSTATUS) 0xc0000005 - The instruction at "0x%08lx"
referenced memory at "0x%08lx". The memory could not be "%s".

Last error for thread 1:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0
```

It was suspected that access violation errors were handled by application exception handlers (**Custom Exception Handler** pattern, Volume 1, page 470) and it was recommended to catch first-chance exception (page 129) crash dumps (**Early Crash Dump** pattern, Volume 1, page 465) and indeed there was one such exception:

```
0:000> r
eax=00000000 ebx=00000000 ecx=00000000 edx=00157554 esi=00000080
edi=00000000
eip=7c829ffa esp=0012ed48 ebp=0012ef64 iopl=0 nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000  efl=00010246
ntdll!RtlAllocateHeap+0x24:
7c829ffa 0b4310  or    eax,dword ptr [ebx+10h] ds:0023:00000010=????????

0:000> kL
ChildEBP RetAddr
0012ef64 7c3416b3 ntdll!RtlAllocateHeap+0x24
0012efa4 7c3416db msvcr71!_heap_alloc+0xe0
0012efac 7c3416f8 msvcr71!_nh_malloc+0x10
0012efb8 67741c01 msvcr71!malloc+0xf
...
```

WINDBG SHORTCUTS

WINDBG AS A BINARY EDITOR

Sometimes we have a binary file or even a text file where we want to alter some bytes but we don't have any binary editor at hand. We can use WinDbg for this purpose. To illustrate this, let's create hello.bin file with "Hello World!" in its contents. Suppose we want to change it to "Hello WinDbg!". First, we need to open any available full process user dump file and then get the list of valid address ranges by either using **!address** or **lm** command:

```
0:000> lm
start             end                 module name
00000000`00400000 00000000`0044d000   TestDefaultDebugger64
00000000`77850000 00000000`77981000   kernel32
00000000`77990000 00000000`77a5a000   user32
00000000`77a60000 00000000`77bda000   ntdll
000007fe`f8940000 000007fe`f8997000   winspool
000007fe`fcb00000 000007fe`fccf0000   comctl32
000007fe`fcfc0000 000007fe`fd012000   uxtheme
000007fe`fe1d0000 000007fe`fe2d4000   msctf
000007fe`fe380000 000007fe`fe3f1000   shlwapi
000007fe`fe660000 000007fe`fe799000   rpcrt4
000007fe`fe9f0000 000007fe`feac8000   oleaut32
000007fe`fead0000 000007fe`ff704000   shell32
000007fe`ff880000 000007fe`ff91a000   usp10
000007fe`ff920000 000007fe`ff9c1000   msvcrt
000007fe`ff9d0000 000007fe`ff9f8000   imm32
000007fe`ffa00000 000007fe`ffbe0000   ole32
000007fe`ffbe0000 000007fe`ffcdf000   advapi32
000007fe`ffce0000 000007fe`ffcec000   lpk
000007fe`ffcf0000 000007fe`ffd51000   gdi32
```

Now let's choose 00000000`00400000 address. It points to the following memory data:

```
0:000> dc 00000000`00400000
00000000`00400000  00905a4d 00000003 00000004 0000ffff  MZ..............
00000000`00400010  000000b8 00000000 00000040 00000000  ........@.......
00000000`00400020  00000000 00000000 00000000 00000000  ................
00000000`00400030  00000000 00000000 00000000 000000e8  ................
00000000`00400040  0eba1f0e cd09b400 4c01b821 685421cd  ........!..L.!Th
00000000`00400050  70207369 72676f72 63206d61 6f6e6e61  is program canno
00000000`00400060  65622074 6e757220 206e6920 20534f44  t be run in DOS
00000000`00400070  65646f6d 0a0d0d2e 00000024 00000000  mode....$.......
```

Next we load our hello.bin by specifying the this address and the number of bytes to load:

```
0:000> .readmem c:\dmitri\hello.bin 00000000`00400000 L0n12
Reading c bytes.
```

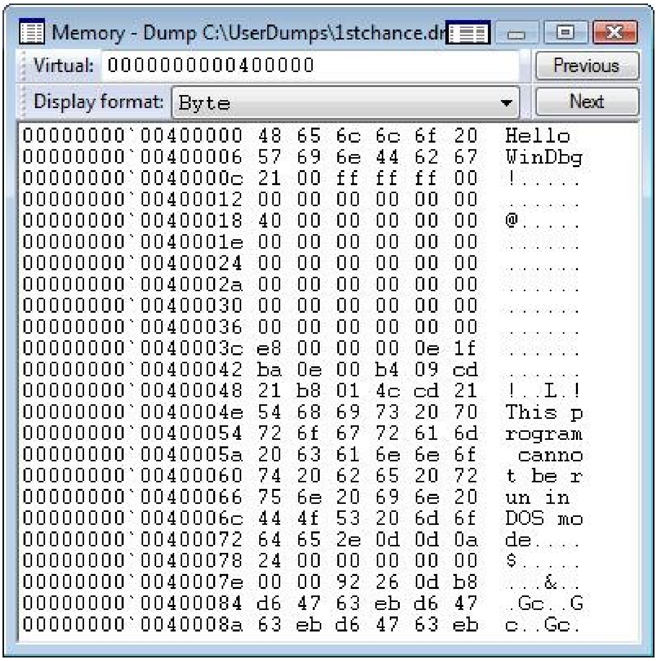
We see the new memory data immediately:

```
0:000> dc 00000000`00400000
00000000`00400000  6c6c6548 6f57206f 21646c72 0000ffff  Hello World!….
00000000`00400010  000000b8 00000000 00000040 00000000  ……..@…….
00000000`00400020  00000000 00000000 00000000 00000000  …………….
00000000`00400030  00000000 00000000 00000000 000000e8  …………….
00000000`00400040  0eba1f0e cd09b400 4c01b821 685421cd  ……..!..L.!Th
00000000`00400050  70207369 72676f72 63206d61 6f6e6e61  is program canno
00000000`00400060  65622074 6e757220 206e6920 20534f44  t be run in DOS
00000000`00400070  65646f6d 0a0d0d2e 00000024 00000000  mode….$…….
```

Then we can change it immediately using any of **e*** commands:

```
0:000> ea 00000000`00400000+6 "WinDbg!"

0:000> dc 00000000`00400000
00000000`00400000  6c6c6548 6957206f 6762446e 2100ffff  Hello WinDbg!…
00000000`00400010  000000b8 00000000 00000040 00000000  ……..@…….
00000000`00400020  00000000 00000000 00000000 00000000  …………….
00000000`00400030  00000000 00000000 00000000 000000e8  …………….
00000000`00400040  0eba1f0e cd09b400 4c01b821 685421cd  ……..!..L.!Th
00000000`00400050  70207369 72676f72 63206d61 6f6e6e61  is program canno
00000000`00400060  65622074 6e757220 206e6920 20534f44  t be run in DOS
00000000`00400070  65646f6d 0a0d0d2e 00000024 00000000  mode….$…….
```

Alternatively we can use GUI memory editor:



Virtual: 0000000000400000

Display format: Byte

```
00000000`00400000 48 65 6c 6c 6f 20  Hello
00000000`00400006 57 69 6e 44 62 67  WinDbg
00000000`0040000c 21 00 ff ff ff 00  !.....
00000000`00400012 00 00 00 00 00 00  ......
00000000`00400018 40 00 00 00 00 00  @.....
00000000`0040001e 00 00 00 00 00 00  ......
00000000`00400024 00 00 00 00 00 00  ......
00000000`0040002a 00 00 00 00 00 00  ......
00000000`00400030 00 00 00 00 00 00  ......
00000000`00400036 00 00 00 00 00 00  ......
00000000`0040003c e8 00 00 00 0e 1f  ......
00000000`00400042 ba 0e 00 b4 09 cd  ......
00000000`00400048 21 b8 01 4c cd 21  !..L.!
00000000`0040004e 54 68 69 73 20 70  This p
00000000`00400054 72 6f 67 72 61 6d  rogram
00000000`0040005a 20 63 61 6e 6e 6f   canno
00000000`00400060 74 20 62 65 20 72  t be r
00000000`00400066 75 6e 20 69 6e 20  un in
00000000`0040006c 44 4f 53 20 6d 6f  DOS mo
00000000`00400072 64 65 2e 0d 0d 0a  de....
00000000`00400078 24 00 00 00 00 00  $.....
00000000`0040007e 00 00 92 26 0d b8  ...&..
00000000`00400084 d6 47 63 eb d6 47  .Gc..G
00000000`0040008a 63 eb d6 47 63 eb  c..Gc.
```

Finally, we can write memory contents back to our file:

```
0:000> .writemem c:\dmitri\hello.bin 00000000`00400000 L0n13
Writing d bytes.
```

COMMAND AUTOCOMPLETION

This is a useful feature in WinDbg. For example, just type **!a**<TAB>. No longer we need to type **!analyze** command fully. As by product, we can see the existence of **!analyzeuexception** command which seems identical to **!analyze** command at least for user dumps.

!ENVVAR

We can check a computer name in various memory dump types (Volume 1, page 616) and there is a shortcut for process memory dumps using WinDbg command **!envvar:**

```
0:003> !envvar COMPUTERNAME
        COMPUTERNAME = MYHOMEPC
```

Of course, we can use it for any other variable. It also works for complete memory dumps but we need to set the appropriate process context first:

```
3: kd> !envvar PATH
        PATH = C:\WINDOWS\system32;C:\WINDOWS;...
```

.QUIT_LOCK

If we have 10-20 or more simultaneously opened debugging sessions there are chances that we can quit the wrong one accidentally or by mistake. Then we have to repeat past commands if we forgot to open a log file. To alleviate this there is a special WinDbg meta-command that prevents us from such accidents:

"**.quit_lock** command sets a password to prevent you from accidentally ending the debugging session" (http://msdn.microsoft.com/en-us/library/cc266836.aspx).

Here is an example:

```
0:001> .quit_lock
No quit lock

0:001> .quit_lock /s "password"
Quit lock string is 'password'

0:001> q
.quit_lock -q required to unlock 'q'

0:001> .quit_lock -q "password"
Quit lock removed
```

.DUMPCAB

Suppose we are debugging a process and we want to send its memory dump to another engineer, perhaps in a different company. We also use some symbol files that are not available on public symbol servers or our dump is a minidump that requires certain images to be loaded too. Then we can use **.dumpcab** WinDbg command to save a dump in a CAB file together with necessary symbols and images. We can only do it when your debugging target is a dump file. If we are debugging a live process we need to save a dump file first:

```
0:000> .dump /ma c:\UserDumps\notepad.dmp
Creating c:\UserDumps\notepad.dmp - mini user dump
Dump successfully written
```

Then we open the dump file and create a CAB file from it:

```
Loading Dump File [C:\UserDumps\notepad.dmp]
User Mini Dump File with Full Memory: Only application data is available

...

0:001> .dumpcab -a c:\UserDumps\notepad.cab
Creating a cab file can take a VERY VERY long time
.Ctrl-C can only interrupt the command after a file has been added to the
cab.
  Adding C:\UserDumps\notepad.dmp - added
  Adding c:\mss\ntdll.pdb\B958B2F91A5A46B889DAFAB4D140CF252\ntdll.pdb -
added
Wrote c:\UserDumps\notepad.cab
```

Additional information can be found in WinDbg help:
http://msdn.microsoft.com/en-us/library/cc266764.aspx

Note: **.dump** WinDbg command with /ba option also saves CAB file with symbols and images included.

.F+, .F-

These are handy shortcuts to **.frame** command. **.f+** shifts the current frame index down the stack trace and **.f**- shifts it up towards the top. More information can be found in WinDbg help:

http://msdn.microsoft.com/en-us/library/cc409477.aspx

Here is an example from notepad process stack trace:

```
0:000> kn
 # ChildEBP RetAddr
00 001bfcfc 761ef837 ntdll!KiFastSystemCallRet
01 001bfd00 761ef86a USER32!NtUserGetMessage+0xc
02 001bfd1c 00c31418 USER32!GetMessageW+0x33
03 001bfd5c 00c3195d notepad!WinMain+0xec
04 001bfdec 76364911 notepad!_initterm_e+0x1a1
05 001bfdf8 76fde4b6 kernel32!BaseThreadInitThunk+0xe
06 001bfe38 76fde489 ntdll!__RtlUserThreadStart+0x23
07 001bfe50 00000000 ntdll!_RtlUserThreadStart+0x1b

0:000> .f+
01 001bfd00 761ef86a USER32!NtUserGetMessage+0xc

0:000> .f+
02 001bfd1c 00c31418 USER32!GetMessageW+0x33

0:000> .f+
03 001bfd5c 00c3195d notepad!WinMain+0xec

0:000> .f+
04 001bfdec 76364911 notepad!_initterm_e+0x1a1

0:000> .f-
03 001bfd5c 00c3195d notepad!WinMain+0xec

0:000> .f-
02 001bfd1c 00c31418 USER32!GetMessageW+0x33

0:000> .f-
01 001bfd00 761ef86a USER32!NtUserGetMessage+0xc

0:000> .f-
00 001bfcfc 761ef837 ntdll!KiFastSystemCallRet

0:000> .f-
         ^ Current frame index underflow '.f-'
```

.EXPTR

Some WinDbg commands are very useful as shortcuts to common debugging actions. One of them is **.exptr.**

"The **.exptr** command displays an EXCEPTION_POINTERS structure." (http://msdn.microsoft.com/en-us/library/cc266821.aspx)

When looking at hidden exceptions (Volume 1, page 271) and manual crash dumps (Volume 1, page 624) we need information from this structure and this command provides a convenient way to see both exception record and exception context in one unified output (shown in smaller font for visual clarity):

```
0:003> kv
ChildEBP RetAddr  Args to Child
00b3e11c 7c822124 7c83970f 0000004c 00000000 ntdll!KiFastSystemCallRet
00b3e120 7c83970f 0000004c 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
00b3e15c 7c839620 00000000 00000004 00000001 ntdll!RtlpWaitOnCriticalSection+0x19c
00b3e17c 7c832ad0 7c889d94 09150000 7c889e0c ntdll!RtlEnterCriticalSection+0xa8
00b3e1b0 77e68de4 00000001 00000000 00b3e1ec ntdll!LdrLockLoaderLock+0xe4
00b3e210 77e98fae 09150000 00b3e5b0 00000104 kernel32!GetModuleFileNameW+0x77
00b3e24c 77e76d5f 091518b8 00b3e5a4 00000000 kernel32!FillUEFInfo+0x51
00b3e8d4 77e84269 00b3e8fc 77e6b831 00b3e904 kernel32!UnhandledExceptionFilter+0xec
00b3e8dc 77e6b831 00b3e904 00000000 00b3e904 kernel32!BaseThreadStart+0×4a
00b3e904 7c82eeb2 00b3e9e8 00b3ffdc 00b3ea04 kernel32!_except_handler3+0×61
00b3e928 7c82ee84 00b3e9e8 00b3ffdc 00b3ea04 ntdll!ExecuteHandler2+0×26
00b3e9d0 7c82ecc6 00b38000 00b3ea04 00b3e9e8 ntdll!ExecuteHandler+0×24
00b3e9d0 7c832335 00b38000 00b3ea04 00b3e9e8 ntdll!KiUserExceptionDispatcher+0xe
(CONTEXT @ 00b3ea04)
00b3eeec 77e67319 00090000 00000000 0000056a ntdll!RtlAllocateHeap+0×9e3
...
```

```
0:003> .exptr 00b3e8fc

----- Exception record at 00b3e9e8:
ExceptionAddress: 7c832335 (ntdll!RtlAllocateHeap+0x000009e3)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 00000001
   Parameter[1]: 0008019d
Attempt to write to address 0008019d

----- Context record at 00b3ea04:
eax=0325f1f0 ebx=00000051 ecx=00090000 edx=00090400 esi=0008019d
edi=0325f1e8
eip=7c832335 esp=00b3ecd0 ebp=00b3eeec iopl=0 nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000  efl=00010283
ntdll!RtlAllocateHeap+0x9e3:
7c832335 8906   mov     dword ptr [esi],eax  ds:0023:0008019d=01000002
```

WINDBG AS A SIMPLE PE VIEWER

If we need to quickly check preferred load address for a DLL we can use WinDbg as a binary editor (page 67) and load that DLL as a crash dump:

```
Loading Dump File [C:\kktools\userdump8.1\x64\usrxcptn.dll]
Symbol search path is:
srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00000000`00400000
00000000`00406000   C:\kktools\userdump8.1\x64\usrxcptn.dll
usrxcptn!DllMainCRTStartupForGS:
00000000`00401200 4883ec28        sub     rsp,28h

0:000> lm
start             end                 module name
00000000`00400000 00000000`00406000   usrxcptn   (pdb
symbols)          c:\mss\usrxcptn.pdb\...\usrxcptn.pdb
```

lm command already shows that but we can also check formatted PE headers as well:

```
0:000> !dh 00000000`00400000

File Type: DLL
FILE HEADER VALUES
    8664 machine (X64)
       5 number of sections
45825DE6 time date stamp Fri Dec 15 08:33:42 2006

       0 file pointer to symbol table
       0 number of symbols
      F0 size of optional header
    2022 characteristics
            Executable
            App can handle >2gb addresses
            DLL

OPTIONAL HEADER VALUES
     20B magic #
    8.00 linker version
     E00 size of code
    1200 size of initialized data
       0 size of uninitialized data
    1200 address of entry point
    1000 base of code
         ----- new -----
0000000000400000 image base
    1000 section alignment
```

```
     200 file alignment
       3 subsystem (Windows CUI)
    5.02 operating system version
    5.02 image version
    5.02 subsystem version
    6000 size of image
     400 size of headers
    DA18 checksum
0000000000040000 size of stack reserve
0000000000001000 size of stack commit
0000000000100000 size of heap reserve
0000000000001000 size of heap commit
    1AB0 [     213] address [size] of Export Directory
    18B4 [      3C] address [size] of Import Directory
    4000 [     418] address [size] of Resource Directory
    3000 [      48] address [size] of Exception Directory
    1E00 [    2580] address [size] of Security Directory
    5000 [      10] address [size] of Base Relocation Directory
    1080 [      1C] address [size] of Debug Directory
       0 [       0] address [size] of Description Directory
       0 [       0] address [size] of Special Directory
       0 [       0] address [size] of Thread Storage Directory
       0 [       0] address [size] of Load Configuration Directory
       0 [       0] address [size] of Bound Import Directory
    1000 [      78] address [size] of Import Address Table Directory
       0 [       0] address [size] of Delay Import Directory
       0 [       0] address [size] of COR20 Header Directory
       0 [       0] address [size] of Reserved Directory
SECTION HEADER #1
   .text name
     CC3 virtual size
    1000 virtual address
     E00 size of raw data
     400 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         (no align specified)
         Execute Read
Debug Directories(1)
 Type       Size     Address  Pointer
 cv           25        10b0      4b0 Format: RSDS, guid, 1, usrxcptn.pdb

SECTION HEADER #2
   .data name
     744 virtual size
    2000 virtual address
     200 size of raw data
    1200 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         (no align specified)
         Read Write

SECTION HEADER #3
  .pdata name
      48 virtual size
    3000 virtual address
     200 size of raw data
    1400 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         (no align specified)
         Read Only

SECTION HEADER #4
   .rsrc name
     418 virtual size
    4000 virtual address
     600 size of raw data
    1600 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         (no align specified)
         Read Only

SECTION HEADER #5
  .reloc name
      34 virtual size
    5000 virtual address
     200 size of raw data
    1C00 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
         Initialized Data
         Discardable
         (no align specified)
         Read Only
```

.SOUND_NOTIFY

Suppose we set up breakpoints to catch a random issue or at the end of a lengthy loop and we don’t want to sit tight, stare at the screen and wait for a debugger notification event. We just want to sit relaxed and read our favorite book or do something else. Here we can use this meta-command where we can specify a wave file to be played every time a debugger breaks into a command prompt:

"The **.sound_notify** command causes a sound to be played when WinDbg enters the wait-for-command state"
(http://msdn.microsoft.com/en-us/library/cc266857.aspx).

For example:

```
(15dc.dd0): Break instruction exception - code 80000003 (first chance)
eax=7ffde000 ebx=00000000 ecx=00000000 edx=77b3d094 esi=00000000
edi=00000000
eip=77af7dfe esp=01c6fbf4 ebp=01c6fc20 iopl=0 nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000
246
ntdll!DbgBreakPoint:
77af7dfe cc              int     3

windbg> .sound_notify /ef c:\Windows\Media\tada.wav
Sound notification: file 'c:\Windows\Media\tada.wav'

0:001> g
(15dc.175c): Break instruction exception - code 80000003 (first chance)
eax=7ffde000 ebx=00000000 ecx=00000000 edx=77b3d094 esi=00000000
edi=00000000
eip=77af7dfe esp=01cafc08 ebp=01cafc34 iopl=0 nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000
246
ntdll!DbgBreakPoint:
77af7dfe cc              int     3
```

[tada.wav is played when we break into]

SIGNALED OBJECTS

Now we elaborate a bit on how to see whether the specific synchronization object has signaled or not. If that object has signaled (is in signaled state) then any wait on it will be satisfied immediately. All such objects like kernel events, processes and threads are structures in memory and have _OBJECT_HEADER structure followed by _DISPATCHER_HEADER structure and then by the actual object body (if it is necessary). These objects can be found in the output of **!process** or **!thread** commands where they are listed as being waited for, for example:

```
THREAD 88865db0  Cid 05d0.5ea8  Teb: 7ffaa000 Win32Thread: 00000000 WAIT:
(Unknown) UserMode Non-Alertable
    89a65a68  SynchronizationEvent
Not impersonating
DeviceMap                 e1536358
Owning Process            8abbdd88       Image:         svchost.exe
Wait Start TickCount      5303768        Ticks: 1132797 (0:04:54:59.953)
Context Switch Count      193
UserTime                  00:00:00.015
KernelTime                00:00:00.000
Start Address kernel32!BaseThreadStartThunk (0×7c82b5f3)
Stack Init b8ca2000 Current b8ca1c60 Base b8ca2000 Limit b8c9f000 Call 0
Priority 10 BasePriority 8 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr
b8ca1c78 80832f7a nt!KiSwapContext+0×26
b8ca1ca4 8082925c nt!KiSwapThread+0×284
b8ca1cec 80937e6a nt!KeWaitForSingleObject+0×346
b8ca1d50 80888c7c nt!NtWaitForSingleObject+0×9a
b8ca1d50 7c94ed54 nt!KiFastCallEntry+0xfc
00c6fea0 7c942124 ntdll!KiFastSystemCallRet
00c6fea4 7c95970f ntdll!NtWaitForSingleObject+0xc
00c6fee0 7c959620 ntdll!RtlpWaitOnCriticalSection+0×19c
00c6ff00 7c95c1e7 ntdll!RtlEnterCriticalSection+0xa8
00c6ffa8 7c8261d6 ntdll!LdrShutdownThread+0×33
00c6ffb8 7c826090 kernel32!ExitThread+0×2f
00c6ffec 00000000 kernel32!BaseThreadStart+0×39
```

We can see them as parameters of KeWaitForSingle(Multiple)Objects functions (shown in smaller font for visual clarity):

```
0: kd> !thread 88865db0
THREAD 88865db0  Cid 05d0.5ea8  Teb: 7ffaa000 Win32Thread: 00000000 WAIT: (Unknown)
UserMode Non-Alertable
    89a65a68  SynchronizationEvent
Not impersonating
DeviceMap                 e1536358
Owning Process            8abbdd88       Image:         svchost.exe
Wait Start TickCount      5303768        Ticks: 1132797 (0:04:54:59.953)
Context Switch Count      193
UserTime                  00:00:00.015
KernelTime                00:00:00.000
Start Address kernel32!BaseThreadStartThunk (0×7c82b5f3)
Stack Init b8ca2000 Current b8ca1c60 Base b8ca2000 Limit b8c9f000 Call 0
Priority 10 BasePriority 8 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr  Args to Child
b8ca1c78 80832f7a 88865e28 88865db0 88865e58 nt!KiSwapContext+0×26
b8ca1ca4 8082925c 00000000 00000000 00000000 nt!KiSwapThread+0×284
b8ca1cec 80937e6a 89a65a68 00000006 00000001 nt!KeWaitForSingleObject+0×346
b8ca1d50 80888c7c 00000470 00000000 00000000 nt!NtWaitForSingleObject+0×9a
b8ca1d50 7c94ed54 00000470 00000000 00000000 nt!KiFastCallEntry+0xfc
00c6fea0 7c942124 7c95970f 00000470 00000000 ntdll!KiFastSystemCallRet
00c6fea4 7c95970f 00000470 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
00c6fee0 7c959620 00000000 00000004 009171a8 ntdll!RtlpWaitOnCriticalSection+0×19c
00c6ff00 7c95c1e7 7c9a9d94 00000000 7ffaa000 ntdll!RtlEnterCriticalSection+0xa8
00c6ffa8 7c8261d6 00000000 00000000 00c6ffec ntdll!LdrShutdownThread+0×33
00c6ffb8 7c826090 00000000 00000000 00000000 kernel32!ExitThread+0×2f
00c6ffec 00000000 77c2de6d 009171a8 00000000 kernel32!BaseThreadStart+0×39
```

We can also see them as the part of other structures, for example:

```
0: kd> !irp 88f4fd28 1
Irp is active with 1 stacks 1 is current (= 0x88f4fd98)
 No Mdl: No System Buffer: Thread 8a9449a0:  Irp stack trace.
Flags = 00000800
ThreadListEntry.Flink = 8a944ba8
ThreadListEntry.Blink = 8a944ba8
IoStatus.Status = 00000000
IoStatus.Information = 00000000
RequestorMode = 00000001
Cancel = 00
CancelIrql = 0
ApcEnvironment = 00
UserIosb = 00dddf04
UserEvent = 8a033c88
Overlay.AsynchronousParameters.UserApcRoutine = 00000000
Overlay.AsynchronousParameters.UserApcContext = 00dddf04
Overlay.AllocationSize = 00000000 - 00000000
CancelRoutine = f75c815c   Npfs!NpCancelListeningQueueIrp
UserBuffer = 00000000
&Tail.Overlay.DeviceQueueEntry = 88f4fd68
Tail.Overlay.Thread = 8a9449a0
Tail.Overlay.AuxiliaryBuffer = 00000000
Tail.Overlay.ListEntry.Flink = e311f7d4
Tail.Overlay.ListEntry.Blink = e311f7d4
Tail.Overlay.CurrentStackLocation = 88f4fd98
Tail.Overlay.OriginalFileObject = 884bc8a0
Tail.Apc = 00000000
Tail.CompletionKey = 00000000
     cmd  flg cl Device   File     Completion-Context
>[  d, 0]   5  1 8a937398 884bc8a0 00000000-00000000    pending
        \FileSystem\Npfs
   Args: 00000000 00000000 00110008 00000000

0: kd> !object 8a033c88
Object: 8a033c88  Type: (8ad7a990) Event
    ObjectHeader: 8a033c70 (old version)
    HandleCount: 1  PointerCount: 3
```

They can also be found in process or kernel (System process) handle tables:

```
0: kd> !process 0 0
...
...
...
PROCESS 8a8fa8c0  SessionId: 0  Cid: 037c    Peb: 7ffd8000  ParentCid: 
04ac
    DirBase: f3b10360  ObjectTable: e1c276b8  HandleCount: 500.
    Image: MyService.exe
...
...
...

0: kd> !kdexts.handle 0 3 037c
...
...
...
0510: Object: 89237d88  GrantedAccess: 001f0fff Entry: e1cafa20
Object: 89237d88  Type: (8ad84900) Process
    ObjectHeader: 89237d70 (old version)
        HandleCount: 1  PointerCount: 2
...
...
...
```

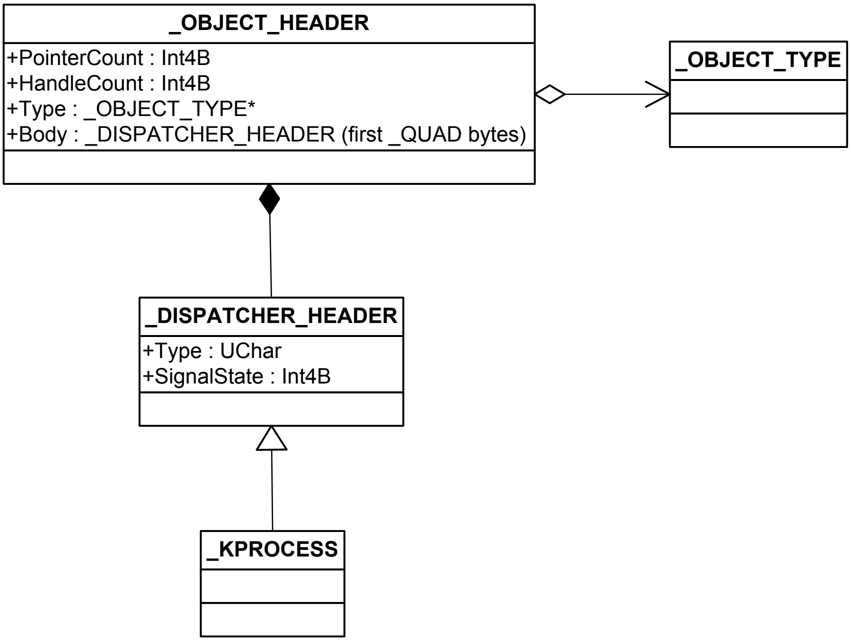
Let’s look at the last process object **89237d88**. This is the start of object-specific structure whose first member is _DISPATCHER_HEADER. Every object-specific structure is preceded in memory by _OBJECT_HEADER structure and in our case its address is ***89237d70***. To illustrate this memory layout we can write this inheritance relationship in C++ pseudo-code where I put relevant structure members together:

```
struct _KPROCESS : _DISPATCHER_HEADER
{
   // ... members
};

struct _DISPATCHER_HEADER
{
   UChar Type;
   // ... members
   Int4B SignalState;
   // ... members
};
```

```
struct _OBJECT_HEADER
{
   Int4B PointerCount;
   Int4B HandleCount;
   // ... members
   _OBJECT_TYPE *Type;
   // ... members
   _QUAD Body; // Int8B - first 8 bytes of _DISPATCHER_HEADER
};
```

HandleCount is the number of open handles and PointerCount is the number of dereferences.

We can also put all this information on the following simplified UML diagram:



Here is the detailed data for our process object:

```
0: kd> dt _OBJECT_HEADER 89237d70
ntdll!_OBJECT_HEADER
   +0x000 PointerCount     : 2
   +0x004 HandleCount      : 1
   +0x004 NextToFree       : 0x00000001
   +0x008 Type             : 0x8ad84900 _OBJECT_TYPE
   +0x00c NameInfoOffset   : 0 ''
   +0x00d HandleInfoOffset : 0 ''
   +0x00e QuotaInfoOffset  : 0 ''
   +0x00f Flags            : 0x20 ' '
   +0x010 ObjectCreateInfo : 0x808ad180 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : 0x808ad180
   +0x014 SecurityDescriptor : 0xe1002bd6
   +0x018 Body             : _QUAD

0: kd> dt _DISPATCHER_HEADER 89237d88
ntdll!_DISPATCHER_HEADER
   +0x000 Type             : 0x3 ''
   +0x001 Absolute         : 0 ''
   +0x001 NpxIrql          : 0 ''
   +0x002 Size             : 0x1e ''
   +0x002 Hand             : 0x1e ''
   +0x003 Inserted         : 0 ''
   +0x003 DebugActive      : 0 ''
   +0x000 Lock             : 1966083
   +0×004 SignalState      : 1
   +0×008 WaitListHead     : _LIST_ENTRY [ 0×89237d90 - 0×89237d90 ]

0: kd> dt _OBJECT_TYPE 0x8ad84900
ntdll!_OBJECT_TYPE
   +0x000 Mutex            : _ERESOURCE
   +0x038 TypeList         : _LIST_ENTRY [ 0x8ad84938 - 0x8ad84938 ]
   +0×040 Name             : _UNICODE_STRING “Process”
   +0×048 DefaultObject    : (null)
   +0×04c Index            : 5
   +0×050 TotalNumberOfObjects : 0×18d
   +0×054 TotalNumberOfHandles : 0×35d
   +0×058 HighWaterNumberOfObjects : 0×2d4
   +0×05c HighWaterNumberOfHandles : 0xd66
   +0×060 TypeInfo         : _OBJECT_TYPE_INITIALIZER
   +0×0ac Key              : 0×636f7250
   +0×0b0 ObjectLocks      : [4] _ERESOURCE
```

We see that SignalState is 1 and this means that the process is in signaled state (exited) and the wait on it will be satisfied immediately (WaitForSingleObject will return). If we close its handle the object will be destroyed.

If we look at other objects that threads are waiting for we would see SignalState equal to 0. For example, in the following thread below, event handle 4b0 corresponds to a34d9940 _KEVENT address and the latter object is inherited from _DISPATCHER_HEADER (shown in smaller font for visual clarity):

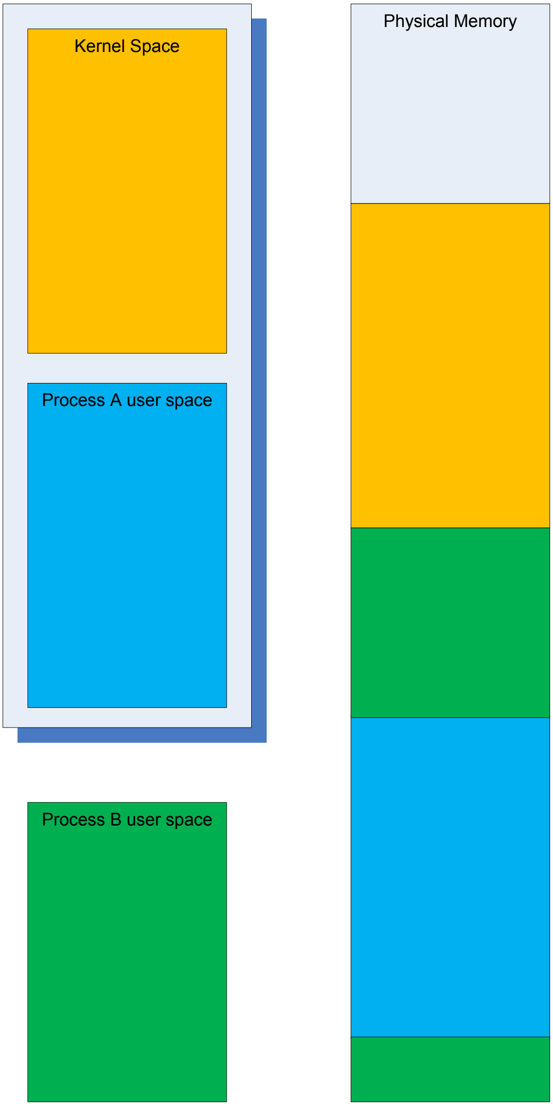
```
3: kd> !thread a34369d0
THREAD a34369d0  Cid 1fc8.1e88  Teb: 7ffae000 Win32Thread: bc6d5818 WAIT: (Unknown)
UserMode Non-Alertable
    a34d9940  SynchronizationEvent
    a3436a48  NotificationTimer
Not impersonating
DeviceMap                 e12256a0
Owning Process            a3340a10       Image:         IEXPLORE.EXE
Wait Start TickCount      1450409        Ticks: 38 (0:00:00:00.593)
Context Switch Count      7091                 LargeStack
UserTime                  00:00:01.015
KernelTime                00:00:02.250
Win32 Start Address mshtml!CExecFT::StaticThreadProc (0×7fab1061)
Start Address kernel32!BaseThreadStartThunk (0×77e617ec)
Stack Init f252b000 Current f252ac60 Base f252b000 Limit f2528000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr  Args to Child
f252ac78 80833465 a34369d0 a3436a78 00000003 nt!KiSwapContext+0×26
f252aca4 80829a62 00000000 f252ad14 00000000 nt!KiSwapThread+0×2e5
f252acec 80938d0c a34d9940 00000006 f252ac01 nt!KeWaitForSingleObject+0×346
f252ad50 8088978c 000004b0 00000000 f252ad14 nt!NtWaitForSingleObject+0×9a
f252ad50 7c8285ec 000004b0 00000000 f252ad14 nt!KiFastCallEntry+0xfc (TrapFrame @
f252ad64)
030dff08 7c827d0b 77e61d1e 000004b0 00000000 ntdll!KiFastSystemCallRet
030dff0c 77e61d1e 000004b0 00000000 030dff50 ntdll!NtWaitForSingleObject+0xc
030dff7c 77e61c8d 000004b0 000927c0 00000000 kernel32!WaitForSingleObjectEx+0xac
030dff90 7fab08a3 000004b0 000927c0 00000000 kernel32!WaitForSingleObject+0×12
030dffa8 7fab109c 00000000 7fab106e 030dffec mshtml!CDwnTaskExec::ThreadExec+0xae
030dffb0 7fab106e 030dffec 77e64829 02714f30 mshtml!CExecFT::ThreadProc+0×28
030dffb8 77e64829 02714f30 00000000 00000000 mshtml!CExecFT::StaticThreadProc+0xd
030dffec 00000000 7fab1061 02714f30 00000000 kernel32!BaseThreadStart+0×34
```

```
3: kd> dt _DISPATCHER_HEADER a34d9940
cutildll!_DISPATCHER_HEADER
   +0x000 Type             : 0x1 ''
   +0x001 Absolute         : 0xda ''
   +0x002 Size             : 0x4 ''
   +0x003 Inserted         : 0xa3 ''
   +0x003 DebugActive      : 0xa3 ''
   +0x000 Lock             : -1559963135
   +0×004 SignalState      : 0
   +0×008 WaitListHead     : _LIST_ENTRY [ 0xa3436a78 - 0xa3436a78 ]
```

For real life example of signaled process objects please see **Zombie Processes** pattern case study (page 196).

MEMORY SEARCH REVISITED

The **s** memory search WinDbg command is well-known but it only searches the current virtual memory range. This is fine if we have a kernel or a user memory dump file where we have the uniform virtual space but for complete memory dump files we have many virtual to physical address mappings to cover user space of various processes. This is better illustrated on the following highly simplified picture where the typical current virtual space in a complete memory dump is enclosed in the shadowed box and consists of the kernel space and process A virtual address space ranges:



Therefore we see that some data is present in physical memory but not accessible through virtual memory search. To search through all physical memory we can use **!search** WinDbg command and by default it searches for specific 32-bit value on 32-bit Windows and 64-bit value on 64-bit Windows. The latter means that we can search for 8 character string fragments on 64-bit Windows. For example, if we want to search for occurrences of ”ImaSrv.exe” string we can specify “ImaS” on 32-bit platform and “ImaSrv.e” on 64-bit platform. Taking into account little endian byte ordering we get the following hexadecimal equivalents:

```
0: kd> .formats 'SamI'
Evaluate expression:
  Hex:      53616d49
  Decimal: 1398893897
  Octal:   12330266511
  Binary:  01010011 01100001 01101101 01001001
  Chars:   SamI
  Time:    Wed Apr 30 22:38:17 2014
  Float:   low 9.68201e+011 high 0
  Double:  6.91145e-315

0: kd> .formats 'e.vrSamI'
Evaluate expression:
  Hex:      652e7672`53616d49
  Decimal: 7290895080156654921
  Octal:   0624563547112330266511
  Binary:  01100101 00101110 01110110 01110010 01010011 01100001 01101101
01001001
  Chars:   e.vrSamI
  Time:    Mon Dec  5 23:20:15.665 24704 (GMT+0)
  Float:   low 9.68201e+011 high 5.14923e+022
  Double:  2.46885e+179
```

Physical memory search gives us plenty of results:

```
0: kd> !search 53616d49
Searching PFNs in range 00000001 - 0013FFFF for [53616D49 - 53616D49]

Pfn      Offset   Hit      Va       Pte
- - - - - - - - - - - - - - - - - - - - - - - - - - -
000079FA 000004C4 53616D49 A3AFB4C4 C051D7D8
 a3afb340+0x184   : Proc (Protected)  -- Process objects
00011442 00000848 53616D49 66EC2848 C0337610
00011442 0000093C 53616D49 66EC293C C0337610
0001328A 000009F4 43616D49 672E59F4 C0339728
000156F6 000009F4 43616D49 672E59F4 C0339728
00018C7C 000009DC 43616D49 671F49DC C0338FA0
0001ADF0 000003D4 52616D49 00000000 DC3E3B48
0001ADF0 000003E4 52616D49 00000000 DC3E3B48
00020BCE 000009DC 43616D49 671F49DC C0338FA0
...
...
...
```

We can dump either a virtual address if it is available and valid by using normal **d*** commands or dump a physical address by using their **!d*** extension equivalents, for example:

```
Pfn      Offset   Hit      Va       Pte
- - - - - - - - - - - - - - - - - - - - - - - - - - -
00011442 00000848 53616D49 66EC2848 C0337610

 0: kd> !dc 11442000+00000848
#11442848 53616d49 65747379 6c642e6d 0000006c ImaSystem.dll...
#11442858 00000000 00000000 00000000 45cd0da0 ...............E
#11442868 00000000 0000293c 00000001 00000012 ....<)..........
#11442878 00000012 00002888 000028d0 00002918 .....(...(...)..
#11442888 00001160 000010b0 00001020 00001000 `....... .......
#11442898 00001130 00001010 000010a0 00001030 0...........0...
#114428a8 000020d8 000020cc 00001300 000013c0 . ... ..........
#114428b8 00001200 00001050 00001040 00001080 ....P...@.......
```

Note: Physical addresses are formed from PFN (Page Frame Numbers) by shifting them to the left by 12 bits (by adding 3 zeroes to the left). For example: 00011442 -> 11442000.

The cool feature of **!search** command is that it automatically recognizes pool tags and in our case it has found the process object:

```
Pfn      Offset   Hit      Va       Pte
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
000079FA 000004C4 53616D49 A3AFB4C4 C051D7D8
 a3afb340+0×184   : Proc (Protected)  — Process objects
...
...
...

0: kd> !pool A3AFB4C4
Pool page a3afb340 region is Nonpaged pool
 a3afb000 size:  120 previous size:    0  (Allocated)  MQAC
 a3afb120 size:   20 previous size:  120  (Allocated)  VadS
 a3afb140 size:   98 previous size:   20  (Allocated)  File (Protected)
 a3afb1d8 size:   30 previous size:   98  (Allocated)  MQAC
 a3afb208 size:   20 previous size:   30  (Allocated)  VadS
 a3afb228 size:   28 previous size:   20  (Free)       CcBc
 a3afb250 size:   30 previous size:   28  (Allocated)  Vad
 a3afb280 size:   30 previous size:   30  (Allocated)  Vad
 a3afb2b0 size:   20 previous size:   30  (Allocated)  VadS
 a3afb2d0 size:   40 previous size:   20  (Allocated)  SeTd
 a3afb310 size:   30 previous size:   40  (Allocated)  Vad
*a3afb340 size:  298 previous size:   30  (Allocated) *Proc (Protected)
  Pooltag Proc : Process objects, Binary : nt!ps
 a3afb5d8 size:    8 previous size:  298  (Free)       Irp
 a3afb5e0 size:   20 previous size:    8  (Allocated)  VadS
 a3afb600 size:    8 previous size:   20  (Free)       Irp
 a3afb608 size:   30 previous size:    8  (Allocated)  Even (Protected)
 a3afb638 size:   30 previous size:   30  (Allocated)  Vad
 a3afb668 size:   40 previous size:   30  (Allocated)  Vadl
 a3afb6a8 size:   70 previous size:   40  (Allocated)  NWFS
 a3afb718 size:   30 previous size:   70  (Allocated)  Vad
 a3afb748 size:   28 previous size:   30  (Allocated)  NpFr Process:
a3afb360
 a3afb770 size:   98 previous size:   28  (Allocated)  File (Protected)
 a3afb808 size:   60 previous size:   98  (Allocated)  MmCa
 a3afb868 size:   98 previous size:   60  (Allocated)  File (Protected)
 a3afb900 size:   30 previous size:   98  (Allocated)  Even (Protected)
 a3afb930 size:   20 previous size:   30  (Allocated)  VadS
 a3afb950 size:   98 previous size:   20  (Allocated)  File (Protected)
 a3afb9e8 size:   30 previous size:   98  (Allocated)  MQAC
 a3afba18 size:  120 previous size:   30  (Allocated)  MQAC
 a3afbb38 size:   50 previous size:  120  (Allocated)  NpFc Process:
a5659d88
 a3afbb88 size:   20 previous size:   50  (Allocated)  Port
 a3afbba8 size:   98 previous size:   20  (Allocated)  File (Protected)
 a3afbc40 size:   30 previous size:   98  (Allocated)  MQAC
 a3afbc70 size:  120 previous size:   30  (Allocated)  MQAC
 a3afbd90 size:  270 previous size:  120  (Allocated)  Thre (Protected)
```

```
0: kd> dc A3AFB4C4 l10
a3afb4c4  53616d49 652e7672 00006578 00000000  ImaSrv.exe......
a3afb4d4  00000000 00000000 00000000 a3afbfd4  ................
a3afb4e4  a282fe44 00000000 f7a60500 0000004c  D...........L...
a3afb4f4  001f07fb 00008005 00000000 7ffdd000  ................
```

Seems the search has found ImageFileName field of _EPROCESS structure. The field has 0×164 offset and we can dump the whole structure:

```
0: kd> dt _EPROCESS A3AFB4C4-0x164
ntdll!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x078 ProcessLock      : _EX_PUSH_LOCK
   +0x080 CreateTime       : _LARGE_INTEGER 0x1c86f66`5b95204a
   +0x088 ExitTime         : _LARGE_INTEGER 0x0
   +0x090 RundownProtect   : _EX_RUNDOWN_REF
   +0x094 UniqueProcessId  : 0x000009d0
   +0x098 ActiveProcessLinks : _LIST_ENTRY [ 0xa3af2598 - 0xa3b0a0b8 ]
   +0x0a0 QuotaUsage       : [3] 0x17030
   +0x0ac QuotaPeak        : [3] 0x18028
   +0x0b8 CommitCharge     : 0x37b6
   +0x0bc PeakVirtualSize  : 0x132d4000
   +0x0c0 VirtualSize      : 0x12d64000
   +0x0c4 SessionProcessLinks : _LIST_ENTRY [ 0xa3af25c4 - 0xa3b0a0e4 ]
   +0x0cc DebugPort        : (null)
   +0x0d0 ExceptionPort    : 0xd738e828
   +0x0d4 ObjectTable      : 0xdc23a008 _HANDLE_TABLE
   +0x0d8 Token            : _EX_FAST_REF
   +0x0dc WorkingSetPage   : 0x11741f
   +0x0e0 AddressCreationLock : _KGUARDED_MUTEX
   +0x100 HyperSpaceLock   : 0
   +0x104 ForkInProgress   : (null)
   +0x108 HardwareTrigger  : 0
   +0x10c PhysicalVadRoot  : 0xa3c21448 _MM_AVL_TABLE
   +0x110 CloneRoot        : (null)
   +0x114 NumberOfPrivatePages : 0x318d
   +0x118 NumberOfLockedPages : 7
   +0x11c Win32Process     : 0xbc33d968
   +0x120 Job              : (null)
   +0x124 SectionObject    : 0xdc2710c8
   +0x128 SectionBaseAddress : 0x00400000
   +0x12c QuotaBlock       : 0xa3bb2f38 _EPROCESS_QUOTA_BLOCK
   +0x130 WorkingSetWatch  : (null)
   +0x134 Win32WindowStation : 0x00000078
   +0x138 InheritedFromUniqueProcessId : 0x00000228
   +0x13c LdtInformation   : (null)
   +0x140 VadFreeHint      : (null)
   +0x144 VdmObjects       : (null)
   +0x148 DeviceMap        : 0xd7d0ff30
   +0x14c Spare0           : [3] (null)
   +0x158 PageDirectoryPte : _HARDWARE_PTE_X86
   +0x158 Filler           : 0
```

```
+0x160 Session          : 0xf79d5000
+0×164 ImageFileName    : [16]  “ImaSrv.exe”
+0×174 JobLinks         : _LIST_ENTRY [ 0×0 - 0×0 ]
+0×17c LockedPagesList  : (null)
+0×180 ThreadListHead   : _LIST_ENTRY [ 0xa3afbfd4 - 0xa282fe44 ]
+0×188 SecurityPort     : (null)
+0×18c PaeTop           : 0xf7a60500
+0×190 ActiveThreads    : 0×4c
+0×194 GrantedAccess    : 0×1f07fb
+0×198 DefaultHardErrorProcessing : 0×8005
+0×19c LastThreadExitStatus : 0
+0×1a0 Peb              : 0×7ffdd000 _PEB
+0×1a4 PrefetchTrace    : _EX_FAST_REF
+0×1a8 ReadOperationCount : _LARGE_INTEGER 0xf01d3
+0×1b0 WriteOperationCount : _LARGE_INTEGER 0×3b08c
+0×1b8 OtherOperationCount : _LARGE_INTEGER 0×67845
+0×1c0 ReadTransferCount : _LARGE_INTEGER 0×9b087eec
+0×1c8 WriteTransferCount : _LARGE_INTEGER 0×39f8a27a
+0×1d0 OtherTransferCount : _LARGE_INTEGER 0×25dd749
+0×1d8 CommitChargeLimit : 0
+0×1dc CommitChargePeak : 0×394b
+0×1e0 AweInfo          : (null)
+0×1e4 SeAuditProcessCreationInfo : _SE_AUDIT_PROCESS_CREATION_INFO
+0×1e8 Vm               : _MMSUPPORT
+0×230 MmProcessLinks   : _LIST_ENTRY [ 0xa3af2730 - 0xa3b0a250 ]
+0×238 ModifiedPageCount : 0xc835
+0×23c JobStatus        : 0
+0×240 Flags            : 0×4d0801
+0×240 CreateReported   : 0y1
+0×240 NoDebugInherit   : 0y0
+0×240 ProcessExiting   : 0y0
+0×240 ProcessDelete    : 0y0
+0×240 Wow64SplitPages  : 0y0
+0×240 VmDeleted        : 0y0
+0×240 OutswapEnabled   : 0y0
+0×240 Outswapped       : 0y0
+0×240 ForkFailed       : 0y0
+0×240 Wow64VaSpace4Gb  : 0y0
+0×240 AddressSpaceInitialized : 0y10
+0×240 SetTimerResolution : 0y0
+0×240 BreakOnTermination : 0y0
+0×240 SessionCreationUnderway : 0y0
+0×240 WriteWatch       : 0y0
+0×240 ProcessInSession : 0y1
+0×240 OverrideAddressSpace : 0y0
+0×240 HasAddressSpace  : 0y1
+0×240 LaunchPrefetched : 0y1
+0×240 InjectInpageErrors : 0y0
+0×240 VmTopDown        : 0y0
+0×240 ImageNotifyDone  : 0y1
+0×240 PdeUpdateNeeded  : 0y0
+0×240 VdmAllowed       : 0y0
+0×240 SmapAllowed      : 0y0
+0×240 CreateFailed     : 0y0
```

```
   +0×240 DefaultIoPriority : 0y000
   +0×240 Spare1            : 0y0
   +0×240 Spare2            : 0y0
   +0×244 ExitStatus        : 259
   +0×248 NextPageColor     : 0xf91e
   +0×24a SubSystemMinorVersion : 0 ”
   +0×24b SubSystemMajorVersion : 0×4 ”
   +0×24a SubSystemVersion : 0×400
   +0×24c PriorityClass     : 0×6 ”
   +0×250 VadRoot           : _MM_AVL_TABLE
   +0×270 Cookie            : 0×5a583219
```

We can also see that by default **!search** command finds entries differing in a single bit, for example:

```
Pfn      Offset   Hit      Va       Pte
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
...
...
...
00011442 0000093C 53616D49 66EC293C C0337610
0001328A 000009F4 43616D49 672E59F4 C0339728
...
...

0: kd> !dc 0001328A9F4
#1328a9f4 43616d49 6f6d6d6f 64702e6e 00000062 ImaCommon.pdb...
#1328aa04 672e7170 00000000 00000000 ffffffff pq.g............
#1328aa14 00000000 00000000 672e5a04 00000000 .........Z.g....
#1328aa24 00000000 00000000 00000001 672e5a1c .............Z.g
#1328aa34 00000000 00000000 00000000 672e7170 ............pq.g
#1328aa44 672e5a24 00000000 00000000 00004c24 $Z.g........$L..
#1328aa54 00000000 00000000 00000000 00000000 ................
#1328aa64 00000000 672e7138 00000000 ffffffff ....8q.g........
```

In the case of kernel memory dump file physical memory search can be better alternative to virtual memory search if we need to see pool tags corresponding to search hits or search for data differing in some bits, for example:

```
3: kd> .ignore_missing_pages 1
Suppress kernel summary dump missing page error message

3: kd> s -d 80000000 L?20000000 53616d49
86a6eeec  53616d49 652e7672 00006578 00000000  ImaSrv.exe......
```

```
3: kd> !search 53616d49
Debuggee is a kernel summary dump, some physical pages may not be present.
Searches will miss hits from those pages.
Searching PFNs in range 00000001 - 0007FFFF for [53616D49 - 53616D49]

Pfn      Offset   Hit      Va       Pte
- - - - - - - - - - - - - - - - - - - - - - - - - - -
00005DED 00000EEC 53616D49 86A6EEEC C021A9B8
 86a6ed68+0x184   : Proc (Protected)  -- Process objects
Search done.
```

WDF AND PNP BSOD: CASE STUDY

Let's look at the following bugcheck happened on my home computer:

```
IRQL_NOT_LESS_OR_EQUAL (a)
An attempt was made to access a pageable (or completely invalid) address
at an interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If a kernel debugger is available get the stack backtrace.
Arguments:
Arg1: a112883e, memory referenced
Arg2: 0000001b, IRQL
Arg3: 00000000, bitfield :
 bit 0 : value 0 = read operation, 1 = write operation
 bit 3 : value 0 = not an execute operation, 1 = execute operation (only
on chips which support this level of status)
Arg4: 81c28750, address which referenced memory

READ_ADDRESS:  a112883e Paged pool
```

The address belongs to paged pool indeed:

```
0: kd> !pool a112883e
Pool page a112883e region is Paged pool
 a1128000 size:  6d0 previous size:    0  (Allocated)  Toke (Protected)
 a11286d0 size:    8 previous size:  6d0  (Free)       SeSd
 a11286d8 size:   a8 previous size:    8  (Allocated)  SpSy
 a1128780 size:   10 previous size:   a8  (Free)       AlEB
*a1128790 size:  1a0 previous size:   10  (Allocated) *KFlt
   Owning component : Unknown (update pooltag.txt)
 a1128930 size:  6d0 previous size:  1a0  (Allocated)  Toke (Protected)
```

Search for KFlt tag points to KbdMagic.sys:

```
C:\Windows\system32>findstr /S /m /l hKFlt *.sys
drivers\KbdMagic.sys
DriverStore\FileRepository\KbdMagic.inf_c8736569\KbdMagic.sys
```

When we look at the trap address we notice that it seems to be valid:

```
TRAP_FRAME:  85bdf8e8 -- (.trap 0xffffffff85bdf8e8)
ErrCode = 00000000
eax=a1128828 ebx=00000001 ecx=81d323c0 edx=00000000 esi=84ca6f38
edi=84ca6f40
eip=81c28750 esp=85bdf95c ebp=85bdf970 iopl=0 nv up ei pl zr na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000  efl=00010246
nt!KeSetEvent+0x4d:
81c28750 385816    cmp     byte ptr [eax+16h],bl      ds:0023:a112883e=01
```

However, as explained in **Another Look at Page Faults** article (Volume 1, page 132) we have a page in transition and this violates IRQL contract:

```
0: kd> !pte a112883e
               VA a112883e
PDE at 00000000C0602840    PTE at 00000000C0508940
contains 000000001CEC5863  contains 000000001AFB48C2
pfn 1cec5 ---DA--KWEV                           not valid
                       Transition: 1afb4
                       Protect: 6 - ReadWriteExecute
```

When we look at stack trace parameters we notice that the first parameter passed to KeSetEvent function belongs to nonpaged pool:

```
STACK_TEXT:
85bdf8e8 81c28750 badb0d00 00000000 00000000 nt!KiTrap0E+0x2ac
85bdf970 876394df 84ca6f0000000000 00000000 nt!KeSetEvent+0×4d
WARNING: Stack unwind information not available. Following frames may be
wrong.
85bdf98c 8763a145 84ca68a0 8399e3b8 85bdf9ac KbdMagic+0×14df
85bdf99c 806f57a0 7b359920 7c98c800 85bdf9d4 KbdMagic+0×2145
85bdf9ac 806f514e 8399e3b8 8070e2a0 8399e3b8
Wdf01000!FxPkgPnp::PnpEventFailedOwnHardware+0×3b

0: kd> !pool 84ca6f00
Pool page 84ca6f00 region is Nonpaged pool
 84ca6000 size:  2b8 previous size:    0  (Allocated)  Thre (Protected)
 84ca62b8 size:   10 previous size:  2b8  (Free)       ....
 84ca62c8 size:   48 previous size:   10  (Allocated)  Vadl
 84ca6310 size:   30 previous size:   48  (Allocated)  Ntfn
 84ca6340 size:   38 previous size:   30  (Allocated)  usbp
 84ca6378 size:   98 previous size:   38  (Allocated)  NDam
 84ca6410 size:  188 previous size:   98  (Allocated)  NDoa
 84ca6598 size:    8 previous size:  188  (Free)       FOCX
 84ca65a0 size:   30 previous size:    8  (Allocated)  UHUB
 84ca65d0 size:   20 previous size:   30  (Allocated)  Wnln
 84ca65f0 size:   28 previous size:   20  (Allocated)  Io
 84ca6618 size:   18 previous size:   28  (Allocated)  Ala6
 84ca6630 size:   98 previous size:   18  (Allocated)  NDam
*84ca66c8 size:  938 previous size:   98  (Allocated) *KeyM
  Owning component : Unknown (update pooltag.txt)
```

We see that it is not a pointer to a valid _KEVENT structure:

```
0: kd> dt -r _KEVENT 84ca6f00
Wdf01000!_KEVENT
   +0x000 Header           : _DISPATCHER_HEADER
      +0x000 Type             : 0xc8 ''
      +0x001 Abandoned        : 0x87 ''
      +0x001 Absolute         : 0x87 ''
      +0x001 NpxIrql          : 0x87 ''
      +0x001 Signalling       : 0x87 ''
      +0x002 Size             : 0x12 ''
      +0x002 Hand             : 0x12 ''
      +0x003 Inserted         : 0xa1 ''
      +0x003 DebugActive      : 0xa1 ''
      +0x003 DpcActive        : 0xa1 ''
      +0x000 Lock             : -1592621112
      +0x004 SignalState      : -1592621112
      +0x008 WaitListHead     : _LIST_ENTRY [ 0x40000 - 0x0 ]
         +0x000 Flink            : 0x00040000 _LIST_ENTRY
         +0x004 Blink            : (null)
```

Moreover, we see from disassembly and nonpaged pool entry contents that KeSetEvent function tried to dereference wrong WaitListHead that points to paged pool (the same pool entry that caused the bugcheck):

```
0: kd> uf nt!KeSetEvent
nt!KeSetEvent:
81c28703 mov     edi,edi
81c28705 push    ebp
81c28706 mov     ebp,esp
81c28708 push    ecx
81c28709 push    ecx
81c2870a push    ebx
81c2870b push    esi
81c2870c mov     esi,dword ptr [ebp+8]
81c2870f xor     ebx,ebx
81c28711 inc     ebx
81c28712 cmp     byte ptr [esi],0
81c28715 push    edi
81c28716 jne     nt!KeSetEvent+0x27 (81c2872a)

nt!KeSetEvent+0x15:
81c28718 cmp     dword ptr [esi+4],ebx
81c2871b jne     nt!KeSetEvent+0x27 (81c2872a)

nt!KeSetEvent+0x1a:
81c2871d cmp     byte ptr [ebp+10h],0
81c28721 jne     nt!KeSetEvent+0x27 (81c2872a)
```

```
nt!KeSetEvent+0x20:
81c28723 mov     eax,ebx
81c28725 jmp     nt!KeSetEvent+0xcf (81c287d6)

nt!KeSetEvent+0x27:
81c2872a xor     ecx,ecx
81c2872c call    dword ptr [nt!_imp_KeAcquireQueuedSpinLockRaiseToSynch
(81c010a4)]
81c28732 mov     byte ptr [ebp+8],al ; clears the first byte of 84ca6f00
so PKEVENT could have been any 84ca6fXX
81c28735 mov     eax,dword ptr [esi+4]
81c28738 test    eax,eax
81c2873a mov     dword ptr [ebp-4],eax
81c2873d mov     dword ptr [esi+4],ebx
81c28740 jne     nt!KeSetEvent+0×9a (81c287a1)

nt!KeSetEvent+0x3f:
81c28742 lea     edi,[esi+8]
81c28745 cmp     dword ptr [edi],edi
81c28747 je      nt!KeSetEvent+0x9a (81c287a1)

nt!KeSetEvent+0x46:
81c28749 cmp     byte ptr [esi],0
81c2874c mov     eax,dword ptr [edi]
81c2874e jne     nt!KeSetEvent+0x70 (81c28775)

nt!KeSetEvent+0x4d:
81c28750 cmp     byte ptr [eax+16h],bl
81c28753 mov     ecx,dword ptr [eax+8]
81c28756 push    dword ptr [ebp+0Ch]
81c28759 jne     nt!KeSetEvent+0x5e (81c28761)

nt!KeSetEvent+0x58:
81c2875b movzx   edx,word ptr [eax+14h]
81c2875f jmp     nt!KeSetEvent+0x63 (81c28766)

nt!KeSetEvent+0x5e:
81c28761 mov     edx,100h

nt!KeSetEvent+0x63:
81c28766 call    nt!KiUnwaitThread (81ca9097)
81c2876b mov     eax,dword ptr [edi]
81c2876d cmp     eax,edi
81c2876f je      nt!KeSetEvent+0x9a (81c287a1)

nt!KeSetEvent+0x6e:
81c28771 jmp     nt!KeSetEvent+0x4d (81c28750)

nt!KeSetEvent+0x70:
81c28775 cmp     byte ptr [eax+16h],bl
81c28778 mov     ecx,dword ptr [eax+8]
81c2877b push    dword ptr [ebp+0Ch]
81c2877e je      nt!KeSetEvent+0x8d (81c28794)
```

```
nt!KeSetEvent+0x7b:
81c28780 mov     edx,100h
81c28785 call    nt!KiUnwaitThread (81ca9097)
81c2878a mov     eax,dword ptr [edi]
81c2878c cmp     eax,edi
81c2878e je      nt!KeSetEvent+0x9a (81c287a1)

nt!KeSetEvent+0x8b:
81c28790 jmp     nt!KeSetEvent+0x70 (81c28775)

nt!KeSetEvent+0x8d:
81c28794 and     dword ptr [esi+4],0
81c28798 movzx   edx,word ptr [eax+14h]
81c2879c call    nt!KiUnwaitThread (81ca9097)

nt!KeSetEvent+0x9a:
81c287a1 cmp     byte ptr [ebp+10h],0
81c287a5 je      nt!KeSetEvent+0xb2 (81c287b9)

nt!KeSetEvent+0xa0:
81c287a7 mov     eax,dword ptr fs:[00000124h]
81c287ad mov     cl,byte ptr [ebp+8]
81c287b0 or      dword ptr [eax+68h],8
81c287b4 mov     byte ptr [eax+5Eh],cl
81c287b7 jmp     nt!KeSetEvent+0xcc (81c287d3)

nt!KeSetEvent+0xb2:
81c287b9 mov     ecx,dword ptr fs:[20h]
81c287c0 add     ecx,418h
81c287c6 call    nt!KeReleaseQueuedSpinLockFromDpcLevel (81c8bf0c)
81c287cb push    dword ptr [ebp+8]
81c287ce call    nt!KiExitDispatcher (81ca9c12)

nt!KeSetEvent+0xcc:
81c287d3 mov     eax,dword ptr [ebp-4]

nt!KeSetEvent+0xcf:
81c287d6 pop     edi
81c287d7 pop     esi
81c287d8 pop     ebx
81c287d9 leave
81c287da ret     0Ch

0: kd> dd 84ca6f00 84ca6fff
84ca6f00 a11287c8 a11287c8 00040000 00000000
84ca6f10 a11287e0 a11287e0 00040000 00000000
84ca6f20 a11287f8 a11287f8 00040000 00000000
84ca6f30 a1128810 a1128810 00040000 00000001
84ca6f40 a1128828 a1128828 00040000 00000000
84ca6f50 a1128840 a1128840 00040000 00000000
84ca6f60 a1128858 a1128858 00040000 00000000
84ca6f70 a1128888 a1128888 00040000 00000000
84ca6f80 a11288a0 a11288a0 00040000 00000000
84ca6f90 a1128870 a1128870 00040000 00000000
```

```
84ca6fa0 a11288b8 a11288b8 00040000 00000000
84ca6fb0 a11288d0 a11288d0 00040000 00000000
84ca6fc0 a11288e8 a11288e8 00040000 00000000
84ca6fd0 a1128900 a1128900 00040000 00000000
84ca6fe0 a1128918 a1128918 5e55aec0 6003be28
84ca6ff0 60181fe8 00000000 00000000 00000000

0: kd> !pool a11287c8
Pool page a11287c8 region is Paged pool
 a1128000 size:  6d0 previous size:    0  (Allocated)  Toke (Protected)
 a11286d0 size:    8 previous size:  6d0  (Free)       SeSd
 a11286d8 size:   a8 previous size:    8  (Allocated)  SpSy
 a1128780 size:   10 previous size:   a8  (Free)       AlEB
*a1128790 size:  1a0 previous size:   10  (Allocated) *KFlt
  Owning component : Unknown (update pooltag.txt)
 a1128930 size:  6d0 previous size:  1a0  (Allocated)  Toke (Protected)
```

Let’s look at our stack trace:

```
0: kd> !thread 82f49020 1f
THREAD 82f49020  Cid 0004.0034  Teb: 00000000 Win32Thread: 00000000
RUNNING on processor 0
IRP List:
    8391e008: (0006,02b0) Flags: 00000000  Mdl: 00000000
Not impersonating
DeviceMap                 85c03048
Owning Process            82f00ab0       Image:         System
Wait Start TickCount      4000214        Ticks: 0
Context Switch Count      21886
UserTime                  00:00:00.000
KernelTime                00:00:00.421
Win32 Start Address nt!ExpWorkerThread (0x81c78ea3)
Stack Init 85be0000 Current 85bdf7c0 Base 85be0000 Limit 85bdd000 Call 0
Priority 14 BasePriority 12 PriorityDecrement 0 IoPriority 2 PagePriority
5
ChildEBP RetAddr
85bdf8e8 81c28750 nt!KiTrap0E+0x2ac (TrapFrame @ 85bdf8e8)
85bdf970 876394df nt!KeSetEvent+0x4d
WARNING: Stack unwind information not available. Following frames may be
wrong.
85bdf98c 8763a145 KbdMagic+0x14df
85bdf99c 806f57a0 KbdMagic+0x2145
85bdf9ac 806f514e Wdf01000!FxPkgPnp::PnpEventFailedOwnHardware+0x3b
85bdf9d4 806f5ea9 Wdf01000!FxPkgPnp::PnpEnterNewState+0x15c
85bdf9fc 806f61b3 Wdf01000!FxPkgPnp::PnpProcessEventInner+0x1f5
85bdfa20 806ecf6b Wdf01000!FxPkgPnp::PnpProcessEvent+0x1c8
85bdfa2c 806f34b4 Wdf01000!FxPkgPnp::PnpSurpriseRemoval+0x29
85bdfa38 806edf86 Wdf01000!FxPkgFdo::_PnpSurpriseRemoval+0x10
85bdfa5c 806d7d0a Wdf01000!FxPkgPnp::Dispatch+0x26e
85bdfa68 806d7f0f Wdf01000!FxDevice::Dispatch+0x7f
85bdfa84 81c27f83 Wdf01000!FxDevice::DispatchWithLock+0x5d
85bdfa9c a4966e7f nt!IofCallDriver+0x63
85bdfac0 a496c9ae hidbth!HidBthCallDriverSynchronous+0x55
```

```
85bdfae0 85ac5a5d hidbth!HidBthPnP+0x68
85bdfaf4 85acd4c2 HIDCLASS!HidpCallDriver+0x3f
85bdfb10 85acd62e HIDCLASS!HidpFdoPnp+0x60
85bdfb20 85ac64fd HIDCLASS!HidpIrpMajorPnp+0x1e
85bdfb30 81c27f83 HIDCLASS!HidpMajorHandler+0x79
85bdfb48 81daf465 nt!IofCallDriver+0x63
85bdfb7c 81daf6cb nt!IopSynchronousCall+0xce
85bdfbd8 81da5da4 nt!IopRemoveDevice+0xd5
85bdfc00 81da5c97 nt!PnpSurpriseRemoveLockedDeviceNode+0xbd
85bdfc14 81da5f17 nt!PnpDeleteLockedDeviceNode+0x1f
85bdfc44 81daa554 nt!PnpDeleteLockedDeviceNodes+0x4c
85bdfd04 81daabe1 nt!PnpProcessQueryRemoveAndEject+0x572
85bdfd1c 81da9743 nt!PnpProcessTargetDeviceEvent+0x38
85bdfd44 81c78fa0 nt!PnpDeviceEventWorker+0x201
85bdfd7c 81e254e0 nt!ExpWorkerThread+0xfd
85bdfdc0 81c9159e nt!PspSystemThreadStartup+0x9d
00000000 00000000 nt!KiThreadStartup+0x16
```

IRP and device examination shows that KbdMagic is a lower filter driver to bluetooth HID driver and an upper filter driver to BthEnum (see Bluetooth Driver Stack WDK article, http://msdn2.microsoft.com/en-us/library/aa938547.aspx):

```
0: kd> !irp 8391e008
Irp is active with 16 stacks 14 is current (= 0x8391e24c)
 No Mdl: No System Buffer: Thread 82f49020:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000
```

```
   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

   Args: 00000000 00000000 00000000 00000000
>[ 1b,17]   0 e1 a1b9b120 00000000 a4966d36-85bdfab0 Success Error Cancel
pending
        \Driver\KbdMagic hidbth!HidBthSynchronousCompletion
   Args: 00000000 00000000 00000000 00000000
 [ 1b,17]   0  0 8a1fc030 00000000 00000000-00000000
        \Driver\HidBth
   Args: 00000000 00000000 00000000 00000000
 [ 1b,17]   0  0 8a1fc030 00000000 00000000-00000000
        \Driver\HidBth
   Args: 00000000 00000000 00000000 00000000

0: kd> !devobj 8a1fc030
Device object (8a1fc030) is for:
 _HID00000006 \Driver\HidBth DriverObject 836225e0
Current Irp 00000000 RefCount 0 Type 00000022 Flags 00002050
Dacl 85c60218 DevExt 8a1fc0e8 DevObjExt 8a1fce98
ExtensionFlags (0x00000800)
                             Unknown flags 0x00000800
AttachedTo (Lower) a1b9b120 \Driver\KbdMagic
Device queue is not busy.

0: kd> !devobj a1b9b120
Device object (a1b9b120) is for:
  \Driver\KbdMagic DriverObject 83712d70
Current Irp 00000000 RefCount 0 Type 00000022 Flags 00002004
DevExt 84ca68a0 DevObjExt a1b9b1f0
ExtensionFlags (0x00000800)
                             Unknown flags 0x00000800
AttachedDevice (Upper) 8a1fc030 \Driver\HidBth
AttachedTo (Lower) 8a1ef030 \Driver\BthEnum
Device queue is not busy.

0: kd> !devstack 8a1ef030
  !DevObj   !DrvObj            !DevExt   ObjectName
  8a1fc030  \Driver\HidBth     8a1fc0e8  _HID00000006
  a1b9b120  \Driver\KbdMagic   84ca68a0
> 8a1ef030  \Driver\BthEnum    8a1ef0e8  00000068
```

lmv command doesn’t show detailed module information:

```
0: kd> lmv m KbdMagic
start    end        module name
87638000 87642000   KbdMagic   (no symbols)
    Loaded symbol image file: KbdMagic.sys
    Image path: \SystemRoot\system32\DRIVERS\KbdMagic.sys
    Image name: KbdMagic.sys
    Timestamp:        Thu Aug 30 22:59:01 2007 (46D73DA5)
    CheckSum:         0000B906
    ImageSize:        0000A000
    Translations:     0000.04b0 0000.04e0 0409.04b0 0409.04e0
```

But dumping the module contents shows more version data (**Unknown Component** pattern, Volume 1, page 367):

```
0: kd> dc 87638000 87642000
...
...
...
8763b120  \.r.e.g.i.s.t.r.
8763b130  y.\.m.a.c.h.i.n.
8763b140  e.\.S.y.s.t.e.m.
8763b150  \.C.u.r.r.e.n.t.
8763b160  C.o.n.t.r.o.l.S.
8763b170  e.t.\.S.e.r.v.i.
8763b180  c.e.s.\.k.b.d.m.
8763b190  a.g.i.c.....FILT
8763b1a0  ER_EXTENSION....
8763b1b0  NEW_LAYOUT..OLD_
8763b1c0  LAYOUT..UNKNOWN_
8763b1d0  LAYOUT..EXTERNAL
8763b1e0  _BLUETOOTH..EXTE
8763b1f0  RNAL_CORDED.INTE
8763b200  RNAL....UNKNOWN_
8763b210  TYPE....JIS.ANSI
8763b220  ....ISO.UNKNOWN_
8763b230  LANG............
8763b240  u.....%.........
8763b250  ................
8763b260  ............K.m.
8763b270  d.f.L.i.b.r.a.r.
8763b280  y...RSDS.....W.M
8763b290  .V..A..e....c:\b
8763b2a0  wa\vendrkeyboard
8763b2b0  win-200.1.4\srcr
8763b2c0  oot\vendrkeyboar
8763b2d0  d\objfre_wlh_x86
8763b2e0  \i386\KbdMagic.p
8763b2f0  db
...
```

Therefore we have enough evidence for KbdMagic.sys to contact the vendor for updates. To be absolutely sure we can enable IRQL checking in Driver Verifier for KbdMagic.sys.

EXPLORING NDIS EXTENSION

There is a good Microsoft white paper about **!ndiskd** commands to interrogate kernel dumps:

Debugging NDIS Drivers
http://download.microsoft.com/download/5/7/7/577a5684-8a83-43ae-9272-ff260a9c20e2/ndisWinHec2003.doc

Applying **!ndiskd.protocols** command we can see that there are more protocols added to Vista:

Windows Server 2003 SP2:

```
kd> !ndiskd.protocols
 Protocol 862db330: NDISUIO
    Open 86420650 - Miniport: 862e2ab0 AMD PCNET Family PCI Ethernet
Adapter

 Protocol 86324780: TCPIP_WANARP
    Open 86324008 - Miniport: 863a2130 WAN Miniport (IP)

 Protocol 86318790: TCPIP
    Open 8637c008 - Miniport: 862e2ab0 AMD PCNET Family PCI Ethernet
Adapter

 Protocol 863e3c28: NDPROXY
    Open 8639e0d0 - Miniport: 86361530 Direct Parallel
    Open 8639bb48 - Miniport: 86361530 Direct Parallel
    Open 863e48b0 - Miniport: 863e3130 WAN Miniport (L2TP)
    Open 86404008 - Miniport: 863e3130 WAN Miniport (L2TP)

 Protocol 863a9d80: RASPPPOE

 Protocol 863a9008: NDISWAN
    Open 863e3ab0 - Miniport: 86361530 Direct Parallel
    Open 86398c30 - Miniport: 862c4530 WAN Miniport (PPTP)
    Open 864618f8 - Miniport: 8637a870 WAN Miniport (PPPOE)
    Open 86468a28 - Miniport: 863e3130 WAN Miniport (L2TP)
```

Vista:

```
1: kd> !ndiskd.protocols
 Protocol fffffa8004569580: RSPNDR
    Open fffffa8004566a20 - Miniport: fffffa80039711a0 Broadcom NetXtreme
57xx Gigabit Controller

 Protocol fffffa80043a4900: LLTDIO
    Open fffffa800428a1d0 - Miniport: fffffa80039711a0 Broadcom NetXtreme
57xx Gigabit Controller

 Protocol fffffa8003f6c820: WANARPV6
    Open fffffa8003f1c010 - Miniport: fffffa800399f1a0 WAN Miniport (IPv6)

 Protocol fffffa8003f6cd20: WANARP
    Open fffffa8003f1c670 - Miniport: fffffa80039d61a0 WAN Miniport (IP)

 Protocol fffffa8003eedb10: TCPIP6TUNNEL
    Open fffffa8003f33010 - Miniport: fffffa800396c1a0 isatap.company.com
    Open fffffa8003f0f010 - Miniport: fffffa80038f21a0 Teredo Tunneling
Pseudo-Interface

 Protocol fffffa8003eeb580: TCPIPTUNNEL

 Protocol fffffa8003eeb010: TCPIP6
    Open fffffa8003f452e0 - Miniport: fffffa80039711a0 Broadcom NetXtreme
57xx Gigabit Controller

 Protocol fffffa8003ee90d0: TCPIP
    Open fffffa8003ffc480 - Miniport: fffffa80039711a0 Broadcom NetXtreme
57xx Gigabit Controller

 Protocol fffffa8003c56010: NDPROXY
    Open fffffa8003d41450 - Miniport: fffffa800399d1a0 WAN Miniport (L2TP)
    Open fffffa8003d41a30 - Miniport: fffffa800399d1a0 WAN Miniport (L2TP)

 Protocol fffffa80039ad790: RASPPPOE

 Protocol fffffa80039af4e0: NDISWAN
    Open fffffa8004737a10 - Miniport: fffffa8004a321a0 RAS Async Adapter
    Open fffffa8003bf8ac0 - Miniport: fffffa80039c21a0 WAN Miniport (PPTP)
    Open fffffa8003c5cac0 - Miniport: fffffa80039c01a0 WAN Miniport
(PPPOE)
    Open fffffa8003c565a0 - Miniport: fffffa800399d1a0 WAN Miniport (L2TP)
```

Let's try this extension on a bugcheck memory dump from the 3rd-party custom protocol driver:

```
SYSTEM_PTE_MISUSE (da)
The stack trace identifies the guilty driver.
Arguments:
Arg1: 00000400, Type of error.
Arg2: f7a9a413
Arg3: 00000001
Arg4: 00000000

0: kd> kL
ChildEBP RetAddr
f5c68a68 8083b6e1 nt!KeBugCheckEx+0x1b
f5c68a90 8083d478 nt!MiRemoveIoSpaceMap+0x5d
f5c68b38 f5b6ebea nt!MmUnmapIoSpace+0x10c
WARNING: Stack unwind information not available. Following frames may be
wrong.
f5c68b90 f5b69abe protocol!foo2+0x28ac
f5c68bf4 f70fd4be protocol!foo+0x1aa0
f5c68c90 f70fd2fc NDIS!ndisInitializeBinding+0x189
f5c68d18 f70fce48 NDIS!ndisCheckAdapterBindings+0xd9
f5c68d98 f70fca66 NDIS!ndisCheckProtocolBindings+0xd2
f5c68dac 80949b7c NDIS!ndisWorkerThread+0x74
f5c68ddc 8088e062 nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16
```

Arg1 0×400 one tells us this (from WinDbg help):

0×400	*The base address of the I/O space mapping*	*The number of pages to be freed*	*0*	*(WINDOWS XP AND LATER ONLY) The caller is trying to free an I/O space mapping that the system is unaware of.*

PTE looks unknown indeed:

```
0: kd> !pte f7a9a413
VA f7a9a413
PDE at 00000000C0603DE8 PTE at 00000000C07BD4D0
contains 0000000000A87863 contains 0000000000000000
pfn a87 —DA–KWEV
```

We can see this protocol in the list:

```
0: kd> !ndiskd.protocols
 Protocol 89df10a0: CustomProtocol
    Open 89b4e6d8 - Miniport: 8a59d290 Broadcom BCM5708S NetXtreme II GigE
(NDIS VBD Client)

 Protocol 8918f248: NDISUIO

 Protocol 89dd8008: TCPIP_WANARP
    Open 8a4da6f0 - Miniport: 8a50a9e8 WAN Miniport (IP)

 Protocol 89b4ec88: TCPIP

 Protocol 8a4cd5a0: NDPROXY
    Open 8a59b128 - Miniport: 8a58eab0 Direct Parallel
    Open 8a59b328 - Miniport: 8a58eab0 Direct Parallel
    Open 8a4f1580 - Miniport: 8a58a328 WAN Miniport (L2TP)
    Open 8a507008 - Miniport: 8a58a328 WAN Miniport (L2TP)

 Protocol 8a4e7008: RASPPPOE

 Protocol 8a5cb490: NDISWAN
    Open 8a59b988 - Miniport: 8a58eab0 Direct Parallel
    Open 8a5976c0 - Miniport: 8a591628 WAN Miniport (PPTP)
    Open 8a594468 - Miniport: 8a4e93f0 WAN Miniport (PPPOE)
    Open 8a4d3580 - Miniport: 8a58a328 WAN Miniport (L2TP)
```

THE HUNT FOR THE DEBUGGER

Sometimes we have processes that actively monitor debugger attachments to prevent reverse engineering and terminate themselves if such attempts are detected. Some of them use very simple methods to achieve this like creating a thread that periodically calls IsDebuggerPresent API or waits for debugger events. In such cases attempts of any application to actively attach to these processes result in their termination.

Consider the following stack trace from the postmortem crash dump saved by NTSD on Windows Server 2003:

```
0:000> kL
ChildEBP RetAddr
00fefbcc 098b84a1 kernel32!RaiseException+0x53
...
...
...
00fefd28 0116a86a component!_CRT_INIT+0x187
00fefd6c 0116a8e6 component!__DllMainCRTStartup+0xb7
00fefd74 7c81a352 component!_DllMainCRTStartup+0x1d
00fefd94 7c830e70 ntdll!LdrpCallInitRoutine+0x14
00fefe4c 77e668a3 ntdll!LdrShutdownProcess+0x182
00feff38 77e66905 kernel32!_ExitProcess+0x43
00feff4c 00561ab9 kernel32!ExitProcess+0x14
00feffb8 77e64829 application!foo+0x41
00feffec 00000000 kernel32!BaseThreadStart+0x34
```

Disassembling application!foo shows the call to WaitForDebugEvent API:

```
0:000> u application!foo
application!foo:
00561a78 push    ebp
00561a79 mov     ebp,esp
00561a7b sub     esp,60h
00561a7e push    0FFFFFFFFh
00561a80 lea     eax,[ebp-60h]
00561a83 push    eax
00561a84 call    dword ptr [application!_imp__WaitForDebugEvent
(00655224)]
00561a8a mov     eax,dword ptr [ebp+8]
```

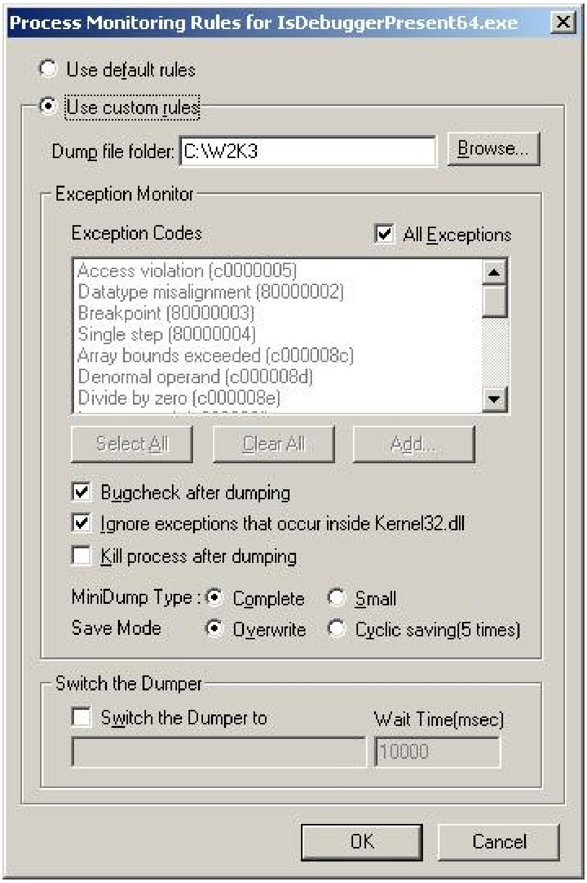
We also see it on the raw stack (this might help in more complex cases):

```
0:000> !teb
TEB at 7ffdd000
    ExceptionList:        00fefbf8
    StackBase:            00ff0000
    StackLimit:           00fef000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffdd000
    EnvironmentPointer:   00000000
    ClientId:             000063fc . 00003270
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffdb000
    LastErrorValue:       0
    LastStatusValue:      c0000034
    Count Owned Locks:    0
    HardErrorMode:        0

0:000> dds 00fef000 00ff0000
...
...
...
00fefecc  00000000
00fefed0  00feff48
00fefed4  77e9c4d7 kernel32!WaitForDebugEvent+0×66
00fefed8  c0000008
00fefedc  00000000
00fefee0  77e41ef3 kernel32!SleepEx+0×91
00fefee4  00000000
00fefee8  00000000
...
...
...
```

How would we find what process was trying to attach to our application? Let’s go with pure crash dump analysis approach. We can take the advantage of RaiseException call and get a kernel or a complete memory dump to examine all running processes and their threads. In order to model this we can create a small program that simulates the behavior shown above:

```
// IsDebuggerPresent64

#include "stdafx.h"
#include "windows.h"

int _tmain(int argc, _TCHAR* argv[])
{
    while (1)
    {
        IsDebuggerPresent()
            ? puts ("Yes"),
                RaiseException(0x12345678,
                    0, 0, NULL)
            : puts ("No");
        Sleep(5000);
    }

    return 0;
}
```

We configure Process Monitoring Rules in Userdump Process Dumper Control Panel applet to bugcheck the system after dumping:



Process Monitoring Rules for IsDebuggerPresent64.exe

○ Use default rules

● Use custom rules

Dump file folder: C:\W2K3

Exception Monitor

Exception Codes ☑ All Exceptions

Access violation (c0000005)
Datatype misalignment (80000002)
Breakpoint (80000003)
Single step (80000004)
Array bounds exceeded (c000008c)
Denormal operand (c000008d)
Divide by zero (c000008e)

☑ Bugcheck after dumping

☑ Ignore exceptions that occur inside Kernel32.dll

□ Kill process after dumping

MiniDump Type : ● Complete ○ Small

Save Mode ● Overwrite ○ Cyclic saving(5 times)

Switch the Dumper

□ Switch the Dumper to

Wait Time(msec) 10000

Then we can ask someone to debug the running instance of that application and let us know if there was any sudden BSOD. Indeed there was one and we got our complete memory dump, although kernel dump would suffice here. Let’s look at it.

We see that our process has an open debug port and its main thread is suspended:

```
kd> !process /r /p fffffadfe73f9c20
PROCESS fffffadfe73f9c20
    SessionId: 0  Cid: 0e4c    Peb: 7fffffd4000  ParentCid: 0e54
    DirBase: 2c472000  ObjectTable: fffffa80006f1690  HandleCount:  12.
    Image: IsDebuggerPresent64.exe
    VadRoot fffffadfe6ef3e30 Vads 26 Clone 0 Private 97. Modified 0.
Locked 0.
    DeviceMap fffffa8000930540
    Token                             fffffa80030e7910
    ElapsedTime                       00:00:03.062
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         16288
    QuotaPoolUsage[NonPagedPool]      3488
    Working Set Sizes (now,min,max)  (1322, 50, 345) (5288KB, 200KB,
1380KB)
    PeakWorkingSetSize                1322
    VirtualSize                       7 Mb
    PeakVirtualSize                   7 Mb
    PageFaultCount                    1314
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      107
    DebugPort                         fffffadfe6ec9040

THREAD fffffadfe69a2bf0  Cid 0e4c.0e74  Teb: 000007fffffde000 Win32Thread:
0000000000000000 WAIT: (Unknown) KernelMode Non-Alertable
SuspendCount 1
    fffffadfe69a2e90  Semaphore Limit 0×2
Not impersonating
DeviceMap                 fffffa8000930540
Owning
Process            fffffadfe73f9c20       Image:         IsDebuggerPresent
64.exe
Wait Start TickCount      37247          Ticks: 49 (0:00:00:00.765)
Context Switch Count      45
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address IsDebuggerPresent64 (0×00000001400013b0)
Start Address kernel32!BaseProcessStart (0×0000000077d59620)
Stack Init fffffadfdf7b5e00 Current fffffadfdf7b54f0
Base fffffadfdf7b6000 Limit fffffadfdf7b0000 Call 0
```

```
Priority 12 BasePriority 8 PriorityDecrement 2
Child-SP          RetAddr           Call Site
fffffadf`df7b5530 fffff800`0103b063 nt!KiSwapContext+0x85
fffffadf`df7b56b0 fffff800`0103c403 nt!KiSwapThread+0xc3
fffffadf`df7b56f0 fffff800`0105dc7c nt!KeWaitForSingleObject+0x528
fffffadf`df7b5780 fffff800`0105db2b nt!KiSuspendThread+0x2c
fffffadf`df7b57c0 fffff800`01058e71 nt!KiDeliverApc+0x20a
fffffadf`df7b5840 fffff800`0103c403 nt!KiSwapThread+0xde
fffffadf`df7b5880 fffffadf`dfd4a20c nt!KeWaitForSingleObject+0x528
fffffadf`df7b5910 fffffadf`dfd4a3be
userdump!UdpCompleteExceptionForwarding+0x11c
fffffadf`df7b5990 fffffadf`dfd49dd8 userdump!UdpForwardException+0x13e
fffffadf`df7b59c0 fffff800`012ce9cf userdump!UdIoctl+0x618
fffffadf`df7b5a70 fffff800`012df026 nt!IopXxxControlFile+0xa5a
fffffadf`df7b5b90 fffff800`010410fd nt!NtDeviceIoControlFile+0x56
fffffadf`df7b5c00 00000000`77ef0a5a nt!KiSystemServiceCopyEnd+0x3
(TrapFrame @ fffffadf`df7b5c70)
```

If we search for a process that has NtWaitForDebugEvent function present on one of its stack traces we would find the debugger:

```
kd> !stacks 2 nt!NtWaitForDebugEvent
...
...
...
                [fffffadfe63da3b0 ntsd.exe]
 e54.000e50  fffffadfe6afbbf0 ffff6e8a Blocked    nt!KiSwapContext+0×85
                nt!KiSwapThread+0xc3
                nt!KeWaitForSingleObject+0×528
                nt!NtWaitForDebugEvent+0×342
                nt!KiSystemServiceCopyEnd+0×3
                ntdll!ZwWaitForDebugEvent+0xa
                dbgeng!LiveUserDebugServices::WaitForEvent+0xee
                dbgeng!LiveUserTargetInfo::WaitForEvent+0×488
                dbgeng!RawWaitForEvent+0×23c
                dbgeng!DebugClient::WaitForEvent+0×96
                ntsd!MainLoop+0xb7
                ntsd!main+0×18e
                ntsd!mainCRTStartup+0×171
                kernel32!BaseProcessStart+0×29
```

We see that it is NTSD.

COMPLETE DUMP: USER SPACE CRITICAL SECTIONS

Suppose we have a complete memory dump and we want to check critical sections to see any anomalies. We can do this by using **!for_each_process** extension command:

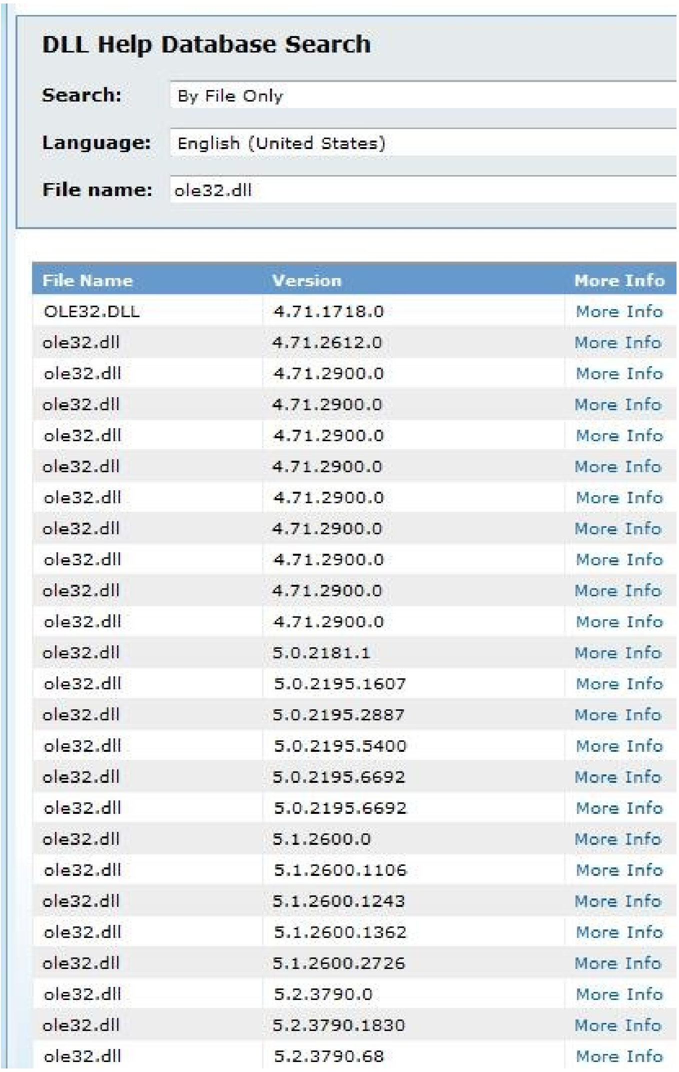
```
0: kd> !for_each_process ".process /r /p @#Process; !ntsdexts.locks"
Implicit process is now a59a4648
Loading User Symbols

NTSDEXTS: Unable to resolve ntdll!RtlCriticalSectionList
NTSDEXTS: Please check your symbols
Implicit process is now a553cd88
Loading User Symbols
....

Scanned 11 critical sections
Implicit process is now a518b1b0
Loading User Symbols
....

Scanned 105 critical sections
Implicit process is now a513a348
Loading User Symbols
....

Scanned 977 critical sections
Implicit process is now a5659d88
Loading User Symbols
....

Scanned 438 critical sections
Implicit process is now a551abb8
Loading User Symbols
....
...
...
...
...
```

Here the first NTSDEXTS warning is normal because we don’t have user space for System process.

MICROSOFT DLL HELP DATABASE

This is sometimes useful resource where we can check the product and indirectly any updates for the particular module if we have its file name and version from a crash dump:

http://support.microsoft.com/dllhelp/



DLL Help Database Search

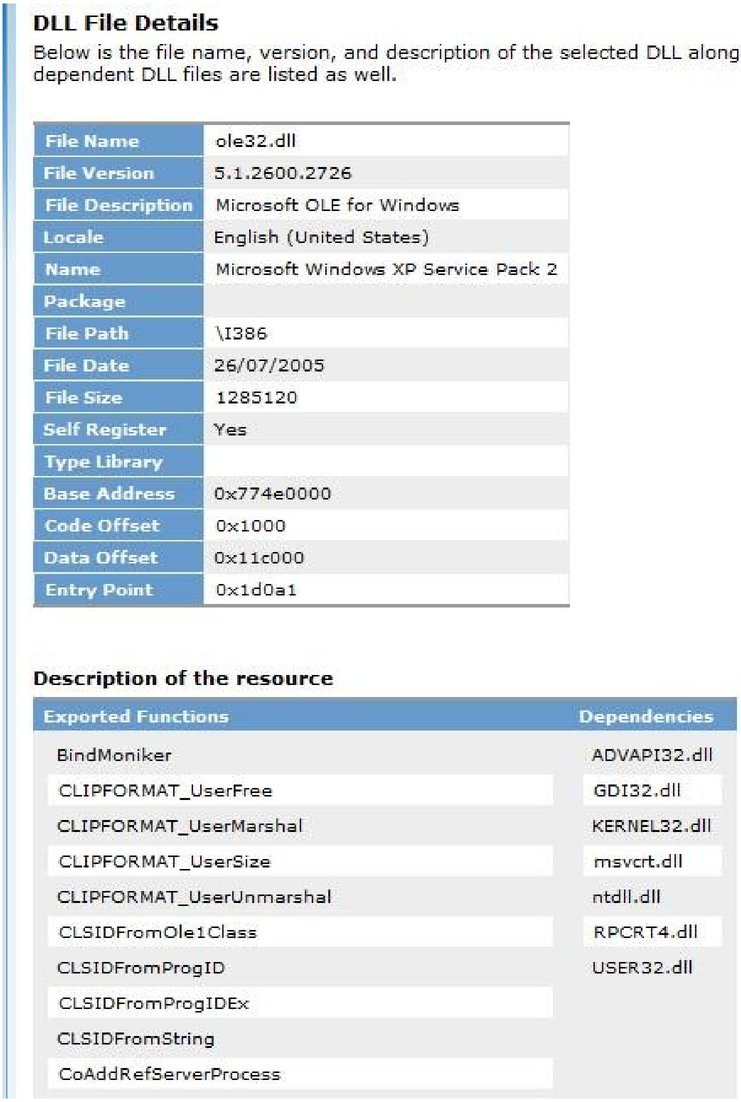
Search: By File Only

Language: English (United States)

File name: ole32.dll

File Name	Version	More Info
OLE32.DLL	4.71.1718.0	More Info
ole32.dll	4.71.2612.0	More Info
ole32.dll	4.71.2900.0	More Info
ole32.dll	4.71.2900.0	More Info
ole32.dll	4.71.2900.0	More Info
ole32.dll	4.71.2900.0	More Info
ole32.dll	4.71.2900.0	More Info
ole32.dll	4.71.2900.0	More Info
ole32.dll	4.71.2900.0	More Info
ole32.dll	4.71.2900.0	More Info
ole32.dll	4.71.2900.0	More Info
ole32.dll	5.0.2181.1	More Info
ole32.dll	5.0.2195.1607	More Info
ole32.dll	5.0.2195.2887	More Info
ole32.dll	5.0.2195.5400	More Info
ole32.dll	5.0.2195.6692	More Info
ole32.dll	5.0.2195.6692	More Info
ole32.dll	5.1.2600.0	More Info
ole32.dll	5.1.2600.1106	More Info
ole32.dll	5.1.2600.1243	More Info
ole32.dll	5.1.2600.1362	More Info
ole32.dll	5.1.2600.2726	More Info
ole32.dll	5.2.3790.0	More Info
ole32.dll	5.2.3790.1830	More Info
ole32.dll	5.2.3790.68	More Info

We can event see exports and component dependencies if we need to quickly check them without running depends.exe:



DLL File Details

Below is the file name, version, and description of the selected DLL along dependent DLL files are listed as well.

File Name	ole32.dll
File Version	5.1.2600.2726
File Description	Microsoft OLE for Windows
Locale	English (United States)
Name	Microsoft Windows XP Service Pack 2
Package	
File Path	\I386
File Date	26/07/2005
File Size	1285120
Self Register	Yes
Type Library	
Base Address	0x774e0000
Code Offset	0x1000
Data Offset	0x11c000
Entry Point	0x1d0a1

Description of the resource

Exported Functions	Dependencies
BindMoniker	ADVAPI32.dll
CLIPFORMAT_UserFree	GDI32.dll
CLIPFORMAT_UserMarshal	KERNEL32.dll
CLIPFORMAT_UserSize	msvcrt.dll
CLIPFORMAT_UserUnmarshal	ntdll.dll
CLSIDFromOle1Class	RPCRT4.dll
CLSIDFromProgID	USER32.dll
CLSIDFromProgIDEx	
CLSIDFromString	
CoAddRefServerProcess	

WHAT DOES THIS FUNCTION DO?

Often we need to guess what a particular function that we see on a stack trace does. The following function name and purpose mining techniques and resources can be useful:

- We might need to strip or replace prefixes and suffixes like

  ```
  NtUserGetMessage

  GetMessageW

  ZwReadFile <-> NtReadFile
  ```

- Search in MSDN, Platform SDK and WDK (formerly DDK) help
- Various blogs like this excellent summary:

  A catalog of NTDLL kernel mode to user mode callbacks
  http://www.nynaeve.net/?tag=ntdll

- Reverse engineering and logical deduction:

  What is KiFastSystemCallRet? (Volume 1, page 649)

- Various books like this:

  Windows NT/2000 Native API Reference (ISBN: 978-1578701995).

- Win32 API emulators like WINE (Volume 1, page 697)
- and finally Windows source code if you are a Microsoft source code licensee or a participant in Windows Academic Program.
- Sometimes Internet search finds the description of the whole stack trace collection from the class of common processes like this one:

Production Debugging for .NET Framework Applications
http://msdn.microsoft.com/en-us/library/ms954593.aspx

WHAT WAS THIS PROCESS DOING?

This is a common question we have when faced with stack traces for which we don’t have symbols. Consider the following stack trace from a complete memory dump where a bugcheck thread belongs to one graphical application:

```
2: kd> kL 100
ChildEBP RetAddr
aa1999b4 8082d800 nt!KeBugCheckEx+0x1b
aa199d78 8088a262 nt!KiDispatchException+0x3a2
aa199de0 8088a216 nt!CommonDispatchException+0x4a
aa199e5c bfe7e5b7 nt!KiExceptionExit+0x186
...
aa19a110 bf8b2fe6 win32k!GrePolyPatBlt+0x45
aa19a148 bf89422b win32k!FillRect+0x58
aa19a16c bf8942f7 win32k!xxxPaintRect+0x70
aa19a19c bf8942ac win32k!xxxFillWindow+0x3e
aa19a1b4 bf8adf6e win32k!xxxDWP_EraseBkgnd+0x51
aa19a214 bf884771 win32k!xxxRealDefWindowProc+0x318
aa19a22c bf8847a1 win32k!xxxWrapRealDefWindowProc+0x16
aa19a248 bf8c1459 win32k!NtUserfnNCDESTROY+0x27
aa19a280 8088978c win32k!NtUserMessageCall+0xc0
aa19a280 7c8285ec nt!KiFastCallEntry+0xfc
0013f68c 7739d1ec ntdll!KiFastSystemCallRet
0013f6e0 7739c6ae USER32!NtUserMessageCall+0xc
0013f6fc 7739c718 USER32!RealDefWindowProcW+0x47
0013f744 3003a5b3 USER32!DefWindowProcW+0x72
0013f75c 300a0d72 Application+0x3a5b3
0013f7bc 300a0cb2 Application+0xa0d72
0013f7f4 7739b6e3 Application+0xa0cb2
0013f820 7739b874 USER32!InternalCallWinProc+0x28
0013f898 7739c8b8 USER32!UserCallWinProcCheckWow+0x151
0013f8f4 7739c9c6 USER32!DispatchClientMessage+0xd9
0013f91c 7c828536 USER32!__fnDWORD+0x24
0013f91c 808308f4 ntdll!KiUserCallbackDispatcher+0x2e
aa19a564 8091d6d1 nt!KiCallUserMode+0x4
aa19a5bc bf8a2622 nt!KeUserModeCallback+0x8f
aa19a640 bf8a242d win32k!SfnDWORD+0xb4
aa19a688 bf8a13d9 win32k!xxxSendMessageToClient+0x176
aa19a6d4 bf8a12ee win32k!xxxSendMessageTimeout+0x1a6
aa19a6f8 bf8c1342 win32k!xxxSendMessage+0x1b
aa19a71c bf85e0a1 win32k!xxxSendEraseBkgnd+0x5c
aa19a73c bf85dee1 win32k!xxxSimpleDoSyncPaint+0xc6
aa19a758 bf8ae16d win32k!xxxInternalDoSyncPaint+0x12
aa19a7b4 bf884771 win32k!xxxRealDefWindowProc+0x753
aa19a7cc bf8847a1 win32k!xxxWrapRealDefWindowProc+0x16
aa19a7e8 bf8c1459 win32k!NtUserfnNCDESTROY+0x27
aa19a820 8088978c win32k!NtUserMessageCall+0xc0
aa19a820 7c8285ec nt!KiFastCallEntry+0xfc
0013f91c 7c828536 ntdll!KiFastSystemCallRet
0013f91c 808308f4 ntdll!KiUserCallbackDispatcher+0x2e
```

```
aa19ab00 8091d6d1 nt!KiCallUserMode+0x4
aa19ab58 bf8a2622 nt!KeUserModeCallback+0x8f
aa19abdc bf8a242d win32k!SfnDWORD+0xb4
aa19ac24 bf8c4177 win32k!xxxSendMessageToClient+0x176
aa19ac94 bf89b829 win32k!xxxReceiveMessage+0x2b5
aa19ace4 bf89c4d9 win32k!xxxRealInternalGetMessage+0x1da
aa19ad48 8088978c win32k!NtUserPeekMessage+0x42
aa19ad48 7c8285ec nt!KiFastCallEntry+0xfc
0013fbd8 7c828536 ntdll!KiFastSystemCallRet
0013fc04 7739bde5 ntdll!KiUserCallbackDispatcher+0x2e
0013fc30 7739be5e USER32!NtUserPeekMessage+0xc
0013fc5c 3002baa0 USER32!PeekMessageW+0xab
0013fc84 3002b556 Application+0x2baa0
0013fca8 3000abf5 Application+0x2b556
0013fcf4 30005dfd Application+0xabf5
0013ff34 3000248c Application+0x5dfd
0013ffc0 77e6f23b Application+0x248c
0013fff0 00000000 kernel32!BaseProcessStart+0x23
```

The thread seems to be doing some drawing in response to WM_ERASEBKGND message generated from the code processing WM_TIMER:

```
2: kd> kv 100
...
aa19a6f8 bf8c1342 be63f8b8 00000014 91010979 win32k!xxxSendMessage+0×1b
aa19a71c bf85e0a1 be63f8b8 00000000 00000001 win32k!xxxSendEraseBkgnd+0×5c
...
0013fc5c 3002baa0 0013fcc0 00000000 00000000 USER32!PeekMessageW+0xab
...

2: kd> dd 0013fcc0 l4
0013fcc0  00000000 00000113 000066c2 00000000
```

The first parameter to PeekMessage function is a pointer to MSG structure whose second member is a message code (from MSDN):

```
BOOL PeekMessage(
    LPMSG lpMsg,
    HWND hWnd,
    UINT wMsgFilterMin,
    UINT wMsgFilterMax,
    UINT wRemoveMsg
);
```

```
typedef struct {
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG;
```

In WinUser.h we can find message codes:

```
#define WM_ERASEBKGND  0x0014
#define WM_TIMER       0x0113
```

Now we can ask the next troubleshooting question: what was the application file loaded before the system crash? We know that the application uses EXT file extension for its data. If we look at the handle table we find the only one such instance of File object:

```
2: kd> !handle
processor number 2, process a31a4a08
PROCESS a31a4a08  SessionId: 1  Cid: 2440    Peb: 7ffd7000  ParentCid:
1180
    DirBase: bffca720  ObjectTable: ddc38eb8  HandleCount: 291.
    Image: Application.EXE

Handle table at dcb65000 with 291 Entries in use

...

03f4: Object: a2ee85b0  GrantedAccess: 00120089 Entry: dcb657e8
Object: a2ee85b0  Type: (a55c8ca0) File
    ObjectHeader: a2ee8598 (old version)
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name:
\Profiles\MYNAME\LOCALS~1\Temp\APPDATA\MyFile.ext {HarddiskVolume3}

...
```

Now we can check other crash dumps to see whether there is any consistency in file names.

STL AND WINDBG

Some applications are written using Standard Template Library and it is good that there is **!stl** WinDbg extension that works with a few types from Plauger's STL implementation used in Visual C++ CRT library:

```
0:000> !stl
!stl [options] <varname>
  stl [options] <varname> - dumps an STL variable
  stl [options] -n <type-name> <address>
             currently works with string, wstring
             vector<string>, vector<wstring>
             list<string>, vector<wstring>
             (and pointer varieties therein)
   [options]
       -n <type-name> The name of the type. If the
               type has spaces, surround with
               parentheses ().
       -v      verbose output
       -V      extremely verbose output
```

If we have public symbols and know variable names we can simply dump their values, for example:

```
0:000> dv /i /V
prv local  @ecx @ecx            this = 0x0012fbdc
prv local  0012fbf8 @ebp-0x2c   MyName = class
std::basic_string<char,std::char_traits<char>,std::allocator<char> >

0:000> !stl MyName
[da 0x12fbfc]
0012fbfc  "COMPANY__NAME"
```

We can also supply full STL type name:

```
0:000> !stl -n
(std::basic_string<char,std::char_traits<char>,std::allocator<char> >)
0012fbf8
[da 0x12fbfc]
0012fbfc  "COMPANY__NAME"
```

Let’s dump this string type internal structure to be able to recognize it later in raw data:

```
0:000> dt -r -n
std::basic_string<char,std::char_traits<char>,std::allocator<char> >
0012fbf8
application!std::basic_string<char,std::char_traits<char>,std::allocator<c
har> >
   +0x000 _Alval            : std::allocator<char>
   =00400000 npos              : 0x905a4d
   +0×004 _Bx               :
std::basic_string<char,std::char_traits<char>,std::allocator<char>
>::_Bxty
      +0×000 _Buf              : [16]  “COMPANY__NAME”
      +0×000 _Ptr              : 0×43415250  “”
   +0×014 _Mysize           : 0xd
   +0×018 _Myres            : 0xf
```

We see that for short strings less than 16 bytes std::basic_string<char> data starts from offset +4 and followed by the actual string size and its reserved size:

```
0:000> dd 0012fbf8
0012fbf8  00000000 43415250 45434954 53504d5f
0012fc08  41bf0033 0000000d 0000000f 41bf3b72
0012fc18  0012fc6c 0046107b 00000000 0012fc78
0012fc28  0041a441 00000000 41bf3b2e 00ed6380
0012fc38  00000003 00ed6128 00ed6128 00f41b00
0012fc48  00ed6128 41bf3b3e 0012fc3c 00000000
0012fc58  0000000f 00f41b98 00f469a0 00000000
0012fc68  014487c8 0012fcfc 00463fdd 00000002
```

For bigger strings implementation starts with a pointer from offset +4 to the actual string data and then followed by 12 bytes of garbage and then by the actual string size and its reserved size:

```
0:000> dt -r -n
std::basic_string<char,std::char_traits<char>,std::allocator<char> >
application!std::basic_string<char,std::char_traits<char>,std::allocator<c
har> >
   +0x000 _Alval            : std::allocator<char>
   =00400000 npos              : Uint4B
   +0×004 _Bx               :
std::basic_string<char,std::char_traits<char>,std::allocator<char>
>::_Bxty
      +0×000 _Buf              : [16] Char
      +0×000 _Ptr              : Ptr32 Char
   +0×014 _Mysize           : Uint4B
   +0×018 _Myres            : Uint4B
```

```
0:000> dt -r -n
std::basic_string<char,std::char_traits<char>,std::allocator<char> >
0012ff08
application!std::basic_string<char,std::char_traits<char>,std::allocator<c
har> >
   +0x000 _Alval           : std::allocator<char>
   =00400000 npos            : 0x905a4d
   +0×004 _Bx              :
std::basic_string<char,std::char_traits<char>,std::allocator<char>
>::_Bxty
      +0×000 _Buf            : [16]  “???”
      +0×000 _Ptr            : 0×00ed4ba0  “/h /c:100 /enum”
   +0×014 _Mysize          : 0×10
   +0×018 _Myres           : 0×1f
```

In such cases **dpa** or **dpu** commands help to show this additional dereference:

```
0:000> dpa 0012ff08
0012ff08  00ed2f90 "."
0012ff0c  00ed4ba0 “/h /c:100 /enum”
0012ff10  41eafd01
0012ff14  0012ffc0 “…”
0012ff18  0045890a “……U..SVWUj”
0012ff1c  00000010
0012ff20  0000001f
0012ff24  41bf3996
0012ff28  0012ffc0 “…”
0012ff2c  0044b528 “.E..}.”
0012ff30  00400000 “MZ.”
```

WINDBG CHEAT SHEET

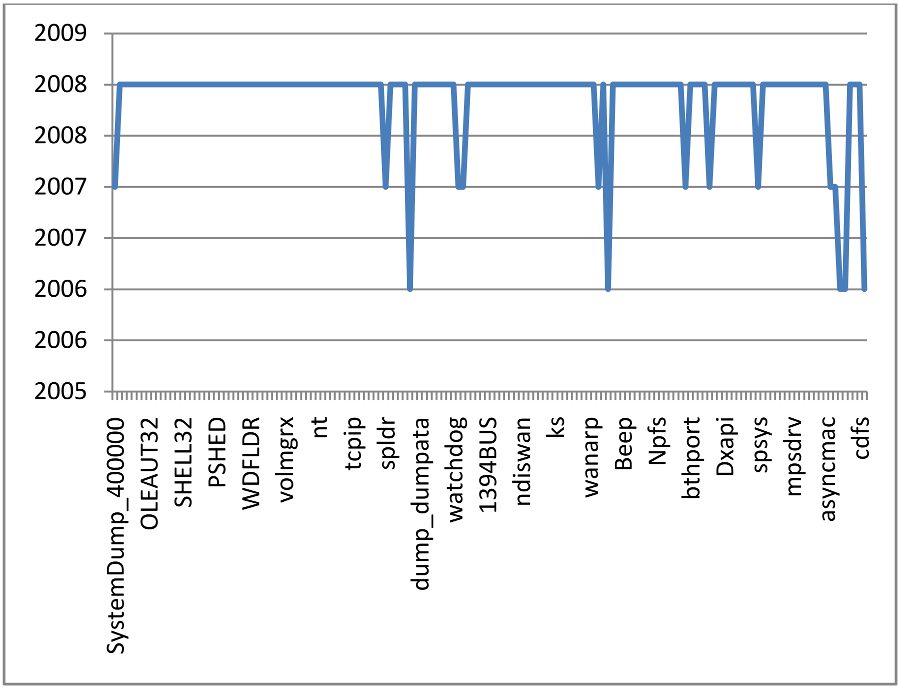
Introduced in Volume 1, page 253, HTML version of CDA Poster was updated to point to online links instead of the local help CHM file:

http://www.dumpanalysis.org/CDAPoster.html

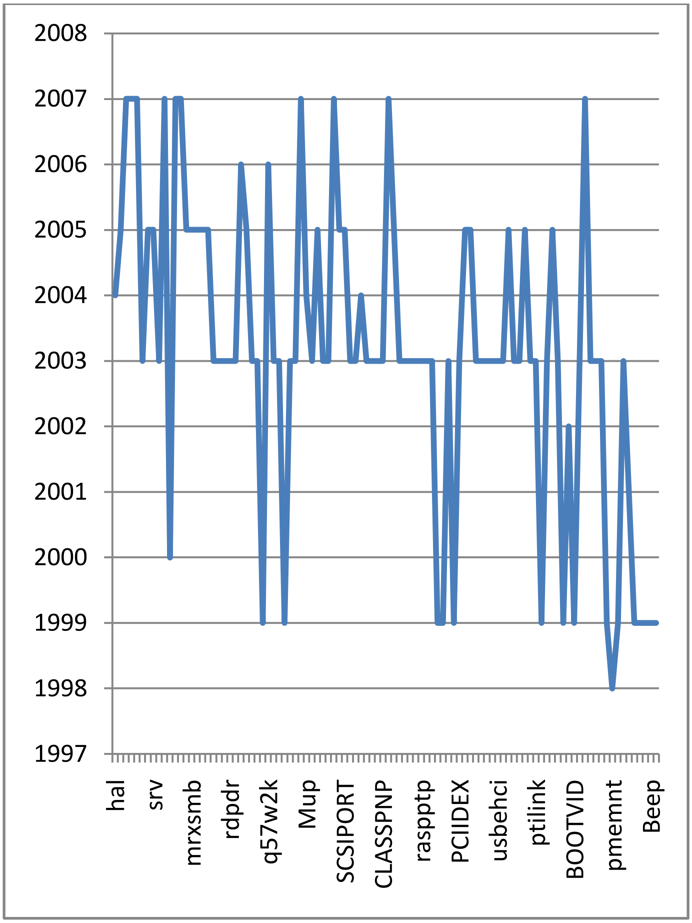
It is also featured on http://windbg.org/

HOW OLD IS YOUR APPLICATION OR SYSTEM?

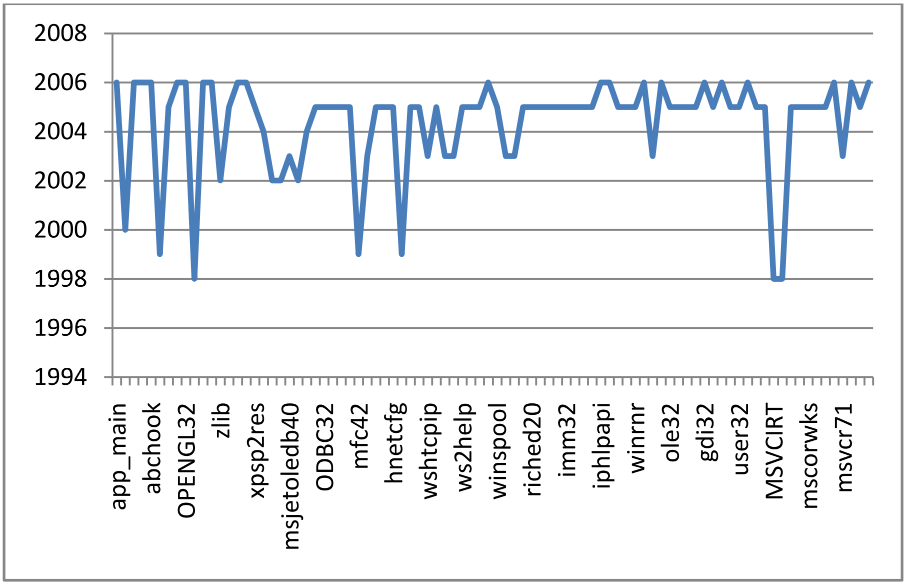
Component Age Diagram (CAD) helps to visualize and pinpoint anomalies in component timestamps. Excel helps here (Volume 1, page 635). We can import the output of **lmt** WinDbg command and get these graphs where peaks can be used to identify old modules. For example, here is a CAD from my Windows Vista SP1 running on Mac-Mini:



Here is another CAD from Windows 2000 server where the oldest driver can be easily identified:



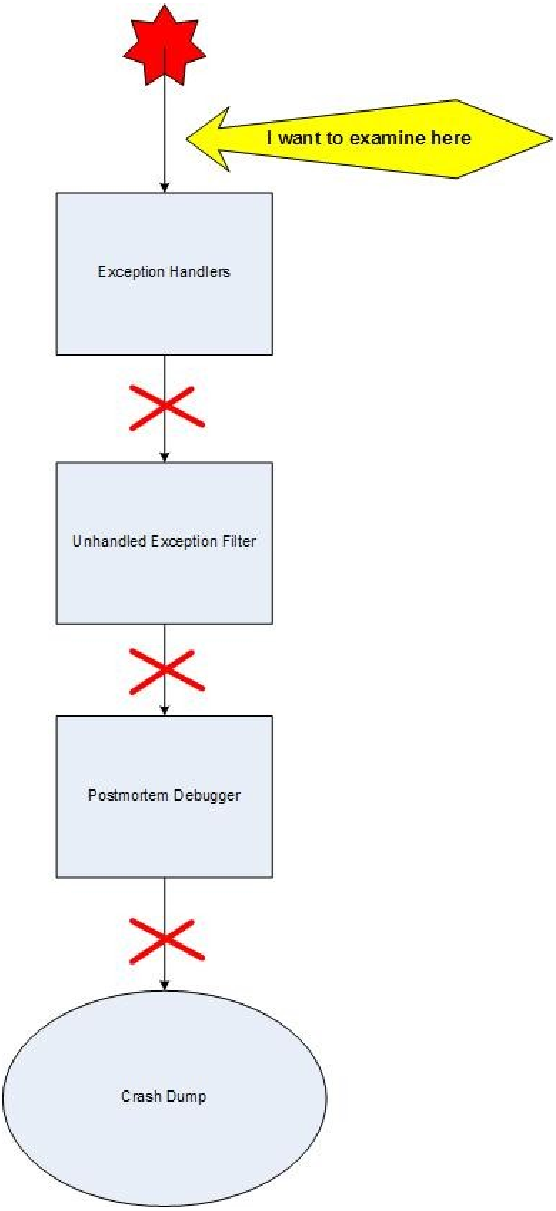
The following CAD diagram is created from **lmt** output used for **Module Variety** pattern example (Volume 1, page 310):



DEMYSTIFYING FIRST-CHANCE EXCEPTIONS

There is a bit of confusion around the division of exceptions between the first- and second-chance. For example, one question often asked is whether saving crash dumps on first-chance exceptions should be disabled or enabled. Let's clarify this issue.

First, we can say that the concept of first-chance exceptions is purely debugger-related. There is only one exception that happens when we access an invalid address, for example. However, that exception may be handled or may not be handled by exception handlers. Or it might be handled in peculiar way and terminate the thread that caused the exception. If it wasn't handled then an unhandled exception filter may be called. The default one might launch a postmortem debugger (or any process that can read process memory) to save a postmortem memory dump. Any thread can replace the default filter with a custom exception filter that may also do peculiar things and quietly terminate or exit. Even the properly configured postmortem debugger can fail to save a dump file. All these possibilities complicate the issue. Therefore we have this question: how can we catch the exception and examine the process state as earlier as possible, before the execution flow goes through the exception handling mechanism?



Here we have the concept of the first chance exception dispatched to the attached user-mode debugger. if it wasn't handled by the debugger we would have had the same exception but called the second chance that was dispatched to the same debugger again. We see that it has nothing to do with the postmortem debugger although the attached live debugger can save crash dump files too. This is what ADPlus does, for example.

.NET MANAGED CODE ANALYSIS IN COMPLETE MEMORY DUMPS

It is possible to analyze managed code in complete memory dumps. We just need to switch to the process in question and load SOS DLL (if memory dumps are from 64-bit Windows we need to run 64-bit WinDbg because it needs to load 64-bit SOS from Microsoft.NET \ Framework64 \ vX.X.XXXXX folder).

Here is some command output from a complete memory dump generated from **SystemDump** (Volume 1, page 646) when running TestDefaultDebugger.NET application (Volume 1, page 641) where we try to find and disassemble IL code for this function (some output is shown in smaller font for visual clarity):

```
namespace WindowsApplication1
{
  public partial class Form1 : Form
  {
    ...
    private void button1_Click(object sender, EventArgs e)
    {
       System.Collections.Stack stack = new System.Collections.Stack();

       stack.Pop();
    }
  }
}
```

```
Loading Dump File [C:\W2K3\MEMORY_NET.DMP]
Kernel Complete Dump File: Full address space is available

Symbol search path is:
srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows Server 2003 Kernel Version 3790 (Service Pack 2) UP Free x64
Product: Server, suite: Enterprise TerminalServer
Built by: 3790.srv03_sp2_gdr.070321-2337
Kernel base = 0xfffff800`01000000 PsLoadedModuleList = 0xfffff800`01198b00

kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
...
PROCESS fffffadfe7287610
    SessionId: 0  Cid: 0ad8    Peb: 7fffffdf000  ParentCid: 0acc
    DirBase: 12cd9000  ObjectTable: fffffa800163c260  HandleCount: 114.
    Image: TestDefaultDebugger.NET.exe

PROCESS fffffadfe67905a0
    SessionId: 0  Cid: 085c    Peb: 7fffffd4000  ParentCid: 0acc
    DirBase: 232e2000  ObjectTable: fffffa8000917e10  HandleCount:  55.
    Image: SystemDump.exe
```

```
kd> .process /r /p fffffadfe7287610
Implicit process is now fffffadf`e7287610
Loading User Symbols

kd> .loadby sos mscorwks

kd> !threads

ThreadCount: 2
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                              PreEmptive
                              Lock
       ID OSID        ThreadOBJ     State   GC     GC Alloc
Context                  Domain           Count APT Exception
       1  a94 0000000000161150      6020
Enabled  0000000000000000:0000000000000000 000000000014ccb0     0 STA
       2  604 00000000001688b0      b220
Enabled  0000000000000000:0000000000000000 000000000014ccb0     0 MTA
(Finalizer)

kd> !process fffffadfe7287610 4
PROCESS fffffadfe7287610
    SessionId: 0  Cid: 0ad8    Peb: 7fffffdf000  ParentCid: 0acc
    DirBase: 12cd9000  ObjectTable: fffffa800163c260  HandleCount: 114.
    Image: TestDefaultDebugger.NET.exe

        THREAD fffffadfe668cbf0  Cid 0ad8.0a94  Teb: 000007fffffdd000 Win32Thread:
fffff97ff4df2830 WAIT
        THREAD fffffadfe727e6d0  Cid 0ad8.0f54  Teb: 000007fffffdb000 Win32Thread:
0000000000000000 WAIT
        THREAD fffffadfe72d5bf0  Cid 0ad8.0604  Teb: 000007fffffd9000 Win32Thread:
0000000000000000 WAIT
        THREAD fffffadfe679cbf0  Cid 0ad8.06b0  Teb: 000007fffffd7000 Win32Thread:
0000000000000000 WAIT
        THREAD fffffadfe67d23d0  Cid 0ad8.0b74  Teb: 000007fffffd5000 Win32Thread:
fffff97ff4b99010 WAIT

kd> !EEHeap -gc
Number of GC Heaps: 1
generation 0 starts at 0x0000000002a41030
generation 1 starts at 0x0000000002a41018
generation 2 starts at 0x0000000002a41000
ephemeral segment allocation context: (0x0000000002a8d528, 0x0000000002a8dfe8)
         segment            begin          allocated             size
00000000001a1260 0000064274e28f60  0000064274e5f610 0x00000000000366b0(222896)
00000000001a1070 000006427692ffe8  000006427695af20 0x000000000002af38(175928)
0000000000164f60 00000642787c7380  0000064278809150 0x0000000000041dd0(269776)
0000000002a40000 0000000002a41000  0000000002a8dfe8 0x000000000004cfe8(315368)
Large object heap starts at 0x0000000012a41000
         segment            begin          allocated             size
0000000012a40000 0000000012a41000  0000000012a4e738 0x000000000000d738(55096)
Total Size           0xfdad8(1039064)
------------------------------
GC Heap Size           0xfdad8(1039064)
```

```
kd> !gchandles
Bad MethodTable for Obj at 0000000002a7a7b8
Bad MethodTable for Obj at 0000000002a7a750
Bad MethodTable for Obj at 0000000002a445b0
GC Handle Statistics:
Strong Handles: 25
Pinned Handles: 7
Async Pinned Handles: 0
Ref Count Handles: 1
Weak Long Handles: 30
Weak Short Handles: 63
Other Handles: 0
Statistics:
              MT    Count    TotalSize Class Name
...
0000064280016580        1          464 WindowsApplication1.Form1
...

kd> !dumpmt -md 0000064280016580
EEClass: 0000064280143578
Module: 0000064280012e00
Name: WindowsApplication1.Form1
mdToken: 02000002  (C:\TestDefaultDebugger.NET.exe)
BaseSize: 0×1d0
ComponentSize: 0×0
Number of IFaces in IFaceMap: 15
Slots in VTable: 375
—————————————-
MethodDesc Table
           Entry       MethodDesc      JIT Name
...
0000064280150208 00000642800164d0      JIT
WindowsApplication1.Form1.InitializeComponent()
0000064280150210 00000642800164e0      JIT
WindowsApplication1.Form1..ctor()
0000064280150218 00000642800164f0     NONE
WindowsApplication1.Form1.button1_Click(System.Object, System.EventArgs)

kd> !dumpil 00000642800164f0
ilAddr = 00000000004021bc
IL_0000: newobj System.Collections.Stack::.ctor
IL_0005: stloc.0
IL_0006: ldloc.0
IL_0007: callvirt System.Collections.Stack::Pop
IL_000c: pop
IL_000d: ret
```

WHO OPENED THAT FILE?

Sometimes certain files are opened but not closed when not needed and this prevents other applications and users from using, deleting or replacing them. Sometimes on behalf of certain API calls another process opens them. One of the questions I was asked recently is how to find that process. The answer is very simple: we just need to list all handles from all processes and search for that file name there. This works in kernel and complete memory dumps and also in live kernel debugging session including its local kernel debugging variant. The following WinDbg command lists all objects (we need to open a log because the output is usually of several megabytes):

```
3: kd> !for_each_process "!handle 0 3 @#Process"

!handle 0 3 @#Process
processor number 3, process 8a392818
PROCESS 8a392818  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid:
0000
    DirBase: bfc51000  ObjectTable: e1001e00  HandleCount: 2650.
    Image: System

Handle table at e16a3000 with 2650 Entries in use
0004: Object: 8a392818  GrantedAccess: 001f0fff Entry: e1004008
Object: 8a392818  Type: (8a392e70) Process
    ObjectHeader: 8a392800 (old version)
        HandleCount: 3  PointerCount: 235

0008: Object: 8a391db0  GrantedAccess: 00000000 Entry: e1004010
Object: 8a391db0  Type: (8a392ca0) Thread
    ObjectHeader: 8a391d98 (old version)
        HandleCount: 1  PointerCount: 1

000c: Object: e101bca0  GrantedAccess: 00000000 Entry: e1004018
Object: e101bca0  Type: (8a37db00) Key
    ObjectHeader: e101bc88 (old version)
        HandleCount: 1  PointerCount: 3
        Directory Object: 00000000  Name: \REGISTRY

...

1fac: Object: 88ec72b0  GrantedAccess: 00000003 (Protected) Entry:
e1ed7f58
Object: 88ec72b0  Type: (8a36f900) File
    ObjectHeader: 88ec7298 (old version)
        HandleCount: 1  PointerCount: 2
        Directory Object: 00000000  Name: \Documents and
Settings\MyUserName\NTUSER.DAT {HarddiskVolume1}

...
```

```
07fc: Object: e1000768  GrantedAccess: 00000003 Entry: e2fefff8
Object: e1000768  Type: (8a387e70) KeyedEvent
    ObjectHeader: e1000750 (old version)
        HandleCount: 273  PointerCount: 274
        Directory Object: e1001a48  Name: CritSecOutOfMemoryEvent

processor number 3, process 8873f3b8
PROCESS 8873f3b8  SessionId: 6  Cid: 4c1c    Peb: 7ffdf000  ParentCid:
42bc
    DirBase: 49dbb000  ObjectTable: e325f788  HandleCount:  90.
    Image: PROFLWIZ.EXE

Handle table at e36c3000 with 90 Entries in use
0004: Object: e1000768  GrantedAccess: 00000003 Entry: e36c3008
Object: e1000768  Type: (8a387e70) KeyedEvent
    ObjectHeader: e1000750 (old version)
        HandleCount: 273  PointerCount: 274
        Directory Object: e1001a48  Name: CritSecOutOfMemoryEvent

...
```

IN SEARCH OF LOST CID

Paraphrasing the title of Marcel Proust's "In Search of Lost Time" 6-volume classic we can say there is timeless empirical knowledge. One is target CID (Client ID, PID:TID) for RPC calls. We just need to search for even 16-bit numbers and compare them with the list of available PIDs. The example can be found on ntdebugging blog:

Tracking Down a Multi-Process Deadlock
http://blogs.msdn.com/ntdebugging/archive/2008/07/01/tracking-down-a-multi-process-deadlock.aspx

Actually the second dword after PID can contain even 16-bit TID number as can be seen from another example (stack trace is shown in smaller font for visual clarity):

```
1: kd> kv
ChildEBP RetAddr  Args to Child
...
00faf828 7778c38b 00faf8f0 00faf9f0 06413b54
ole32!CRpcChannelBuffer::SwitchAptAndDispatchCall+0x112
00faf908 776c0565 06413b54 00fafa00 00faf9f0
ole32!CRpcChannelBuffer::SendReceive2+0xd3
00faf974 776c04fa 06413b54 00fafa00 00faf9f0 ole32!CAptRpcChnl::SendReceive+0xab
00faf9c8 77ce247f 06413b54 00fafa00 00faf9f0 ole32!CCtxComChnl::SendReceive+0×1a9
00faf9e4 77ce252f 03213bdc 00fafa2c 0600016e RPCRT4!NdrProxySendReceive+0×43
00fafdcc 77ce25a6 763050e8 76306bba 00fafe04 RPCRT4!NdrClientCall2+0×206
00fafdec 77c64f87 0000000c 00000005 00fafe3c RPCRT4!ObjectStublessClient+0×8b
00fafdfc 7784ba75 03213bdc 03235858 03235850 RPCRT4!ObjectStubless+0xf
...

1: kd> dpp 06413b54 l8
06413b54  77672418 7778bcdf ole32!CRpcChannelBuffer::QueryInterface
06413b58  776723e8 777267f5 ole32!CRpcChannelBuffer::QueryInterface
06413b5c  00000003
06413b60  00000002
06413b64  0743fde8 0316f080
06413b68  07676528 031d5ad0
06413b6c  064c9c80 064c9d00
06413b70  078843c0 00000000

1: kd> dd 064c9c80 l4
064c9c80  064c9d00 064c9c00 00003108 00001dac ; PID TID
```

LARGE HEAP ALLOCATIONS

To check heap data structures and how they change in the case of heap memory leaks (Volume 1, page 356) we can write the very small C program that allocates memory in a loop using malloc function. If we run it VM size grows very fast and we save process memory dumps at 200Mb and 500Mb. When checking heap segments we see that they had not increased although the process was allocating 0×1000000 chunks of heap memory:

```
0:000> !heap 0 0
Index   Address  Name      Debugging options enabled
  1:   00260000
    Segment at 0000000000260000 to 0000000000360000 (00008000 bytes
committed)
  2:   00360000
    Segment at 0000000000360000 to 0000000000370000 (00004000 bytes
committed)
  3:   00440000
    Segment at 0000000000440000 to 0000000000450000 (00010000 bytes
committed)
    Segment at 0000000000450000 to 0000000000550000 (00021000 bytes
committed)
  4:   00560000
    Segment at 0000000000560000 to 0000000000570000 (00010000 bytes
committed)
    Segment at 0000000000570000 to 0000000000670000 (0003a000 bytes
committed)
```

It can be puzzling because inspection of virtual memory shows those chunks as belonging to heap regions:

```
0:000> !address
...
    0000000009700000 : 0000000009700000 - 0000000001002000
                    Type     00020000 MEM_PRIVATE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageHeap
                    Handle   0000000000560000
    000000000a702000 : 000000000a702000 - 000000000000e000
                    Type     00000000
                    Protect  00000001 PAGE_NOACCESS
                    State    00010000 MEM_FREE
                    Usage    RegionUsageFree
    000000000a710000 : 000000000a710000 - 0000000001002000
                    Type     00020000 MEM_PRIVATE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageHeap
                    Handle   0000000000560000
    000000000b712000 : 000000000b712000 - 0000000004aee000
                    Type     00000000
                    Protect  00000001 PAGE_NOACCESS
                    State    00010000 MEM_FREE
                    Usage    RegionUsageFree
...
```

However large allocations for a process heap go to a separate linked list:

```
0:000> !peb
PEB at 000007fffffdb000

0:000> dt _PEB 000007fffffdb000
ntdll!_PEB
...
   +0x0f0 ProcessHeaps     : 0×00000000`77fa3460  -> 0×00000000`00260000
...

0:000> dq 0×00000000`77fa3460
00000000`77fa3460  00000000`00260000 00000000`00360000
00000000`77fa3470  00000000`00440000 00000000`00560000
00000000`77fa3480  00000000`00000000 00000000`00000000
00000000`77fa3490  00000000`00000000 00000000`00000000
00000000`77fa34a0  00000000`00000000 00000000`00000000
00000000`77fa34b0  00000000`00000000 00000000`00000000
00000000`77fa34c0  00000000`00000000 00000000`00000000
00000000`77fa34d0  00000000`00000000 00000000`00000000

0:000> dt _HEAP 00000000`00260000
ntdll!_HEAP
...
   +0×090 VirtualAllocdBlocks : _LIST_ENTRY [ 0×00000000`00260090 -
0×260090 ]
...

0:000> dl 00000000`00260000+90 10 2
00000000`00260090  00000000`00260090 00000000`00260090

0:000> dl 00000000`00360000+90 10 2
00000000`00360090  00000000`00360090 00000000`00360090

0:000> dl 00000000`00440000+90 10 2
00000000`00440090  00000000`00440090 00000000`00440090
```

```
0:000> dl 00000000`00560000+90 10 2
00000000`00560090  00000000`00670000 00000000`0a710000
00000000`00670000  00000000`01680000 00000000`00560090
00000000`01680000  00000000`02690000 00000000`00670000
00000000`02690000  00000000`036a0000 00000000`01680000
00000000`036a0000  00000000`046b0000 00000000`02690000
00000000`046b0000  00000000`056c0000 00000000`036a0000
00000000`056c0000  00000000`066d0000 00000000`046b0000
00000000`066d0000  00000000`076e0000 00000000`056c0000
00000000`076e0000  00000000`086f0000 00000000`066d0000
00000000`086f0000  00000000`09700000 00000000`076e0000
00000000`09700000  00000000`0a710000 00000000`086f0000
00000000`0a710000  00000000`00560090 00000000`09700000
```

We see that the last process heap has large allocations directly from virtual memory, for example:

```
0:000> !address 00000000`0a710000
    000000000a710000 : 000000000a710000 - 0000000001002000
                    Type      00020000 MEM_PRIVATE
                    Protect   00000004 PAGE_READWRITE
                    State     00001000 MEM_COMMIT
                    Usage     RegionUsageHeap
                    Handle    0000000000560000
```

Actually if we use heap statistics option for **!heap** command we see these large allocations (shown in smaller font for visual clarity):

```
0:000> !heap -s
LFH Key                   : 0x000000a4e8aa078c
          Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                            (k)     (k)    (k)     (k) length      blocks cont. heap
0000000000260000 00000002    1024      32     32      7     1     1    0      0    L
0000000000360000 00008000      64      16     16     12     1     1    0      0
0000000000440000 00001002    1088     196    196      4     1     1    0      0   LFH
Virtual block: 0000000000670000 - 0000000000670000
Virtual block: 0000000001680000 - 0000000001680000
Virtual block: 0000000002690000 - 0000000002690000
Virtual block: 00000000036a0000 - 00000000036a0000
Virtual block: 00000000046b0000 - 00000000046b0000
Virtual block: 00000000056c0000 - 00000000056c0000
Virtual block: 00000000066d0000 - 00000000066d0000
Virtual block: 00000000076e0000 - 00000000076e0000
Virtual block: 00000000086f0000 - 00000000086f0000
Virtual block: 0000000009700000 - 0000000009700000
Virtual block: 000000000a710000 - 000000000a710000
0000000000560000 00001002    1088     296    296     18     3     1   11      0   LFH
```

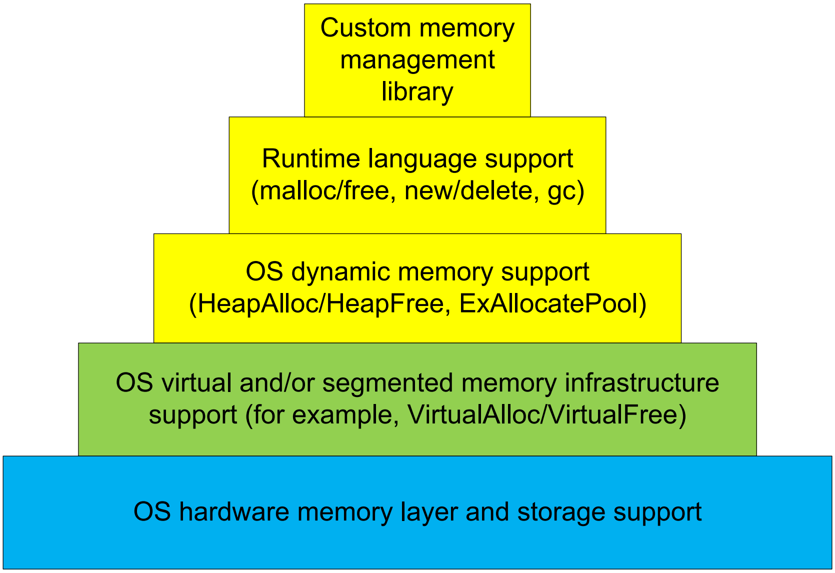
The dump file can be downloaded from FTP to play with:
ftp://dumpanalysis.org/pub/LargeHeapAllocations.zip

FIRST-ORDER AND SECOND-ORDER MEMORY LEAKS

Dynamic memory allocation architecture usually consists of different layers where the lower layers provide support for the upper ones and some general layers can be combined or omitted like in TCP/IP implementation of OSI reference model:

1a. Custom memory management library.
1b. Runtime language support (malloc/free, new/delete, gc).
1c. OS dynamic memory support (HeapAlloc/HeapFree, ExAllocatePool/ExFreePool).
2. OS virtual and/or segmented memory infrastructure support (VirtualAlloc/VirtualFree).
3. OS hardware memory layer and storage support.

We can call it **DMI** (**D**ynamic **M**emory **I**nfrastructure) and this can be summarized on the following diagram:


Custom memory management library
Runtime language support (malloc/free, new/delete, gc)
OS dynamic memory support (HeapAlloc/HeapFree, ExAllocatePool)
OS virtual and/or segmented memory infrastructure support (for example, VirtualAlloc/VirtualFree)
OS hardware memory layer and storage support

First-order memory leaks happen when an application uses layers 1a, 1b or 1c and doesn't free allocated memory. Typical pattern examples include:

- **Memory Leak** (process heap, Volume 1, page 356)
- **Memory Leak** (.NET heap, Volume 1, page 371)
- **Insufficient Memory** (kernel pool, Volume 1, page 440)

- **Insufficient Memory** (handle leak, Volume 1, page 327)

What we cover here are second-order leaks in layers 2 an 3. These include cases when an application frees memory but the underlying supporting layer doesn't do it due to its design or factors like fragmentation. Consider an example of a Windows service that undergone committed memory increase from 600Mb to almost 1.2Gb during peak hours and then remained at that size even when no activity happened afterwards. We can examine virtual memory statistics using **!address** WinDbg command from 3 sampled memory dumps:

Before peak hours:

```
-------------------- Usage SUMMARY --------------------------
    TotSize (      KB)   Pct(Tots) Pct(Busy)   Usage
    734d000 (  118068) : 05.63%    07.50%    : RegionUsageIsVAD
   1ff11000 (  523332) : 24.96%    00.00%    : RegionUsageFree
    4352000 (   68936) : 03.29%    04.38%    : RegionUsageImage
    5a00000 (   92160) : 04.39%    05.86%    : RegionUsageStack
      5a000 (     360) : 00.02%    00.02%    : RegionUsageTeb
   4efe3000 ( 1294220) : 61.72%    82.24%    : RegionUsageHeap
          0 (       0) : 00.00%    00.00%    : RegionUsagePageHeap
       1000 (       4) : 00.00%    00.00%    : RegionUsagePeb
       1000 (       4) : 00.00%    00.00%    : RegionUsageProcessParametrs
       1000 (       4) : 00.00%    00.00%    : RegionUsageEnvironmentBlock
       Tot: 7fff0000 (2097088 KB) Busy: 600df000 (1573756 KB)

-------------------- Type SUMMARY --------------------------
    TotSize (      KB)   Pct(Tots)  Usage
   1ff11000 (  523332) : 24.96%   : <free>
    4352000 (   68936) : 03.29%   : MEM_IMAGE
     b78000 (   11744) : 00.56%   : MEM_MAPPED
   5b215000 ( 1493076) : 71.20%   : MEM_PRIVATE

-------------------- State SUMMARY --------------------------
    TotSize (      KB)   Pct(Tots)  Usage
   25e50000 (  620864) : 29.61%   : MEM_COMMIT
   1ff11000 (  523332) : 24.96%   : MEM_FREE
    3a28f000 (  952892) : 45.44%   : MEM_RESERVE
```

During peak hours:

```
-------------------- Usage SUMMARY --------------------------
  TotSize ( KB) Pct(Tots) Pct(Busy) Usage
  734d000 ( 118068) : 05.63% 07.49% : RegionUsageIsVAD
  1fd0f000 ( 521276) : 24.86% 00.00% : RegionUsageFree
  4352000 ( 68936) : 03.29% 04.37% : RegionUsageImage
  5c00000 ( 94208) : 04.49% 05.98% : RegionUsageStack
  5c000 ( 368) : 00.02% 00.02% : RegionUsageTeb
```

```
4efe3000 ( 1294220) : 61.72% 82.13% : RegionUsageHeap
  0 ( 0) : 00.00% 00.00% : RegionUsagePageHeap
  1000 ( 4) : 00.00% 00.00% : RegionUsagePeb
  1000 ( 4) : 00.00% 00.00% : RegionUsageProcessParametrs
  1000 ( 4) : 00.00% 00.00% : RegionUsageEnvironmentBlock
  Tot: 7fff0000 (2097088 KB) Busy: 602e1000 (1575812 KB)

-------------------- Type SUMMARY --------------------------
  TotSize ( KB) Pct(Tots) Usage
  1fd0f000 ( 521276) : 24.86% :
  4352000 ( 68936) : 03.29% : MEM_IMAGE
  b78000 ( 11744) : 00.56% : MEM_MAPPED
  5b417000 ( 1495132) : 71.30% : MEM_PRIVATE

-------------------- State SUMMARY --------------------------
  TotSize ( KB) Pct(Tots) Usage
41498000 ( 1069664) : 51.01% : MEM_COMMIT
  1fd0f000 ( 521276) : 24.86% : MEM_FREE
  1ee49000 ( 506148) : 24.14% : MEM_RESERVE
```

After peak hours:

```
-------------------- Usage SUMMARY --------------------------
    TotSize (      KB)   Pct(Tots) Pct(Busy)   Usage
    734d000 (  118068) : 05.63%    07.49%    : RegionUsageIsVAD
   1fd0f000 (  521276) : 24.86%    00.00%    : RegionUsageFree
    4352000 (   68936) : 03.29%    04.37%    : RegionUsageImage
    5c00000 (   94208) : 04.49%    05.98%    : RegionUsageStack
      5c000 (     368) : 00.02%    00.02%    : RegionUsageTeb
   4efe3000 ( 1294220) : 61.72%    82.13%    : RegionUsageHeap
          0 (       0) : 00.00%    00.00%    : RegionUsagePageHeap
       1000 (       4) : 00.00%    00.00%    : RegionUsagePeb
       1000 (       4) : 00.00%    00.00%    : RegionUsageProcessParametrs
       1000 (       4) : 00.00%    00.00%    : RegionUsageEnvironmentBlock
       Tot: 7fff0000 (2097088 KB) Busy: 602e1000 (1575812 KB)

-------------------- Type SUMMARY --------------------------
    TotSize (      KB)   Pct(Tots)  Usage
   1fd0f000 (  521276) : 24.86%   : <free>
    4352000 (   68936) : 03.29%   : MEM_IMAGE
     b78000 (   11744) : 00.56%   : MEM_MAPPED
   5b417000 ( 1495132) : 71.30%   : MEM_PRIVATE

-------------------- State SUMMARY --------------------------
    TotSize (      KB)   Pct(Tots)  Usage
   4505d000 ( 1130868) : 53.93%   : MEM_COMMIT
   1fd0f000 (  521276) : 24.86%   : MEM_FREE
   1b284000 (  444944) : 21.22%   : MEM_RESERVE
```

We see that in every memory dump the amount of process heap is the same 1.2Gb but during peak hours the amount of committed memory increased by 20% and remained the same even after. At the same time if we look at process heap statistics we would see the increase of free heap KB and blocks and this means that allocated memory was freed after peak hours but underlying virtual memory ranges were not de-committed and fragmentation increased by 25%.

Before peak hours:

```
0:000> !heap -s
LFH Key : 0x07262959
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock
                    (k)     (k)    (k)     (k) length      blocks cont.
...
00310000 00001002 1255320 512712 1177236 260583 45362 41898 2 3751a5 L
  External fragmentation 50 % (45362 free blocks)
  Virtual address fragmentation 56 % (41898 uncommited ranges)
  Lock contention 3625381
...
```

During peak hours:

```
0:000> !heap -s
LFH Key : 0x07262959
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock
                    (k)     (k)    (k)     (k) length      blocks cont.
...
00310000 00001002 1255320 961480 1249548 105378 0 16830 2 453093 L
  Virtual address fragmentation 23 % (16830 uncommited ranges)
  Lock contention 4534419
...
```

After peak hours:

```
0:000> !heap -s
LFH Key : 0x07262959
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock
                    (k)     (k)    (k)     (k) length      blocks cont.
...
00310000 00001002 1255320 1022648 1224344 772682 264787 17512 2 580634 L
  External fragmentation 75 % (264787 free blocks)
  Virtual address fragmentation 16 % (17512 uncommited ranges)
  Lock contention 5768756
...
```

Another example would be custom memory management library that by design never releases virtual memory allocated to accommodate the increased number of allocation requests after all of them are freed.

HOOKED MODULES

In **Hooked Functions** pattern (Volume 1, page 468) we used **!chkimg** WinDbg command. To check all modules we can use this simple command:

```
!for_each_module !chkimg -lo 50 -d !${@#ModuleName} -v
```

For example:

```
0:000:x86> !for_each_module !chkimg -lo 50 -d !${@#ModuleName} -v
...
Scanning section:    .text
Size: 74627
Range to scan: 71c01000-71c13383
    71c02430-71c02434  5 bytes - WS2_32!WSASend
 [ 8b ff 55 8b ec:e9 cb db 1c 0d ]
    71c0279b-71c0279f  5 bytes - WS2_32!select (+0x36b)
 [ 6a 14 68 58 28:e9 60 d8 15 0d ]
    71c0290e-71c02912  5 bytes - WS2_32!WSASendTo (+0x173)
 [ 8b ff 55 8b ec:e9 ed d6 1b 0d ]
    71c02cb2-71c02cb6  5 bytes - WS2_32!closesocket (+0x3a4)
 [ 8b ff 55 8b ec:e9 49 d3 19 0d ]
    71c02e12-71c02e16  5 bytes - WS2_32!WSAIoctl (+0x160)
 [ 8b ff 55 8b ec:e9 e9 d1 1e 0d ]
    71c02ec2-71c02ec6  5 bytes - WS2_32!send (+0xb0)
 [ 8b ff 55 8b ec:e9 39 d1 14 0d ]
    71c02f7f-71c02f83  5 bytes - WS2_32!recv (+0xbd)
 [ 8b ff 55 8b ec:e9 7c d0 17 0d ]
    71c03c04-71c03c08  5 bytes - WS2_32!WSAGetOverlappedResult (+0xc85)
 [ 8b ff 55 8b ec:e9 f7 c3 1f 0d ]
    71c03c75-71c03c79  5 bytes - WS2_32!recvfrom (+0x71)
 [ 8b ff 55 8b ec:e9 86 c3 16 0d ]
    71c03d14-71c03d18  5 bytes - WS2_32!sendto (+0x9f)
 [ 8b ff 55 8b ec:e9 e7 c2 13 0d ]
    71c03da8-71c03dac  5 bytes - WS2_32!WSACleanup (+0x94)
 [ 8b ff 55 8b ec:e9 53 c2 25 0d ]
    71c03f38-71c03f3c  5 bytes - WS2_32!WSASocketW (+0x190)
 [ 6a 20 68 08 40:e9 c3 c0 11 0d ]
    71c0446a-71c0446e  5 bytes - WS2_32!connect (+0x532)
 [ 8b ff 55 8b ec:e9 91 bb 18 0d ]
    71c04f3b-71c04f3f  5 bytes - WS2_32!WSAStartup (+0xad1)
 [ 6a 14 68 60 50:e9 c0 b0 29 0d ]
    71c06162-71c06166  5 bytes - WS2_32!shutdown (+0x1227)
 [ 8b ff 55 8b ec:e9 99 9e 12 0d ]
    71c069e9-71c069ed  5 bytes - WS2_32!WSALookupServiceBeginW (+0x887)
 [ 8b ff 55 8b ec:e9 12 96 0f 0d ]
    71c06c91-71c06c95  5 bytes - WS2_32!WSALookupServiceNextW (+0x2a8)
 [ 8b ff 55 8b ec:e9 6a 93 10 0d ]
    71c06ecd-71c06ed1  5 bytes - WS2_32!WSALookupServiceEnd (+0x23c)
 [ 8b ff 55 8b ec:e9 2e 91 0e 0d ]
    71c090be-71c090c2  5 bytes - WS2_32!WSAEventSelect (+0x21f1)
 [ 8b ff 55 8b ec:e9 3d 6f 20 0d ]
    71c09129-71c0912d  5 bytes - WS2_32!WSACreateEvent (+0x6b)
 [ 33 c0 50 50 6a:e9 d2 6e 22 0d ]
    71c0938e-71c09392  5 bytes - WS2_32!WSACloseEvent (+0x265)
 [ 6a 0c 68 c8 93:e9 6d 6c 24 0d ]
    71c093d9-71c093dd  5 bytes - WS2_32!WSAWaitForMultipleEvents (+0x4b)
 [ 8b ff 55 8b ec:e9 22 6c 1a 0d ]
    71c093ea-71c093ee  5 bytes - WS2_32!WSAEnumNetworkEvents (+0x11)
 [ 8b ff 55 8b ec:e9 11 6c 21 0d ]
    71c09480-71c09484  5 bytes - WS2_32!WSARecv (+0x96)
 [ 8b ff 55 8b ec:e9 7b 6b 1d 0d ]
    71c0eecb-71c0eecf  5 bytes - WS2_32!WSACancelAsyncRequest (+0x5a4b)
 [ 8b ff 55 8b ec:e9 30 11 26 0d ]
    71c10d39-71c10d3d  5 bytes - WS2_32!WSAAsyncSelect (+0x1e6e)
 [ 8b ff 55 8b ec:e9 c2 f2 26 0d ]
    71c10ee3-71c10ee7  5 bytes - WS2_32!WSAConnect (+0x1aa)
 [ 8b ff 55 8b ec:e9 18 f1 22 0d ]
    71c10f9f-71c10fa3  5 bytes - WS2_32!WSAAccept (+0xbc)
 [ 8b ff 55 8b ec:e9 5c f0 27 0d ]
Total bytes compared: 74627(100%)
Number of errors: 140
140 errors : !WS2_32 (71c02430-71c10fa3)
...
```

WAIT CHAIN (EXECUTIVE RESOURCES)

The most common and easily detectable example of **Wait Chain** pattern (Volume 1, page 481) in kernel and complete memory dumps is when objects are executive resources (Volume 1, page 323). In some complex cases we can even have multiple wait chains. For example, in the output of **!locks** WinDbg command below we can find at least three wait chains marked in bold, italics and bold italics:

883db310 -> 8967d020 -> 889fa230
89a74228 -> 883ad4e0 -> 88d7a3e0
88e13990 -> 899da538 -> 8805fac8

The manual procedure to figure chains is simple. Pick up a thread marked with <*>. This is one that currently holds the resource. See what threads are waiting on exclusive access to that resource. Then search for other occurrences of <*> thread to see whether it is waiting exclusively for another resource blocked by some other thread and so on.

```
1: kd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****

Resource @ 0x8b4425c8    Shared 1 owning threads
    Contention Count = 45
    NumberOfSharedWaiters = 43
    NumberOfExclusiveWaiters = 2
     Threads: 8967d020-01<*> 88748b10-01    88b8b020-01    8b779ca0-01
              88673248-01    8b7797c0-01    889c8358-01    8b777b40-01
              8b7763f0-01    8b776b40-01    8b778b40-01    8841b020-01
              8b7788d0-01    88cc7b00-01    8b776020-01    8b775020-01
              87f6d6c8-01    8816b020-01    8b7773f0-01    89125db0-01
              8b775db0-01    8846a020-01    87af2238-01    8b7792e0-01
              8b777020-01    87f803b8-01    8b7783f0-01    8b7768d0-01
              8b778020-01    8b779a30-01    8b778660-01    8943a020-01
              88758020-01    8b777db0-01    88ee3590-01    896f3020-01
              89fc4b98-01    89317938-01    8867f1e0-01    89414118-01
              88e989a8-01    88de5b70-01    88c4b588-01    8907dd48-01
     Threads Waiting On Exclusive Access:
              883db310       8907d280

Resource @ 0x899e27ac    Exclusively owned
    Contention Count = 1
    NumberOfExclusiveWaiters = 1
     Threads: 889fa230-01<*>
     Threads Waiting On Exclusive Access:
              8967d020
```

```
Resource @ 0x881a38f8    Exclusively owned
    Contention Count = 915554
    NumberOfExclusiveWaiters = 18
     Threads: 883ad4e0-01<*>
     Threads Waiting On Exclusive Access:
              89a74228       8844c630       8955e020       891aa440
              8946a270       898b7ab0       89470d20       881e5760
              8b594af8       88dce020       899df328       8aa86900
              897ff020       8920adb0       8972b1c0       89657c70
              88bcc868       88cb0cb0

Resource @ 0x88a8d5b0    Exclusively owned
    Contention Count = 39614
    NumberOfExclusiveWaiters = 3
     Threads: 88d7a3e0-01<*>
     Threads Waiting On Exclusive Access:
              883ad4e0       89a5f020       87d00020

Resource @ 0x89523658    Exclusively owned
    Contention Count = 799193
    NumberOfExclusiveWaiters = 18
     Threads: 899da538-01<*>
     Threads Waiting On Exclusive Access:
              88e13990       89a11cc0       88f4b2f8       898faab8
              8b3200c0       88758468       88b289f0       89fa4a58
              88bf2510       8911a020       87feb548       8b030db0
              887ad2c8       8872e758       89665020       89129810
              886be480       898a6020

Resource @ 0x897274b0    Exclusively owned
    Contention Count = 37652
    NumberOfExclusiveWaiters = 2
     Threads: 8805fac8-01<*>
     Threads Waiting On Exclusive Access:
              899da538       88210db0

Resource @ 0x8903db88    Exclusively owned
    Contention Count = 1127998
    NumberOfExclusiveWaiters = 17
     Threads: 882c9b68-01<*>
     Threads Waiting On Exclusive Access:
              8926fdb0       8918fd18       88036430       89bc2c18
              88fca478       8856d710       882778f0       887c3240
              88ee15e0       889d3640       89324c68       8887b020
              88d826a0       8912ca08       894edb10       87e518f0
              89896688

Resource @ 0x89351430    Exclusively owned
    Contention Count = 51202
    NumberOfExclusiveWaiters = 1
     Threads: 882c9b68-01<*>
     Threads Waiting On Exclusive Access:
              88d01378
```

```
Resource @ 0x87d6b220    Exclusively owned
    Contention Count = 303813
    NumberOfExclusiveWaiters = 14
     Threads: 8835d020-01<*>
     Threads Waiting On Exclusive Access:
              8b4c52a0       891e2ae0       89416888       897968e0
              886e58a0       89b327d8       894ba4c0       8868d648
              88a10968       89a1da88       8985a1d0       88f58a30
              89499020       89661220

Resource @ 0x88332cc8    Exclusively owned
    Contention Count = 21214
    NumberOfExclusiveWaiters = 3
     Threads: 88d15b50-01<*>
     Threads Waiting On Exclusive Access:
              88648020       8835d020       88a20ab8

Resource @ 0x8986ab80    Exclusively owned
    Contention Count = 753246
    NumberOfExclusiveWaiters = 13
     Threads: 88e6ea60-01<*>
     Threads Waiting On Exclusive Access:
              89249020       87e01d50       889fb6c8       89742cd0
              8803b6a8       888015e0       88a89ba0       88c09020
              8874d470       88d97db0       8919a2d0       882732c0
              89a9eb28

Resource @ 0x88c331c0    Exclusively owned
    Contention Count = 16940
    NumberOfExclusiveWaiters = 2
     Threads: 8b31c748-01<*>
     Threads Waiting On Exclusive Access:
              896b3390       88e6ea60

33816 total locks, 20 locks currently held
```

Now we can dump the top thread from selected wait chain to see its call stack and why it is stuck holding other threads. For example,

883db310 -> 8967d020 -> 889fa230

```
1: kd> !thread 889fa230
THREAD 889fa230  Cid 01e8.4054  Teb: 7ffac000 Win32Thread: 00000000
RUNNING on processor 3
IRP List:
    889fc008: (0006,0094) Flags: 00000a00  Mdl: 00000000
Impersonation token:  e29c1630 (Level Impersonation)
DeviceMap                 d7030620
Owning Process            89bcb480       Image:         MyService.exe
Wait Start TickCount      113612852      Ticks: 6 (0:00:00:00.093)
Context Switch Count      19326191
UserTime                  00:00:11.0765
KernelTime                18:52:35.0750
Win32 Start Address 0×0818f190
LPC Server thread working on message Id 818f190
Start Address 0×77e6b5f3
Stack Init 8ca60000 Current 8ca5fb04 Base 8ca60000 Limit 8ca5d000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr  Args to Child
8ca5fbb8 b86c1c93 c00a0006 00000003 8ca5fbfc driver!foo5+0×67
8ca5fbdc b86bdb8a 8b4742e0 00000003 8ca5fbfc driver!foo4+0×71
8ca5fc34 b86bf682 8b4742e0 889fc008 889fc078 driver!foo3+0xd8
8ca5fc50 b86bfc74 889fc008 889fc078 889fc008 driver!foo2+0×40
8ca5fc68 8081dcdf 8b45a3c8 889fc008 889fa438 driver!foo+0xd0
8ca5fc7c 808f47b7 889fc078 00000001 889fc008 nt!IofCallDriver+0×45
8ca5fc90 808f24ee 8b45a3c8 889fc008 89507e90
nt!IopSynchronousServiceTail+0×10b
8ca5fd38 80888c7c 0000025c 0000029d 00000000 nt!NtWriteFile+0×65a
8ca5fd38 7c82ed54 0000025c 0000029d 00000000 nt!KiFastCallEntry+0xfc
```

Because of huge kernel time, contention count and RUNNING status it is most probably the instance of **Spiking Thread** pattern (Volume 1, page 305) involving driver.sys called in the context of MyService.exe process.

CORRUPT DUMP

This is quite frequent pattern and usually becomes the consequence of **Truncated Dump** pattern (Volume 1, page 340). When we open such crash dumps we usually notice immediate errors in WinDbg output. We can distinguish between two classes of corrupt memory dumps: totally corrupt and partially corrupt. Total corruption is less frequent, results from invalid file header and manifests itself in an error message box with the following Win32 error:

```
Loading Dump File [C:\Documents and Settings\All Users\Application
Data\Microsoft\Dr Watson\user_corrupted.dmp]
ERROR: Directory not present in dump (RVA 0x20202020)
Could not open dump file [C:\Documents and Settings\All Users\Application
Data\Microsoft\Dr Watson\user_corrupted.dmp], Win32 error 1392
    "The file or directory is corrupted and unreadable."
```

Partially corrupt files can be loaded but some critical information is missing like the list of loaded modules and context for all or some processors. We can see lots of messages in WinDbg output like:

```
GetContextState failed, 0x80070026
Unable to get current machine context, Win32 error 0n38
```

or

```
GetContextState failed, 0x80004005
```

or

```
GetContextState failed, 0xD0000147
```

which mean:

```
?: kd> !error 0x80070026
Error code: (HRESULT) 0x80070026 (2147942438) - Reached the end of the
file.

?: kd> !error 0x80004005
Error code: (HRESULT) 0x80004005 (2147500037) - Unspecified error

?: kd> !error 0xD0000147
Error code: (NTSTATUS) 0xd0000147 (3489661255) - {No Paging File
Specified}  No paging file was specified in the system configuration.
```

However, in many such cases we can still see system information and bugcheck parameters:

```
************************************
THIS DUMP FILE IS PARTIALLY CORRUPT.
KdDebuggerDataBlock is not present or unreadable.
************************************
Unable to read PsLoadedModuleList
KdDebuggerData.KernBase < SystemRangeStart
Windows Server 2003 Kernel Version 3790 MP (4 procs) Free x86 compatible
Product: Server, suite: TerminalServer
Kernel base = 0×00000000 PsLoadedModuleList = 0×808af9c8
Debug session time: Wed Nov 21 20:29:31.373 2007 (GMT+0)
System Uptime: 0 days 0:45:02.312
Unable to read PsLoadedModuleList
KdDebuggerData.KernBase < SystemRangeStart
Loading Kernel Symbols
Unable to read PsLoadedModuleList
GetContextState failed, 0×80070026
GetContextState failed, 0×80070026
CS descriptor lookup failed
GetContextState failed, 0×80070026
GetContextState failed, 0×80070026
GetContextState failed, 0×80070026
GetContextState failed, 0×80070026
Unable to get program counter
GetContextState failed, 0×80070026
Unable to get current machine context, Win32 error 0n38
GetContextState failed, 0×80070026
GetContextState failed, 0×80070026

Use !analyze -v to get detailed debugging information.

BugCheck 20, {0, ffff, 0, 1}

***** Debugger could not find nt in module list, module list might be
corrupt, error 0x80070057.

GetContextState failed, 0x80070026
Unable to read selector for PCR for processor 0
GetContextState failed, 0x80070026
Unable to read selector for PCR for processor 0
GetContextState failed, 0x80070026
Unable to read selector for PCR for processor 0
GetContextState failed, 0x80070026
GetContextState failed, 0x80070026
Unable to get current machine context, Win32 error 0n38
GetContextState failed, 0x80070026
Unable to get current machine context, Win32 error 0n38
GetContextState failed, 0x80070026
```

Looking at bugcheck number and parameters we can form some signature and check in our crash database. We can also request a kernel minidump corresponding to debug session time.

DISPATCH LEVEL SPIN

Spiking Thread pattern (Volume 1, page 305) includes normal threads running at PASSIVE_LEVEL or APC_LEVEL IRQL that can be preempted by any other higher priority thread. Therefore, spiking threads are not necessarily ones that were in RUNNING state when the memory dump was saved. They consumed much CPU and this is reflected in their *User* and *Kernel* time values. The pattern also includes threads running at DISPATCH_LEVEL and higher IRQL. These threads cannot be preempted by another thread so they usually remain in RUNNING state all the time unless they lower their IRQL. Some of them can be trying to acquire a spinlock and we need more specialized pattern for them. We would see it when a spinlock for some data structure wasn't released or was corrupt and some thread tries to acquire it and enters endless spinning loop unless interrupted by higher IRQL interrupt. These infinite loops can also happen due to software defects in code running at dispatch level or higher IRQL.

Let's look at one example. The following running thread was interrupted by a keyboard interrupt apparently to save **Manual Dump** (Volume 1, page 479). We see that it spent almost 11 minutes in kernel:

```
0: kd> !thread
THREAD 830c07c0  Cid 0588.0528  Teb: 7ffa3000 Win32Thread: e29546a8
RUNNING on processor 0
Not impersonating
DeviceMap                 e257b7c8
Owning Process            831ec608       Image:          MyApp.EXE
Wait Start TickCount      122850         Ticks: 40796 (0:00:10:37.437)
Context Switch Count      191                 LargeStack
UserTime                  00:00:00.000
KernelTime                00:10:37.406
Win32 Start Address MyApp!ThreadImpersonation (0×35f76821)
Start Address kernel32!BaseThreadStartThunk (0×7c810659)
Stack Init a07bf000 Current a07beca0 Base a07bf000 Limit a07bb000 Call 0
Priority 11 BasePriority 8 PriorityDecrement 2 DecrementCount 16
ChildEBP RetAddr
a07be0f8 f77777fa nt!KeBugCheckEx+0×1b
a07be114 f7777032 i8042prt!I8xProcessCrashDump+0×237
a07be15c 805448e5 i8042prt!I8042KeyboardInterruptService+0×21c
a07be15c 806e4a37 nt!KiInterruptDispatch+0×45 (FPO: [0,2] TrapFrame @
a07be180)
a07be220 a1342755 hal!KeAcquireInStackQueuedSpinLock+0×47
a07be220 a1342755 MyDriver!RcvData+0×98
```

To see the code and context we switch to the trap frame (Volume 1, page 83) and disassemble the interrupted function:

```
1: kd> .trap a07be180
ErrCode = 00000000
eax=a07be200 ebx=a07be228 ecx=831dabf5 edx=a07beb94 esi=831d02a8
edi=831dabd8
eip=806e4a37 esp=a07be1f4 ebp=a07be220 iopl=0 nv up ei pl nz na po nc
cs=0008 ss=0010 ds=0000 es=0000 fs=0000 gs=0000 efl=00000202
hal!KeAcquireInStackQueuedSpinLock+0x47:
806e4a37 ebf3            jmp     hal!KeAcquireInStackQueuedSpinLock+0×3c
(806e4a2c)

1: kd> uf hal!KeAcquireInStackQueuedSpinLock
hal!KeAcquireInStackQueuedSpinLock:
806e49f0 mov     eax,dword ptr ds:[FFFE0080h]
806e49f5 shr     eax,4
806e49f8 mov     al,byte ptr hal!HalpVectorToIRQL (806ef218)[eax]
806e49fe mov     dword ptr ds:[0FFFE0080h],41h
806e4a08 mov     byte ptr [edx+8],al
806e4a0b mov     dword ptr [edx+4],ecx
806e4a0e mov     dword ptr [edx],0
806e4a14 mov     eax,edx
806e4a16 xchg    edx,dword ptr [ecx]
806e4a18 cmp     edx,0
806e4a1b jne     hal!KeAcquireInStackQueuedSpinLock+0x34 (806e4a24)

hal!KeAcquireInStackQueuedSpinLock+0x2d:
806e4a1d or      ecx,2
806e4a20 mov     dword ptr [eax+4],ecx

hal!KeAcquireInStackQueuedSpinLock+0x33:
806e4a23 ret

hal!KeAcquireInStackQueuedSpinLock+0x34:
806e4a24 or      ecx,1
806e4a27 mov     dword ptr [eax+4],ecx
806e4a2a mov     dword ptr [edx],eax

hal!KeAcquireInStackQueuedSpinLock+0x3c:
806e4a2c test    dword ptr [eax+4],1
806e4a33 je      hal!KeAcquireInStackQueuedSpinLock+0×33 (806e4a23)

hal!KeAcquireInStackQueuedSpinLock+0x45:
806e4a35 pause
806e4a37 jmp     hal!KeAcquireInStackQueuedSpinLock+0×3c (806e4a2c)
```

JMP instruction transfers execution to the code that tests the first bit at [EAX+4] address. If it isn’t set it falls through to the same JMP instruction. We know the value of EAX from the trap frame so we can dereference that address:

```
1: kd> dyd eax+4 l1
           3          2          1          0
          10987654 32109876 54321098 76543210
          -------- -------- -------- --------
a07be204  10000011 00011101 10101011 11110101  831dabf5
```

The value is odd: the first leftmost bit is set. Therefore the code will loop indefinitely unless a different thread running on another processor clears that bit. However the second processor is idle:

```
0: kd> ~0s

0: kd> k
ChildEBP RetAddr
f794cd54 00000000 nt!KiIdleLoop+0x14
```

Seems we have a problem. We need to examine MyDriver.sys code to understand how it uses queued spinlocks.

Note: In addition to user-defined there are internal system queued spinlocks we can check by using **!qlocks** WinDbg command.

NO PROCESS DUMPS

The absence of crash dumps when we expect them can be considered as a pattern on its own and. This can happen due to variety of reasons and troubleshooting should be based on the distinction between crashes and hangs (Volume 1, page 36). We have three combinations here:

1. A process is visible in Task Manager and is functioning normally
2. A process is visible in Task Manager and has stopped functioning normally
3. A process is not visible in Task Manager

If a process is visible in the task list and is functioning normally then the following reasons should be considered:

- Exceptions haven't happened yet due to different code execution paths or the time has not come yet and we need to wait more
- Exceptions haven't happened yet due to a different memory layout. This can be the instance of **Changed Environment** pattern (Volume 1, page 283).

If a process is visible in Task Manager and has stopped functioning normally then it might be hanging and waiting for some input. In such cases it is better to get process dumps proactively (Volume 1, page 39). If a process is not visible in Task Manager then the following reasons should be considered:

- Debugger value for AeDebug key is invalid, missing or points to a wrong path or a command line has wrong arguments. For examples see **Custom Postmortem Debuggers on Vista** (Volume 1, page 618) or **NTSD on x64 Windows 2003** (Volume 1, page 633).
- Something is wrong with exception handling mechanism or WER settings. Use Process Monitor to see what processes are launched and modules are loaded when an exception happens. Check WER settings in Control panel.
- Try *LocalDumps* registry key for Vista SP1 and Windows Server 2008 (Volume 1, page 606)
- Use live debugging techniques like attaching to a process or running a process under a debugger to monitor exceptions and saving first chance exception crash dumps (Volume 1, page 465).

This is very important pattern for technical support environments that rely on post-mortem analysis.

NO SYSTEM DUMPS

This is corresponding pattern similar to **No Process Dumps** (page 157) where the system bluescreens either on demand or because of a bugcheck condition but no kernel or complete dumps are saved. In such cases we would advise to check free space on a drive where memory dumps are supposed to be saved. This is because crash dumps are saved to a page file first and then copied to a separate file during boot time, by default to memory.dmp file. Related Microsoft links can be found here (Volume 1, page 628). In case we have enough free space but not enough page file space we might get an instance of **Truncated Dump** (Volume 1, page 340) or **Corrupt Dump** pattern (page 151).

INSUFFICIENT MEMORY (PTE)

In order to maintain virtual to physical address translation OS needs page tables. These tables occupy memory too. If memory is not enough for new tables the system will fail to create processes, allocate I/O buffers and memory from pools. We might see the following diagnostic message from WinDbg:

```
4: kd> !vm

*** Virtual Memory Usage ***
 Physical Memory:      851422 (   3405688 Kb)
 Page File: \??\C:\pagefile.sys
   Current:   2095104 Kb  Free Space:   2081452 Kb
   Minimum:   2095104 Kb  Maximum:      4190208 Kb
 Available Pages:      683464 (   2733856 Kb)
 ResAvail Pages:       800927 (   3203708 Kb)
 Locked IO Pages:         145 (       580 Kb)
 Free System PTEs:      23980 (     95920 Kb)

 ******* 356363 system PTE allocations have failed ******

 Free NP PTEs:           6238 (     24952 Kb)
 Free Special NP:           0 (         0 Kb)
 Modified Pages:          482 (      1928 Kb)
 Modified PF Pages:       482 (      1928 Kb)
 NonPagedPool Usage:    18509 (     74036 Kb)
 NonPagedPool Max:      31970 (    127880 Kb)
 PagedPool 0 Usage:      8091 (     32364 Kb)
 PagedPool 1 Usage:      2495 (      9980 Kb)
 PagedPool 2 Usage:      2580 (     10320 Kb)
 PagedPool 3 Usage:      2552 (     10208 Kb)
 PagedPool 4 Usage:      2584 (     10336 Kb)
 PagedPool Usage:       18302 (     73208 Kb)
 PagedPool Maximum:     39936 (    159744 Kb)

 ********** 48530 pool allocations have failed **********

 Shared Commit:          5422 (     21688 Kb)
 Special Pool:              0 (         0 Kb)
 Shared Process:         5762 (     23048 Kb)
 PagedPool Commit:      18365 (     73460 Kb)
 Driver Commit:          2347 (      9388 Kb)
 Committed pages:      129014 (    516056 Kb)
 Commit limit:        1342979 (   5371916 Kb)
```

We could also see another diagnostic message about pool allocation failures which may be the consequence of PTE allocation failures.

The cause of system PTE allocation failures might be incorrect value of SystemPages registry key that needs to be adjusted as explained in the following TechNet article:

The number of free page table entries is low, which can cause system instability
http://technet.microsoft.com/en-us/library/aa995783.aspx

Another cause would be /3GB boot option on x86 systems especially used for hosting terminal sessions. This case is explained in this blog post which also shows how to detect /3GB kernel and complete memory dumps:

Consequences of running 3GB and PAE together
http://blogs.technet.com/brad_rutkowski/archive/2007/04/03/consequences-of-running-3gb-and-pae-together.aspx

In our case the system was booted with /3GB:

```
4: kd> vertarget
Windows Server 2003 Kernel Version 3790 (Service Pack 2) MP (8 procs) Free
x86 compatible
Product: Server, suite: Enterprise TerminalServer
Built by: 3790.srv03_sp2_gdr.070304-2240
Kernel base = 0xe0800000 PsLoadedModuleList = 0xe08af9c8
Debug session time: Fri Feb  1 09:10:17.703 2008 (GMT+0)
System Uptime: 6 days 17:14:45.528
```

Normal Windows 2003 systems have different kernel base address which can be checked from Reference Stack Traces for Windows Server 2003, Volume 1, Appendix B (Virtual Memory section):

```
kd> vertarget
Windows Server 2003 Kernel Version 3790 (Service Pack 2) UP Free x86
compatible
Product: Server, suite: Enterprise TerminalServer SingleUserTS
Built by: 3790.srv03_sp2_rtm.070216-1710
Kernel base = 0×80800000 PsLoadedModuleList = 0×8089ffa8
Debug session time: Wed Jan 30 17:54:13.390 2008 (GMT+0)
System Uptime: 0 days 0:30:12.000
```

SUSPENDED THREAD

Most of the time threads are not suspended explicitly. If we look at active and waiting threads in kernel and complete memory dumps their SuspendCount member is 0:

```
THREAD 88951bc8  Cid 03a4.0d24  Teb: 7ffaa000 Win32Thread: 00000000 WAIT:
(Unknown) UserMode Non-Alertable
    889d6a78  Semaphore Limit 0x7fffffff
    88951c40  NotificationTimer
Not impersonating
DeviceMap                 e1b80b98
Owning Process            888a9d88       Image:         svchost.exe
Wait Start TickCount      12669          Ticks: 5442 (0:00:01:25.031)
Context Switch Count      3
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address RPCRT4!ThreadStartRoutine (0x77c4b0f5)
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init f482f000 Current f482ec0c Base f482f000 Limit f482c000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
f482ec24 80833465 nt!KiSwapContext+0x26
f482ec50 80829a62 nt!KiSwapThread+0x2e5
f482ec98 809226bd nt!KeWaitForSingleObject+0x346
f482ed48 8088978c nt!NtReplyWaitReceivePortEx+0x521
f482ed48 7c9485ec nt!KiFastCallEntry+0xfc (TrapFrame @ f482ed64)
00efff84 77c58792 ntdll!KiFastSystemCallRet
00efff8c 77c5872d RPCRT4!RecvLotsaCallsWrapper+0xd
00efffac 77c4b110 RPCRT4!BaseCachedThreadRoutine+0x9d
00efffb8 7c824829 RPCRT4!ThreadStartRoutine+0x1b
00efffec 00000000 kernel32!BaseThreadStart+0x34

5: kd> dt _KTHREAD 88951bc8
ntdll!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY [ 0x88951bd8 - 0x88951bd8 ]
   +0x018 InitialStack     : 0xf482f000
   +0x01c StackLimit       : 0xf482c000
   +0x020 KernelStack      : 0xf482ec0c
   +0x024 ThreadLock       : 0
   +0x028 ApcState         : _KAPC_STATE
...
...
...
   +0x14f FreezeCount      : 0 ''
   +0×150 SuspendCount     : 0 ”
```

We won’t find SuspendCount in reference stack traces (Volume 1, Appendix B). Only when some other thread explicitly suspends another thread the latter has non-zero suspend count. Suspended threads are excluded from thread scheduling and therefore can be considered as blocked. This might be the sign of a debugger present, for example, where all threads in a process are suspended when a user debugger is processing a debugger event like a breakpoint or access violation exception. In this case **!process 0 ff** command output shows SuspendCount value:

```
THREAD 888b8668  Cid 0ca8.1448  Teb: 00000000 Win32Thread: 00000000 WAIT:
(Unknown) KernelMode Non-Alertable
SuspendCount 2
    888b87f8  Semaphore Limit 0×2
Not impersonating
DeviceMap                 e10028e8
Owning Process            898285b0       Image:         processA.exe
Wait Start TickCount      13456          Ticks: 4655 (0:00:01:12.734)
Context Switch Count      408
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Start Address driver!DriverThread (0xf6fb8218)
Stack Init f455b000 Current f455a3ac Base f455b000 Limit f4558000 Call 0
Priority 6 BasePriority 6 PriorityDecrement 0
ChildEBP RetAddr
f455a3c4 80833465 nt!KiSwapContext+0×26
f455a3f0 80829a62 nt!KiSwapThread+0×2e5
f455a438 80833178 nt!KeWaitForSingleObject+0×346
f455a450 8082e01f nt!KiSuspendThread+0×18
f455a498 80833480 nt!KiDeliverApc+0×117
f455a4d0 80829a62 nt!KiSwapThread+0×300
f455a518 f6fb7f13 nt!KeWaitForSingleObject+0×346
f455a548 f4edd457 driver!WaitForSingleObject+0×75
f455a55c f4edcdd8 driver!DeviceWaitForRead+0×19
f455ad90 f6fb8265 driver!InputThread+0×17e
f455adac 80949b7c driver!DriverThread+0×4d
f455addc 8088e062 nt!PspSystemThreadStartup+0×2e
00000000 00000000 nt!KiThreadStartup+0×16

5: kd> dt _KTHREAD 888b8668
ntdll!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY [ 0x888b8678 - 0x888b8678 ]
   +0x018 InitialStack     : 0xf455b000
   +0x01c StackLimit       : 0xf4558000
   +0x020 KernelStack      : 0xf455a3ac
   +0x024 ThreadLock       : 0
...
...
...
   +0x14f FreezeCount      : 0 ''
   +0×150 SuspendCount     : 2 ”
```

This pattern should rise suspicion bar and in some cases coupled with **Special Process** pattern (page 164) can lead to immediate problem identification.

SPECIAL PROCESS

Special Stack Trace pattern (Volume 1, page 478) is about stack traces not present in normal crash dumps (Volume 1, Appendix B). **Special Process** is the similar pattern about processes not running during normal operation or highly domain specific processes that are the sign of certain software environment, for example, OS running inside VMWare or VirtualPC (Volume 1, page 219). Here we'll see one example when identifying specific process led to successful problem identification inside a complete memory dump.

Inspection of running processes shows the presence of Dr. Watson:

```
5: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 89d9b648  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid:
0000
    DirBase: 54d5d020  ObjectTable: e1000e20  HandleCount: 1711.
    Image: System

PROCESS 8979b758  SessionId: none  Cid: 01b0    Peb: 7ffdd000  ParentCid:
0004
    DirBase: 54d5d040  ObjectTable: e181d8b0  HandleCount:  29.
    Image: smss.exe

PROCESS 89793cf0  SessionId: 0  Cid: 01e0    Peb: 7ffde000  ParentCid:
01b0
    DirBase: 54d5d060  ObjectTable: e13eea10  HandleCount: 1090.
    Image: csrss.exe

...
...
...

PROCESS 8797a600  SessionId: 1  Cid: 17d0    Peb: 7ffdc000  ParentCid:
1720
    DirBase: 54d5d8c0  ObjectTable: e2870af8  HandleCount: 243.
    Image: explorer.exe

PROCESS 87966d88  SessionId: 2  Cid: 0df0    Peb: 7ffd4000  ParentCid:
01b0
    DirBase: 54d5d860  ObjectTable: e284cd48  HandleCount:  53.
    Image: csrss.exe
```

```
PROCESS 879767c8  SessionId: 0  Cid: 0578    Peb: 7ffde000  ParentCid:
0ca8
    DirBase: 54d5d8a0  ObjectTable: e2c05268  HandleCount: 180.
    Image: drwtsn32.exe
```

Inspecting stack traces shows that drwtsn32.exe is waiting for a debugger event so there must be some debugging target (debuggee):

```
5: kd> .process /r /p 879767c8
Implicit process is now 879767c8
Loading User Symbols

5: kd> !process 879767c8
PROCESS 879767c8  SessionId: 0  Cid: 0578    Peb: 7ffde000  ParentCid:
0ca8
    DirBase: 54d5d8a0  ObjectTable: e2c05268  HandleCount: 180.
    Image: drwtsn32.exe
    VadRoot 88a33cd0 Vads 59 Clone 0 Private 1737. Modified 10792. Locked
0.
    DeviceMap e10028e8
    Token                             e2ee2330
    ElapsedTime                       00:01:12.703
    UserTime                          00:00:00.203
    KernelTime                        00:00:00.031
    QuotaPoolUsage[PagedPool]         52092
    QuotaPoolUsage[NonPagedPool]      2360
    Working Set Sizes (now,min,max)  (2488, 50, 345) (9952KB, 200KB,
1380KB)
    PeakWorkingSetSize                2534
    VirtualSize                       34 Mb
    PeakVirtualSize                   38 Mb
    PageFaultCount                    13685
    MemoryPriority                    BACKGROUND
    BasePriority                      6
    CommitCharge                      1927

THREAD 87976250  Cid 0578.04bc  Teb: 7ffdd000 Win32Thread: bc14a008 WAIT:
(Unknown) UserMode Non-Alertable
    87976558  Thread
Not impersonating
DeviceMap                 e10028e8
Owning Process            879767c8       Image:         drwtsn32.exe
Wait Start TickCount      13460          Ticks: 4651 (0:00:01:12.671)
Context Switch Count      15                 LargeStack
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address drwtsn32!mainCRTStartup (0x01007c1d)
Start Address kernel32!BaseProcessStartThunk (0x7c8217f8)
Stack Init f433b000 Current f433ac60 Base f433b000 Limit f4337000 Call 0
Priority 6 BasePriority 6 PriorityDecrement 0
ChildEBP RetAddr
f433ac78 80833465 nt!KiSwapContext+0x26
```

```
f433aca4 80829a62 nt!KiSwapThread+0x2e5
f433acec 80938d0c nt!KeWaitForSingleObject+0x346
f433ad50 8088978c nt!NtWaitForSingleObject+0x9a
f433ad50 7c9485ec nt!KiFastCallEntry+0xfc
0007fe98 7c821c8d ntdll!KiFastSystemCallRet
0007feac 01005557 kernel32!WaitForSingleObject+0x12
0007ff0c 01003ff8 drwtsn32!NotifyWinMain+0x1ba
0007ff44 01007d4c drwtsn32!main+0x31
0007ffc0 7c82f23b drwtsn32!mainCRTStartup+0x12f
0007fff0 00000000 kernel32!BaseProcessStart+0x23

THREAD 87976558  Cid 0578.0454  Teb: 7ffdc000 Win32Thread: 00000000 WAIT:
(Unknown) UserMode Non-Alertable
    89de2e50  NotificationEvent
    879765d0  NotificationTimer
Not impersonating
DeviceMap                 e10028e8
Owning Process            879767c8       Image:         drwtsn32.exe
Wait Start TickCount      18102          Ticks: 9 (0:00:00:00.140)
Context Switch Count      1163
UserTime                  00:00:00.203
KernelTime                00:00:00.031
Win32 Start Address drwtsn32!DispatchDebugEventThread (0x01003d6d)
Start Address kernel32!BaseThreadStartThunk (0x7c8217ec)
Stack Init f4e26000 Current f4e25be8 Base f4e26000 Limit f4e23000 Call 0
Priority 6 BasePriority 6 PriorityDecrement 0
ChildEBP RetAddr
f4e25c00 80833465 nt!KiSwapContext+0x26
f4e25c2c 80829a62 nt!KiSwapThread+0x2e5
f4e25c74 809a06ab nt!KeWaitForSingleObject+0x346
f4e25d4c 8088978c nt!NtWaitForDebugEvent+0xd5
f4e25d4c 7c9485ec nt!KiFastCallEntry+0xfc
0095ed20 60846f8f ntdll!KiFastSystemCallRet
0095ee6c 60816ecf dbgeng!LiveUserTargetInfo::WaitForEvent+0x1fa
0095ee88 608170d3 dbgeng!WaitForAnyTarget+0x45
0095eecc 60817270 dbgeng!RawWaitForEvent+0x15f
0095eee4 01003f8d dbgeng!DebugClient::WaitForEvent+0×80
0095ffb8 7c824829 drwtsn32!DispatchDebugEventThread+0×220
0095ffec 00000000 kernel32!BaseThreadStart+0×34
```

Knowing that a debugger suspends threads in a debuggee (**Suspended Thread** pattern, page **Error! Bookmark not defined.**) we see the problem process indeed:

```
5: kd> !process 0 2
**** NT ACTIVE PROCESS DUMP ****

...
...
...
```

```
PROCESS 898285b0  SessionId: 0  Cid: 0ca8    Peb: 7ffda000  ParentCid:
022c
    DirBase: 54d5d500  ObjectTable: e2776880  HandleCount:   2.
    Image: svchost.exe

THREAD 888b8668  Cid 0ca8.1448  Teb: 00000000 Win32Thread: 00000000 WAIT:
(Unknown) KernelMode Non-Alertable
SuspendCount 2
  888b87f8  Semaphore Limit 0×2
```

Dumping its thread stacks shows only one system thread where we normally expect plenty of them originated from user space. There is also the presence of a debug port:

```
5: kd> .process /r /p 898285b0
Implicit process is now 898285b0
Loading User Symbols

5: kd> !process 898285b0
PROCESS 898285b0  SessionId: 0  Cid: 0ca8    Peb: 7ffda000  ParentCid:
022c
    DirBase: 54d5d500  ObjectTable: e2776880  HandleCount:   2.
    Image: svchost.exe
    VadRoot 88953220 Vads 209 Clone 0 Private 901. Modified 3. Locked 0.
    DeviceMap e10028e8
    Token                             e27395b8
    ElapsedTime                       00:03:25.640
    UserTime                          00:00:00.156
    KernelTime                        00:00:00.234
    QuotaPoolUsage[PagedPool]         82988
    QuotaPoolUsage[NonPagedPool]      8824
    Working Set Sizes (now,min,max)  (2745, 50, 345) (10980KB, 200KB,
1380KB)
    PeakWorkingSetSize                2819
    VirtualSize                       82 Mb
    PeakVirtualSize                   83 Mb
    PageFaultCount                    4519
    MemoryPriority                    BACKGROUND
    BasePriority                      6
    CommitCharge                      1380
    DebugPort                         89de2e50
```

```
THREAD 888b8668  Cid 0ca8.1448  Teb: 00000000 Win32Thread: 00000000 WAIT:
(Unknown) KernelMode Non-Alertable
SuspendCount 2
    888b87f8  Semaphore Limit 0x2
Not impersonating
DeviceMap                 e10028e8
Owning Process            898285b0       Image:         svchost.exe
Wait Start TickCount      13456          Ticks: 4655 (0:00:01:12.734)
Context Switch Count      408
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Start Address driverA!DriverThread (0xf6fb8218)
Stack Init f455b000 Current f455a3ac Base f455b000 Limit f4558000 Call 0
Priority 6 BasePriority 6 PriorityDecrement 0
ChildEBP RetAddr
f455a3c4 80833465 nt!KiSwapContext+0x26
f455a3f0 80829a62 nt!KiSwapThread+0x2e5
f455a438 80833178 nt!KeWaitForSingleObject+0x346
f455a450 8082e01f nt!KiSuspendThread+0x18
f455a498 80833480 nt!KiDeliverApc+0x117
f455a4d0 80829a62 nt!KiSwapThread+0x300
f455a518 f6fb7f13 nt!KeWaitForSingleObject+0x346
f455a548 f4edd457 driverA!WaitForSingleObject+0x75
f455a55c f4edcdd8 driverB!DeviceWaitForRead+0x19
f455ad90 f6fb8265 driverB!InputThread+0x17e
f455adac 80949b7c driverA!DriverThread+0x4d
f455addc 8088e062 nt!PspSystemThreadStartup+0x2e
00000000 00000000 nt!KiThreadStartup+0x16
```

The most likely scenario was that svchost.exe experienced an unhandled exception that triggered the launch of a postmortem debugger (Volume 1, page 113) such as Dr. Watson.

Other similar examples of this pattern might include the presence of WerFault.exe on Vista, NTSD and other JIT debuggers running.

FRAME POINTER OMISSION

This pattern is the most visible compiler optimization technique and we can notice it in verbose stack traces:

```
0:000> kv
ChildEBP RetAddr
0012ee10 004737a7 application!MemCopy+0x17 (FPO: [3,0,2])
0012ef0c 35878c5b application!ProcessData+0×97 (FPO: [Uses EBP] [3,59,4])
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012ef1c 72a0015b 0×35878c5b
0012ef20 a625e1b0 0×72a0015b
0012ef24 d938bcfe 0xa625e1b0
0012ef28 d4f91bb4 0xd938bcfe
0012ef2c c1c035ce 0xd4f91bb4
...
...
...
```

To recall FPO is a compiler optimization where ESP register is used to address local variables and parameters instead of EBP. EBP may or may not be used for other purposes. When it is used we notice

```
FPO: [Uses EBP]
```

as in the trace above. For description of other FPO number triplets please see Debugging Tools for Windows help section “k, kb, kd, kp, kP, kv (Display Stack Backtrace)”.

Running the analysis command (**!analyze -v**) points to possible stack corruption:

```
PRIMARY_PROBLEM_CLASS:  STACK_CORRUPTION

BUGCHECK_STR:  APPLICATION_FAULT_STACK_CORRUPTION

FAULTING_IP:
application!MemCopy+17
00438637 f3a5 rep movs dword ptr es:[edi],dword ptr [esi]
```

Looking at EBP and ESP shows that they are mismatched:

```
0:000> r
eax=00000100 ebx=00a027f3 ecx=00000040 edx=0012ee58 esi=d938bcfe
edi=0012ee58
eip=00438637 esp=0012ee0c ebp=00a02910 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
application!MemCopy+0×17:
00438637 f3a5 rep movs dword ptr es:[edi],dword ptr [esi]
es:0023:0012ee58=00000010 ds:0023:d938bcfe=????????
```

We might think about **Local Buffer Overflow** pattern (Volume 1, page 460) here but two top stack trace lines are in accordance with each other:

```
0:000> ub 004737a7
application!ProcessData+0x80:
00473790 cmp     eax,edi
00473792 jb      application!ProcessData+0x72 (00473782)
00473794 mov     ecx,dword ptr [esp+104h]
0047379b push    esi
0047379c lea     edx,[esp+38h]
004737a0 push    ecx
004737a1 push    edx
004737a2 call    application!MemCopy (00438620)
```

So perhaps EBP value differs greatly from ESP due to its usage as general purpose register and in fact there was no any stack corruption. Despite using public symbols we have the instance of **Incorrect Stack Trace** pattern (Volume 1, page 288) and we might want to reconstruct it manually. Let's search for EBP value on raw stack below the crash point:

```
0:000> !teb
TEB at 7ffdf000
    ExceptionList:        0012ffb0
    StackBase:            00130000
    StackLimit:           00126000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffdf000
    EnvironmentPointer:   00000000
    ClientId:             0000660c . 00005890
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffd9000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0
```

```
0:000> dds  00126000 00130000
00126000  00000000
00126004  00000000
00126008  00000000
...
...
...
0012eb0c  00a02910
0012eb10  7c90e25e ntdll!NtRaiseException+0xc
0012eb14  7c90eb15 ntdll!KiUserExceptionDispatcher+0x29
0012eb18  0012eb24
0012eb1c  0012eb40
0012eb20  00000000
0012eb24  c0000005
0012eb28  00000000
0012eb2c  00000000
0012eb30  00438637 application!MemCopy+0x17
0012eb34  00000002
0012eb38  00000000
0012eb3c  d938bcfe
...
...
...
0012ebf4  00a02910
0012ebf8  00438637 application!MemCopy+0×17
0012ebfc  0000001b
...
...
...
0012f134  00436323 application!ConstructInfo+0×113
0012f138  00a02910
0012f13c  0000011c
...
...
...
```

Let’s see what functions ConstructInfo calls:

```
0:000> ub 00436323
application!ConstructInfo+0x103
00436313 lea     edx,[esp+10h]
00436317 push    edx
00436318 push    eax
00436319 push    30h
0043631b push    ecx
0043631c push    ebx
0043631d push    ebp
0043631e call    application!EnvelopeData (00438bf0)
```

We notice EBP was pushed prior to calling EnvelopeData function. If we disassemble this function we would see that it calls ProcessData function from our partial stack trace:

```
0:000> uf 00438bf0
application!EnvelopeData:
00438bf0 sub     esp,1F4h
00438bf6 push    ebx
00438bf7 mov     ebx,dword ptr [esp+20Ch]
00438bfe test    ebx,ebx
00438c00 push    ebp
...
...
...
00438c76 rep stos byte ptr es:[edi]
00438c78 lea     eax,[esp+14h]
00438c7c push    eax
00438c7d push    ebp
00438c7e call    application!ProcessData (00473710)
00438c83 pop     edi
00438c84 pop     esi
```

Let’s try the elaborate form of **k** command and supply it with custom ESP and EBP values pointing to

```
0012f134  00436323 application!ConstructInfo+0×113
```

and also EIP of the fault:

```
0:000> k L=0012f134 0012f134 00438637
ChildEBP RetAddr
0012f1cc 00435a65 application!MemCopy+0×17
0012f28c 0043532e application!ClientHandleServerRequest+0×395
0012f344 00434fcd application!Accept+0×23
0012f374 0042e4f3 application!DataArrival+0×17d
0012f43c 0041aea9 application!ProcessInput+0×98
0012ff0c 0045b278 application!AppMain+0xda
0012ff24 0041900e application!WinMain+0×78
0012ffc0 7c816fd7 application!WinMainCRTStartup+0×134
0012fff0 00000000 kernel32!BaseProcessStart+0×23
```

We see that although it misses some initial frames after MemCopy function we aided WinDbg to walk to the bottom of the stack and reconstruct the plausible stack trace for us.

FALSE FUNCTION PARAMETERS

Beginner users of WinDbg sometimes confuse the first 3 parameters (or 4 for x64) displayed by **kb** or **kv** commands with real function parameters:

```
0:000> kbnL
 # ChildEBP RetAddr  Args to Child
00 002df5f4 0041167b 002df97c 00000000 7efdf000 ntdll!DbgBreakPoint
01 002df6d4 004115c9 00000000 40000000 00000001
CallingConventions!A::thiscallFunction+0×2b
02 002df97c 004114f9 00000001 40001000 00000002
CallingConventions!fastcallFunction+0×69
03 002dfbf8 0041142b 00000000 40000000 00000001
CallingConventions!cdeclFunction+0×59
04 002dfe7c 004116e8 00000000 40000000 00000001
CallingConventions!stdcallFunction+0×5b
05 002dff68 00411c76 00000001 005a2820 005a28c8
CallingConventions!wmain+0×38
06 002dffb8 00411abd 002dfff0 7d4e7d2a 00000000
CallingConventions!__tmainCRTStartup+0×1a6
07 002dffc0 7d4e7d2a 00000000 00000000 7efdf000
CallingConventions!wmainCRTStartup+0xd
08 002dfff0 00000000 00411082 00000000 000000c8
kernel32!BaseProcessStart+0×28
```

The calling sequence for it is:

```
stdcallFunction(0, 0x40000000, 1, 0x40001000, 2, 0x40002000) ->
cdeclFunction(0, 0x40000000, 1, 0x40001000, 2, 0x40002000) ->
fastcallFunction(0, 0x40000000, 1, 0x40001000, 2, 0x40002000) ->
A::thiscallFunction(0, 0x40000000, 1, 0x40001000, 2, 0x40002000)
```

and we see that only in the case of **fastcall** calling convention we have discrepancy due to the fact that the first two parameters are passed not via the stack but through ECX and EDX:

```
0:000> ub 004114f9
CallingConventions!cdeclFunction+0x45
004114e5 push    ecx
004114e6 mov     edx,dword ptr [ebp+14h]
004114e9 push    edx
004114ea mov     eax,dword ptr [ebp+10h]
004114ed push    eax
004114ee mov     edx,dword ptr [ebp+0Ch]
004114f1 mov     ecx,dword ptr [ebp+8]
004114f4 call    CallingConventions!ILT+475(?fastcallFunctionYIXHHHHHHZ)
(004111e0)
```

However if we have full symbols we can see all parameters:

```
0:000> .frame 2
02 002df97c 004114f9 CallingConventions!fastcallFunction+0x69

0:000> dv /i /V
prv param  002df974 @ebp-0x08              a = 0
prv param  002df968 @ebp-0x14              b = 1073741824
prv param  002df984 @ebp+0x08              c = 1
prv param  002df988 @ebp+0x0c              d = 1073745920
prv param  002df98c @ebp+0x10              e = 2
prv param  002df990 @ebp+0x14              f = 1073750016
prv local  002df7c7 @ebp-0x1b5           obj = class A
prv local  002df7d0 @ebp-0x1ac         dummy = int [100]
```

How does **dv** command know about values in ECX and EDX which were definitely overwritten by later code? This is because the called function prolog saved them as local variables which you can notice as negative offsets for EBP register in **dv** output above:

```
0:000> uf CallingConventions!fastcallFunction
CallingConventions!fastcallFunction
   32 00411560 push    ebp
   32 00411561 mov     ebp,esp
   32 00411563 sub     esp,27Ch
   32 00411569 push    ebx
   32 0041156a push    esi
   32 0041156b push    edi
   32 0041156c push    ecx
   32 0041156d lea     edi,[ebp-27Ch]
   32 00411573 mov     ecx,9Fh
   32 00411578 mov     eax,0CCCCCCCCh
   32 0041157d rep stos dword ptr es:[edi]
   32 0041157f pop     ecx
   32 00411580 mov     dword ptr [ebp-14h],edx
   32 00411583 mov     dword ptr [ebp-8],ecx
...
...
...
```

In order to spot the occurrences of this pattern double checks and knowledge of calling conventions are required. Sometimes this pattern is a consequence of **Optimized Code** pattern (Volume 1, page 265).

x64 stack traces don't show any discrepancies except the fact that **thiscall** function parameters are shifted to the right:

```
0:000> kbL
RetAddr           : Args to Child                                   : Call Site
00000001`40001397 : cccccccc`cccccccc cccccccc`cccccccc cccccccc`cccccccc
cccccccc`cccccccc : ntdll!DbgBreakPoint
00000001`40001233 : 00000000`0012fa94 cccccccc`00000000 cccccccc`40000000
cccccccc`00000001 : CallingConventions!A::thiscallFunction+0×37
00000001`40001177 : cccccccc`00000000 cccccccc`40000000 cccccccc`00000001
cccccccc`40001000 : CallingConventions!fastcallFunction+0×93
00000001`400010c7 : cccccccc`00000000 cccccccc`40000000 cccccccc`00000001
cccccccc`40001000 : CallingConventions!cdeclFunction+0×87
00000001`400012ae : cccccccc`00000000 cccccccc`40000000 cccccccc`00000001
cccccccc`40001000 : CallingConventions!stdcallFunction+0×87
00000001`400018ec : 00000001`00000001 00000000`00481a80 00000000`00000000
00000001`400026ee : CallingConventions!wmain+0×4e
00000001`4000173e : 00000000`00000000 00000000`00000000 00000000`00000000
00000000`00000000 : CallingConventions!__tmainCRTStartup+0×19c
00000000`77d5964c : 00000000`77d59620 00000000`00000000 00000000`00000000
00000000`0012ffa8 : CallingConventions!wmainCRTStartup+0xe
00000000`00000000 : 00000001`40001730 00000000`00000000 00000000`00000000
00000000`00000000 : kernel32!BaseProcessStart+0×29
```

How this can happen if the standard x64 calling convention passes the first 4 parameters via ECX, EDX, R8 and R9? This is because the called function prolog saved them on stack (this might not be true in the case of optimized code):

```
0:000> uf CallingConventions!fastcallFunction
CallingConventions!fastcallFunction
   32 00000001`400011a0 44894c2420      mov     dword ptr [rsp+20h],r9d
   32 00000001`400011a5 4489442418      mov     dword ptr [rsp+18h],r8d
   32 00000001`400011aa 89542410        mov     dword ptr [rsp+10h],edx
   32 00000001`400011ae 894c2408        mov     dword ptr [rsp+8],ecx
...
...
...
```

A::thiscallFunction function passes **this** pointer via ECX too and this explains the right shift of parameters.

Here is the C++ code I used for experimentation:

```
#include "stdafx.h"
#include <windows.h>

void __stdcall stdcallFunction (int, int, int, int, int, int);
void __cdecl cdeclFunction (int, int, int, int, int, int);
void __fastcall fastcallFunction (int, int, int, int, int, int);

class A
{
public:
 void thiscallFunction (int, int, int, int, int, int) { DebugBreak(); };
};

void __stdcall stdcallFunction (int a, int b, int c, int d, int e, int f)
{
 int dummy[100] = {0};

 cdeclFunction (a, b, c, d, e, f);
}

void __cdecl cdeclFunction (int a, int b, int c, int d, int e, int f)
{
 int dummy[100] = {0};

 fastcallFunction (a, b, c, d, e, f);
}

void __fastcall fastcallFunction (int a, int b, int c, int d, int e, int
f)
{
 int dummy[100] = {0};

 A obj;

 obj.thiscallFunction (a, b, c, d, e, f);
}

int _tmain(int argc, _TCHAR* argv[])
{
 stdcallFunction (0, 0x40000000, 1, 0x40001000, 2, 0x40002000);

 return 0;
}
```

MESSAGE BOX

Another suspicious threads in crash dumps are GUI threads executing message box code. Usually message boxes are displayed to show some error and we can see it by dumping the second and the third MessageBox parameters:

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType);
```

Sometimes message boxes block processes as shown in the example illustrating **Coupled Processes** pattern (Volume 1, page 419). Other threads might point to possibly "hang" sessions and processes in memory dumps coming from terminal services environments. This is another example of the collective **Blocked Thread** pattern (page 184).

Let's look at one example where message box pointed to the right troubleshooting direction. A user process was reported hanging from time to time however it was not specified which one. Searching for MessageBox in the log of all threads in the system produced by **!process 0 ff** WinDbg command revealed the following thread:

```
THREAD 88b14da8  Cid 0a04.0c14  Teb: 7ffdd000 Win32Thread: e5e50ab0 WAIT:
(WrUserRequest) UserMode Non-Alertable
    87b74358  SynchronizationEvent
IRP List:
    87a0ba00: (0006,0244) Flags: 00000000  Mdl: 00000000
Not impersonating
DeviceMap                 e14bec28
Owning Process            888ffb60       Image:         OUTLOOK.EXE
Wait Start TickCount      1275435        Ticks: 210 (0:00:00:03.281)
Context Switch Count      1050203               LargeStack
UserTime                  00:00:16.812
KernelTime                00:00:18.000
Win32 Start Address OUTLOOK (0x30001084)
Start Address kernel32!BaseProcessStartThunk (0x7c810665)
Stack Init a0c98000 Current a0c97cb0 Base a0c98000 Limit a0c90000 Call 0
Priority 11 BasePriority 8 PriorityDecrement 2 DecrementCount 16
ChildEBP RetAddr
a0c97cc8 804e1bd2 nt!KiSwapContext+0x2f
a0c97cd4 804e1c1e nt!KiSwapThread+0x8a
a0c97cfc bf802f70 nt!KeWaitForSingleObject+0x1c2
a0c97d38 bf803776 win32k!xxxSleepThread+0x192
a0c97d4c bf803793 win32k!xxxRealWaitMessageEx+0x12
a0c97d5c 804dd99f win32k!NtUserWaitMessage+0x14
a0c97d5c 7c90eb94 nt!KiFastCallEntry+0xfc
0013f3a8 7e419418 ntdll!KiFastSystemCallRet
0013f3e0 7e42593f USER32!NtUserWaitMessage+0xc
0013f408 7e43a91e USER32!InternalDialogBox+0xd0
0013f6c8 7e43a284 USER32!SoftModalMessageBox+0x938
0013f818 7e4661d3 USER32!MessageBoxWorker+0x2ba
0013f870 7e466278 USER32!MessageBoxTimeoutW+0x7a
0013f8a4 7e450617 USER32!MessageBoxTimeoutA+0x9c
0013f8c4 7e4505cf USER32!MessageBoxExA+0x1b
0013f8e0 088098a9 USER32!MessageBoxA+0x45
...
...
...

0: kd> .process /r /p 888ffb60
Implicit process is now 888ffb60
Loading User Symbols

0: kd> !thread 88b14da8
...
...
...
ChildEBP RetAddr  Args to Child
...
...
...
0013f8e0 088098a9 00000000 0013f944 088708f0 USER32!MessageBoxA+0×45
...
...
...
```

```
0: kd> dA 0013f944
0013f944  "Cannot contact database, Retry"
```

This immediately raised suspicion and looking at other threads in the same application revealed that many of them were trying to open a network connection, for example:

```
THREAD 87a70da8  Cid 0a04.0cc0  Teb: 7ff83000 Win32Thread: 00000000 WAIT:
(UserRequest) UserMode Non-Alertable
    87d690b0  NotificationEvent
    87a70e98  NotificationTimer
IRP List:
    87af7bc8: (0006,0244) Flags: 00000000  Mdl: 00000000
Not impersonating
DeviceMap                 e14bec28
Owning Process            888ffb60       Image:         OUTLOOK.EXE
Wait Start TickCount      1267130        Ticks: 8515 (0:00:02:13.046)
Context Switch Count      18
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address msmapi32!FOpenThreadImpersonationToken (0×35f76963)
Start Address kernel32!BaseThreadStartThunk (0×7c810659)
Stack Init 9fbc5000 Current 9fbc4ca0 Base 9fbc5000 Limit 9fbc2000 Call 0
Priority 8 BasePriority 8 PriorityDecrement 0 DecrementCount 16
Kernel stack not resident.
ChildEBP RetAddr
9fbc4cb8 804e1bd2 nt!KiSwapContext+0×2f
9fbc4cc4 804e1c1e nt!KiSwapThread+0×8a
9fbc4cec 8056d2f9 nt!KeWaitForSingleObject+0×1c2
9fbc4d50 804dd99f nt!NtWaitForSingleObject+0×9a
9fbc4d50 7c90eb94 nt!KiFastCallEntry+0xfc
1bd8f52c 7c90e9c0 ntdll!KiFastSystemCallRet
1bd8f530 7c8025cb ntdll!ZwWaitForSingleObject+0xc
1bd8f594 7c802532 kernel32!WaitForSingleObjectEx+0xa8
1bd8f5a8 77eec4c6 kernel32!WaitForSingleObject+0×12
1bd8f6a4 77eec8b7 RPCRT4!WS_Open+0×31d
1bd8f7c8 77eec96d RPCRT4!TCPOrHTTP_Open+0×19e
1bd8f800 77e83e8d RPCRT4!TCP_Open+0×55
1bd8f84c 77e843f7 RPCRT4!OSF_CCONNECTION::TransOpen+0×5e
1bd8f8b4 77e84581 RPCRT4!OSF_CCONNECTION::OpenConnectionAndBind+0xbc
1bd8f8f8 77e844d0 RPCRT4!OSF_CCALL::BindToServer+0×104
1bd8f95c 77e7f99c RPCRT4!OSF_BINDING_HANDLE::AllocateCCall+0×2b0
1bd8f98c 77e791c1 RPCRT4!OSF_BINDING_HANDLE::NegotiateTransferSyntax+0×28
1bd8f9a4 77e791f8 RPCRT4!I_RpcGetBufferWithObject+0×5b
1bd8f9b4 77e79825 RPCRT4!I_RpcGetBuffer+0xf
1bd8f9c4 77ef460b RPCRT4!NdrGetBuffer+0×28
1bd8fda4 35bae645 RPCRT4!NdrClientCall2+0×195
...
...
...
```

Looking at IRP showed the possible problem with network at TDI level:

```
0: kd> !irp 87af7bc8
Irp is active with 4 stacks 2 is current (= 0x87af7c5c)  No Mdl: No System
Buffer: Thread 87a70da8:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 00000000 00000000 00000000-00000000
   Args: 00000000 00000000 00000000 00000000
>[  f, 3]   0 e1 897b0310 87bb24c8 a1ad7080-87b6f1f0 Success Error Cancel
pending
        \Driver\Tcpip    MYTDI!WaitForNetwork
   Args: 00000000 87a33988 87a33af8 a1a57600
 [  f, 3]   0 e1 88c13020 87bb24c8 a1a6bea5-87a33988 Success Error Cancel
pending
        \Driver\SYMTDI afd!AfdRestartSuperConnect
   Args: 00000000 87a33988 87a33af8 a1a57600
 [  e,31]   5  0 88bedf18 87d3af90 00000000-00000000
        \Driver\AFD
   Args: 00000000 00000016 000120c7 1bd8f52c
```

SELF-DUMP

Sometimes processes dump themselves using Microsoft DbgHelp API when they encounter internal errors or for other debugging purposes. I separate this from unhandled exceptions which usually cause an external postmortem debugger process to dump memory contents as explained here:

- **Who Calls the Postmortem Debugger?** (Volume 1, page 113)
- **Inside Vista Error Reporting** (Volume 1, page 117)

It is important to understand that a process can dump itself for any reasons that came to mind of an application designer and not only as a reaction to hardware and software exceptions. In any case it is useful to look at raw stack dump of all threads (Volume 1, page 231) and search for first chance exceptions like c0000005 and custom exception handlers (Volume 1, page 470).

We can consider this pattern as a specialized version of **Special Stack Trace** pattern (Volume 1, page 478). The typical thread stack might look like and might be **Incorrect Stack Trace** (Volume 1, page 288) that requires manual reconstruction (Volume 1, page 157):

```
0:012> kL
ChildEBP RetAddr
0151f0c4 77e61f0c ntdll!KiFastSystemCallRet
0151f0d4 08000000 kernel32!CreateFileMappingW+0xc8
WARNING: Frame IP not in any known module. Following frames may be wrong.
0151f0f0 7c82728b 0x8000000
0151f0f4 77e63e41 ntdll!NtMapViewOfSection+0xc
0151f12c 77e6440c kernel32!MapViewOfFileEx+0x71
0151f14c 7c826d2b kernel32!MapViewOfFile+0x1b
0151f1d4 028ca67c ntdll!ZwClose+0xc
0151f2a4 028cc2f1 dbghelp!GenGetClrMemory+0xec
0151f2b4 028c8e55 dbghelp!Win32LiveSystemProvider::CloseMapping+0×11
0151f414 00000000 dbghelp!GenAllocateModuleObject+0×3c5
```

Raw stack data should reveal DbgHelp API calls:

```
0151f944  00000000
0151f948  00000000
0151f94c  0151f9c0
0151f950  028c7662 dbghelp!MiniDumpWriteDump+0×1b2
0151f954  ffffffff
0151f958  00000cb0
0151f95c  00c21ea8
0151f960  00c21f88
0151f964  00c21e90
0151f968  00c21fa0
0151f96c  00000002
0151f970  00000000
0151f974  00000000
0151f978  00000000
0151f97c  7c829f60 ntdll!CheckHeapFillPattern+0×64
0151f980  ffffffff
0151f984  7c829f59 ntdll!RtlFreeHeap+0×70f
0151f988  7c34218a msvcr71!free+0xc3
0151f98c  00000000
0151f990  00000000
0151f994  00c21e90
0151f998  00c21fa0
0151f99c  00c21ea8
0151f9a0  00c21f88
0151f9a4  00000002
0151f9a8  021a00da
0151f9ac  001875d8
0151f9b0  7c3416db msvcr71!_nh_malloc+0×10
0151f9b4  0151f998
0151f9b8  001875d8
0151f9bc  00000000
0151f9c0  0151fbe4
0151f9c4  57b77d01 application!write_problem_report+0×18d1
0151f9c8  ffffffff
0151f9cc  00000cb0
0151f9d0  00000718
0151f9d4  00000002
0151f9d8  00000000
0151f9dc  00000000
0151f9e0  00000000
0151f9e4  00029722
0151f9e8  0151fc20
```

Partially reconstructed stack trace might look like this:

```
0:012> k L=0151f94c
ChildEBP RetAddr
0151f0c4 77e61f0c ntdll!KiFastSystemCallRet
0151f94c 028c7662 kernel32!CreateFileMappingW+0xc8
0151f9c0 57b77d01 dbghelp!MiniDumpWriteDump+0×1b2
0151fbe4 57b77056 application!write_problem_report+0×18d1
0151fc30 579c83af application!TerminateThread+0×18
```

BLOCKED THREAD

We often say that particular thread is blocked and/or it blocks other threads. At the same time we know that almost all threads are “blocked” to some degree except those currently running on processors. They are either preempted and in the ready lists, voluntary yielded their execution or they are waiting for some synchronization object. Therefore the notion of **Blocked Thread** is highly context and problem dependent and usually we notice them when comparing current thread stack traces with their expected normal stack traces. Here reference guides (Volume 1, Appendix B) are indispensible especially those created for troubleshooting concrete products.

To show the diversity of “blocked” threads we can propose the following thread classification:

Running threads

Their EIP (RIP) points to some function different from KiSwapContext:

```
3: kd> !running

System Processors f (affinity mask)
  Idle Processors 0

     Prcb      Current   Next
  0  ffdff120  a30a9350            ................
  1  f7727120  a3186448            ................
  2  f772f120  a59a1b40            ................
  3  f7737120  a3085888            ................
```

```
3: kd> !thread a59a1b40
THREAD a59a1b40  Cid 0004.00b8  Teb: 00000000 Win32Thread: 00000000
RUNNING on processor 2
Not impersonating
DeviceMap                 e10028b0
Owning Process            a59aa648       Image:         System
Wait Start TickCount      1450446        Ticks: 1 (0:00:00:00.015)
Context Switch Count      308765
UserTime                  00:00:00.000
KernelTime                00:00:01.250
Start Address nt!ExpWorkerThread (0×80880356)
Stack Init f7055000 Current f7054cec Base f7055000 Limit f7052000 Call 0
Priority 12 BasePriority 12 PriorityDecrement 0
ChildEBP RetAddr
f7054bc4 8093c55c nt!ObfReferenceObject+0×1c
f7054ca0 8093d2ae nt!ObpQueryNameString+0×2ba
f7054cbc 808f7d0f nt!ObQueryNameString+0×18
f7054d80 80880441 nt!IopErrorLogThread+0×197
f7054dac 80949b7c nt!ExpWorkerThread+0xeb
f7054ddc 8088e062 nt!PspSystemThreadStartup+0×2e
00000000 00000000 nt!KiThreadStartup+0×16

3: kd> .thread a59a1b40
Implicit thread is now a59a1b40

3: kd> r
Last set context:
eax=00000028 ebx=e1000228 ecx=e1002b30 edx=e1000234 esi=e1002b18
edi=0000001a
eip=8086c73e esp=f7054bc4 ebp=f7054ca0 iopl=0 nv up ei pl nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00000202
nt!ObfReferenceObject+0×1c:
8086c73e 40              inc     eax
```

These threads can also be identified by RUNNING attribute in the output of **!process 0 ff** command applied for complete and kernel memory dumps.

Ready threads

These threads can be seen in the output of **!ready** command or identified by READY attribute in the output of **!process 0 ff** command:

```
3: kd> !ready
Processor 0: No threads in READY state
Processor 1: Ready Threads at priority 11
    THREAD a3790790  Cid 0234.1108  Teb: 7ffab000 Win32Thread: 00000000
READY
    THREAD a32799a8  Cid 0234.061c  Teb: 7ff83000 Win32Thread: 00000000
READY
    THREAD a3961798  Cid 0c04.0c68  Teb: 7ffab000 Win32Thread: bc204ea8
READY
Processor 1: Ready Threads at priority 10
    THREAD a32bedb0  Cid 1fc8.1a30  Teb: 7ffad000 Win32Thread: bc804468
READY
Processor 1: Ready Threads at priority 9
    THREAD a52dcd48  Cid 0004.04d4  Teb: 00000000 Win32Thread: 00000000
READY
Processor 2: Ready Threads at priority 11
    THREAD a37fedb0  Cid 0c04.11f8  Teb: 7ff8e000 Win32Thread: 00000000
READY
Processor 3: Ready Threads at priority 11
    THREAD a5683db0  Cid 0234.0274  Teb: 7ffd6000 Win32Thread: 00000000
READY
    THREAD a3151b48  Cid 0234.2088  Teb: 7ff88000 Win32Thread: 00000000
READY
    THREAD a5099d80  Cid 0ecc.0d60  Teb: 7ffd4000 Win32Thread: 00000000
READY
    THREAD a3039498  Cid 0c04.275c  Teb: 7ff7d000 Win32Thread: 00000000
READY
```

If we look at these threads we would see that they were either scheduled to run because of a signaled object they were waiting for:

```
3: kd> !thread a3039498
THREAD a3039498  Cid 0c04.275c  Teb: 7ff7d000 Win32Thread: 00000000 READY
IRP List:
    a2feb008: (0006,0094) Flags: 00000870  Mdl: 00000000
Not impersonating
DeviceMap                 e10028b0
Owning Process            a399a770       Image:         svchost.exe
Wait Start TickCount      1450447        Ticks: 0
Context Switch Count      1069
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address 0x001e4f22
LPC Server thread working on message Id 1e4f22
Start Address KERNEL32!BaseThreadStartThunk (0x77e617ec)
Stack Init f171b000 Current f171ac60 Base f171b000 Limit f1718000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr  Args to Child
f171ac78 80833465 a3039498 a3039540 e7561930 nt!KiSwapContext+0x26
f171aca4 80829a62 00000000 00000000 00000000 nt!KiSwapThread+0x2e5
f171acec 80938d0c a301cad8 00000006 f171ad01
nt!KeWaitForSingleObject+0×346
f171ad50 8088978c 00000c99 00000000 00000000 nt!NtWaitForSingleObject+0×9a
f171ad50 7c8285ec 00000c99 00000000 00000000 nt!KiFastCallEntry+0xfc
03d9efa8 00000000 00000000 00000000 00000000 ntdll!KiFastSystemCallRet

3: kd> !object a301cad8
Object: a301cad8  Type: (a59a0720) Event
    ObjectHeader: a301cac0 (old version)
    HandleCount: 1  PointerCount: 3
```

or they were boosted in priority:

```
3: kd> !thread a3790790
THREAD a3790790  Cid 0234.1108  Teb: 7ffab000 Win32Thread: 00000000 READY
IRP List:
    a2f8b7f8: (0006,0094) Flags: 00000900  Mdl: 00000000
Not impersonating
DeviceMap                 e10028b0
Owning Process            a554bcc8       Image:         lsass.exe
Wait Start TickCount      1450447        Ticks: 0
Context Switch Count      384
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address RPCRT4!ThreadStartRoutine (0x77c7b0f5)
Start Address KERNEL32!BaseThreadStartThunk (0x77e617ec)
Stack Init f3ac1000 Current f3ac0ce8 Base f3ac1000 Limit f3abe000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr  Args to Child
f3ac0d00 80831266 a3790790 a50f1870 a3186448 nt!KiSwapContext+0x26
f3ac0d20 8082833a 00000000 a50f1870 8098b56c nt!KiExitDispatcher+0xf8
f3ac0d3c 8098b5b9 a50f1870 00000000 00f5f8d0
nt!KeSetEventBoostPriority+0×156
f3ac0d58 8088978c a50f1870 00f5f8d4 7c8285ec
nt!NtSetEventBoostPriority+0×4d
f3ac0d58 7c8285ec a50f1870 00f5f8d4 7c8285ec nt!KiFastCallEntry+0xfc
00f5f8d4 00000000 00000000 00000000 00000000 ntdll!KiFastSystemCallRet

3: kd> !object a50f1870
Object: a50f1870  Type: (a59a0720) Event
    ObjectHeader: a50f1858 (old version)
    HandleCount: 1  PointerCount: 15
```

or were interrupted and queued to be run again:

```
3: kd> !thread a5683db0
THREAD a5683db0 Cid 0234.0274 Teb: 7ffd6000 Win32Thread: 00000000 READY
IRP List:
  a324d498: (0006,0094) Flags: 00000900 Mdl: 00000000
  a2f97a20: (0006,0094) Flags: 00000900 Mdl: 00000000
  a50c3e70: (0006,0190) Flags: 00000000 Mdl: a50a22d0
  a5167750: (0006,0094) Flags: 00000800 Mdl: 00000000
Not impersonating
DeviceMap e10028b0
Owning Process a554bcc8 Image: lsass.exe
Wait Start TickCount 1450447 Ticks: 0
Context Switch Count 9619
UserTime 00:00:00.156
KernelTime 00:00:00.234
Win32 Start Address RPCRT4!ThreadStartRoutine (0x77c7b0f5)
Start Address KERNEL32!BaseThreadStartThunk (0x77e617ec)
Stack Init f59f3000 Current f59f2d00 Base f59f3000 Limit f59f0000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
f59f2d18 80a5c1ae nt!KiDispatchInterrupt+0xb1
f59f2d48 80a5c577 hal!HalpDispatchSoftwareInterrupt+0x5e
f59f2d54 80a59902 hal!HalEndSystemInterrupt+0x67
f59f2d54 77c6928d hal!HalpIpiHandler+0xd2 (TrapFrame @ f59f2d64)
00c5f908 00000000 RPCRT4!OSF_SCALL::GetBuffer+0×37

3: kd> .thread a5683db0
Implicit thread is now a5683db0

3: kd> r
Last set context:
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000
edi=00000000
eip=8088dba1 esp=f59f2d0c ebp=f59f2d2c iopl=0 nv up di pl nz na po nc
cs=0008 ss=0010 ds=0000 es=0000 fs=0000 gs=0000 efl=00000000
nt!KiDispatchInterrupt+0xb1:
8088dba1 b902000000 mov ecx,2

3: kd> ub
nt!KiDispatchInterrupt+0x8f:
8088db7f mov dword ptr [ebx+124h],esi
8088db85 mov byte ptr [esi+4Ch],2
8088db89 mov byte ptr [edi+5Ah],1Fh
8088db8d mov ecx,edi
8088db8f lea edx,[ebx+120h]
8088db95 call nt!KiQueueReadyThread (80833490)
8088db9a mov cl,1
8088db9c call nt!SwapContext (8088dbd0)
```

```
3: kd> u
nt!KiDispatchInterrupt+0xb1:
8088dba1 mov ecx,2
8088dba6 call dword ptr [nt!_imp_KfLowerIrql (80801108)]
8088dbac mov ebp,dword ptr [esp]
8088dbaf mov edi,dword ptr [esp+4]
8088dbb3 mov esi,dword ptr [esp+8]
8088dbb7 add esp,0Ch
8088dbba pop ebx
8088dbbb ret
```

We can get user space thread stack by using **.trap** WinDbg command but we need to switch to corresponding process context first:

```
3: kd> .process /r /p a554bcc8
Implicit process is now a554bcc8
Loading User Symbols

3: kd> kL
  *** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr
00c5f908 77c7ed60 RPCRT4!OSF_SCALL::GetBuffer+0x37
00c5f924 77c7ed14 RPCRT4!I_RpcGetBufferWithObject+0x7f
00c5f934 77c7f464 RPCRT4!I_RpcGetBuffer+0xf
00c5f944 77ce3470 RPCRT4!NdrGetBuffer+0x2e
00c5fd44 77ce35c4 RPCRT4!NdrStubCall2+0x35c
00c5fd60 77c7ff7a RPCRT4!NdrServerCall2+0x19
00c5fd94 77c8042d RPCRT4!DispatchToStubInCNoAvrf+0x38
00c5fde8 77c80353 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x11f
00c5fe0c 77c68e0d RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
00c5fe40 77c68cb3 RPCRT4!OSF_SCALL::DispatchHelper+0x149
00c5fe54 77c68c2b RPCRT4!OSF_SCALL::DispatchRPCCall+0x10d
00c5fe84 77c68b5e RPCRT4!OSF_SCALL::ProcessReceivedPDU+0x57f
00c5fea4 77c6e8db RPCRT4!OSF_SCALL::BeginRpcCall+0x194
00c5ff04 77c6e7b4 RPCRT4!OSF_SCONNECTION::ProcessReceiveComplete+0x435
00c5ff18 77c7b799 RPCRT4!ProcessConnectionServerReceivedEvent+0x21
00c5ff84 77c7b9b5 RPCRT4!LOADABLE_TRANSPORT::ProcessIOEvents+0x1b8
00c5ff8c 77c8872d RPCRT4!ProcessIOEventsWrapper+0xd
00c5ffac 77c7b110 RPCRT4!BaseCachedThreadRoutine+0x9d
00c5ffb8 77e64829 RPCRT4!ThreadStartRoutine+0x1b
00c5ffec 00000000 kernel32!BaseThreadStart+0x34
```

Waiting threads (wait originated from user space)

```
THREAD a34369d0  Cid 1fc8.1e88  Teb: 7ffae000 Win32Thread: bc6d5818 WAIT:
(Unknown) UserMode Non-Alertable
    a34d9940  SynchronizationEvent
    a3436a48  NotificationTimer
Not impersonating
DeviceMap                 e12256a0
Owning Process            a3340a10       Image:         IEXPLORE.EXE
Wait Start TickCount      1450409        Ticks: 38 (0:00:00:00.593)
Context Switch Count      7091                 LargeStack
UserTime                  00:00:01.015
KernelTime                00:00:02.250
Win32 Start Address mshtml!CExecFT::StaticThreadProc (0x7fab1061)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f252b000 Current f252ac60 Base f252b000 Limit f2528000 Call 0
Priority 11 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
f252ac78 80833465 nt!KiSwapContext+0x26
f252aca4 80829a62 nt!KiSwapThread+0x2e5
f252acec 80938d0c nt!KeWaitForSingleObject+0x346
f252ad50 8088978c nt!NtWaitForSingleObject+0x9a
f252ad50 7c8285ec nt!KiFastCallEntry+0xfc (TrapFrame @ f252ad64)
030dff08 7c827d0b ntdll!KiFastSystemCallRet
030dff0c 77e61d1e ntdll!NtWaitForSingleObject+0xc
030dff7c 77e61c8d kernel32!WaitForSingleObjectEx+0xac
030dff90 7fab08a3 kernel32!WaitForSingleObject+0×12
030dffa8 7fab109c mshtml!CDwnTaskExec::ThreadExec+0xae
030dffb0 7fab106e mshtml!CExecFT::ThreadProc+0×28
030dffb8 77e64829 mshtml!CExecFT::StaticThreadProc+0xd
030dffec 00000000 kernel32!BaseThreadStart+0×34
```

If we had taken user dump of iexplore.exe we would have seen the following stack trace there:

```
030dff08 7c827d0b ntdll!KiFastSystemCallRet
030dff0c 77e61d1e ntdll!NtWaitForSingleObject+0xc
030dff7c 77e61c8d kernel32!WaitForSingleObjectEx+0xac
030dff90 7fab08a3 kernel32!WaitForSingleObject+0×12
030dffa8 7fab109c mshtml!CDwnTaskExec::ThreadExec+0xae
030dffb0 7fab106e mshtml!CExecFT::ThreadProc+0×28
030dffb8 77e64829 mshtml!CExecFT::StaticThreadProc+0xd
030dffec 00000000 kernel32!BaseThreadStart+0×34
```

Another example:

```
THREAD a31f2438  Cid 1fc8.181c  Teb: 7ffaa000 Win32Thread: 00000000 WAIT:
(Unknown) UserMode Non-Alertable
    a30f8c20  NotificationEvent
    a5146720  NotificationEvent
    a376fbb0  NotificationEvent
Not impersonating
DeviceMap                 e12256a0
Owning Process            a3340a10       Image:         IEXPLORE.EXE
Wait Start TickCount      1419690        Ticks: 30757 (0:00:08:00.578)
Context Switch Count      2
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address USERENV!NotificationThread (0x76929dd9)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f5538000 Current f5537900 Base f5538000 Limit f5535000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr
f5537918 80833465 nt!KiSwapContext+0x26
f5537944 80829499 nt!KiSwapThread+0x2e5
f5537978 80938f68 nt!KeWaitForMultipleObjects+0x3d7
f5537bf4 809390ca nt!ObpWaitForMultipleObjects+0x202
f5537d48 8088978c nt!NtWaitForMultipleObjects+0xc8
f5537d48 7c8285ec nt!KiFastCallEntry+0xfc (TrapFrame @ f5537d64)
0851fec0 7c827cfb ntdll!KiFastSystemCallRet
0851fec4 77e6202c ntdll!NtWaitForMultipleObjects+0xc
0851ff6c 77e62fbe kernel32!WaitForMultipleObjectsEx+0x11a
0851ff88 76929e35 kernel32!WaitForMultipleObjects+0×18
0851ffb8 77e64829 USERENV!NotificationThread+0×5f
0851ffec 00000000 kernel32!BaseThreadStart+0×34
```

Waiting threads (wait originated from kernel space)

Examples include explicit wait as a result from calling potentially blocking API:

```
THREAD a33a9740  Cid 1980.1960  Teb: 7ffde000 Win32Thread: bc283ea8 WAIT:
(Unknown) UserMode Non-Alertable
    a35e3168  SynchronizationEvent
Not impersonating
DeviceMap                 e689f298
Owning Process            a342d3a0       Image:         explorer.exe
Wait Start TickCount      1369801        Ticks: 80646 (0:00:21:00.093)
Context Switch Count      1667                 LargeStack
UserTime                  00:00:00.015
KernelTime                00:00:00.093
Win32 Start Address Explorer!ModuleEntry (0x010148a4)
Start Address kernel32!BaseProcessStartThunk (0x77e617f8)
Stack Init f258b000 Current f258ac50 Base f258b000 Limit f2585000 Call 0
Priority 13 BasePriority 10 PriorityDecrement 1
Kernel stack not resident.
ChildEBP RetAddr
f258ac68 80833465 nt!KiSwapContext+0x26
f258ac94 80829a62 nt!KiSwapThread+0x2e5
f258acdc bf89abd3 nt!KeWaitForSingleObject+0×346
f258ad38 bf89da43 win32k!xxxSleepThread+0×1be
f258ad4c bf89e401 win32k!xxxRealWaitMessageEx+0×12
f258ad5c 8088978c win32k!NtUserWaitMessage+0×14
f258ad5c 7c8285ec nt!KiFastCallEntry+0xfc (TrapFrame @ f258ad64)
0007feec 7739bf53 ntdll!KiFastSystemCallRet
0007ff08 7c8fadbd USER32!NtUserWaitMessage+0xc
0007ff14 0100fff1 SHELL32!SHDesktopMessageLoop+0×24
0007ff5c 0101490c Explorer!ExplorerWinMain+0×2c4
0007ffc0 77e6f23b Explorer!ModuleEntry+0×6d
0007fff0 00000000 kernel32!BaseProcessStart+0×23
```

and implicit wait when a thread yields execution to another thread voluntarily via explicit context swap:

```
THREAD a3072b68  Cid 1fc8.1d94  Teb: 7ffaf000 Win32Thread: bc1e3c20 WAIT:
(Unknown) UserMode Non-Alertable
    a37004d8  QueueObject
    a3072be0  NotificationTimer
IRP List:
    a322be00: (0006,01fc) Flags: 00000000  Mdl: a30b8e30
    a30bcc38: (0006,01fc) Flags: 00000000  Mdl: a35bf530
Not impersonating
DeviceMap                 e12256a0
Owning Process            a3340a10       Image:         IEXPLORE.EXE
Wait Start TickCount      1447963        Ticks: 2484 (0:00:00:38.812)
Context Switch Count      3972                 LargeStack
UserTime                  00:00:00.140
KernelTime                00:00:00.250
Win32 Start Address ntdll!RtlpWorkerThread (0x7c839efb)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f1cc3000 Current f1cc2c38 Base f1cc3000 Limit f1cbf000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0
ChildEBP RetAddr
f1cc2c50 80833465 nt!KiSwapContext+0x26
f1cc2c7c 8082b60f nt!KiSwapThread+0x2e5
f1cc2cc4 808ed620 nt!KeRemoveQueue+0×417
f1cc2d48 8088978c nt!NtRemoveIoCompletion+0xdc
f1cc2d48 7c8285ec nt!KiFastCallEntry+0xfc (TrapFrame @ f1cc2d64)
06ceff70 7c8277db ntdll!KiFastSystemCallRet
06ceff74 7c839f38 ntdll!ZwRemoveIoCompletion+0xc
06ceffb8 77e64829 ntdll!RtlpWorkerThread+0×3d
06ceffec 00000000 kernel32!BaseThreadStart+0×34
```

```
THREAD a3612020  Cid 1980.1a48  Teb: 7ffd9000 Win32Thread: 00000000 WAIT:
(Unknown) UserMode Alertable
    a3612098  NotificationTimer
Not impersonating
DeviceMap                 e689f298
Owning Process            a342d3a0       Image:         explorer.exe
Wait Start TickCount      1346718        Ticks: 103729 (0:00:27:00.765)
Context Switch Count      4
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address ntdll!RtlpTimerThread (0x7c83d3dd)
Start Address kernel32!BaseThreadStartThunk (0x77e617ec)
Stack Init f2453000 Current f2452c80 Base f2453000 Limit f2450000 Call 0
Priority 10 BasePriority 10 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr
f2452c98 80833465 nt!KiSwapContext+0x26
f2452cc4 80828f0b nt!KiSwapThread+0x2e5
f2452d0c 80994812 nt!KeDelayExecutionThread+0×2ab
f2452d54 8088978c nt!NtDelayExecution+0×84
f2452d54 7c8285ec nt!KiFastCallEntry+0xfc (TrapFrame @ f2452d64)
0149ff9c 7c826f4b ntdll!KiFastSystemCallRet
0149ffa0 7c83d424 ntdll!NtDelayExecution+0xc
0149ffb8 77e64829 ntdll!RtlpTimerThread+0×47
0149ffec 00000000 kernel32!BaseThreadStart+0×34
```

Explicit waits in kernel can be originated from GUI threads and their message loops, for example, **Main Thread** (Volume 1, page 436). Blocked GUI thread, **Message Box** pattern (page 177) can be seen as an example of a genuine **Blocked Thread.** Some “blocked” threads are just really **Passive Threads** (Volume 1, page 430).

ZOMBIE PROCESSES

Sometimes when listing processes we see the so called z**ombie processes**. They are better visible in the output of **!vm** command as processes with zero private memory size values:

```
0: kd> !vm

*** Virtual Memory Usage ***
 Physical Memory:      999294 (   3997176 Kb)
 Page File: \??\C:\pagefile.sys
   Current:   5995520 Kb  Free Space:   5324040 Kb
   Minimum:   5995520 Kb  Maximum:      5995520 Kb
 Available Pages:      626415 (   2505660 Kb)
 ResAvail Pages:       902639 (   3610556 Kb)
 Locked IO Pages:         121 (       484 Kb)
 Free System PTEs:     201508 (    806032 Kb)
 Free NP PTEs:          32766 (    131064 Kb)
 Free Special NP:           0 (         0 Kb)
 Modified Pages:          256 (      1024 Kb)
 Modified PF Pages:       256 (      1024 Kb)
 NonPagedPool Usage:    12304 (     49216 Kb)
 NonPagedPool Max:      65359 (    261436 Kb)
 PagedPool 0 Usage:     18737 (     74948 Kb)
 PagedPool 1 Usage:      2131 (      8524 Kb)
 PagedPool 2 Usage:      2104 (      8416 Kb)
 PagedPool 3 Usage:      2140 (      8560 Kb)
 PagedPool 4 Usage:      2134 (      8536 Kb)
 PagedPool Usage:       27246 (    108984 Kb)
 PagedPool Maximum:     67072 (    268288 Kb)
 Shared Commit:         60867 (    243468 Kb)
 Special Pool:              0 (         0 Kb)
 Shared Process:        14359 (     57436 Kb)
 PagedPool Commit:      27300 (    109200 Kb)
 Driver Commit:          1662 (      6648 Kb)
 Committed pages:      501592 (   2006368 Kb)
 Commit limit:        2456879 (   9827516 Kb)

 Total Private:        368810 (   1475240 Kb)
...
...
...
         3654 explorer.exe      2083 (      8332 Kb)
         037c MyService.exe     2082 (      8328 Kb)
         315c explorer.exe      2045 (      8180 Kb)
...
...
...
         0588 svchost.exe        360 (      1440 Kb)
         3f94 csrss.exe          288 (      1152 Kb)
         0acc svchost.exe        245 (       980 Kb)
```

```
0380 smss.exe            38 (       152 Kb)
0004 System               7 (        28 Kb)
6ee8 cmd.exe              0 (         0 Kb)
6d7c cmd.exe              0 (         0 Kb)
6ca8 cmd.exe              0 (         0 Kb)
6b48 IEXPLORE.EXE         0 (         0 Kb)
6ac4 cmd.exe              0 (         0 Kb)
69e8 cmd.exe              0 (         0 Kb)
69dc cmd.exe              0 (         0 Kb)
68dc AcroRd32.exe         0 (         0 Kb)
6860 cmd.exe              0 (         0 Kb)
6858 cmd.exe              0 (         0 Kb)
67d8 cmd.exe              0 (         0 Kb)
6684 AcroRd32.exe         0 (         0 Kb)
6484 cmd.exe              0 (         0 Kb)
6464 cmd.exe              0 (         0 Kb)
6288 cmd.exe              0 (         0 Kb)
626c cmd.exe              0 (         0 Kb)
6260 cmd.exe              0 (         0 Kb)
6258 cmd.exe              0 (         0 Kb)
620c IEXPLORE.EXE         0 (         0 Kb)
60f0 cmd.exe              0 (         0 Kb)
5fa4 cmd.exe              0 (         0 Kb)
5f60 cmd.exe              0 (         0 Kb)
5eec cmd.exe              0 (         0 Kb)
5d24 IEXPLORE.EXE         0 (         0 Kb)
5bd4 cmd.exe              0 (         0 Kb)
5b9c cmd.exe              0 (         0 Kb)
5b10 cmd.exe              0 (         0 Kb)
5b08 cmd.exe              0 (         0 Kb)
5a4c cmd.exe              0 (         0 Kb)
5a08 cmd.exe              0 (         0 Kb)
5934 cmd.exe              0 (         0 Kb)
58b8 cmd.exe              0 (         0 Kb)
56dc cmd.exe              0 (         0 Kb)
558c cmd.exe              0 (         0 Kb)
5588 cmd.exe              0 (         0 Kb)
5574 cmd.exe              0 (         0 Kb)
5430 cmd.exe              0 (         0 Kb)
5424 cmd.exe              0 (         0 Kb)
53b0 cmd.exe              0 (         0 Kb)
5174 explorer.exe         0 (         0 Kb)
5068 cmd.exe              0 (         0 Kb)
5028 IEXPLORE.EXE         0 (         0 Kb)
5004 cmd.exe              0 (         0 Kb)
4f3c javaw.exe            0 (         0 Kb)
4de4 cmd.exe              0 (         0 Kb)
4dd8 cmd.exe              0 (         0 Kb)
4c50 cmd.exe              0 (         0 Kb)
4c48 cmd.exe              0 (         0 Kb)
4c08 cmd.exe              0 (         0 Kb)
4a8c cmd.exe              0 (         0 Kb)
49ac cmd.exe              0 (         0 Kb)
4938 cmd.exe              0 (         0 Kb)
```

```
4928 cmd.exe              0 (         0 Kb)
491c cmd.exe              0 (         0 Kb)
4868 POWERPNT.EXE         0 (         0 Kb)
4724 cmd.exe              0 (         0 Kb)
46cc cmd.exe              0 (         0 Kb)
44a8 cmd.exe              0 (         0 Kb)
43cc cmd.exe              0 (         0 Kb)
4350 cmd.exe              0 (         0 Kb)
4208 cmd.exe              0 (         0 Kb)
41f4 cmd.exe              0 (         0 Kb)
41ec cmd.exe              0 (         0 Kb)
4170 cmd.exe              0 (         0 Kb)
40bc cmd.exe              0 (         0 Kb)
3ddc cmd.exe              0 (         0 Kb)
3dcc cmd.exe              0 (         0 Kb)
3db8 cmd.exe              0 (         0 Kb)
3d88 cmd.exe              0 (         0 Kb)
3d10 cmd.exe              0 (         0 Kb)
3cac cmd.exe              0 (         0 Kb)
3ca4 cmd.exe              0 (         0 Kb)
3c88 cmd.exe              0 (         0 Kb)
337c cmd.exe              0 (         0 Kb)
3310 cmd.exe              0 (         0 Kb)
3308 cmd.exe              0 (         0 Kb)
32f0 cmd.exe              0 (         0 Kb)
32b8 cmd.exe              0 (         0 Kb)
2ed0 cmd.exe              0 (         0 Kb)
2eb8 cmd.exe              0 (         0 Kb)
2e28 cmd.exe              0 (         0 Kb)
2d44 AcroRd32.exe         0 (         0 Kb)
2d24 cmd.exe              0 (         0 Kb)
2c94 cmd.exe              0 (         0 Kb)
2c54 IEXPLORE.EXE         0 (         0 Kb)
2a28 cmd.exe              0 (         0 Kb)
29e4 cmd.exe              0 (         0 Kb)
2990 cmd.exe              0 (         0 Kb)
28c0 cmd.exe              0 (         0 Kb)
25a0 cmd.exe              0 (         0 Kb)
2558 cmd.exe              0 (         0 Kb)
2478 cmd.exe              0 (         0 Kb)
244c cmd.exe              0 (         0 Kb)
23dc cmd.exe              0 (         0 Kb)
2320 cmd.exe              0 (         0 Kb)
2280 cmd.exe              0 (         0 Kb)
2130 cmd.exe              0 (         0 Kb)
205c cmd.exe              0 (         0 Kb)
2014 cmd.exe              0 (         0 Kb)
1fd8 cmd.exe              0 (         0 Kb)
1fa0 cmd.exe              0 (         0 Kb)
1eb8 cmd.exe              0 (         0 Kb)
1d68 IEXPLORE.EXE         0 (         0 Kb)
1cb8 cmd.exe              0 (         0 Kb)
1c9c cmd.exe              0 (         0 Kb)
1c50 cmd.exe              0 (         0 Kb)
```

```
         1a74 cmd.exe              0 (         0 Kb)
         1954 cmd.exe              0 (         0 Kb)
         1948 cmd.exe              0 (         0 Kb)
         06e4 cmd.exe              0 (         0 Kb)
         0650 cmd.exe              0 (         0 Kb)
```

We see lots of cmd.exe processes. Let’s examine a few of them:

```
0: kd> !process 0650
Searching for Process with Cid == 650
PROCESS 89237d88  SessionId: 0  Cid: 0650    Peb: 7ffde000  ParentCid:
037c
    DirBase: f3b31940  ObjectTable: 00000000  HandleCount:   0.
    Image: cmd.exe
    VadRoot 00000000 Vads 0 Clone 0 Private 0. Modified 2. Locked 0.
    DeviceMap e10038a8
    Token                             e4eb5b98
    ElapsedTime                       1 Day 00:16:11.706
    UserTime                          00:00:00.015
    KernelTime                        00:00:00.015
    QuotaPoolUsage[PagedPool]         0
    QuotaPoolUsage[NonPagedPool]      0
    Working Set Sizes (now,min,max)  (7, 50, 345) (28KB, 200KB, 1380KB)
    PeakWorkingSetSize                588
    VirtualSize                       11 Mb
    PeakVirtualSize                   14 Mb
    PageFaultCount                    663
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      0

No active threads
```

```
0: kd> !process 2130
Searching for Process with Cid == 2130
PROCESS 89648020  SessionId: 0  Cid: 2130    Peb: 7ffdc000  ParentCid:
037c
    DirBase: f3b31060  ObjectTable: 00000000  HandleCount:   0.
    Image: cmd.exe
    VadRoot 00000000 Vads 0 Clone 0 Private 0. Modified 2. Locked 0.
    DeviceMap e10038a8
    Token                             e5167bb8
    ElapsedTime                       15:40:17.643
    UserTime                          00:00:00.015
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         0
    QuotaPoolUsage[NonPagedPool]      0
    Working Set Sizes (now,min,max)  (7, 50, 345) (28KB, 200KB, 1380KB)
    PeakWorkingSetSize                545
    VirtualSize                       11 Mb
    PeakVirtualSize                   14 Mb
    PageFaultCount                    621
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      0

No active threads
```

Most of them have Parent PID as 037c which is MyService.exe. Let’s peek inside its handle table:

```
0: kd> !kdexts.handle 0 3 037c
processor number 0, process 0000037c
Searching for Process with Cid == 37c
PROCESS 8a8fa8c0  SessionId: 0  Cid: 037c    Peb: 7ffd8000  ParentCid:
04ac
    DirBase: f3b10360  ObjectTable: e1c276b8  HandleCount: 500.
    Image: MyService.exe

Handle table at e272d000 with 500 Entries in use
0004: Object: e1000638  GrantedAccess: 00000003 Entry: e1caf008
Object: e1000638  Type: (8ad79ad0) KeyedEvent
    ObjectHeader: e1000620 (old version)
        HandleCount: 151  PointerCount: 152
        Directory Object: e1001898  Name: CritSecOutOfMemoryEvent

0008: Object: 8a8cfdf8  GrantedAccess: 001f0003 Entry: e1caf010
Object: 8a8cfdf8  Type: (8ad7a990) Event
    ObjectHeader: 8a8cfde0 (old version)
        HandleCount: 1  PointerCount: 1

000c: Object: e186d690  GrantedAccess: 00000003 Entry: e1caf018
Object: e186d690  Type: (8ad84e70) Directory
    ObjectHeader: e186d678 (old version)
        HandleCount: 150  PointerCount: 181
        Directory Object: e1003b28  Name: KnownDlls
```

```
0010: Object: 8a8d1328  GrantedAccess: 00100020 (Inherit) Entry: e1caf020
Object: 8a8d1328  Type: (8ad74900) File
    ObjectHeader: 8a8d1310 (old version)
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \WINDOWS\system32
{HarddiskVolume1}
...
...
...
0484: Object: 89648020  GrantedAccess: 001f0fff Entry: e1caf908
Object: 89648020  Type: (8ad84900) Process
    ObjectHeader: 89648008 (old version)
        HandleCount: 1  PointerCount: 2
...
...
...
0510: Object: 89237d88  GrantedAccess: 001f0fff Entry: e1cafa20
Object: 89237d88  Type: (8ad84900) Process
    ObjectHeader: 89237d70 (old version)
        HandleCount: 1  PointerCount: 2
...
...
...
```

We may guess that MyService.exe probably forgot to close process handles either after launching cmd.exe or after waiting for their exit when process objects become signaled:

```
0510: Object: 89237d88  GrantedAccess: 001f0fff Entry: e1cafa20
Object: 89237d88  Type: (8ad84900) Process
    ObjectHeader: 89237d70 (old version)
        HandleCount: 1  PointerCount: 2

0: kd> dt _DISPATCHER_HEADER 89237d88
ntdll!_DISPATCHER_HEADER
   +0x000 Type             : 0x3 '' ; PROCESS OBJECT
   +0x001 Absolute         : 0 ''
   +0x001 NpxIrql          : 0 ''
   +0x002 Size             : 0x1e ''
   +0x002 Hand             : 0x1e ''
   +0x003 Inserted         : 0 ''
   +0x003 DebugActive      : 0 ''
   +0x000 Lock             : 1966083
   +0×004 SignalState      : 1
   +0×008 WaitListHead     : _LIST_ENTRY [ 0×89237d90 - 0×89237d90 ]
```

This pattern can also be seen a specialization of a more general **Handle Leak** pattern (Volume 1, page 327).

WILD POINTER

This pattern was briefly mentioned in **Local Buffer Overflow** (Volume 1, page 460) which shows examples of pointers having UNICODE structure in their values. We can also observe pointers with ASCII structure as **.formats** WinDbg command indicates:

```
FAILED_INSTRUCTION_ADDRESS:
65747379`735c5357 ??              ???

IP_ON_HEAP:  65747379735c5357

0:012> .formats rip
Evaluate expression:
  Hex:     65747379`735c5357
  Decimal: 7310595060592825175
  Octal:   0625643467456327051527
  Binary:  01100101 01110100 01110011 01111001 01110011 01011100 01010011
01010111
  Chars:   etsys\SW
  Time:    Wed May 10 22:00:59.282 24767 (GMT+1)
  Float:   low 1.7456e+031 high 7.21492e+022
  Double:  5.30388e+180
```

Here is another example of a pointer having UNICODE structure:

```
FAILED_INSTRUCTION_ADDRESS:
0045004c`00490046 ??              ???

0:014> .formats rip
Evaluate expression:
  Hex:     0045004c`00490046
  Decimal: 19422099815333958
  Octal:   0001050004600022200106
  Binary:  00000000 01000101 00000000 01001100 00000000 01001001 00000000
01000110
  Chars:   .E.L.I.F
  Time:    Wed Jul 19 07:46:21.533 1662 (GMT+1)
  Float:   low 6.70409e-039 high 6.33676e-039
  Double:  2.33646e-307
```

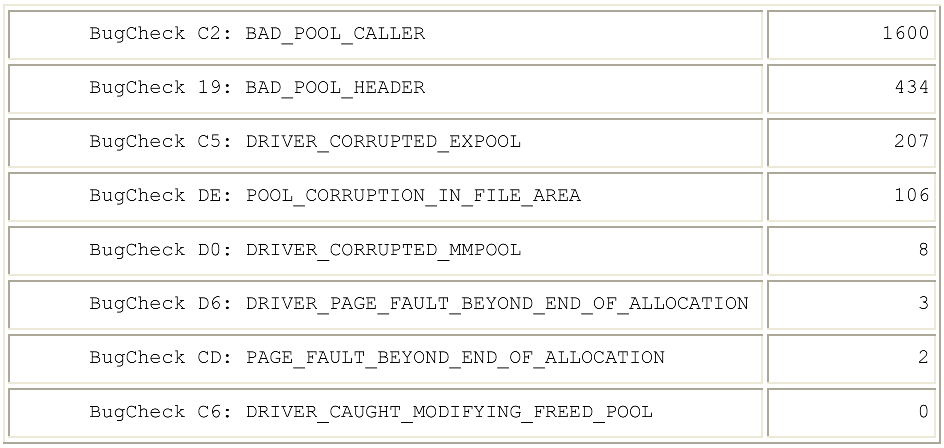
When we have EIP or RIP pointers we have another pattern to name when the value is coincidentally lies inside some valid region of memory: **Wild Code** (page 219)**.** Here is one example of the latter pattern:

```
IP_ON_STACK:
+e11ffa8

STACK_TEXT:
0e11ff7c 098eeef2 0xe11ffa8
0e11ff84 77b6b530 dll!StartWorking+0xcab
0e11ffb8 7c826063 msvcrt!_endthreadex+0xa3
0e11ffec 00000000 kernel32!BaseThreadStart+0×34

0:020> u
0e11ffa8 dcff fdiv st(7),st
...
...
...
```

We see that EIP is very close to EBP/ESP and this explains why **!analyze -v** reports IP_ON_STACK. Clearly floating-point code is not what we should expect. This example shows that wild pointers sometimes are valid but either through code chain or pointer chain the execution reaches **Invalid Pointer** (Volume 1, page 267) and a process or a system crashes.

DYNAMIC MEMORY CORRUPTION (KERNEL POOL)

This is an additional kernel space example to **Dynamic Memory Corruption** pattern (Volume 1, page 257). If kernel pools are corrupt then calls that allocate or free memory result in bugchecks C2 or 19 and in other less frequent bugchecks (from Google stats, page 429):



BugCheck C2: BAD_POOL_CALLER	1600
BugCheck 19: BAD_POOL_HEADER	434
BugCheck C5: DRIVER_CORRUPTED_EXPOOL	207
BugCheck DE: POOL_CORRUPTION_IN_FILE_AREA	106
BugCheck D0: DRIVER_CORRUPTED_MMPOOL	8
BugCheck D6: DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION	3
BugCheck CD: PAGE_FAULT_BEYOND_END_OF_ALLOCATION	2
BugCheck C6: DRIVER_CAUGHT_MODIFYING_FREED_POOL	0

Bugchecks 0xC2 and 0×19 have parameters in their arguments that tell the type of detected pool corruption. Refer to WinDbg help for details or use the variant of **!analyze** command where you can supply optional bugcheck arguments:

```
1: kd> !analyze -show c2
BAD_POOL_CALLER (c2)
The current thread is making a bad pool request.  Typically this is at a
bad IRQL level or double freeing the same allocation, etc.
Arguments:
Arg1: 00000000, The caller is requesting a zero byte pool allocation.
Arg2: 00000000, zero.
Arg3: 00000000, the pool type being allocated.
Arg4: 00000000, the pool tag being used.
```

```
1: kd> !analyze -show 19 2 1 1 1
BAD_POOL_HEADER (19)
The pool is already corrupt at the time of the current request.
This may or may not be due to the caller.
The internal pool links must be walked to figure out a possible cause of
the problem, and then special pool applied to the suspect tags or the
driver
verifier to a suspect driver.
Arguments:
Arg1: 00000002, the verifier pool pattern check failed.  The owner has
likely corrupted the pool block
Arg2: 00000001, the pool entry being checked.
Arg3: 00000001, size of the block.
Arg4: 00000001, 0.
```

If we enable special pool on suspected drivers we might get these bugchecks too with the following Google frequency:

BugCheck C1: SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION	59
BugCheck D5: DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL	5
BugCheck CC: PAGE_FAULT_IN_FREED_SPECIAL_POOL	1

Here is one example of nonpaged pool corruption detected during *free* operation with the following **!analyze -v** output:

```
BAD_POOL_HEADER (19)
The pool is already corrupt at the time of the current request.
This may or may not be due to the caller.
The internal pool links must be walked to figure out a possible cause of
the problem, and then special pool applied to the suspect tags or the
driver
verifier to a suspect driver.
Arguments:
Arg1: 00000020, a pool block header size is corrupt.
Arg2: a34583b8, The pool entry we were looking for within the page.
Arg3: a34584f0, The next pool entry.
Arg4: 0a270001, (reserved)

POOL_ADDRESS:  a34583b8 Nonpaged pool

PROCESS_NAME:  process.exe

CURRENT_IRQL:  2
```

```
STACK_TEXT:
b80a60cc 808927bb nt!KeBugCheckEx+0x1b
b80a6134 80892b6f nt!ExFreePoolWithTag+0x477
b80a6144 b9591400 nt!ExFreePool+0xf
WARNING: Stack unwind information not available. Following frames may be
wrong.
b80a615c b957b954 driver+0x38400
b80a617c b957d482 driver+0x22954
b80a61c0 b957abf4 driver+0x24482
b80a6260 b957ccef driver+0x21bf4
b80a62a8 8081df65 driver+0x23cef
b80a62bc f721ac45 nt!IofCallDriver+0x45
b80a62e4 8081df65 fltMgr!FltpDispatch+0x6f
b80a62f8 b99de70b nt!IofCallDriver+0x45
b80a6308 b99da6ee filter!Dispatch+0xfb
b80a6318 8081df65 filter!dispatch+0x6e
b80a632c b9bdebfe nt!IofCallDriver+0x45
b80a6334 8081df65 2ndfilter!Redirect+0x7ea
b80a6348 b9bd1756 nt!IofCallDriver+0x45
b80a6374 b9bd1860 3rdfilter!PassThrough+0x136
b80a6384 8081df65 3rdfilter!Dispatch+0x80
b80a6398 808f5437 nt!IofCallDriver+0x45
b80a63ac 808ef963 nt!IopSynchronousServiceTail+0x10b
b80a63d0 8088978c nt!NtQueryDirectoryFile+0x5d
b80a63d0 7c8285ec nt!KiFastCallEntry+0xfc
00139524 7c8274eb ntdll!KiFastSystemCallRet
00139528 77e6ba40 ntdll!NtQueryDirectoryFile+0xc
00139830 77e6bb5f kernel32!FindFirstFileExW+0x3d5
00139850 6002665e kernel32!FindFirstFileW+0x16
00139e74 60026363 process+0x2665e
0013a328 60027852 process+0x26363
0013a33c 60035b58 process+0x27852
0013b104 600385ff process+0x35b58
0013b224 612cb643 process+0x385ff
0013b988 612cc109 dll!FileDialog+0xc53
0013bba0 612cb47b dll!FileDialog+0x1719
0013c2c0 7739b6e3 dll!FileDialog+0xa8b
0013c2ec 77395f82 USER32!InternalCallWinProc+0x28
0013c368 77395e22 USER32!UserCallDlgProcCheckWow+0x147
0013c3b0 7739c9c6 USER32!DefDlgProcWorker+0xa8
0013c3d8 7c828536 USER32!__fnDWORD+0x24
0013c3d8 808308f4 ntdll!KiUserCallbackDispatcher+0x2e
b80a66b8 8091d6d1 nt!KiCallUserMode+0x4
b80a6710 bf8a2622 nt!KeUserModeCallback+0x8f
b80a6794 bf8a2517 win32k!SfnDWORD+0xb4
b80a67dc bf8a13d9 win32k!xxxSendMessageToClient+0x133
b80a6828 bf85ae67 win32k!xxxSendMessageTimeout+0x1a6
b80a684c bf8847a1 win32k!xxxWrapSendMessage+0x1b
b80a6868 bf8c1459 win32k!NtUserfnNCDESTROY+0x27
b80a68a0 8088978c win32k!NtUserMessageCall+0xc0
b80a68a0 7c8285ec nt!KiFastCallEntry+0xfc
0013c3d8 7c828536 ntdll!KiFastSystemCallRet
0013c3d8 808308f4 ntdll!KiUserCallbackDispatcher+0x2e
b80a6b7c 8091d6d1 nt!KiCallUserMode+0x4
```

```
b80a6bd4 bf8a2622 nt!KeUserModeCallback+0x8f
b80a6c58 bf8a23a0 win32k!SfnDWORD+0xb4
b80a6ca0 bf8a13d9 win32k!xxxSendMessageToClient+0x118
b80a6cec bf85ae67 win32k!xxxSendMessageTimeout+0x1a6
b80a6d10 bf8c148c win32k!xxxWrapSendMessage+0x1b
b80a6d40 8088978c win32k!NtUserMessageCall+0x9d
b80a6d40 7c8285ec nt!KiFastCallEntry+0xfc
0013f474 7c828536 ntdll!KiFastSystemCallRet
0013f4a0 7739d1ec ntdll!KiUserCallbackDispatcher+0x2e
0013f4dc 7738cf29 USER32!NtUserMessageCall+0xc
0013f4fc 612d3276 USER32!SendMessageA+0x7f
0013f63c 611add41 dll!SubWindow+0x3dc6
0013f658 7739b6e3 dll!SetWindowText+0x37a1
0013f684 7739b874 USER32!InternalCallWinProc+0x28
0013f6fc 7739ba92 USER32!UserCallWinProcCheckWow+0x151
0013f764 7739bad0 USER32!DispatchMessageWorker+0x327
0013f774 61221ca8 USER32!DispatchMessageW+0xf
0013f7e0 0040156d dll!MainLoop+0x2c8
0013ff24 00401dfa process+0x156d
0013ffc0 77e6f23b process+0x1dfa
0013fff0 00000000 kernel32!BaseProcessStart+0x23

MODULE_NAME: driver

IMAGE_NAME:  driver.sys
```

We see that WinDbg pointed to driver.sys by using a procedure described in **Component Identification** article (Volume 1, page 46).

Any OS component could corrupt the pool prior to the detection as the bugcheck description says: “The pool is already corrupt at the time of the current request.”. What other evidence can reinforce our belief in driver.sys? Let’s look at our pool entry tag first:

```
1: kd> !pool a34583b8
Pool page a34583b8 region is Nonpaged pool
 a3458000 size:  270 previous size:    0  (Allocated)  Thre (Protected)
 a3458270 size:   10 previous size:  270  (Free)       RxIr
 a3458280 size:   40 previous size:   10  (Allocated)  Vadl
 a34582c0 size:   98 previous size:   40  (Allocated)  File (Protected)
 a3458358 size:    8 previous size:   98  (Free)       Vadl
 a3458360 size:   50 previous size:    8  (Allocated)  Gsem
 a34583b0 size:    8 previous size:   50  (Free)       CcSc
*a34583b8 size:  138 previous size:    8  (Allocated) *DRIV
  Owning component : Unknown (update pooltag.txt)
a34584f0 is not a valid large pool allocation, checking large session
pool…
a34584f0 is freed (or corrupt) pool
Bad allocation size @a34584f0, zero is invalid

***
*** An error (or corruption) in the pool was detected;
*** Attempting to diagnose the problem.
***
*** Use !poolval a3458000 for more details.
***

Pool page [ a3458000 ] is __inVALID.

Analyzing linked list...
[ a34583b8 --> a34583d8 (size = 0x20 bytes)]: Corrupt region
[ a34583f8 --> a34585e8 (size = 0x1f0 bytes)]: Corrupt region

Scanning for single bit errors...

None found
```

We see that the tag is DRIV and we know either from association or from similar problems in the past that it belongs to driver.sys. Let’s dump our pool entry contents to see if there are any symbolic hints in it:

```
1: kd> dps a34583b8
a34583b8 0a270001
a34583bc 5346574e
a34583c0 00000000
a34583c4 00000000
a34583c8 b958f532 driver+0×36532
a34583cc a3471010
a34583d0 0000012e
a34583d4 00000001
a34583d8 00041457
a34583dc 05af0026
a34583e0 00068002
a34583e4 7b9ec6f5
a34583e8 ffffff00
a34583ec 73650cff
```

```
a34583f0 7461445c
a34583f4 97a10061
a34583f8 ff340004
a34583fc c437862a
a3458400 6a000394
a3458404 00000038
a3458408 00000000
a345840c bf000000
a3458410 bf0741b5
a3458414 f70741b5
a3458418 00000000
a345841c 00000000
a3458420 00000000
a3458424 00000000
a3458428 05000000
a345842c 34303220
a3458430 31323332
a3458434 ff322d36
```

Indeed we see that the possible code pointer driver+0×36532 and the code around this address look normal:

```
3: kd> .asm no_code_bytes
Assembly options: no_code_bytes

3: kd> u b958f532
driver+0x36532:
b958f532 push    2Ch
b958f534 push    offset driver+0x68d08 (b95c1d08)
b958f539 call    driver+0x65c50 (b95bec50)
b958f53e mov     byte ptr [ebp-19h],0
b958f542 and     dword ptr [ebp-24h],0
b958f546 call    dword ptr [driver+0x65f5c (b95bef5c)]
b958f54c mov     ecx,dword ptr [ebp+0Ch]
b958f54f cmp     eax,ecx

3: kd> ub b958f532
driver+0x36528:
b958f528 leave
b958f529 ret     18h
b958f52c int     3
b958f52d int     3
b958f52e int     3
b958f52f int     3
b958f530 int     3
b958f531 int     3
```

INSUFFICIENT VIRTUAL MEMORY

I'm coming back to my old **Insufficient Memory** pattern range. See **Committed Memory** (Volume 1, page 302), **Handle Leak** (Volume 1, page 327), **Kernel Pool** (Volume 1, page 440), **PTE** (page 159). Here we discuss the user space case when we don't have enough virtual memory available for reservation due to memory fragmentation. For example, a java virtual machine is pre-allocating memory for its garbage-collected heap. However after installing some 3rd-party software the amount of pre-allocated memory is less than expected. In such cases it is possible to do comparative memory dump analysis to see the difference in virtual address spaces. Original memory dump has this module distribution in memory:

```
0:000> lm
start    end        module name
00400000 0040b000   javaw      (deferred)
009e0000 009e7000   hpi        (deferred)
00a30000 00a3e000   verify     (deferred)
00a40000 00a59000   java       (deferred)
00a60000 00a6d000   zip        (deferred)
03ff0000 03fff000   net        (deferred)
040a0000 040a8000   nio        (deferred)
040b0000 0410a000   hnetcfg    (deferred)
041d0000 042e2000   awt        (deferred)
04540000 04591000   fontmanager   (deferred)
04620000 04670000   msctf      (deferred)
047c0000 047de000   jpeg       (deferred)
05820000 05842000   dcpr       (deferred)
05920000 05932000   pkcs11wrapper   (deferred)
08000000 08139000   jvm        (deferred)
10000000 100e0000   moduleA    (deferred)
68000000 68035000   rsaenh     (deferred)
6e220000 6e226000   RMProcessLink   (deferred)
71ae0000 71ae8000   wshtcpip   (deferred)
71b20000 71b61000   mswsock    (deferred)
71bf0000 71bf8000   ws2help    (deferred)
71c00000 71c17000   ws2_32     (deferred)
71c20000 71c32000   tsappcmp   (deferred)
71c40000 71c97000   netapi32   (deferred)
73070000 73097000   winspool   (deferred)
76290000 762ad000   imm32      (deferred)
76920000 769e2000   userenv    (deferred)
76aa0000 76acd000   winmm      (deferred)
76b70000 76b7b000   psapi      (deferred)
76ed0000 76efa000   dnsapi     (deferred)
76f10000 76f3e000   wldap32    (deferred)
76f50000 76f63000   secur32    (deferred)
76f70000 76f77000   winrnr     (deferred)
76f80000 76f85000   rasadhlp   (deferred)
77380000 77411000   user32     (deferred)
```

```
77670000 777a9000   ole32      (deferred)
77ba0000 77bfa000   msvcrt     (deferred)
77c00000 77c48000   gdi32      (deferred)
77c50000 77cef000   rpcrt4     (deferred)
77e40000 77f42000   kernel32   (deferred)
77f50000 77feb000   advapi32   (deferred)
78130000 781cb000   msvcr80    (deferred)
7c800000 7c8c0000   ntdll      (pdb symbols)
```

We see the big gap between 100e0000 and 68000000 addresses. This means that it is theoretically possible to reserve and/or commit up to 57F20000 bytes (about 1.4Gb). **!address** WinDbg command shows that at least 1.1Gb region (shown in bold below) was reserved indeed:

```
0:000> !address
00000000 : 00000000 - 00010000
              Type     00000000
              Protect  00000001 PAGE_NOACCESS
              State    00010000 MEM_FREE
              Usage    RegionUsageFree
00010000 : 00010000 - 00001000
              Type     00020000 MEM_PRIVATE
              Protect  00000004 PAGE_READWRITE
              State    00001000 MEM_COMMIT
              Usage    RegionUsageEnvironmentBlock
00011000 : 00011000 - 0000f000
              Type     00000000
              Protect  00000001 PAGE_NOACCESS
              State    00010000 MEM_FREE
              Usage    RegionUsageFree
00020000 : 00020000 - 00001000
              Type     00020000 MEM_PRIVATE
              Protect  00000004 PAGE_READWRITE
              State    00001000 MEM_COMMIT
              Usage    RegionUsageProcessParametrs
00021000 : 00021000 - 0000f000
              Type     00000000
              Protect  00000001 PAGE_NOACCESS
              State    00010000 MEM_FREE
              Usage    RegionUsageFree
00030000 : 00030000 - 00003000
              Type     00020000 MEM_PRIVATE
              Protect  00000140 <unk>
              State    00001000 MEM_COMMIT
              Usage    RegionUsageStack
              Pid.Tid  97c.1b3c
...
...
...
100e0000 : 100e0000 - 000a0000
              Type     00020000 MEM_PRIVATE
              Protect  00000040 PAGE_EXECUTE_READWRITE
              State    00001000 MEM_COMMIT
              Usage    RegionUsageIsVAD
         10180000 - 05cc0000
              Type     00020000 MEM_PRIVATE
              State    00002000 MEM_RESERVE
              Usage    RegionUsageIsVAD
         15e40000 - 004f3000
              Type     00020000 MEM_PRIVATE
              Protect  00000040 PAGE_EXECUTE_READWRITE
              State    00001000 MEM_COMMIT
              Usage    RegionUsageIsVAD
         16333000 - 45bad000
              Type     00020000 MEM_PRIVATE
              State    00002000 MEM_RESERVE
              Usage    RegionUsageIsVAD
         5bee0000 - 00ac0000
              Type     00020000 MEM_PRIVATE
              Protect  00000040 PAGE_EXECUTE_READWRITE
              State    00001000 MEM_COMMIT
              Usage    RegionUsageIsVAD
         5c9a0000 - 03540000
              Type     00020000 MEM_PRIVATE
              State    00002000 MEM_RESERVE
              Usage    RegionUsageIsVAD
5fee0000 : 5fee0000 - 00120000
              Type     00000000
              Protect  00000001 PAGE_NOACCESS
              State    00010000 MEM_FREE
              Usage    RegionUsageFree
...
...
...
```

Looking at the problem memory dump we see that the gap is smaller (less than 1.1Gb):

```
0:000> lm
start    end         module name
00400000 0040b000   javaw      (deferred)
08000000 08139000   jvm        (deferred)
10000000 10007000   hpi        (deferred)
51120000 511bb000   msvcr80  # (private pdb symbols)
520f0000 520fe000   verify     (deferred)
52100000 52119000   java       (deferred)
52120000 5212d000   zip        (deferred)
52130000 5213f000   net        (deferred)
52140000 52148000   nio        (deferred)
52150000 52262000   awt        (deferred)
52270000 522c1000   fontmanager   (deferred)
522d0000 52320000   MSCTF      (deferred)
52330000 5234e000   jpeg       (deferred)
52350000 52372000   dcpr       (deferred)
```

```
52510000 52522000   pkcs11wrapper   (deferred)
5f270000 5f2ca000   hnetcfg    (deferred)
60000000 60029000   3rdPartyHook   (deferred)
61e80000 61e86000   detoured   (export symbols)
68000000 68035000   rsaenh     (deferred)
71ae0000 71ae8000   wshtcpip   (deferred)
71b20000 71b61000   mswsock    (deferred)
71bf0000 71bf8000   ws2help    (deferred)
71c00000 71c17000   ws2_32     (deferred)
71c20000 71c32000   tsappcmp   (deferred)
71c40000 71c97000   netapi32   (deferred)
73070000 73097000   winspool   (deferred)
76290000 762ad000   imm32      (deferred)
76920000 769e2000   userenv    (deferred)
76aa0000 76acd000   winmm      (deferred)
76b70000 76b7b000   psapi      (deferred)
76ed0000 76efa000   dnsapi     (deferred)
76f10000 76f3e000   wldap32    (deferred)
76f50000 76f63000   secur32    (deferred)
76f70000 76f77000   winrnr     (deferred)
76f80000 76f85000   rasadhlp   (deferred)
77380000 77411000   user32     (pdb symbols)
77670000 777a9000   ole32      (deferred)
77ba0000 77bfa000   msvcrt     (deferred)
77c00000 77c48000   gdi32      (deferred)
77c50000 77cef000   rpcrt4     (deferred)
77e40000 77f42000   kernel32   (pdb symbols)
77f50000 77feb000   advapi32   (pdb symbols)
7c340000 7c396000   msvcr71    (deferred)
7c800000 7c8c0000   ntdll      (pdb symbols)
```

!address command shows that less memory was reserved in the latter case (about 896Mb):

```
0:000> !address
...
...
...
10010000 : 10010000 - 000a0000
             Type     00020000 MEM_PRIVATE
              Protect  00000040 PAGE_EXECUTE_READWRITE
              State    00001000 MEM_COMMIT
              Usage    RegionUsageIsVAD
         100b0000 - 04a70000
              Type     00020000 MEM_PRIVATE
              State    00002000 MEM_RESERVE
              Usage    RegionUsageIsVAD
         14b20000 - 004a6000
              Type     00020000 MEM_PRIVATE
              Protect  00000040 PAGE_EXECUTE_READWRITE
              State    00001000 MEM_COMMIT
              Usage    RegionUsageIsVAD
```

```
          14fc6000 - 3804a000
              Type     00020000 MEM_PRIVATE
              State    00002000 MEM_RESERVE
              Usage    RegionUsageIsVAD
          4d010000 - 00ac0000
              Type     00020000 MEM_PRIVATE
              Protect  00000040 PAGE_EXECUTE_READWRITE
              State    00001000 MEM_COMMIT
              Usage    RegionUsageIsVAD
          4dad0000 - 03540000
              Type     00020000 MEM_PRIVATE
              State    00002000 MEM_RESERVE
              Usage    RegionUsageIsVAD
51010000 : 51010000 - 00110000
              Type     00000000
              Protect  00000001 PAGE_NOACCESS
              State    00010000 MEM_FREE
              Usage    RegionUsageFree
...
...
...
```

Looking at module list again we notice that most java runtime modules were shifted to 50000000 address range. We also notice that new the *3rdPartyHook* and detoured modules appear in our problem memory dump.

Module information is missing for detoured module:

```
0:000> lmv m detoured
start    end        module name
61e80000 61e86000   detoured   (deferred)
    Image path: C:\WINDOWS\system32\detoured.dll
    Image name: detoured.dll
    Timestamp:        Thu Feb 07 04:14:16 2008 (47AA8598)
    CheckSum:         0000EF91
    ImageSize:        00006000
    File version:     0.0.0.0
    Product version:  0.0.0.0
    File flags:       0 (Mask 0)
    File OS:          0 Unknown Base
    File type:        0.0 Unknown
    File date:        00000000.00000000
    Translations:     0000.04b0 0000.04e0 0409.04b0 0409.04e0
```

Applying **Unknown Component** pattern (Volume 1, page 367) we see that is Microsoft Research Detours Package:

```
0:000> db 61e80000 61e86000
61e80000  MZ..............
61e80010  ........@.......
61e80020  ................
61e80030  ................
61e80040  ........!..L.!Th
61e80050  is program canno
61e80060  t be run in DOS
61e80070  mode....$.......
61e80080  5...q...q...q...
61e80090  ....r...q...p...
61e800a0  V%..p...V%..p...
61e800b0  V%..p...V%..p...
61e800c0  Richq...........
61e800d0  ................
61e800e0  PE..L......G....
61e800f0  .......!........
61e80100  ................
...
...
...
...
61e84390  ..P.r.o.d.u.c.t.
61e843a0  N.a.m.e…..M.i.
61e843b0  c.r.o.s.o.f.t. .
61e843c0  R.e.s.e.a.r.c.h.
61e843d0  .D.e.t.o.u.r.s.
61e843e0  .P.a.c.k.a.g.e.
61e843f0  ….j.#…P.r.o.
61e84400  d.u.c.t.V.e.r.s.
61e84410  i.o.n…P.r.o.f.
61e84420  e.s.s.i.o.n.a.l.
61e84430  .V.e.r.s.i.o.n.
61e84440  .2…1. .B.u.i.
61e84450  l.d._.2.1.0…..
61e84460  D…..V.a.r.F.i.
...
...
...

0:000> du 61e843a0+C
61e843ac  "Microsoft Research Detours Packa"
61e843ec  "ge"
```

We can also see that 3rdPartyHook module imports this library and lots of kernel32 API related to memory allocation, file mapping and loading DLLs (see **No Component Symbols** pattern, Volume 1, page 298):

```
0:000> !dh 60000000
...
...
...
OPTIONAL HEADER VALUES
     10B magic #
    8.00 linker version
   18000 size of code
    F000 size of initialized data
       0 size of uninitialized data
   13336 address of entry point
    1000 base of code
         ----- new -----
60000000 image base
    1000 section alignment
    1000 file alignment
       2 subsystem (Windows GUI)
    4.00 operating system version
    0.00 image version
    4.00 subsystem version
   29000 size of image
    1000 size of headers
   3376F checksum
00100000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
       0 [       0] address [size] of Export Directory
   218CC [      8C] address [size] of Import Directory
   25000 [     5F4] address [size] of Resource Directory
       0 [       0] address [size] of Exception Directory
   28000 [    19E0] address [size] of Security Directory
   26000 [    2670] address [size] of Base Relocation Directory
   19320 [      1C] address [size] of Debug Directory
       0 [       0] address [size] of Description Directory
       0 [       0] address [size] of Special Directory
       0 [       0] address [size] of Thread Storage Directory
   1F3C0 [      40] address [size] of Load Configuration Directory
       0 [       0] address [size] of Bound Import Directory
   19000 [     2B0] address [size] of Import Address Table Directory
       0 [       0] address [size] of Delay Import Directory
       0 [       0] address [size] of COR20 Header Directory
       0 [       0] address [size] of Reserved Directory
...
...
...

0:000> dds 60000000+19000 60000000+19000+2B0
60019064  7c82b0dc ntdll!RtlReAllocateHeap
60019068  77e4ec39 kernel32!HeapDestroy
6001906c  77e41fba kernel32!GetSystemTimeAsFileTime
60019070  77e619d1 kernel32!GetTickCount
60019074  77e69577 kernel32!QueryPerformanceCounter
60019078  7c82a9be ntdll!RtlSizeHeap
```

```
6001907c  77e82060 kernel32!SetUnhandledExceptionFilter
60019080  77e7690d kernel32!UnhandledExceptionFilter
60019084  77e42004 kernel32!TerminateProcess
60019088  7c82a136 ntdll!RtlRestoreLastWin32Error
6001908c  77e77a5f kernel32!SuspendThread
60019090  77e76a26 kernel32!SetThreadContext
60019094  77e77ae3 kernel32!GetThreadContext
60019098  77e73347 kernel32!FlushInstructionCache
6001909c  77e5f38b kernel32!ResumeThread
600190a0  77e616a8 kernel32!InterlockedCompareExchange
600190a4  77e645a9 kernel32!VirtualAlloc
600190a8  77e41fe3 kernel32!VirtualProtect
600190ac  77e66ed1 kernel32!VirtualQuery
600190b0  77e44960 kernel32!GetLogicalDriveStringsA
600190b4  77eab401 kernel32!GetVolumeNameForVolumeMountPointA
600190b8  77e6794d kernel32!GetACP
600190bc  77e6f3cf kernel32!GetLocaleInfoA
600190c0  77e622b7 kernel32!GetThreadLocale
600190c4  77e69d74 kernel32!GetVersionExA
600190c8  77e4beab kernel32!RaiseException
600190cc  77e60037 kernel32!GetSystemDirectoryA
600190d0  77e52bf4 kernel32!GetWindowsDirectoryA
600190d4  77e5c7a8 kernel32!lstrcmpA
600190d8  77e46c99 kernel32!OutputDebugStringA
600190dc  77e5bd7d kernel32!CreateEventA
600190e0  77e62311 kernel32!SetEvent
600190e4  77e51281 kernel32!ExpandEnvironmentStringsA
600190e8  77e9f365 kernel32!MoveFileA
600190ec  77e5da00 kernel32!IsDebuggerPresent
600190f0  77e9e4b1 kernel32!QueryDosDeviceA
600190f4  7c829e08 ntdll!RtlGetLastWin32Error
600190f8  77e63d7a kernel32!GetProcAddress
600190fc  77e41dc6 kernel32!LoadLibraryA
60019100  7c829e17 ntdll!RtlFreeHeap
60019104  77e62419 kernel32!LocalFree
60019108  7c829fd6 ntdll!RtlAllocateHeap
6001910c  77e63ec7 kernel32!GetProcessHeap
60019110  77ea2186 kernel32!VerifyVersionInfoA
60019114  7c81379f ntdll!VerSetConditionMask
60019118  77e63143 kernel32!WideCharToMultiByte
6001911c  77e70550 kernel32!SizeofResource
60019120  77e6b11b kernel32!SetHandleCount
60019124  77e69bf9 kernel32!LoadResource
60019128  77e511e1 kernel32!FindResourceA
6001912c  77e7388c kernel32!FindResourceExA
60019130  77e5be30 kernel32!lstrlenA
60019134  77e424de kernel32!Sleep
60019138  77ea2cb1 kernel32!WaitNamedPipeA
6001913c  77e63e6f kernel32!CloseHandle
60019140  77e5e123 kernel32!OpenEventA
60019144  77e622c9 kernel32!lstrlenW
60019148  77e62fc7 kernel32!GetCurrentThreadId
6001914c  77e5fdd4 kernel32!OpenProcess
60019150  77e63c78 kernel32!GetCurrentProcessId
```

```
60019154  7c81a3ab ntdll!RtlLeaveCriticalSection
60019158  7c81a360 ntdll!RtlEnterCriticalSection
6001915c  77e4cabf kernel32!GetComputerNameA
60019160  77e6f032 kernel32!ProcessIdToSessionId
60019164  77e645ff kernel32!GetModuleFileNameA
60019168  77e6474a kernel32!GetModuleHandleA
6001916c  77e62f9d kernel32!GetCurrentProcess
60019170  77e49968 kernel32!GetCurrentDirectoryA
60019174  77e61c7b kernel32!WaitForSingleObject
60019178  77e63868 kernel32!GetCurrentThread
6001917c  7c82c988 ntdll!RtlDeleteCriticalSection
60019180  77e67861 kernel32!InitializeCriticalSection
60019184  77e6b1a1 kernel32!FreeLibrary
60019188  77e63f41 kernel32!UnmapViewOfFile
6001918c  77e643f1 kernel32!MapViewOfFile
60019190  77e6b65f kernel32!OpenFileMappingA
60019194  77e61694 kernel32!InterlockedExchange
60019198  77e4d2fb kernel32!DeleteFileA
...
...
...
60019294  00000000
60019298  61e81000 detoured!Detoured
6001929c  00000000
...
...
...
```

This warrants the suspicion that 3rdPartyHook somehow optimized the virtual address space for its own purposes and this resulted in more fragmented virtual address space.

WILD CODE

The case when a function pointer or a return address becomes a **Wild Pointer** (page 202) and EIP or RIP value lies in a valid region of memory the execution path may continue through a region called **Wild Code**. This might loop on itself or eventually reach non-executable or invalid pages and produce an exception. **Local Buffer Overflow** (Volume 1, page 460) might lead to this behavior and also data corruption that overwrites function pointers with valid memory addresses.

My favorite example is when a function pointer points to zeroed pages with EXECUTE page attribute. What will happen next when we dereference it? All zeroes are perfect x86/x64 code:

```
0:001> dd 0000000`771afdf0
00000000`771afdf0  00000000 00000000 00000000 00000000
00000000`771afe00  00000000 00000000 00000000 00000000
00000000`771afe10  00000000 00000000 00000000 00000000
00000000`771afe20  00000000 00000000 00000000 00000000
00000000`771afe30  00000000 00000000 00000000 00000000
00000000`771afe40  00000000 00000000 00000000 00000000
00000000`771afe50  00000000 00000000 00000000 00000000
00000000`771afe60  00000000 00000000 00000000 00000000

0:001> u
ntdll!DbgUserBreakPoint:
00000000`771afe00 0000    add     byte ptr [rax],al
00000000`771afe02 0000    add     byte ptr [rax],al
00000000`771afe04 0000    add     byte ptr [rax],al
00000000`771afe06 0000    add     byte ptr [rax],al
00000000`771afe08 0000    add     byte ptr [rax],al
00000000`771afe0a 0000    add     byte ptr [rax],al
00000000`771afe0c 0000    add     byte ptr [rax],al
00000000`771afe0e 0000    add     byte ptr [rax],al
```

Now if RAX points to a valid memory page with WRITE attribute the code will modify the first byte at that address:

```
0:001> dq @rax
000007ff`fffdc000 00000000`00000000 00000000`035a0000
000007ff`fffdc010 00000000`0359c000 00000000`00000000
000007ff`fffdc020 00000000`00001e00 00000000`00000000
000007ff`fffdc030 000007ff`fffdc000 00000000`00000000
000007ff`fffdc040 00000000`0000142c 00000000`00001504
000007ff`fffdc050 00000000`00000000 00000000`00000000
000007ff`fffdc060 000007ff`fffd8000 00000000`00000000
000007ff`fffdc070 00000000`00000000 00000000`00000000
```

Therefore the code will be perfectly executed:

```
0:001> t
ntdll!DbgBreakPoint+0x2:
00000000`771afdf2 0000    add     byte ptr [rax],al
ds:000007ff`fffdc000=00

0:001> t
ntdll!DbgBreakPoint+0x4:
00000000`771afdf4 0000    add     byte ptr [rax],al
ds:000007ff`fffdc000=00

0:001> t
ntdll!DbgBreakPoint+0x6:
00000000`771afdf6 0000    add     byte ptr [rax],al
ds:000007ff`fffdc000=00

0:001> t
ntdll!DbgBreakPoint+0x8:
00000000`771afdf8 0000    add     byte ptr [rax],al
ds:000007ff`fffdc000=00

0:001> t
ntdll!DbgBreakPoint+0xa:
00000000`771afdfa 0000    add     byte ptr [rax],al
ds:000007ff`fffdc000=00
```

HARDWARE ERROR

This pattern occurs frequently and it can be internal CPU malfunction due to overheating, RAM or hard disk I/O problem that usually results in the appropriate bug-check. The most frequent one is the 6th from the top of **Bugcheck Frequency Table** (page 429):

- BUGCHECK 9C: MACHINE_CHECK_EXCEPTION

Other relevant bugchecks include:

- BUGCHECK 7B: INACCESSIBLE_BOOT_DEVICE
- BUGCHECK 77: KERNEL_STACK_INPAGE_ERROR
- BUGCHECK 7A: KERNEL_DATA_INPAGE_ERROR

Another bugcheck from this category can be triggered on purpose to get a crash dump of a hanging or slow system (Volume 1, page 135):

- BUGCHECK 80: NMI_HARDWARE_FAILURE

Please also note that other popular bugchecks like

- BUGCHECK 7F: UNEXPECTED_KERNEL_MODE_TRAP
- BUGCHECK 50: PAGE_FAULT_IN_NONPAGED_AREA

can result from RAM problems but we should try to find a software cause first.

Sometimes the following bugchecks like

- BUGCHECK 7E: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED

report EXCEPTION_DOESNOT_MATCH_CODE where read or write address doesn't correspond to faulted instruction at EIP:

```
SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (7e)
This is a very common bugcheck. Usually the exception address pinpoints
the driver/function that caused the problem. Always note this address
as well as the link date of the driver/image that contains this address.
Arguments:
Arg1: c0000005, The exception code that was not handled
Arg2: bf802671, The address that the exception occurred at
Arg3: f10b8c74, Exception Record Address
Arg4: f10b88c4, Context Record Address

FAULTING_IP:
driver!AcquireSemaphoreShared+4
bf802671 90 nop

EXCEPTION_RECORD: f10b8c74 -- (.exr fffffffff10b8c74)
ExceptionAddress: bf802671 (driver!AcquireSemaphoreShared+0x00000004)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000000
NumberParameters: 2
Parameter[0]: 00000001
Parameter[1]: 0000000c
Attempt to write to address 0000000c

CONTEXT: f10b88c4 -- (.cxr fffffffff10b88c4)
eax=884d2d01 ebx=0000000c ecx=00000000 edx=80010031 esi=8851ef60
edi=bc3846d4
eip=bf802671 esp=f10b8d3c ebp=f10b8d70 iopl=0 nv up ei pl nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010206
driver!AcquireSemaphoreShared+0x4:
bf802671 90 nop
Resetting default scope

WRITE_ADDRESS: 0000000c

EXCEPTION_DOESNOT_MATCH_CODE: This indicates a hardware error.
Instruction at bf802671 does not read/write to 0000000c
```

Code mismatch can also happen in user mode but from my experience it usually results from improper **Hooked Function** (Volume 1, page 468) or similar corruption:

```
EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 7c848768 (ntdll!_LdrpInitialize+0x00000184)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000001
NumberParameters: 0

DEFAULT_BUCKET_ID: CODE_ADDRESS_MISMATCH

WRITE_ADDRESS: f774f120
```

```
FAULTING_IP:
ntdll!_LdrpInitialize+184
7c848768 cc int 3

EXCEPTION_DOESNOT_MATCH_CODE: This indicates a hardware error.
Instruction at 7c848768 does not read/write to f774f120

STACK_TEXT:
0012fd14 7c8284c5 0012fd28 7c800000 00000000 ntdll!_LdrpInitialize+0x184
00000000 00000000 00000000 00000000 00000000
ntdll!KiUserApcDispatcher+0x25
```

In such cases EIP might point to the middle of the expected instruction (see **Wild Code**, page **Error! Bookmark not defined.**):

```
FAULTING_IP:
+59c3659
059c3659 86990508f09b xchg bl,byte ptr [ecx-640FF7FBh]
```

Here is another example of the real hardware error shown in smaller font for visual clarity (note the concatenated error code for bugcheck 0×9C):

```
MACHINE_CHECK_EXCEPTION (9c)
A fatal Machine Check Exception has occurred.
KeBugCheckEx parameters;
    x86 Processors
        If the processor has ONLY MCE feature available (For example Intel
        Pentium), the parameters are:
        1 - Low  32 bits of P5_MC_TYPE MSR
        2 - Address of MCA_EXCEPTION structure
        3 - High 32 bits of P5_MC_ADDR MSR
        4 - Low  32 bits of P5_MC_ADDR MSR
        If the processor also has MCA feature available (For example Intel
        Pentium Pro), the parameters are:
        1 - Bank number
        2 - Address of MCA_EXCEPTION structure
        3 - High 32 bits of MCi_STATUS MSR for the MCA bank that had the error
        4 - Low  32 bits of MCi_STATUS MSR for the MCA bank that had the error
    IA64 Processors
        1 - Bugcheck Type
            1 - MCA_ASSERT
            2 - MCA_GET_STATEINFO
                SAL returned an error for SAL_GET_STATEINFO while processing MCA.
            3 - MCA_CLEAR_STATEINFO
                SAL returned an error for SAL_CLEAR_STATEINFO while processing MCA.
            4 - MCA_FATAL
                FW reported a fatal MCA.
            5 - MCA_NONFATAL
                SAL reported a recoverable MCA and we don't support currently
                support recovery or SAL generated an MCA and then couldn't
                produce an error record.
            0xB - INIT_ASSERT
            0xC - INIT_GET_STATEINFO
                  SAL returned an error for SAL_GET_STATEINFO while processing INIT
event.
```

```
            0xD - INIT_CLEAR_STATEINFO
                  SAL returned an error for SAL_CLEAR_STATEINFO while processing INIT
event.
            0xE - INIT_FATAL
                  Not used.
        2 - Address of log
        3 - Size of log
        4 - Error code in the case of x_GET_STATEINFO or x_CLEAR_STATEINFO
    AMD64 Processors
        1 - Bank number
        2 - Address of MCA_EXCEPTION structure
        3 - High 32 bits of MCi_STATUS MSR for the MCA bank that had the error
        4 - Low  32 bits of MCi_STATUS MSR for the MCA bank that had the error
Arguments:
Arg1: 00000000
Arg2: 808a07a0
Arg3: be000300
Arg4: 1008081f

Debugging Details:
------------------

   NOTE:  This is a hardware error.  This error was reported by the CPU
   via Interrupt 18.  This analysis will provide more information about
   the specific error.  Please contact the manufacturer for additional
   information about this error and troubleshooting assistance.

   This error is documented in the following publication:

      - IA-32 Intel(r) Architecture Software Developer's Manual
        Volume 3: System Programming Guide

   Bit Mask:

    MA                           Model Specific       MCA
 O  ID      Other Information      Error Code     Error Code
VV  SDP ___________|____________ _______|_______ _______|______
AEUECRC|                        |               |
LRCNVVC|                        |               |
^^^^^^^|                        |               |
   6         5         4         3         2         1
3210987654321098765432109876543210987654321098765432109876543210
----------------------------------------------------------------
1011111000000000000000110000000000010000000010000000100000011111

VAL   - MCi_STATUS register is valid
        Indicates that the information contained within the IA32_MCi_STATUS
        register is valid.  When this flag is set, the processor follows the
        rules given for the OVER flag in the IA32_MCi_STATUS register when
        overwriting previously valid entries.  The processor sets the VAL
        flag and software is responsible for clearing it.

UC    - Error Uncorrected
        Indicates that the processor did not or was not able to correct the
        error condition.  When clear, this flag indicates that the processor
        was able to correct the error condition.

EN    - Error Enabled
        Indicates that the error was enabled by the associated EEj bit of the
        IA32_MCi_CTL register.
```

```
MISCV - IA32_MCi_MISC Register Valid
        Indicates that the IA32_MCi_MISC register contains additional
        information regarding the error.  When clear, this flag indicates
        that the IA32_MCi_MISC register is either not implemented or does
        not contain additional information regarding the error.

ADDRV - IA32_MCi_ADDR register valid
        Indicates that the IA32_MCi_ADDR register contains the address where
        the error occurred.

PCC   - Processor Context Corrupt
        Indicates that the state of the processor might have been corrupted
        by the error condition detected and that reliable restarting of the
        processor may not be possible.

BUSCONNERR - Bus and Interconnect Error   BUS{LL}_{PP}_{RRRR}_{II}_{T}_err
        These errors match the format 0000 1PPT RRRR IILL

   Concatenated Error Code:
   --------------------------
   _VAL_UC_EN_MISCV_ADDRV_PCC_BUSCONNERR_1F

   This error code can be reported back to the manufacturer.
   They may be able to provide additional information based upon
   this error.  All questions regarding STOP 0x9C should be
   directed to the hardware manufacturer.

BUGCHECK_STR:  0x9C_IA32_GenuineIntel

DEFAULT_BUCKET_ID:  DRIVER_FAULT

PROCESS_NAME:  Idle

CURRENT_IRQL:  2

LAST_CONTROL_TRANSFER:  from 80a7fbd8 to 8087b6be

STACK_TEXT:
f773d280 80a7fbd8 0000009c 00000000 f773d2b0 nt!KeBugCheckEx+0x1b
f773d3b4 80a7786f f7737fe0 00000000 00000000 hal!HalpMcaExceptionHandler+0x11e
f773d3b4 f75a9ca2 f7737fe0 00000000 00000000 hal!HalpMcaExceptionHandlerWrapper+0x77
f78c6d50 8083abf2 00000000 0000000e 00000000 intelppm!AcpiC1Idle+0x12
f78c6d54 00000000 0000000e 00000000 00000000 nt!KiIdleLoop+0xa
```

HANDLE LIMIT (GDI)

Among various memory leaks leading to **Insufficient Memory** pattern (Volume 1, page 302) there is so called session pool leak briefly touched in the pattern about kernel pool leaks (Volume 1, page 440). It also involves GDI handles and structures allocated per user session that has the limit on how many of them can be created and this pattern should rather be called **Handle Limit**. Such leaks can result in poor visual application behavior after some time when drawing requests are not satisfied anymore. In severe cases, when the same bugs are present in a display driver, it can result in bugchecks like

BugCheck AB: SESSION_HAS_VALID_POOL_ON_EXIT

or, if a handle allocation request was not satisfied, it may result in a NULL pointer stored somewhere with the subsequent **Invalid Pointer** access (Volume 1, page 267):

SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (7e)

```
CONTEXT:  b791e010 -- (.cxr 0xffffffffb791e010)
eax=00000000 ebx=bc43d004 ecx=a233add8 edx=00000000 esi=bc430fff
edi=00000000
eip=bfe7d380 esp=b791e3dc ebp=b791e480 iopl=0 nv up ei pl zr na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010246
DisplayDriver+0×3e380:
bfe7d380 8a4702   mov     al,byte ptr [edi+2]        ds:0023:00000002=??
```

We can write 3 Win32 applications in Visual C++ that simulate GDI leaks. All of them create GDI objects in a loop and select them into their current graphics device context (DC) on Windows Server 2003 x64 SP2. Before running them we get the following session paged pool statistics (shown in smaller font for visual clarity):

```
lkd> !poolused c

   Sorting by Session Paged Pool Consumed

  Pool Used:
            NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 NV_x        0        0         5 14024704 UNKNOWN pooltag 'NV_x', please update
pooltag.txt
 BIG         0        0       257  3629056 Large session pool allocations
(ntos\ex\pool.c) , Binary: nt!mm
 NV          0        0       203  1347648 nVidia video driver
 Ttfd        0        0       233  1053152 TrueType Font driver
 Gh05        0        0       391  1050400 Gdi Handle manager specific object types:
defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Gla1        0        0       348   785088 Gdi handle manager specific object types
allocated from lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary:
```

```
win32k.sys
 Gcac        0        0        25   640880 Gdi glyph cache
 Gla5        0        0       631   323072 Gdi handle manager specific object types
allocated from lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary:
win32k.sys
 Gdrs        0        0        33   172288 Gdi GDITAG_DRVSUP
 Gla:        0        0       212   139072 Gdi handle manager specific object types
allocated from lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary:
win32k.sys
 Gla4        0        0       487   116880 Gdi handle manager specific object types
allocated from lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary:
win32k.sys
 Usti        0        0       148    97088 THREADINFO , Binary:
win32k!AllocateW32Thread
 Gla8        0        0       383    91920 Gdi handle manager specific object types
allocated from lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary:
win32k.sys
 Gla@        0        0       339    70512 Gdi handle manager specific object types
allocated from lookaside memory: defined in w32\ntgdi\inc\ntgdistr.h , Binary:
win32k.sys
 Gbaf        0        0        48    67584 UNKNOWN pooltag ‘Gbaf’, please update
pooltag.txt
 knlf        0        0        20    66496 UNKNOWN pooltag ‘knlf’, please update
pooltag.txt
 GDev        0        0         7    57344 Gdi pdev
 Usqu        0        0       152    53504 Q , Binary: win32k!InitQEntryLookaside
 Uscu        0        0       334    53440 CURSOR , Binary:
win32k!_CreateEmptyCursorObject
 Bmfd        0        0        21    50224 Font related stuff
 Uspi        0        0       153    40000 PROCESSINFO , Binary: win32k!MapDesktop
 Gfnt        0        0        47    39856 UNKNOWN pooltag ‘Gfnt’, please update
pooltag.txt
 Ggb         0        0        34    39088 Gdi glyph bits
 Gh08        0        0        33    38656 Gdi Handle manager specific object types:
defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Ghab        0        0       228    32832 Gdi Handle manager specific object types:
defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Ovfl        0        0         1    32768 The internal pool tag table has overflowed
- usually this is a result of nontagged allocations being made
 Gpff        0        0        88    27712 Gdi physical font file
 Gpfe        0        0        88    27600 UNKNOWN pooltag ‘Gpfe’, please update
pooltag.txt
 thdd        0        0         1    20480 DirectDraw/3D handle manager table
 Gebr        0        0        17    19776 Gdi ENGBRUSH
 Gh0@        0        0        86    19264 Gdi Handle manager specific object types:
defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Gsp         0        0        79    18960 Gdi sprite
 HT40        0        0         2    16384 UNKNOWN pooltag ‘HT40’, please update
pooltag.txt
 Gpat        0        0         4    16192 UNKNOWN pooltag ‘Gpat’, please update
pooltag.txt
 Ggls        0        0       169    12944 Gdi glyphset
 Glnk        0        0       371    11872 Gdi PFELINK
 Gldv        0        0         9    11248 Gdi Ldev
 Gffv        0        0        84     9408 Gdi FONTFILEVIEW
 Gfsb        0        0         1     8192 Gdi font sustitution list
 Uskt        0        0         2     7824 KBDTABLE , Binary: win32k!ReadLayoutFile
 Gh04        0        0         7     5856 Gdi Handle manager specific object types:
defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Gdcf        0        0        51     5712 UNKNOWN pooltag ‘Gdcf’, please update
pooltag.txt
```

```
 Gh0<        0        0        88     5632 Gdi Handle manager specific object types:
defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Gglb        0        0         1     4096 Gdi temp buffer
 Ustm        0        0        30     3360 TIMER , Binary: win32k!InternalSetTimer
 Gspm        0        0        39     3120 UNKNOWN pooltag ‘Gspm’, please update
pooltag.txt
 Usac        0        0        16     3056 ACCEL , Binary:
win32k!_CreateAcceleratorTable
 Usqm        0        0        25     2800 QMSG , Binary: win32k!InitQEntryLookaside
 Ghas        0        0         3     2592 Gdi Handle manager specific object types:
defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Uscl        0        0        20     2128 CLASS , Binary: win32k!ClassAlloc
 Uswl        0        0         1     2032 WINDOWLIST , Binary: win32k!BuildHwndList
 Gmul        0        0        19     1520 UNKNOWN pooltag ‘Gmul’, please update
pooltag.txt
 Dddp        0        0         8     1472 UNKNOWN pooltag ‘Dddp’, please update
pooltag.txt
 Ggdv        0        0         8     1472 Gdi GDITAG_GDEVICE
 UsDI        0        0         4     1408 DEVICEINFO , Binary:
win32k!CreateDeviceInfo
 Vtfd        0        0         4     1312 Font file/context
 Ushk        0        0        20     1280 HOTKEY , Binary: win32k!_RegisterHotKey
 Gspr        0        0         3     1264 Gdi sprite grow range
 Gtmw        0        0        13     1248 Gdi TMW_INTERNAL
 Gxlt        0        0         8     1152 Gdi Xlate
 Gpft        0        0         2      944 Gdi font table
 Uspp        0        0         5      944 PNP , Binary:
win32k!AllocateAndLinkHidTLCInf
 Ussm        0        0         7      896 SMS , Binary: win32k!InitSMSLookaside
 Gdbr        0        0        10      800 Gdi driver brush realization
 Usdc        0        0         8      768 DCE , Binary: win32k!CreateCacheDC
 Usct        0        0        12      768 CHECKPT , Binary: win32k!CkptRestore
 Usim        0        0         2      736 IME , Binary: win32k!CreateInputContext
 Usci        0        0         3      720 CLIENTTHREADINFO , Binary:
win32k!InitSystemThread
 Gh09        0        0         1      640 Gdi Handle manager specific object types:
defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Ussy        1       80         4      608 SYSTEM , Binary: win32k!xxxDesktopThread
 Urdr        0        0         9      576 REDIRECT , Binary:
win32k!SetRedirectionBitmap
 Uswd        0        0         2      576 WINDOW , Binary: win32k!xxxCreateWindowEx
 Uscb        0        0         3      544 CLIPBOARD , Binary:
win32k!_ConvertMemHandle
 Gcsl        0        0         1      496 Gdi string resource script names
 Ustx        0        0        10      496 TEXT , Binary:
win32k!NtUserDrawCaptionTemp
 Ussw        0        0         1      496 SWP , Binary: win32k!_BeginDeferWindowPos
 Gdev        0        0         2      480 Gdi GDITAG_DEVMODE
 Usih        0        0        10      480 IMEHOTKEY , Binary: win32k!SetImeHotKey
 Gdrv        0        0         1      368 UNKNOWN pooltag ‘Gdrv’, please update
pooltag.txt
 GVdv        0        0         1      320 UNKNOWN pooltag ‘GVdv’, please update
pooltag.txt
 Gmap        0        0         1      320 Gdi font map signature table
 Uskb        0        0         2      288 KBDLAYOUT , Binary:
win32k!xxxLoadKeyboardLayoutEx
 Uskf        0        0         2      288 KBDFILE , Binary:
win32k!LoadKeyboardLayoutFile
 Uswe        0        0         2      224 WINEVENT , Binary: win32k!_SetWinEventHook
 Gddf        0        0         2      224 Gdi ddraw driver heaps
 Gddv        0        0         2      192 Gdi ddraw driver video memory list
 GFil        0        0          2      192 Gdi engine descriptor list
 Gdwd        0        0          2       96 Gdi watchdog support objects , Binary:
win32k.sys
 Usd9        0        0          1       80 DDE9 , Binary: win32k!xxxCsDdeInitialize
 Gvds        0        0          1       64 UNKNOWN pooltag 'Gvds', please update
pooltag.txt
 GreA        0        0          1       64 UNKNOWN pooltag 'GreA', please update
pooltag.txt
 Usse        0        0          1       48 SECURITY , Binary:
win32k!SetDisconnectDesktopSecu
 Usvl        0        0          1       48 VWPL , Binary: win32k!VWPLAdd
 Mdxg        1      112          0        0 UNKNOWN pooltag 'Mdxg', please update
pooltag.txt
 Gini        3      128          0        0 Gdi fast mutex
 Usev        1       64          0        0 EVENT , Binary:
win32k!xxxPollAndWaitForSingleO
 Gdde        3      240          0        0 Gdi ddraw event
 TOTAL           9      624      6256 24408704
```

The first application leaks fonts:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
 int wmId, wmEvent;
 PAINTSTRUCT ps;
 HDC hdc;

 switch (message)
 {
   case WM_PAINT:
     hdc = BeginPaint(hWnd, &ps);
     while (true)
     {
        HFONT hf = CreateFont(10, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
L"Arial");
        SelectObject(ps.hdc, hf);
     }
     EndPaint(hWnd, &ps);
     break;
```

We clearly see the leak as the greatly increased the number of allocations for ”Gla:” pool tag:

```
  Pool Used:
            NonPaged             Paged
 Tag    Allocs     Used    Allocs     Used
 NV_x        0        0         5 14024704 UNKNOWN pooltag 'NV_x', please
update pooltag.txt
 Gla:        0        0     10194  6687264 Gdi handle manager specific
object types allocated from lookaside memory: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 BIG         0        0       248  3690496 Large session pool allocations
(ntos\ex\pool.c) , Binary: nt!mm
 NV          0        0       203  1347648 nVidia video driver
 Gh05        0        0       396  1057888 Gdi Handle manager specific
object types: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Ttfd        0        0       226  1043264 TrueType Font driver
```

The second application leaks fonts and pens:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
 int wmId, wmEvent;
 PAINTSTRUCT ps;
 HDC hdc;

 switch (message)
 {
   case WM_PAINT:
     hdc = BeginPaint(hWnd, &ps);
     while (true)
     {
       HFONT hf = CreateFont(10, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
L"Arial");
       HPEN hp = CreatePen(0, 10, RGB(10, 20, 30));
       SelectObject(ps.hdc, hf);
       SelectObject(ps.hdc, hp);
     }
     EndPaint(hWnd, &ps);
     break;
```

We see that roughly the same number of allocations is split between ”Gla:” and “Gh0@” pool tags:

```
  Pool Used:
            NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 NV_x        0        0         5 14024704 UNKNOWN pooltag 'NV_x', please
update pooltag.txt
 BIG         0        0       262  3874816 Large session pool allocations
(ntos\ex\pool.c) , Binary: nt!mm
 Gla:        0        0      5203  3413168 Gdi handle manager specific
object types allocated from lookaside memory: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 NV          0        0       203  1347648 nVidia video driver
 Gh0@        0        0      5077  1137248 Gdi Handle manager specific
object types: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Ttfd        0        0       233  1053152 TrueType Font driver
```

The third program leaks fonts, pens and brushes:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
 int wmId, wmEvent;
 PAINTSTRUCT ps;
 HDC hdc;

 switch (message)
 {
   case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);
   while (true)
   {
     HFONT hf = CreateFont(10, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
L"Arial");
     HPEN hp = CreatePen(0, 10, RGB(10, 20, 30));
     HBRUSH hb = CreateSolidBrush(RGB(10, 20, 30));
     SelectObject(ps.hdc, hf);
     SelectObject(ps.hdc, hp);
     SelectObject(ps.hdc, hb);
   }
   EndPaint(hWnd, &ps);
   break;
```

Now we see that the same number of allocations is almost equally split between ”Gla:”, “Gh0@” and “Gla@” pool tags:

```
  Pool Used:
            NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 NV_x        0        0         5 14024704 UNKNOWN pooltag 'NV_x', please
update pooltag.txt
 BIG         0        0       262  3874816 Large session pool allocations
(ntos\ex\pool.c) , Binary: nt!mm
 Gla:        0        0      3539  2321584 Gdi handle manager specific
object types allocated from lookaside memory: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 NV          0        0       203  1347648 nVidia video driver
 Ttfd        0        0       233  1053152 TrueType Font driver
 Gh05        0        0       392  1052768 Gdi Handle manager specific
object types: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Gla1        0        0       353   796368 Gdi handle manager specific
object types allocated from lookaside memory: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Gh0@        0        0      3414   764736 Gdi Handle manager specific
object types: defined in w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Gla@        0        0      3665   762320 Gdi handle manager specific
object types allocated from lookaside memory: defined in
w32\ntgdi\inc\ntgdistr.h , Binary: win32k.sys
 Gcac        0        0        25   640880 Gdi glyph cache
```

When the certain amount of handles is reached all subsequent GDI Create calls fail and other applications start showing various visual defects. Print screen operation also fails with insufficient memory message.

MISSING COMPONENT

Sometimes the code raises an exception when certain DLL is missing. We need to guess that component name if we don't have symbols and source code. This can be done by inspecting raw stack data in the close proximity of the exception ESP/RSP values.

Consider the crash dump of App.exe with the following incomplete unmanaged stack trace (shown in smaller font for visual clarity):

```
EXCEPTION_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 76f442eb (kernel32!RaiseException+0x00000058)
   ExceptionCode: c06d007f
  ExceptionFlags: 00000000
NumberParameters: 1
   Parameter[0]: 0024f21c
```

```
0:000> kL
ChildEBP RetAddr
0024f1f8 6eb1081e kernel32!RaiseException+0x58
WARNING: Stack unwind information not available. Following frames may be wrong.
0024f260 6eac62fb AppNativeLib!AppLibraryExports::InteropNotifyUnAdvise+0x6aa9
0024f2ac 6ea9e269 AppNativeLib!AppLibraryExports::Phase2Initialization+0x24c9
0024f32c 79e74d79 AppNativeLib!AppLibraryExports::QueryDatabase+0x99da
0024f3d4 664bd6af mscorwks!MethodTable::IsValueType+0x35
0024f3e8 319cec9e AppShell_ni+0x2d6af
0024f3f4 31a15d19 UIX_ni+0x1ec9e
0024f3f8 00000000 UIX_ni+0x65d19
```

We can try to interpret the crash as **Managed Code Exception** (Volume 1, page 331) but let's first to check the exception code. Google search shows that the error code c06d007f means "DelayLoad Export Missing" and this definitely has to do with some missing DLL. It is not possible to tell which one was missing from the stack trace output. Additional digging is required.

Let's look at the raw stack. First, we can try to see whether there are any calls to LoadLibrary on thread raw stack data:

```
0:000> !teb
TEB at 7ffdf000
    ExceptionList:        0024f8c4
    StackBase:            00250000
    StackLimit:           00249000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffdf000
    EnvironmentPointer:   00000000
    ClientId:             000012f4 . 00001080
    RpcHandle:            00000000
    Tls Storage:          004e8a18
    PEB Address:          7ffde000
    LastErrorValue:       126
    LastStatusValue:      c0000135
    Count Owned Locks:    0
    HardErrorMode:        0

0:000> dds 00249000 00250000
00249000 00000000
00249004 00000000
...
0024f1a0 00000000
0024f1a4 00000000
0024f1a8 c06d007f
0024f1ac 00000000
0024f1b0 00000000
0024f1b4 76f442eb kernel32!RaiseException+0x58
0024f1b8 00000001
0024f1bc 0024f21c
0024f1c0 00000000
0024f1c4 00000000
0024f1c8 00000000
0024f1cc 00000000
0024f1d0 76f00000 kernel32!_imp___aullrem (kernel32+0x0)
0024f1d4 f7bd2a5d
0024f1d8 0024f1e8
0024f1dc 76fb8e8f kernel32!LookupHandler+0x10
0024f1e0 6ecb9b40 AppNativeLib!ShutdownSingletonMgr+0x15b024
0024f1e4 0024f21c
0024f1e8 0024f200
0024f1ec 6ec74e2a AppNativeLib!ShutdownSingletonMgr+0x11630e
0024f1f0 6ecb9b40 AppNativeLib!ShutdownSingletonMgr+0x15b024
0024f1f4 6ecb9ff0 AppNativeLib!ShutdownSingletonMgr+0x15b4d4
0024f1f8 0024f260
0024f1fc 6eb1081e
AppNativeLib!AppLibraryExports::InteropNotifyUnAdvise+0x6aa9
0024f200 c06d007f
0024f204 00000000
0024f208 00000001
...
```

There are no such calls in our crash dump. Then we can try to interpret raw stack data as a byte stream to see “.dll” strings:

```
0:000> db 00249000 00250000
00249000 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
00249010 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
00249020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
00249030 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
...
```

There are no such strings except “user32.dll”.

Now we can try to interpret every double word as a pointer to a Unicode string:

```
0:000> dpu 00249000 00250000
...
```

There are no strings with “.dll” inside. Finally, if we try to interpret every double word as a pointer to an ASCII string we get a few references to “AppService.dll”:

```
0:000> dpa 00249000 00250000
...
0024f1d0 76f00000 "MZ."
0024f1d4 f7bd2a5d
0024f1d8 0024f1e8 ""
0024f1dc 76fb8e8f "..t-.E."
0024f1e0 6ecb9b40 “AppService.dll”
0024f1e4 0024f21c “$”
0024f1e8 0024f200 “.”
0024f1ec 6ec74e2a “..^.._]..”
0024f1f0 6ecb9b40 “AppService.dll”
0024f1f4 6ecb9ff0 “CreateServiceInstance”
0024f1f8 0024f260 “..$”
0024f1fc 6eb1081e “.]…….e.”
0024f200 c06d007f
0024f204 00000000
0024f208 00000001
0024f20c 0024f268 “..$”
0024f210 00000000
0024f214 0024f2c8 “…n ..n<.$”
0024f218 6ecbe220 “”
0024f21c 00000024
0024f220 6ecb9960 “.”
0024f224 6ecbe05c “.c.n.2.n”
0024f228 6ecb9b40 “AppService.dll”
0024f22c 00000001
0024f230 6ecb9ff0 “CreateServiceInstance”
0024f234 ffffffff
0024f238 00000000
```

If we search for 0024f1e0 pointer in **dps** WinDbg command output we would see that it is in the close proximity to RaiseException call and it seems that all our pointers to “AppService.dll” string fall into AppNativeLib address range:

```
0024f1b4 76f442eb kernel32!RaiseException+0x58
0024f1b8 00000001
0024f1bc 0024f21c
0024f1c0 00000000
0024f1c4 00000000
0024f1c8 00000000
0024f1cc 00000000
0024f1d0 76f00000 kernel32!_imp___aullrem (kernel32+0x0)
0024f1d4 f7bd2a5d
0024f1d8 0024f1e8
0024f1dc 76fb8e8f kernel32!LookupHandler+0x10
0024f1e0 6ecb9b40 AppNativeLib!ShutdownSingletonMgr+0×15b024
0024f1e4 0024f21c
0024f1e8 0024f200
0024f1ec 6ec74e2a AppNativeLib!ShutdownSingletonMgr+0×11630e
0024f1f0 6ecb9b40 AppNativeLib!ShutdownSingletonMgr+0×15b024
0024f1f4 6ecb9ff0 AppNativeLib!ShutdownSingletonMgr+0×15b4d4
0024f1f8 0024f260
0024f1fc 6eb1081e
AppNativeLib!AppLibraryExports::InteropNotifyUnAdvise+0×6aa9
0024f200 c06d007f
0024f204 00000000
0024f208 00000001
0024f20c 0024f268
0024f210 00000000
0024f214 0024f2c8
0024f218 6ecbe220 AppNativeLib!ShutdownSingletonMgr+0×15f704
0024f21c 00000024
0024f220 6ecb9960 AppNativeLib!ShutdownSingletonMgr+0×15ae44
0024f224 6ecbe05c AppNativeLib!ShutdownSingletonMgr+0×15f540
0024f228 6ecb9b40 AppNativeLib!ShutdownSingletonMgr+0×15b024
0024f22c 00000001
0024f230 6ecb9ff0 AppNativeLib!ShutdownSingletonMgr+0×15b4d4
0024f234 ffffffff
0024f238 00000000
```

When examining the system it was found that AppService.dll was missing there indeed.

NULL POINTER (CODE)

This is a specialization of **Invalid Pointer** pattern (Volume 1, page 267) and it is the most easily recognized pattern with a straightforward fix most of the time according to my experience. Checking the pointer value to be non-NULL might not work if the pointer value is random (**Wild Pointer** pattern, page **Error! Bookmark not defined.**) but at least it eliminates this class of problems. NULL pointers can be NULL data pointers or NULL code pointers. The latter happens when we have a pointer to some function and we try to call it. Consider this example:

```
0:002> r
eax=00000000 ebx=00000000 ecx=93630000 edx=00000000 esi=00000000
edi=00000000
eip=00000000 esp=0222ffbc ebp=0222ffec iopl=0  nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
00000000 ??              ???

0:002> kv
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
0222ffb8 7d4dfe21 00000000 00000000 00000000 0×0
0222ffec 00000000 00000000 00000000 00000000 kernel32!BaseThreadStart+0×34
```

Clearly we have a NULL code pointer here and if we disassemble backwards the return address 7d4dfe21 or BaseThreadStart+0×34 address we would suspect that BaseThreadStart function tried to call a thread start procedure:

```
0:002> ub 7d4dfe21
kernel32!BaseThreadStart+0x10:
7d4dfdfd mov     eax,dword ptr fs:[00000018h]
7d4dfe03 cmp     dword ptr [eax+10h],1E00h
7d4dfe0a jne     kernel32!BaseThreadStart+0x2e (7d4dfe1b)
7d4dfe0c cmp     byte ptr [kernel32!BaseRunningInServerProcess
(7d560008)],0
7d4dfe13 jne     kernel32!BaseThreadStart+0x2e (7d4dfe1b)
7d4dfe15 call    dword ptr [kernel32!_imp__CsrNewThread (7d4d0310)]
7d4dfe1b push    dword ptr [ebp+0Ch]
7d4dfe1e call    dword ptr [ebp+8]

0:002> dp ebp+8 l1
0222fff4  00000000
```

To confirm this suspicion we can write a code that calls CreateThread function similar to this one:

```
typedef DWORD (WINAPI *THREADPROC)(PVOID);

DWORD WINAPI ThreadProc(PVOID pvParam)
{
  // Does some work
  return 0;
}

void foo()
{
  //..
  THREADPROC thProc = ThreadProc;
  //..
  // thProc becomes NULL because of a bug
  //..
  HANDLE Thread = CreateThread(NULL, 0, thProc, 0, 0, NULL);
  CloseHandle(hThread);
}
```

EXECUTION RESIDUE

For the pattern about NULL code pointer (page 237) I created a simple program that crashes when we pass a NULL thread procedure pointer to CreateThread function. We might expect to see little in the raw stack data (Volume 1, page 231) because there was no user-supplied thread code. In reality, if we dump it we would see lots of symbolic information for code and data including ASCII and UNICODE fragments that I call **Execution Residue** patterns and one of them is **Exception Handling Residue** we can use to check for hidden exceptions (Volume 1, page 271) and differentiate between 1st and 2nd chance exceptions (Volume 1, page 109). Code residues are very powerful in reconstructing stack traces manually (Volume 1, page 157) or looking for partial stack traces and historical information (Volume 1, page 457).

To show typical execution residues I created another small program with two additional threads based on Visual Studio Win32 project. After we dismiss About box we create the first thread and then we crash the process when creating the second thread because of the NULL thread procedure:

```
typedef DWORD (WINAPI *THREADPROC)(PVOID);

DWORD WINAPI ThreadProc(PVOID pvParam)
{
   for (unsigned int i = 0xFFFFFFFF; i; --i);
   return 0;
}

// Message handler for about box.
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM
lParam)
{
   UNREFERENCED_PARAMETER(lParam);
   switch (message)
   {
   case WM_INITDIALOG:
      return (INT_PTR)TRUE;

   case WM_COMMAND:
      if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
      {
         EndDialog(hDlg, LOWORD(wParam));
         THREADPROC thProc = ThreadProc;
         HANDLE hThread = CreateThread(NULL, 0, ThreadProc, 0, 0, NULL);
         CloseHandle(hThread);
         Sleep(1000);
         hThread = CreateThread(NULL, 0, NULL, 0, 0, NULL);
         CloseHandle(hThread);
         return (INT_PTR)TRUE;
      }
      break;
   }
   return (INT_PTR)FALSE;
}
```

When we open the crash dump we see these threads:

```
0:002> ~*kL

   0  Id: cb0.9ac Suspend: 1 Teb: 7efdd000 Unfrozen
ChildEBP RetAddr
0012fdf4 00411554 user32!NtUserGetMessage+0x15
0012ff08 00412329 NullThread!wWinMain+0xa4
0012ffb8 0041208d NullThread!__tmainCRTStartup+0x289
0012ffc0 7d4e7d2a NullThread!wWinMainCRTStartup+0xd
0012fff0 00000000 kernel32!BaseProcessStart+0x28

   1  Id: cb0.8b4 Suspend: 1 Teb: 7efda000 Unfrozen
ChildEBP RetAddr
01eafea4 7d63f501 ntdll!NtWaitForMultipleObjects+0x15
01eaff48 7d63f988 ntdll!EtwpWaitForMultipleObjectsEx+0xf7
01eaffb8 7d4dfe21 ntdll!EtwpEventPump+0x27f
01eaffec 00000000 kernel32!BaseThreadStart+0x34
```

```
   2  Id: cb0.ca8 Suspend: 1 Teb: 7efd7000 Unfrozen
ChildEBP RetAddr
0222ffb8 7d4dfe21 NullThread!ThreadProc+0×34
0222ffec 00000000 kernel32!BaseThreadStart+0×34

#  3  Id: cb0.5bc Suspend: 1 Teb: 7efaf000 Unfrozen
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0236ffb8 7d4dfe21 0×0
0236ffec 00000000 kernel32!BaseThreadStart+0×34

   4  Id: cb0.468 Suspend: -1 Teb: 7efac000 Unfrozen
ChildEBP RetAddr
01f7ffb4 7d674807 ntdll!NtTerminateThread+0x12
01f7ffc4 7d66509f ntdll!RtlExitUserThread+0x26
01f7fff4 00000000 ntdll!DbgUiRemoteBreakin+0x41
```

We see our first created thread looping:

```
0:003> ~2s
eax=cbcf04b5 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000
edi=0222ffb8
eip=00411aa4 esp=0222fee0 ebp=0222ffb8 iopl=0 nv up ei ng nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000282
NullThread!ThreadProc+0x34:
00411aa4 7402   je      NullThread!ThreadProc+0x38 (00411aa8)   [br=0]

0:002> u
NullThread!ThreadProc+0x34:
00411aa4 je      NullThread!ThreadProc+0x38 (00411aa8)
00411aa6 jmp     NullThread!ThreadProc+0x27 (00411a97)
00411aa8 xor     eax,eax
00411aaa pop     edi
00411aab pop     esi
00411aac pop     ebx
00411aad mov     esp,ebp
00411aaf pop     ebp
```

We might expect it having very little in its raw stack data but what we see when we dump its stack range from **!teb** command is **Thread Startup Residue** where some symbolic information might be coincidental too (Volume 1, page 390):

```
0:002> dds 0222f000  02230000
0222f000  00000000
0222f004  00000000
0222f008  00000000
...
0222f104  00000000
0222f108  00000000
0222f10c  00000000
0222f110  7d621954 ntdll!RtlImageNtHeaderEx+0xee
```

```
0222f114  7efde000
0222f118  00000000
0222f11c  00000001
0222f120  000000e8
0222f124  004000e8 NullThread!_enc$textbss$begin <PERF> (NullThread+0xe8)
0222f128  00000000
0222f12c  0222f114
0222f130  00000000
0222f134  0222fca0
0222f138  7d61f1f8 ntdll!_except_handler3
0222f13c  7d621958 ntdll!RtlpRunTable+0x4a0
0222f140  ffffffff
0222f144  7d621954 ntdll!RtlImageNtHeaderEx+0xee
0222f148  7d6218ab ntdll!RtlImageNtHeader+0x1b
0222f14c  00000001
0222f150  00400000 NullThread!_enc$textbss$begin <PERF> (NullThread+0x0)
0222f154  00000000
0222f158  00000000
0222f15c  0222f160
0222f160  004000e8 NullThread!_enc$textbss$begin <PERF> (NullThread+0xe8)
0222f164  0222f7bc
0222f168  7d4dfea3 kernel32!ConsoleApp+0xe
0222f16c  00400000 NullThread!_enc$textbss$begin <PERF> (NullThread+0x0)
0222f170  7d4dfe77 kernel32!ConDllInitialize+0x1f5
0222f174  00000000
0222f178  7d4dfe8c kernel32!ConDllInitialize+0x20a
0222f17c  00000000
0222f180  00000000
...
0222f290  00000000
0222f294  0222f2b0
0222f298  7d6256e8 ntdll!bsearch+0x42
0222f29c  00180144
0222f2a0  0222f2b4
0222f2a4  7d625992 ntdll!ARRAY_FITS+0x29
0222f2a8  00000a8c
0222f2ac  00001f1c
0222f2b0  0222f2c0
0222f2b4  0222f2f4
0222f2b8  7d625944 ntdll!RtlpLocateActivationContextSection+0x1da
0222f2bc  00001f1c
0222f2c0  000029a8
...
0222f2e0  536cd652
0222f2e4  0222f334
0222f2e8  7d625b62 ntdll!RtlpFindUnicodeStringInSection+0x7b
0222f2ec  0222f418
0222f2f0  00000000
0222f2f4  0222f324
0222f2f8  7d6257f1 ntdll!RtlpFindNextActivationContextSection+0x64
0222f2fc  00181f1c
0222f300  c0150008
...
0222f320  7efd7000
```

```
0222f324  0222f344
0222f328  7d625cd2 ntdll!RtlFindNextActivationContextSection+0x46
0222f32c  0222f368
0222f330  0222f3a0
0222f334  0222f38c
0222f338  0222f340
0222f33c  00181f1c
0222f340  00000000
0222f344  0222f390
0222f348  7d625ad8 ntdll!RtlFindActivationContextSectionString+0xe1
0222f34c  0222f368
0222f350  0222f3a0
...
0222f38c  00000a8c
0222f390  0222f454
0222f394  7d626381 ntdll!CsrCaptureMessageMultiUnicodeStringsInPlace+0xa57
0222f398  00000003
0222f39c  00000000
0222f3a0  00181f1c
0222f3a4  0222f418
0222f3a8  0222f3b4
0222f3ac  7d6a0340 ntdll!LdrApiDefaultExtension
0222f3b0  7d6263df ntdll!CsrCaptureMessageMultiUnicodeStringsInPlace+0xb73
0222f3b4  00000040
0222f3b8  00000000
...
0222f420  00000000
0222f424  0222f458
0222f428  7d625f9a ntdll!CsrCaptureMessageMultiUnicodeStringsInPlace+0x4c1
0222f42c  00020000
0222f430  0222f44c
0222f434  0222f44c
0222f438  0222f44c
0222f43c  00000002
0222f440  00000002
0222f444  7d625f9a ntdll!CsrCaptureMessageMultiUnicodeStringsInPlace+0x4c1
0222f448  00020000
0222f44c  00000000
0222f450  00003cfb
0222f454  0222f5bc
0222f458  0222f4f4
0222f45c  0222f5bc
0222f460  7d626290 ntdll!RtlDosApplyFileIsolationRedirection_Ustr+0x346
0222f464  0222f490
0222f468  00000000
0222f46c  0222f69c
0222f470  7d6262f5 ntdll!RtlDosApplyFileIsolationRedirection_Ustr+0x3de
0222f474  0222f510
0222f478  7d6a0340 ntdll!LdrApiDefaultExtension
0222f47c  7d626290 ntdll!RtlDosApplyFileIsolationRedirection_Ustr+0x346
0222f480  00000000
0222f484  00800000
...
0222f544  00000000
```

```
0222f548  00000001
0222f54c  7d6a0290 ntdll!LdrpHashTable+0x50
0222f550  00000000
0222f554  00500000
...
0222f59c  00000000
0222f5a0  0222f5d4
0222f5a4  7d6251d0 ntdll!LdrUnlockLoaderLock+0x84
0222f5a8  7d6251d7 ntdll!LdrUnlockLoaderLock+0xad
0222f5ac  00000000
0222f5b0  0222f69c
0222f5b4  00000000
0222f5b8  00003cfb
0222f5bc  0222f5ac
0222f5c0  7d626de0 ntdll!LdrGetDllHandleEx+0xbe
0222f5c4  0222f640
0222f5c8  7d61f1f8 ntdll!_except_handler3
0222f5cc  7d6251e0 ntdll!`string'+0x74
0222f5d0  ffffffff
0222f5d4  7d6251d7 ntdll!LdrUnlockLoaderLock+0xad
0222f5d8  7d626fb3 ntdll!LdrGetDllHandleEx+0x368
0222f5dc  00000001
0222f5e0  0ca80042
0222f5e4  7d626f76 ntdll!LdrGetDllHandleEx+0x329
0222f5e8  00000000
0222f5ec  7d626d0b ntdll!LdrGetDllHandle
0222f5f0  00000002
0222f5f4  001a0018
...
0222f640  0222f6a8
0222f644  7d61f1f8 ntdll!_except_handler3
0222f648  7d626e60 ntdll!`string'+0xb4
0222f64c  ffffffff
0222f650  7d626f76 ntdll!LdrGetDllHandleEx+0x329
0222f654  7d626d23 ntdll!LdrGetDllHandle+0x18
0222f658  00000001
...
0222f66c  0222f6b8
0222f670  7d4dff0e kernel32!GetModuleHandleForUnicodeString+0x20
0222f674  00000001
0222f678  00000000
0222f67c  0222f6d4
0222f680  7d4dff1e kernel32!GetModuleHandleForUnicodeString+0x97
0222f684  00000000
0222f688  7efd7c00
0222f68c  00000002
0222f690  00000001
0222f694  00000000
0222f698  0222f6f0
0222f69c  7d4c0000 kernel32!_imp__NtFsControlFile <PERF> (kernel32+0x0)
0222f6a0  0222f684
0222f6a4  7efd7c00
0222f6a8  0222fb20
0222f6ac  7d4d89c4 kernel32!_except_handler3
```

```
0222f6b0  7d4dff28 kernel32!`string'+0x18
0222f6b4  ffffffff
0222f6b8  7d4dff1e kernel32!GetModuleHandleForUnicodeString+0x97
0222f6bc  7d4e001f kernel32!BasepGetModuleHandleExW+0x17f
0222f6c0  7d4e009f kernel32!BasepGetModuleHandleExW+0x23c
0222f6c4  00000000
0222f6c8  0222fc08
0222f6cc  00000001
0222f6d0  ffffffff
0222f6d4  001a0018
0222f6d8  7efd7c00
0222f6dc  0222fb50
0222f6e0  00000000
0222f6e4  00000000
0222f6e8  00000000
0222f6ec  02080000 oleaut32!_PictSaveEnhMetaFile+0x76
0222f6f0  0222f90c
0222f6f4  02080000 oleaut32!_PictSaveEnhMetaFile+0x76
0222f6f8  0222f704
0222f6fc  00000000
0222f700  7d4c0000 kernel32!_imp__NtFsControlFile <PERF> (kernel32+0x0)
0222f704  00000000
0222f708  02080000 oleaut32!_PictSaveEnhMetaFile+0x76
0222f70c  0222f928
0222f710  02080000 oleaut32!_PictSaveEnhMetaFile+0x76
0222f714  0222f720
0222f718  00000000
0222f71c  7d4c0000 kernel32!_imp__NtFsControlFile <PERF> (kernel32+0x0)
0222f720  00000000
0222f724  00000000
...
0222f7b8  0000f949
0222f7bc  0222fbf4
0222f7c0  7d4dfdd0 kernel32!_BaseDllInitialize+0x6b
0222f7c4  00000002
0222f7c8  00000000
0222f7cc  00000000
0222f7d0  7d4dfde4 kernel32!_BaseDllInitialize+0x495
0222f7d4  00000000
0222f7d8  7efde000
0222f7dc  7d4c0000 kernel32!_imp__NtFsControlFile <PERF> (kernel32+0x0)
0222f7e0  00000000
0222f7e4  00000000
...
0222f894  01c58ae0
0222f898  0222fac0
0222f89c  7d62155b ntdll!RtlAllocateHeap+0x460
0222f8a0  7d61f78c ntdll!RtlAllocateHeap+0xee7
0222f8a4  00000000
0222f8a8  0222fc08
...
0222f8d8  00000000
0222f8dc  7d621954 ntdll!RtlImageNtHeaderEx+0xee
0222f8e0  0222f9a4
```

```
0222f8e4  7d614c88 ntdll!$$VProc_ImageExportDirectory+0x2c48
0222f8e8  0222f9a6
0222f8ec  7d612040 ntdll!$$VProc_ImageExportDirectory
0222f8f0  00000221
0222f8f4  0222f944
0222f8f8  7d627405 ntdll!LdrpSnapThunk+0xc0
0222f8fc  0222f9a6
0222f900  00000584
0222f904  7d600000 ntdll!RtlDosPathSeperatorsString <PERF> (ntdll+0x0)
0222f908  7d613678 ntdll!$$VProc_ImageExportDirectory+0x1638
0222f90c  7d614c88 ntdll!$$VProc_ImageExportDirectory+0x2c48
0222f910  0222f9a4
0222f914  00000001
0222f918  0222f9a4
0222f91c  00000000
0222f920  0222f990
0222f924  7d6000f0 ntdll!RtlDosPathSeperatorsString <PERF> (ntdll+0xf0)
0222f928  0222f968
0222f92c  00000001
0222f930  0222f9a4
0222f934  7d6000f0 ntdll!RtlDosPathSeperatorsString <PERF> (ntdll+0xf0)
0222f938  0222f954
0222f93c  00000000
0222f940  00000000
0222f944  0222fa00
0222f948  7d62757a ntdll!LdrpGetProcedureAddress+0x189
0222f94c  0222f95c
0222f950  00000098
0222f954  00000005
0222f958  01c44f48
0222f95c  0222fb84
0222f960  7d62155b ntdll!RtlAllocateHeap+0x460
0222f964  7d61f78c ntdll!RtlAllocateHeap+0xee7
0222f968  00000000
0222f96c  0000008c
0222f970  00000000
0222f974  7d4d8472 kernel32!$$VProc_ImageExportDirectory+0x6d4e
0222f978  0222fa1c
0222f97c  7d627607 ntdll!LdrpGetProcedureAddress+0x274
0222f980  7d612040 ntdll!$$VProc_ImageExportDirectory
0222f984  002324f8
0222f988  7d600000 ntdll!RtlDosPathSeperatorsString <PERF> (ntdll+0x0)
0222f98c  0222faa8
0222f990  0000a7bb
0222f994  00221f08
0222f998  0222f9a4
0222f99c  7d627c2e ntdll!RtlDecodePointer
0222f9a0  00000000
0222f9a4  74520000
0222f9a8  6365446c
0222f9ac  5065646f
0222f9b0  746e696f
0222f9b4  00007265
0222f9b8  7d627c2e ntdll!RtlDecodePointer
```

```
0222f9bc  00000000
...
0222f9f8  01c40640
0222f9fc  00000000
0222fa00  7d6275b2 ntdll!LdrpGetProcedureAddress+0xb3
0222fa04  7d627772 ntdll!LdrpSnapThunk+0x31c
0222fa08  7d600000 ntdll!RtlDosPathSeperatorsString <PERF> (ntdll+0x0)
0222fa0c  0222fa44
0222fa10  00000000
0222fa14  0222faa8
0222fa18  00000000
0222fa1c  0222fab0
0222fa20  00000001
0222fa24  00000001
0222fa28  00000000
0222fa2c  0222fa9c
0222fa30  7d4c00e8 kernel32!_imp__NtFsControlFile <PERF> (kernel32+0xe8)
0222fa34  01c44fe0
0222fa38  00000001
0222fa3c  01c401a0
0222fa40  7d4c00e8 kernel32!_imp__NtFsControlFile <PERF> (kernel32+0xe8)
0222fa44  00110010
0222fa48  7d4d8478 kernel32!$$VProc_ImageExportDirectory+0x6d54
0222fa4c  00000000
0222fa50  0222fb0c
0222fa54  7d62757a ntdll!LdrpGetProcedureAddress+0x189
0222fa58  7d600000 ntdll!RtlDosPathSeperatorsString <PERF> (ntdll+0x0)
0222fa5c  00000000
0222fa60  0022faa8
0222fa64  0222fab0
0222fa68  0222fb0c
0222fa6c  7d627607 ntdll!LdrpGetProcedureAddress+0x274
0222fa70  7d6a0180 ntdll!LdrpLoaderLock
0222fa74  7d6275b2 ntdll!LdrpGetProcedureAddress+0xb3
0222fa78  102ce1ac msvcr80d!`string'
0222fa7c  0222fc08
0222fa80  0000ffff
0222fa84  0022f8b0
0222fa88  0022f8a0
0222fa8c  00000003
0222fa90  0222fbd4
0222fa94  020215fc oleaut32!DllMain+0x2c
0222fa98  02020000 oleaut32!_imp__RegFlushKey <PERF> (oleaut32+0x0)
0222fa9c  00000002
0222faa0  00000000
0222faa4  00000000
0222faa8  00000002
0222faac  0202162d oleaut32!DllMain+0x203
0222fab0  65440000
0222fab4  02020000 oleaut32!_imp__RegFlushKey <PERF> (oleaut32+0x0)
0222fab8  00000001
0222fabc  00726574
0222fac0  0222facc
0222fac4  7d627c2e ntdll!RtlDecodePointer
```

```
0222fac8  00000000
0222facc  65440000
0222fad0  00000000
0222fad4  00000000
0222fad8  00726574
0222fadc  00000005
0222fae0  00000000
0222fae4  1021af95 msvcr80d!_heap_alloc_dbg+0x375
0222fae8  002322f0
0222faec  00000000
0222faf0  01c40238
0222faf4  0222fa78
0222faf8  7efd7bf8
0222fafc  00000020
0222fb00  7d61f1f8 ntdll!_except_handler3
0222fb04  7d6275b8 ntdll!`string'+0xc
0222fb08  ffffffff
0222fb0c  7d6275b2 ntdll!LdrpGetProcedureAddress+0xb3
0222fb10  00000000
0222fb14  00000000
0222fb18  0222fb48
0222fb1c  00000000
0222fb20  01000000
0222fb24  00000001
0222fb28  0222fb50
0222fb2c  7d4dac3a kernel32!GetProcAddress+0x44
0222fb30  0222fb50
0222fb34  7d4dac4c kernel32!GetProcAddress+0x5c
0222fb38  0222fc08
0222fb3c  00000013
0222fb40  00000000
0222fb44  01c44f40
0222fb48  01c4015c
0222fb4c  00000098
0222fb50  01c44f40
0222fb54  01c44f48
0222fb58  01c40238
0222fb5c  10204f9f msvcr80d!_initptd+0x10f
0222fb60  00000098
0222fb64  00000000
0222fb68  01c40000
0222fb6c  0222f968
0222fb70  7d4c0000 kernel32!_imp__NtFsControlFile <PERF> (kernel32+0x0)
0222fb74  00000ca8
0222fb78  4b405064 msctf!g_timlist
0222fb7c  0222fbb8
0222fb80  4b3c384f msctf!CTimList::Leave+0x6
0222fb84  4b3c14d7 msctf!CTimList::IsThreadId+0x5a
0222fb88  00000ca8
0222fb8c  4b405064 msctf!g_timlist
0222fb90  4b3c0000 msctf!_imp__CheckTokenMembership <PERF> (msctf+0x0)
0222fb94  01c70000
0222fb98  00000000
0222fb9c  4b405064 msctf!g_timlist
```

```
0222fba0  0222fb88
0222fba4  7d4dfd40 kernel32!FlsSetValue+0xc7
0222fba8  0222fca0
0222fbac  4b401dbd msctf!_except_handler3
0222fbb0  4b3c14e0 msctf!`string'+0x78
0222fbb4  0222fbd4
0222fbb8  0022f8a0
0222fbbc  00000001
0222fbc0  00000000
0222fbc4  00000000
0222fbc8  0222fc80
0222fbcc  0022f8a0
0222fbd0  0000156f
0222fbd4  0222fbf4
0222fbd8  020215a4 oleaut32!_DllMainCRTStartup+0x52
0222fbdc  02020000 oleaut32!_imp__RegFlushKey <PERF> (oleaut32+0x0)
0222fbe0  00000002
0222fbe4  00000000
0222fbe8  00000000
0222fbec  0222fc08
0222fbf0  00000001
0222fbf4  0222fc14
0222fbf8  7d610024 ntdll!LdrpCallInitRoutine+0x14
0222fbfc  02020000 oleaut32!_imp__RegFlushKey <PERF> (oleaut32+0x0)
0222fc00  00000001
0222fc04  00000000
0222fc08  00000001
0222fc0c  00000000
0222fc10  0022f8a0
0222fc14  00000001
0222fc18  00000000
0222fc1c  0222fcb0
0222fc20  7d62822e ntdll!LdrpInitializeThread+0x1a5
0222fc24  7d6a0180 ntdll!LdrpLoaderLock
0222fc28  7d62821c ntdll!LdrpInitializeThread+0x18f
0222fc2c  00000000
0222fc30  7efde000
0222fc34  00000000
...
0222fc6c  00000070
0222fc70  ffffffff
0222fc74  ffffffff
0222fc78  7d6281c7 ntdll!LdrpInitializeThread+0xd8
0222fc7c  7d6280d6 ntdll!LdrpInitializeThread+0x12c
0222fc80  00000000
0222fc84  00000000
0222fc88  0022f8a0
0222fc8c  0202155c oleaut32!_DllMainCRTStartup
0222fc90  7efde000
0222fc94  7d6a01f4 ntdll!PebLdr+0x14
0222fc98  0222fc2c
0222fc9c  00000000
0222fca0  0222fcfc
0222fca4  7d61f1f8 ntdll!_except_handler3
```

```
0222fca8  7d628148 ntdll!`string'+0xac
0222fcac  ffffffff
0222fcb0  7d62821c ntdll!LdrpInitializeThread+0x18f
0222fcb4  7d61e299 ntdll!ZwTestAlert+0x15
0222fcb8  7d628088 ntdll!_LdrpInitialize+0x1de
0222fcbc  0222fd20
0222fcc0  00000000
...
0222fcfc  0222ffec
0222fd00  7d61f1f8 ntdll!_except_handler3
0222fd04  7d628090 ntdll!`string'+0xfc
0222fd08  ffffffff
0222fd0c  7d628088 ntdll!_LdrpInitialize+0x1de
0222fd10  7d61ce0d ntdll!NtContinue+0x12
0222fd14  7d61e9b2 ntdll!KiUserApcDispatcher+0x3a
0222fd18  0222fd20
0222fd1c  00000001
0222fd20  0001002f
...
0222fdc8  00000000
0222fdcc  00000000
0222fdd0  00411032 NullThread!ILT+45(?ThreadProcYGKPAXZ)
0222fdd4  00000000
0222fdd8  7d4d1504 kernel32!BaseThreadStartThunk
0222fddc  00000023
0222fde0  00000202
...
0222ffb4  cccccccc
0222ffb8  0222ffec
0222ffbc  7d4dfe21 kernel32!BaseThreadStart+0x34
0222ffc0  00000000
0222ffc4  00000000
0222ffc8  00000000
0222ffcc  00000000
0222ffd0  00000000
0222ffd4  0222ffc4
0222ffd8  00000000
0222ffdc  ffffffff
0222ffe0  7d4d89c4 kernel32!_except_handler3
0222ffe4  7d4dfe28 kernel32!`string'+0x18
0222ffe8  00000000
0222ffec  00000000
0222fff0  00000000
0222fff4  00411032 NullThread!ILT+45(?ThreadProcYGKPAXZ)
0222fff8  00000000
0222fffc  00000000
02230000  ????????
```

The second crashed thread has much more symbolic information in it overwriting previous thread startup residue. It is mostly exception handling residue because exception handling consumes stack space as explained in the article **Who calls the postmortem debugger?** (Volume 1, page 113):

```
0:003> dds 0236a000 02370000
0236a000  00000000
...
0236a060  00000000
0236a064  0236a074
0236a068  00220000
0236a06c  7d61f7b4 ntdll!RtlpAllocateFromHeapLookaside+0x13
0236a070  00221378
0236a074  0236a29c
0236a078  7d61f748 ntdll!RtlAllocateHeap+0x1dd
0236a07c  7d61f78c ntdll!RtlAllocateHeap+0xee7
0236a080  0236a5f4
0236a084  00000000
...
0236a1b4  0236a300
0236a1b8  0236a1dc
0236a1bc  7d624267 ntdll!RtlIsDosDeviceName_Ustr+0x2f
0236a1c0  0236a21c
0236a1c4  7d624274 ntdll!RtlpDosSlashCONDevice
0236a1c8  00000001
0236a1cc  0236a317
0236a1d0  00000000
0236a1d4  0236a324
0236a1d8  0236a290
0236a1dc  7d6248af ntdll!RtlGetFullPathName_Ustr+0x80b
0236a1e0  7d6a00e0 ntdll!FastPebLock
0236a1e4  7d62489d ntdll!RtlGetFullPathName_Ustr+0x15b
0236a1e8  0236a5f4
0236a1ec  00000208
...
0236a224  00000000
0236a228  00000038
0236a22c  02080038 oleaut32!_PictSaveMetaFile+0x33
0236a230  00000000
...
0236a27c  00000000
0236a280  0236a53c
0236a284  7d61f1f8 ntdll!_except_handler3
0236a288  7d6245f0 ntdll!`string'+0x5c
0236a28c  ffffffff
0236a290  7d62489d ntdll!RtlGetFullPathName_Ustr+0x15b
0236a294  0236a5c8
0236a298  00000008
0236a29c  00000000
0236a2a0  0236a54c
0236a2a4  7d624bcf ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0x3d8
0236a2a8  7d6a00e0 ntdll!FastPebLock
0236a2ac  7d624ba1 ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0x3cb
0236a2b0  00000000
0236a2b4  0236e6d0
...
0236a2e0  000a0008
0236a2e4  7d624be8 ntdll!`string'
0236a2e8  00000000
```

```
0236a2ec  003a0038
...
0236a330  00650070
0236a334  0050005c
0236a338  00480043 advapi32!LsaGetQuotasForAccount+0x25
0236a33c  00610046
0236a340  006c0075
0236a344  00520074
0236a348  00700065
0236a34c  00780045
0236a350  00630065
0236a354  00690050
0236a358  00650070
0236a35c  00000000
0236a360  00000000
...
0236a4a0  0236a4b0
0236a4a4  00000001
0236a4a8  7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22
0236a4ac  00230b98
0236a4b0  0236a590
0236a4b4  7d61f5d1 ntdll!RtlFreeHeap+0x20e
0236a4b8  00221378
0236a4bc  7d61f5ed ntdll!RtlFreeHeap+0x70f
0236a4c0  00000000
0236a4c4  7d61f4ab ntdll!RtlFreeHeap
0236a4c8  00000000
0236a4cc  00000000
...
0236a538  00000000
0236a53c  0236a678
0236a540  7d61f1f8 ntdll!_except_handler3
0236a544  7d624ba8 ntdll!`string'+0x1c
0236a548  ffffffff
0236a54c  7d624ba1 ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0x3cb
0236a550  7d624c43 ntdll!RtlpDosPathNameToRelativeNtPathName_U+0x55
0236a554  00000001
0236a558  0236a56c
...
0236a590  0236a5c0
0236a594  7d620304 ntdll!RtlNtStatusToDosError+0x38
0236a598  7d620309 ntdll!RtlNtStatusToDosError+0x3d
0236a59c  7d61c828 ntdll!ZwWaitForSingleObject+0x15
0236a5a0  7d4d8c82 kernel32!WaitForSingleObjectEx+0xac
0236a5a4  00000124
0236a5a8  00000000
0236a5ac  7d4d8ca7 kernel32!WaitForSingleObjectEx+0xdc
0236a5b0  00000124
0236a5b4  7d61f49c ntdll!RtlGetLastWin32Error
0236a5b8  80070000
0236a5bc  00000024
...
0236a5f8  00000000
0236a5fc  0236a678
```

```
0236a600  7d4d89c4 kernel32!_except_handler3
0236a604  7d4d8cb0 kernel32!`string'+0x68
0236a608  ffffffff
0236a60c  7d4d8ca7 kernel32!WaitForSingleObjectEx+0xdc
0236a610  7d4d8bf1 kernel32!WaitForSingleObject+0x12
0236a614  7d61f49c ntdll!RtlGetLastWin32Error
0236a618  7d61c92d ntdll!NtClose+0x12
0236a61c  7d4d8e4f kernel32!CloseHandle+0x59
0236a620  00000124
0236a624  0236a688
0236a628  69511753 <Unloaded_faultrep.dll>+0x11753
0236a62c  6951175b <Unloaded_faultrep.dll>+0x1175b
0236a630  0236c6d0
...
0236a668  00000120
0236a66c  00000000
0236a670  0236a630
0236a674  7d94a2e9 user32!GetSystemMetrics+0x62
0236a678  0236f920
0236a67c  69510078 <Unloaded_faultrep.dll>+0x10078
0236a680  69503d10 <Unloaded_faultrep.dll>+0x3d10
0236a684  ffffffff
0236a688  6951175b <Unloaded_faultrep.dll>+0x1175b
0236a68c  69506136 <Unloaded_faultrep.dll>+0x6136
0236a690  0236e6d0
0236a694  0236c6d0
0236a698  0000009c
0236a69c  0236a6d0
0236a6a0  00002000
0236a6a4  0236eae4
0236a6a8  695061ff <Unloaded_faultrep.dll>+0x61ff
0236a6ac  00000000
0236a6b0  00000001
0236a6b4  0236f742
0236a6b8  69506210 <Unloaded_faultrep.dll>+0x6210
0236a6bc  00000028
0236a6c0  0236c76c
...
0236e6e0  0050005c
0236e6e4  00480043 advapi32!LsaGetQuotasForAccount+0x25
0236e6e8  00610046
...
0236e718  002204d8
0236e71c  0236e890
0236e720  77b940bb <Unloaded_VERSION.dll>+0x40bb
0236e724  77b91798 <Unloaded_VERSION.dll>+0x1798
0236e728  ffffffff
0236e72c  77b9178e <Unloaded_VERSION.dll>+0x178e
0236e730  69512587 <Unloaded_faultrep.dll>+0x12587
0236e734  0236e744
0236e738  00220000
0236e73c  7d61f7b4 ntdll!RtlpAllocateFromHeapLookaside+0x13
0236e740  00221378
0236e744  0236e96c
```

```
0236e748  7d61f748 ntdll!RtlAllocateHeap+0x1dd
0236e74c  7d61f78c ntdll!RtlAllocateHeap+0xee7
0236e750  0236eca4
0236e754  00000000
0236e758  0236ec94
0236e75c  7d620309 ntdll!RtlNtStatusToDosError+0x3d
0236e760  0236e7c8
0236e764  7d61c9db ntdll!NtQueryValueKey
0236e768  0236e888
0236e76c  0236e760
0236e770  7d61c9ed ntdll!NtQueryValueKey+0x12
0236e774  0236f920
0236e778  7d61f1f8 ntdll!_except_handler3
0236e77c  7d620310 ntdll!RtlpRunTable+0x490
0236e780  0236e790
0236e784  00220000
0236e788  7d61f7b4 ntdll!RtlpAllocateFromHeapLookaside+0x13
0236e78c  00221378
0236e790  0236e9b8
0236e794  7d61f748 ntdll!RtlAllocateHeap+0x1dd
0236e798  7d61f78c ntdll!RtlAllocateHeap+0xee7
0236e79c  0236ef18
0236e7a0  00000000
0236e7a4  00000000
0236e7a8  00220000
0236e7ac  0236e89c
0236e7b0  00000000
0236e7b4  00000128
0236e7b8  00000000
0236e7bc  0236e8c8
0236e7c0  0236e7c8
0236e7c4  c0000034
0236e7c8  0236e814
0236e7cc  7d61f1f8 ntdll!_except_handler3
0236e7d0  7d61f5f0 ntdll!CheckHeapFillPattern+0x64
0236e7d4  ffffffff
0236e7d8  7d61f5ed ntdll!RtlFreeHeap+0x70f
0236e7dc  7d4ded95 kernel32!FindClose+0x9b
0236e7e0  00220000
0236e7e4  00000000
0236e7e8  00220000
0236e7ec  00000000
0236e7f0  002314b4
0236e7f4  7d61ca1d ntdll!NtQueryInformationProcess+0x12
0236e7f8  7d4da465 kernel32!GetErrorMode+0x18
0236e7fc  ffffffff
0236e800  0000000c
0236e804  7d61ca65 ntdll!ZwSetInformationProcess+0x12
0236e808  7d4da441 kernel32!SetErrorMode+0x37
0236e80c  ffffffff
0236e810  0000000c
0236e814  0236e820
0236e818  00000004
0236e81c  00000000
```

```
0236e820  00000005
0236e824  0236eae8
0236e828  7d4e445f kernel32!GetLongPathNameW+0x38f
0236e82c  7d4e4472 kernel32!GetLongPathNameW+0x3a2
0236e830  00000001
0236e834  00000103
0236e838  00000000
0236e83c  0236f712
0236e840  7efaf000
0236e844  002316f0
0236e848  0000005c
0236e84c  7efaf000
0236e850  00000004
0236e854  002314b4
0236e858  0000ea13
0236e85c  0236e894
0236e860  00456b0d advapi32!RegQueryValueExW+0x96
0236e864  00000128
0236e868  0236e888
0236e86c  0236e8ac
0236e870  0236e8c8
0236e874  0236e8a4
0236e878  0236e89c
0236e87c  0236e88c
0236e880  7d635dc4 ntdll!iswdigit+0xf
0236e884  00000064
0236e888  00000004
0236e88c  7d624d81 ntdll!RtlpValidateCurrentDirectory+0xf6
0236e890  7d635d4e ntdll!RtlIsDosDeviceName_Ustr+0x1c0
0236e894  00000064
0236e898  0236e9d0
0236e89c  0236e9e7
0236e8a0  00000000
0236e8a4  0236e9f4
0236e8a8  0236e960
0236e8ac  7d6248af ntdll!RtlGetFullPathName_Ustr+0x80b
0236e8b0  7d6a00e0 ntdll!FastPebLock
0236e8b4  7d62489d ntdll!RtlGetFullPathName_Ustr+0x15b
0236e8b8  0236eca4
0236e8bc  00000208
0236e8c0  0236ec94
0236e8c4  00000000
0236e8c8  00220178
0236e8cc  00000004
0236e8d0  0236eb3c
0236e8d4  0236e8c8
0236e8d8  7d624d81 ntdll!RtlpValidateCurrentDirectory+0xf6
0236e8dc  0236e8f8
0236e8e0  7d6246c1 ntdll!RtlIsDosDeviceName_Ustr+0x14
0236e8e4  0236ea1c
0236e8e8  0236ea33
0236e8ec  00000000
0236e8f0  0236ea40
0236e8f4  0236e9ac
```

```
0236e8f8  7d6248af ntdll!RtlGetFullPathName_Ustr+0x80b
0236e8fc  7d6a00e0 ntdll!FastPebLock
0236e900  7d62489d ntdll!RtlGetFullPathName_Ustr+0x15b
0236e904  0236ef18
0236e908  00000208
...
0236e934  00000022
0236e938  00460044 advapi32!GetPerflibKeyValue+0x19e
0236e93c  0236ecd0
0236e940  00000000
0236e944  00000044
0236e948  02080044 oleaut32!_PictSaveMetaFile+0x3f
0236e94c  00000000
0236e950  4336ec0c
...
0236e9a8  0236ebd0
0236e9ac  7d62155b ntdll!RtlAllocateHeap+0x460
0236e9b0  7d61f78c ntdll!RtlAllocateHeap+0xee7
0236e9b4  00000000
0236e9b8  000003ee
0236e9bc  0236ed2c
0236e9c0  7d624bcf ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0x3d8
0236e9c4  7d6a00e0 ntdll!FastPebLock
0236e9c8  00000ab0
0236e9cc  00000381
0236e9d0  00233950
0236e9d4  0236ebfc
0236e9d8  7d62155b ntdll!RtlAllocateHeap+0x460
0236e9dc  7d61f78c ntdll!RtlAllocateHeap+0xee7
0236e9e0  00000003
0236e9e4  fffffffc
0236e9e8  00000aa4
0236e9ec  00230ba0
0236e9f0  00000004
0236e9f4  003a0043
0236e9f8  00000000
0236e9fc  000a0008
0236ea00  7d624be8 ntdll!`string'
0236ea04  00000000
0236ea08  00460044 advapi32!GetPerflibKeyValue+0x19e
0236ea0c  0236ecd0
0236ea10  00233948
...
0236ea44  00220640
0236ea48  7d62273d ntdll!RtlIntegerToUnicode+0x126
0236ea4c  0000000c
...
0236eab4  0236f79c
0236eab8  7d61f1f8 ntdll!_except_handler3
0236eabc  7d622758 ntdll!RtlpIntegerWChars+0x54
0236eac0  00220178
0236eac4  0236ed3c
0236eac8  00000005
0236eacc  0236ed00
```

```
0236ead0  7d622660 ntdll!RtlConvertSidToUnicodeString+0x1cb
0236ead4  00220178
0236ead8  0236eaf0
0236eadc  0236eaec
0236eae0  00000001
0236eae4  7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22
0236eae8  00223620
0236eaec  00220178
0236eaf0  7d61f5d1 ntdll!RtlFreeHeap+0x20e
0236eaf4  002217f8
0236eaf8  7d61f5ed ntdll!RtlFreeHeap+0x70f
0236eafc  00000000
0236eb00  00220178
...
0236eb48  0236eb58
0236eb4c  7d635dc4 ntdll!iswdigit+0xf
0236eb50  00220178
0236eb54  00000381
0236eb58  002343f8
0236eb5c  0236eb78
0236eb60  7d620deb ntdll!RtlpCoalesceFreeBlocks+0x383
0236eb64  00000381
0236eb68  002343f8
0236eb6c  00220000
0236eb70  00233948
0236eb74  00220000
0236eb78  00000000
0236eb7c  00220000
0236eb80  0236ec60
0236eb84  7d620fbe ntdll!RtlFreeHeap+0x6b0
0236eb88  00220608
0236eb8c  7d61f5ed ntdll!RtlFreeHeap+0x70f
0236eb90  000000e8
0236eb94  7d61cd23 ntdll!ZwWriteVirtualMemory
0236eb98  7efde000
0236eb9c  000000e8
0236eba0  00233948
0236eba4  7efde000
0236eba8  000002e8
0236ebac  0000005d
0236ebb0  00220178
0236ebb4  00000156
0236ebb8  0236e9b4
0236ebbc  00233948
0236ebc0  7d61f1f8 ntdll!_except_handler3
0236ebc4  00000ab0
0236ebc8  00233948
0236ebcc  00233950
0236ebd0  00220178
0236ebd4  00220000
0236ebd8  00000ab0
0236ebdc  00220178
0236ebe0  00000000
0236ebe4  00233950
```

```
0236ebe8  7d4ddea8 kernel32!`string'+0x50
0236ebec  00000000
0236ebf0  00233950
0236ebf4  00220178
0236ebf8  00000aa4
0236ebfc  00000000
0236ec00  0236ec54
0236ec04  7d63668a ntdll!RtlCreateProcessParameters+0x375
0236ec08  7d63668f ntdll!RtlCreateProcessParameters+0x37a
0236ec0c  7d6369e9 ntdll!RtlCreateProcessParameters+0x35f
0236ec10  00000000
...
0236ec4c  0000007f
0236ec50  0236ef4c
0236ec54  7d61f1f8 ntdll!_except_handler3
0236ec58  7d61f5f0 ntdll!CheckHeapFillPattern+0x64
0236ec5c  ffffffff
0236ec60  7d61f5ed ntdll!RtlFreeHeap+0x70f
0236ec64  7d6365e2 ntdll!RtlDestroyProcessParameters+0x1b
0236ec68  00220000
0236ec6c  00000000
0236ec70  00233950
0236ec74  0236ef5c
0236ec78  7d4ec4bc kernel32!BasePushProcessParameters+0x806
0236ec7c  00233950
0236ec80  7d4ec478 kernel32!BasePushProcessParameters+0x7c5
0236ec84  7efde000
0236ec88  0236f748
0236ec8c  00000000
0236ec90  0236ed92
0236ec94  00000000
0236ec98  00000000
0236ec9c  01060104
0236eca0  0236f814
0236eca4  0020001e
0236eca8  7d535b50 kernel32!`string'
0236ecac  00780076
0236ecb0  002314e0
0236ecb4  00780076
0236ecb8  0236ed2c
0236ecbc  00020000
0236ecc0  7d4ddee4 kernel32!`string'
0236ecc4  0236efec
...
0236ed3c  006d0061
0236ed40  00460020 advapi32!GetPerflibKeyValue+0x17a
0236ed44  006c0069
0236ed48  00730065
0236ed4c  00280020
0236ed50  00380078
0236ed54  00290036
0236ed58  0044005c advapi32!CryptDuplicateHash+0x3
0236ed5c  00620065
0236ed60  00670075
```

```
...
0236ee7c  0236ee8c
0236ee80  00000001
0236ee84  7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22
0236ee88  00230dc0
0236ee8c  0236ef6c
0236ee90  0236eea0
0236ee94  00000001
0236ee98  7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22
0236ee9c  00223908
0236eea0  0236ef80
0236eea4  7d61f5d1 ntdll!RtlFreeHeap+0x20e
0236eea8  00221d38
0236eeac  7d61f5ed ntdll!RtlFreeHeap+0x70f
0236eeb0  7d61f4ab ntdll!RtlFreeHeap
0236eeb4  7d61c91b ntdll!NtClose
0236eeb8  00000000
...
0236ef08  00000000
0236ef0c  7d621954 ntdll!RtlImageNtHeaderEx+0xee
0236ef10  7efde000
0236ef14  00001000
0236ef18  00000000
0236ef1c  000000e8
0236ef20  004000e8 NullThread!_enc$textbss$begin <PERF> (NullThread+0xe8)
0236ef24  00000000
0236ef28  0236ef10
0236ef2c  00000000
0236ef30  0236f79c
0236ef34  7d61f1f8 ntdll!_except_handler3
0236ef38  7d621954 ntdll!RtlImageNtHeaderEx+0xee
0236ef3c  00220000
...
0236ef68  0236eeb0
0236ef6c  7d61f5ed ntdll!RtlFreeHeap+0x70f
0236ef70  0236f79c
0236ef74  7d61f1f8 ntdll!_except_handler3
0236ef78  7d61f5f0 ntdll!CheckHeapFillPattern+0x64
0236ef7c  ffffffff
0236ef80  7d61f5ed ntdll!RtlFreeHeap+0x70f
0236ef84  7d4ea183 kernel32!CreateProcessInternalW+0x21f5
0236ef88  00220000
0236ef8c  00000000
0236ef90  00223910
0236ef94  7d4ebc0b kernel32!CreateProcessInternalW+0x1f26
0236ef98  00000000
0236ef9c  00000096
0236efa0  0236f814
0236efa4  00000103
0236efa8  7efde000
0236efac  00000001
0236efb0  0236effc
0236efb4  00000200
0236efb8  00000cb0
```

```
0236efbc  0236f00c
0236efc0  0236efdc
0236efc4  7d6256e8 ntdll!bsearch+0x42
0236efc8  00180144
0236efcc  0236efe0
0236efd0  7d625992 ntdll!ARRAY_FITS+0x29
0236efd4  00000a8c
0236efd8  00000000
0236efdc  00000000
0236efe0  00080000
0236efe4  00070000
0236efe8  00040000
0236efec  00000044
0236eff0  00000000
0236eff4  7d535b50 kernel32!`string'
0236eff8  00000000
0236effc  00000000
...
0236f070  00000001
0236f074  7d625ad8 ntdll!RtlFindActivationContextSectionString+0xe1
0236f078  004000e8 NullThread!_enc$textbss$begin <PERF> (NullThread+0xe8)
0236f07c  0236f0cc
0236f080  00000000
0236f084  7d6256e8 ntdll!bsearch+0x42
0236f088  00180144
0236f08c  0236f0a0
0236f090  7d625992 ntdll!ARRAY_FITS+0x29
0236f094  00000a8c
...
0236f0d0  0236f120
0236f0d4  7d625b62 ntdll!RtlpFindUnicodeStringInSection+0x7b
0236f0d8  0236f204
0236f0dc  00000020
...
0236f190  000002a8
0236f194  7d625b62 ntdll!RtlpFindUnicodeStringInSection+0x7b
0236f198  00000001
0236f19c  00000000
0236f1a0  0236f1d0
0236f1a4  7d6257f1 ntdll!RtlpFindNextActivationContextSection+0x64
0236f1a8  00181f1c
...
0236f1f0  7efaf000
0236f1f4  7d625ad8 ntdll!RtlFindActivationContextSectionString+0xe1
0236f1f8  0236f214
0236f1fc  0236f24c
0236f200  00000000
0236f204  7d6256e8 ntdll!bsearch+0x42
0236f208  00180144
...
0236f24c  00000200
0236f250  00000734
0236f254  7d625b62 ntdll!RtlpFindUnicodeStringInSection+0x7b
0236f258  0236f384
```

```
...
0236f3f0  00000000
0236f3f4  00000000
0236f3f8  01034236
0236f3fc  00000000
0236f400  7d4d1510 kernel32!BaseProcessStartThunk
0236f404  00000018
0236f408  00003000
...
0236f62c  0236f63c
0236f630  00000001
0236f634  7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22
0236f638  00231088
0236f63c  0236f71c
...
0236f70c  002333b8
0236f710  0236f720
0236f714  00000001
0236f718  7d61f645 ntdll!RtlpFreeToHeapLookaside+0x22
0236f71c  00228fb0
0236f720  0236f800
0236f724  7d61f5d1 ntdll!RtlFreeHeap+0x20e
0236f728  00221318
0236f72c  7d61f5ed ntdll!RtlFreeHeap+0x70f
0236f730  00000000
0236f734  00000096
0236f738  0236f814
0236f73c  00220608
0236f740  7d61f5ed ntdll!RtlFreeHeap+0x70f
0236f744  0236f904
0236f748  008e0000
0236f74c  002334c2
...
0236f784  0236f7bc
0236f788  7d63d275 ntdll!_vsnwprintf+0x30
0236f78c  0236f79c
0236f790  0000f949
0236f794  0236ef98
0236f798  00000095
0236f79c  0236fb7c
0236f7a0  7d4d89c4 kernel32!_except_handler3
0236f7a4  7d4ed1d0 kernel32!`string'+0xc
0236f7a8  ffffffff
0236f7ac  7d4ebc0b kernel32!CreateProcessInternalW+0x1f26
0236f7b0  7d4d14a2 kernel32!CreateProcessW+0x2c
0236f7b4  00000000
...
0236f7f0  0236fb7c
0236f7f4  7d61f1f8 ntdll!_except_handler3
0236f7f8  7d61d051 ntdll!NtWaitForMultipleObjects+0x15
0236f7fc  7d61c92d ntdll!NtClose+0x12
0236f800  7d4d8e4f kernel32!CloseHandle+0x59
0236f804  00000108
0236f808  0236fb8c
```

```
0236f80c  7d535b07 kernel32!UnhandledExceptionFilter+0x815
0236f810  00000108
0236f814  00430022 advapi32!_imp__OutputDebugStringW <PERF>
(advapi32+0x22)
0236f818  005c003a
0236f81c  00720050
...
0236f8ec  0055005c
0236f8f0  00650073
0236f8f4  00440072 advapi32!CryptDuplicateHash+0x19
0236f8f8  006d0075
0236f8fc  00730070
0236f900  006e005c
0236f904  00770065
0236f908  0064002e
0236f90c  0070006d
0236f910  0020003b
0236f914  00220071
0236f918  00000000
0236f91c  00000096
0236f920  7d4dda47 kernel32!DuplicateHandle+0xd0
0236f924  7d4dda47 kernel32!DuplicateHandle+0xd0
0236f928  0236fb8c
0236f92c  7d5358cb kernel32!UnhandledExceptionFilter+0x5f1
0236f930  0236f9f0
0236f934  00000001
0236f938  00000000
0236f93c  7d535b43 kernel32!UnhandledExceptionFilter+0x851
0236f940  00000000
0236f944  00000000
0236f948  00000000
0236f94c  0236f95c
0236f950  00000098
0236f954  000001a2
0236f958  01c423b0
0236f95c  0236fb84
0236f960  7d62155b ntdll!RtlAllocateHeap+0x460
0236f964  7d61f78c ntdll!RtlAllocateHeap+0xee7
0236f968  00000000
0236f96c  0000008c
0236f970  00000000
0236f974  7d4d8472 kernel32!$$VProc_ImageExportDirectory+0x6d4e
0236f978  0236fa1c
0236f97c  00000044
0236f980  00000000
0236f984  7d535b50 kernel32!`string'
0236f988  00000000
0236f98c  00000000
0236f990  00000000
0236f994  00000000
0236f998  00000000
0236f99c  00000000
0236f9a0  00000000
0236f9a4  00000000
```

```
0236f9a8  00000000
0236f9ac  00000000
0236f9b0  00000000
0236f9b4  00000000
0236f9b8  00000000
0236f9bc  00000000
0236f9c0  0010000e
0236f9c4  7ffe0030 SharedUserData+0x30
0236f9c8  000000e8
0236f9cc  00000108
0236f9d0  00000200
0236f9d4  00000734
0236f9d8  00000018
0236f9dc  00000000
0236f9e0  7d5621d0 kernel32!ProgramFilesEnvironment+0x74
0236f9e4  00000040
0236f9e8  00000000
0236f9ec  00000000
0236f9f0  0000000c
0236f9f4  00000000
0236f9f8  00000001
0236f9fc  00000118
0236fa00  000000e8
0236fa04  c0000005
0236fa08  00000000
0236fa0c  00000008
0236fa10  00000000
0236fa14  00000110
0236fa18  0236f814
0236fa1c  6950878a <Unloaded_faultrep.dll>+0x878a
0236fa20  00120010
0236fa24  7d51c5e4 kernel32!`string'
0236fa28  00000003
0236fa2c  05bc0047
...
0236fa74  0057005c
0236fa78  004b0032 advapi32!szPerflibSectionName <PERF> (advapi32+0x80032)
0236fa7c  005c0033
0236fa80  00790073
...
0236fac8  0000002b
0236facc  00000000
0236fad0  7d61e3e6 ntdll!ZwWow64CsrNewThread+0x12
0236fad4  00000000
...
0236fb44  00000000
0236fb48  00000000
0236fb4c  7d61cb0d ntdll!ZwQueryVirtualMemory+0x12
0236fb50  7d54eeb8 kernel32!_ValidateEH3RN+0xb6
0236fb54  ffffffff
0236fb58  7d4dfe28 kernel32!`string'+0x18
0236fb5c  00000000
0236fb60  0236fb78
0236fb64  0000001c
```

```
0236fb68  0000000f
0236fb6c  7d4dfe28 kernel32!`string'+0x18
0236fb70  0000f949
0236fb74  0236f814
0236fb78  7d4df000 kernel32!CheckForSameCurdir+0x39
0236fb7c  0236fbd4
0236fb80  7d4d89c4 kernel32!_except_handler3
0236fb84  7d535be0 kernel32!`string'+0xc
0236fb88  ffffffff
0236fb8c  7d535b43 kernel32!UnhandledExceptionFilter+0x851
0236fb90  7d508f4e kernel32!BaseThreadStart+0x4a
0236fb94  0236fbb4
0236fb98  7d4d8a25 kernel32!_except_handler3+0x61
0236fb9c  0236fbbc
0236fba0  00000000
0236fba4  0236fbbc
0236fba8  00000000
0236fbac  00000000
0236fbb0  00000000
0236fbb4  0236fca0
0236fbb8  0236fcf0
0236fbbc  0236fbe0
0236fbc0  7d61ec2a ntdll!ExecuteHandler2+0x26
0236fbc4  0236fca0
0236fbc8  0236ffdc
0236fbcc  0236fcf0
0236fbd0  0236fc7c
0236fbd4  0236ffdc
0236fbd8  7d61ec3e ntdll!ExecuteHandler2+0x3a
0236fbdc  0236ffdc
0236fbe0  0236fc88
0236fbe4  7d61ebfb ntdll!ExecuteHandler+0x24
0236fbe8  0236fca0
0236fbec  0236ffdc
0236fbf0  00000000
0236fbf4  0236fc7c
0236fbf8  7d4d89c4 kernel32!_except_handler3
0236fbfc  00000000
0236fc00  0036fca0
0236fc04  0236fc18
0236fc08  7d640ca6 ntdll!RtlCallVectoredContinueHandlers+0x15
0236fc0c  0236fca0
0236fc10  0236fcf0
0236fc14  7d6a0608 ntdll!RtlpCallbackEntryList
0236fc18  0236fc88
0236fc1c  7d6354c9 ntdll!RtlDispatchException+0x11f
0236fc20  0236fca0
0236fc24  0236fcf0
0236fc28  00000000
0236fc2c  00000000
...
0236fc88  0236ffec
0236fc8c  7d61dd26 ntdll!NtRaiseException+0x12
0236fc90  7d61ea51 ntdll!KiUserExceptionDispatcher+0x29
```

```
0236fc94  0236fca0
0236fc98  0236fcf0
0236fc9c  00000000
0236fca0  c0000005
0236fca4  00000000
0236fca8  00000000
0236fcac  00000000
0236fcb0  00000002
0236fcb4  00000008
0236fcb8  00000000
0236fcbc  00000000
0236fcc0  00000000
0236fcc4  6b021fa0
0236fcc8  78b83980
0236fccc  00000000
0236fcd0  00000000
0236fcd4  00000000
0236fcd8  7efad000
0236fcdc  023afd00
0236fce0  023af110
0236fce4  78b83980
0236fce8  010402e1
0236fcec  00000000
0236fcf0  0001003f
0236fcf4  00000000
0236fcf8  00000000
0236fcfc  00000000
0236fd00  00000000
0236fd04  00000000
0236fd08  00000000
0236fd0c  0000027f
0236fd10  00000000
0236fd14  0000ffff
0236fd18  00000000
0236fd1c  00000000
0236fd20  00000000
0236fd24  00000000
0236fd28  00000000
0236fd2c  00000000
0236fd30  00000000
0236fd34  00000000
0236fd38  00000000
0236fd3c  00000000
0236fd40  00000000
0236fd44  00000000
0236fd48  00000000
0236fd4c  00000000
0236fd50  00000000
0236fd54  00000000
0236fd58  00000000
0236fd5c  00000000
0236fd60  00000000
0236fd64  00000000
0236fd68  00000000
```

```
0236fd6c  00000000
0236fd70  00000000
0236fd74  00000000
0236fd78  00000000
0236fd7c  0000002b
0236fd80  00000053
0236fd84  0000002b
0236fd88  0000002b
0236fd8c  00000000
0236fd90  00000000
0236fd94  00000000
0236fd98  00000000
0236fd9c  47f30000
0236fda0  00000000
0236fda4  0236ffec
0236fda8  00000000
0236fdac  00000023
0236fdb0  00010246
0236fdb4  0236ffbc
0236fdb8  0000002b
0236fdbc  0000027f
0236fdc0  00000000
0236fdc4  00000000
0236fdc8  00000000
0236fdcc  00000000
0236fdd0  00000000
0236fdd4  00001f80
0236fdd8  00000000
0236fddc  00000000
...
0236ffb4  00000000
0236ffb8  00000000
0236ffbc  7d4dfe21 kernel32!BaseThreadStart+0x34
0236ffc0  00000000
0236ffc4  00000000
0236ffc8  00000000
0236ffcc  00000000
0236ffd0  c0000005
0236ffd4  0236ffc4
0236ffd8  0236fbb4
0236ffdc  ffffffff
0236ffe0  7d4d89c4 kernel32!_except_handler3
0236ffe4  7d4dfe28 kernel32!`string'+0x18
0236ffe8  00000000
0236ffec  00000000
0236fff0  00000000
0236fff4  00000000
0236fff8  00000000
0236fffc  00000000
02370000  ????????
```

OPTIMIZED VM LAYOUT

This pattern is a specialization of the general **Changed Environment** pattern (Volume 1, page 283) where the whole modules are moved in virtual memory by changing their load order and addresses. This can result in dormant bugs being exposed and one of workarounds usually is to disable such external optimization programs and services or adding applications that behave improperly to corresponding exclusion lists. Some optimized virtual memory cases can be easily detected by looking at module list where system DLLs are remapped to lower addresses instead of 0×7X000000 range:

```
0:000> lm
start    end         module name
00400000 00416000   Application
00470000 0050b000   advapi32
00520000 00572000   shlwapi
02340000 023cb000   oleaut32
04b80000 0523e000   System_Data_ni
1a400000 1a524000   urlmon
4dd60000 4df07000   GdiPlus
5f120000 5f12e000   ntlanman
5f860000 5f891000   netui1
5f8a0000 5f8b6000   netui0
637a0000 63d28000   System_Xml_ni
64890000 6498c000   System_Configuration_ni
64e70000 6515c000   System_Data
65ce0000 65ecc000   System_Web_Services_ni
71bd0000 71be1000   mpr
71bf0000 71bf8000   ws2help
71c00000 71c17000   ws2_32
71c20000 71c32000   tsappcmp
71c40000 71c97000   netapi32
73070000 73097000   winspool
75e90000 75e97000   drprov
75ea0000 75eaa000   davclnt
76190000 761a2000   msasn1
761b0000 76243000   crypt32
76a80000 76a92000   atl
76b80000 76bae000   credui
76dc0000 76de8000   adsldpc
76df0000 76e24000   activeds
76f00000 76f08000   wtsapi32
76f10000 76f3e000   wldap32
771f0000 77201000   winsta
77670000 777a9000   ole32
77ba0000 77bfa000   msvcrt
78130000 781cb000   msvcr80
79000000 79046000   mscoree
79060000 790b6000   mscorjit
790c0000 79bf6000   mscorlib_ni
79e70000 7a3ff000   mscorwks
```

```
7a440000 7ac2a000   System_ni
7ade0000 7af7c000   System_Drawing_ni
7afd0000 7bc6c000   System_Windows_Forms_ni
7c340000 7c396000   msvcr71
7c8d0000 7d0ce000   shell32
7d4c0000 7d5f0000   kernel32
7d600000 7d6f0000   ntdll
7d800000 7d890000   gdi32
7d8d0000 7d920000   secur32
7d930000 7da00000   user32
7da20000 7db00000   rpcrt4
7dbd0000 7dcd3000   comctl32
7df50000 7dfc0000   uxtheme
7e020000 7e02f000   samlib
```

The similar address space reshuffling happens with ASLR-enabled applications (Volume 1, page 674) with the difference that system modules are never re-mapped below 0×70000000 address.

INVALID HANDLE

Invalid handle exception (0xC0000008) is frequently seen in crash dumps. It results from an invalid handle value passed to CloseHandle function and other Win32 API or when a handle or a return status is checked manually for validity and the same exception is raised via RaiseException or internally via RtlRaiseStatus.

For example, critical sections are implemented using events and invalid event handle can result in this exception (shown in smaller font for visual clarity):

```
STACK_TEXT:
025bff00 7c94243c c0000008 7c9010ed 00231af0 ntdll!RtlRaiseStatus+0×26
025bff80 7c90104b 0015b4ac 77e76a6f 0015b4ac ntdll!RtlpWaitForCriticalSection+0×204
025bff88 77e76a6f 0015b4ac 010d2040 00000000 ntdll!RtlEnterCriticalSection+0×46
025bffa8 77e76c0a 0015b420 025bffec 7c80b683 rpcrt4!BaseCachedThreadRoutine+0xad
025bffb4 7c80b683 001feae8 010d2040 00000000 rpcrt4!ThreadStartRoutine+0×1a
025bffec 00000000 77e76bf0 001feae8 00000000 kernel32!BaseThreadStart+0×37
```

By default, unless raised manually, this exception doesn’t result in a default postmortem debugger launched to save a crash dump. In order to do this we need to run the application under a debugger and save a crash dump upon this exception or use exception monitoring tools that save first-chance exceptions like Debug Diagnostics, ADPlus or Exception Monitor (see **Early Crash Dump** pattern, Volume 1, page 465):

```
0:002> g
(7b0.d1c): Invalid handle - code c0000008 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000000 ecx=00000000 edx=00000000 esi=7d999906
edi=00403378
eip=7d61c92d esp=0012ff68 ebp=0012ff70 iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000202
ntdll!NtClose+0×12:
7d61c92d c20400          ret     4

0:000> g
(7b0.d1c): Invalid handle - code c0000008 (!!! second chance !!!)
eax=00000001 ebx=00000000 ecx=00000000 edx=00000000 esi=7d999906
edi=00403378
eip=7d61c92d esp=0012ff68 ebp=0012ff70 iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000202
ntdll!NtClose+0×12:
7d61c92d c20400          ret     4
```

In order to catch it using postmortem debuggers we can use Application Verifier and configure its basic checks to include invalid handles. Then we will have crash dumps if a postmortem debugger or WER is properly configured. The typical stack might look like below and point straight to the problem component:

```
EXCEPTION_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 6b006369
   ExceptionCode: 80000003 (Break instruction exception)
  ExceptionFlags: 00000000
NumberParameters: 1
   Parameter[0]: 00000000

DEFAULT_BUCKET_ID:  STATUS_BREAKPOINT

0:000> kL
ChildEBP RetAddr
0301ff44 0489a480 ntdll!NtClose+0x12
WARNING: Stack unwind information not available. Following frames may be
wrong.
0301ff54 7d4d8e4f vfbasics+0xa480
0301ff60 04894df9 kernel32!CloseHandle+0×59
0301ff70 00401022 vfbasics+0×4df9
0301ffc0 7d4e7d2a BadHandle+0×1022
0301fff0 00000000 kernel32!BaseProcessStart+0×28
```

or it might look like this:

```
0:000> kL
Child-SP          RetAddr           Call Site
00000000`0012ed58 00000000`01f9395a ntdll!DbgBreakPoint
00000000`0012ed60 00000000`023e29a7 vrfcore!VerifierStopMessageEx+0×846
00000000`0012f090 00000000`023d9384 vfbasics+0×129a7
00000000`0012f0f0 00000000`77f251ec vfbasics+0×9384
00000000`0012f180 00000000`77ee5f36 ntdll!RtlpCallVectoredHandlers+0×26f
00000000`0012f210 00000000`77ee6812 ntdll!RtlDispatchException+0×46
00000000`0012f8c0 00000000`77ef325a ntdll!RtlRaiseException+0xae
00000000`0012fe00 00000000`77d6e314
ntdll!KiRaiseUserExceptionDispatcher+0×3a
00000000`0012fed0 00000001`40001028 kernel32!CloseHandle+0×5f
00000000`0012ff00 00000001`40001294 BadHandle+0×1028
00000000`0012ff30 00000000`77d5964c BadHandle+0×1294
00000000`0012ff80 00000000`00000000 kernel32!BaseProcessStart+0×29
```

vfbasics and ***vrfcore*** are Application Verifier DLLs that possibly translate an invalid handle exception to a breakpoint exception and therefore trigger the launch of a postmortem debugger from an unhandled exception filter. Application Verifier version (x64 or x86) must match the application platform (64-bit or 32-bit, page 413).

If invalid handle exception is raised manually we get the status code and possibly problem component immediately from **!analyze** command:

```
FAULTING_IP:
kernel32!RaiseException+53
7d4e2366 5e              pop     esi

EXCEPTION_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 7d4e2366 (kernel32!RaiseException+0x00000053)
   ExceptionCode: c0000008 (Invalid handle)
  ExceptionFlags: 00000000
NumberParameters: 0
Thread tried to close a handle that was invalid or illegal to close

DEFAULT_BUCKET_ID:  STATUS_INVALID_HANDLE

PROCESS_NAME:  BadHandle.exe

ERROR_CODE: (NTSTATUS) 0xc0000008 - An invalid HANDLE was specified.

STACK_TEXT:
0012ff64 00401043 c0000008 00000000 00000000 kernel32!RaiseException+0×53
WARNING: Stack unwind information not available. Following frames may be
wrong.
0012ffc0 7d4e7d2a 00000000 00000000 7efde000 BadHandle+0×1043
0012fff0 00000000 004012f9 00000000 00000000
kernel32!BaseProcessStart+0×28

FAULTING_THREAD:  00000b64

PRIMARY_PROBLEM_CLASS:  STATUS_INVALID_HANDLE

BUGCHECK_STR:  APPLICATION_FAULT_STATUS_INVALID_HANDLE
```

Because we have WinDbg warning about stack unwind we can double check the disassembly of RaiseException return address:

```
0:000> ub 00401043
BadHandle+0×1029:
00401029 push    offset BadHandle+0×212c (0040212c)
0040102e push    0
00401030 call    esi
00401032 push    0
00401034 push    0
00401036 push    0
00401038 push    0C0000008h
0040103d call    dword ptr [BadHandle+0×2004 (00402004)]

0:000> dps 00402004 l1
00402004  7d4e2318 kernel32!RaiseException
```

In such cases the real problem could be memory corruption overwriting stored valid handle values.

OVERAGED SYSTEM

Software ageing can be the cause of some problems. Sometimes a look at the following WinDbg output can give irresistible temptation to suggest periodic reboots:

```
Debug session time: Wed April 28 15:36:52.330 2008 (GMT+0)
System Uptime: 124 days 6:27:16.658
```

THREAD STARVATION

This pattern happens when some threads or processes have higher priority and are favored by OS thread dispatcher effectively starving other threads. If prioritized threads are CPU-bound we also see CPU spikes (**Spiking Thread** pattern, Volume 1, page 305). However, if their thread priorities were normal they would have been preempted by other threads and latter threads would not be starved. Here is one example where 2 threads from 2 different applications but from one user session are spiking on 2 processors (threads running on other processors have above normal and normal priority):

```
System Uptime: 0 days 6:40:41.143

0: kd> !running

System Processors f (affinity mask)
  Idle Processors None

     Prcb      Current   Next
  0  f773a120  89864c86 ................
  1  f773d120  89f243d2 ................
  2  f7737120  89f61398 ................
  3  f773f120  897228a0 ................

0: kd> .thread /r /p 89f61398
Implicit thread is now 89f61398
Implicit process is now 88bcc2e0
Loading User Symbols
```

```
0: kd> !thread 89f61398 1f
THREAD 89f61398  Cid 16f8.2058  Teb: 7ffdf000 Win32Thread: bc41aea8
RUNNING on processor 2
Not impersonating
DeviceMap                 e48a6508
Owning Process            88bcc2e0       Image:         application.exe
Wait Start TickCount      1569737        Ticks: 0
Context Switch Count      7201654                 LargeStack
UserTime                  01:24:06.687
KernelTime                00:14:53.828
Win32 Start Address application (0×0040a52c)
Start Address kernel32!BaseProcessStartThunk (0×77e617f8)
Stack Init ba336000 Current ba335d00 Base ba336000 Limit ba330000 Call 0
Priority 24 BasePriority 24 PriorityDecrement 0
ChildEBP RetAddr
0012e09c 762c3b7d USER32!IsWindowVisible
0012e0c4 762d61bb MSVBVM50!RbyCountVisibleDesks+0×3c
0012e0d0 004831f6 MSVBVM50!rtcDoEvents+0×7
0012e348 0046d1ae application+0×831f6
0012e3a0 762ce5a9 application+0×6d1ae
0012e3dc 762ce583 MSVBVM50!CallProcWithArgs+0×20
0012e3f8 762db781 MSVBVM50!InvokeVtblEvent+0×33
0012e434 762cfbc2 MSVBVM50!InvokeEvent+0×32
0012e514 762cfa4a MSVBVM50!EvtErrFireWorker+0×175
0012e55c 762b1aa3 MSVBVM50!EvtErrFire+0×18
0012e5ac 7739bffa MSVBVM50!CThreadPool::GetThreadData+0xf
0012e58c 762cd13b USER32!CallHookWithSEH+0×21
0012e5ac 7739bffa MSVBVM50!VBDefControlProc_2787+0xad
0012e618 762d3348 USER32!CallHookWithSEH+0×21
0012e640 762cda44 MSVBVM50!PushCtlProc+0×2e
0012e668 762cd564 MSVBVM50!CommonGizWndProc+0×4e
0012e6b8 7739b6e3 MSVBVM50!StdCtlWndProc+0×171
0012e6e4 7739b874 USER32!InternalCallWinProc+0×28
0012e75c 7739ba92 USER32!UserCallWinProcCheckWow+0×151
0012e7c4 773a16e5 USER32!DispatchMessageWorker+0×327
0012e7d4 762d616e USER32!DispatchMessageA+0xf
0012e828 762d6054 MSVBVM50!ThunderMsgLoop+0×97
0012e874 762d5f55 MSVBVM50!SCM::FPushMessageLoop+0xaf
0012e8b4 004831f6 MSVBVM50!CMsoComponent::PushMsgLoop+0×24
0012e8c0 00d3b3c8 application+0×831f6
00184110 00000000 0xd3b3c8
```

```
0: kd> .thread /r /p 897228a0
Implicit thread is now 897228a0
Implicit process is now 897348a8
Loading User Symbols

0: kd> !thread 897228a0 1f 100
THREAD 897228a0  Cid 2984.2988  Teb: 7ffdf000 Win32Thread: bc381488
RUNNING on processor 3
IRP List:
    89794bb8: (0006,0220) Flags: 00000000  Mdl: 8a145878
Not impersonating
DeviceMap                 e3ec0360
Owning Process            897348a8       Image:         application2.exe
Wait Start TickCount      1569737        Ticks: 0
Context Switch Count      10239625                 LargeStack
UserTime                  02:38:18.890
KernelTime                00:29:36.187
Win32 Start Address application2 (0×00442e4c)
Start Address kernel32!BaseProcessStartThunk (0×77e617f8)
Stack Init f1d90000 Current f1d8fd00 Base f1d90000 Limit f1d88000 Call 0
Priority 24 BasePriority 24 PriorityDecrement 0
ChildEBP RetAddr
0012f66c 762d61bb USER32!_SEH_prolog+0×5
0012f678 00fdb0b9 MSVBVM50!rtcDoEvents+0×7
0012f92c 00fca760 application2+0xbdb0b9
0012fa20 762ce5a9 application2+0xbca760
0012fa40 762ce583 MSVBVM50!CallProcWithArgs+0×20
0012fa5c 762db781 MSVBVM50!InvokeVtblEvent+0×33
0012fa98 762cfbc2 MSVBVM50!InvokeEvent+0×32
0012fb78 762cfa4a MSVBVM50!EvtErrFireWorker+0×175
0012fb90 76330b2b MSVBVM50!EvtErrFire+0×18
0012fbf0 762cd13b MSVBVM50!ErrDefMouse_100+0×16d
0012fca4 762cda44 MSVBVM50!VBDefControlProc_2787+0xad
0012fccc 7631c826 MSVBVM50!CommonGizWndProc+0×4e
0012fd08 762cd523 MSVBVM50!StdCtlPreFilter_50+0×9e
0012fd5c 7739b6e3 MSVBVM50!StdCtlWndProc+0×130
0012fd88 7739b874 USER32!InternalCallWinProc+0×28
0012fe00 7739ba92 USER32!UserCallWinProcCheckWow+0×151
0012fe68 773a16e5 USER32!DispatchMessageWorker+0×327
0012fe78 762d616e USER32!DispatchMessageA+0xf
0012fea8 762bb78f MSVBVM50!ThunderMsgLoop+0×97
0012feb8 762d60cb MSVBVM50!MsoFInitPx+0×39
0012fecc 762d6054 MSVBVM50!CMsoCMHandler::FPushMessageLoop+0×1a
0012ff18 762d5f55 MSVBVM50!SCM::FPushMessageLoop+0xaf
0012ffa0 8082ea41 MSVBVM50!CMsoComponent::PushMsgLoop+0×24
0012fef8 762d5f8e nt!KiDeliverApc+0×11f
0012ff18 762d5f55 MSVBVM50!SCM_MsoCompMgr::FPushMessageLoop+0×2f
0012ffa0 8082ea41 MSVBVM50!CMsoComponent::PushMsgLoop+0×24
0012ff18 762d5f55 nt!KiDeliverApc+0×11f
0012ffa0 8082ea41 MSVBVM50!CMsoComponent::PushMsgLoop+0×24
0012ffd4 0012ffc8 nt!KiDeliverApc+0×11f
00000000 00000000 0×12ffc8
```

What we see here is unusually high Priority and BasePriority values (24 and 24). This means that the base priority for these processes was most likely artificially increased to Realtime. Most processes have base priority 8 (Normal):

```
0: kd> !thread 88780db0 1f
THREAD 88780db0  Cid 44a8.1b8c  Teb: 7ffaf000 Win32Thread: bc315d20 WAIT:
(Unknown) UserMode Non-Alertable
    887b8650  Semaphore Limit 0x7fffffff
    88780e28  NotificationTimer
Not impersonating
DeviceMap                 e1085298
Owning Process            889263a0       Image:         explorer.exe
Wait Start TickCount      1565543        Ticks: 4194 (0:00:01:05.531)
Context Switch Count      7                 LargeStack
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Start Address kernel32!BaseThreadStartThunk (0×77e617ec)
Stack Init b6754000 Current b6753c0c Base b6754000 Limit b6750000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0
ChildEBP RetAddr
b6753c24 80833e8d nt!KiSwapContext+0×26
b6753c50 80829b74 nt!KiSwapThread+0×2e5
b6753c98 809249cd nt!KeWaitForSingleObject+0×346
b6753d48 8088ac4c nt!NtReplyWaitReceivePortEx+0×521
b6753d48 7c8285ec nt!KiFastCallEntry+0xfc
01a2fe18 7c82783b ntdll!KiFastSystemCallRet
01a2fe1c 77c885ac ntdll!NtReplyWaitReceivePortEx+0xc
01a2ff84 77c88792 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0×198
01a2ff8c 77c8872d RPCRT4!RecvLotsaCallsWrapper+0xd
01a2ffac 77c7b110 RPCRT4!BaseCachedThreadRoutine+0×9d
01a2ffb8 77e64829 RPCRT4!ThreadStartRoutine+0×1b
01a2ffec 00000000 kernel32!BaseThreadStart+0×34
```

Some important system processes like csrss.exe have base priority 13 (High) but their threads wait most of the time and this doesn’t create any problems:

```
0: kd> !thread 887eb3d0 1f
THREAD 887eb3d0  Cid 4cf4.2bd4  Teb: 7ffaf000 Win32Thread: bc141cc0 WAIT:
(Unknown) UserMode Non-Alertable
    888769b0  SynchronizationEvent
Not impersonating
DeviceMap                 e1000930
Owning Process            8883f7c0       Image:         csrss.exe
Wait Start TickCount      1540456        Ticks: 29281 (0:00:07:37.515)
Context Switch Count      40                 LargeStack
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Start Address winsrv!ConsoleInputThread (0×75a81b18)
Stack Init b5c5a000 Current b5c59bac Base b5c5a000 Limit b5c55000 Call 0
Priority 15 BasePriority 13 PriorityDecrement 0
Kernel stack not resident.
ChildEBP RetAddr
b5c59bc4 80833e8d nt!KiSwapContext+0×26
b5c59bf0 80829b74 nt!KiSwapThread+0×2e5
b5c59c38 bf89b1c3 nt!KeWaitForSingleObject+0×346
b5c59c94 bf89b986 win32k!xxxSleepThread+0×1be
b5c59cec bf89da22 win32k!xxxRealInternalGetMessage+0×46a
b5c59d4c 8088ac4c win32k!NtUserGetMessage+0×3f
b5c59d4c 7c8285ec nt!KiFastCallEntry+0xfc
00ffff64 7739c811 ntdll!KiFastSystemCallRet
00ffff84 75a81c47 USER32!NtUserGetMessage+0xc
00fffff4 00000000 winsrv!ConsoleInputThread+0×16c
```

It is very unusual for a process to have Realtime base priority. We can speculate what had really happened before the system crash was forced. The system administrator noticed two applications consuming CPU over the long period of time and decided to intervene. Unfortunately the hand slipped when browsing Task Manager Set Priority menu and Realtime was selected instead of Low. Human error…

STACK OVERFLOW (USER MODE)

This is one of the simplest patterns to see in crash dumps. It has its own characteristic exception code and stack trace:

```
FAULTING_IP:
StackOverflow!SoFunction+27
00401317 6a00            push    0

EXCEPTION_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 00401300 (StackOverflow!SoFunction+0x00000010)
   ExceptionCode: c00000fd (Stack overflow)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 00000001
   Parameter[1]: 00082ffc

0:000> kL
ChildEBP RetAddr
00083000 00401317 StackOverflow!SoFunction+0x10
00083010 00401317 StackOverflow!SoFunction+0×27
00083020 00401317 StackOverflow!SoFunction+0×27
00083030 00401317 StackOverflow!SoFunction+0×27
00083040 00401317 StackOverflow!SoFunction+0×27
00083050 00401317 StackOverflow!SoFunction+0×27
00083060 00401317 StackOverflow!SoFunction+0×27
00083070 00401317 StackOverflow!SoFunction+0×27
00083080 00401317 StackOverflow!SoFunction+0×27
00083090 00401317 StackOverflow!SoFunction+0×27
000830a0 00401317 StackOverflow!SoFunction+0×27
000830b0 00401317 StackOverflow!SoFunction+0×27
000830c0 00401317 StackOverflow!SoFunction+0×27
000830d0 00401317 StackOverflow!SoFunction+0×27
000830e0 00401317 StackOverflow!SoFunction+0×27
000830f0 00401317 StackOverflow!SoFunction+0×27
00083100 00401317 StackOverflow!SoFunction+0×27
00083110 00401317 StackOverflow!SoFunction+0×27
00083120 00401317 StackOverflow!SoFunction+0×27
00083130 00401317 StackOverflow!SoFunction+0×27
```

There could be thousands of stack frames:

```
0:000> kL 2000
...
000a2fa0 00401317 StackOverflow!SoFunction+0x27
000a2fb0 00401317 StackOverflow!SoFunction+0x27
000a2fc0 00401317 StackOverflow!SoFunction+0x27
000a2fd0 00401317 StackOverflow!SoFunction+0x27
000a2fe0 00401317 StackOverflow!SoFunction+0x27
000a2ff0 00401317 StackOverflow!SoFunction+0x27
```

To reach the bottom and avoid over scrolling we can dump the raw stack data, search for the end of the repeating pattern of *StackOverflow!SoFunction+0×27* and try to manually reconstruct the bottom of the stack trace:

```
0:000> !teb
TEB at 7efdd000
    ExceptionList:        0017fdf0
    StackBase:            00180000
    StackLimit:           00081000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7efdd000
    EnvironmentPointer:   00000000
    ClientId:             00001dc4 . 00001b74
    RpcHandle:            00000000
    Tls Storage:          7efdd02c
    PEB Address:          7efde000
    LastErrorValue:       0
    LastStatusValue:      c0000034
    Count Owned Locks:    0
    HardErrorMode:        0

0:000> dds 00081000 00180000
...
0017fc74  00401317 StackOverflow!SoFunction+0×27
0017fc78  00000000
0017fc7c  a3a8ea65
0017fc80  0017fc90
0017fc84  00401317 StackOverflow!SoFunction+0×27
0017fc88  10001843
0017fc8c  a3a8ea95
0017fc90  0017fca0
0017fc94  00401317 StackOverflow!SoFunction+0×27
0017fc98  0017fcb8
0017fc9c  a3a8ea85
0017fca0  0017fcb0
0017fca4  00401317 StackOverflow!SoFunction+0×27
0017fca8  00000003
0017fcac  a3a8eab5
```

```
0017fcb0  0017fcc0
0017fcb4  00401317 StackOverflow!SoFunction+0×27
0017fcb8  76c68738 user32!_EndUserApiHook+0×11
0017fcbc  a3a8eaa5
0017fcc0  0017fcd0
0017fcc4  00401317 StackOverflow!SoFunction+0×27
0017fcc8  76c6a6cc user32!DefWindowProcW+0×94
0017fccc  a3a8ead5
0017fcd0  0017fce0
0017fcd4  00401317 StackOverflow!SoFunction+0×27
0017fcd8  0037311e
0017fcdc  a3a8eac5
0017fce0  0017fcf0
0017fce4  00401317 StackOverflow!SoFunction+0×27
0017fce8  0017fcd0
0017fcec  a3a8eaf5
0017fcf0  0017fd00
0017fcf4  00401317 StackOverflow!SoFunction+0×27
0017fcf8  76c6ad0f user32!NtUserBeginPaint+0×15
0017fcfc  a3a8eae5
0017fd00  0017fd5c
0017fd04  00401272 StackOverflow!WndProc+0xe2
0017fd08  00401190 StackOverflow!WndProc
0017fd0c  00000003
0017fd10  cf017ada
...
```

We use the extended version of **k** WinDbg command and supply EBP, ESP and EIP to see the function it started from:

```
0:000> r
eax=a3b739e5 ebx=00000000 ecx=ac430000 edx=ffefd944 esi=0037311e
edi=00000000
eip=00401300 esp=00082ff8 ebp=00083000 iopl=0  nv up ei ng nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b   efl=00010282
StackOverflow!SoFunction+0×10:
00401300 89442404  mov    dword ptr [esp+4],eax ss:002b:00082ffc=00000000

0:000> k L=0017fcf0 00082ff8 00401300
ChildEBP RetAddr
0017fcb0 00401317 StackOverflow!SoFunction+0×10
0017fd00 00401272 StackOverflow!SoFunction+0×27
0017fd5c 76c687af StackOverflow!WndProc+0xe2
0017fd88 76c68936 user32!InternalCallWinProc+0×23
0017fe00 76c6a571 user32!UserCallWinProcCheckWow+0×109
0017fe5c 76c6a5dd user32!DispatchClientMessage+0xe0
0017fe98 77ccee2e user32!__fnDWORD+0×2b
0017fedc 0040107d ntdll!KiUserCallbackDispatcher+0×2e
0017ff08 0040151e StackOverflow!wWinMain+0×7d
00402ba0 20245c8b StackOverflow!__tmainCRTStartup+0×176
```

We see that it started in WndProc.

MISSING COMPONENT (STATIC LINKAGE)

In previous **Missing Component** pattern (page 233) the example and emphasis was on dynamically loaded modules. Here we cover statically linked modules. Failure for a loader to find one of them results in a software exception. The most frequent of them are (numbers were taken from Google search):

```
C0000142 918
C0000143 919
C0000145 1,530
C0000135 24,900

0:001> !error c0000142
Error code: (NTSTATUS) 0xc0000142 (3221225794) - {DLL Initialization
Failed}  Initialization of the dynamic link library %hs failed. The
process is terminating abnormally.

0:001> !error c0000143
Error code: (NTSTATUS) 0xc0000143 (3221225795) - {Missing System
File}  The required system file %hs is bad or missing.

0:001> !error c0000145
Error code: (NTSTATUS) 0xc0000145 (3221225797) - {Application Error}  The
application failed to initialize properly (0x%lx). Click on OK to
terminate the application.

0:000> !error c0000135
Error code: (NTSTATUS) 0xc0000135 (3221225781) - {Unable To Locate
Component}  This application has failed to start because %hs was not
found. Re-installing the application may fix this problem.
```

We only consider user mode exceptions. If we have a default debugger configured it usually saves a crash dump. To model this problem one of applications was modified by changing all occurrences of KERNEL32.DLL to KERNEL32.DL using Visual Studio Binary Editor. CDB was configured as a default postmortem debugger (see **Custom Postmortem Debuggers on Vista**, Volume 1, page 618). When the application was launched CDB attached to it and saved a crash dump. If we open it in WinDbg we get characteristic **Special Stack Trace** (Volume 1, page 478) involving loader functions:

```
Loading Dump File [C:\UserDumps\CDAPatternMissingComponent.dmp]
User Mini Dump File with Full Memory: Only application data is available

Symbol search path is:
srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows Vista Version 6000 MP (2 procs) Free x86 compatible
Product: WinNt, suite: SingleUserTS
Debug session time: Thu Jun 12 12:03:28.000 2008 (GMT+1)
System Uptime: 1 days 8:46:23.167
Process Uptime: 0 days 0:00:48.000

This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(da4.f60): Wake debugger - code 80000007 (first/second chance not
available)
eax=00000000 ebx=77c4a174 ecx=75ce3cf9 edx=00000000 esi=7efde028
edi=7efdd000
eip=77bcf1d1 esp=0017fca4 ebp=0017fd00 iopl=0  nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b  efl=00000246
ntdll!_LdrpInitialize+0x6d:
77bcf1d1 8b45b8  mov   eax,dword ptr [ebp-48h] ss:002b:0017fcb8=7efde000

0:000> kL
ChildEBP RetAddr
0017fd00 77b937ea ntdll!_LdrpInitialize+0×6d
0017fd10 00000000 ntdll!LdrInitializeThunk+0×10
```

Verbose analysis command doesn’t give us an indication of what had happened so we need to dig further:

```
0:000> !analyze -v
...

FAULTING_IP:
+0
00000000 ??              ???

EXCEPTION_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 00000000
   ExceptionCode: 80000007 (Wake debugger)
  ExceptionFlags: 00000000
NumberParameters: 0

BUGCHECK_STR:  80000007

PROCESS_NAME:  StackOverflow.exe

ERROR_CODE: (NTSTATUS) 0x80000007 - {Kernel Debugger Awakened}  the system
debugger was awakened by an interrupt.

NTGLOBALFLAG:  400
```

```
APPLICATION_VERIFIER_FLAGS:  0

DERIVED_WAIT_CHAIN:

Dl Eid Cid     WaitType
-- --- ------- --------------------------
   0   da4.f60 Unknown

WAIT_CHAIN_COMMAND:  ~0s;k;;

BLOCKING_THREAD:  00000f60

DEFAULT_BUCKET_ID:  APPLICATION_HANG_BusyHang

PRIMARY_PROBLEM_CLASS:  APPLICATION_HANG_BusyHang

LAST_CONTROL_TRANSFER:  from 77b937ea to 77bcf1d1

FAULTING_THREAD:  00000000

STACK_TEXT:
0017fd00 77b937ea 0017fd24 77b60000 00000000 ntdll!_LdrpInitialize+0x6d
0017fd10 00000000 0017fd24 77b60000 00000000 ntdll!LdrInitializeThunk+0x10

FOLLOWUP_IP:
ntdll!_LdrpInitialize+6d
77bcf1d1 8b45b8          mov     eax,dword ptr [ebp-48h]

SYMBOL_STACK_INDEX:  0

SYMBOL_NAME:  ntdll!_LdrpInitialize+6d

FOLLOWUP_NAME:  MachineOwner

MODULE_NAME: ntdll

IMAGE_NAME:  ntdll.dll

DEBUG_FLR_IMAGE_TIMESTAMP:  4549bdf8

STACK_COMMAND:  ~0s ; kb

BUCKET_ID:  80000007_ntdll!_LdrpInitialize+6d

FAILURE_BUCKET_ID:  ntdll.dll!_LdrpInitialize_80000007_APPLICATION_HANG_Bu
syHang

Followup: MachineOwner
```

Last event and error code are not helpful too:

```
0:000> .lastevent
Last event: da4.f60: Wake debugger - code 80000007 (first/second chance
not available)
  debugger time: Thu Jun 12 15:04:38.917 2008 (GMT+1)

0:000> !gle
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0
```

Now we search thread raw stack data for any signs of exceptions:

```
0:000> !teb
TEB at 7efdd000
    ExceptionList:        0017fcf0
    StackBase:            00180000
    StackLimit:           0017e000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7efdd000
    EnvironmentPointer:   00000000
    ClientId:             00000da4 . 00000f60
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7efde000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0

0:000> dds 0017e000 00180000
...
0017f8d8  7efdd000
0017f8dc  0017f964
0017f8e0  77c11c78 ntdll!_except_handler4
0017f8e4  00000000
0017f8e8  0017f988
0017f8ec  0017f900
0017f8f0  77ba1ddd ntdll!RtlCallVectoredContinueHandlers+0x15
0017f8f4  0017f988
0017f8f8  0017f9d8
0017f8fc  77c40370 ntdll!RtlpCallbackEntryList
0017f900  0017f970
0017f904  77ba1db5 ntdll!RtlDispatchException+0×11f
0017f908  0017f988
0017f90c  0017f9d8
0017f910  7efde028
0017f914  00000001
0017f918  77630000 kernel32!_imp___aullrem <PERF> (kernel32+0×0)
0017f91c  00000001
```

```
0017f920  776ced81 kernel32!_DllMainCRTStartupForGS2+0×10
0017f924  0017f938
0017f928  7765d4d9 kernel32!BaseDllInitialize+0×18
0017f92c  76042340 user32!$$VProc_ImageExportDirectory
0017f930  00000001
0017f934  00000000
0017f938  0017f9e0
0017f93c  77b8f890 ntdll!LdrpSnapThunk+0xc9
0017f940  0040977a StackOverflow+0×977a
0017f944  0000030b
0017f948  76030000 user32!_imp__RegSetValueExW <PERF> (user32+0×0)
0017f94c  76042f94 user32!$$VProc_ImageExportDirectory+0xc54
0017f950  77bb8881 ntdll!LdrpSnapThunk+0×40d
0017f954  0017bb30
0017f958  00409770 StackOverflow+0×9770
0017f95c  00881a50
0017f960  004098b2 StackOverflow+0×98b2
0017f964  77bac282 ntdll!ZwRaiseException+0×12
0017f968  00180000
0017f96c  0017fc48
0017f970  0017fd00
0017f974  77bac282 ntdll!ZwRaiseException+0×12
0017f978  77b7ee72 ntdll!KiUserExceptionDispatcher+0×2a
0017f97c  0017f988 ; exception record
0017f980  0017f9d8 ; exception context
0017f984  00000000
0017f988  c0000135
0017f98c  00000001
0017f990  00000000
0017f994  77bcf1d1 ntdll!_LdrpInitialize+0×6d
0017f998  00000000
0017f99c  77c11c78 ntdll!_except_handler4
0017f9a0  77b8dab8 ntdll!RtlpRunTable+0×218
0017f9a4  fffffffe
0017f9a8  77ba2515 ntdll!vDbgPrintExWithPrefixInternal+0×214
0017f9ac  77ba253b ntdll!DbgPrintEx+0×1e
0017f9b0  77b7f356 ntdll! ?? ::FNODOBFM::`string’
0017f9b4  00000055
0017f9b8  00000003
0017f9bc  77b809c2 ntdll! ?? ::FNODOBFM::`string’
0017f9c0  0017fc9c
0017f9c4  00000001
0017f9c8  0017fd00
0017f9cc  77bcf28e ntdll!_LdrpInitialize+0×12a
0017f9d0  00000055
0017f9d4  75ce3cf9
0017f9d8  0001003f
0017f9dc  00000000
0017f9e0  00000000
0017f9e4  00000000
0017f9e8  00000000
0017f9ec  00000000
...
```

We see exception dispatching calls highlighted above (**Hidden Exception**, Volume 1, page 271). One of their parameters is an exception record and we try to get one:

```
0:000> .exr 0017f988
ExceptionAddress: 77bcf1d1 (ntdll!_LdrpInitialize+0x0000006d)
   ExceptionCode: c0000135
  ExceptionFlags: 00000001
NumberParameters: 0
```

Error c0000135 means that the loader was unable to locate a component. Now we try to examine the same raw stack data for any string patterns. For example, the following UNICODE pattern is clearly visible:

```
0017f2fc  00000000
0017f300  00880ec4
0017f304  77b910d7 ntdll!RtlpDosPathNameToRelativeNtPathName_Ustr+0x344
0017f308  00000000
0017f30c  43000043
0017f310  0042002a
0017f314  0017f33c
0017f318  00000000
0017f31c  00000002
0017f320  00000008
0017f324  00000000
0017f328  0000008c
0017f32c  000a0008
0017f330  77b91670 ntdll!`string'
0017f334  00b92bd6
0017f338  0017f5d4
0017f33c  003a0043
0017f340  0050005c
0017f344  006f0072
0017f348  00720067
0017f34c  006d0061
0017f350  00460020
0017f354  006c0069
0017f358  00730065
0017f35c  00280020
0017f360  00380078
0017f364  00290036
0017f368  0043005c
0017f36c  006d006f
0017f370  006f006d
0017f374  0020006e
0017f378  00690046
0017f37c  0065006c
0017f380  005c0073
0017f384  006f0052
0017f388  00690078
0017f38c  0020006f
0017f390  00680053
```

```
0017f394  00720061
0017f398  00640065
0017f39c  0044005c
0017f3a0  004c004c
0017f3a4  00680053
0017f3a8  00720061
0017f3ac  00640065
0017f3b0  004b005c
0017f3b4  00520045
0017f3b8  0045004e
0017f3bc  0033004c
0017f3c0  002e0032
0017f3c4  006c0064
0017f3c8  00000000
0017f3cc  00000000
```

It is a path to a DLL that was probably missing:

```
0:000> du 0017f33c
0017f33c  "C:\Program Files (x86)\Common Fi"
0017f37c  "les\Roxio Shared\DLLShared\KERNE"
0017f3bc  "L32.dl"
```

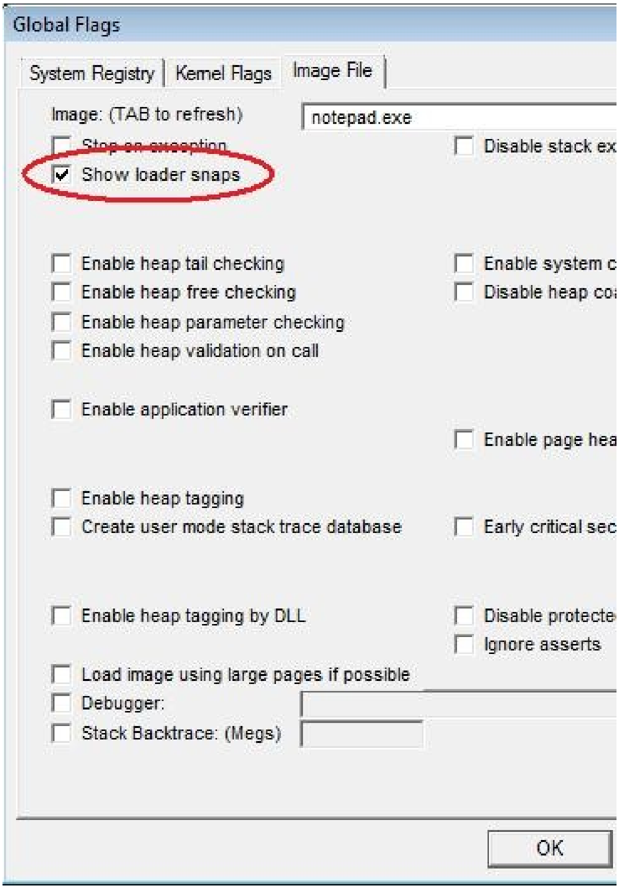
I think the loader was trying to find KERNEL32.dl following the DLL search order and this was the last path element:

```
0:000> !peb
PEB at 7efde000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00400000
    Ldr                       77c40080
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00881ad0 . 008831b8
    Ldr.InLoadOrderModuleList:           00881a50 . 00883dc8
    Ldr.InMemoryOrderModuleList:         00881a58 . 00883dd0
            Base TimeStamp                     Module
...
    Environment:  00881de8
...
Path=C:\Windows\system32; C:\Windows; C:\Windows\System32\Wbem; C:\Program
Files\ATI Technologies\ATI.ACE; c:\Program Files (x86)\Microsoft SQL
Server\90\Tools\binn\; C:\Program Files (x86)\Common Files\Roxio
Shared\DLLShared\
...
```

In similar situations **!dlls** command might help. It shows the load order (**-l** switch) and points to the last processed DLL:

```
0:001> !dlls -l

0x004740e8: C:\Program Files\Application\Application.exe
      Base   0x012a0000  EntryPoint  0x012b0903  Size        0x00057000
      Flags  0x00004010  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_ENTRY_PROCESSED

0x00474158: C:\Windows\SysWOW64\ntdll.dll
      Base   0x77d00000  EntryPoint  0x00000000  Size        0x00160000
      Flags  0x00004014  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED

0x00474440: C:\Windows\syswow64\kernel32.dll
      Base   0x77590000  EntryPoint  0x775a1f3e  Size        0x00110000
      Flags  0x00084014  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED
             LDRP_PROCESS_ATTACH_CALLED


      ...

0x00498ff8: C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_...\comctl32.dll
      Base   0x74d90000  EntryPoint  0x74dc43e5  Size        0x0019e000
      Flags  0x100c4014  LoadCount   0x00000003  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED
             LDRP_DONT_CALL_FOR_THREADS
             LDRP_PROCESS_ATTACH_CALLED

0x004991b8: C:\Windows\WinSxS\x86_microsoft.vc80.mfcloc_...\MFC80ENU.DLL
      Base   0x71b10000  EntryPoint  0x00000000  Size        0x0000e000
      Flags  0x10004014  LoadCount   0x00000001  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED
```

If it is difficult to identify what had really happened in crash dumps we can enable loader snaps using gflags and run the application under a debugger. For example, for notepad.exe we have:



Global Flags

System Registry | Kernel Flags | Image File

Image: (TAB to refresh) notepad.exe

□ Stop on exception
☑ Show loader snaps
□ Disable stack ex
□ Enable heap tail checking
□ Enable heap free checking
□ Enable heap parameter checking
□ Enable heap validation on call
□ Enable system c
□ Disable heap coa
□ Enable application verifier
□ Enable page hea
□ Enable heap tagging
□ Create user mode stack trace database
□ Early critical sec
□ Enable heap tagging by DLL
□ Disable protecte
□ Ignore asserts
□ Load image using large pages if possible
□ Debugger:
□ Stack Backtrace: (Megs)

```
Microsoft (R) Windows Debugger Version 6.8.0004.0 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: C:\Windows\notepad.exe
Symbol search path is:
srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00000000`ffac0000 00000000`ffaef000   notepad.exe
ModLoad: 00000000`779b0000 00000000`77b2a000   ntdll.dll
LDR: NEW PROCESS
     Image Path: C:\Windows\notepad.exe (notepad.exe)
     Current Directory: C:\Program Files\Debugging Tools for Windows 64-
bit\
     Search Path: C:\Windows; C:\Windows\system32; C:\Windows\system;
C:\Windows;.; C:\Program Files\Debugging Tools for Windows 64-
bit\winext\arcade; C:\Windows\system32;C:\Windows;
C:\Windows\System32\Wbem; C:\Program Files\ATI Technologies\ATI.ACE;
c:\Program Files (x86)\Microsoft SQL Server\90\Tools\binn\; C:\Program
Files (x86)\Common Files\Roxio Shared\DLLShared\
LDR: LdrLoadDll, loading kernel32.dll from
ModLoad: 00000000`777a0000
```

```
00000000`778d1000   C:\Windows\system32\kernel32.dll
LDR: kernel32.dll bound to ntdll.dll
LDR: kernel32.dll has stale binding to ntdll.dll
LDR: Stale Bind ntdll.dll from kernel32.dll
LDR: LdrGetProcedureAddress by NAME - BaseThreadInitThunk
[3d8,1278] LDR: Real INIT LIST for process C:\Windows\notepad.exe pid 984
0x3d8
[3d8,1278]    C:\Windows\system32\kernel32.dll init routine
00000000777DC960
[3d8,1278] LDR: kernel32.dll loaded - Calling init routine at
00000000777DC960
LDR: notepad.exe bound to ADVAPI32.dll
ModLoad: 000007fe`fe520000
000007fe`fe61f000   C:\Windows\system32\ADVAPI32.dll
LDR: ADVAPI32.dll bound to ntdll.dll
LDR: ADVAPI32.dll has stale binding to ntdll.dll
LDR: Stale Bind ntdll.dll from ADVAPI32.dll
LDR: ADVAPI32.dll bound to KERNEL32.dll
LDR: ADVAPI32.dll has stale binding to KERNEL32.dll
LDR: ADVAPI32.dll bound to ntdll.dll via forwarder(s) from kernel32.dll
LDR: ADVAPI32.dll has stale binding to ntdll.dll
LDR: Stale Bind KERNEL32.dll from ADVAPI32.dll
LDR: LdrGetProcedureAddress by NAME - RtlAllocateHeap
LDR: LdrGetProcedureAddress by NAME - RtlReAllocateHeap
LDR: LdrGetProcedureAddress by NAME - RtlEncodePointer
LDR: LdrGetProcedureAddress by NAME - RtlDecodePointer
LDR: LdrGetProcedureAddress by NAME - RtlSizeHeap
LDR: LdrGetProcedureAddress by NAME - RtlDeleteCriticalSection
LDR: LdrGetProcedureAddress by NAME - RtlEnterCriticalSection
LDR: LdrGetProcedureAddress by NAME - RtlLeaveCriticalSection
LDR: ADVAPI32.dll bound to RPCRT4.dll
...
```

This technique only works for native platform loader snaps. For example, it doesn’t show loader snaps for 32-bit modules loaded under WOW64:

```
Microsoft (R) Windows Debugger Version 6.8.0004.0 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: C:\Apps\StackOverflow.exe
Symbol search path is:
srv*c:\mss*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 00418000   StackOverflow.exe
ModLoad: 77b60000 77cb0000   ntdll.dll
LDR: NEW PROCESS
     Image Path: C:\Apps\StackOverflow.exe (StackOverflow.exe)
...
LDR: Loading (STATIC, NON_REDIRECTED) C:\Windows\system32\wow64cpu.dll
LDR: wow64cpu.dll bound to ntdll.dll
LDR: wow64cpu.dll has stale binding to ntdll.dll
LDR: Stale Bind ntdll.dll from wow64cpu.dll
```

```
LDR: wow64cpu.dll bound to wow64.dll
LDR: wow64cpu.dll has stale binding to wow64.dll
LDR: Stale Bind wow64.dll from wow64cpu.dll
LDR: wow64.dll has stale binding to wow64cpu.dll
LDR: Stale Bind wow64cpu.dll from wow64.dll
LDR: Refcount wow64cpu.dll (1)
LDR: Refcount wow64.dll (2)
LDR: Refcount wow64win.dll (1)
LDR: Refcount wow64.dll (3)
LDR: LdrGetProcedureAddress by NAME - Wow64LdrpInitialize
...
ModLoad: 77630000 77740000   C:\Windows\syswow64\kernel32.dll
ModLoad: 76030000 76100000   C:\Windows\syswow64\USER32.dll
ModLoad: 775a0000 77630000   C:\Windows\syswow64\GDI32.dll
ModLoad: 76d00000 76dbf000   C:\Windows\syswow64\ADVAPI32.dll
ModLoad: 76df0000 76ee0000   C:\Windows\syswow64\RPCRT4.dll
ModLoad: 75d60000 75dc0000   C:\Windows\syswow64\Secur32.dll
(1ec.1290): Unknown exception - code c0000135 (first chance)
(1ec.1290): Unknown exception - code c0000135 (!!! second chance !!!)
eax=00000000 ebx=77c4a174 ecx=75ce3cf9 edx=00000000 esi=7efde028
edi=7efdd000
eip=77bcf1d1 esp=0017fca4 ebp=0017fd00 iopl=0  nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b  efl=00000246
ntdll!_LdrpInitialize+0×6d:
77bcf1d1 8b45b8  mov    eax,dword ptr [ebp-48h] ss:002b:0017fcb8=7efde000
```

The dump file that we used can be downloaded from FTP to play with:

ftp://dumpanalysis.org/pub/CDAPatternMissingComponent.zip

DUPLICATED MODULE

In addition to **Module Variety** (Volume 1, page 310) this is another DLL Hell pattern. Here the same module is loaded at least twice and we can detect this when we see the module load address appended to its name in the output of **lm** commands (this is done to make the name of the module unique):

```
0:000> lm
start    end        module name
00b20000 0147f000   MSO_b20000
30000000 309a7000   EXCEL
30c90000 31848000   mso
71c20000 71c32000   tsappcmp
745e0000 7489e000   msi
76290000 762ad000   imm32
76b70000 76b7b000   psapi
76f50000 76f63000   secur32
77380000 77411000   user32
77670000 777a9000   ole32
77ba0000 77bfa000   msvcrt
77c00000 77c48000   gdi32
77c50000 77cef000   rpcrt4
77da0000 77df2000   shlwapi
77e40000 77f42000   kernel32
77f50000 77feb000   advapi32
7c800000 7c8c0000   ntdll
```

Usually his happens when the DLL is loaded from different locations. It can also be exactly the same DLL version. The problems usually surface when there are different DLL versions and the new code loads the old version of the DLL and uses it. This may result in interface incompatibility issues and ultimately in application fault like an access violation.

In order to provide a dump to play with I created a small toy program called 2DLLS to model the worst case scenario similar to the one I encountered in a production environment. The program periodically loads MyDLL module to call one of its functions. Unfortunately in one place it uses hardcoded relative path:

```
HMODULE hLib = LoadLibrary(L".\\DLL\\MyDLL.dll");
```

and in another place it relies on DLL search order (http://msdn.microsoft.com/en-us/library/ms682586.aspx):

```
hLib = LoadLibrary(L".\\MyDLL.dll");
```

PATH variable directories are used for search if this DLL was not found in other locations specified by DLL search order. We see that the problem can happen when another application is installed which uses the old version of that DLL and modifies the PATH variable to point to its location. To model interface incompatibility I compiled the version of MyDLL that causes NULL pointer access violation when the same function is called from it. The DLL was placed into a separate folder and the PATH variable was modified to reference that folder:

```
C:\>set PATH=C:\OLD;%PATH%
```

The application crashes and the installed default postmortem debugger (CDB, Volume 1, page 618) saves its crash dump. If we open it we would see that it crashed in MyDLL_1e60000 module which should trigger suspicion:

```
0:000> r
rax=0000000001e61010 rbx=0000000000000000 rcx=0000775dcac00000
rdx=0000000000000000 rsi=0000000000000006 rdi=0000000000001770
rip=0000000001e61010 rsp=000000000012fed8 rbp=0000000000000000
 r8=0000000000000000  r9=000000000012fd58 r10=0000000000000001
r11=000000000012fcc0 r12=0000000000000000 r13=0000000000000002
r14=0000000000000000 r15=0000000000000000
iopl=0 nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b  efl=00010200
MyDLL_1e60000!fnMyDLL:
00000000`01e61010 c704250000000000000000 mov dword ptr [0],0
ds:00000000`00000000=????????

0:000> kL
Child-SP          RetAddr           Call Site
00000000`0012fed8 00000001`40001093 MyDLL_1e60000!fnMyDLL
00000000`0012fee0 00000001`40001344 2DLLs+0×1093
00000000`0012ff10 00000000`773acdcd 2DLLs+0×1344
00000000`0012ff60 00000000`774fc6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0012ff90 00000000`00000000 ntdll!RtlUserThreadStart+0×1d
```

Looking at the list of modules we see two versions of MyDLL loaded from two different folders:

```
0:000> lm
start             end                 module name
00000000`01e60000 00000000`01e71000   MyDLL_1e60000
00000000`772a0000 00000000`7736a000   user32
00000000`77370000 00000000`774a1000   kernel32
00000000`774b0000 00000000`7762a000   ntdll
00000001`40000000 00000001`40010000   2DLLs
00000001`80000000 00000001`80011000   MyDLL
000007fe`fc9e0000 000007fe`fca32000   uxtheme
000007fe`fe870000 000007fe`fe9a9000   rpcrt4
000007fe`fe9b0000 000007fe`fe9bc000   lpk
000007fe`fea10000 000007fe`feae8000   oleaut32
000007fe`fecd0000 000007fe`fed6a000   usp10
000007fe`fedd0000 000007fe`fefb0000   ole32
000007fe`fefb0000 000007fe`ff0af000   advapi32
000007fe`ff0d0000 000007fe`ff131000   gdi32
000007fe`ff2e0000 000007fe`ff381000   msvcrt
000007fe`ff390000 000007fe`ff3b8000   imm32
000007fe`ff4b0000 000007fe`ff5b4000   msctf

0:000> lmv m MyDLL_1e60000
start             end                 module name
00000000`01e60000 00000000`01e71000   MyDLL_1e60000
    Loaded symbol image file: MyDLL.dll
    Image path: C:\OLD\MyDLL.dll
    Image name: MyDLL.dll
    Timestamp:        Wed Jun 18 14:49:13 2008 (48591259)
...

0:000> lmv m MyDLL
start             end                 module name
00000001`80000000 00000001`80011000   MyDLL
    Image path: C:\2DLLs\DLL\MyDLL.dll
    Image name: MyDLL.dll
    Timestamp:        Wed Jun 18 14:50:56 2008 (485912C0)
...
```

We can also see that the old version of MyDLL was the last loaded DLL:

```
0:000> !dlls -l

0x002c2680: C:\2DLLs\2DLLs.exe
      Base   0x140000000  EntryPoint  0x1400013b0  Size        0x00010000
      Flags  0x00004000  LoadCount   0x0000ffff  TlsIndex    0x00000000
             LDRP_ENTRY_PROCESSED

...
```

```
0x002ea9b0: C:\2DLLs\DLL\MyDLL.dll
      Base   0x180000000  EntryPoint  0x1800013d0  Size        0x00011000
      Flags  0x00084004  LoadCount   0x00000001  TlsIndex    0x00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED
             LDRP_PROCESS_ATTACH_CALLED

...

0x002ec430: C:\OLD\MyDLL.dll
      Base   0×01e60000  EntryPoint  0×01e613e0  Size        0×00011000
      Flags  0×00284004  LoadCount   0×00000001  TlsIndex    0×00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED
             LDRP_PROCESS_ATTACH_CALLED
             LDRP_IMAGE_NOT_AT_BASE
```

We can also see that the PATH variable points to its location and this might explain why it was loaded:

```
0:000> !peb
PEB at 000007fffffd6000
...
Path=C:\OLD;C:\Windows\system32;C:\Windows;...
...
```

We might think that the module having address in its name was loaded the last but this is not true. If we save another copy of the dump from the existing one using **.dump** command and load the new dump file we would see that order of the module names is reversed:

```
0:000> kL
Child-SP          RetAddr           Call Site
00000000`0012fed8 00000001`40001093 MyDLL!fnMyDLL
00000000`0012fee0 00000001`40001344 2DLLs+0×1093
00000000`0012ff10 00000000`773acdcd 2DLLs+0×1344
00000000`0012ff60 00000000`774fc6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0012ff90 00000000`00000000 ntdll!RtlUserThreadStart+0×1d
```

```
0:000> lm
start             end                 module name
00000000`01e60000 00000000`01e71000   MyDLL
00000000`772a0000 00000000`7736a000   user32
00000000`77370000 00000000`774a1000   kernel32
00000000`774b0000 00000000`7762a000   ntdll
00000001`40000000 00000001`40010000   2DLLs
00000001`80000000 00000001`80011000   MyDLL_180000000
000007fe`fc9e0000 000007fe`fca32000   uxtheme
000007fe`fe870000 000007fe`fe9a9000   rpcrt4
000007fe`fe9b0000 000007fe`fe9bc000   lpk
000007fe`fea10000 000007fe`feae8000   oleaut32
000007fe`fecd0000 000007fe`fed6a000   usp10
000007fe`fedd0000 000007fe`fefb0000   ole32
000007fe`fefb0000 000007fe`ff0af000   advapi32
000007fe`ff0d0000 000007fe`ff131000   gdi32
000007fe`ff2e0000 000007fe`ff381000   msvcrt
000007fe`ff390000 000007fe`ff3b8000   imm32
000007fe`ff4b0000 000007fe`ff5b4000   msctf

0:000> !dlls -l

...

0x002ec430: C:\OLD\MyDLL.dll
      Base   0×01e60000  EntryPoint  0×01e613e0  Size        0×00011000
      Flags  0×00284004  LoadCount   0×00000001  TlsIndex    0×00000000
             LDRP_IMAGE_DLL
             LDRP_ENTRY_PROCESSED
             LDRP_PROCESS_ATTACH_CALLED
             LDRP_IMAGE_NOT_AT_BASE
```

The postprocessed dump file used for this example can be downloaded from FTP to play with:

ftp://dumpanalysis.org/pub/CDAPatternDuplicatedModule.zip

NOT MY VERSION

This is another basic pattern of DLL Hell variety. It is when we look at component timestamps and paths and realize that that one of the modules from the production environment is older than we had during development and testing. The **lmft** WinDbg command will produce the necessary output. If there are many modules we might want to create a CAD graph (**Component Age Diagram**, page 126) to spot anomalies visually. Component version check is one of the basic troubleshooting and system administration activities that is illustrated in the forthcoming book "Crash Dump Analysis for System Administrators" (ISBN-13: 978-1-906717-02-5). Here is one example (module start and end load addresses are removed for visual clarity):

```
0:000> kL
Child-SP          RetAddr           Call Site
00000000`0012fed8 00000001`40001093 MyDLL!fnMyDLL
00000000`0012fee0 00000001`40001344 2DLLs+0×1093
00000000`0012ff10 00000000`773acdcd 2DLLs+0×1344
00000000`0012ff60 00000000`774fc6e1 kernel32!BaseThreadInitThunk+0xd
00000000`0012ff90 00000000`00000000 ntdll!RtlUserThreadStart+0×1d

0:000> lmft
module name
MyDLL    C:\OLD\MyDLL.dll Wed Jun 18 14:49:13 2004
user32   C:\Windows\System32\user32.dll Thu Feb 15 05:22:33 2007
kernel32 C:\Windows\System32\kernel32.dll Thu Nov 02 11:14:48 2006
ntdll    C:\Windows\System32\ntdll.dll Thu Nov 02 11:16:02 2006
2DLLs    C:\2DLLs\2DLLs.exe Thu Jun 19 10:46:44 2008 (485A2B04)
uxtheme  C:\Windows\System32\uxtheme.dll Thu Nov 02 11:15:07 2006
rpcrt4   C:\Windows\System32\rpcrt4.dll Tue Jul 17 05:21:15 2007
lpk      C:\Windows\System32\lpk.dll Thu Nov 02 11:12:33 2006
oleaut32 C:\Windows\System32\oleaut32.dll Thu Dec 06 05:09:35 2007
usp10    C:\Windows\System32\usp10.dll Thu Nov 02 11:15:03 2006
ole32    C:\Windows\System32\ole32.dll Thu Nov 02 11:14:31 2006
advapi32 C:\Windows\System32\advapi32.dll Thu Nov 02 11:11:35 2006
gdi32    C:\Windows\System32\gdi32.dll Thu Feb 21 04:40:51 2008
msvcrt   C:\Windows\System32\msvcrt.dll Thu Nov 02 11:13:37 2006
imm32    C:\Windows\System32\imm32.dll Thu Nov 02 11:13:15 2006
msctf    C:\Windows\System32\msctf.dll Thu Nov 02 11:13:42 2006
```

This pattern should be checked when we have instances of **Module Variety** (Volume 1, page 310) and, especially, **Duplicated Module** (page 294). Note that this pattern can also easily become an anti-pattern when applied to an unknown component: **Alien Component** (Volume 1, page 493).

DATA CONTENTS LOCALITY

This is a comparative pattern that helps not only in identifying the class of the problem but increases our confidence and degree of belief in the specific hypothesis. Suppose we have a database of notes of previous problems. If we see the same or similar data accessed in the new memory dump we might suppose that the issue is similar. If **Data Contents Locality** is complemented by **Code Path Locality** (similar partial stack traces and code residues, page **Error! Bookmark not defined.**) it even greater boosts our confidence in suggesting specific troubleshooting steps, recommending fixes and service packs or routing the problem to the next support or development service supply chain (like escalating the issue).

Suppose we got a new kernel memory dump with IRQL_NOT_LESS_OR_EQUAL (A) bugcheck pointing to our module and we notice the write access to a structure in nonpaged pool having specific pool tag:

```
3: kd> .trap 9ee8d9b0
ErrCode = 00000002
eax=85407650 ebx=858f6650 ecx=ffffffff edx=85407648 esi=858f65a8
edi=858f6620
eip=8083df4c esp=9ee8da24 ebp=9ee8da64 iopl=0 nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000 efl=00010246
nt!KeWaitForSingleObject+0x24f:
8083df4c 8919   mov     dword ptr [ecx],ebx  ds:0023:ffffffff=????????

STACK_TEXT:
9ee8d9b0 8083df4c badb0d00 85407648 00000000 nt!KiTrap0E+0x2a7
9ee8da64 80853f3f 85407648 0000001d 00000000
nt!KeWaitForSingleObject+0×24f
9ee8da7c 8081d45f 865b18d8 854076b0 f4b9e53b nt!KiAcquireFastMutex+0×13
9ee8da88 f4b9e53b 00000004 86940110 85407638 nt!ExAcquireFastMutex+0×20
9ee8daa8 f4b9ed98 85407638 00000000 86940110 driver!Query+0×143
...

3: kd> !pool 85407648
Pool page 85407648 region is Nonpaged pool
 85407000 size:   80 previous size:    0  (Allocated)  Mdl
 85407080 size:   30 previous size:   80  (Allocated)  Even (Protected)
 854070b0 size:   28 previous size:   30  (Allocated)  Ntfn
 854070d8 size:   28 previous size:   28  (Allocated)  NtFs
 85407100 size:   28 previous size:   28  (Allocated)  Ntfn
...
 85407570 size:   28 previous size:   70  (Allocated)  Ntfn
 85407598 size:   98 previous size:   28  (Allocated)  File (Protected)
*85407630 size:   b0 previous size:   98  (Free ) *DrvA
```

Dumping the memory address passed to KeWaitForSingleObject function shows simple but peculiar pattern:

```
3: kd> dd 85407648
85407648  ffffffff ffffffff ffffffff ffffffff
85407658  ffffffff ffffffff ffffffff ffffffff
85407668  ffffffff ffffffff ffffffff ffffffff
85407678  ffffffff ffffffff ffffffff ffffffff
85407688  ffffffff ffffffff ffffffff ffffffff
85407698  ffffffff ffffffff ffffffff ffffffff
854076a8  ffffffff ffffffff ffffffff ffffffff
854076b8  ffffffff ffffffff ffffffff ffffffff
```

We find several similar cases in our database but with different overall call stacks except the topmost wait call. Then we notice that in previous cases there were mutants associated with their thread structure and we have the same now:

```
0: kd> !thread
THREAD 858f65a8 Cid 474c.4530 Teb: 7ffdf000 Win32Thread: bc012410 RUNNING
on processor 0
...

3: kd> dt /r _KTHREAD 858f65a8 MutantListHead
nt!_KTHREAD
   +0×010 MutantListHead : _LIST_ENTRY [ 0×86773040 - 0×86773040 ]

3: kd> !pool 86773040
Pool page 86773040 region is Nonpaged pool
*86773000 size:   50 previous size:    0  (Allocated) *Muta (Protected)
  Pooltag Muta : Mutant objects
...
```

This narrows the issue to only a few previous cases. In one previous case WaitBlockList associated with a thread structure had 0xffffffff in its pointers. Our block shows the same pattern:

```
0: kd> dt -r _KTHREAD 858f65a8  WaitBlockList
nt!_KTHREAD
   +0×054 WaitBlockList : 0×858f6650 _KWAIT_BLOCK

0: kd> dt _KWAIT_BLOCK 0x858f6650
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x85407650 - 0xffffffff ]
   +0×008 Thread           : 0×858f65a8 _KTHREAD
   +0×00c Object           : 0×85407648
   +0×010 NextWaitBlock    : 0×858f6650 _KWAIT_BLOCK
   +0×014 WaitKey          : 0
   +0×016 WaitType         : 0×1 ”
   +0×017 SpareByte        : 0 ”
```

We have probably narrowed down the issue to a specific case. Although this doesn’t work always and mostly based on intuition there are spectacular cases where it really helps in troubleshooting. Here is another example where the contents of EDI register from exception context provided specific recommendation hints. When looking at the crash point we see an instance of **Wild Code** pattern (page 219):

```
0:000> kv
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
49ab5bba 00000000 00000000 00000000 00000000 0x60f1011a

0:000> r
eax=38084ff0 ebx=52303340 ecx=963f1416 edx=0000063d esi=baaff395
edi=678c5804
eip=60f1011a esp=5a9d0f48 ebp=49ab5bba iopl=0  nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000  efl=00210206
60f1011a cd01            int     1

0:000> u
60f1011a cd01            int     1
60f1011c cc              int     3
60f1011d 8d              ???
60f1011e c0eb02          shr     bl,2
60f10121 0f840f31cd01    je      62be3236
60f10127 8d              ???
60f10128 c0cc0f          ror     ah,0Fh
60f1012b 0bce            or      ecx,esi
```

Looking at raw stack data we notice the presence of a specific component that is known to patch the process import table. Applying techniques outlined in **Hooked Functions** pattern (Volume 1, page 468) we notice two different 3rd-party components that patched two different modules (kernel32 and user32):

```
0:000> !chkimg -lo 50 -d !kernel32 -v
Searching for module with expression: !kernel32
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: c:\mss\kernel32.dll\4626487F102000\kernel32.dll
No range specified

Scanning section:    .text
Size: 564709
Range to scan: 77e41000-77ecade5
    77e41ae5-77e41ae9  5 bytes - kernel32!LoadLibraryExW
 [ 6a 34 68 48 7b:e9 16 e5 f4 07 ]
    77e44a8a-77e44a8e  5 bytes - kernel32!WaitNamedPipeW (+0×2fa5)
 [ 8b ff 55 8b ec:e9 71 b5 f9 07 ]
    77e5106a-77e5106e  5 bytes - kernel32!CreateProcessInternalW (+0xc5e0)
```

```
...
Total bytes compared: 564709(100%)
Number of errors: 49
49 errors : !kernel32 (77e41ae5-77e9aa16)

0:000> u 77e41ae5
kernel32!LoadLibraryExW:
77e41ae5 jmp     7fd90000
77e41aea out     77h,al
77e41aec call    kernel32!_SEH_prolog (77e6b779)
77e41af1 xor     edi,edi
77e41af3 mov     dword ptr [ebp-28h],edi
77e41af6 mov     dword ptr [ebp-2Ch],edi
77e41af9 mov     dword ptr [ebp-20h],edi
77e41afc cmp     dword ptr [ebp+8],edi

0:000> u 7fd90000
*** ERROR: Symbol file could not be found.  Defaulted to export symbols
for ComponentA.dll -
7fd90000 jmp     ComponentA!DllUnregisterServer+0×2700 (678c4280)
7fd90005 push    34h
7fd90007 push    offset kernel32!`string’+0xc (77e67b48)
7fd9000c jmp     kernel32!LoadLibraryExW+0×7 (77e41aec)
7fd90011 add     byte ptr [eax],al
7fd90013 add     byte ptr [eax],al
7fd90015 add     byte ptr [eax],al
7fd90017 add     byte ptr [eax],al

0:000> !chkimg -lo 50 -d !user32 -v
Searching for module with expression: !user32
Will apply relocation fixups to file used for comparison
Will ignore NOP/LOCK errors
Will ignore patched instructions
Image specific ignores will be applied
Comparison image path: c:\mss\user32.dll\45E7BFD692000\user32.dll
No range specified

Scanning section:    .text
Size: 396943
Range to scan: 77381000-773e1e8f
    77383f38-77383f3c  5 bytes - user32!EnumDisplayDevicesW
 [ 8b ff 55 8b ec:e9 c3 c0 82 08 ]
    77384406-7738440a  5 bytes - user32!EnumDisplaySettingsExW (+0×4ce)
 [ 8b ff 55 8b ec:e9 f5 bb 7e 08 ]
    773844d9-773844dd  5 bytes - user32!EnumDisplaySettingsW (+0xd3)
 [ 8b ff 55 8b ec:e9 22 bb 80 08 ]
    7738619b-7738619f  5 bytes - user32!EnumDisplayDevicesA (+0×1cc2)
 [ 8b ff 55 8b ec:e9 60 9e 83 08 ]
    7738e985-7738e989  5 bytes - user32!CreateWindowExA (+0×87ea)
 [ 8b ff 55 8b ec:e9 76 16 8c 08 ]
...
Total bytes compared: 396943(100%)
Number of errors: 119
119 errors : !user32 (77383f38-773c960c)
```

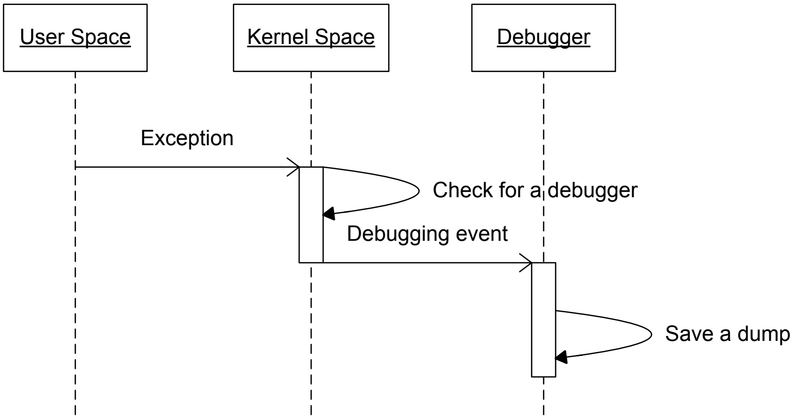
```
0:000> u 77383f38
user32!EnumDisplayDevicesW:
77383f38 e9c3c08208      jmp     7fbb0000
77383f3d 81ec58030000    sub     esp,358h
77383f43 a1ac243e77      mov     eax,dword ptr [user32!__security_cookie
(773e24ac)]
77383f48 8b5508          mov     edx,dword ptr [ebp+8]
77383f4b 83a5acfcffff00  and     dword ptr [ebp-354h],0
77383f52 53              push    ebx
77383f53 56              push    esi
77383f54 8b7510          mov     esi,dword ptr [ebp+10h]

0:000> u 7fbb0000
*** ERROR: Symbol file could not be found.  Defaulted to export symbols
for ComponentB.dll -
7fbb0000 e91b43d5e5      jmp     ComponentB+0×4320 (65904320)
7fbb0005 8bff            mov     edi,edi
7fbb0007 55              push    ebp
7fbb0008 8bec            mov     ebp,esp
7fbb000a e92e3f7df7      jmp     user32!EnumDisplayDevicesW+0×5 (77383f3d)
7fbb000f 0000            add     byte ptr [eax],al
7fbb0011 0000            add     byte ptr [eax],al
7fbb0013 0000            add     byte ptr [eax],al
```

Which one should we try to eliminate first to test our assumption that they somehow resulted in application faults? Looking at register context again we see that one specific register (EDI) has a value that lies in ComponentA address range:

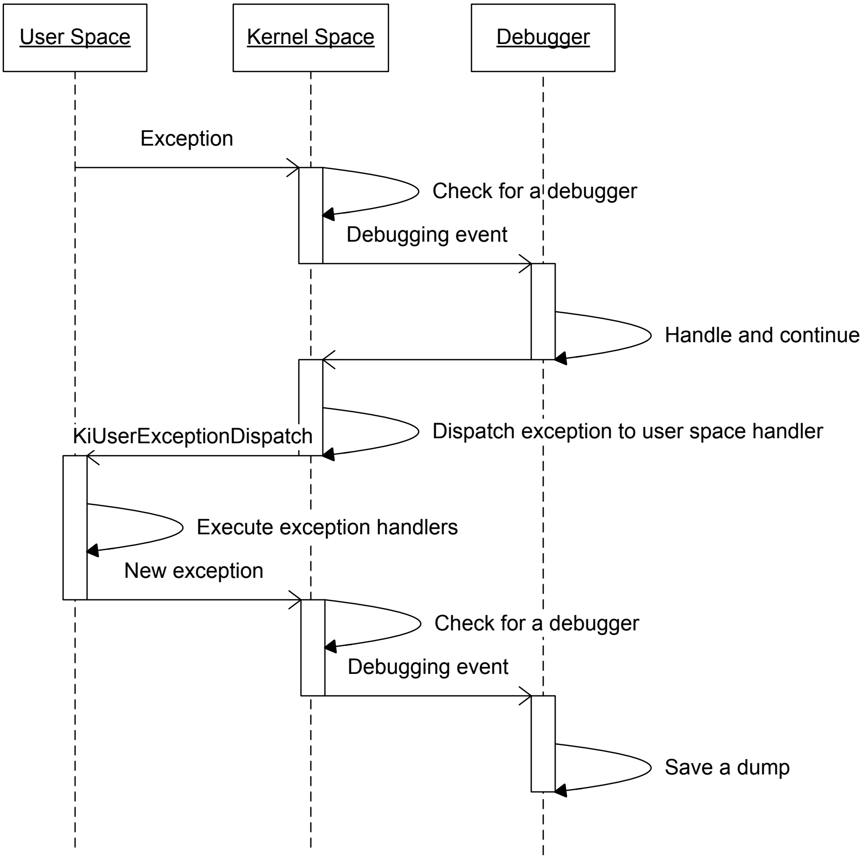
```
0:000> r
eax=38084ff0 ebx=52303340 ecx=963f1416 edx=0000063d esi=baaff395
edi=678c5804
eip=60f1011a esp=5a9d0f48 ebp=49ab5bba iopl=0  nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000 efl=00210206
60f1011a cd01            int     1

0:000> lm
start    end        module name
00400000 01901000   Application
...
678c0000 6791d000   ComponentA   ComponentA.DLL
...
```

NESTED EXCEPTIONS (UNMANAGED CODE)

In the case of a first-chance exception it is not possible to see it in a process crash dump because the entire exception processing was done in the kernel space (see **How to Distinguish Between 1st and 2nd Chances**, Volume 1, page 109):



However the picture changes when we have nested exceptions**.** In this case we should expect traces of inner exception processing like exception dispatcher code or exception handlers to be present on a raw stack dump:



Consider the following C++ code with two exception handlers:

```
__try
{
  __try
  {
    *(int *)NULL = 0;  // Exception 1
                       // Dump1 1st chance
  }
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
    std::cout << "Inner" << std::endl;
    *(int *)NULL = 0;  // Exception 2
                       // Dump2 1st chance
  }
}
```

```
 __except (EXCEPTION_EXECUTE_HANDLER)
 {
   std::cout << "Outer" << std::endl;
   *(int *)NULL = 0;     // Exception 3
                         // Dump3 1st chance
                         // Dump4 2nd chance
 }
```

If we run the actual program and we have set a default postmortem debugger (Volume 1, page 618) we get a second-chance exception dump (Dump4). The program first outputs "Inner" and then "Outer" on a console and then crashes. When we look at the dump we see second-chance exception processing code where the exception record for NtRaiseException is the same and points to Exception 3 context (shown in bold underlined):

```
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(11dc.f94): Access violation - code c0000005 (first/second chance not
available)
*** ERROR: Module load completed but symbols could not be loaded for
NestedException.exe
NestedException+0x1a72:
00000001`40001a72 c704250000000000000000 mov dword ptr [0],0
ds:00000000`00000000=????????

0:000> !teb
TEB at 000007fffffde000
    ExceptionList:        0000000000000000
    StackBase:            0000000000130000
    StackLimit:           000000000012d000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 000007fffffde000
    EnvironmentPointer:   0000000000000000
    ClientId:             00000000000011dc . 0000000000000f94
    RpcHandle:            0000000000000000
    Tls Storage:          000007fffffde058
    PEB Address:          000007fffffd5000
    LastErrorValue:       0
    LastStatusValue:      c000000d
    Count Owned Locks:    0
    HardErrorMode:        0
```

```
0:000> dqs 000000000012d000 0000000000130000
...
00000000`0012f918  00000000`00000006
00000000`0012f920  00000000`00000000
00000000`0012f928  00000000`775a208d ntdll!KiUserExceptionDispatch+0×53
00000000`0012f930  00000000`00000000
00000000`0012f938  00000000`0012f930 ; exception context
00000000`0012f940  01c8d5f0`00000000
00000000`0012f948  00000000`00000000
...

0:000> ub ntdll!KiUserExceptionDispatch+0x53
ntdll!KiUserExceptionDispatch+0x35:
00000000`775a206f xor     edx,edx
00000000`775a2071 call    ntdll!RtlRestoreContext (00000000`775a2255)
00000000`775a2076 jmp     ntdll!KiUserExceptionDispatch+0x53
(00000000`775a208d)
00000000`775a2078 mov     rcx,rsp
00000000`775a207b add     rcx,4D0h
00000000`775a2082 mov     rdx,rsp
00000000`775a2085 xor     r8b,r8b
00000000`775a2088 call    ntdll!NtRaiseException (00000000`775a1550)

0:000> .cxr 00000000`0012f930
rax=00000001400223d0 rbx=0000000000000000 rcx=0000000140022128
rdx=0000000000000001 rsi=0000000000000006 rdi=0000000140022120
rip=0000000140001a72 rsp=000000000012fed0 rbp=0000000000000000
 r8=000007fffffde000  r9=0000000000000001 r10=0000000000000000
r11=0000000000000246 r12=0000000000000000 r13=0000000000000002
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr ac po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010
256
NestedException+0x1a72:
00000001`40001a72 c704250000000000000000 mov dword ptr [0],0
ds:00000000`00000000=????????
```

However if we have first-chance exception Dump3 from some exception monitoring program (Volume 1, page 465) we see that NtRaiseException parameter points to ”Inner” Exception 2 context (a different and earlier address, shown in italics underlined):

```
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(11dc.f94): Access violation - code c0000005 (first/second chance not
available)
*** ERROR: Module load completed but symbols could not be loaded for
NestedException.exe
NestedException+0x1a72:
00000001`40001a72 c704250000000000000000 mov dword ptr [0],0
ds:00000000`00000000=????????

0:000> dqs 000000000012d000 0000000000130000
...
00000000`0012f918  00000000`00000006
00000000`0012f920  00000000`00000000
00000000`0012f928  00000000`775a2068 ntdll!KiUserExceptionDispatch+0×2e
00000000`0012f930  00000000`00000000
00000000`0012f938  00000000`0012f930 ; exception context
00000000`0012f940  01c8d5f0`00000000
...

0:000>  .cxr 00000000`0012f930
rax=00000001400223d0 rbx=0000000000000000 rcx=0000000140022128
rdx=0000000000000001 rsi=0000000000000006 rdi=0000000140022120
rip=00000001400019fa rsp=000000000012fed0 rbp=0000000000000000
 r8=000007fffffde000  r9=0000000000000001 r10=0000000000000000
r11=0000000000000246 r12=0000000000000000 r13=0000000000000002
r14=0000000000000000 r15=0000000000000000
iopl=0 nv up ei pl zr ac po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b  efl=00010256
NestedException+0x19fa:
00000001`400019fa c704250000000000000000 mov dword ptr [0],0
ds:00000000`00000000=????????
```

Similar can be said about Dump2 where NtRaiseException parameter points to Exception 1 context. But Dump1 doesn't have any traces of exception processing as expected.

All 4 dump files can be downloaded from FTP to play with from the following location:

ftp://dumpanalysis.org/pub/CDAPatternNestedExceptions.zip

NESTED EXCEPTIONS (MANAGED CODE)

Nested exception analysis is much simpler in managed code than in unmanaged (page 305). Exception object references the inner exception if any and so on:

Exception.InnerException
http://msdn.microsoft.com/en-us/library/system.exception.innerexception.aspx

WinDbg does a good job of traversing all nested exceptions when executing **!analyze -v** command. In the following example of a Windows forms application crash, ObjectDisposedException (shown in bold) was re-thrown as Exception object with “Critical error” message (shown in bold italics) which was re-thrown several times as Exception object with “Critical program error” message (shown in bold underlined) that finally resulted in the process termination request from the top level exception handler:

```
MANAGED_STACK:
(TransitionMU)
001374B0 0B313757 System!System.Diagnostics.Process.Kill()+0x37
001374E4 0B3129C7
Component!Foo.HandleUnhandledException(System.Exception)+0x137
001374F4 07C0A7D3
Component!Foo+FooBarProcessMenuCommand(System.String)+0x33
(TransitionUM)
(TransitionMU)
...

EXCEPTION OBJECT: !pe 3398614
Exception object: 03398614
Exception type: System.Exception
Message: Critical program error
InnerException: System.Exception, use !PrintException 03398560 to see more
StackTrace (generated):
    SP       IP       Function
    001371A8 07C0CDD8 Foo.BarUserInteraction.Process(System.String)
    00137258 07C0CCA6 Foo.BarUserInteraction.ProcessUserInteraction(Sub,
BarStepType)
    00137268 07C0A9BA Foo.BarMenu.Process(CMD)
    00137544 07C0A8D8 Foo.BarMenu.ProcessCMD(CMD)
    0013756C 07C0A7BE Foo+FooBar.ProcessBarMenuCommand(System.String)

StackTraceString: <none>
HResult: 80131500
```

```
EXCEPTION OBJECT: !pe 3398560
Exception object: 03398560
Exception type: System.Exception
Message: Critical program error
InnerException: System.Exception, use !PrintException 033984ac to see more
StackTrace (generated):
    SP       IP       Function
    00137154 07C0D4CA Foo.BarThreads+ProcessOpenQuery.Execute()
    00137218 07C0D3B3 Foo.BarMenu.ProcessQuery()
    00137220 07C0CCF3 Foo.BarUserInteraction.Process(System.String)

StackTraceString: <none>
HResult: 80131500

EXCEPTION OBJECT: !pe 33984ac
Exception object: 033984ac
Exception type: System.Exception
Message: Critical program error
InnerException: System.Exception, use !PrintException 033983ec to see more
StackTrace (generated):
    SP       IP       Function
    0013704C 0A6149DD Foo.Bar.OpenQueryThreaded(Foo.BarParameter)
    00137154 0A6140D0 Foo.BarThreads+ProcessParameter.Execute()
...

StackTraceString: <none>
HResult: 80131500

EXCEPTION OBJECT: !pe 33983ec
Exception object: 033983ec
Exception type: System.Exception
Message: Critical program error
InnerException: System.Exception, use !PrintException 033982fc to see more
StackTrace (generated):
    SP       IP       Function
    00137008 0ACA59F1 Foo.BarApplication.Refresh(Boolean, Boolean)
    001370C4 0A6144E0 Foo.Bar.OpenQueryThreaded(Foo.BarParameter)

StackTraceString: <none>
HResult: 80131500

EXCEPTION OBJECT: !pe 33982fc
Exception object: 033982fc
Exception type: System.Exception
Message: Critical program error
InnerException: System.Exception, use !PrintException 03398260 to see more
StackTrace (generated):
    SP       IP       Function
    00136F3C 0AE24983 
Foo.BarDisplay.ShowVariableScreen(Foo.variables.BarVariables)
    00136FDC 0AE204F6 Foo.variables.BarVariables.ShowVariableScreen()
    00137070 0ACAFE1D Foo.BarApplication.ShowVariableScreen(Boolean)
    00137080 0ACA5977 Foo.BarApplication.Refresh(Boolean, Boolean)
```

```
StackTraceString: <none>
HResult: 80131500

EXCEPTION_OBJECT: !pe 3398260
Exception object: 03398260
Exception type: System.Exception
Message: Critical error
InnerException: System.ObjectDisposedException, use !PrintException
03397db8 to see more StackTrace (generated):
    SP       IP       Function
    00136FB4 0AE24905
Foo.BarDisplay.ShowVariableScreen(Foo.variables.BarVariables)

StackTraceString: <none>
HResult: 80131500

EXCEPTION_OBJECT: !pe 3397db8
Exception object: 03397db8
Exception type: System.ObjectDisposedException
Message: Cannot access a disposed object.
InnerException: <none>
StackTrace (generated):
    SP       IP       Function
    00136258 06D36158 System.Windows.Forms.Control.CreateHandle()
    001362B8 06D38F96 System.Windows.Forms.Control.get_Handle()
    001362C4 0B0C8C68
System.Windows.Forms.Control.PointToScreen(System.Drawing.Point)
    001362F0 0B0CECB4
System.Windows.Forms.Button.OnMouseUp(System.Windows.Forms.MouseEventArgs)
    00136314 0B0C8BB7
System.Windows.Forms.Control.WmMouseUp(System.Windows.Forms.Message ByRef,
System.Windows.Forms.MouseButtons, Int32)
    00136384 06D385A0
System.Windows.Forms.Control.WndProc(System.Windows.Forms.Message ByRef)
    001363E8 0A69C73E
System.Windows.Forms.ButtonBase.WndProc(System.Windows.Forms.Message
ByRef)
    00136424 0A69C54D
System.Windows.Forms.Button.WndProc(System.Windows.Forms.Message ByRef)
    0013642C 06D37FAD
System.Windows.Forms.Control+ControlNativeWindow.OnMessage(System.Windows.
Forms.Message ByRef)
    00136430 06D37F87
System.Windows.Forms.Control+ControlNativeWindow.WndProc(System.Windows.Fo
rms.Message ByRef)
    00136440 06D37D9F System.Windows.Forms.NativeWindow.Callback(IntPtr,
Int32, IntPtr, IntPtr)

StackTraceString: <none>
HResult: 80131622
```

```
EXCEPTION_MESSAGE:  Cannot access a disposed object.

STACK_TEXT:
00137448 7c827c1b ntdll!KiFastSystemCallRet
0013744c 77e4201b ntdll!NtTerminateProcess+0xc
0013745c 05d78202 kernel32!TerminateProcess+0x20
...
```

AFFINE THREAD

Setting a thread affinity mask to a specific processor or core makes that thread running in a single processor environment from that thread point of view. It is always scheduled to run on that processor. This potentially creates a problem found on real single processor environments if the processor runs another higher priority thread (**Thread Starvation** pattern, page 274) or loops at dispatch level IRQL (**Dispatch Level Spin** pattern, page 154).

Here is one example. A dual core laptop was hanging and kernel memory dump revealed the following **Wait Chain** pattern (page 147):

```
Resource @ nt!PopPolicyLock (0x80563080)    Exclusively owned
    Contention Count = 32
    NumberOfExclusiveWaiters = 9
     Threads: 8b3b08b8-01<*>
     Threads Waiting On Exclusive Access:
              872935f0       8744cb30       87535da8       8755a6b0
              8588dba8       8a446c10       85891c50       87250020
              8a6e7da8
```

The thread 8b3b08b8 blocked other 9 threads and had the following stack trace:

```
0: kd> !thread 8b3b08b8 1f
THREAD 8b3b08b8  Cid 0004.002c  Teb: 00000000 Win32Thread: 00000000 READY
Not impersonating
DeviceMap                 e1009248
Owning Process            8b3b2830       Image:         System
Wait Start TickCount      44419          Ticks: 8744 (0:00:02:16.625)
Context Switch Count      4579
UserTime                  00:00:00.000
KernelTime                00:00:01.109
Start Address nt!ExpWorkerThread (0x8053867e)
Stack Init bad00000 Current bacffcb0 Base bad00000 Limit bacfd000 Call 0
Priority 15 BasePriority 12 PriorityDecrement 3 DecrementCount 16
ChildEBP RetAddr
bacffcc8 804fd2c9 nt!KiUnlockDispatcherDatabase+0x9e
bacffcdc 8052a16f nt!KeSetSystemAffinityThread+0×5b
bacffd04 805caf03 nt!PopCompositeBatteryUpdateThrottleLimit+0×2d
bacffd24 805ca767 nt!PopCompositeBatteryDeviceHandler+0×1c5
bacffd3c 80529d3b nt!PopPolicyWorkerMain+0×25
bacffd7c 8053876d nt!PopPolicyWorkerThread+0xbf
bacffdac 805cff64 nt!ExpWorkerThread+0xef
bacffddc 805460de nt!PspSystemThreadStartup+0×34
00000000 00000000 nt!KiThreadStartup+0×16
```

Note this function and its first parameter (shown in smaller font for visual clarity):

```
0: kd> !thread 8b3b08b8
...
bacffcdc 8052a16f 00000002 8a5b8cd8 00000030 nt!KeSetSystemAffinityThread+0×5b
...
```

The first parameter is KAFFINITY mask and 0×2 is 0y10 (binary) which is the second core. This thread had been already set to run on that core only:

```
0: kd> dt _KTHREAD 8b3b08b8
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
...
   +0×124 Affinity         : 2
...
```

Let’s look at our second core:

```
0: kd> ~1s

1: kd> kL 100
ChildEBP RetAddr
a8f00618 acd21947 hal!KeAcquireInStackQueuedSpinLock+0x43
a8f00618 acd21947 tcpip!IndicateData+0x98
a8f00684 acd173e5 tcpip!IndicateData+0x98
a8f0070c acd14ef5 tcpip!TCPRcv+0xbb0
a8f0076c acd14b19 tcpip!DeliverToUser+0x18e
a8f007e8 acd14836 tcpip!DeliverToUserEx+0x95e
a8f008a0 acd13928 tcpip!IPRcvPacket+0x6cb
a8f008e0 acd13853 tcpip!ARPRcvIndicationNew+0x149
a8f0091c ba56be45 tcpip!ARPRcvPacket+0x68
a8f00970 b635801d NDIS!ethFilterDprIndicateReceivePacket+0x307
a8f00984 b63581b4 psched!PsFlushReceiveQueue+0x15
a8f009a8 b63585f9 psched!PsEnqueueReceivePacket+0xda
a8f009c0 ba56c8ed psched!ClReceiveComplete+0x13
a8f009d8 b7defdb5 NDIS!EthFilterDprIndicateReceiveComplete+0x7c
a8f00a08 b7df0f78 driverA+0x17db5
a8f00a64 ba55ec2c driverA+0x18f78
a8f00a88 b6b0962c NDIS!ndisMSendCompleteX+0x8d
a8f00a9c b6b0a36d driverB+0x62c
a8f00ab8 ba55e88f driverB+0x136d
a8f00ae0 b7de003c NDIS!NdisReturnPackets+0xe9
a8f00af0 ba55e88f driverA+0x803c
a8f00b18 b6358061 NDIS!NdisReturnPackets+0xe9
a8f00b30 ba55e88f psched!MpReturnPacket+0x3b
a8f00b58 acc877cc NDIS!NdisReturnPackets+0xe9
87749da0 00000000 afd!AfdReturnBuffer+0xe1
```

```
1: kd> r
eax=a8f005f8 ebx=a8f00624 ecx=8a9862ed edx=a8f00b94 esi=874e2ed0
edi=8a9862d0
eip=806e6a33 esp=a8f005ec ebp=a8f00618 iopl=0 nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000  efl=00000202
hal!KeAcquireInStackQueuedSpinLock+0x43:
806e6a33 74ee je hal!KeAcquireInStackQueuedSpinLock+0x33 (806e6a23) [br=0]

1: kd> !running

System Processors 3 (affinity mask)
  Idle Processors 1

     Prcb      Current   Next
  1  bab38120  8a0c8ae8  8b3a7318  …………….
```

We see the thread 8a0c8ae8 had been spinning on the second core for more than 2 minutes:

```
1: kd> !thread 8a0c8ae8 1f
THREAD 8a0c8ae8  Cid 0660.0124  Teb: 7ffd7000 Win32Thread: e338c498
RUNNING on processor 1
IRP List:
    8a960008: (0006,01b4) Flags: 00000900  Mdl: 87535908
Not impersonating
DeviceMap                 e2f155b8
Owning Process            87373020       Image:         APPLICATION.EXE
Wait Start TickCount      43918          Ticks: 9245 (0:00:02:24.453)
Context Switch Count      690                 LargeStack
UserTime                  00:00:00.000
KernelTime                00:02:24.453
...
```

Its kernel time looks consistent with the starved thread waiting time:

```
0: kd> !thread 8b3b08b8 1f
THREAD 8b3b08b8  Cid 0004.002c  Teb: 00000000 Win32Thread: 00000000 READY
Not impersonating
DeviceMap                 e1009248
Owning Process            8b3b2830       Image:         System
Wait Start TickCount      44419          Ticks: 8744 (0:00:02:16.625)
...
```

For comparison, the spinning thread has affinity mask 0y11 (0×3) which means that it could be scheduled to run on both cores:

```
0: kd> dt _KTHREAD 8a0c8ae8
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
...
   +0×124 Affinity         : 3
...
```

SELF-DIAGNOSIS

Sometimes patterns like **Message Box** (page 177) and / or **Stack Trace** semantics (Volume 1, page 395) reveal another pattern that I call **Self-Diagnosis** which may or may not result in **Self-Dump** (page 181). The diagnostic message may reveal the problem internally detected by runtime environment.

Consider the following stack trace (shown in smaller font for visual clarity):

```
0:000> kv
ChildEBP RetAddr  Args to Child
0012e8c0 77f4bf53 77f4610a 00000000 00000000 ntdll!KiFastSystemCallRet
0012e8f8 77f3965e 000101a2 00000000 00000001 user32!NtUserWaitMessage+0xc
0012e920 77f4f762 77f30000 00151768 00000000 user32!InternalDialogBox+0xd0
0012ebe0 77f4f047 0012ed3c 00000000 ffffffff user32!SoftModalMessageBox+0x94b
0012ed30 77f4eec9 0012ed3c 00000028 00000000 user32!MessageBoxWorker+0x2ba
0012ed88 77f87d0d 00000000 001511a8 0014ef50 user32!MessageBoxTimeoutW+0x7a
0012edbc 77f742c8 00000000 0012ee70 1001d7d4 user32!MessageBoxTimeoutA+0x9c
0012eddc 77f742a4 00000000 0012ee70 1001d7d4 user32!MessageBoxExA+0x1b
0012edf8 10014c9a 00000000 0012ee70 1001d7d4 user32!MessageBoxA+0×45
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ee2c 10010221 0012ee70 1001d7d4 00012010 component!Error+0×7e4a
...
```

Dumping the message box message and its title shows that Visual C++ runtime detected a buffer overflow condition:

```
0:000> da 0012ee70
0012ee70  "Buffer overrun detected!..Progra”
0012ee90  “m: E:\W\program.exe..A buffer ov”
0012eeb0  “errun has been detected which ha”
0012eed0  “s corrupted the program’s.intern”
0012eef0  “al state.  The program cannot sa”
0012ef10  “fely continue execution and must”
0012ef30  “.now be terminated..”

0:000> da 1001d7d4
1001d7d4  "Microsoft Visual C++ Runtime Lib"
1001d7f4  "rary"
```

WAITING THREAD TIME (USER DUMPS)

Waiting Thread Time kernel pattern (Volume 1, page 343) shows how to calculate thread waiting time in kernel and complete memory dumps. Now we would see how to do it for user dumps. Unfortunately this information is not available in user space and therefore this can be done only approximately by calculating average or maximum waiting time.

The key is to take advantage of **!runaway** WinDbg command. Modern versions of postmortem debuggers and process dumpers usually save thread time information additionally. For example, **/ma** switch for **.dump** command does it. The later versions of userdump.exe also do it. When **!runaway** is invoked with 0y111 (7) flag it shows each thread user and kernel times and also the time elapsed since each thread was created. Therefore for threads that didn't do much CPU intensive computation their average waiting time will almost correspond to their elapsed time. Unfortunately we cannot say when that computation took place. It could be the case that the thread in question was waiting 98.9% of the time, then did some computation for 0.01% and was waiting again 1% of the time or it could be the case that it was doing computation 0.01% of the time initially and then waiting the rest of the time (99.9%) until the dump was saved.

Consider this example of a sleeping thread:

```
0:000> ~32kL 100
ChildEBP RetAddr
0a18f03c 7c826f4b ntdll!KiFastSystemCallRet
0a18f040 77e41ed1 ntdll!NtDelayExecution+0xc
0a18f0a8 77e424ed kernel32!SleepEx+0x68
0a18f0b8 1b0ac343 kernel32!Sleep+0xf
0a18f100 1b00ba49 msjet40!System::AllocatePages+0x15e
0a18f11c 1b00ba08 msjet40!PageDesc::WireCurrentPage+0x2f
0a18f13c 1b00b8dd msjet40!PageDesc::ReadPage+0x119
0a18f164 1b01b567 msjet40!Database::ReadPage+0x7a
0a18f190 1b00e309 msjet40!TableMover::GetNextRow+0xc9
0a18f1a8 1b015de9 msjet40!TableMover::Move+0xc4
0a18f1d8 1b015d9c msjet40!ErrIsamMove+0x6c
0a18f1f0 1b038aa4 msjet40!ErrDispMove+0x43
0a18f43c 1b015d9c msjet40!ErrJPMoveRange+0x350
0a18f454 1b0200b3 msjet40!ErrDispMove+0x43
0a18f6a0 1b021e4d msjet40!ErrMaterializeRows+0x1fd
0a18f6f0 1b021c0d msjet40!ErrJPSetColumnSort+0x191
0a18f718 1b0210a4 msjet40!ErrJPOpenSort+0x105
0a18f750 1b020de5 msjet40!ErrJPOpenRvt+0x171
0a18f9f0 1b039b82 msjet40!ErrExecuteQuery+0x548
0a18fa3c 1b05548b msjet40!ErrExecuteTempQuery+0x13d
0a18fa6c 4c23b01e msjet40!JetExecuteTempQuery+0xc9
0a18fa94 4c23a8d1 odbcjt32!DoJetCloseTable+0x64
0a18fd10 4c23a6b6 odbcjt32!SQLInternalExecute+0x217
0a18fd20 4c23a694 odbcjt32!SQLExecuteCover+0x1f
0a18fd28 488b3fc1 odbcjt32!SQLExecute+0x9
0a18fd44 0c4898b5 odbc32!SQLExecute+0xd3
...
```

The process uptime can be seen from **vertarget** WinDbg command:

```
0:000> vertarget
...
Process Uptime: 0 days 1:17:22.000
...
```

Then we dump thread times (the output is long and only information for our 32nd thread is shown here):

```
0:000> !runaway 0y111
 User Mode Time
  Thread       Time
  32:cfc       0 days 0:00:00.109
...
 Kernel Mode Time
  Thread       Time
  32:cfc       0 days 0:00:00.062
...
 Elapsed Time
  Thread       Time
...
  32:cfc       0 days 1:17:20.703
...
```

We see that 1 hour, 17 minutes and almost 21 seconds passed since the thread was created. By subtracting this time from process uptime we see that it was created in the first 2 seconds and it was consuming less than one second of CPU time. Therefore, most of the time this thread was waiting. Unfortunately we cannot say when it was waiting most of the time, in the beginning before it started sleeping or after. Fortunately we might continue guessing by looking at Sleep argument:

```
0:032> kv
ChildEBP RetAddr  Args to Child
0a18f03c 7c826f4b 77e41ed1 00000000 0a18f080 ntdll!KiFastSystemCallRet
0a18f040 77e41ed1 00000000 0a18f080 00000000 ntdll!NtDelayExecution+0xc
0a18f0a8 77e424ed 00000032 00000000 04390000 kernel32!SleepEx+0x68
0a18f0b8 1b0ac343 00000032 042a34b4 0a18f1c8 kernel32!Sleep+0xf
...
```

We see that it had been sleeping for at most 32 milliseconds and perhaps this is a retry / sleep loop. We might guess that the thread was recently spinning in that loop and therefore waiting all the time before that. Alternatively we might guess that the retry portion is very small and fast and 32 milliseconds are spread over all elapsed time and the thread was in this loop the significant proportion of time. What we can surely say that the last waiting time was no more than 32 milliseconds. in the case of waiting for an event, for example, at present, it seems there is no any reliable way of calculating this time. If anyone knows please post a comment.

INLINE FUNCTION OPTIMIZATION

Sometimes compilers optimize code by replacing function calls with their bodies. This procedure is called function inlining and functions themselves are called inline. On one platform we can see the real function call on the stack trace but on another platform or product version we only see the same problem instruction. Fortunately the rest of stack trace should be the same. Therefore when comparing stack traces (Volume 1, page 395) we shouldn't pay attention only to the top function call.

This pattern is frequently seen when threads crash while copying or moving memory. Consider this stack trace:

```
0: kd> kL
ChildEBP RetAddr
f22efaf4 f279ec3d driver!QueueValue+0x26b
f22efb30 8081dcdf driver!BufferAppendData+0x35f
f22efc7c 808f47b7 nt!IofCallDriver+0x45
f22efc90 808f24ee nt!IopSynchronousServiceTail+0x10b
f22efd38 80888c7c nt!NtWriteFile+0x65a
f22efd38 7c82ed54 nt!KiFastCallEntry+0xfc
```

When looking at **rep movs** instruction we might suspect that QueueValue was copying memory:

```
0: kd> r
eax=00000640 ebx=89b23000 ecx=00000190 edx=89b3c828 esi=02124220
edi=e2108f58
eip=f279c797 esp=f22efadc ebp=f22efaf4 iopl=0 nv up ei pl nz na pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010206
driver!QueueValue+0x26b:
f279c797 f3a5 rep movs dword ptr es:[edi],dword ptr [esi]
es:0023:e2108f58=dfefbecf ds:0023:02124220=????????
```

On x64 bit platform the same driver had the similar stack trace but with memcpy at its top:

```
fffffadf`8955f4a8 fffffadf`8d1bef46 driver!memcpy+0x1c0
fffffadf`8955f4b0 fffffadf`8d1c15c9 driver!QueueValue+0x2fe
fffffadf`8955f550 fffff800`01273ed9 driver!BufferAppendData+0x481
...
```

We also see how QueueValue+0×2fe and QueueValue+0×26b are close. In fact the source code for the driver calls RtlCopyMemory function only once and it is defined as memcpy in wdm.h. The latter function is also exported from **nt** module:

```
0: kd> x nt!
...
80881780 nt!memcpy = <no type information>
...
```

but usually can be found in any driver that links it from C runtime library, for example, on x64 Windows:

```
1: kd> x nt!memcpy
fffff800`01c464e0 nt!memcpy = <no type information>

1: kd> x srv!memcpy
fffff980`0eafdf20 srv!memcpy = <no type information>

1: kd> x win32k!memcpy
fffff960`000c1b40 win32k!memcpy = <no type information>
```

Therefore we see that when compiling for x86 platform Visual C++ compiler decided to inline memcpy code but AMD compiler on x64 platform didn't inline it. The overall stack trace without offsets is very similar and we can suppose that the problem was identical.

CRITICAL SECTION CORRUPTION

Dynamic memory corruption patterns in user (Volume 1, page 257) and kernel (page 204) spaces are specializations of one big parent pattern called **Corrupt Structure** because crashes happen there due to corrupt or overwritten heap or pool control structures (for the latter see **Double Free** pattern, Volume 1, page 378). Critical sections are linked together through statically pre-allocated or heap-allocated helper structure (shown in bold italic) although themselves they can be stored anywhere from static and stack area to process heap:

```
0:001> dt -r1 ntdll!_RTL_CRITICAL_SECTION 77795240
   +0×000 DebugInfo        : 0×00175d28 _RTL_CRITICAL_SECTION_DEBUG
      +0×000 Type             : 0
      +0×002 CreatorBackTraceIndex : 0
      +0×004 CriticalSection  : 0×77795240 _RTL_CRITICAL_SECTION
      +0×008 ProcessLocksList : _LIST_ENTRY [ 0×173a08 - 0×173298 ]
      +0×010 EntryCount       : 0
      +0×014 ContentionCount  : 0
      +0×018 Spare            : [2] 0
   +0×004 LockCount        : -1
   +0×008 RecursionCount   : 0
   +0×00c OwningThread     : (null)
   +0×010 LockSemaphore    : (null)
   +0×014 SpinCount        : 0

0:001> !address 77795240
    77670000 : 77792000 - 00005000
                    Type     01000000 MEM_IMAGE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                  Usage    RegionUsageImage
                    FullPath C:\WINDOWS\system32\ole32.dll

0:001> !address 0×00175d28
    00140000 : 00173000 - 0000d000
                    Type     00020000 MEM_PRIVATE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageHeap
                    Handle   00140000
```

```
0:000> !locks

CritSec ntdll!LdrpLoaderLock+0 at 7c8877a0
WaiterWoken        No
LockCount          0
RecursionCount     1
OwningThread       1184
EntryCount         0
ContentionCount    b04707
*** Locked

0:000> dt -r1 _RTL_CRITICAL_SECTION 7c8877a0
   +0×000 DebugInfo         : 0×7c8877c0 _RTL_CRITICAL_SECTION_DEBUG
      +0×000 Type              : 0
      +0×002 CreatorBackTraceIndex : 0
      +0×004 CriticalSection   : 0×7c8877a0 _RTL_CRITICAL_SECTION
      +0×008 ProcessLocksList  : _LIST_ENTRY [ 0×7c887be8 - 0×7c887bc8 ]
      +0×010 EntryCount        : 0
      +0×014 ContentionCount   : 0xb04707
      +0×018 Spare             : [2] 0
   +0×004 LockCount         : -2
   +0×008 RecursionCount    : 1
   +0×00c OwningThread      : 0×00001184
   +0×010 LockSemaphore     : 0×0000013c
   +0×014 SpinCount         : 0

0:000> !address 7c8877a0
    7c800000 : 7c887000 - 00003000
                    Type     01000000 MEM_IMAGE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageImage
                    FullPath C:\WINDOWS\system32\ntdll.dll

0:000> !address 0×7c8877c0
    7c800000 : 7c887000 - 00003000
                    Type     01000000 MEM_IMAGE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageImage
                    FullPath C:\WINDOWS\system32\ntdll.dll
```

Consider the case when CRITICAL_SECTION structure is defined on a stack and there was **Local Buffer Overflow** (Volume 1, page 460) overwriting DebugInfo pointer. Then we have an example of **Wild Pointer** pattern (page 202) and traversing the list of critical sections from this point will diverge into completely unrelated memory area or stop there. Consider another example of heap corruption or race condition overwriting ProcessLocksList or CriticalSection pointer. Then we have another instance of Wild Pointer pattern illustrated below:

```
0:000> !locks

CritSec ntdll!LdrpLoaderLock+0 at 7c8877a0
WaiterWoken        No
LockCount          0
RecursionCount     1
OwningThread       1184
EntryCount         0
ContentionCount    b04707
*** Locked

CritSec +1018de08 at 1018de08
WaiterWoken        Yes
LockCount          -49153
RecursionCount     5046347
OwningThread       460050
EntryCount         0
ContentionCount    0
*** Locked

CritSec +1018ddd8 at 1018ddd8
WaiterWoken        Yes
LockCount          -1
RecursionCount     0
OwningThread       0
*** Locked

CritSec +1018de28 at 1018de28
WaiterWoken        Yes
LockCount          -1
RecursionCount     0
OwningThread       0
*** Locked

CritSec +1018de08 at 1018de08
WaiterWoken        Yes
LockCount          -49153
RecursionCount     5046347
OwningThread       460050
EntryCount         0
ContentionCount    0
*** Locked

CritSec +1018de28 at 1018de28
WaiterWoken        Yes
LockCount          -1
RecursionCount     0
OwningThread       0
*** Locked
```

```
CritSec +1018ddd8 at 1018ddd8
WaiterWoken        Yes
LockCount          -1
RecursionCount     0
OwningThread       0
*** Locked

Scanned 841 critical sections
```

We see the signs of corruption at 1018de08 address which is interpreted as pointing to a locked critical section. To see where the corruption started we need to look at the list of all critical sections either locked or not locked:

```
0:000> !locks -v

CritSec ntdll!RtlCriticalSectionLock+0 at 7c887780
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    28

CritSec ntdll!LdrpLoaderLock+0 at 7c8877a0
WaiterWoken        No
LockCount          0
RecursionCount     1
OwningThread       1184
EntryCount         0
ContentionCount    b04707
*** Locked

CritSec ntdll!FastPebLock+0 at 7c887740
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    42c9

CritSec ntdll!RtlpCalloutEntryLock+0 at 7c888ea0
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0

CritSec ntdll!PMCritSect+0 at 7c8883c0
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0
```

```
CritSec ntdll!UMLogCritSect+0 at 7c888400
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0

CritSec ntdll!RtlpProcessHeapsListLock+0 at 7c887960
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0

CritSec +80608 at 00080608
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    22

...

CritSec cabinet!_adbgmsg+13c at 74fb4658
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0

CritSec +c6c17c at 00c6c17c
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0

CritSec +c6c0e4 at 00c6c0e4
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0

CritSec at 1018de08 does not point back to the debug info at 00136a40
Perhaps the memory that held the critical section has been reused without
calling DeleteCriticalSection() ?
```

```
CritSec +1018de08 at 1018de08
WaiterWoken        Yes
LockCount          -49153
RecursionCount     5046347
OwningThread       460050
EntryCount         0
ContentionCount    0
*** Locked

CritSec at 1018ddd8 does not point back to the debug info at 00136a68
Perhaps the memory that held the critical section has been reused without
calling DeleteCriticalSection() ?

CritSec +1018ddd8 at 1018ddd8
WaiterWoken        Yes
LockCount          -1
RecursionCount     0
OwningThread       0
*** Locked

...
```

We see that the problem appears when the heap-allocated critical section at 00c6c0e4 address is linked to an inconsistent critical section at 0×1018de08 address where its memory data contains UNICODE string fragment (shown in smaller font for visual clarity):

```
0:000> !address 00c6c0e4
    00c60000 : 00c60000 - 00010000
                    Type     00020000 MEM_PRIVATE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageHeap
                    Handle   00c60000

0:000> dt -r1 _RTL_CRITICAL_SECTION 00c6c0e4
   +0x000 DebugInfo        : 0x00161140 _RTL_CRITICAL_SECTION_DEBUG
      +0x000 Type             : 0
      +0x002 CreatorBackTraceIndex : 0
      +0x004 CriticalSection  : 0x00c6c0e4 _RTL_CRITICAL_SECTION
      +0×008 ProcessLocksList : _LIST_ENTRY [ 0×136a48 - 0×119f58 ]
      +0×010 EntryCount       : 0
      +0×014 ContentionCount  : 0
      +0×018 Spare            : [2] 0
   +0×004 LockCount        : -1
   +0×008 RecursionCount   : 0
   +0×00c OwningThread     : (null)
   +0×010 LockSemaphore    : (null)
   +0×014 SpinCount        : 0
```

```
0:000> dt -r _RTL_CRITICAL_SECTION_DEBUG 0×00136a48-0×008
   +0×000 Type             : 0
   +0×002 CreatorBackTraceIndex : 0
   +0×004 CriticalSection  : 0×1018de08 _RTL_CRITICAL_SECTION
      +0×000 DebugInfo        : 0×000d001b _RTL_CRITICAL_SECTION_DEBUG
         +0×000 Type             : 0
         +0×002 CreatorBackTraceIndex : 0
         +0×004 CriticalSection  : (null)
         +0×008 ProcessLocksList : _LIST_ENTRY [ 0×0 - 0×0 ]
         +0×010 EntryCount       : 0
         +0×014 ContentionCount  : 0×37e3c700
         +0×018 Spare            : [2] 0×8000025
      +0×004 LockCount        : 196609
      +0×008 RecursionCount   : 5046347
      +0×00c OwningThread     : 0×00460050
      +0×010 LockSemaphore    : 0×00310033
      +0×014 SpinCount        : 0×520044
   +0×008 ProcessLocksList : _LIST_ENTRY [ 0×136a70 - 0×161148 ]
      +0×000 Flink            : 0×00136a70 _LIST_ENTRY [ 0×136a98 - 0×136a48 ]
         +0×000 Flink            : 0×00136a98 _LIST_ENTRY [ 0×136ae8 - 0×136a70 ]
         +0×004 Blink            : 0×00136a48 _LIST_ENTRY [ 0×136a70 - 0×161148 ]
      +0×004 Blink            : 0×00161148 _LIST_ENTRY [ 0×136a48 - 0×119f58 ]
         +0×000 Flink            : 0×00136a48 _LIST_ENTRY [ 0×136a70 - 0×161148 ]
         +0×004 Blink            : 0×00119f58 _LIST_ENTRY [ 0×161148 - 0×16cc3c0 ]
   +0×010 EntryCount       : 0
   +0×014 ContentionCount  : 0
   +0×018 Spare            : [2] 0×2e760000
```

```
0:000> !address 0x1018de08
    10120000 : 10120000 - 00100000
                    Type     00020000 MEM_PRIVATE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageIsVAD
```

The address points miraculously to some DLL:

```
0:000> du 1018de08
1018de08  "....componentA.dll"
```

We might suggest that componentA.dll played some role there.

There are other messages from verbose version of **!locks** WinDbg command pointing to critical section problems:

```
CritSec componentB!Section+0 at 74004008
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0

The CritSec componentC!Info+c at 72455074 has been RE-INITIALIZED.
The critical section points to DebugInfo at 00107cc8 instead of 000f4788
```

```
CritSec componentC!Info+c at 72455074
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0

CritSec componentD!foo+8ec0 at 0101add0
LockCount          NOT LOCKED
RecursionCount     0
OwningThread       0
EntryCount         0
ContentionCount    0
```

LOST OPPORTUNITY

It is important to save crash dumps at the right time, for example, when an error message box is shown (Volume 1, page 39).

However, sometimes, crash dumps are saved after a visual indicator of the problem disappeared and the opportunity to see the stack trace was lost. Here is one example of a service memory dump saved manually after it became unresponsive. In the dump file there was only one thread left excluding the thread created by a debugger (shown in smaller font for visual clarity):

```
0:001> ~*kv

   0  Id: a3c.700 Suspend: 1 Teb: 7ff59000 Unfrozen
ChildEBP RetAddr  Args to Child
1178fd60 7c822124 77e6bad8 00000574 00000000 ntdll!KiFastSystemCallRet
1178fd64 77e6bad8 00000574 00000000 00000000 ntdll!NtWaitForSingleObject+0xc
1178fdd4 77e6ba42 00000574 ffffffff 00000000 kernel32!WaitForSingleObjectEx+0xac
1178fde8 67e223dd 00000574 ffffffff 1178fe10 kernel32!WaitForSingleObject+0x12
1178fdfc 7c82257a 67e20000 00000000 00000001 componentA!DllInitialize+0xed
1178fe1c 7c8118b0 67e222f0 67e20000 00000000 ntdll!LdrpCallInitRoutine+0x14
1178feb8 77e53002 00000000 00000000 00000000 ntdll!LdrShutdownProcess+0x130
1178ffa4 77e53065 c0000005 77e8f3b0 ffffffff kernel32!_ExitProcess+0×43
1178ffb8 77e84277 c0000005 00000000 00000000 kernel32!ExitProcess+0×14
1178ffec 00000000 77c5de6d 0a078138 00000000 kernel32!BaseThreadStart+0×5f

#  1  Id: a3c.18bc Suspend: 1 Teb: 7ffde000 Unfrozen
ChildEBP RetAddr  Args to Child
0a6cffc8 7c845ea0 00000005 00000004 00000001 ntdll!DbgBreakPoint
0a6cfff4 00000000 00000000 00905a4d 00000003 ntdll!DbgUiRemoteBreakin+0x36
```

We also see exception code 0xc0000005 as ExitProcess parameter. The raw stack reveals the call to NtRaiseHardError function that definitely resulted in some error message box:

```
0:001> ~0s
...
```

```
0:000> !teb
TEB at 7ff59000
    ExceptionList:        1178fdc4
    StackBase:            11790000
    StackLimit:           11789000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ff59000
    EnvironmentPointer:   00000000
    ClientId:             00000a3c . 00000700
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffdf000
    LastErrorValue:       0
    LastStatusValue:      c0000008
    Count Owned Locks:    0
    HardErrorMode:        0

0:000> dds 11789000  11790000
11789000  00000000
11789004  00000000
11789008  00000000
1178900c  00000000
11789010  00000000
...
1178f058  0a4016f4
1178f05c  1178efe0
1178f060  695040c4 <Unloaded_faultrep.dll>+0x40c4
1178f064  7ffdf000
1178f068  00000000
1178f06c  0a4016f4
1178f070  0a4016f4
1178f074  00000000
1178f078  1178f06c
1178f07c  0a4016b8
1178f080  0a4016b8
1178f084  1178efa0
1178f088  7c821b74 ntdll!NtRaiseHardError+0xc
1178f08c  77e99af9 kernel32!UnhandledExceptionFilter+0×54b
1178f090  d0000144
1178f094  00000004
1178f098  00000000
1178f09c  1178f164
1178f0a0  00000001
1178f0a4  77e996a7 kernel32!UnhandledExceptionFilter+0×873
1178f0a8  00000000
1178f0ac  00000000
1178f0b0  00000000
1178f0b4  02f049f0
1178f0b8  1178f13c
1178f0bc  00000000
...
```

It was that time when the dump should have been saved. See also Volume 1, page 624 for another example and full explanation.

YOUNG SYSTEM

Opposite to **Overaged System** (page 273) sometimes we can see this pattern. This means that the system didn't have time to initialize and subsequently mature or reach the state when the problem could surface. Usual signs are less than a minute system uptime (or larger, depends on a problem context) and the low number of processes and services running. Also, sometimes, the problem description mentions a terminal services session but there is only one console session in the dump, or two as in Vista and Windows Server 2008:

```
System Uptime: 0 days 0:00:18.562

3: kd> !vm
...
         0248 lsass.exe         1503 (      6012 Kb)
         020c winlogon.exe      1468 (      5872 Kb)
         03b8 svchost.exe        655 (      2620 Kb)
         023c services.exe       416 (      1664 Kb)
         01f0 csrss.exe          356 (      1424 Kb)
         0338 svchost.exe        298 (      1192 Kb)
         02dc svchost.exe        259 (      1036 Kb)
         0374 svchost.exe        240 (       960 Kb)
         039c svchost.exe        224 (       896 Kb)
         01bc smss.exe            37 (       148 Kb)
         0004 System               8 (        32 Kb)

3: kd> !session
Sessions on machine: 1
Valid Sessions: 0
```

In the case of the fully initialized system the manual dump (Volume 1, page 479) could be taken after reboot when the bugcheck already happened or any other reason stemming from the usual confusion between crashes and hangs (Volume 1, page 36)..

Similar considerations apply to a young process as well, where *Process Uptime* value from user dumps or *ElapsedTime* value from kernel or complete memory dumps is too small unless we have obvious crash or hang signs inside, for example, exceptions, **Deadlock** (Volume 1, page 276), **Wait Chain** (Volume 1, page 490) or **Blocked Thread** (page 184) waiting for another **Coupled Processes** (Volume 1, page 419):

```
Process Uptime: 0 days 0:00:10.000

3: kd> !process 8a389d88
PROCESS 8a389d88  SessionId: 0  Cid: 020c    Peb: 7ffdf000  ParentCid:
01bc
    DirBase: 7fbe6080  ObjectTable: e1721008  HandleCount: 455.
    Image: winlogon.exe
    VadRoot 8a65d070 Vads 194 Clone 0 Private 1166. Modified 45. Locked 0.
    DeviceMap e10030f8
    Token                             e139bde0
    ElapsedTime                       00:00:01.062
    UserTime                          00:00:00.046
    KernelTime                        00:00:00.015
    QuotaPoolUsage[PagedPool]         71228
    QuotaPoolUsage[NonPagedPool]      72232
    Working Set Sizes (now,min,max)  (2265, 50, 345) (9060KB, 200KB,
1380KB)
    PeakWorkingSetSize                2267
    VirtualSize                       41 Mb
    PeakVirtualSize                   42 Mb
    PageFaultCount                    2605
    MemoryPriority                    BACKGROUND
    BasePriority                      13
    CommitCharge                      1468
```

LAST ERROR COLLECTION

Sometimes a dump file looks normal inside and we don’t see any suspicious past activity. However, as it often happens, the dump was saved manually as a response to some failure. This pattern might help in finding further troubleshooting suggestions. If we have a process memory dump we can get all errors and NTSTATUS values at once using **!gle** command with **-all** parameter:

```
0:000> !gle -all
Last error for thread 0:
LastErrorValue: (Win32) 0x3e5 (997) - Overlapped I/O operation is in
progress.
LastStatusValue: (NTSTATUS) 0x103 - The operation that was requested is
pending completion.

Last error for thread 1:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

Last error for thread 2:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

Last error for thread 3:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

...

Last error for thread 28:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

Last error for thread 29:
LastErrorValue: (Win32) 0×6ba (1722) - The RPC server is unavailable.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

Last error for thread 2a:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

Last error for thread 2b:
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0 - STATUS_WAIT_0

...
```

For complete memory dumps we can employ the following command or similar to it:

!for_each_thread ".thread /r /p @#Thread; .if (@$teb != 0) {!teb; !gle;}"

```
0: kd> !for_each_thread ".thread /r /p @#Thread; .if (@$teb != 0) { !teb;
!gle; }"

...

Implicit thread is now 8941eb40
Implicit process is now 8a4ac498
Loading User Symbols
TEB at 7ff3e000
    ExceptionList:        0280ffa8
    StackBase:            02810000
    StackLimit:           0280b000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ff3e000
    EnvironmentPointer:   00000000
    ClientId:             00001034 . 000012b0
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffde000
    LastErrorValue:       0
    LastStatusValue:      c00000a3
    Count Owned Locks:    0
    HardErrorMode:        0
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.
LastStatusValue: (NTSTATUS) 0xc00000a3 - {Drive Not Ready}  The drive is
not ready for use; its door may be open.  Please check drive %hs and make
sure that a disk is inserted and that the drive door is closed.

...
```

HIDDEN MODULE

Sometimes we look for modules that were loaded and unloaded at some time. **lm** command lists unloaded modules but some of them could be mapped to address space without using the runtime loader. The latter case is common for drm-type protection tools, rootkits, malware or crimeware which can influence a process execution. In such cases we can hope that they still remain in virtual memory and search for them. WinDbg **.imgscan** command greatly helps in identifying MZ/PE module headers. The following example illustrates this command without implying that the found module did any harm:

```
0:000> .imgscan
MZ at 000d0000, prot 00000002, type 01000000 - size 6000
  Name: usrxcptn.dll
MZ at 00350000, prot 00000002, type 01000000 - size 9b000
  Name: ADVAPI32.dll
MZ at 00400000, prot 00000002, type 01000000 - size 23000
  Name: javaw.exe
MZ at 01df0000, prot 00000002, type 01000000 - size 8b000
  Name: OLEAUT32.dll
MZ at 01e80000, prot 00000002, type 01000000 - size 52000
  Name: SHLWAPI.dll
...
```

We don't see **usrxcptn** in either loaded or unloaded module lists:

```
0:002> lm
start    end        module name
00350000 003eb000   advapi32
00400000 00423000   javaw
01df0000 01e7b000   oleaut32
01e80000 01ed2000   shlwapi
...

Unloaded modules:
```

Then we can use **Unknown Component** pattern (Volume 1, page 367) to see the module resources if present in memory:

```
0:002> !dh 000d0000

...

SECTION HEADER #4
   .rsrc name
     418 virtual size
    4000 virtual address
     600 size of raw data
    1600 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         (no align specified)
         Read Only

...

0:002> dc 000d0000+4000 L418
...
000d4140  ... n…z.)…F.i.l.
000d4150  ... e.D.e.s.c.r.i.p.
000d4160  ... t.i.o.n…..U.s.
000d4170  ...   e.r. .D.u.m.p. .
000d4180  ... U.s.e.r. .M.o.d.
000d4190  ... e. .E.x.c.e.p.t.
000d41a0  ... i.o.n. .D.i.s.p.
000d41b0  ... a.t.c.h.e.r…..

0:002> du 000d416C
000d416c  "User Dump User Mode Exception Di"
000d41ac  "spatcher"
```

This component seems to be loaded or mapped only if userdump package was fully installed where usrxcptn.dll is a part of its redistribution and the application was added to Process Dumper applet in Control Panel. Although from the memory dump comment we also see that the dump was taken manually using command line userdump.exe we see that the full userdump package was additionally installed which was probably not necessary (see **Correcting Microsoft Article About userdump.exe**, Volume 1, page 612):

```
Loading Dump File [javaw.dmp]
User Mini Dump File with Full Memory: Only application data is available

Comment: 'Userdump generated complete user-mode minidump with Standalone
function on COMPUTER-NAME'
```

HIGH CONTENTION (CRITICAL SECTIONS)

High Contention pattern in kernel mode involving executive resources is explained in Volume 1, page 421. The same pattern can be observed in user space involving critical sections guarding shared regions like serialized process heap or a memory database, for example, in one Windows service process during increased workload:

```
0:000> !locks

CritSec +310608 at 00310608
WaiterWoken         No
LockCount           6
RecursionCount      1
OwningThread        d9c
EntryCount          0
ContentionCount     453093
*** Locked

CritSec +8f60f78 at 08f60f78
WaiterWoken         No
LockCount           8
RecursionCount      1
OwningThread        d9c
EntryCount          0
ContentionCount     af7f0
*** Locked

CritSec +53bf8f10 at 53bf8f10
WaiterWoken         No
LockCount           0
RecursionCount      1
OwningThread        1a9c
EntryCount          0
ContentionCount     e
*** Locked

Scanned 7099 critical sections
```

When looking at the owning thread we see that the contention involves process heap:

```
0:000> ~~[d9c]kL
ChildEBP RetAddr
0e2ff9d4 7c81e845 ntdll!RtlpFindAndCommitPages+0×14e
0e2ffa0c 7c81e4ef ntdll!RtlpExtendHeap+0xa6
0e2ffc38 7c3416b3 ntdll!RtlAllocateHeap+0×645
0e2ffc78 7c3416db msvcr71!_heap_alloc+0xe0
```

```
0e2ffc80 7c3416f8 msvcr71!_nh_malloc+0×10
0e2ffc8c 672e14fd msvcr71!malloc+0xf
0e2ffc98 0040bc28 dll!MemAlloc+0xd
...
0e2fff84 7c349565 dll!WorkItemThread+0×152
0e2fffb8 77e6608b msvcr71!_endthreadex+0xa0
0e2fffec 00000000 kernel32!BaseThreadStart+0×34
```

However two critical section addresses belong to the same heap:

```
0:000> !address 00310608
    00310000 : 00310000 - 00010000
                    Type     00020000 MEM_PRIVATE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageHeap
                    Handle   00310000

0:000> !address 08f60f78
    08f30000 : 08f30000 - 00200000
                    Type     00020000 MEM_PRIVATE
                    Protect  00000004 PAGE_READWRITE
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageHeap
                    Handle   00310000
```

Lock contention is confirmed in heap statistics as well (shown in smaller font for visual clarity):

```
0:000> !heap -s
LFH Key                   : 0x07262959
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                    (k)     (k)    (k)     (k) length      blocks cont. heap
00140000 00000002    8192    2876   3664    631   140    46    0     1e   L
    External fragmentation  21 % (140 free blocks)
00240000 00008000      64      12     12     10     1     1    0      0
Virtual block: 0ea20000 - 0ea20000 (size 00000000)
Virtual block: 0fa30000 - 0fa30000 (size 00000000)
00310000 00001002 1255320 961480 1249548 105378     0 16830    2 453093   L
    Virtual address fragmentation  23 % (16830 uncommited ranges)
    Lock contention  4534419
003f0000 00001002      64      36     36      0     0     1    0      0   L
00610000 00001002      64      16     16      4     2     1    0      0   L
...
```

PART 4: CRASH DUMP ANALYSIS ANTIPATTERNS

DEBUGGING ARCHITECTS

They know buzzwords like heap corruption, buffer overflow and multi-threading, talk about designing maintainable software but unable to cope with real-life debugging scenarios. There is a story about one company where the executable Rational Rose Real-Time model with *IDebug* interface crashed and the whole team of software designers couldn't find the defect in code and when one engineer pointed them to the problem source code line after loading and running the executable under the Visual Studio debugger he was nominated as an expert in implementation.

SYMBOLLESS ANALYSIS

This is another anti-pattern when an engineer either in a hurry or due to laziness doesn't apply proper symbols and relies only on timestamps and module/offsets or trusts what WinDbg says and ignores symbol warnings. It is usually safer to apply symbols even in obvious cases and in hard ones we should strive to apply them until all possibilities are exhausted including file search using PDBFinder (Volume 1, page 668).

Another weak variant is called **Imageless Analysis** when an engineer doesn't specify proper Executable Image Search Path when it is necessary perhaps due to ignorance or just plain laziness again. Please see **Minidump Analysis** example (Volume 1, page 63) for proper minidump analysis.

MYOPIC TROUBLESHOOTING AND DEBUGGING

Often engineers spend 10 minutes pursuing a certain investigation path and then prematurely closing it and switching to another. This anti-pattern name was inspired by Daniel Dennett's discussion of insufficiently patient scientists doing computer simulations:

"mistaking a failure of imagination for an insight into necessity" (Darwin's Dangerous Idea, page 175).

Paraphrasing we can say that engineers think of impossibility where their imagination fails.

PART 5: A BIT OF SCIENCE

MEMORETICS

Memory dump analysis needs to be put on relevant scientific grounds and this branch of science needs its own name. After considering different alternative names the word **Memoretics** was finally chosen. Here is the brief definition:

Computer Memoretics *studies computer memory snapshots (Volume 1, page 501) and their evolution in time.*

Obviously this domain of research has many links with application and system debugging. However its scope is wider than debugging because it doesn't necessarily study memory snapshots from systems and applications experiencing faulty behaviour.

MEMORY ANALYSIS

Here is an attempt to come up with memory analysis classification:

Memory Analysis Forensics:

Answering questions related to a committed computer crime. The suspect may be a human or a software / hardware component. Incident response, troubleshooting and debugging belong to this category. Postmortem memory analysis is usually analysis of dump files saved and detached from the original system or operating conditions.

Memory Analysis Intelligence:

Monitoring memory state for behavioural and structural patterns to prevent certain events from occurring. Usually done in situ. However digital dumpster divers and spies may also collect and analyze memory data that was detached from the original computer system.

Each category can be further subdivided into:

Functional Memory Analysis:

Tracing of events.

Memoretics (page 347)**:**

Analysis of memory states and their evolution.

The latter can be subdivided into:

Static Memory Analysis:

Traditional memory dump analysis.

Dynamic Memory Analysis:

Live debugging.

MEMOIDEALISM

Looking at memory dumps every day and writing about them has an unfortunate implication: every state of the world looks like a gigantic memory dump to me. Everything is memory and every state is memory dump. The current state of the world is an infinite (or an immense) number of memuons[*]. Infinite can be any cardinal number greater or equal to that of natural numbers. In any case we can say it is N bits where this number is either finite or ∞. Therefore we have 2^N possible memory states (S). The set of possible transitions between them (S -> S) has the number of 2^N ^ 2^N elements. Which is the memory itself and we have transitions between its states too. Ad infinitum we have a limiting process from which arises the perceived flow of events.

(*) **Memuon** is an indivisible entity similar to a bit of information.

MEMIOTICS

Analysis of computer memory snapshots (memory dumps, Volume 1, page 501) and their evolution is the domain of memoretics (page 347). Computer memory semiotics (**mem**iotics or **memo**semiotics) is the branch of memoretics that studies the interpretation of computer memory, its meaning, signs and symbols.

PART 6: FUN WITH CRASH DUMPS

MUSIC FOR DEBUGGING

THE GLORY OF DEBUGGING

I would like to recommend El Greco Vangelis album that I used to listen to long time ago. Various movements correspond to different debugging sessions, some glorious and some filled with tension and worries, gradually building up the ultimate problem resolution. I think almost all Vangelis music is very suitable to accompany debugging.

MEMORY ANALYSIS ALBUM

Here I review this CD from Polish band Gutter Sirens:

After listening to it I can say that it is wonderful! Clearly belongs to my Music for Debugging collection. Here is my commentary (in italics) on track titles:

1. Chameleon (*It crashes then hangs then spikes then leaks. You never know what happens next…*)
2. Silence Before The Storm (*Waiting for the problem to start dumping memory*)
3. Appearances (*Multiple occurrences of the issue ease crash dump collection*)
4. Memory Analysis (*Memory dump analysis activity*)
5. Voices From Heaven (*It's my blog full of crash dump analysis patterns*)
6. Figure In The Fog (*It's that DLL! I see it clearly!!!*)
7. Diamond Tear (*The Customer is happy*)
8. The Toy Soldier (*WinDbg*)
9. Tears In Dragon's Eyes (*Operating system vendor is happy too*)
10. Forgotten Song (*Bugs NO PASARAN!*)
11. Skies (*We are debugging gods!*)

BIOGRAPHY OF A BUG

Klaus Schulze was one of my favorite electronic composers in 90s. I recommend ”X” album featuring several musical biographies including Nietzsche. “X” here stands for 10th album not for **“x”** WinDbg command. Good for meditation during slow debugging sessions. All music is very sad and mystical. I even think about this title: **“Musical Biography of a Bug“**.

VISUAL COMPUTER MEMORIES

Looking at computer memory visual images (Volume 1, page 556) combined with listening to the incredible nostalgic music composed by Oystein Sevag (Visual and Link albums) is highly recommended to relieve stress while immersing yourself in the vast depths of memory hierarchy. I really like “Painful Love” tracks. Is love and passion for programming painful?…

THE FIRST DEFECT

Although the first bug (the real one) was found in Mark II the first general-purpose electronic computer was ENIAC and surely it had defects like overflow. In 80s there was an electronic music group called Software. I have a few CDs and listen to them sometimes. One album is called Software-Visions and it has these tracks (one of them is called Xenix - Microsoft UNIX-based OS):

1. Software Visions
2. Secrets
3. Stranger
4. Realtime
5. Mainframe
6. Snobol
7. Xenix
8. Syntax
9. Overflow
10. Interface
11. Eniac

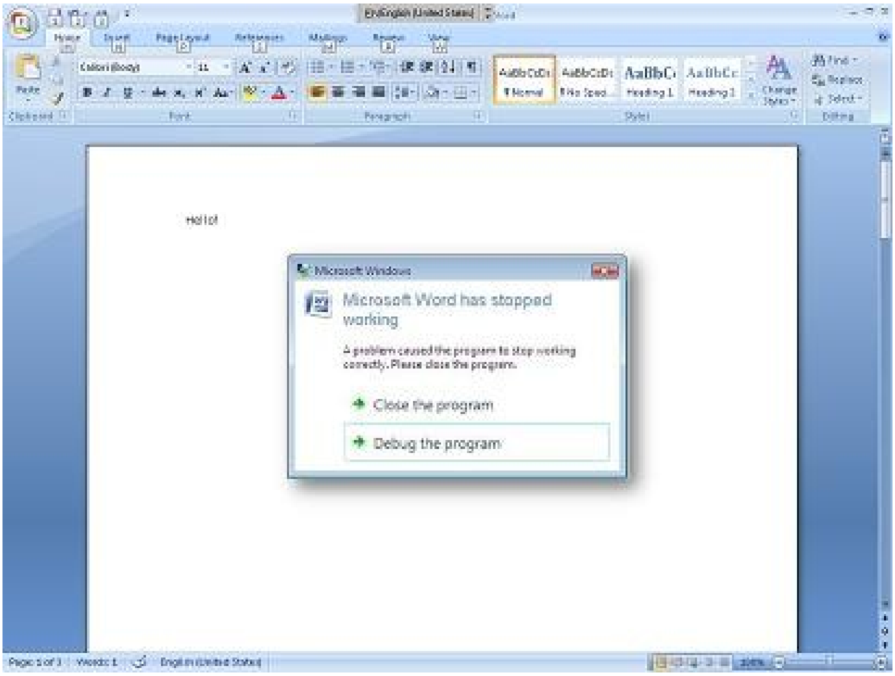
THE SONGS FOR REMOTE DEBUGGING

"The Songs of Distant Earth" is my favorite Mike Oldfield album. Highly recommended to keep optimism when doing remote debugging on different systems. Here is my alternative track naming:

1. The Decision To Go Remote
2. Let There Be A Connection
3. Super System Crash
4. Connection Established
5. First Break In
6. The Sea Of Threads
7. Setting Breakpoints
8. Prayer For A Match
9. Lament For Users
10. The Kernel
11. Screensaver Starts
12. Tabular Output
13. The Shining Threads
14. Breakpoint Match
15. The Sunken Debugger
16. Contemplating Observations
17. A New Session

THINKING OUT OF THE BOX

Q. Every time you open the specific Microsoft Word 2007 document on Vista WER error message box appears on the screen:



Microsoft Windows

Microsoft Word has stopped working

A problem caused the program to stop working correctly. Please close the program.

What might be the cause of it?

You can find the correct answer in the comments here:

http://www.dumpanalysis.org/blog/index.php/2008/03/07/thinking-out-of-the-box/

CRASH DUMPS AND SCIENCE FICTION

In Dan Simmons’s book "The Fall of Hyperion" in chapter 33 on page 303 we read a poetic description of a process crash (italics are mine):

“Johnny twists a second in the AI’s massive grip *(fault injection)*, and then his analog - Keats’s small but beautiful body *(GUI)* - is torn, compacted, smashed into an unrecognizable mass *(corrupt dump)* which Ummon sets against his megalith flesh *(private bytes)*, absorbing the analogs’ remains *(overwriting discarded pages)* back into the orange-and-red depths of itself *(working set)*.”

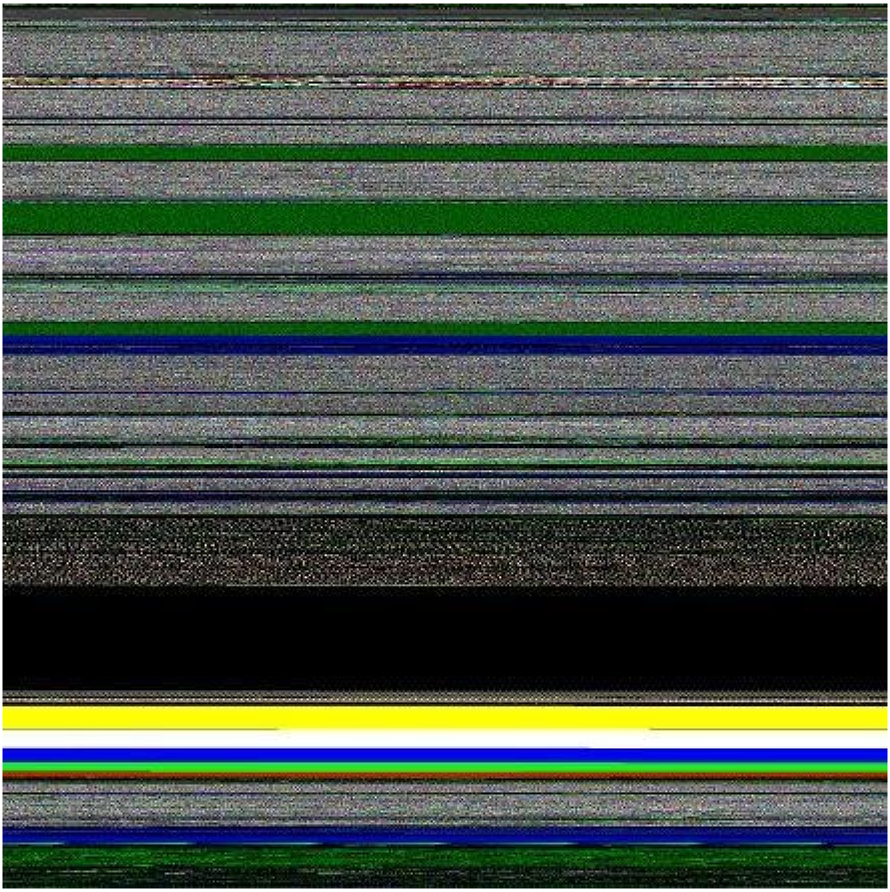
COLOMETRIC COMPUTER MEMORY DATING

Similar to radiometric dating using isotopes we can use memory visualization techniques to see distribution of allocated buffers and their retention over time. The key is to allocate colored memory. For example, to append a red buffer that contains RGBA values 0xFF000000 to specific allocations.

We call these colored memory marks **isomemotopes**. We can either inject a different isomemotope for a different data or change the isomemotope over time to mark specific allocation times. The following test program allocates buffers marked by a different amount of different isomemotopes every time:

```
#include "stdafx.h"
#include <stdlib.h>
#include <memory.h>
#include <windows.h>

typedef unsigned int ISOMEMOTOPE;

void *alloc_and_mark_with_isomemotope(size_t size,
                                      ISOMEMOTOPE color,
                                      size_t amount)
{
  char *p = (char *)malloc(size+amount);

  for (char *isop = p+size;
       p && isop  < p+size+amount;
       isop+=sizeof(ISOMEMOTOPE))
  {
    *(ISOMEMOTOPE *)isop=color;
  }

  return p;
}

int _tmain(int argc, _TCHAR* argv[])
{
  alloc_and_mark_with_isomemotope(0x1000,
                                  0xFF000000, // red
                                  0x10000);
  alloc_and_mark_with_isomemotope(0x1000,
                                  0x00FF0000, // green
                                  0x20000);
  alloc_and_mark_with_isomemotope(0x1000,
                                  0x0000FF00, // blue
                                  0x30000);
  alloc_and_mark_with_isomemotope(0x1000,
                                  0xFFFFFF00, // white
                                  0x40000);
  alloc_and_mark_with_isomemotope(0x1000,
                                  0xFFFF0000, // yellow
                                  0x50000);

  DebugBreak();

  return 0;
}
```

Corresponding **Dump2Picture** (Volume 1, page 532) image is this (0×00000000 address is at the bottom) and also featured on the back cover in full color:



ON CSI ABBREVIATION

In the article about memory dump analysis as forensic science:

http://blogs.msdn.com/ntdebugging/archive/2008/04/15/the-digital-dna-of-bugs-dump-analysis-as-forensic-science.aspx

CSI was proposed to mean “Crashed Server Investigation”. After reading the book by E. J. Wagner about the emergence of forensic science in 19th and 20th centuries:

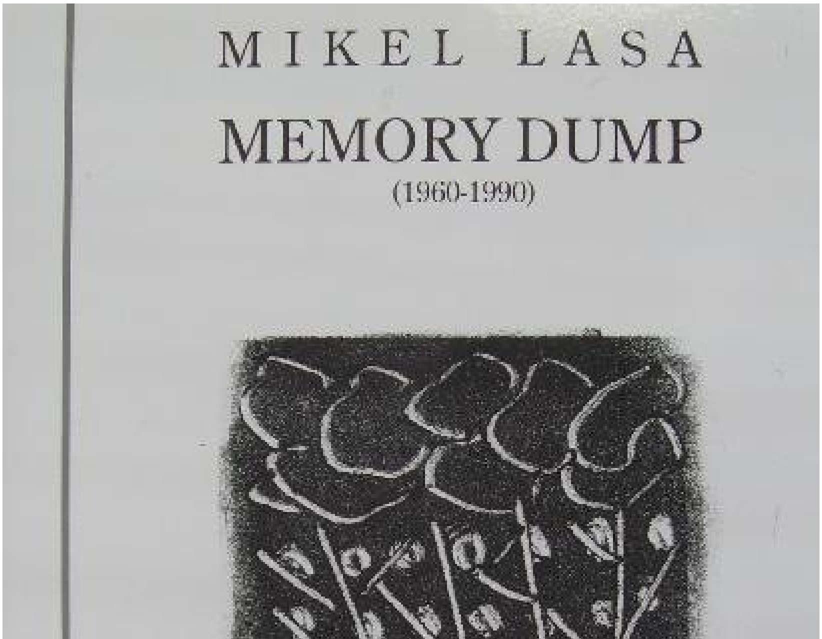
"The Science of Sherlock Holmes: From Baskerville Hall to the Valley of Fear, the Real Forensics Behind the Great Detective’s Greatest Cases".

I was rethinking CSI again and found these similar meanings:

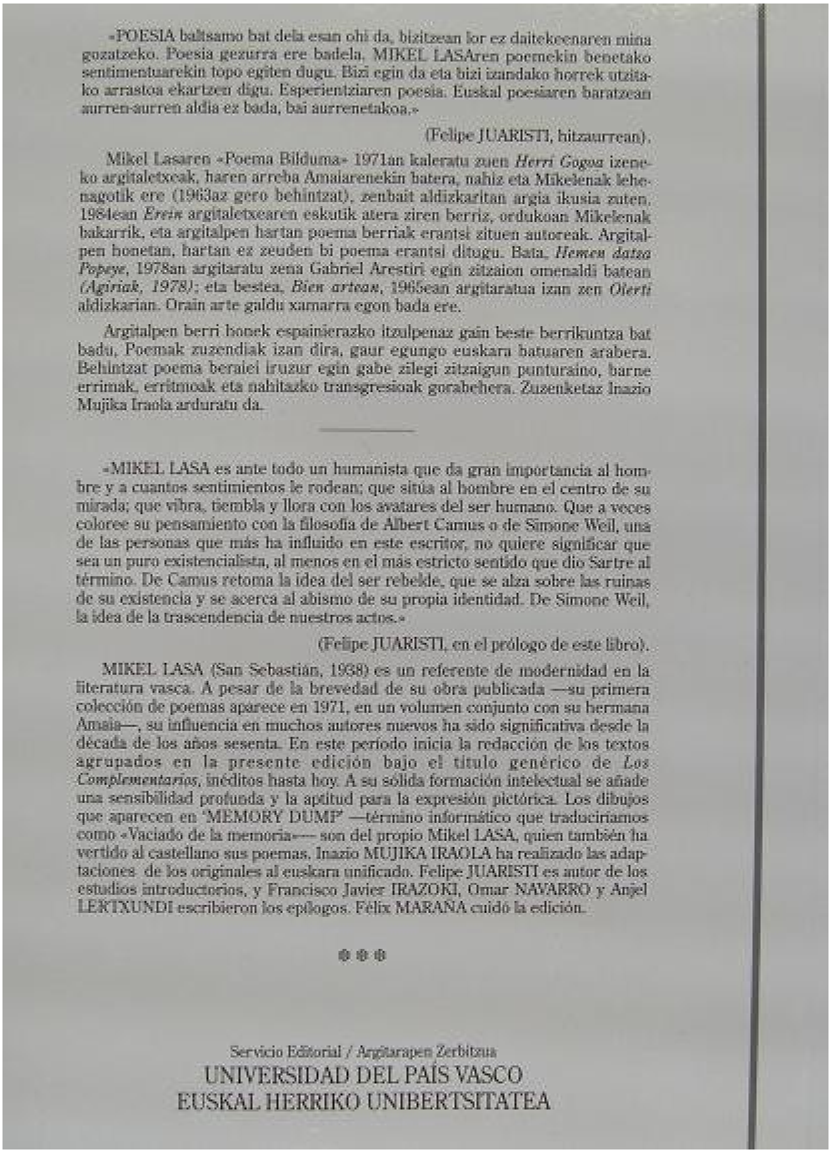
- Crashed Software Investigation
- Crashed System Investigation

THE FIRST MEMORY DUMP BOOK

The priority goes to Mikel Lasa "Memory Dump: 1960-1990":


MIKEL LASA
MEMORY DUMP
(1960-1990)
UNIVERSIDAD DEL PAÍS VASCO
EUSKAL HERRIKO UNIBERTSITATEA
10
POESÍA VASCA HOY
GAUR EGUNGO EUSKAL POESIA

On the back cover the author explains why he chose "MEMORY DUMP" computer terminology for the title of the book:



«POESIA baltsamo bat dela esan ohi da, bizitzean lor ez daitekeenaren mina gozatzeko. Poesia gezurra ere badela, MIKEL LASAren poemekin benetako sentimentuarekin topo egiten dugu. Bizi egin da eta bizi izandako horrek utzitako arrastoa ekartzen digu. Esperientziaren poesia. Euskal poesiaren baratzean aurren-aurren aldia ez bada, bai aurrenetakoa.»

(Felipe JUARISTI, hitzaurrean).

Mikel Lasaren «Poema Bilduma» 1971an kaleratu zuen *Herri Gogoa* izeneko argitaletxeak, haren arreba Amaiarenekin batera, nahiz eta Mikelenak lehenagotik ere (1963az gero behintzat), zenbait aldizkaritan argia ikusia zuten. 1984ean *Erein* argitaletxearen eskutik atera ziren berriz, ordukoan Mikelenak bakarrik, eta argitalpen hartan poema berriak erantsi zituen autoreak. Argitalpen honetan, hartan ez zeuden bi poema erantsi ditugu. Bata, *Hemen datza Popeye*, 1978an argitaratu zena Gabriel Arestiri egin zitzaion omenaldi batean *(Agiriak, 1978)*; eta bestea, *Bien artean*, 1965ean argitaratua izan zen *Olerti* aldizkarian. Orain arte galdu xamarra egon bada ere.

Argitalpen berri honek espainierazko itzulpenaz gain beste berrikuntza bat badu. Poemak zuzendiak izan dira, gaur egungo euskara batuaren arabera. Behintzat poema beraiei iruzur egin gabe zilegi zitzaigun punturaino, barne errimak, erritmoak eta nahitazko transgresioak gorabehera. Zuzenketaz Inazio Mujika Iraola arduratu da.

«MIKEL LASA es ante todo un humanista que da gran importancia al hombre y a cuantos sentimientos le rodean; que sitúa al hombre en el centro de su mirada; que vibra, tiembla y llora con los avatares del ser humano. Que a veces coloree su pensamiento con la filosofía de Albert Camus o de Simone Weil, una de las personas que más ha influido en este escritor, no quiere significar que sea un puro existencialista, al menos en el más estricto sentido que dio Sartre al término. De Camus retoma la idea del ser rebelde, que se alza sobre las ruinas de su existencia y se acerca al abismo de su propia identidad. De Simone Weil, la idea de la trascendencia de nuestros actos.»

(Felipe JUARISTI, en el prólogo de este libro).

MIKEL LASA (San Sebastián, 1938) es un referente de modernidad en la literatura vasca. A pesar de la brevedad de su obra publicada —su primera colección de poemas aparece en 1971, en un volumen conjunto con su hermana Amaia—, su influencia en muchos autores nuevos ha sido significativa desde la década de los años sesenta. En este período inicia la redacción de los textos agrupados en la presente edición bajo el título genérico de *Los Complementarios*, inéditos hasta hoy. A su sólida formación intelectual se añade una sensibilidad profunda y la aptitud para la expresión pictórica. Los dibujos que aparecen en 'MEMORY DUMP' —término informático que traduciríamos como «Vaciado de la memoria»— son del propio Mikel LASA, quien también ha vertido al castellano sus poemas. Inazio MUJIKA IRAOLA ha realizado las adaptaciones de los originales al euskara unificado. Felipe JUARISTI es autor de los estudios introductorios, y Francisco Javier IRAZOKI, Omar NAVARRO y Anjel LERTXUNDI escribieron los epílogos. Félix MARAÑA cuidó la edición.

Servicio Editorial / Argitarapen Zerbitzua
UNIVERSIDAD DEL PAÍS VASCO
EUSKAL HERRIKO UNIBERTSITATEA

ON SOS ABBREVIATION

When reading a book about overparenting I came across **SOS** abbreviation used in parenting and education context: **S**tressed **O**ut **S**tudents (Carl Honore, Under Pressure). Immediately a thought struck me that **SOS** might also mean two sometimes related conditions:

- **S**tressed **O**perating **S**ystem
- **S**piking **O**perating **S**ystem

May also mean **S**inking **O**perating **S**ystem (in production environment context).

SOFTWARE EXCEPTIONS: A PARANORMAL VIEW

Some view minds as software and some view software as minds. There is also mind / body problem for humans and less known mind / body problem for computers. This is what I define as ”**Metaphorical Bijection**“. Some view minds as constrained by brains. Therefore we can say that software might be constrained by hardware too and exceptions (faults) arise when software is accidentally written for hardware or another software if hardware is virtualized, simulated, without limitations that constrain software execution. The current hardware constrains that accidentally written software and generates faults because it cannot deal with paranormal effects.

BUG ENTANGLEMENT (BUGTANGLEMENT)

We all noticed how software bugs twist together or entwine into a confusing mass of an intricate trap that complicates and confuses debugging. This is the so called **Bug Entanglement** or just **Bugtanglement**, the new term inspired by quantum entanglement in quantum mechanics. We don't see a software bug until an observer makes a measurement… And how uncertain these measurements (memory dumps, for example) are! If an observer interferes, it is not the same system anymore, like we see it from observation. And once we made our measurement, the software system continues to evolve according to its internal design function which we never know fully and only approximate with our requirements specifications.

Welcome to **Quantum Theory of Software Bugs**. it was discovered more than 10 years ago by Bernard Robertson-Dunn:

Quantum Theory of Software
(http://www.anu.edu.au/mail-archives/link/link9712/0115.html)

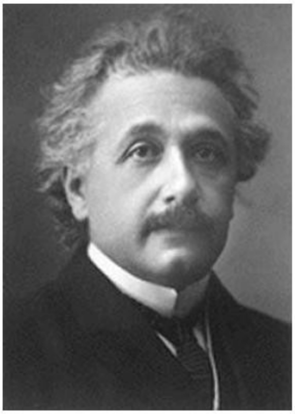
THE STANDARD MODEL OF DEBUGGING

This is a simply-symmetrical model consisting of **Bugluon** - **Debugluon** pair of particles where one is a particle and the other is the corresponding antiparticle. The interaction between them is completely of non-gravitational nature. When they annihilate we get the light at the end of a long debugging tunnel, called Large Hard-debugging Collider (LHC). A bugluon particle moving in memory space usually leaves traces and various defects. A photographic picture of tracks left by bugluons is called a memory space dump. The analysis of various track patterns is called memory dump analysis.

PHYSICS OF DEBUGGING

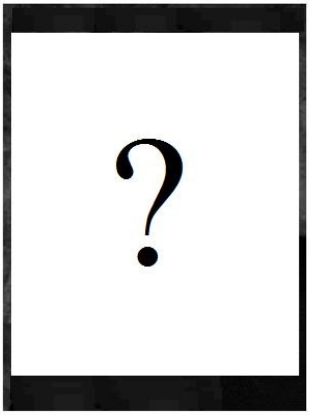
Elaborating on threads in abstract space idea (Volume 1, page 503) we can try to apply canonical formalism of classical mechanics. Thread kinematics involves two abstract coordinates $\mathbf{q_1}$ and $\mathbf{q_2}$ which correspond to memory addresses and their dereferenced values respectively. Although these are discrete variables ($\mathbf{N}$), we can generalize them to be continuous ($\mathbf{R_+}$). The motivation lies in the discreteness of physical measurement: if we divide [0,1] interval into 2^{64} sub-intervals we get approximately 5.421e-20 values which are small indeed even by today's experimental standards. Next we introduce dynamic variables called $\mathbf{v_1}$ and $\mathbf{v_2}$ which correspond to the rate of change of an address and the rate of change of a value respectively. These are called generalized velocities. These can also be continualized according to the same line of thought we used for generalized coordinates. So finally we have $\mathbf{R_+}^{\mathbf{2}}$ x $\mathbf{R_+}^{\mathbf{2}}$ space. $\mathbf{R_+}^{\mathbf{2}}$ can be complexificated into the subset of $\mathbf{C}$ and we get the subset of $\mathbf{C}^{\mathbf{2}}$. If we allow negative addresses and values we get full $\mathbf{R}^{\mathbf{2}}$ x $\mathbf{R}^{\mathbf{2}}$ space or, after complexification, the full complex $\mathbf{C}^{\mathbf{2}}$ space which is well-known for its magic in physical theories. If we have N threads we get $\mathbf{C}^{\mathbf{2n}}$ space.

Now we can go forward and employ all apparatus of classical physics. Just one final remark for now, we need to call the particle: I propose to name it **classical μ-memuon**.



Albert Einstein

Dmitry Vostokov[1]



[1] The founder of Physics of Debugging

CAN COMPUTERS DEBUG?

Consider an application randomly crashing at different addresses or hanging sometimes. One day we are lucky to get this process postmortem memory dump:

```
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
(f34.c6c): Access violation - code c0000005 (first/second chance not
available)
eax=73726946 ebx=00403378 ecx=656c2070 edx=656c2074 esi=00403374
edi=00000004
eip=7d64d233 esp=0012ff24 ebp=0012ff4c iopl=0 nv up ei pl nz ac pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b  efl=00010217
ntdll!RtlpWaitOnCriticalSection+0xdf:
7d64d233 ff4014   inc     dword ptr [eax+14h]  ds:002b:7372695a=????????
```

Aha! It involves critical sections! Let's see whether we have an instance of **Critical Section Corruption** pattern (page 324). The first disappointment comes when **!locks** command takes ages to finish so we break it:

```
0:000> !locks

Stopped scanning because of control-C

Scanned 154686373 critical sections
```

Next we try to list all critical sections but without any success:

```
0:000> !locks -v

CritSec at 00000000 could not be read
Perhaps the critical section was a global variable in a dll that was
unloaded?

CritSec at 00000000 could not be read
Perhaps the critical section was a global variable in a dll that was
unloaded?

CritSec at 00000000 could not be read
Perhaps the critical section was a global variable in a dll that was
unloaded?
```

```
CritSec at 00000000 could not be read
Perhaps the critical section was a global variable in a dll that was
unloaded?

...
```

Next we look at the stack trace to find the address of the critical section (shown in small font for visual clarity):

```
0:000> kv
ChildEBP RetAddr  Args to Child
0012ff4c 7d628576 64726f77 00000004 00000000 ntdll!RtlpWaitOnCriticalSection+0xdf
0012ff6c 00401074 00403374 00403394 00000001 ntdll!RtlEnterCriticalSection+0xa8
0012ff7c 004011e9 00000001 004d2fc0 004d3030 application!wmain+0×74
0012ffc0 7d4e7d2a 00000000 00000000 7efde000 application!__tmainCRTStartup+0×10f
0012fff0 00000000 00401332 00000000 00000000 kernel32!BaseProcessStart+0×28

0:000> dt CRITICAL_SECTION 00403374
application!CRITICAL_SECTION
   +0×000 DebugInfo        : 0×73726946 _RTL_CRITICAL_SECTION_DEBUG
   +0×004 LockCount        : 1701585008
   +0×008 RecursionCount   : 1919251571
   +0×00c OwningThread     : 0×20666f20
   +0×010 LockSemaphore    : 0×64726f77
   +0×014 SpinCount        : 0×73
```

It looks corrupt indeed so let’s see if it has ASCII fragments in its data:

```
0:000> db 00403374
00403374  46 69 72 73 70 20 6c 65-73 74 65 72 20 6f 66 20  Firsp lester of
00403384  77 6f 72 64 73 00 00 00-00 00 00 00 02 00 00 00  words………..
...

0:000> da 00403374
00403374 “Firsp lester of words”
```

It looks like the garbled sentence “Firs**t** le**t**ter of words”. Who wrote this? Sherlock would say: “Elementary, my dear Watson”, take the first letters, literally: “**F**irst **l**etter **o**f **w**ords”. **Flow.dll** component or a component with similar name causes corruption at random addresses! We can’t believe this, run **lm** WinDbg command and to our astonishment we see **Flows** module:

```
0:000> lm
start    end        module name
00400000 00405000   application
00410000 004ab000   advapi32
71c20000 71c32000   tsappcmp
75490000 754f5000   usp10
77ba0000 77bfa000   msvcrt
78130000 781cb000   msvcr80
7d4c0000 7d5f0000   kernel32
7d600000 7d6f0000   ntdll
7d800000 7d890000   gdi32
7d8d0000 7d920000   secur32
7d930000 7da00000   user32
7da20000 7db00000   rpcrt4
7dbc0000 7dbc9000   Flows
7dee0000 7df40000   imm32

Unloaded modules:
77b90000 77b98000   VERSION.dll
76920000 769e2000   USERENV.dll
71c40000 71c97000   NETAPI32.dll
771f0000 77201000   WINSTA.dll
770e0000 771e8000   SETUPAPI.dll
004e0000 00532000   SHLWAPI.dll
69500000 69517000   faultrep.dll
```

If we check the module information we would see that it is the part of some unstable 3rd-party hookware and removing it solves the problem of elusive crashes. The example of the problem solving power of mind is a bit contrived but my point here is that there are problems computers would never debug and troubleshoot. Answering the question of Dreyfus' book "What computers still can't do" (ISBN 978-0262540674): they still can't debug…

PART 7: DATA RECOVERY

WITH THE HELP OF MEMORY DUMP ANALYSIS

My friend was typing a long message in IE to one of his old schoolmates that he had just found on Internet. He spent about an hour writing and rewriting and when finally hit the Send button he got a page saying that connection was probably lost. Going back in URL history brought the empty edit box and all data was lost. Or was it? He called me and I immediately advised him to save a crash dump of iexplore.exe using Task Manager (Vista). I also asked him for a word he used to start his message. It was "Hello" in Russian. I got his dump file and opened it in WinDbg. Because the language of his message was Russian I assumed that it was still there in local buffers or heap entries in UNICODE format so I typed "ello" in Notepad and saved this in a Unicode text file. I loaded it in a Visual C++ binary editor (I used Visual C++) and found the following sequence of bytes:

```
40 04 38 04 32 04 35 04 42 04
```

Then I did a search in WinDbg for this sequence from the first loaded module address till the end of user space:

```
0:000> lm
start    end         module name
003c0000 0045b000    iexplore
...

0:000> s 003c0000 L?7FFFFFFF 40 04 38 04 32 04 35 04 42 04
...
048971e4 40 04 38 04 32 04 35 04-42 04 2c 00 20 00 1c 04  @.8.2.5.B.,. ...
...
08530fe4 40 04 38 04 32 04 35 04-42 04 2c 00 20 00 1c 04 @.8.2.5.B.,. ...
...
201ea65c 40 04 38 04 32 04 35 04-42 04 2c 00 20 00 1c 04 @.8.2.5.B.,. ...
...
```

The number of found entries was big and I decided to output every entry into a file using the following script:

```
.foreach ( address { s-[1]b 003c0000 L?7FFFFFFF 40 04 38 04 32 04 35 04 42
04 }) {.writemem c:\dmitry\ieout${address}.txt ${address}-10
${address}+1000}
```

I got numerous files:

```
C:\dmitry>dir ieout*.txt
...
09/06/2008  08:53               4112 ieout0x048971e4.txt
09/06/2008  08:53               4112 ieout0x0489784c.txt
09/06/2008  08:53               4112 ieout0x0489b854.txt
09/06/2008  08:53               4112 ieout0x0489bc5c.txt
...
```

I combined all of them into one big file and sent it to my friend:

```
C:\dmitry>type ieout0x*.txt >ieoutall.txt
```

The file contained not only the final message but all intermediate typing histories too. He was very happy.

PART 8: SOFTWARE TROUBLESHOOTING

TROUBLESHOOTER’S BLOCK

Many engineers had a problem when they didn’t know what question to ask. This is called **Troubleshooter’s Block** by analogy with famous Writer’s Block. If such block happens we can advise to turn to the list of questions and try to find the similar one to our problem or assemble the new one based on some analogy. For example, Citrix engineers use Brief Troubleshooting Guide:

http://support.citrix.com/article/ctx106727

It contains plenty of questions that can be used as a template.

CAUSAL MODELS

Looking at traces, system and application event logs and logs from other tools, technical support engineers see correlations between various events and build causal models that are used to trace symptoms back to their causes. They use prior knowledge, assumptions, informed guessing and event order to discern causal structure. Clearly event order in logs influences that so it is important to understand how we think in causal terms in order to learn about our biases.

Another important question from software engineering perspective is how to design tracing components to help technical support and software maintenance engineers build correct causal models of software issues.

OBJECT-ORIENTED DEBUGGING AND TROUBLESHOOTING

OODT (pronounced ”oddity”) is not a paradigm shift for support and software maintenance environments but a recognized way to solve problems using object-oriented techniques. In contrast to **Structured Debugging and Troubleshooting** methods (**SDT**) where engineers have sequence of questions and structure troubleshooting plans around them OODT is based on targeting specific objects, subsystems and systems (sending “messages” to them) and evaluating response and changes in their behavior.

Note: OODT doesn’t mean troubleshooting OO systems - it means applying OO techniques to troubleshooting.

COMPONENT-BASED DEBUGGING AND TROUBLESHOOTING

Component identification is one of the main goals of post-mortem memory dump analysis and troubleshooting process in general. Using the definition of components as units of deployment and 3rd-party composition taken from Clemens Szyperski's seminal book discussing component software in general and COM, CORBA, Java and .NET in particular (Component Software: Beyond Object-Oriented Programming, 2nd Edition, ISBN 978-0201745726) we can say that **CBDT** is centered around component isolation and replacement.

DOMAIN-DRIVEN DEBUGGING AND TROUBLESHOOTING

SDT (Structured Debugging and Troubleshooting) is procedural (action-based). Once we get the description of the problem we jump to actions:

1. Ask this
2. Ask that
3. Do this
4. Do that
5. …

Whereas **OODT** (page 379) is centered around objects (systems and customers are also objects):

1. Get objects from the problem description and problem environment
2. Interrogate them sending messages (could be an email at high levels of organizational structure) like changing a registry key is a message to configuration management subsystem
3. ...

OODT depends on troubleshooting domain and therefore finally we finally come to **DDDT**.

MYTHS AND FACTS ABOUT SOFTWARE SUPPORT

There are widespread myths about software technical support. The first one is:

Technical support engineers can’t and don’t write code (myth). Technical support engineers do write code and sometimes fairly advanced one (fact).

There is a prevalent view of a technical support engineer spending all the time on the phone as a shield from introvert software engineers who hate customers. This is not true. There are usually several layers of support from very basic ones requiring only customer communication and foreign language skills to the very advanced problem identification and troubleshooting skills that requires a thousand page knowledge from Windows Internals book. My point here is that advanced troubleshooting requires tools that sometimes don’t exist and this prompts support engineers to develop their own. Sometimes it is easy to query information from the customer environment and/or fix the problem on the spot by writing a tool or a script. And this is pure unconstrained development limited only by individual imagination, skills and complexity of the task.

The weak form of this myth is the view of a support engineer using only Visual Basic or its scripting variant.

CETERIS PARIBUS IN COMPARATIVE TROUBLESHOOTING

Ceteris Paribus means *"with other things being the same"* (Latin) and when applied to software troubleshooting and debugging means equal environment and configuration. My favorite example is troubleshooting an issue using two Citrix CDF traces (ETW based): one is for the problem and another for the expected behavior. Say we have a terminal services connectivity problem where a published application doesn't start on the one particular server in Citrix farm. Here **Ceteris Paribus** means that the application, connection method, configuration, user name, and so on, are all the same for both traces.

Looks like Latin is used here to obfuscate something obvious but surely many engineers forget it when facing complex issues. This equally applies to debugging as well.

DANCING IN SOFTWARE SUPPORT ENVIRONMENT

From *“coordinated coping”* to *“what feels like chaos can feel like a dance - a fast one”* - say Francoise Tourniaire and Richard Farrell in their influential book “The Art of Software Support”. I totally agree and from my observation the most successful (not stressful) people in software support are “dancing” when dealing with everyday and hot customer issues. You see them literally thriving on inherent chaos. Truly multitaskers and even multithreaders!

PARTS: PROBLEM SOLVING POWER OF THOUGHT

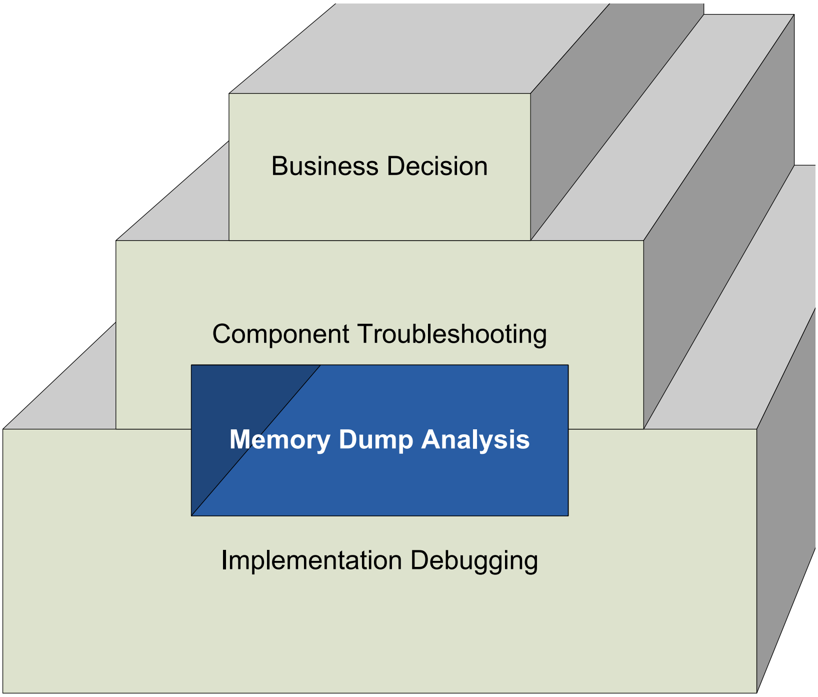
Problem **A**nalysis and **R**esolution **T**roubleshooting **S**ystem (**PARTS**) is the new troubleshooting methodology for critical problem analysis and resolution. It consists of **P**roblem **A**nalysis and **R**esolution **T**asks (**PART**s). The motivation to create this system came to me after looking at various software support processes in various companies around the globe, how they relate to software engineering methodologies and the scientific method, and finally after looking at “The Master Key System” devised by Charles Haanel almost 100 years ago. Borrowing the idea of “Creative Power of Thought” I subtitle PARTS as *Problem Solving Power of Thought*.

THE HIDDEN TOMB IN PYRAMID OF SOFTWARE CHANGE

How does software change in a production environment? My experience suggests 3 major ways:

1. Executive decision to replace the whole software product with another competing product.
2. Software troubleshooting at component level like upgrading or eliminating suspicious components and unrelated products that influence behaviour.
3. Correction of individual components after debugging to address implementation and functional defects, non-functional, design or architecture deficiencies.

This can be shown on the following rough diagram (excluding possible overlapping of levels) highlighting the often hidden role of memory dump analysis in software change:



TRACING

CDF TRACES: ANALYZING PROCESS LAUNCH SEQUENCE

Citrix CDF traces are based on ETW (Event Tracing for Windows) and therefore Citrix customers, their support personnel and developers can use MS TraceView tool for troubleshooting Citrix terminal service environments:

Viewing Common Diagnostics Facility (CDF) Traces Using TraceView
http://support.citrix.com/article/ctx106233

In cases with slow logon or slow process startup we can analyze process launch sequence to determine delays. In the output trace we can filter tzhook module messages which also contain session id (this is quite handy to differentiate between different sessions), for example (shown in smaller font for visual clarity):

```
PID   TID   TIME         MESSAGE
21864 21912 06:34:53.598 tzhook: Attach on process - cmd.exe session=51
21620 20372 06:34:59.754 tzhook: Attach on process - acregl.exe session=51
18668 21240 06:35:02.704 tzhook: Attach on process - cmstart.exe session=51
18560 18832 06:35:02.735 tzhook: Attach on process - wfshell.exe session=51
18204 20060 06:35:06.575 tzhook: Attach on process - icast.exe session=51
20640 21104 06:35:07.717 tzhook: Attach on process - LOGON.EXE session=51
21188 21032 06:35:07.938 tzhook: Attach on process - cscript.exe session=51
21888 19592 06:35:11.157 tzhook: Attach on process - WScript.exe session=51
20600 20732 06:35:11.780 tzhook: Attach on process - admin.exe session=51
17976 20456 06:35:18.752 tzhook: Attach on process - winlogon.exe session=53
21332 13156 06:35:51.625 tzhook: Attach on process - mpnotify.exe session=53
10988 10732 06:35:57.043 tzhook: Attach on process - rundll32.exe session=53
```

Here is another process launch sequence for published Notepad application:

```
PID     TID     TIME          MESSAGE
15828   18720   15:34:02.637  tzhook: Attach on process - winlogon.exe session=2
5300    18508   15:34:03.043  tzhook: Attach on process - mpnotify.exe session=2
17948   19300   15:34:03.356  tzhook: Attach on process - userinit.exe session=2
17956   19316   15:34:03.415  tzhook: Attach on process - cmd.exe session=2
5384    5324    15:34:03.443  tzhook: Attach on process - cmd.exe session=2
19432   19264   15:34:03.461  tzhook: Attach on process - SSONSVR.EXE session=2
12480   7472    15:34:03.466  tzhook: Attach on process - cmd.exe session=2
19448   19364   15:34:03.474  tzhook: Attach on process - net.exe session=2
19416   19656   15:34:03.489  tzhook: Attach on process - acregl.exe session=2
19480   19596   15:34:03.544  tzhook: Attach on process - cmstart.exe session=2
664     19512   15:34:03.559  tzhook: Attach on process - wfshell.exe session=2
19904   13140   15:34:03.610  tzhook: Attach on process - net.exe session=2
6864    20036   15:34:03.746  tzhook: Attach on process - icast.exe session=2
19540   20016   15:34:03.749  tzhook: Attach on process - ctfmon.exe session=2
19944   19032   15:34:03.757  tzhook: Attach on process - net.exe session=2
10232   18356   15:34:03.787  tzhook: Attach on process - notepad.exe session=2
```

Such sequences are also useful to determine a process upon which the session initialization or startup sequence hangs. In this case a user dump of that process might be useful.

Of course we can do all this with Process Monitor and other similar tools but here we get other Citrix related trace messages as well. All in one.

ETW TRACING TOOLS

There are few tools currently available. Because I work with mainly Citrix terminal service environments I put links to Citrix articles where possible:

- CDFControl

  http://support.citrix.com/article/ctx111961

- Citrix ASC/AMC

  http://support.citrix.com/article/ctx104578

- tracelog

  http://support.citrix.com/article/ctx111405

- Windows Performance Tools Kit

  http://www.microsoft.com/whdc/system/sysperf/perftools.mspx

LEAN TRACING

Sometimes ETW (or CDF) traces can be really huge. Unless we trace the elusive but the specific error we already know about, there is no need to make such traces if we can reproduce the issue. My favourite example is connectivity problems when you cannot connect to a terminal services server. The best way is to start tracing, try to connect, get an error and stop tracing. Usually it takes no more than a couple of minutes. We can even trace all modules here just to make sure that we don't miss anything. It is also better to focus on one specific scenario per one lean trace instead of packing several of them into one big trace.

DEBUGWARE PATTERNS

API QUERY

Debugware patterns are solutions for common recurrent problems occurring during design of troubleshooting and debugging tools. This is the first pattern.

Software products use various API and external interfaces to query data or get notifications from operating system environment. Their behaviour depends on API return values or output parameters that are not always logged or logged with insufficient detail. In order to reproduce or diagnose problems an engineer can write a small API Query tool that will periodically or asynchronously query the same set of API and log their input and output data. If the problem happens with the product at some point this additional log will help in problem identification and resolution.

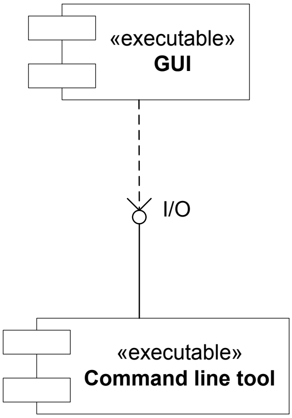
Examples include:

TSUserLog
http://support.citrix.com/article/CTX114179

WindowHistory
http://support.citrix.com/article/CTX106985

TOOL FAÇADE

This pattern is used when there is a tool with a cumbersome interface like command line with many options and there is a need for a better and easier way to use that interface, for example, GUI. This can be illustrated on the following UML component diagram:



Excellent example of this pattern is:

StressPrinters
http://support.citrix.com/article/CTX109374

This tool was designed as a GUI wrapper around command line tool AddPrinter. Adding the powerful GUI interface allowed to extend its functionality and even find new domains, like testing, where the tool can be used.

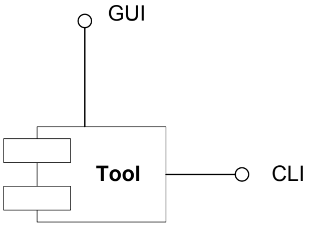
CONFIGURATION WRAPPER

Many products have lots of configuration parameters stored in OS configuration database, Windows registry. Some of parameters are internal and some are public but never exposed via product GUI or management consoles. Configuration parameters can be related to product functionality or can make troubleshooting and debugging easier, for example, additional tracing parameters to set the verbosity level of debugging output or enable additional safety checks. These parameters can be scattered across different registry branches or keys. Therefore Configuration Wrapper addresses the problem and this pattern is frequently seen in troubleshooting and debugging tools, for example:

Gflags (http://msdn.microsoft.com/en-us/library/cc265944.aspx)

DUAL INTERFACE

Good troubleshooting tools usually have two interfaces: one is graphical (GUI) and the other is command line (CLI). The latter is very useful when GUI console is not available or there is a need to automate the tool. Both interfaces can be implemented in one component:

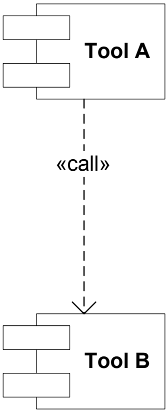


or there could be a separate GUI wrapper for complex CUI interface or when CUI interface was developed earlier and we don't want to touch the code (see **Tool Façade** pattern, page 392). Therefore this common pattern is called **Dual Interface**. Some tool examples:

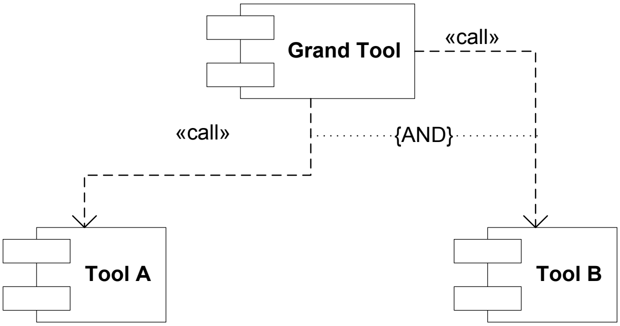
Gflags (http://msdn.microsoft.com/en-us/library/cc265901.aspx)
CDFControl (http://support.citrix.com/article/CTX111961)
SystemDump (http://support.citrix.com/article/ctx111072)

TOOL CHAIN

Usually after writing and using a tool we think about an extension of it or we find another tool that is suitable as that extension. In such cases it is better to reuse existing components and adapt the former tool to use the latter. A programming analogy could be a compiler and linker:

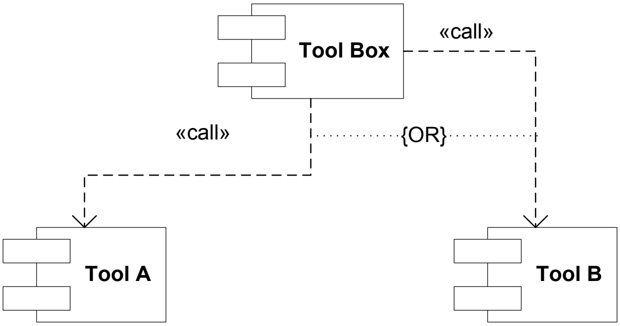


Typical example could be a logging tool that now acquires log viewer functionality by implementing a button that launches a separate log viewer or vice versa, a log viewer that can now do logging. Another simple example is a command file that launches different tools in a sequence. The tools might not be related by the data they produce and operate upon: this not the same as well-known architectural *Filters and Pipes* pattern. What is important is the fact that tools are related buy overall tool chain goal, for example, to debug problems by recording and analyzing log files or monitor some process activity and periodically create memory dumps:



TOOL BOX

This is a pattern that logically flows from **Tool Chain** (page 395). Their principal difference is that the latter launches subordinated tools in a sequence to reach the common goal and the former is the tool that launches other tools in any sequence independently:



Often tool boxes are implemented as toolbars. Another boundary example is the so called resource kit tools where an HTML page or taskbar menu serves the role of tool box.

PART 9: SECURITY

DATA HIDING IN CRASH DUMPS

Suppose we want to send a complete memory dump to a vendor but want to remove certain sensitive details or perhaps the whole process or image from it. In this case we can use **f** WinDbg command (virtual addresses) or **fp** (physical addresses) to fill pages with zeroes. Let's open a complete memory dump and erase environment variables for a process:

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fffffadfe7afd8e0
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 0014a000  ObjectTable: fffffa8000000c10  HandleCount: 730.
    Image: System

PROCESS fffffadfe6edc040
    SessionId: none  Cid: 0130    Peb: 7fffffdf000  ParentCid: 0004
    DirBase: 34142000  ObjectTable: fffffa80009056d0  HandleCount:  19.
    Image: smss.exe

...

PROCESS fffffadfe67905a0
    SessionId: 0  Cid: 085c    Peb: 7fffffd4000  ParentCid: 0acc
    DirBase: 232e2000  ObjectTable: fffffa8000917e10  HandleCount:  55.
    Image: SystemDump.exe

kd> .process /r /p fffffadfe7287610
Implicit process is now fffffadf`e7287610
Loading User Symbols

kd> !peb
PEB at 000007fffffd4000
...
    Environment:  0000000000010000

kd> dd 10000
00000000`00010000  004c0041 0055004c 00450053 00530052
00000000`00010010  00520050 0046004f 004c0049 003d0045
00000000`00010020  003a0043 0044005c 0063006f 006d0075
00000000`00010030  006e0065 00730074 00610020 0064006e
00000000`00010040  00530020 00740065 00690074 0067006e
00000000`00010050  005c0073 006c0041 0020006c 00730055
00000000`00010060  00720065 002e0073 00320057 0033004b
00000000`00010070  00410000 00500050 00410044 00410054

kd> f 10000 10000+1000 0
Filled 0x1000 bytes
```

```
kd> dd 10000
00000000`00010000  00000000 00000000 00000000 00000000
00000000`00010010  00000000 00000000 00000000 00000000
00000000`00010020  00000000 00000000 00000000 00000000
00000000`00010030  00000000 00000000 00000000 00000000
00000000`00010040  00000000 00000000 00000000 00000000
00000000`00010050  00000000 00000000 00000000 00000000
00000000`00010060  00000000 00000000 00000000 00000000
00000000`00010070  00000000 00000000 00000000 00000000
```

Now we can save the modified complete dump file:

```
kd> .dump /f c:\Dumps\SecuredDump.dmp
```

If we want to find and erase read-write pages, for example, we can use **!vad** WinDbg command to get the description of virtual address ranges:

```
kd> !process
PROCESS fffffadfe67905a0
    SessionId: 0  Cid: 085c    Peb: 7fffffd4000  ParentCid: 0acc
    DirBase: 232e2000  ObjectTable: fffffa8000917e10  HandleCount:  55.
    Image: SystemDump.exe
    VadRoot fffffadfe6f293e0 Vads 65 Clone 0 Private 388. Modified 84.
Locked 0.
    DeviceMap fffffa80020777c0
    Token                             fffffa80008e5b50
    ElapsedTime                       00:00:06.265
    UserTime                          00:00:00.031
    KernelTime                        00:00:00.062
    QuotaPoolUsage[PagedPool]         113464
    QuotaPoolUsage[NonPagedPool]      5152
    Working Set Sizes (now,min,max)  (1429, 50, 345) (5716KB, 200KB,
1380KB)
    PeakWorkingSetSize                1429
    VirtualSize                       61 Mb
    PeakVirtualSize                   63 Mb
    PageFaultCount                    1555
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      471
```

```
kd> !vad fffffadfe6f293e0
VAD             level      start      end    commit
fffffadfe682bdf0 ( 6)         10       10         1 Private      READWRITE
fffffadfe73a0e10 ( 5)         20       20         1 Private      READWRITE
fffffadfe73a0dd0 ( 4)         30      12f         8 Private      READWRITE
fffffadfe71a4770 ( 5)        130      134         0 Mapped       READONLY
fffffadfe781bbe0 ( 3)        140      141         0 Mapped       READONLY
...
fffffadfe772d630 (-2)   7fffffdc 7fffffdd         2 Private      READWRITE
fffffadfe788e180 (-1)   7fffffde 7fffffdf         2 Private      READWRITE

Total VADs:    65  average level: 66076419  maximum depth: -1
```

In the output START and END columns refer to virtual page numbers (VPN). To get an address we need to multiply by 0×1000, for example, 7fffffde**000**.

Filling memory with zeroes to hide data with subsequent saving of a modified crash dump is applicable to user dumps too. Please also check for additional security-related flags in **.dump** command (**WinDbg is Privacy-Aware**, Volume 1, page 600).

Another application for data hiding and modification could be customized crash dumps for digital forensics exercises and contests.

HARDENING DUMP SECURITY: BEWARE OF PEB DATA

In full user dumps PEB is included with sensitive data despite stack and page cleaning and removing module paths (Volume 1, page 600). Module paths are removed indeed from **lmv** command output but _PEB.Ldr lists contain full module path information:

```
0:000> dt _PEB Ldr
ntdll!_PEB
   +0x018 Ldr : Ptr64 _PEB_LDR_DATA

0:000> dt _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr64 Void
   +0×010 InLoadOrderModuleList : _LIST_ENTRY
   +0×020 InMemoryOrderModuleList : _LIST_ENTRY
   +0×030 InInitializationOrderModuleList : _LIST_ENTRY
   +0×040 EntryInProgress  : Ptr64 Void

0:000> !peb
PEB at 000007fffffdb000
...
```

We can see this in the dump sample saved with /r and /R options:

ftp://dumpanalysis.org/pub/LargeHeapAllocations.zip

The only options currently available are:

- Custom minidumps: do not save process dumps containing full user space with /ma or /mf option for **.dump** command
- Include PEB but erase specific sections and regions pointed to like environment blocks. See page 397.
- Erase specific ASCII or UNICODE fragments manually using any binary editor. This was done for the dump file above.
- Do not send dumps but logs. See 38.

Anyway manual inspection of a dump saved with security options is recommended before sending it.

PART 10: THE ORIGIN OF CRASH DUMPS

MEMORY DUMPS FROM XEN-VIRTUALIZED WINDOWS

Suppose we got a kernel, complete or minidump from Windows running under Xen hypervisor. How would we distinguish it from a memory dump of Windows running on non-virtualized hardware? We can check machine id:

```
kd> !sysinfo machineid
Machine ID Information [From Smbios 2.4, DMIVersion 36, Size=348]
BiosMajorRelease = 3
BiosMinorRelease = 1
BiosVendor = Xen
BiosVersion = 3.1.0
SystemManufacturer = Xen
SystemProductName = HVM domU
SystemVersion = 3.1.0
```

and drivers:

```
kd> lm m *xen*
start end module name
f6012000 f605f000 dump_xenvbd (deferred)
f794b000 f795c000 xennet (deferred)
f82c0000 f830d000 xenvbd (deferred)
f845f000 f846b000 XENUTIL (deferred)
f84cf000 f84db000 dump_XENUTIL (deferred)
```

Note: similar information can be checked for VMWare and Virtual PC (Volume 1, page 219).

BUGCHECKS: SYSTEM_SERVICE_EXCEPTION

Bugcheck 0×3B is forced on x64 Windows platforms when an exception happens during a system service and unwind leads to a transition from a kernel to a user mode. Let's see this in a complete memory dump:

```
SYSTEM_SERVICE_EXCEPTION (3b)
An exception happened while executing a system service routine.
Arguments:
Arg1: 00000000c0000005, Exception code that caused the bugcheck
Arg2: fffff80001048a1d, Address of the exception record for the exception
that caused the bugcheck
Arg3: fffffade643f6870, Address of the context record for the exception
that caused the bugcheck
Arg4: 0000000000000000, zero.

CONTEXT: fffffade643f6870 -- (.cxr 0xfffffade643f6870)
rax=005300450053005c rbx=0000000000000048 rcx=0000000000000020
rdx=fffffa8007c9da20 rsi=0000000000000048 rdi=fffffade643f71d0
rip=fffff80001048a1d rsp=fffffade643f7088 rbp=0000000000000000
 r8=0000000000000048 r9=0000000000000002 r10=00490046002d0054
r11=0000000000000000 r12=fffffadf19744010 r13=fffffade643f7a78
r14=0000000000000800 r15=fffffadf1da71ee8
iopl=0 nv up ei pl nz na pe nc
cs=0010 ss=0018 ds=002b es=002b fs=0053 gs=002b efl=00010202
nt!memmove+0xbd:
fffff800`01048a1d 488941e0 mov qword ptr [rcx-20h],rax
ds:002b:00000000`00000000=????????????????
```

```
0: kd> kL
Child-SP RetAddr Call Site
fffffade`643f5eb8 fffff800`0104e834 nt!KeBugCheckEx
fffffade`643f5ec0 fffff800`0104e2fb nt!KiBugCheckDispatch+0x74
fffffade`643f6040 fffff800`0105c09d nt!KiSystemServiceHandler+0x7b
fffffade`643f6080 fffff800`01031561 nt!RtlpExecuteHandlerForException+0xd
fffffade`643f60b0 fffff800`010174fa nt!RtlDispatchException+0x2c0
fffffade`643f6770 fffff800`0104e92f nt!KiDispatchException+0xd9
fffffade`643f6d70 fffff800`0104d7e1 nt!KiExceptionExit
fffffade`643f6ef0 fffff800`01048a1d nt!KiPageFault+0x1e1
fffffade`643f7088 fffff800`01025977 nt!memmove+0xbd
fffffade`643f7090 fffffadf`101f858d nt!RtlAppendUnicodeStringToString+0x67
fffffade`643f70c0 fffffadf`101f8a1d driver+0x558d
fffffade`643f7a20 fffff800`012c3b21 driver+0x5a1d
fffffade`643f7a70 fffff800`012c3bd6 nt!IopXxxControlFile+0xa6b
fffffade`643f7b90 fffff800`0104e5fd nt!NtDeviceIoControlFile+0x56
fffffade`643f7c00 00000000`77ef12ca nt!KiSystemServiceCopyEnd+0×3
00000000`00e6ba08 00000000`77d67963 ntdll!ZwDeviceIoControlFile+0xa
00000000`00e6ba10 00000000`6340239f kernel32!DeviceIoControl+0×237
00000000`00e6bbf0 00000000`0000000e application!DllUnregisterServer+0×40f
...
```

On x64 Windows platforms KiSystemServiceCopyEnd has the similar purpose as KiFastSystemCallRet on x86 platforms (Volume 1, page 649).

We see that the chain of exception handlers spans protection boundary where KiSystemServiceCopyEnd has KiSystemServiceHandler as its exception handler:

```
0: kd> !exchain
100 stack frames, scanning for handlers...
Frame 0x03: nt!RtlpExecuteHandlerForException+0xd (fffff800`0105c09d)
  ehandler nt!RtlpExceptionHandler (fffff800`0105c060)
Frame 0x05: nt!KiDispatchException+0xd9 (fffff800`010174fa)
  ehandler nt!_C_specific_handler (fffff800`010356e0)
Frame 0x0a: driver+0x558d (fffffadf`101f858d)
  ehandler driver+0x1242 (fffffadf`101f4242)
Frame 0x0c: nt!IopXxxControlFile+0xa6b (fffff800`012c3b21)
  ehandler nt!_C_specific_handler (fffff800`010356e0)
Frame 0×0e: nt!KiSystemServiceCopyEnd+0×3 (fffff800`0104e5fd)
  ehandler nt!KiSystemServiceHandler (fffff800`0104e280)
Frame 0×10: kernel32!DeviceIoControl+0×237 (00000000`77d67963)
  ehandler kernel32!_C_specific_handler (00000000`77d92200)
```

If we disassemble KiSystemServiceHandler we get this code with bugcheck 3B branch:

```
kd> uf nt!KiSystemServiceHandler
nt!KiSystemServiceHandler:
...
fffff800`01040ddc cmp     byte ptr [rax+153h],0
fffff800`01040de3 je      nt!KiSystemServiceHandler+0×7b
(fffff800`01040dfb)

nt!KiSystemServiceHandler+0x65:
fffff800`01040de5 xor     r10,r10
fffff800`01040de8 mov     r9,r8
fffff800`01040deb mov     r8,qword ptr [rcx+10h]
fffff800`01040def mov     edx,dword ptr [rcx]
fffff800`01040df1 mov     ecx,3Bh
fffff800`01040df6 call    nt!KiBugCheckDispatch (fffff800`01041300)

nt!KiSystemServiceHandler+0x7b:
fffff800`01040dfb mov     eax,1
fffff800`01040e00 add     rsp,38h
fffff800`01040e04 ret
...
```

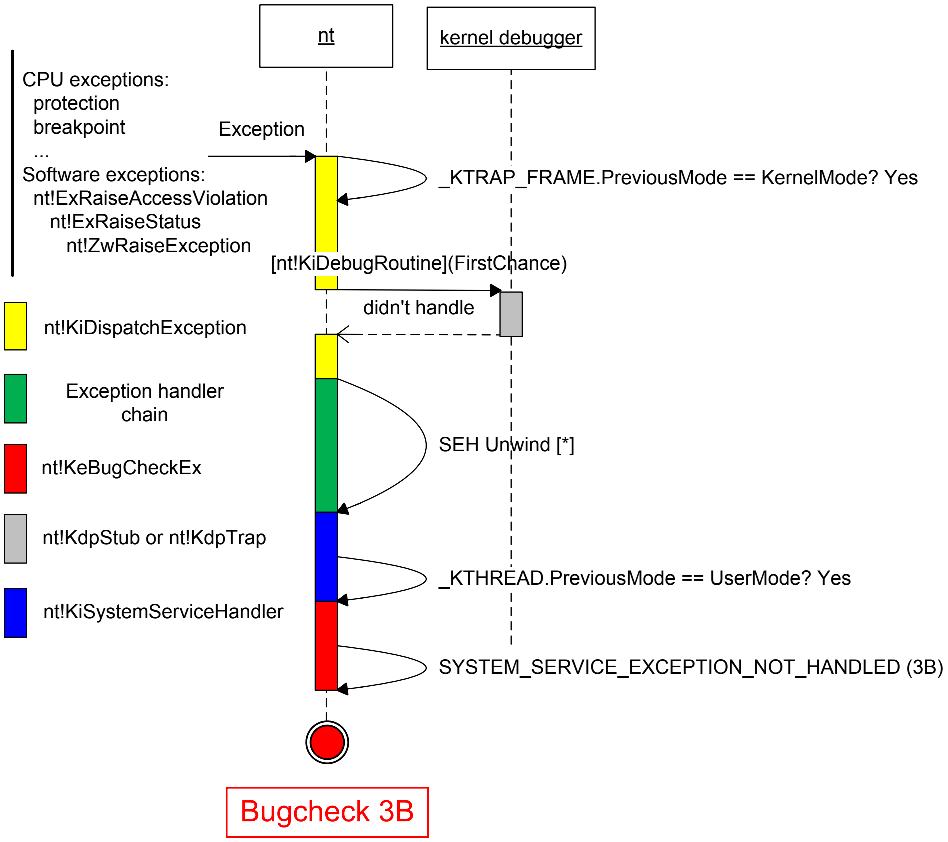
Here we see that the code checks whether the previous mode for a thread was UserMode and if this is the case it bugchecks the system because transitioning back to user space in exception unwind would have had disastrous consequences. The system wants to save a controlled crash dump for later problem analysis:

```
kd> dt _KTHREAD
ntdll!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 MutantListHead   : _LIST_ENTRY
   +0x028 InitialStack     : Ptr64 Void
   +0x030 StackLimit       : Ptr64 Void
   +0x038 KernelStack      : Ptr64 Void
...
   +0×153 PreviousMode     : Char
...
```

Note that _KTHREAD.PreviousMode should not be confused with _KTRAP_FRAME.PreviousMode. The latter has KernelMode value if an exception happened while CPU was in kernel mode but the former structure field shows the previous CPU mode of a thread, for example, it has UserMode value if a user space thread called a system service.

```
kd> dt _KTRAP_FRAME
ntdll!_KTRAP_FRAME
  +0x000 P1Home : Uint8B
  +0x008 P2Home : Uint8B
  +0x010 P3Home : Uint8B
  +0x018 P4Home : Uint8B
  +0x020 P5 : Uint8B
  +0×028 PreviousMode : Char
  +0×029 PreviousIrql : UChar
  +0×02a FaultIndicator : UChar
...
```

We can put all of this on a colored sequence UML diagram:



BUGCHECK CALLBACKS

There are some improvements in Vista and Windows Server 2008 regarding various WER callbacks to write user-defined data in the case of application crashes and hangs. However, many engineers are not aware that the similar mechanism had existed in kernel for many years:

Writing a Bug Check Callback Routine
(http://msdn.microsoft.com/en-us/library/aa489630.aspx)

We can check this data using **!bugdump** and **.enumtag** WinDbg commands:

```
0: kd> !bugdump
**** Dump of Bug Check Data ****
8526ba7c: Bug check callback record could not be read
```

We get “could not be read” message probably because for systems newer than Windows XP SP1 **!bugdump** command shows callback data written to memory AFTER the crash dump was saved. So it is useful for live debugging only. However we can see bugcheck callbacks form a linked list:

```
0: kd> dps 8526ba7c
8526ba7c  849eca7c
8526ba80  81b36ce0 nt!KeBugCheckCallbackListHead
8526ba84  858a7dea ndis!ndisBugcheckHandler
8526ba88  8526b438
8526ba8c  00000b28
8526ba90  8594dd76 ndis! ?? ::LNCPHCLB::`string’
8526ba94  90461ac0
8526ba98  00000001
8526ba9c  85936767 ndis!ndisMDispatchReceiveNetBufferLists
8526baa0  85936767 ndis!ndisMDispatchReceiveNetBufferLists
8526baa4  85969274 ndis!ethFilterDprIndicateReceivePacket
8526baa8  8de66c5c bthpan!MpReturnPacket
8526baac  8526ea80
8526bab0  859495ef ndis!ndisSynchReturnPacketsForTranslation
8526bab4  8526b438
8526bab8  00000000

0: kd> !list -x "dps @$extret l10" 81b36ce0
81b36ce0  8526ba7c
81b36ce4  81ddbe40 hal!HalpCallbackRecord
81b36ce8  00000000
81b36cec  00000001
81b36cf0  00000000
81b36cf4  00000000
81b36cf8  00000101
```

```
81b36cfc  00000001
81b36d00  00000000
81b36d04  00000000
81b36d08  00000000
81b36d0c  00000000
81b36d10  00000000
81b36d14  00000000
81b36d18  00000000
81b36d1c  00000000

8526ba7c  849eca7c
8526ba80  81b36ce0 nt!KeBugCheckCallbackListHead
8526ba84  858a7dea ndis!ndisBugcheckHandler
8526ba88  8526b438
8526ba8c  00000b28
8526ba90  8594dd76 ndis! ?? ::LNCPHCLB::`string'
8526ba94  90461ac0
8526ba98  00000001
8526ba9c  85936767 ndis!ndisMDispatchReceiveNetBufferLists
8526baa0  85936767 ndis!ndisMDispatchReceiveNetBufferLists
8526baa4  85969274 ndis!ethFilterDprIndicateReceivePacket
8526baa8  8de66c5c bthpan!MpReturnPacket
8526baac  8526ea80
8526bab0  859495ef ndis!ndisSynchReturnPacketsForTranslation
8526bab4  8526b438
8526bab8  00000000

849eca7c  849ea72c
849eca80  8526ba7c
849eca84  858a7dea ndis!ndisBugcheckHandler
849eca88  849ec438
849eca8c  00000b28
849eca90  8594dd76 ndis! ?? ::LNCPHCLB::`string'
849eca94  8fbe2ac0
849eca98  00000001
849eca9c  85936767 ndis!ndisMDispatchReceiveNetBufferLists
849ecaa0  85936767 ndis!ndisMDispatchReceiveNetBufferLists
849ecaa4  859432ca ndis!ndisMIndicatePacket
849ecaa8  00000000
849ecaac  00000000
849ecab0  859495ef ndis!ndisSynchReturnPacketsForTranslation
849ecab4  849ec438
849ecab8  00000000

849ea72c  849c272c
849ea730  849eca7c
849ea734  858a7dea ndis!ndisBugcheckHandler
849ea738  849ea0e8
849ea73c  00000b28
849ea740  8594dd76 ndis! ?? ::LNCPHCLB::`string'
849ea744  8fbe0770
849ea748  00000001
849ea74c  85936767 ndis!ndisMDispatchReceiveNetBufferLists
849ea750  85936767 ndis!ndisMDispatchReceiveNetBufferLists
```

```
849ea754  85969274 ndis!ethFilterDprIndicateReceivePacket
849ea758  00000000
849ea75c  00000000
849ea760  859495ef ndis!ndisSynchReturnPacketsForTranslation
849ea764  849ea0e8
849ea768  00000000

849c272c  849c172c
849c2730  849ea72c
849c2734  858a7dea ndis!ndisBugcheckHandler
849c2738  849c20e8
849c273c  00000b28
849c2740  8594dd76 ndis! ?? ::LNCPHCLB::`string'
849c2744  8fbb8770
849c2748  00000001
849c274c  85936767 ndis!ndisMDispatchReceiveNetBufferLists
849c2750  85936767 ndis!ndisMDispatchReceiveNetBufferLists
849c2754  85969274 ndis!ethFilterDprIndicateReceivePacket
849c2758  85df579a tunmp!TunMpReturnPacket
849c275c  84a45538
849c2760  859495ef ndis!ndisSynchReturnPacketsForTranslation
849c2764  849c20e8
849c2768  00000000

849c172c  849a072c
849c1730  849c272c
849c1734  858a7dea ndis!ndisBugcheckHandler
849c1738  849c10e8
849c173c  00000b28
849c1740  8594dd76 ndis! ?? ::LNCPHCLB::`string'
849c1744  8fbb7770
849c1748  00000001
849c174c  85936767 ndis!ndisMDispatchReceiveNetBufferLists
849c1750  85936767 ndis!ndisMDispatchReceiveNetBufferLists
849c1754  859432ca ndis!ndisMIndicatePacket
849c1758  00000000
849c175c  00000000
849c1760  859495ef ndis!ndisSynchReturnPacketsForTranslation
849c1764  849c10e8
849c1768  00000000

849a072c  8499d72c
849a0730  849c172c
849a0734  858a7dea ndis!ndisBugcheckHandler
849a0738  849a00e8
849a073c  00000b28
849a0740  8594dd76 ndis! ?? ::LNCPHCLB::`string'
849a0744  8fb96770
849a0748  00000001
849a074c  85936767 ndis!ndisMDispatchReceiveNetBufferLists
849a0750  85936767 ndis!ndisMDispatchReceiveNetBufferLists
849a0754  859432ca ndis!ndisMIndicatePacket
849a0758  00000000
849a075c  00000000
```

```
849a0760  859495ef ndis!ndisSynchReturnPacketsForTranslation
849a0764  849a00e8
849a0768  00000000

8499d72c  8499f72c
8499d730  849a072c
8499d734  858a7dea ndis!ndisBugcheckHandler
8499d738  8499d0e8
8499d73c  00000b28
8499d740  8594dd76 ndis! ?? ::LNCPHCLB::`string'
8499d744  8fb93770
8499d748  00000001
8499d74c  85936767 ndis!ndisMDispatchReceiveNetBufferLists
8499d750  85936767 ndis!ndisMDispatchReceiveNetBufferLists
8499d754  859432ca ndis!ndisMIndicatePacket
8499d758  00000000
8499d75c  00000000
8499d760  859495ef ndis!ndisSynchReturnPacketsForTranslation
8499d764  8499d0e8
8499d768  00000000

8499f72c  81ddbe40 hal!HalpCallbackRecord
8499f730  8499d72c
8499f734  858a7dea ndis!ndisBugcheckHandler
8499f738  8499f0e8
8499f73c  00000b28
8499f740  8594dd76 ndis! ?? ::LNCPHCLB::`string'
8499f744  8fb95770
8499f748  00000001
8499f74c  85936767 ndis!ndisMDispatchReceiveNetBufferLists
8499f750  85936767 ndis!ndisMDispatchReceiveNetBufferLists
8499f754  859432ca ndis!ndisMIndicatePacket
8499f758  00000000
8499f75c  00000000
8499f760  859495ef ndis!ndisSynchReturnPacketsForTranslation
8499f764  8499f0e8
8499f768  00000000

81ddbe40  81b36ce0 nt!KeBugCheckCallbackListHead
81ddbe44  8499f72c
81ddbe48  81dcebdc hal!HalpBugCheckCallback
81ddbe4c  00000000
81ddbe50  00000000
81ddbe54  81dc2550 hal!HalName
81ddbe58  03b9112c
81ddbe5c  00000001
81ddbe60  00000000
81ddbe64  00000000
81ddbe68  00000000
81ddbe70  6d46da80
```

Another WinDbg command **.enumtag** shows data written before saving a crash dump and therefore useful for postmortem crash dump analysis (binary output is removed for visual clarity):

```
0: kd> .enumtag
{BC5C008F-1E3A-44D7-988D86F6884C6758} - 0x5cd bytes
...
  Apple Inc..    M
  M21.88Z.009A.B00
  .0706281359.06/2
  8/07............
  ................
  .Apple Inc..Macm
  ini2,1.1.0.
        .System SK
  UNumber.Napa Mac
  ................
  ..Apple Inc..Mac
  -F4208EAA.PVT. .
  .Part Compon
  ent.............
  ..........Apple
  Inc..Mac-F4208EA
  A.           .
  ............J6H1
  :1-X CMOS CLEAR(
  default); J8H1:1
  -X BIOS RECOVERY
  ...........None.
  Ethernet........
  ...None.DVI.....
  ......None.USB0.
  ..........None.U
  SB1...........No
  ne.USB2.........
  ..None.USB3.....
  ....!.None.FireW
  ire0...........N
  one.Audio Line I
  n...........None
  .Audio Line Out.
  ..............Ai
  rPort........Int
  egrated Graphics
  Controller ....
  ....Yukon Ethern
  et Controller...
  .....Azalia Audi
  o Codec........S
  ATA........PATA.
  .A..........Inte
  l(R) Core(TM)2 C
  PU         T.Int
  el(R) Corporatio
  n.U2E1.       ..
...
  .......Intel(R)
  Core(TM)2 CPU
       T.Intel(R)
  Corporation.U2E
  1.       .......
...
  ...........DIMM0
  .BANK 0.0x2C0000
  0000000000.
      .       .0x
  3848544636343634
  4844592D36363744
  3320....!.......
  .. .$........"..
  ...@.@..........
  ......DIMM1.BANK
  1.0x2C000000000
  00000.
  .       .0x38485
  4463634363448445
  92D363637443320.
...
{6C7AC389-4313-47DC-9F34A8800A0FB56C} - 0x266 bytes
  ....~.M.H.z.....
  ......)...,...C.
  o.m.p.o.n.e.n.t.
  .I.n.f.o.r.m.a.
  t.i.o.n.........
  ..&...C.o.n.f.i.
  g.u.r.a.t.i.o.n.
  .D.a.t.a.......
  .......I.d.e.n.
  t.i.f.i.e.r.....
  ..B...x.8.6. .F.
  a.m.i.l.y. .6. .
  M.o.d.e.l. .1.5.
  .S.t.e.p.p.i.n.
  g. .2...(...P.r.
  o.c.e.s.s.o.r.N.
  a.m.e.S.t.r.i.n.
  g.......`...I.n.
  t.e.l.(.R.). .C.
  o.r.e.(.T.M.).2.
  .C.P.U. . . . .
  . . . . .T.5.6.
  0.0. . .@. .1...
  8.3.G.H.z..."...
  U.p.d.a.t.e. .S.
  i.g.n.a.t.u.r.e.
  .............W.
  ......U.p.d.a.t.
  e. .S.t.a.t.u.s.
  ..............".
  ..V.e.n.d.o.r.I.
  d.e.n.t.i.f.i.e.
  r...........G.e.
  n.u.i.n.e.I.n.t.
  e.l.......M.S.R.
...
{D03DC06F-D88E-44C5-BA2AFAE035172D19} - 0x438 bytes
  ............Genu
  ntelineI....Genu
  ntelineI........
...
  ........Intel(R)
  Core(TMIntel(R)
  Core(TM........
  )2 CPU         T
  )2 CPU         T
  ........5600  @
  1.83GHz.5600  @
  1.83GHz.........
...
{E83B40D2-B0A0-4842-ABEA71C9E3463DD1} - 0x184 bytes
  APICh.....APPLE
  Apple00.....Loki
  _.......FACP....
  .aAPPLE Apple00.
  ....Loki_......>
  HPET8.....APPLE
  Apple00.....Loki
  _.......MCFG<...
  ..APPLE Apple00.
  ....Loki_.......
  ASF!.... .APPLE
  Apple00.....Loki
  _.......SBST0...
  ..APPLE Apple00.
  ....Loki_.......
  ECDTS....9APPLE
  Apple00.....Loki
  _.......SSDTO...
  .>APPLE SataPri.
  ....INTL... SSDT
  O....>APPLE Sata
  Pri.....INTL...
  SSDTO....>APPLE
  SataPri.....INTL
{270A33FD-3DA6-460D-BA893C1BAE21E39B} - 0xfc8 bytes
...
```

Of course, this is much more useful if our drivers save additional data for troubleshooting and we have written a WinDbg extension to interpret it.

APPLICATION VERIFIER ON X64 PLATFORMS

A small note from the field. Sometimes on x64 Windows platform we set a default postmortem debugger or configure WER and then install Microsoft Application Verifier to do some checks. However no crash dump files are saved and this might be because we installed and configured amd64 bit version of Application Verifier but the problem application was 32-bit. For this application we need to install and configure x86 version of Application Verifier.

WHO SAVED THE DUMP FILE?

Sometimes the question arises about which postmortem debugger saved a crash dump resulted from an unhandled exception. For example, for pre-Vista systems the customer may believe that they used NTSD and but we know that properly configured NTSD as a default debugger (http://support.citrix.com/article/ctx105888) never saves mini-dumps. However the WinDbg shows this:

```
Loading Dump File [application.mdmp]
User Mini Dump File: Only registers, stack and portions of memory are
available
```

We know that the default unhandled exception filter launches a default postmortem debugger (Volume 1, page 113). Because CreateProcess call needs a path and it is taken from AeDebug registry key the value is stored on a stack. So it is easy to dump the stack data, find UNICODE pattern and dump the string. This can be done using raw stack data or from the full exception processing stack trace where unhandled exception filter is present:

```
STACK_TEXT:
0dadc884 7c827cfb ntdll!KiFastSystemCallRet
0dadc888 77e76792 ntdll!NtWaitForMultipleObjects+0xc
0dadcb78 77e792a3 kernel32!UnhandledExceptionFilter+0×7c0
0dadcb80 77e61ac1 kernel32!BaseThreadStart+0×4a
0dadcba8 7c828752 kernel32!_except_handler3+0×61
0dadcbcc 7c828723 ntdll!ExecuteHandler2+0×26
0dadcc74 7c82855e ntdll!ExecuteHandler+0×24
0dadcc74 7c35042b ntdll!KiUserExceptionDispatcher+0xe
0dadcf70 0964a32a msvcr71!wcscpy+0xb
...
```

```
0:086> dds 0dadc884
0dadc884  7c828270 ntdll!_except_handler3
0dadc888  7c827cfb ntdll!NtWaitForMultipleObjects+0xc
0dadc88c  77e76792 kernel32!UnhandledExceptionFilter+0×7c0
0dadc890  00000002
0dadc894  0dadc9e8
0dadc898  00000001
0dadc89c  00000001
0dadc8a0  00000000
0dadc8a4  003a0043
0dadc8a8  0057005c
0dadc8ac  004e0049
0dadc8b0  004f0044
0dadc8b4  00530057
0dadc8b8  0073005c
0dadc8bc  00730079
0dadc8c0  00650074
0dadc8c4  0033006d
0dadc8c8  005c0032
0dadc8cc  00720064
0dadc8d0  00740077
0dadc8d4  006e0073
0dadc8d8  00320033
0dadc8dc  002d0020
0dadc8e0  00200070
0dadc8e4  00390032
0dadc8e8  00320035
0dadc8ec  002d0020
0dadc8f0  00200065
0dadc8f4  00300031
0dadc8f8  00380038
0dadc8fc  002d0020
0dadc900  00000067

0:086> du 0dadc8a4
0dadc8a4  "C:\WINDOWS\system32\drwtsn32 -p ”
0dadc8e4  “2952 -e 1088 -g”
```

ADPLUS IN 21 SECONDS AND 13 STEPS

When dealing with a problem where NTSD fails to save a dump file because of improper configuration for a default postmortem debugger or for other reasons we can advise to use ADPlus from Debugging Tools for Windows in crash mode.

Here is the quick polished tutorial:

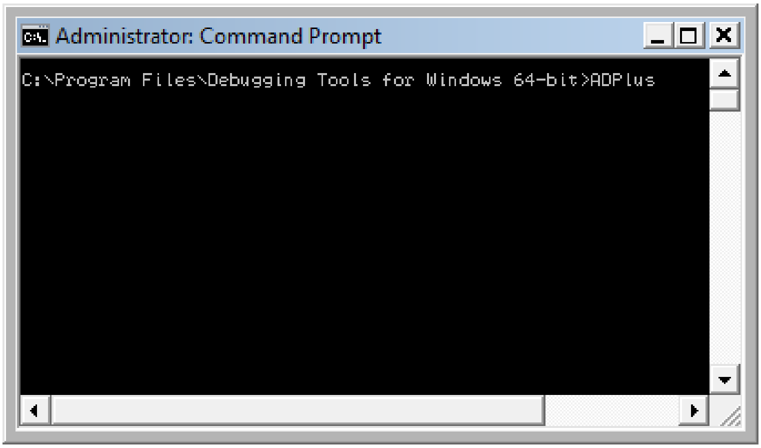
1. Download and install Debugging Tools for Windows appropriate for your application or service platform. For example, if your service is 32-bit but runs on x64 you need to download 32-bit package. Refer to windbg.org for quick download links.

2. Get ready for the test and download TestDefaultDebugger package from

http://support.citrix.com/article/ctx111901

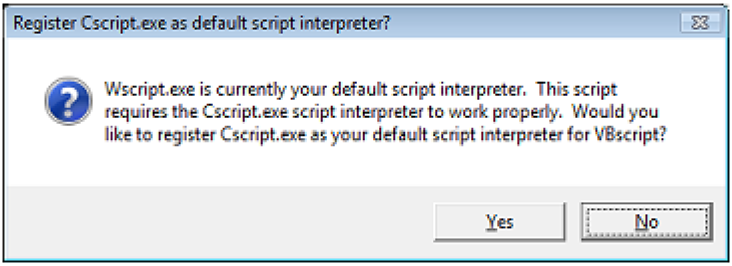
3. Open a command prompt elevated as Administrator and CD to Debugging Tools for Windows installation folder.

4. Run ADPlus command:



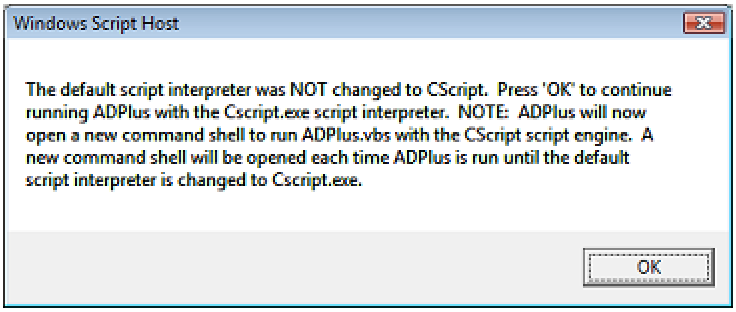
```
C:\Program Files\Debugging Tools for Windows 64-bit>ADPlus
```

5. Skip any warnings related to script interpreter if any:



Register Cscript.exe as default script interpreter?

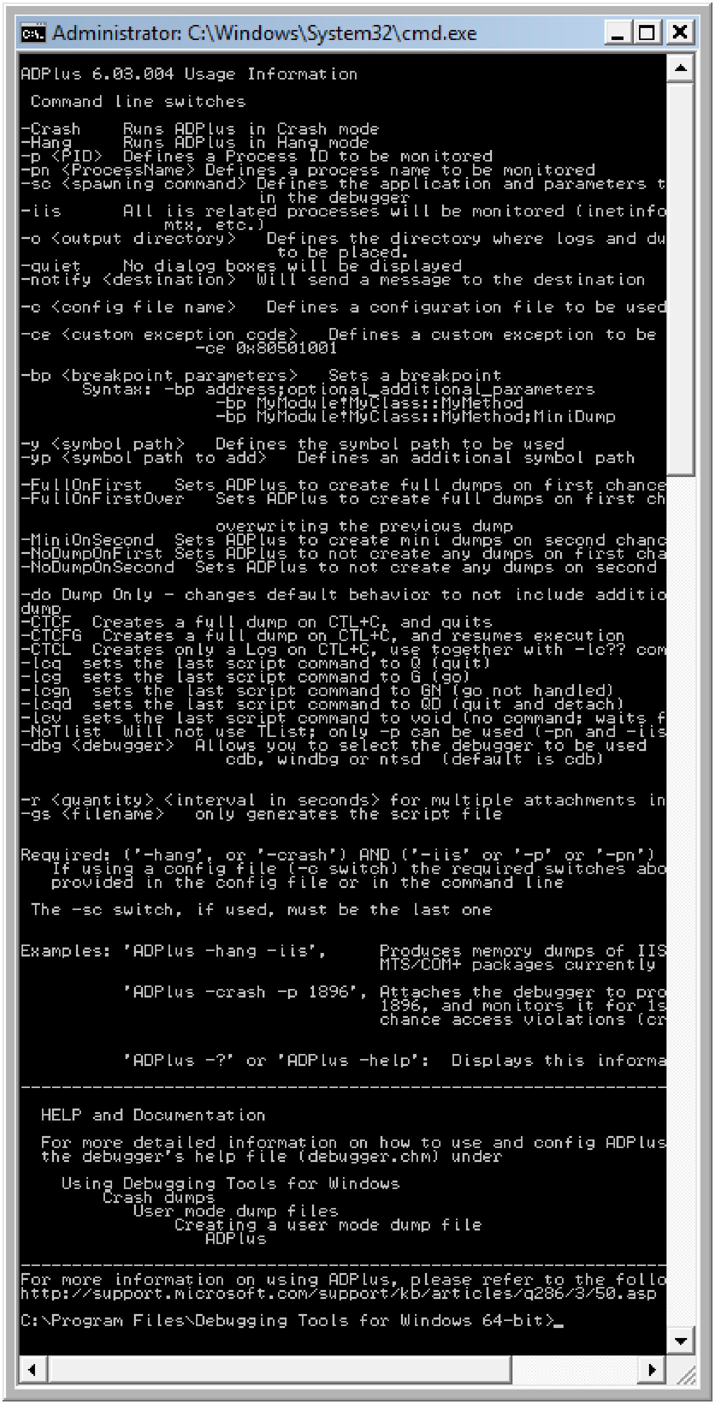
Wscript.exe is currently your default script interpreter. This script requires the Cscript.exe script interpreter to work properly. Would you like to register Cscript.exe as your default script interpreter for VBscript?



Windows Script Host

The default script interpreter was NOT changed to CScript. Press 'OK' to continue running ADPlus with the Cscript.exe script interpreter. NOTE: ADPlus will now open a new command shell to run ADPlus.vbs with the CScript script engine. A new command shell will be opened each time ADPlus is run until the default script interpreter is changed to Cscript.exe.

6. Another command line window appears with ADPlus switches:

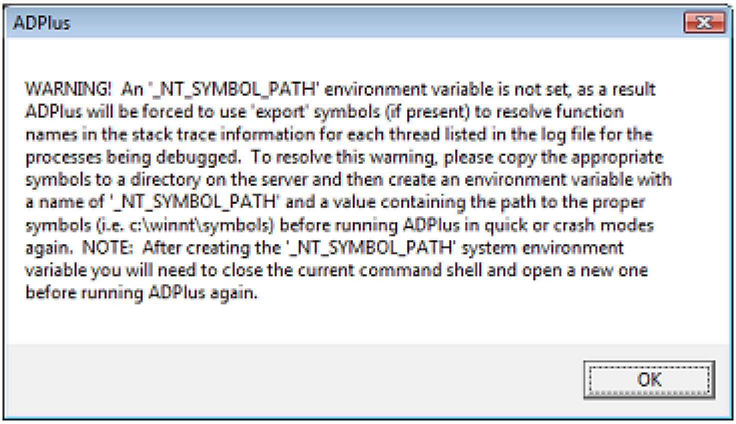


```
ADPlus 6.03.004 Usage Information

 Command line switches

-Crash    Runs ADPlus in Crash mode
-Hang     Runs ADPlus in Hang mode
-p <PID>  Defines a Process ID to be monitored
-pn <ProcessName> Defines a process name to be monitored
-sc <spawning command> Defines the application and parameters t
                       in the debugger
-iis      All iis related processes will be monitored (inetinfo
              mtx, etc.)
-o <output directory>   Defines the directory where logs and du
                         to be placed.
-quiet    No dialog boxes will be displayed
-notify <destination>  Will send a message to the destination

-c <config file name>   Defines a configuration file to be used

-ce <custom exception code>   Defines a custom exception to be
                 -ce 0x80501001

-bp <breakpoint parameters>   Sets a breakpoint
      Syntax: -bp address;optional_additional_parameters
                   -bp MyModule!MyClass::MyMethod
                   -bp MyModule!MyClass::MyMethod;MiniDump

-y <symbol path>   Defines the symbol path to be used
-yp <symbol path to add>   Defines an additional symbol path

-FullOnFirst   Sets ADPlus to create full dumps on first chance
-FullOnFirstOver   Sets ADPlus to create full dumps on first ch

                   overwriting the previous dump
-MiniOnSecond  Sets ADPlus to create mini dumps on second chanc
-NoDumpOnFirst Sets ADPlus to not create any dumps on first cha
-NoDumpOnSecond  Sets ADPlus to not create any dumps on second

-do Dump Only - changes default behavior to not include additio
dump
-CTCF  Creates a full dump on CTL+C, and quits
-CTCFG  Creates a full dump on CTL+C, and resumes execution
-CTCL  Creates only a Log on CTL+C, use together with -lc?? com
-lcq   sets the last script command to Q (quit)
-lcg   sets the last script command to G (go)
-lcgn   sets the last script command to GN (go not handled)
-lcqd   sets the last script command to QD (quit and detach)
-lcv   sets the last script command to void (no command; waits f
-NoTlist  Will not use TList; only -p can be used (-pn and -iis
-dbg <debugger>  Allows you to select the debugger to be used
                    cdb, windbg or ntsd  (default is cdb)


-r <quantity> <interval in seconds> for multiple attachments in
-gs <filename>   only generates the script file


Required: ('-hang', or '-crash') AND ('-iis' or '-p' or '-pn')
   If using a config file (-c switch) the required switches abo
   provided in the config file or in the command line

 The -sc switch, if used, must be the last one


Examples: 'ADPlus -hang -iis',     Produces memory dumps of IIS
                                   MTS/COM+ packages currently

          'ADPlus -crash -p 1896', Attaches the debugger to pro
                                   1896, and monitors it for 1s
                                   chance access violations (cr


          'ADPlus -?' or 'ADPlus -help':  Displays this informa

---------------------------------------------------------------

  HELP and Documentation

  For more detailed information on how to use and config ADPlus
  the debugger's help file (debugger.chm) under

    Using Debugging Tools for Windows
        Crash dumps
           User mode dump files
               Creating a user mode dump file
                  ADPlus

---------------------------------------------------------------
For more information on using ADPlus, please refer to the follo
http://support.microsoft.com/support/kb/articles/q286/3/50.asp

C:\Program Files\Debugging Tools for Windows 64-bit>_
```

7. Close it, go back to the first command line window we used to launch-test ADPlus and type this command:

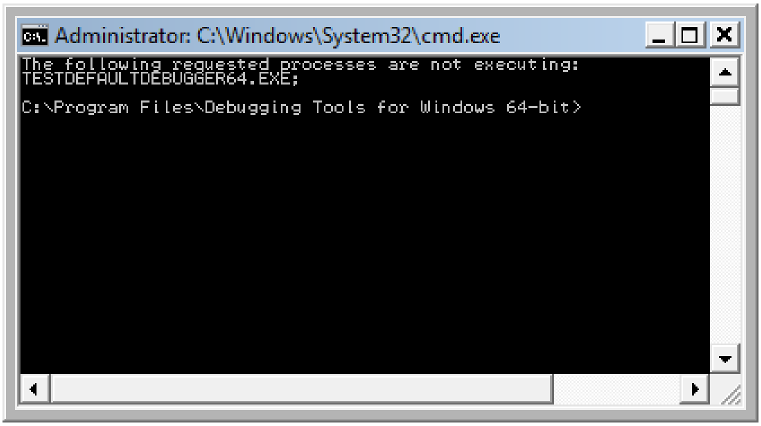
```
ADPlus -crash -pn TestDefaultDebugger64.exe
```

8. Skip warnings from step 5 if any and the symbol path warning if it appears too:



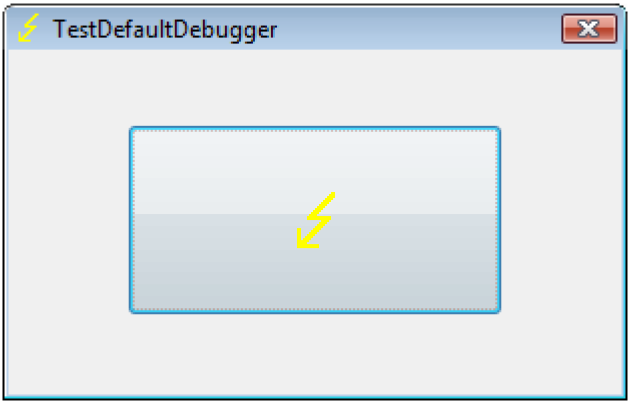
WARNING! An '_NT_SYMBOL_PATH' environment variable is not set, as a result ADPlus will be forced to use 'export' symbols (if present) to resolve function names in the stack trace information for each thread listed in the log file for the processes being debugged. To resolve this warning, please copy the appropriate symbols to a directory on the server and then create an environment variable with a name of '_NT_SYMBOL_PATH' and a value containing the path to the proper symbols (i.e. c:\winnt\symbols) before running ADPlus in quick or crash modes again. NOTE: After creating the '_NT_SYMBOL_PATH' system environment variable you will need to close the current command shell and open a new one before running ADPlus again.

9. The second command line window is opened with the following expected output because we didn't run TestDefaultDebugger64.exe:

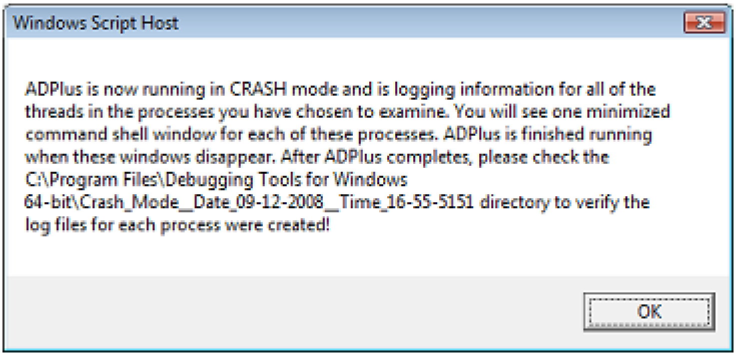


```
The following requested processes are not executing:
TESTDEFAULTDEBUGGER64.EXE;

C:\Program Files\Debugging Tools for Windows 64-bit>
```

10. Close it and launch TestDefaultDebugger64.exe from the package downloaded in step 2:

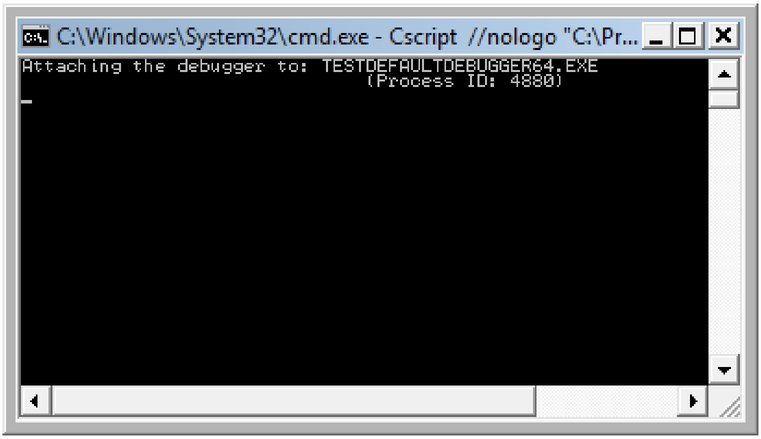


11. Go back to the first command line window and repeat the command from step 7. You can also specify PID by `-p <PID>` instead of `-pn <Name>` option. Skip warnings from steps 5 and 8 if any and you would see the following message showing where ADPlus will store logs and memory dumps if any:



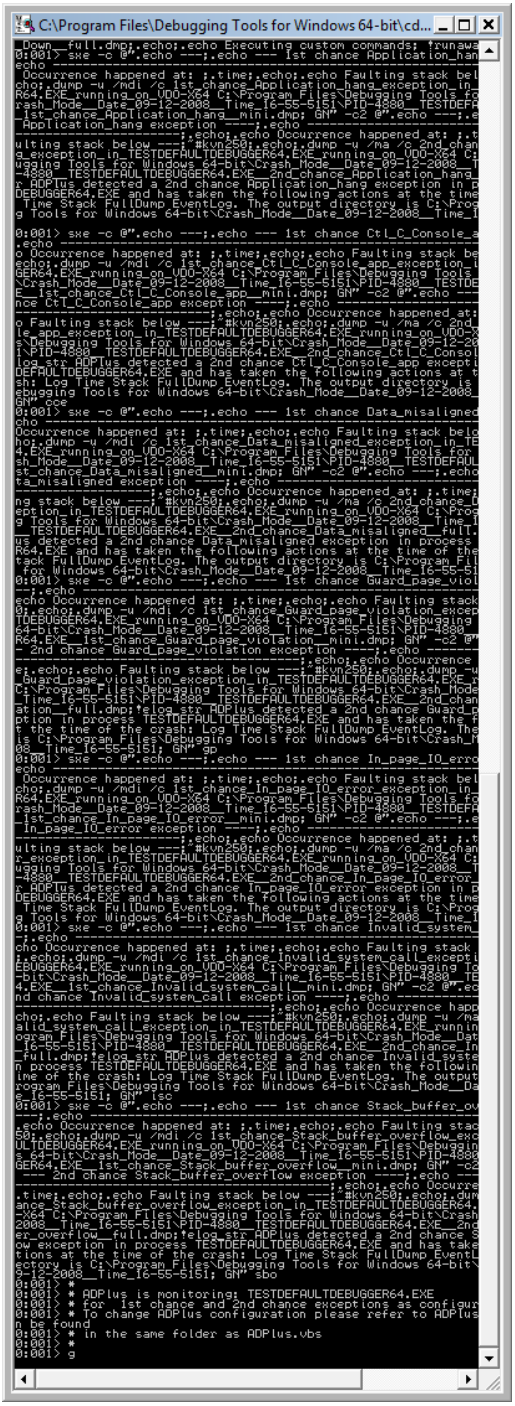
ADPlus is now running in CRASH mode and is logging information for all of the threads in the processes you have chosen to examine. You will see one minimized command shell window for each of these processes. ADPlus is finished running when these windows disappear. After ADPlus completes, please check the C:\Program Files\Debugging Tools for Windows 64-bit\Crash_Mode__Date_09-12-2008__Time_16-55-5151 directory to verify the log files for each process were created!

There are also 2 additional command line windows appear. One is showing which PID the debugger was attached to:



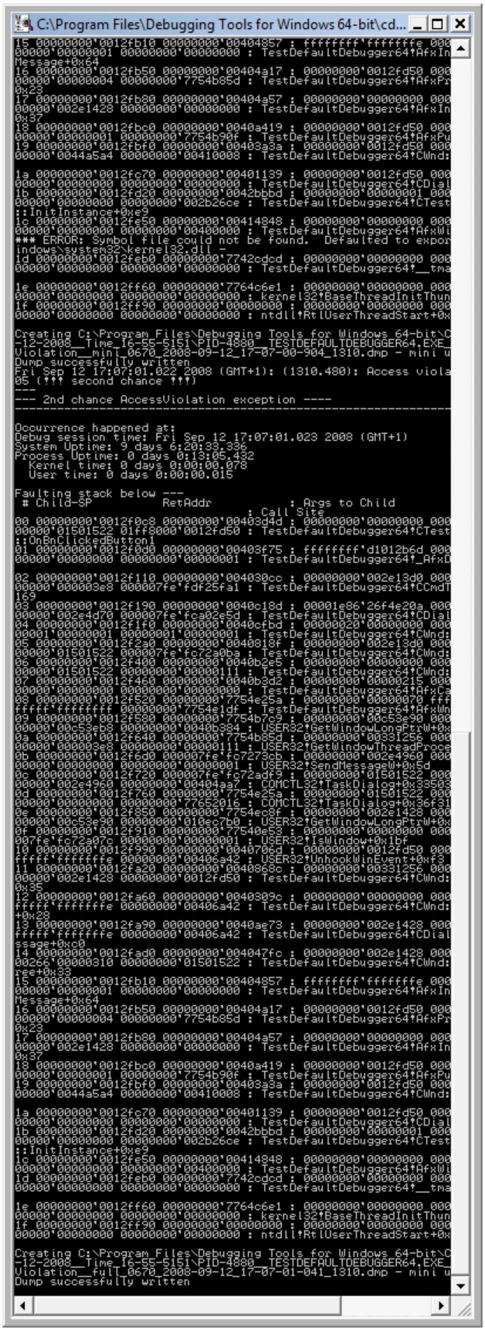
```
Attaching the debugger to: TESTDEFAULTDEBUGGER64.EXE
                           (Process ID: 4880)
_
```

and the other showing the output of attached console debugger, CDB by default:



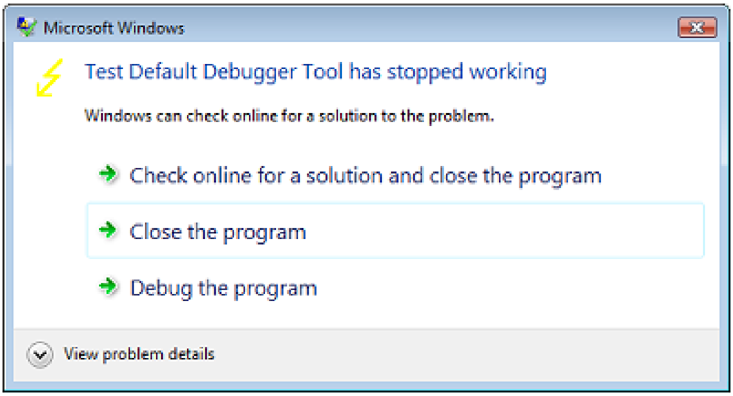
```
_Down__full.dmp;.echo;.echo Executing custom commands; !runawa
0:001> sxe -c @".echo ---;.echo --- 1st chance Application_han
echo ----------------------------------------------------------
 Occurrence happened at: ;.time;.echo;.echo Faulting stack bel
cho;.dump -u /mdi /c 1st_chance_Application_hang_exception_in_
R64.EXE_running_on_VDO-X64 C:\Program Files\Debugging Tools fo
rash_Mode__Date_09-12-2008__Time_16-55-5151\PID-4880__TESTDEFA
_1st_chance_Application_hang__mini.dmp; GN" -c2 @".echo ---;.e
 Application_hang exception ----;.echo -----------------------
-----------------------;.echo;.echo Occurrence happened at: ;.t
ulting stack below ---;~#kvn250;.echo;.dump -u /ma /c 2nd_chan
g_exception_in_TESTDEFAULTDEBUGGER64.EXE_running_on_VDO-X64 C:
ugging Tools for Windows 64-bit\Crash_Mode__Date_09-12-2008__T
-4880__TESTDEFAULTDEBUGGER64.EXE__2nd_chance_Application_hang_
r ADPlus detected a 2nd chance Application_hang exception in p
DEBUGGER64.EXE and has taken the following actions at the time
 Time Stack FullDump EventLog. The output directory is C:\Prog
g Tools for Windows 64-bit\Crash_Mode__Date_09-12-2008__Time_1

0:001> sxe -c @".echo ---;.echo --- 1st chance Ctl_C_Console_a
.echo ---------------------------------------------------------
o Occurrence happened at: ;.time;.echo;.echo Faulting stack be
echo;.dump -u /mdi /c 1st_chance_Ctl_C_Console_app_exception_i
GER64.EXE_running_on_VDO-X64 C:\Program Files\Debugging Tools
\Crash_Mode__Date_09-12-2008__Time_16-55-5151\PID-4880__TESTDE
E__1st_chance_Ctl_C_Console_app__mini.dmp; GN" -c2 @".echo ---
nce Ctl_C_Console_app exception ----;.echo -------------------
---------------------------;.echo;.echo Occurrence happened at:
o Faulting stack below ---;~#kvn250;.echo;.dump -u /ma /c 2nd_
le_app_exception_in_TESTDEFAULTDEBUGGER64.EXE_running_on_VDO-X
s\Debugging Tools for Windows 64-bit\Crash_Mode__Date_09-12-20
1\PID-4880__TESTDEFAULTDEBUGGER64.EXE__2nd_chance_Ctl_C_Consol
log_str ADPlus detected a 2nd chance Ctl_C_Console_app excepti
DEFAULTDEBUGGER64.EXE and has taken the following actions at t
sh: Log Time Stack FullDump EventLog. The output directory is
ebugging Tools for Windows 64-bit\Crash_Mode__Date_09-12-2008_
GN" cce
0:001> sxe -c @".echo ---;.echo --- 1st chance Data_misaligned
cho -----------------------------------------------------------
Occurrence happened at: ;.time;.echo;.echo Faulting stack belo
ho;.dump -u /mdi /c 1st_chance_Data_misaligned_exception_in_TE
4.EXE_running_on_VDO-X64 C:\Program Files\Debugging Tools for
sh_Mode__Date_09-12-2008__Time_16-55-5151\PID-4880__TESTDEFAUL
st_chance_Data_misaligned__mini.dmp; GN" -c2 @".echo ---;.echo
ta_misaligned exception ----;.echo ---------------------------
-------------------;.echo;.echo Occurrence happened at: ;.time;
ng stack below ---;~#kvn250;.echo;.dump -u /ma /c 2nd_chance_D
eption_in_TESTDEFAULTDEBUGGER64.EXE_running_on_VDO-X64 C:\Prog
g Tools for Windows 64-bit\Crash_Mode__Date_09-12-2008__Time_1
__TESTDEFAULTDEBUGGER64.EXE__2nd_chance_Data_misaligned__full.
us detected a 2nd chance Data_misaligned exception in process
R64.EXE and has taken the following actions at the time of the
tack FullDump EventLog. The output directory is C:\Program Fil
 for Windows 64-bit\Crash_Mode__Date_09-12-2008__Time_16-55-51
0:001> sxe -c @".echo ---;.echo --- 1st chance Guard_page_viol
--;.echo ------------------------------------------------------
echo Occurrence happened at: ;.time;.echo;.echo Faulting stack
0;.echo;.dump -u /mdi /c 1st_chance_Guard_page_violation_excep
TDEBUGGER64.EXE_running_on_VDO-X64 C:\Program Files\Debugging
64-bit\Crash_Mode__Date_09-12-2008__Time_16-55-5151\PID-4880_
R64.EXE__1st_chance_Guard_page_violation__mini.dmp; GN" -c2 @"
- 2nd chance Guard_page_violation exception ----;.echo -------
-------------------------------------;.echo;.echo Occurrence
e;.echo;.echo Faulting stack below ---;~#kvn250;.echo;.dump -u
_Guard_page_violation_exception_in_TESTDEFAULTDEBUGGER64.EXE_r
C:\Program Files\Debugging Tools for Windows 64-bit\Crash_Mode
_Time_16-55-5151\PID-4880__TESTDEFAULTDEBUGGER64.EXE__2nd_chan
ation__full.dmp;!elog_str ADPlus detected a 2nd chance Guard_p
ption in process TESTDEFAULTDEBUGGER64.EXE and has taken the f
t the time of the crash: Log Time Stack FullDump EventLog. The
is C:\Program Files\Debugging Tools for Windows 64-bit\Crash_M
08__Time_16-55-5151; GN" gp
0:001> sxe -c @".echo ---;.echo --- 1st chance In_page_IO_erro
echo ----------------------------------------------------------
 Occurrence happened at: ;.time;.echo;.echo Faulting stack bel
cho;.dump -u /mdi /c 1st_chance_In_page_IO_error_exception_in_
R64.EXE_running_on_VDO-X64 C:\Program Files\Debugging Tools fo
rash_Mode__Date_09-12-2008__Time_16-55-5151\PID-4880__TESTDEFA
_1st_chance_In_page_IO_error__mini.dmp; GN" -c2 @".echo ---;.e
 In_page_IO_error exception ----;.echo -----------------------
-----------------------;.echo;.echo Occurrence happened at: ;.t
ulting stack below ---;~#kvn250;.echo;.dump -u /ma /c 2nd_chan
r_exception_in_TESTDEFAULTDEBUGGER64.EXE_running_on_VDO-X64 C:
ugging Tools for Windows 64-bit\Crash_Mode__Date_09-12-2008__T
-4880__TESTDEFAULTDEBUGGER64.EXE__2nd_chance_In_page_IO_error_
r ADPlus detected a 2nd chance In_page_IO_error exception in p
DEBUGGER64.EXE and has taken the following actions at the time
 Time Stack FullDump EventLog. The output directory is C:\Prog
g Tools for Windows 64-bit\Crash_Mode__Date_09-12-2008__Time_1
0:001> sxe -c @".echo ---;.echo --- 1st chance Invalid_system_
-;.echo -------------------------------------------------------
cho Occurrence happened at: ;.time;.echo;.echo Faulting stack
;.echo;.dump -u /mdi /c 1st_chance_Invalid_system_call_excepti
EBUGGER64.EXE_running_on_VDO-X64 C:\Program Files\Debugging To
-bit\Crash_Mode__Date_09-12-2008__Time_16-55-5151\PID-4880__TE
4.EXE__1st_chance_Invalid_system_call__mini.dmp; GN" -c2 @".ec
nd chance Invalid_system_call exception ----;.echo -----------
-----------------------------------;.echo;.echo Occurrence happ
cho;.echo Faulting stack below ---;~#kvn250;.echo;.dump -u /ma
alid_system_call_exception_in_TESTDEFAULTDEBUGGER64.EXE_runnin
ogram Files\Debugging Tools for Windows 64-bit\Crash_Mode__Dat
_16-55-5151\PID-4880__TESTDEFAULTDEBUGGER64.EXE__2nd_chance_In
_full.dmp;!elog_str ADPlus detected a 2nd chance Invalid_syste
n process TESTDEFAULTDEBUGGER64.EXE and has taken the followin
ime of the crash: Log Time Stack FullDump EventLog. The output
rogram Files\Debugging Tools for Windows 64-bit\Crash_Mode__Da
e_16-55-5151; GN" isc
0:001> sxe -c @".echo ---;.echo --- 1st chance Stack_buffer_ov
---;.echo -----------------------------------------------------
.echo Occurrence happened at: ;.time;.echo;.echo Faulting stac
50;.echo;.dump -u /mdi /c 1st_chance_Stack_buffer_overflow_exc
ULTDEBUGGER64.EXE_running_on_VDO-X64 C:\Program Files\Debuggin
s 64-bit\Crash_Mode__Date_09-12-2008__Time_16-55-5151\PID-4880
GER64.EXE__1st_chance_Stack_buffer_overflow__mini.dmp; GN" -c2
 --- 2nd chance Stack_buffer_overflow exception ----;.echo ---
------------------------------------------;.echo;.echo Occurre
.time;.echo;.echo Faulting stack below ---;~#kvn250;.echo;.dum
ance_Stack_buffer_overflow_exception_in_TESTDEFAULTDEBUGGER64.
-X64 C:\Program Files\Debugging Tools for Windows 64-bit\Crash
2008__Time_16-55-5151\PID-4880__TESTDEFAULTDEBUGGER64.EXE__2nd
er_overflow__full.dmp;!elog_str ADPlus detected a 2nd chance S
ow exception in process TESTDEFAULTDEBUGGER64.EXE and has take
tions at the time of the crash: Log Time Stack FullDump EventL
ectory is C:\Program Files\Debugging Tools for Windows 64-bit\
9-12-2008__Time_16-55-5151; GN" sbo
0:001> *
0:001> * ADPlus is monitoring: TESTDEFAULTDEBUGGER64.EXE
0:001> * for  1st chance and 2nd chance exceptions as configur
0:001> * To change ADPlus configuration please refer to ADPlus
n be found
0:001> * in the same folder as ADPlus.vbs
0:001> *
0:001> g
```

12. We can dismiss the message from the previous step and wait for the crash to occur when we push the big button on TestDefaultDebugger window from step 10. We see the crash instantaneously in debugger console window if it is still running:



```
15 00000000'0012fb10 00000000'00404857 : ffffffff'fffffffe 000
00000'00000001 00000000'00000000 : TestDefaultDebugger64!AfxIn
Message+0x64
16 00000000'0012fb50 00000000'00404a17 : 00000000'0012fd50 000
00000'00000004 00000000'7754b85d : TestDefaultDebugger64!AfxPr
0x23
17 00000000'0012fb80 00000000'00404a57 : 00000000'00000000 000
00000'002e1428 00000000'00000000 : TestDefaultDebugger64!AfxIn
0x37
18 00000000'0012fbc0 00000000'0040a419 : 00000000'0012fd50 000
00000'00000001 00000000'7754b90f : TestDefaultDebugger64!AfxPu
19 00000000'0012fbf0 00000000'00403a3a : 00000000'0012fd50 000
00000'0044a5a4 00000000'00410008 : TestDefaultDebugger64!CWnd:

1a 00000000'0012fc70 00000000'00401139 : 00000000'0012fd50 000
00000'00000000 00000000'00000000 : TestDefaultDebugger64!CDial
1b 00000000'0012fd20 00000000'0042bbbd : 00000000'00000001 000
00000'00000000 00000000'002b26ce : TestDefaultDebugger64!CTest
::InitInstance+0xe9
1c 00000000'0012fe50 00000000'00414848 : 00000000'00000000 000
00000'00000000 00000000'00400000 : TestDefaultDebugger64!AfxWi
*** ERROR: Symbol file could not be found.  Defaulted to expor
indows\system32\kernel32.dll -
1d 00000000'0012feb0 00000000'7742cdcd : 00000000'00000000 000
00000'00000000 00000000'00000000 : TestDefaultDebugger64!__tma

1e 00000000'0012ff60 00000000'7764c6e1 : 00000000'00000000 000
00000'00000000 00000000'00000000 : kernel32!BaseThreadInitThun
1f 00000000'0012ff90 00000000'00000000 : 00000000'00000000 000
00000'00000000 00000000'00000000 : ntdll!RtlUserThreadStart+0x

Creating C:\Program Files\Debugging Tools for Windows 64-bit\C
-12-2008__Time_16-55-5151\PID-4880__TESTDEFAULTDEBUGGER64.EXE_
Violation__mini_0670_2008-09-12_17-07-00-904_1310.dmp - mini w
Dump successfully written
Fri Sep 12 17:07:01.022 2008 (GMT+1): (1310.480): Access viola
05 (!!! second chance !!!)
---
--- 2nd chance AccessViolation exception ----
--------------------------------------------------------------

Occurrence happened at:
Debug session time: Fri Sep 12 17:07:01.023 2008 (GMT+1)
System Uptime: 9 days 6:20:33.336
Process Uptime: 0 days 0:13:05.432
  Kernel time: 0 days 0:00:00.078
  User time: 0 days 0:00:00.015

Faulting stack below ---
 # Child-SP          RetAddr           : Args to Child
                                  : Call Site
00 00000000'0012f0c8 00000000'00403d4d : 00000000'00000000 000
00000'01501522 01ff8000'0012fd50 : TestDefaultDebugger64!CTest
::OnBnClickedButton1
01 00000000'0012f0d0 00000000'00403f75 : ffffffff'd1012b6d 000
00000'00000000 00000000'00000001 : TestDefaultDebugger64!_AfxD

02 00000000'0012f110 00000000'004030cc : 00000000'002e13d0 000
00000'000003e8 000007fe'fdf25fa1 : TestDefaultDebugger64!CCmdT
169
03 00000000'0012f190 00000000'0040c18d : 00001e86'26f4e20a 000
00000'002e4d70 000007fe'fca02e5d : TestDefaultDebugger64!CDial
04 00000000'0012f1f0 00000000'0040cfbd : 00000020'00000000 000
00001'00000001 00000001'00000001 : TestDefaultDebugger64!CWnd:
05 00000000'0012f2a0 00000000'0040818f : 00000000'002e13d0 000
00000'01501522 000007fe'fc72a0ba : TestDefaultDebugger64!CWnd:
06 00000000'0012f400 00000000'0040b2e5 : 00000000'00000000 000
00000'01501522 00000000'00000111 : TestDefaultDebugger64!CWnd:
07 00000000'0012f460 00000000'0040b3d2 : 00000000'00000215 000
00000'00000000 00000000'00000000 : TestDefaultDebugger64!AfxCa
08 00000000'0012f520 00000000'7754e25a : 00000000'00000070 fff
fffff'ffffffff 00000000'7754e1df : TestDefaultDebugger64!AfxWn
09 00000000'0012f580 00000000'7754b7c9 : 00000000'00c53e90 000
00000'00c53eb8 00000000'0040b384 : USER32!GetWindowLongPtrW+0x
0a 00000000'0012f640 00000000'7754b85d : 00000000'00331256 000
00000'000003e8 00000000'00000111 : USER32!GetWindowThreadProce
0b 00000000'0012f6d0 000007fe'fc7273cb : 00000000'002e4960 000
00000'00000000 00000000'00000001 : USER32!SendMessageW+0x5d
0c 00000000'0012f720 000007fe'fc72adf9 : 00000000'01501522 000
00000'002e4960 00000000'00404aa7 : COMCTL32!TaskDialog+0x33503
0d 00000000'0012f760 00000000'7754e25a : 00000000'01501522 000
00000'00000000 00000000'77652016 : COMCTL32!TaskDialog+0x36f31
0e 00000000'0012f850 00000000'7754ec8f : 00000000'002e1428 000
00000'00c53e90 00000000'010ec7b0 : USER32!GetWindowLongPtrW+0x
0f 00000000'0012f910 00000000'77540e53 : 00000000'00000000 000
007fe'fc72a07c 00000000'00000001 : USER32!IsWindow+0x1bf
10 00000000'0012f990 00000000'0040706d : 00000000'0012fd50 000
fffff'fffffffe 00000000'00406a42 : USER32!UnhookWinEvent+0xf3
11 00000000'0012fa20 00000000'0040868c : 00000000'00331256 000
00000'002e1428 00000000'0012fd50 : TestDefaultDebugger64!CWnd:
0x35
12 00000000'0012fa60 00000000'0040309c : 00000000'00000000 000
fffff'fffffffe 00000000'00406a42 : TestDefaultDebugger64!CWnd:
+0x28
13 00000000'0012fa90 00000000'0040ae73 : 00000000'002e1428 000
fffff'fffffffe 00000000'00406a42 : TestDefaultDebugger64!CDial
ssage+0xc0
14 00000000'0012fad0 00000000'004047fc : 00000000'002e1428 000
00266'00000310 00000000'01501522 : TestDefaultDebugger64!CWnd:
ree+0x33
15 00000000'0012fb10 00000000'00404857 : ffffffff'fffffffe 000
00000'00000001 00000000'00000000 : TestDefaultDebugger64!AfxIn
Message+0x64
16 00000000'0012fb50 00000000'00404a17 : 00000000'0012fd50 000
00000'00000004 00000000'7754b85d : TestDefaultDebugger64!AfxPr
0x23
17 00000000'0012fb80 00000000'00404a57 : 00000000'00000000 000
00000'002e1428 00000000'00000000 : TestDefaultDebugger64!AfxIn
0x37
18 00000000'0012fbc0 00000000'0040a419 : 00000000'0012fd50 000
00000'00000001 00000000'7754b90f : TestDefaultDebugger64!AfxPu
19 00000000'0012fbf0 00000000'00403a3a : 00000000'0012fd50 000
00000'0044a5a4 00000000'00410008 : TestDefaultDebugger64!CWnd:

1a 00000000'0012fc70 00000000'00401139 : 00000000'0012fd50 000
00000'00000000 00000000'00000000 : TestDefaultDebugger64!CDial
1b 00000000'0012fd20 00000000'0042bbbd : 00000000'00000001 000
00000'00000000 00000000'002b26ce : TestDefaultDebugger64!CTest
::InitInstance+0xe9
1c 00000000'0012fe50 00000000'00414848 : 00000000'00000000 000
00000'00000000 00000000'00400000 : TestDefaultDebugger64!AfxWi
1d 00000000'0012feb0 00000000'7742cdcd : 00000000'00000000 000
00000'00000000 00000000'00000000 : TestDefaultDebugger64!__tma

1e 00000000'0012ff60 00000000'7764c6e1 : 00000000'00000000 000
00000'00000000 00000000'00000000 : kernel32!BaseThreadInitThun
1f 00000000'0012ff90 00000000'00000000 : 00000000'00000000 000
00000'00000000 00000000'00000000 : ntdll!RtlUserThreadStart+0x

Creating C:\Program Files\Debugging Tools for Windows 64-bit\C
-12-2008__Time_16-55-5151\PID-4880__TESTDEFAULTDEBUGGER64.EXE_
Violation__full_0670_2008-09-12_17-07-01-041_1310.dmp - mini w
Dump successfully written
```

The following message box might appear and that depends on AEDebug registry key and WER settings which are beyond the scope of this post:



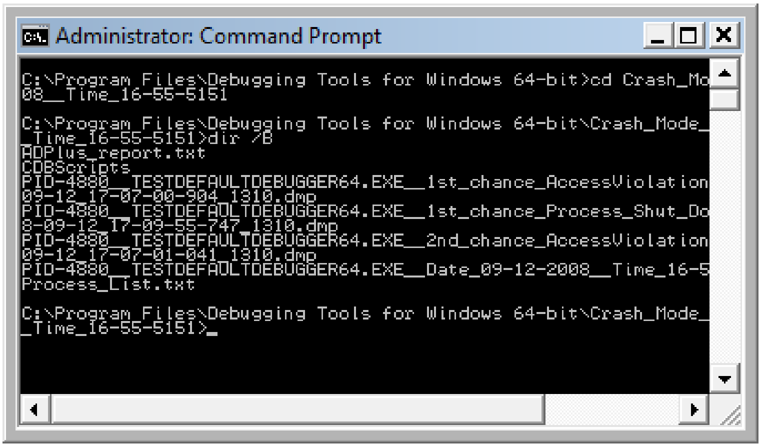
Test Default Debugger Tool has stopped working

Windows can check online for a solution to the problem.

In case it appears you can simply choose to close the program.

13. That’s it. All files appear in this folder:

```
C:\Program Files\Debugging Tools for Windows 64-bit\Crash_Mode__Date_09-12-2008__Time_16-55-5151:
```



```
C:\Program Files\Debugging Tools for Windows 64-bit>cd Crash_Mo
08__Time_16-55-5151

C:\Program Files\Debugging Tools for Windows 64-bit\Crash_Mode_
_Time_16-55-5151>dir /B
ADPlus_report.txt
CDBScripts
PID-4880__TESTDEFAULTDEBUGGER64.EXE__1st_chance_AccessViolation
09-12_17-07-00-904_1310.dmp
PID-4880__TESTDEFAULTDEBUGGER64.EXE__1st_chance_Process_Shut_Do
8-09-12_17-09-55-747_1310.dmp
PID-4880__TESTDEFAULTDEBUGGER64.EXE__2nd_chance_AccessViolation
09-12_17-07-01-041_1310.dmp
PID-4880__TESTDEFAULTDEBUGGER64.EXE__Date_09-12-2008__Time_16-5
Process_List.txt

C:\Program Files\Debugging Tools for Windows 64-bit\Crash_Mode_
_Time_16-55-5151>_
```

PART 11: MISCELLANIOUS

THREE MAIN IDEAS OF DEBUGGING

There is common tripartite view of intellectual history (Peter Watson, “Ideas: A History of Thought and Invention, from Fire to Freud”, ISBN 978-0060935641). The history of debugging can also be divided into main three ideas:

- Forward debugging

Conventional debugging where an engineer starts with initial conditions and during debugging tries to reproduce the problem or see the anomalies on the way to it. Delta debugging (http://en.wikipedia.org/wiki/Delta_Debugging) also falls into this category.

- Memory dump analysis

It is best described as taking memory slices for remote or postmortem analysis. Helps in problem identification, effective and efficient troubleshooting and also in debugging hard to reproduce or non-reproducible bugs.

- Backward debugging

Also called time travel debugging (page 440). Although in its early stages of development this debugging method is the future. In the most simple way, but technologically infeasible at the moment, it can be implemented as recording memory dumps in succession with every tick. Currently, to avoid saving redundant information and conserve storage the code is altered to save context dependent information for every processor instruction or high-level programming language statement. Another approach that comes with virtualization is coarse-grained backward debugging where memory and execution state is saved at certain important points or after specified time intervals.

PSEUDO-CORRUPT MEMORY DUMPS

Sometimes we get these errors when opening a few crash dumps:

```
...
Ignored in-page I/O error
Ignored in-page I/O error
Ignored in-page I/O error
Ignored in-page I/O error
Exception 0xc0000006 while accessing file mapping
Unable to read KLDR_DATA_TABLE_ENTRY at 8a3dd228 - NTSTATUS 0xC0000006
Ignored in-page I/O error
Ignored in-page I/O error
...
```

We might wonder whether something was wrong with our disk or network drive mapping where file dumps are stored or this is another sign of **Corrupt Dump** pattern (page 151). Because we can also notice these errors when we keep dump files open for weeks and then come back to them we should close and open new drive mappings and/or reopen dump files.

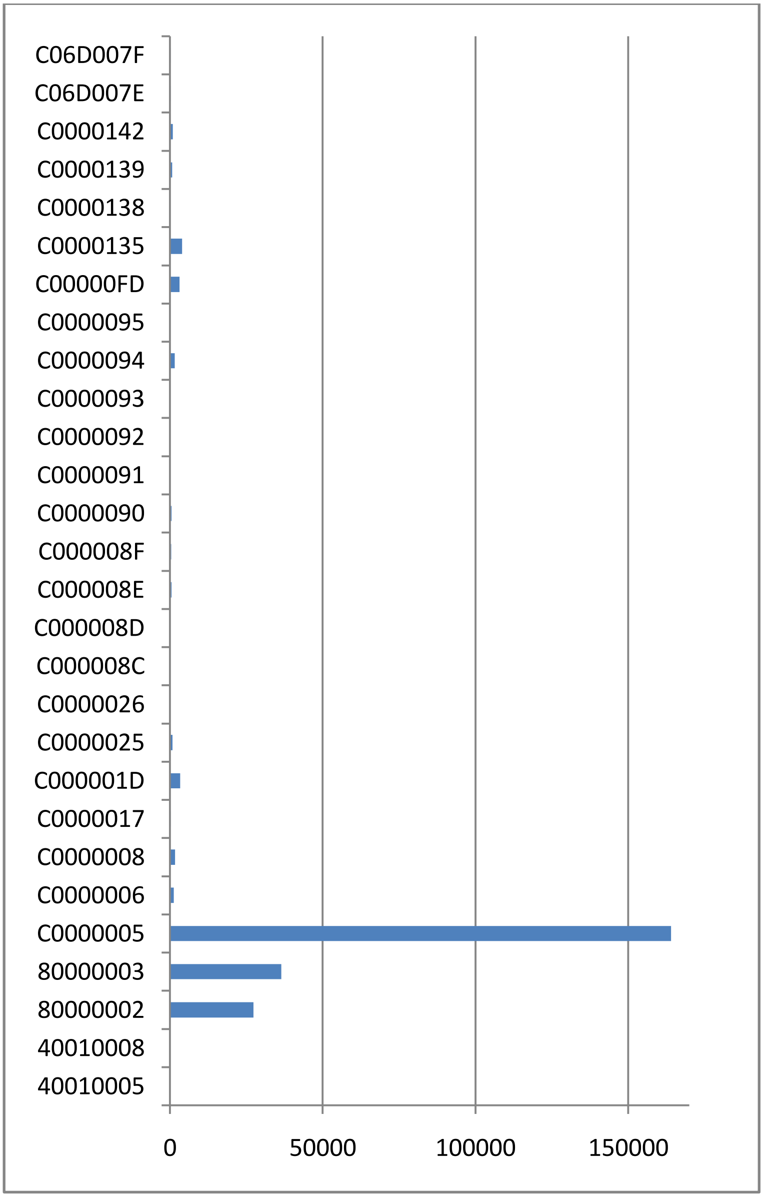
WIN32 EXCEPTION FREQUENCIES

Now we do the same Google counting procedure for exceptions like we did for bugcheck frequencies (page 429). Here are results for exceptions listed in Visual C++ Debug \ Exceptions dialog:

Control-C	40010005	43
Control-Break	40010008	7
Datatype misalignment	**80000002**	**27300**
Breakpoint	**80000003**	**36400**
Access violation	**C0000005**	**164000**
In page error	C0000006	1210
Invalid handle	**C0000008**	**1670**
Not enough quota	C0000017	176
Illegal instruction	**C000001D**	**3400**
Cannot continue	C0000025	804
Invalid exception disposition	C0000026	121
Array bounds exceeded	C000008C	100
Floating-point denormal operand	C000008D	84
Floating-point division by zero	C000008E	523
Floating-point inexact result	C000008F	401
Floating-point invalid operation	C0000090	509
Floating-point overflow	C0000091	121
Floating-point stack check	C0000092	102
Floating-point underflow	C0000093	138
Integer division by zero	**C0000094**	**1610**
Integer overflow	C0000095	99
Stack overflow	**C00000FD**	**3110**
Unable to locate component	**C0000135**	**3970**
Ordinal not found	C0000138	43
Entry point not found	C0000139	724

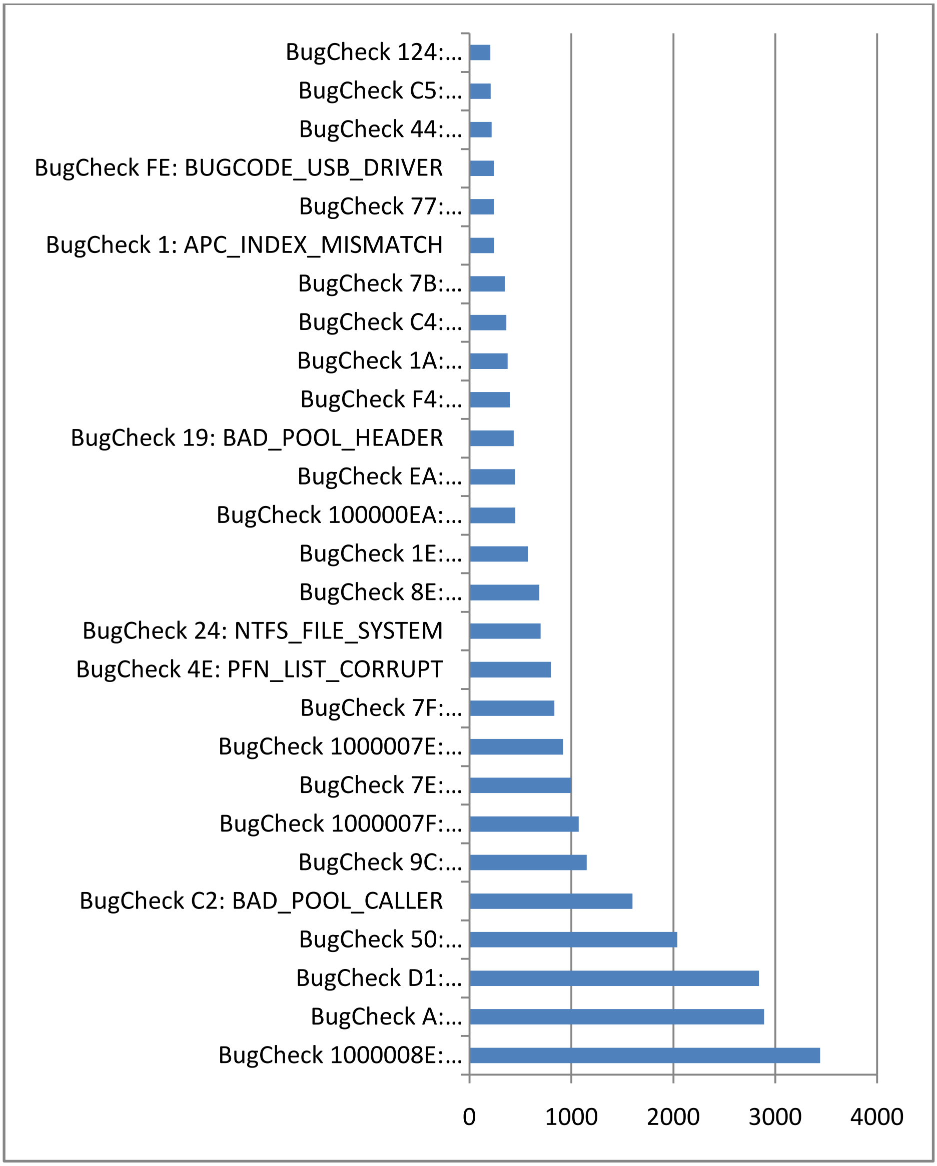
DLL initialization failed	C0000142	918
Module not found	C06D007E	171
Procedure not found	C06D007F	248

The corresponding graph:



BUGCHECK FREQUENCIES

259 bugchecks are documented in WinDbg help. Google search for every one at the time of this writing give results shown on the following distribution graph which was cut off for data with less than 200 matches:



There is some noise in search results and matches do not always correspond to WinDbg bugcheck analysis output but we can get rough idea about bugcheck frequency. For example, unhandled exceptions in kernel mode, IRQL contract violation, pool corruption and hardware failures are the most frequent. Here is the full table:

BugCheck 1000008E: KERNEL_MODE_EXCEPTION_NOT_HANDLED_M	3440
BugCheck A: IRQL_NOT_LESS_OR_EQUAL	2890
BugCheck D1: DRIVER_IRQL_NOT_LESS_OR_EQUAL	2840
BugCheck 50: PAGE_FAULT_IN_NONPAGED_AREA	2040
BugCheck C2: BAD_POOL_CALLER	1600
BugCheck 9C: MACHINE_CHECK_EXCEPTION	1150
BugCheck 1000007F: UNEXPECTED_KERNEL_MODE_TRAP_M	1070
BugCheck 7E: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED	998
BugCheck 1000007E: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED_M	917
BugCheck 7F: UNEXPECTED_KERNEL_MODE_TRAP	834
BugCheck 4E: PFN_LIST_CORRUPT	799
BugCheck 24: NTFS_FILE_SYSTEM	697
BugCheck 8E: KERNEL_MODE_EXCEPTION_NOT_HANDLED	686
BugCheck 1E: KMODE_EXCEPTION_NOT_HANDLED	571
BugCheck 100000EA: THREAD_STUCK_IN_DEVICE_DRIVER_M	450
BugCheck EA: THREAD_STUCK_IN_DEVICE_DRIVER	446
BugCheck 19: BAD_POOL_HEADER	434
BugCheck F4: CRITICAL_OBJECT_TERMINATION	397
BugCheck 1A: MEMORY_MANAGEMENT	373
BugCheck C4: DRIVER_VERIFIER_DETECTED_VIOLATION	360
BugCheck 7B: INACCESSIBLE_BOOT_DEVICE	347
BugCheck 1: APC_INDEX_MISMATCH	242
BugCheck 77: KERNEL_STACK_INPAGE_ERROR	240
BugCheck FE: BUGCODE_USB_DRIVER	239
BugCheck 44: MULTIPLE_IRP_COMPLETE_REQUESTS	216
BugCheck C5: DRIVER_CORRUPTED_EXPOOL	207

BugCheck 124: WHEA_UNCORRECTABLE_ERROR	204
BugCheck C000021A: STATUS_SYSTEM_PROCESS_TERMINATED	187
BugCheck 20: KERNEL_APC_PENDING_DURING_EXIT	168
BugCheck B8: ATTEMPTED_SWITCH_FROM_DPC	124
BugCheck 5: INVALID_PROCESS_ATTACH_ATTEMPT	123
BugCheck C: MAXIMUM_WAIT_OBJECTS_EXCEEDED	110
BugCheck 7A: KERNEL_DATA_INPAGE_ERROR	110
BugCheck DE: POOL_CORRUPTION_IN_FILE_AREA	106
BugCheck A0: INTERNAL_POWER_ERROR	104
BugCheck FC: ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY	101
BugCheck 9F: DRIVER_POWER_STATE_FAILURE	98
BugCheck E2: MANUALLY_INITIATED_CRASH	98
BugCheck 2: DEVICE_QUEUE_NOT_BUSY	97
BugCheck AB: SESSION_HAS_VALID_POOL_ON_EXIT	86
BugCheck 93: INVALID_KERNEL_HANDLE	83
BugCheck 51: REGISTRY_ERROR	73
BugCheck 3: INVALID_AFFINITY_SET	71
BugCheck 35: NO_MORE_IRP_STACK_LOCATIONS	71
BugCheck 3B: SYSTEM_SERVICE_EXCEPTION	70
BugCheck CE: DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS	65
BugCheck C1: SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION	59
BugCheck E3: RESOURCE_NOT_OWNED	43
BugCheck 109: CRITICAL_STRUCTURE_CORRUPTION	33
BugCheck E: NO_USER_MODE_CONTEXT	32
BugCheck D: MUTEX_LEVEL_NUMBER_VIOLATION	23
BugCheck 12: TRAP_CAUSE_UNKNOWN	14
BugCheck 23: FAT_FILE_SYSTEM	10
BugCheck 116: VIDEO_TDR_ERROR	10

BugCheck 9: IRQL_NOT_GREATER_OR_EQUAL	9
BugCheck 10D: WDF_VIOLATION	9
BugCheck 4: INVALID_DATA_ACCESS_TRAP	8
BugCheck 6: INVALID_PROCESS_DETACH_ATTEMPT	8
BugCheck 3F: NO_MORE_SYSTEM_PTES	8
BugCheck C9: DRIVER_VERIFIER_IOMANAGER_VIOLATION	8
BugCheck D0: DRIVER_CORRUPTED_MMPOOL	8
BugCheck 117: VIDEO_TDR_TIMEOUT_DETECTED	8
BugCheck F: SPIN_LOCK_ALREADY_OWNED	7
BugCheck 11: THREAD_NOT_MUTEX_OWNER	6
BugCheck 6B: PROCESS1_INITIALIZATION_FAILED	6
BugCheck BE: ATTEMPTED_WRITE_TO_READONLY_MEMORY	6
BugCheck B: NO_EXCEPTION_HANDLING_SUPPORT	5
BugCheck 27: RDR_FILE_SYSTEM	5
BugCheck 41: MUST_SUCCEED_POOL_EMPTY	5
BugCheck 96: INVALID_WORK_QUEUE_ITEM	5
BugCheck D5: DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL	5
BugCheck DA: SYSTEM_PTE_MISUSE	5
BugCheck E1: WORKER_THREAD_RETURNED_AT_BAD_IRQL	5
BugCheck E6: DRIVER_VERIFIER_DMA_VIOLATION	5
BugCheck 10E: VIDEO_MEMORY_MANAGEMENT_INTERNAL	5
BugCheck 8: IRQL_NOT_DISPATCH_LEVEL	4
BugCheck 18: REFERENCE_BY_POINTER	4
BugCheck 34: CACHE_MANAGER	4
BugCheck 76: PROCESS_HAS_LOCKED_PAGES	4
BugCheck CA: PNP_DETECTED_FATAL_ERROR	4
BugCheck CB: DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS	4
BugCheck D2: BUGCODE_ID_DRIVER	4

BugCheck D4: SYSTEM_SCAN_AT_RAISED_IRQL_CAUGHT_IMPROPER_DRIVER_UNLOAD	4
BugCheck D9: LOCKED_PAGES_TRACKER_CORRUPTION	4
BugCheck F7: DRIVER_OVERRAN_STACK_BUFFER	4
BugCheck 101: CLOCK_WATCHDOG_TIMEOUT	4
BugCheck C0000218: STATUS_CANNOT_LOAD_REGISTRY_FILE	4
BugCheck 7: INVALID_SOFTWARE_INTERRUPT	3
BugCheck 13: EMPTY_THREAD_REAPER_LIST	3
BugCheck 14: CREATE_DELETE_LOCK_NOT_LOCKED	3
BugCheck 22: FILE_SYSTEM	3
BugCheck 29: SECURITY_SYSTEM	3
BugCheck 39: SYSTEM_EXIT_OWNED_MUTEX	3
BugCheck 46: DEREF_UNKNOWN_LOGON_SESSION	3
BugCheck 4D: NO_PAGES_AVAILABLE	3
BugCheck 9E: USER_MODE_HEALTH_MONITOR	3
BugCheck BA: SESSION_HAS_VALID_VIEWS_ON_EXIT	3
BugCheck D3: DRIVER_PORTION_MUST_BE_NONPAGED	3
BugCheck D6: DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION	3
BugCheck E7: INVALID_FLOATING_POINT_STATE	3
BugCheck 100: LOADER_BLOCK_MISMATCH	3
BugCheck 106: AGP_ILLEGALLY_REPROGRAMMED	3
BugCheck DEADDEAD: MANUALLY_INITIATED_CRASH1	3
BugCheck 10: SPIN_LOCK_NOT_OWNED	2
BugCheck 21: QUOTA_UNDERFLOW	2
BugCheck 26: CDFS_FILE_SYSTEM	2
BugCheck 30: SET_OF_INVALID_CONTEXT	2
BugCheck 33: UNEXPECTED_INITIALIZATION_CALL	2
BugCheck 36: DEVICE_REFERENCE_COUNT_NOT_ZERO	2
BugCheck 37: FLOPPY_INTERNAL_ERROR	2

BugCheck 40: TARGET_MDL_TOO_SMALL	2
BugCheck 42: ATDISK_DRIVER_INTERNAL	2
BugCheck 4A: IRQL_GT_ZERO_AT_SYSTEM_SERVICE	2
BugCheck 72: ASSIGN_DRIVE_LETTERS_FAILED	2
BugCheck 73: CONFIG_LIST_FAILED	2
BugCheck CD: PAGE_FAULT_BEYOND_END_OF_ALLOCATION	2
BugCheck CF: TERMINAL_SERVER_DRIVER_MADE_INCORRECT_MEMORY_REFERENCE	2
BugCheck E4: WORKER_INVALID	2
BugCheck 104: AGP_INVALID_ACCESS	2
BugCheck 15: LAST_CHANCE_CALLED_FROM_KMODE	1
BugCheck 16: CID_HANDLE_CREATION	1
BugCheck 17: CID_HANDLE_DELETION	1
BugCheck 28: CORRUPT_ACCESS_TOKEN	1
BugCheck 38: SERIAL_DRIVER_INTERNAL	1
BugCheck 43: NO_SUCH_PARTITION	1
BugCheck 45: INSUFFICIENT_SYSTEM_MAP_REGS	1
BugCheck 48: CANCEL_STATE_IN_COMPLETED_IRP	1
BugCheck 49: PAGE_FAULT_WITH_INTERRUPTS_OFF	1
BugCheck 57: XNS_INTERNAL_ERROR	1
BugCheck 58: FTDISK_INTERNAL_ERROR	1
BugCheck 5C: HAL_INITIALIZATION_FAILED	1
BugCheck 60: PROCESS_INITIALIZATION_FAILED	1
BugCheck 62: OBJECT1_INITIALIZATION_FAILED	1
BugCheck 63: SECURITY1_INITIALIZATION_FAILED	1
BugCheck 64: SYMBOLIC_INITIALIZATION_FAILED	1
BugCheck 67: CONFIG_INITIALIZATION_FAILED	1
BugCheck 68: FILE_INITIALIZATION_FAILED	1
BugCheck 70: SESSION4_INITIALIZATION_FAILED	1

BugCheck 74: BAD_SYSTEM_CONFIG_INFO	1
BugCheck 75: CANNOT_WRITE_CONFIGURATION	1
BugCheck 79: MISMATCHED_HAL	1
BugCheck 7D: INSTALL_MORE_MEMORY	1
BugCheck 80: NMI_HARDWARE_FAILURE	1
BugCheck 82: DFS_FILE_SYSTEM	1
BugCheck 85: SETUP_FAILURE	1
BugCheck 99: INVALID_REGION_OR_SEGMENT	1
BugCheck A1: PCI_BUS_DRIVER_INTERNAL	1
BugCheck BF: MUTEX_ALREADY_OWNED	1
BugCheck CC: PAGE_FAULT_IN_FREED_SPECIAL_POOL	1
BugCheck DB: DRIVER_CORRUPTED_SYSPTES	1
BugCheck E8: INVALID_CANCEL_OF_FILE_OPEN	1
BugCheck ED: UNMOUNTABLE_BOOT_VOLUME	1
BugCheck FD: DIRTY_NOWRITE_PAGES_CONGESTION	1
BugCheck 108: THIRD_PARTY_FILE_SYSTEM_FAILURE	1
BugCheck 111: RECURSIVE_NMI	1
BugCheck 121: DRIVER_VIOLATION	1
BugCheck 122: WHEA_INTERNAL_ERROR	1
BugCheck 1B: PFN_SHARE_COUNT	0
BugCheck 1C: PFN_REFERENCE_COUNT	0
BugCheck 1D: NO_SPIN_LOCK_AVAILABLE	0
BugCheck 1F: SHARED_RESOURCE_CONV_ERROR	0
BugCheck 25: NPFS_FILE_SYSTEM	0
BugCheck 2A: INCONSISTENT_IRP	0
BugCheck 2B: PANIC_STACK_SWITCH	0
BugCheck 2C: PORT_DRIVER_INTERNAL	0
BugCheck 2D: SCSI_DISK_DRIVER_INTERNAL	0
BugCheck 2E: DATA_BUS_ERROR	0

BugCheck 2F: INSTRUCTION_BUS_ERROR	0
BugCheck 31: PHASE0_INITIALIZATION_FAILED	0
BugCheck 32: PHASE1_INITIALIZATION_FAILED	0
BugCheck 3A: SYSTEM_UNWIND_PREVIOUS_USER	0
BugCheck 3C: INTERRUPT_UNWIND_ATTEMPTED	0
BugCheck 3D: INTERRUPT_EXCEPTION_NOT_HANDLED	0
BugCheck 3E: MULTIPROCESSOR_CONFIGURATION_NOT_SUPPORTED	0
BugCheck 47: REF_UNKNOWN_LOGON_SESSION	0
BugCheck 4B: STREAMS_INTERNAL_ERROR	0
BugCheck 4C: FATAL_UNHANDLED_HARD_ERROR	0
BugCheck 4F: NDIS_INTERNAL_ERROR	0
BugCheck 52: MAILSLOT_FILE_SYSTEM	0
BugCheck 53: NO_BOOT_DEVICE	0
BugCheck 54: LM_SERVER_INTERNAL_ERROR	0
BugCheck 55: DATA_COHERENCY_EXCEPTION	0
BugCheck 56: INSTRUCTION_COHERENCY_EXCEPTION	0
BugCheck 59: PINBALL_FILE_SYSTEM	0
BugCheck 5A: CRITICAL_SERVICE_FAILED	0
BugCheck 5B: SET_ENV_VAR_FAILED	0
BugCheck 5D: UNSUPPORTED_PROCESSOR	0
BugCheck 5E: OBJECT_INITIALIZATION_FAILED	0
BugCheck 5F: SECURITY_INITIALIZATION_FAILED	0
BugCheck 61: HAL1_INITIALIZATION_FAILED	0
BugCheck 65: MEMORY1_INITIALIZATION_FAILED	0
BugCheck 66: CACHE_INITIALIZATION_FAILED	0
BugCheck 69: IO1_INITIALIZATION_FAILED	0
BugCheck 6A: LPC_INITIALIZATION_FAILED	0
BugCheck 6C: REFMON_INITIALIZATION_FAILED	0
BugCheck 6D: SESSION1_INITIALIZATION_FAILED	0

BugCheck 6E: SESSION2_INITIALIZATION_FAILED	0
BugCheck 6F: SESSION3_INITIALIZATION_FAILED	0
BugCheck 71: SESSION5_INITIALIZATION_FAILED	0
BugCheck 78: PHASE0_EXCEPTION	0
BugCheck 7C: BUGCODE_NDIS_DRIVER	0
BugCheck 81: SPIN_LOCK_INIT_FAILURE	0
BugCheck 8B: MBR_CHECKSUM_MISMATCH	0
BugCheck 8F: PP0_INITIALIZATION_FAILED	0
BugCheck 90: PP1_INITIALIZATION_FAILED	0
BugCheck 92: UP_DRIVER_ON_MP_SYSTEM	0
BugCheck 94: KERNEL_STACK_LOCKED_AT_EXIT	0
BugCheck 97: BOUND_IMAGE_UNSUPPORTED	0
BugCheck 98: END_OF_NT_EVALUATION_PERIOD	0
BugCheck 9A: SYSTEM_LICENSE_VIOLATION	0
BugCheck 9B: UDFS_FILE_SYSTEM	0
BugCheck A2: MEMORY_IMAGE_CORRUPT	0
BugCheck A3: ACPI_DRIVER_INTERNAL	0
BugCheck A4: CNSS_FILE_SYSTEM_FILTER	0
BugCheck A5: ACPI_BIOS_ERROR	0
BugCheck A7: BAD_EXHANDLE	0
BugCheck AC: HAL_MEMORY_ALLOCATION	0
BugCheck AD: VIDEO_DRIVER_DEBUG_REPORT_REQUEST	0
BugCheck B4: VIDEO_DRIVER_INIT_FAILURE	0
BugCheck B9: CHIPSET_DETECTED_ERROR	0
BugCheck BB: NETWORK_BOOT_INITIALIZATION_FAILED	0
BugCheck BC: NETWORK_BOOT_DUPLICATE_ADDRESS	0
BugCheck C6: DRIVER_CAUGHT_MODIFYING_FREED_POOL	0
BugCheck C7: TIMER_OR_DPC_INVALID	0
BugCheck C8: IRQL_UNEXPECTED_VALUE	0

BugCheck D7: DRIVER_UNMAPPING_INVALID_VIEW	0
BugCheck D8: DRIVER_USED_EXCESSIVE_PTES	0
BugCheck DC: DRIVER_INVALID_STACK_ACCESS	0
BugCheck DF: IMPERSONATING_WORKER_THREAD	0
BugCheck E0: ACPI_BIOS_FATAL_ERROR	0
BugCheck E9: ACTIVE_EX_WORKER_THREAD_TERMINATION	0
BugCheck EB: DIRTY_MAPPED_PAGES_CONGESTION	0
BugCheck EC: SESSION_HAS_VALID_SPECIAL_POOL_ON_EXIT	0
BugCheck EF: CRITICAL_PROCESS_DIED	0
BugCheck F1: SCSI_VERIFIER_DETECTED_VIOLATION	0
BugCheck F3: DISORDERLY_SHUTDOWN	0
BugCheck F5: FLTMGR_FILE_SYSTEM	0
BugCheck F6: PCI_VERIFIER_DETECTED_VIOLATION	0
BugCheck F8: RAMDISK_BOOT_INITIALIZATION_FAILED	0
BugCheck F9: DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN	0
BugCheck FA: HTTP_DRIVER_CORRUPTED	0
BugCheck FF: RESERVE_QUEUE_OVERFLOW	0
BugCheck 105: AGP_GART_CORRUPTION	0
BugCheck 10A: APP_TAGGING_INITIALIZATION_FAILED	0
BugCheck 10C: FSRTL_EXTRA_CREATE_PARAMETER_VIOLATION	0
BugCheck 10F: RESOURCE_MANAGER_EXCEPTION_NOT_HANDLED	0
BugCheck 112: MSRPC_STATE_VIOLATION	0
BugCheck 113: VIDEO_DXGKRNL_FATAL_ERROR	0
BugCheck 114: VIDEO_SHADOW_DRIVER_FATAL_ERROR	0
BugCheck 115: AGP_INTERNAL	0
BugCheck 119: VIDEO_SCHEDULER_INTERNAL_ERROR	0
BugCheck 11A: EM_INITIALIZATION_FAILURE	0
BugCheck 11B: DRIVER_RETURNED_HOLDING_CANCEL_LOCK	0

BugCheck 11C: ATTEMPTED_WRITE_TO_CM_PROTECTED_STORAGE	0
BugCheck 11D: EVENT_TRACING_FATAL_ERROR	0
BugCheck 127: PAGE_NOT_ZERO	0
BugCheck 12B: FAULTY_HARDWARE_CORRUPTED_PAGE	0
BugCheck 12C: EXFAT_FILE_SYSTEM	0
BugCheck C0000221: STATUS_IMAGE_CHECKSUM_MISMATCH	0

TIME TRAVEL DEBUGGING

There is some research going for tools that allow to step through unmanaged and native code backwards when we do crash dump analysis or live debugging on Windows platforms:

- *iDNA: Time Travel Debugging*

  http://www.cs.wisc.edu/areas/pl/seminar/fall05/Bhansali.ppt

- *Framework for Instruction-level Tracing and Analysis of Program Executions*

  http://blogs.msdn.com/cse/attachment/1077668.ashx

It would be really good if Microsoft integrates this into Debugging Tools for Windows.

I/O AND MEMORY PRIORITY IN VISTA

There are additional IoPriority and PagePriority values when using **!thread** command in kernel and complete memory dumps coming from Vista:

```
THREAD 8362a390  Cid 0b90.0b94  Teb: 7ffdf000 Win32Thread: ff34c848 WAIT:
(WrUserRequest) UserMode Non-Alertable
    83656de0  SynchronizationEvent
Not impersonating
DeviceMap                 a7766db8
Owning Process            8362a638       Image:         explorer.exe
Wait Start TickCount      43496          Ticks: 3 (0:00:00:00.046)
Context Switch Count      14502
UserTime                  00:00:00.436
KernelTime                00:00:00.608
Win32 Start Address Explorer!wWinMainCRTStartup (0x0052d070)
Stack Init 9e8b3000 Current 9e8b2c10 Base 9e8b3000 Limit 9e8b0000 Call 0
Priority 12 BasePriority 8 PriorityDecrement 2 IoPriority 2 PagePriority 5
ChildEBP RetAddr
9e8b2c28 81cac9cf nt!KiSwapContext+0×26
9e8b2c64 81c293a7 nt!KiSwapThread+0×389
9e8b2cc0 8cedb8ed nt!KeWaitForSingleObject+0×414
9e8b2d1c 8cedb724 win32k!xxxRealSleepThread+0×1ad
9e8b2d38 8ced573c win32k!xxxSleepThread+0×2d
9e8b2d4c 8ced5759 win32k!xxxRealWaitMessageEx+0×12
9e8b2d5c 81c8caaa win32k!NtUserWaitMessage+0×14
9e8b2d5c 77490f34 nt!KiFastCallEntry+0×12a
000ffb94 761db5bc ntdll!KiFastSystemCallRet
000ffb98 765e07f6 USER32!NtUserWaitMessage+0xc
000ffbb0 76566f4e SHELL32!CDesktopBrowser::_MessageLoop+0×4c
000ffbbc 00529039 SHELL32!SHDesktopMessageLoop+0×24
000ffea8 0052d1e0 Explorer!wWinMain+0×447
000fff3c 75f33833 Explorer!_initterm_e+0×1b1
000fff48 7746a9bd kernel32!BaseThreadInitThunk+0xe
000fff88 00000000 ntdll!_RtlUserThreadStart+0×23
```

These are new thread priorities added to Vista kernel and explained in the following articles:

Inside the Windows Vista Kernel: Part 1 (I/O Priority section)
Inside the Windows Vista Kernel: Part 2 (Memory Priorities section)
http://www.microsoft.com/technet/technetmag/issues/2007/02/VistaKernel/

We can change it for any process according to:
blogs.technet.com/vitalipro/archive/2007/02/16/645675.aspx

For example, for notepad.exe:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image
File Execution Options\notepad.exe\PerfOptions]
"IoPriority"=dword:00000001
"PagePriority"=dword:00000001
```

```
THREAD 838edd78  Cid 0378.0d3c  Teb: 7ffdf000 Win32Thread: fed7e848 WAIT:
(WrUserRequest) UserMode Non-Alertable
    838c84a0  SynchronizationEvent
Not impersonating
DeviceMap                 a7766db8
Owning Process            838e25a0       Image:         notepad.exe
Wait Start TickCount      12967          Ticks: 30532 (0:00:07:56.302)
Context Switch Count      490
UserTime                  00:00:00.000
KernelTime                00:00:00.109
Win32 Start Address notepad!WinMainCRTStartup (0x003631f8)
Stack Init a691b000 Current a691ab68 Base a691b000 Limit a6918000 Call 0
Priority 12 BasePriority 8 PriorityDecrement 2 IoPriority 1 PagePriority 1
Kernel stack not resident.
ChildEBP RetAddr
a691ab80 81cac9cf nt!KiSwapContext+0×26
a691abbc 81c293a7 nt!KiSwapThread+0×389
a691ac18 8cedb8ed nt!KeWaitForSingleObject+0×414
a691ac74 8cedb724 win32k!xxxRealSleepThread+0×1ad
a691ac90 8ced9976 win32k!xxxSleepThread+0×2d
a691ace8 8cedd983 win32k!xxxRealInternalGetMessage+0×4a4
a691ad4c 81c8caaa win32k!NtUserGetMessage+0×3f
a691ad4c 77490f34 nt!KiFastCallEntry+0×12a
0006f6d0 761e199a ntdll!KiFastSystemCallRet
0006f6d4 761e19cd USER32!NtUserGetMessage+0xc
0006f6f0 0036149c USER32!GetMessageW+0×33
0006f730 00361971 notepad!WinMain+0xec
0006f7c0 75f33833 notepad!_initterm_e+0×1a1
0006f7cc 7746a9bd kernel32!BaseThreadInitThunk+0xe
0006f80c 00000000 ntdll!_RtlUserThreadStart+0×23
```

APPENDIX A

CRASH DUMP FILE EXAMPLES

Selected crash dumps for some case studies and patterns are custom made from toy programs and preprocessed to clear sensitive information. They can be found at the following public FTP address:

ftp://dumpanalysis.org/pub/

Here is the list at the time of this writing:

- **Missing Component** (page 233) - CDAPatternMissingComponent.zip
- **Duplicated Module** (page 294) - CDAPatternDuplicatedModule.zip
- **Nested Exceptions** (page 305) - CDAPatternNestedExceptions.zip
- **Large Heap Allocations** (page 137) - LargeHeapAllocations.zip

APPENDIX B

WINDBG.ORG: WINDBG QUICK LINKS

Sometimes we need a quick link to install Debugging Tools for Windows and for other related information such as standard symbol server path, common commands, etc. For this purpose there is handy windbg.org domain. Currently its main page has the following links:

- Download links to 32-bit and 64-bit versions
- The example of standard symbol paths for Microsoft and Citrix
- Link to HTML version of Crash Dump Analysis Poster
- Link to Crash Dump Analysis Checklist
- Books on WinDbg to order

APPENDIX C

DUMP2WAVE SOURCE CODE

```
// Dump2Wave version 1.2.1 (c) Dmitry Vostokov
// GNU GENERAL PUBLIC LICENSE
// http://www.gnu.org/licenses/gpl-3.0.txt

#include <iostream>
#include <process.h>
#include <windows.h>

#pragma pack(1)

typedef struct {

 unsigned int   riff;   // 'RIFF'
 unsigned int   length; // dump size + sizeof(WAVEFILEHDR) - 8
 unsigned int   wave;   // 'WAVE'
 unsigned int   fmt;    // 'fmt '
 unsigned int   fmtlen; // 16
 unsigned short code;   // 1
 unsigned short channels;  // 2
 unsigned int   sps;    // 44100
 unsigned int   avgbps; // sps*channels*(sbits/8)
 unsigned short alignment; // 4
 unsigned short sbits;  // 16
 unsigned int   data;   // 'data'
 unsigned int   datalen;

} WAVEFILEHDR;

#pragma pack()

WAVEFILEHDR hdr = {'FFIR', sizeof(WAVEFILEHDR) - 8, 'EVAW',
  ' tmf', 16, 1, 2, 44100, 44100*4, 4, 16, 'atad', 0};
```

```
void DisplayError (LPCSTR szPrefix)
{
 LPSTR errMsg;
 CHAR  szMsg[256];
 strncpy(szMsg, szPrefix, 128);
 DWORD gle = GetLastError();
 if (gle && FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER|
  FORMAT_MESSAGE_FROM_SYSTEM, NULL, gle, 0,
  (LPSTR)&errMsg, 0, NULL))
 {
  strcat(szMsg, ": ");
  strncat(szMsg, errMsg, 120);
 }
 std::cout << szMsg << std::endl;
 LocalFree(errMsg);
}

int main(int argc, char* argv[])
{
 std::cout << std::endl << "Dump2Wave version 1.2.1" <<
    std::endl << "Written by Dmitry Vostokov, 2006" <<
    std::endl << std::endl;
 if (argc < 3)
 {
  std::cout << "Usage: Dump2Wave dumpfile wavefile [44100|22050|11025|8000
16|8 2|1]" << std::endl;
  return -1;
 }

 HANDLE hFile = CreateFile(argv[1],
  GENERIC_READ, FILE_SHARE_READ, NULL,
  OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
 if (hFile == INVALID_HANDLE_VALUE)
 {
  DisplayError("Cannot read dump file");
  return -1;
 }

 DWORD dwDumpSizeHigh = 0;
 DWORD dwDumpSizeLow = GetFileSize(hFile, &dwDumpSizeHigh);
 CloseHandle(hFile);

 if (dwDumpSizeHigh)
 {
  std::cout << "The dump file must be less than 4Gb" <<
  std::endl;
  return -1;
 }
 if (argc == 6)
 {
  hdr.channels = atoi(argv[5]);
  hdr.sps = atoi(argv[3]);
  hdr.sbits = atoi(argv[4]);
  hdr.avgbps = hdr.sps*hdr.channels*(hdr.sbits/8);
  hdr.alignment = hdr.channels*(hdr.sbits/8);
 }

 dwDumpSizeLow = (dwDumpSizeLow/hdr.alignment)*(hdr.alignment);

 hdr.length += dwDumpSizeLow;
 hdr.datalen = dwDumpSizeLow;

 hFile = CreateFile(argv[2], GENERIC_WRITE, 0,
  NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
 if (hFile == INVALID_HANDLE_VALUE)
 {
  DisplayError("Cannot create wave header file");
  return -1;
 }

 DWORD dwWritten;
 if (!WriteFile(hFile, &hdr, sizeof(hdr), &dwWritten, NULL))
 {
  DisplayError("Cannot write wave header file");
  CloseHandle(hFile);
  return -1;
 }


 CloseHandle(hFile);

 std::string str = "copy \"";
 str += argv[2];
 str += "\" /B + \"";
 str += argv[1];
 str += "\" /B \"";
 str += argv[2];
 str += "\" /B";

 system(str.c_str());

 return 0;
}
```

APPENDIX D

DUMP2PICTURE SOURCE CODE

```
// Dump2Picture version 1.1 (c) Dmitry Vostokov
// GNU GENERAL PUBLIC LICENSE
// http://www.gnu.org/licenses/gpl-3.0.txt

#include <math.h>
#include <iostream>
#include <process.h>
#include <windows.h>

BITMAPFILEHEADER bmfh = { 'MB', 0, 0, 0,
   sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER) };
BITMAPINFOHEADER bmih = { sizeof(BITMAPINFOHEADER), 0, 0, 1, 32,
   0, 0, 0, 0, 0, 0 };
RGBQUAD rgb[256];

void DisplayError (LPCSTR szPrefix)
{
 LPSTR errMsg;
 CHAR  szMsg[256];
 strncpy(szMsg, szPrefix, 128);
 DWORD gle = GetLastError();
 if (gle && FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER|FORMAT_MESSAGE_FROM_SYSTEM,
    NULL, gle, 0, (LPSTR)&errMsg, 0, NULL))
 {
  strcat(szMsg, ": ");
  strncat(szMsg, errMsg, 120);
 }
 std::cout << szMsg << std::endl;
 LocalFree(errMsg);
}
```

```
int main(int argc, char* argv[])
{
 std::cout << std::endl << "Dump2Picture version 1.1"
    << std::endl << "Written by Dmitry Vostokov, 2007"
    << std::endl << std::endl;
 if (argc < 3)
 {
  std::cout << "Usage: Dump2Picture dumpfile bmpfile [8|16|24|32]" <<
std::endl;
  return -1;
 }

 HANDLE hFile = CreateFile(argv[1], GENERIC_READ,
    FILE_SHARE_READ, NULL, OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL, NULL);
 if (hFile == INVALID_HANDLE_VALUE)
 {
  DisplayError("Cannot read dump file");
  return -1;
 }

 DWORD dwDumpSizeHigh = 0;
 DWORD dwDumpSizeLow = GetFileSize(hFile, &dwDumpSizeHigh);
 CloseHandle(hFile);

 if (dwDumpSizeHigh)
 {
  std::cout << "The dump file must be less than 4Gb"
     << std::endl;
  return -1;
 }

 if (argc == 4)
 {
  if (!strcmp(argv[argc-1],"8"))
  {
   bmih.biBitCount = 8;
   for (int i = 0; i < 256; ++i)
   {
    rgb[i].rgbBlue = rgb[i].rgbGreen = rgb[i].rgbRed = i;
    rgb[i].rgbReserved = 0;
   }
  }
  else if (!strcmp(argv[argc-1],"16"))
  {
   bmih.biBitCount = 16;
  }
  else if (!strcmp(argv[argc-1],"24"))
  {
   bmih.biBitCount = 24;
  }
  else
  {
   bmih.biBitCount = 32;
  }
 }

 bmih.biWidth = bmih.biHeight = sqrt((double)(dwDumpSizeLow/
    (bmih.biBitCount/8)));
 bmih.biWidth -= bmih.biWidth%2;
 if (bmih.biBitCount == 8 )
 {
  bmih.biWidth -= bmih.biWidth%8;
 }
 bmih.biHeight -= bmih.biHeight%2;
 bmih.biSizeImage = bmih.biWidth*bmih.biHeight*
    (bmih.biBitCount/8);
 if (bmih.biBitCount == 8 )
 {
  bmfh.bfOffBits += sizeof(rgb);
 }
 bmfh.bfSize = bmfh.bfOffBits + bmih.biSizeImage;

 hFile = CreateFile(argv[2], GENERIC_WRITE, 0, NULL,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
 if (hFile == INVALID_HANDLE_VALUE)
 {
  DisplayError("Cannot create bitmap header file");
  return -1;
 }

 DWORD dwWritten;
 if (!WriteFile(hFile, &bmfh, sizeof(bmfh), &dwWritten, NULL))
 {
  DisplayError("Cannot write bitmap header file");
  CloseHandle(hFile);
  return -1;
 }

 if (!WriteFile(hFile, &bmih, sizeof(bmih), &dwWritten, NULL))
 {
  DisplayError("Cannot write bitmap header file");
  CloseHandle(hFile);
  return -1;
 }

 if (bmih.biBitCount == 8 )
 {
  if (!WriteFile(hFile, &rgb, sizeof(rgb), &dwWritten, NULL))
  {
   DisplayError("Cannot write bitmap header file");
   CloseHandle(hFile);
   return -1;
  }
 }
```

```
 CloseHandle(hFile);

 std::string str = "copy \"";
 str += argv[2];
 str += "\" /B + \"";
 str += argv[1];
 str += "\" /B \"";
 str += argv[2];
 str += "\" /B";

 system(str.c_str());

 return 0;
}
```

APPENDIX E

CRASH DUMP ANALYSIS CHECKLIST

General:

- Internal database(s) search
- Google or Microsoft search for suspected components as this could be a known issue. Sometimes a simple search immediately points to the fix on a vendor's site
- The tool used to save a dump (to flag false positive, incomplete or inconsistent dumps)
- OS/SP version
- Language
- Debug time
- System uptime
- Computer name
- _.kframes 100_

Application crash or hang:

- Default analysis _(!analyze -v_ or _!analyze -v -hang_ for hangs)
- Critical sections (_!locks and !locks -v, !cs -s -l -o_) for both crashes and hangs
- Component timestamps, duplication and paths. DLL Hell?
- Do any newer components exist?
- Process threads (_~*kv_ or _!uniqstack_)
- Process uptime
- Your components on the full raw stack of the problem thread
- Your components on the full raw stack of the main application thread
- Process size
- Number of threads
- Gflags value (_!gflag_)
- Time consumed by thread (_!runaway_)
- Environment (_!peb_)
- Import table (_!dh_)
- Hooked functions (_!chkimg_)
- Exception handlers (_!exchain_)
- Computer name (_!envvar COMPUTERNAME_)

System hang:

- Default analysis (*!analyze -v -hang*)
- ERESOURCE contention (*!locks*)
- Processes and virtual memory including session space (*!vm 4*)
- Pools (*!poolused*)
- Waiting threads (*!stacks*)
- Critical system queues (*!exqueue f*)
- I/O (*!irpfind*)
- The list of all thread stack traces (*!process 0 ff* for W2K3/XP/Vista, ListProcessStacks script for W2K, Volume 1, page 222)
- LPC chain for suspected threads (*!lpc message*)
- Critical sections for suspected processes (*!ntsdexts.locks*)
- Sessions, session processes (*!session, !sprocess*)
- Processes (size, handle table size) (*!process 0 0*)
- Running threads (*!running*)
- Ready threads (*!ready*)
- DPC queues (*!dpcs*)
- The list of APCs (*!apc*)
- Internal queued spinlocks (*!qlocks*)
- Computer name (*dS srv!srvcomputername*)

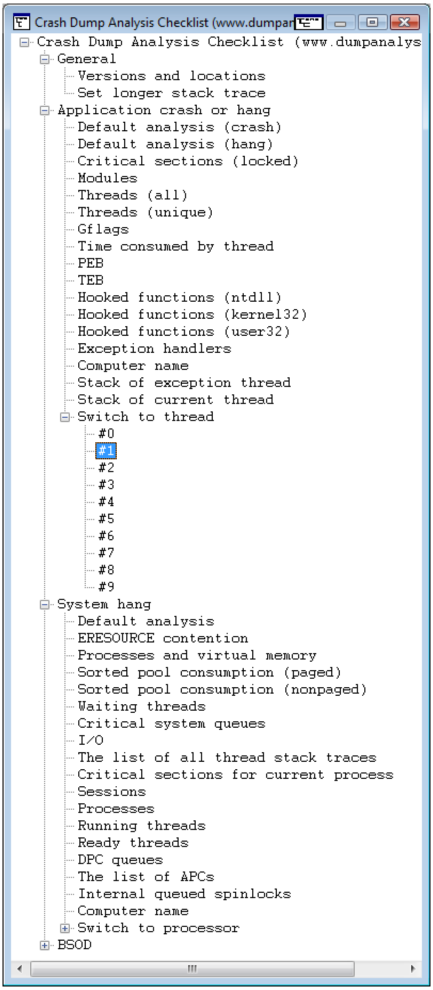
BSOD:

- Default analysis (*!analyze -v*)
- Pool address (*!pool*)
- Component timestamps.
- Processes and virtual memory (*!vm 4*)
- Current threads on other processors
- Raw stack
- Bugcheck description (including *ln exception address* for corrupt or truncated dumps)
- Bugcheck callback data (*!bugdump* for systems prior to Windows XP SP1)
- Bugcheck secondary callback data (*.enumtag*)
- Computer name (*dS srv!srvcomputername*)

CMDTREE.TXT

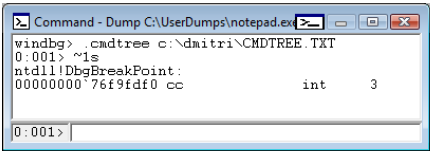
Corresponding version of cmdtree.txt for **.cmdtree** WinDbg command can be found on windbg.org or here: http://www.dumpanalysis.org/blog/files/CMDTREE.TXT

Here is the screenshot of its execution:



Crash Dump Analysis Checklist (www.dumpar

```
Crash Dump Analysis Checklist (www.dumpanalys
  General
    Versions and locations
    Set longer stack trace
  Application crash or hang
    Default analysis (crash)
    Default analysis (hang)
    Critical sections (locked)
    Modules
    Threads (all)
    Threads (unique)
    Gflags
    Time consumed by thread
    PEB
    TEB
    Hooked functions (ntdll)
    Hooked functions (kernel32)
    Hooked functions (user32)
    Exception handlers
    Computer name
    Stack of exception thread
    Stack of current thread
    Switch to thread
      #0
      #1
      #2
      #3
      #4
      #5
      #6
      #7
      #8
      #9
  System hang
    Default analysis
    ERESOURCE contention
    Processes and virtual memory
    Sorted pool consumption (paged)
    Sorted pool consumption (nonpaged)
    Waiting threads
    Critical system queues
    I/O
    The list of all thread stack traces
    Critical sections for current process
    Sessions
    Processes
    Running threads
    Ready threads
    DPC queues
    The list of APCs
    Internal queued spinlocks
    Computer name
    Switch to processor
  BSOD
```



Command - Dump C:\UserDumps\notepad.exe

```
windbg> .cmdtree c:\dmitri\CMDTREE.TXT
0:001> ~1s
ntdll!DbgBreakPoint:
00000000`76f9fdf0 cc              int     3
```

```
0:001>
```

INDEX

NOTES

Front cover image is the picture of my personal book library and the back cover image is visualized virtual process memory generated from a memory dump of colometric computer memory dating sample using **Dump2Picture**.