Development Lifecycle » Debug Tips » General **Advanced**

ASM, VC6Win2K, WinXP, Win2003, Dev, QA

Debug Tutorial Part 6: Navigating The Kernel Debugger

By Toby Opferman

Posted: **7 Aug 2004**
Views: **76,831**
Bookmarked: **102 times**

Learn the basics of the kernel debugger.

25 votes for this article.
Popularity: 6.54 Rating: **4.68** out of 5 1 2 3 4 5

Introduction

In this tutorial, we will be covering a few of the basic features of the kernel debugger, and get used to using it. I obviously can't cover everything, so only select topics will be covered to just get used to the debugger. Hopefully, you will find this article useful in your debugging adventures.

Setup

To setup the Kernel Debugger, you will need to modify your *BOOT.INI* file as mentioned in "Debug Tutorial Part 1". /DEBUG /DEBUGPORT=COM1 /BAUDRATE=115200 should be entered as options. If you do not specify the speed, the default I believe is 19200. I would create a separate entry in the *boot.ini* so you can select between debugging and not debugging. You can do this by just copying what is already there and modifying it.

```
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
                                                                   /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
    /fastdetect /debug /debugport=com1 /baudrate=115200
(Shown on seperate line to prevent long lines in this article.
 Should be on the same line in boot.ini).
```

You can also use Firewire if you have that setup. To connect, connect a NULL Modem cable from the specified COM port of the machine you want to debug, to the COM port of the machine you want to use to debug.

There are also other kernel debuggers out there that run on the same machine, like Softice. There's even a "LiveKD" that can be used. This tutorial will only refer to the free tools that can be downloaded from Microsoft for Windows NT. There are other versions of the tools for Windows 9x.

To connect to the machine, simply open WINDBG and select "Kernel Debug" from the File menu. Select "COM" tab, or other tab if you used Firewire, and then enter the options. The COM port you specify is where you connect the cable to the local machine, not the one you specified in the remote machine's *BOOT.INI*. If you're using VMWare, when you setup the COM port, you can select it to output to a Pipe. This URL shows the setup for VMWare.

You may need to press "Control + C" if the machine has already booted in order for you to enter the debugger.

Some other tips; if you are having trouble connecting, you may want to try some of the following:

1. Control + Break or "Break" in Debug menu.
2. Resynchronize (Debug -> Kernel Connection).
3. Cycle Baud Rates (Debug -> Kernel Connection).
4. Try a different COM Port, ensure the cable is on correct ports.

5. Close the debugger and try again by opening another debugger.
6. Ensure *boot.ini* is correct. When booting, should see "DEBUGGER ENABLED" by the name.

Ready To Go!

Now that we are connected, we are ready to go. First, ensure you are pointing to the Microsoft Symbol server using !symfix or any method specified in "Debug Tutorial Part 1". Set your symbol path using the environment variable, for example, and adding any other symbol paths you may need such as for your own software.

First, let's get started with something simple. Let's find all processes on the system. This is done by breaking into the kernel debugger and using the "!process 0 0". Remember, if you get an error, make sure you are pointing to Microsoft's symbol server. The debugger needs to locate certain variables in memory to perform different actions.

You may see something like this for each process:

```
PROCESS fcdbe2c0  SessionId: 0  Cid: 00a4    Peb: 7ffdf000  ParentCid: 008c
    DirBase: 0401d000  ObjectTable: fcdc39c8  TableSize: 354.
    Image: winlogon.exe
```

So, what does it all mean?

PROCESS

The first is the address of the `EPROCESS` structure. This is a kernel data structure that describes that process. The address for this process is "fcdbe2c0". If you have symbols, you can use the "dt" command which was mentioned in an earlier tutorial, to view this structure.

```
dt _EPROCESS fcdbe2c0
```

or

```
dt nt!_EPROCESS fcdbe2c0
```

If this does not work and you are debugging a Windows 2000 machine, you may want to use "!processfields" instead. It will show you the offsets into fcdbe2c0, and then you can dump the address manually to see the data you want.

SESSION ID

The next is the "Session ID". In a multi-user environment (Terminal Services), different users who log into a machine get their own "session". Normally, Session 0 is the console, although this is a bit different in Windows XP with the fast user switching feature. For kicks, log into Windows XP with a few users (unless you have 2000 or 2003 server edition, you can try the same thing) and type "qwinsta". It should give you output of the user and their session ID.

CID

The "Cid" is the Process Identifier. If you look in Task Manager and select "PID" to display in the columns, this is all it is. It's the PID of this process.

PEB

PEB is the Process Environment Block. This is a user mode data structure, and what does that mean? Generally, every process you will see the same address for this location. I hope that anyone wanting to use the kernel debugger understands Virtual Addresses. In the kernel, the system address space is mapped for all processes (without going into the details of "session space"), but the user mode addresses are mapped for that process. So, obviously, the same address in one process is not the same physical address as another. So, if all the addresses are the same, how would we view the PEB for a particular process? We need to change our "context" to that process. That way, the correct page directory tables are loaded, and we can correctly translate to the data in that

process. We will learn how to set the context later. When you do though, just as in the local debugger, you can view the PEB in the kernel debugger.

```
dt _PEB 7ffdf000
```

or

```
!peb.
```

PARENT CID

The "Parent Cid" is exactly what you think it is. It's the process that created this process, the parent process.

DIRBASE

The "DirBase" is the directory base. This is the value of CR3 when this process is running. This is the address that starts the translation of Virtual Addresses to Physical Addresses. In fact, there is a command that can do this for you. !ptov. So, the Directory Base is 0401d000, you must take off the last 3 digits (which should always be 0).

```
!ptov 0401d
```

```
kd> !ptov 0401d
4195000 10000
40d6000 20000
3a5d000 6d000
40f9000 6e000
4117000 6f000
405a000 70000
409b000 71000
409c000 72000
412f000 73000
413a000 74000
42d1000 76000
426f000 77000
4416000 78000
4458000 7a000
439d000 7b000
43b6000 7c000
4880000 7d000
4831000 7e000
4ac3000 7f000
4b66000 80000
4a67000 81000
4f42000 84000
50a7000 89000
4f81000 8d000
523f000 94000
5240000 a8000
524f000 ac000
5226000 b3000
52c4000 bc000
4fda000 d6000
52af000 dc000
5403000 e2000
51cc000 e5000
509f000 ec000
5325000 ee000
52aa000 f0000
512c000 f2000
522e000 f3000
52f1000 f4000
5174000 f6000
5249000 f7000
575e000 f8000
56df000 f9000
5721000 fb000
58e3000 fd000
576c000 fe000
57ad000 ff000
5833000 105000
583c000 10e000
...
```

The address on the left is the Physical Address. The address on the right is the Virtual Address used by the application. To view a physical address, you do not need to switch contexts. You use the "!dc", "!db", etc. commands instead of the "dc", "db" commands which work on Virtual Address. You can also change the data stored at a physical address by using the "!eb", etc. commands to write data.

```
kd> !dc 5833000
# 5833000 72656d6d 6c616963 666f5320 72617774 mmercial Softwar
# 5833010 75502065 73696c62 73726568 30414320 e Publishers CA0
# 5833020 0d309f81 862a0906 0df78648 05010101 ..0...*.H.......
# 5833030 8d810300 89813000 00818102 6569d3c3 .....0........ie
# 5833040 54940152 62c628ab 5554b318 458744c5 R..T.(.b..TU.D.E
# 5833050 7ec23b4a c8d7d3d8 d88d8680 9c16f10c J;.~............
# 5833060 29a96bcc 73768fb2 62c5c892 1eed3ca6 .k.)..vs...b.<..
# 5833070 13f07505 4d146c00 079098d4 817369be .u...l.M.....is.
```

OBJECT TABLE

This is the table of the handle mappings for that process. This is a data structure that contains the size and a pointer to the object to handle list. This is how the kernel converts a usermode handle

into its matching kernel object. We will look at handles again later.

TABLESIZE

This is the size of the table in base 10. Yes, this is a bit non-standard now since we have some numbers in hex like the PID, but the size of the table is in decimal. This shows how many entries this process has in its handle table. If you open Task Manager and view "Handle Count", it should match this number. This is how many handle entries are being used in this process.

IMAGE

This is a simple one. The image of this process, in this case it's *WINLOGON.EXE*.

Changing Contexts

There are different ways to change contexts. You can do *.context <DirBase>*, you can use *.trap <trap address>*, but here I will show you how to switch to a thread.

First, let's dump WinLogon's threads.

```
!process fcdbe2c0  ff
```

The `EPROCESS` structure must be supplied to the kernel debugger along with some flags. Now, *ff* is to show everything.

```
kd> !process fcdbe2c0  ff
PROCESS fcdbe2c0  SessionId: 0  Cid: 00a4    Peb: 7ffdf000  ParentCid: 008c
    DirBase: 0401d000  ObjectTable: fcdc39c8  TableSize: 354.
    Image: winlogon.exe
    VadRoot fcda2ce8 Clone 0 Private 798. Modified 1085. Locked 0.
    DeviceMap fcebfa48
    Token                             e1b762d0
    ElapsedTime                       21:07:55.0882
    UserTime                          0:00:01.0241
    KernelTime                        0:00:03.0805
    QuotaPoolUsage[PagedPool]         36408
    QuotaPoolUsage[NonPagedPool]      62184
    Working Set Sizes (now,min,max)  (652, 50, 345) (2608KB, 200KB, 1380KB)
    PeakWorkingSetSize                2034
    VirtualSize                       34 Mb
    PeakVirtualSize                   36 Mb
    PageFaultCount                    4919
    MemoryPriority                    BACKGROUND
    BasePriority                      13
    CommitCharge                      1357

        THREAD fcdbd020  Cid a4.84  Teb: 7ffde000  Win32Thread: e1b7b8e8
WAIT: (WrUserRequest) UserMode Non-Alertable
            fcdb9820  SynchronizationEvent
        Not impersonating
        Owning Process fcdbe2c0
        Wait Start TickCount    49551906      Elapsed Ticks: 13726
        Context Switch Count    3360                  LargeStack
        UserTime                  0:00:00.0290
        KernelTime                0:00:01.0041
        Start Address 0x01001674
        Stack Init f7660000 Current f765fcc8 Base f7660000 Limit f765d000 Call 0
        Priority 15 BasePriority 15 PriorityDecrement 0 DecrementCount 0
Kernel stack not resident.

        ChildEBP RetAddr  Args to Child
        f765fce0 8042d61c 00000000 e1b7b8e8 00000001 nt!KiSwapThread+0xc5
        f765fd08 a00159cb fcdb9820 0000000d 00000001 nt!KeWaitForSingleObject+0x1a1
        f765fd44 a0014f6c 000020ff 00000000 00000001 win32k!xxxSleepThread+0x183
        f765fd54 a0014f53 0006fde8 80461691 00000000 win32k!xxxWaitMessage+0xe
        f765fd5c 80461691 00000000 00000000 00000018 win32k!NtUserWaitMessage+0xb
        f765fd5c 77e14b53 00000000 00000000 00000018 nt!KiSystemService+0xc4
        0006fe14 77e23570 00070022 00000000 00000010 +0x77e14b53
        0006fe38 77e2381e 01000000 01024c10 00000000 +0x77e23570
        0006fe58 77e3dcf8 01000000 01024c10 00000000 +0x77e2381e
        0006fe7c 01006052 01000000 00000578 00000000 +0x77e3dcf8
        0006feb8 01005fa8 000766b8 01000000 00000578 +0x1006052
        0006fef0 010099c7 000766b8 01000000 00000578 +0x1005fa8
        0006ff28 010019e4 000766b8 00000005 0007366c +0x10099c7
        0006ff58 0100179e 01000000 00000000 0007366c +0x10019e4
        0006fff4 00000000 7ffdf000 000000c8 00000100 +0x100179e

        THREAD fcdb8b80  Cid a4.c0  Teb: 7ffdd000  Win32Thread: 00000000
WAIT: (DelayExecution) UserMode Alertable
            fcdb8c68  NotificationTimer
        Not impersonating
        Owning Process fcdbe2c0
        Wait Start TickCount    49564648      Elapsed Ticks: 984
        Context Switch Count    33085
        UserTime                  0:00:00.0000
        KernelTime                0:00:00.0000
        Start Address 0x77e92c50
        Win32 Start Address 0x77f88164
        Stack Init f78f4000 Current f78f3cc4 Base f78f4000 Limit f78f1000 Call 0
        Priority 13 BasePriority 13 PriorityDecrement 0 DecrementCount 0
Kernel stack not resident.

        ChildEBP RetAddr  Args to Child
        f78f3cdc 8042d00e f78f3d64 003dffac 003dffac nt!KiSwapThread+0xc5
        f78f3d04 804c04e6 003dfc01 00000001 f78f3d34 nt!KeDelayExecutionThread+0x180
        f78f3d54 80461691 00000001 003dffac 00000000 nt!NtDelayExecution+0x7f
        f78f3d54 77f90333 00000001 003dffac 00000000 nt!KiSystemService+0xc4
        003dffb4 77e92ca8 0006feb8 00000000 00000000 +0x77f90333
        003dffec 00000000 77f88164 0006feb8 00000000 +0x77e92ca8

... (Continues on with more information)
```

Don't get worried by the "Kernel stack not resident". We know that memory gets swapped to disk, right? So, not all memory will always be available when using the kernel debugger. We just have to bear with this. In the above case, the stack is in memory, at least enough to get a stack trace, which isn't always true. The above just means that part of the stack isn't in memory, in this case.

So, let's set our context to the first thread.

```
kd> .process fcdbe2c0
Implicit process is now fcdbe2c0
WARNING: .cache forcedecodeuser is not enabled
kd> .cache forcedecodeuser

Max cache size is       : 1024000 bytes (0x3e8 KB)
Total memory in cache   : 0 bytes (0 KB)
Number of regions cached: 0
0 full reads broken into 0 partial reads
    counts: 0 cached/0 uncached, 0.00% cached
    bytes : 0 cached/0 uncached, 0.00% cached
** Transition PTEs are implicitly decoded
** User virtual addresses are translated to physical addresses before access
kd> .thread fcdbd020
Implicit thread is now fcdbd020
kd> kb
  *** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr  Args to Child
f765fce0 8042d61c 00000000 e1b7b8e8 00000001 nt!KiSwapThread+0xc5
f765fd08 a00159cb fcdb9820 0000000d 00000001 nt!KeWaitForSingleObject+0x1a1
f765fd44 a0014f6c 000020ff 00000000 00000001 win32k!xxxSleepThread+0x183
f765fd54 a0014f53 0006fde8 80461691 00000000 win32k!xxxWaitMessage+0xe
f765fd5c 80461691 00000000 00000000 00000018 win32k!NtUserWaitMessage+0xb
f765fd5c 77e14b53 00000000 00000000 00000018 nt!KiSystemService+0xc4
WARNING: Frame IP not in any known module. Following frames may be wrong.
0006fe14 77e23570 00070022 00000000 00000010 0x77e14b53
0006fe38 77e2381e 01000000 01024c10 00000000 0x77e23570
0006fe58 77e3dcf8 01000000 01024c10 00000000 0x77e2381e
0006fe7c 01006052 01000000 00000578 00000000 0x77e3dcf8
0006feb8 01005fa8 000766b8 01000000 00000578 0x1006052
0006fef0 010099c7 000766b8 01000000 00000578 0x1005fa8
0006ff28 010019e4 000766b8 00000005 0007366c 0x10099c7
0006ff58 0100179e 01000000 00000000 0007366c 0x10019e4
0006fff4 00000000 7ffdf000 000000c8 00000100 0x100179e
kd> !reload
Connected to Windows 2000 2195 x86 compatible target, ptr64 FALSE
Loading Kernel Symbols
...................................................................................
Loading unloaded module list
No unloaded module list present
Loading User Symbols
..................................................
kd> kb
  *** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr  Args to Child
f765fce0 8042d61c 00000000 e1b7b8e8 00000001 nt!KiSwapThread+0xc5
f765fd08 a00159cb fcdb9820 0000000d 00000001 nt!KeWaitForSingleObject+0x1a1
f765fd44 a0014f6c 000020ff 00000000 00000001 win32k!xxxSleepThread+0x183
f765fd54 a0014f53 0006fde8 80461691 00000000 win32k!xxxWaitMessage+0xe
f765fd5c 80461691 00000000 00000000 00000018 win32k!NtUserWaitMessage+0xb
f765fd5c 77e14b53 00000000 00000000 00000018 nt!KiSystemService+0xc4
0006fde0 77e23680 00000000 00000000 0000ffff USER32!NtUserWaitMessage+0xb
0006fe14 77e23570 00070022 00000000 00000010 USER32!DialogBox2+0x216
0006fe38 77e2381e 01000000 01024c10 00000000 USER32!InternalDialogBox+0xd1
0006fe58 77e3dcf8 01000000 01024c10 00000000 USER32!DialogBoxIndirectParamAorW+0x34
0006fe7c 01006052 01000000 00000578 00000000 USER32!DialogBoxParamW+0x3d
0006feb8 01005fa8 000766b8 01000000 00000578 winlogon!TimeoutDialogBoxParam+0x27
0006fef0 010099c7 000766b8 01000000 00000578 winlogon!WlxDialogBoxParam+0x7b
0006ff10 01004bdc 000766b8 00071fc8 000766b8 winlogon!BlockWaitForUserAction+0x3c
0006ff28 010019e4 000766b8 00000005 0007366c winlogon!MainLoop+0x1bf
0006ff58 0100179e 01000000 00000000 0007366c winlogon!WinMain+0x2a5
0006fff4 00000000 7ffdf000 000000c8 00000100 winlogon!WinMainCRTStartup+0x156
```

Now, you notice once I set my context, I had to reload symbols in order to view usermode symbols? That's because we need to force the kernel debugger to read the new context and load the new symbols for this process. If you switch processes multiple times, you may have noticed the kernel

debugger will tell you the symbols for the previous process. This is why you should try to reload symbols once you switch contexts.

You should also note that the kernel stack is different from the usermode stack. When a thread calls into the kernel, it switches the stack. The kernel has its own stack, and usermode has its own stack. This is just an FYI.

So, now you may be asking what do all the parameters mean in the PROCESS and THREAD listings? The top header we already covered, so let's start with the Process header. Though, most of it will usually just be useless information to you.

```
VadRoot fcda2ce8 Clone 0 Private 798. Modified 1085. Locked 0.
```

The "VadRoot" is basically a pointer to a binary tree that contains the address of all Virtual Address in the process, including their start and end ranges. This is called a "Virtual Address Descriptor". If you walk this tree, you would end up seeing the same output you would if you did an x *! in the user mode debugger, but actually you may see more. This is because, this doesn't just include modules, but all mapped virtual address and their ranges. For example, if you did Virtual Alloc for some large blocks of memory, these would show up in this tree list.

Try !vad fcda2ce8

(Notice that you should obviously put the address that's listed in your kernel debugger there.)

```
   DeviceMap fcebfa48
```

This is the Object Device Map for the system. While each process gets its own pointer to this object, as you can see below, it is stored globally in the kernel.

```
kd> x nt!ObSystemDeviceMap
8047f79c nt!ObSystemDeviceMap = <NO type information>
kd> dd nt!ObSystemDeviceMap L 1
8047f79c  fcebfa48
```

This article will not go into displaying device stacks or anything like that. This article is just an introduction to the basics.

```
    Token                             e1b762d0
```

The user's token for this process. Try !token e1b762d0:

```
    ElapsedTime                       21:07:55.0882
    UserTime                          0:00:01.0241
    KernelTime                        0:00:03.0805
```

The amount of time spent in the kernel and in usermode.

```
    QuotaPoolUsage[PagedPool]         36408
    QuotaPoolUsage[NonPagedPool]      62184
    Working Set Sizes (now,min,max)  (652, 50, 345) (2608KB, 200KB, 1380KB)
    PeakWorkingSetSize                2034
    VirtualSize                       34 Mb
    PeakVirtualSize                   36 Mb
    PageFaultCount                    4919
    MemoryPriority                    BACKGROUND
    BasePriority                      13
    CommitCharge                      1357
```

These are pretty self explanatory as well. They simply display memory usage and statistics.

Now, let's take a look at the thread information. We can do this by using !thread <thread>.

```
kd> !thread fcdbd020
THREAD fcdbd020  Cid a4.84  Teb: 7ffde000  Win32Thread: e1b7b8e8 WAIT:
(WrUserRequest) UserMode Non-Alertable
    fcdb9820  SynchronizationEvent
Not impersonating
Owning Process fcdbe2c0
Wait Start TickCount    49551906      Elapsed Ticks: 14128
Context Switch Count    3360                   LargeStack
UserTime                  0:00:00.0290
KernelTime                0:00:01.0041
Start Address winlogon!WinMainCRTStartup (0x01001674)
Stack Init f7660000 Current f765fcc8 Base f7660000 Limit f765d000 Call 0
Priority 15 BasePriority 15 PriorityDecrement 0 DecrementCount 0
Kernel stack not resident.

ChildEBP RetAddr  Args to Child
f765fce0 8042d61c 00000000 e1b7b8e8 00000001 nt!KiSwapThread+0xc5
f765fd08 a00159cb fcdb9820 0000000d 00000001 nt!KeWaitForSingleObject+0x1a1
f765fd44 a0014f6c 000020ff 00000000 00000001 win32k!xxxSleepThread+0x183
f765fd54 a0014f53 0006fde8 80461691 00000000 win32k!xxxWaitMessage+0xe
f765fd5c 80461691 00000000 00000000 00000018 win32k!NtUserWaitMessage+0xb
f765fd5c 77e14b53 00000000 00000000 00000018 nt!KiSystemService+0xc4
0006fde0 77e23680 00000000 00000000 0000ffff USER32!NtUserWaitMessage+0xb
0006fe14 77e23570 00070022 00000000 00000010 USER32!DialogBox2+0x216
0006fe38 77e2381e 01000000 01024c10 00000000 USER32!InternalDialogBox+0xd1
0006fe58 77e3dcf8 01000000 01024c10 00000000 USER32!DialogBoxIndirectParamAorW+0x34
0006fe7c 01006052 01000000 00000578 00000000 USER32!DialogBoxParamW+0x3d
0006feb8 01005fa8 000766b8 01000000 00000578 winlogon!TimeoutDialogBoxParam+0x27
0006fef0 010099c7 000766b8 01000000 00000578 winlogon!WlxDialogBoxParam+0x7b
0006ff10 01004bdc 000766b8 00071fc8 000766b8 winlogon!BlockWaitForUserAction+0x3c
0006ff28 010019e4 000766b8 00000005 0007366c winlogon!MainLoop+0x1bf
0006ff58 0100179e 01000000 00000000 0007366c winlogon!WinMain+0x2a5
0006fff4 00000000 7ffdf000 000000c8 00000100 winlogon!WinMainCRTStartup+0x156
```

Now, let's take a look at each piece of header information. We remember from before that the process address was actually an `EPROCESS` structure. Well, this is an `ETHREAD` structure. This can be displayed by using dt nt!_ETHREAD <address> or !threadfields.

So, let's break it down.

```
THREAD fcdbd020  Cid a4.84  Teb: 7ffde000
       Win32Thread: e1b7b8e8 WAIT: (WrUserRequest)
UserMode Non-Alertable
    fcdb9820  SynchronizationEvent
```

The first address is the address of the `ETHREAD` structure as we mentioned. The second is the Process.Thread ID. The process ID is A4h, the thread ID is 84h. Thread IDs are useful when looking at deadlocks since the synchronization objects are owned by threads. It's also good when looking for window information as each window is owned by a particular thread. These thread IDs are available in the user mode debugger.

The TEB. The Thread Environment Block is as mentioned in a previous tutorial. This has thread specific information as well as it points to the PEB, so threads can easily access certain information. The stack limit and end is here as well. !teb or dt _TEB <address>. You obviously need to be in the context of the process and thread to use !teb. If the TEB is `NULL`, this means it's a system thread. The thread is only executing in the kernel, it's not a user mode thread.

The next is the "Win32Thread". What's this? This is a per-thread data structure that *WIN32k.SYS* maintains.

The last part is the state of the thread. This thread is in a wait state. It's waiting in a non-alertable state in usermode. For more information on alertable, non-alertable, kernel verse. usermode waits, please refer to MSDN's articles about KeWaitForMultipleObjects or information about APC (Asynchronous Procedure Calls).

```
Not impersonating
```

Whether this thread is impersonating or not. If you are familar with RPC, the server can

impersonate the caller to perform operations on the caller's behalf or get information about the caller. If this thread was impersonating, you would see an impersonation token here which you could do *!token <address>* on to get the information. Certain calls into the kernel will look at this impersonation token if it is there and use it instead of the process' user token.

```
Owning Process fcdbe2c0
```

The own process' `EPROCESS` address. In this case, it's Winlogon. You may see weird things sometimes where a thread seems to have a different owner than it's being displayed in. The kernel may call `KeAttachProcess()` to perform actions in a different context.

```
Wait Start TickCount    49551906      Elapsed Ticks: 14128
Context Switch Count    3360                   LargeStack
UserTime                  0:00:00.0290
KernelTime                0:00:01.0041
```

Time this thread spent in usermode and kernel mode. If you are looking for a process hog, these are the values you look at to find the process, then the thread. "Elapsed Ticks" is how long it's been since this thread was executed. If you're looking for a hog right after it happened, this is another field to look at, what threads have the least elapsed ticks.

```
Start Address winlogon!WinMainCRTStartup (0x01001674)
```

This is the starting address of the thread.

```
Stack Init f7660000 Current f765fcc8 Base f7660000 Limit f765d000 Call 0
```

This refers to the kernel stack, not the user mode stack which can be found in the TEB. The current position of the kernel mode stack and its limits, and where it started.

```
Priority 15 BasePriority 15 PriorityDecrement 0 DecrementCount 0
```

The priority of this thread.

You may see other things, but this is a basic listing. If you notice an LPC, you can do things like "!lpc message <message id>" to get the originator of the call if they are still around, for example.

Now that we are in the context, we can do things like set break points, change memory, etc. Just remember that changing memory in a driver will change this memory for everyone. Setting breakpoints is the same way, you set the breakpoint for everyone on the system. This is a kernel debugger and the kernel mode drivers are global to the system (or the session, for other certain drivers we will not cover here).

What else can I do?

There're a lot of things that you can do, but like I mentioned before, we will just be covering some of the basics. It also depends on the area that you work in. For example, someone who is developing hardware drivers may never use or even look at anything in usermode or even that high up the driver stack. They may only know some of the hardware commands. Someone who only works with usermode applications or interaction may only use the kernel debugger for some extra information, or figure out who's calling with an RPC, etc.

Page Table Entries

The only problem with using the kernel debugger or kernel dump is that some information may not be there, it may be paged out. So, how can we tell if memory is valid or paged out?

In the x86 processor, there are levels of indirection when it comes to translating a virtual address to a physical address. There is a Page Directory Entry that points to a Page Table Entry. If you want to find the Page Table Entry, this is how you can do it. Remember, we already know how to find physical addresses for virtual addresses using the page directory of the process.

!pte is the answer to find Page Table Entries.

```
kd> !pte 80461691
80461691  - PDE at C0300804        PTE at C0201184
          contains 00034163      contains 00461121
        pfn 34 G-DA--KWV    pfn 461 G--A--KRV
kd> dc C0300804  L1
c0300804  00034163
kd> dc C0201184 L1
c0201184  00461121
```

Just to show, if you dc those addresses, that's what you get. The Virtual Address is first added to the GDT/LDT and then it's translated. Part of the Virtual Address is taken as an offset to get the Page Directory Entry. Then next part of the Address is taken to get the Page Entry Table. The last 3 nibbles of the virtual address are actually the last 3 nibbles of where the PTE points to. So, the PTE says 461xxx is the physical address. Let's find out and dump the virtual and physical addresses:

```
kd> !dc 461691
#  461690 8be58bd3 dff1240d 3c558bff 01289189 .....$....U<..(.
#  4616a0 f7fa0000 00007045 06750002 016c45f6 ....Ep....u..El.
#  4616b0 1d8b5874 ffdff124 002e43c6 004a7b80 tX..$....C...{J.
#  4616c0 dd8b4874 c7444389 003b5043 43c70000 tH...CD.CP;....C
#  4616d0 00002338 3443c700 00000023 003043c7 8#....C4#....C0.
#  4616e0 b9000000 00000001 059815ff fb508040 ............@.P.
#  4616f0 6a006a53 f132e801 ff59fffc 40059c15 Sj.j..2...Y....@
#  461700 44438b80 90abebfa 548bff8b 8b644c24 ..CD.......T$Ld.
kd> dc 80461691
80461691  0d8be58b ffdff124 893c558b 00012891  ....$....U<..(..
804616a1  45f7fa00 02000070 f6067500 74016c45  ...Ep....u..El.t
804616b1  241d8b58 c6ffdff1 80002e43 74004a7b  X..$....C...{J.t
804616c1  89dd8b48 43c74443 00003b50 3843c700  H...CD.CP;....C8
804616d1  00000023 233443c7 c7000000 00003043  #....C4#....C0..
804616e1  01b90000 ff000000 40059815 53fb5080  ...........@.P.S
804616f1  016a006a fcf132e8 15ff59ff 8040059c  j.j..2...Y....@.
80461701  fa44438b 8b90abeb 24548bff 1d8b644c  .CD.......T$Ld..
```

As you can see, it's true. The Physical Address dump is a little different as it dumped it at the start of the memory rather than 1 so it was aligned, but you can see it's the same information. If we had dumped them "dc 80461690", you would see they are the same.

You can also go backwards, take a PTE address, and find the virtual address it belongs to.

```
kd> !pte C0201184
80461000  - PDE at C0300804        PTE at C0201184
          contains 00034163      contains 00461121
        pfn 34 G-DA--KWV    pfn 461 G--A--KRV
```

What happens on invalid addresses?

```
kd> !pte 40
00000040  - PDE at C0300000        PTE at C0000000
          contains 01800067      contains 00000000
        pfn 1800 --DA--UWV       not valid
kd> dc C0300000  L1
c0300000  01800067
kd> dc C0000000 L1
c0000000  00000000
kd> !pte 66740020
66740020  - PDE at C0300664        PTE at C0199D00
          contains 00000000        unavailable
kd> dc C0300664 L1
c0300664  00000000
kd> dc C0199D00 L1
c0199d00  00a8d4c0
```

"Not Valid" means it's not a valid virtual address. (Some addresses may be "not valid" just because they have never been used yet, so they're not mapped I have found, at least that's my theory! For example, stack memory is mapped into your process but it's never been dirty, so why put it in the

page file?) "Unavailable" means it's not available, but it is valid or it believes it to be. Otherwise, the address is valid. Sometimes, if an address is "unavailable", it will list a pagefile offset, and sometimes it will not. Executables and binaries are done using memory mapped files. You can see this by doing !memusage. That means that these, unless modified, would not be in the pagefile, and thus there is no pagefile offset.

There is also a !sysptes that can dump all system ptes.

Handle Information

Displaying handle information in the kernel is easy. Once you set your context to the correct process, just !handle as you do in user mode, with an exception. You need to specify the process to dump the handles.

```
kd> !handle 0 0 fcdbe2c0
processor number 0
PROCESS fcdbe2c0  SessionId: 0  Cid: 00a4    Peb: 7ffdf000  ParentCid: 008c
    DirBase: 0401d000  ObjectTable: fcdc39c8  TableSize: 354.
    Image: winlogon.exe

Handle Table at e1b78000 with 354 Entries in use
0004: Object: e1267030  GrantedAccess: 000f001f

0008: Object: fcdee160  GrantedAccess: 00100003

000c: Object: fcdee120  GrantedAccess: 00100003

0010: Object: fcdbe280  GrantedAccess: 00100003
```

The first is the handle # used in usermode. The second is the "Object" or kernel address of the object, and the last is the access granted to the object. You may also specify flags to dump more information as seen below:

```
kd> !handle 0 ff fcdbe2c0
processor number 0
PROCESS fcdbe2c0  SessionId: 0  Cid: 00a4    Peb: 7ffdf000  ParentCid: 008c
    DirBase: 0401d000  ObjectTable: fcdc39c8  TableSize: 354.
    Image: winlogon.exe

Handle Table at e1b78000 with 354 Entries in use
0000: free handle
0004: Object: e1267030  GrantedAccess: 000f001f
Object: e1267030  Type: (fcebc580) Section
    ObjectHeader: e1267018
        HandleCount: 1  PointerCount: 1

0008: Object: fcdee160  GrantedAccess: 00100003
Object: fcdee160  Type: (fcebf420) Event
    ObjectHeader: fcdee148
        HandleCount: 1  PointerCount: 1

000c: Object: fcdee120  GrantedAccess: 00100003
Object: fcdee120  Type: (fcebf420) Event
    ObjectHeader: fcdee108
        HandleCount: 1  PointerCount: 1

0010: Object: fcdbe280  GrantedAccess: 00100003
Object: fcdbe280  Type: (fcebf420) Event
    ObjectHeader: fcdbe268
        HandleCount: 1  PointerCount: 1

0014: Object: fcec6c50  GrantedAccess: 00000003
Object: fcec6c50  Type: (fcee4b40) Directory
    ObjectHeader: fcec6c38
        HandleCount: 15  PointerCount: 44
        Directory Object: fcebfab0  Name: KnownDlls
        HashBucket[ 00 ]: e137ca00 Section 'gdi32.dll'
                          e134b900 Section 'imagehlp.dll'
                          e13154a0 Section 'url.dll'
        HashBucket[ 02 ]: e13dee00 Section 'MPR.dll'
        HashBucket[ 03 ]: e135b040 Section 'ole32.dll'
                          e135d2a0 Section 'urlmon.dll'
        HashBucket[ 04 ]: e1373600 Section 'lz32.dll'
                          e138bde0 Section 'olesvr32.dll'
        HashBucket[ 06 ]: e137ef40 Section 'shell32.dll'
                          e12c7880 Section 'wldap32.dll'
        HashBucket[ 09 ]: e13126e0 Section 'user32.dll'
                          e13118a0 Section 'version.dll'
        HashBucket[ 10 ]: e135af80 Section 'olecli32.dll'
        HashBucket[ 14 ]: e13713e0 Section 'MSASN1.DLL'
        HashBucket[ 16 ]: e1314c60 Section 'COMCTL32.DLL'
                          fcde6c50 SymbolicLink 'KnownDllPath'
        HashBucket[ 17 ]: e13171c0 Section 'CRYPT32.dll'
        HashBucket[ 18 ]: e1316e60 Section 'advapi32.dll'
                          e13583a0 Section 'oleaut32.dll'
        HashBucket[ 19 ]: e135c980 Section 'SHLWAPI.DLL'
                          e12c2fc0 Section 'wow32.dll'
                          e1316460 Section 'olecnv32.dll'
        HashBucket[ 23 ]: e131a520 Section 'comdlg32.dll'
        HashBucket[ 26 ]: e1317b80 Section 'wininet.dll'
        HashBucket[ 27 ]: e12d5fc0 Section 'olethk32.dll'
        HashBucket[ 28 ]: e124c800 Section 'MSVCRT.DLL'
        HashBucket[ 31 ]: e134dbe0 Section 'rpcrt4.dll'
        HashBucket[ 32 ]: e130b460 Section 'kernel32.dll'

0018: Object: fccc3c88  GrantedAccess: 00100020 (Inherit)
Object: fccc3c88  Type: (fced7c40) File
    ObjectHeader: fccc3c70
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \WINNT\system32 {HarddiskVolume1}

001c: Object: fcdfb730  GrantedAccess: 000f000f
Object: fcdfb730  Type: (fcee4b40) Directory
    ObjectHeader: fcdfb718
        HandleCount: 14  PointerCount: 18
        Directory Object: fcebfab0  Name: Windows
        HashBucket[ 04 ]: e1b76d40 Port 'SbApiPort'
        HashBucket[ 09 ]: e1b65a00 Port 'ApiPort'
        HashBucket[ 32 ]: fcdc1750 Directory 'WindowStations'
```

You can also, instead of using 0, specify an actual handle value to just list one handle.

```
kd> !handle 2c ff fcdbe2c0
processor number 0
PROCESS fcdbe2c0  SessionId: 0  Cid: 00a4    Peb: 7ffdf000  ParentCid: 008c
    DirBase: 0401d000  ObjectTable: fcdc39c8  TableSize: 354.
    Image: winlogon.exe

Handle Table at e1b78000 with 354 Entries in use
002c: Object: fcdb99a0  GrantedAccess: 00100003
Object: fcdb99a0  Type: (fcebf420) Event
    ObjectHeader: fcdb9988
        HandleCount: 1  PointerCount: 1
```

There are other methods to dump the object. You notice there's an Object: and an address, !object can also be used on that address. All the kernel debugger really does when finding handle information is look for that ObjectTable address, and find the handle table, then just dump the information since it knows the data structure.

Let's attempt to interpret this table manually. First, the ObjectTable is at fcdc39c8.

```
kd> dc fcdc39c8
fcdc39c8  00000000 00000162 e1b78000 fcdbe2c0  ....b...........
fcdc39d8  000000a4 0000016f 00000300 fcdbe0a8  ....o...........
fcdc39e8  fcdbe5a8 00000000 00000000 00000000  ................
fcdc39f8  00000000 00000000 00000000 00000000  ................
fcdc3a08  00000000 00000000 00000000 00000000  ................
fcdc3a18  00000000 fcdb079c fcdc42bc 00040000  .........B......
fcdc3a28  00000000 fcdc3a2c fcdc3a2c 0030005c  ....,:..,:..\.0.
fcdc3a38  00300030 00000030 07018004 69436d4d  0.0.0.......MmCi
kd> ? 162
Evaluate expression: 354 = 00000162
```

You notice that I dumped a variable that basically specified the number of handles in use. You also notice that the next value is the address that !handle said was the handle table. Now, let's dump this.

```
kd> dc e1b78000
e1b78000  e1b78400 00000000 00000000 00000000  ................
e1b78010  00000000 00000000 00000000 00000000  ................
e1b78020  00000000 00000000 00000000 00000000  ................
e1b78030  00000000 00000000 00000000 00000000  ................
e1b78040  00000000 00000000 00000000 00000000  ................
e1b78050  00000000 00000000 00000000 00000000  ................
e1b78060  00000000 00000000 00000000 00000000  ................
e1b78070  00000000 00000000 00000000 00000000  ................
kd> dc e1b78400
e1b78400  e1b78800 e1d47000 e1d47800 00000000  .....p...x......
e1b78410  00000000 00000000 00000000 00000000  ................
e1b78420  00000000 00000000 00000000 00000000  ................
e1b78430  00000000 00000000 00000000 00000000  ................
e1b78440  00000000 00000000 00000000 00000000  ................
e1b78450  00000000 00000000 00000000 00000000  ................
e1b78460  00000000 00000000 00000000 00000000  ................
e1b78470  00000000 00000000 00000000 00000000  ................
kd> dc e1b78800
e1b78800  00000000 00000001 61267018 000f001f  .........p&a....
e1b78810  7cdee148 00100003 7cdee108 00100003  H..|.......|....
e1b78820  7cdbe268 00100003 7cec6c38 00000003  h..|....8l.|....
e1b78830  7ccc3c72 00100020 7cdfb718 000f000f  r<.| ......|....
e1b78840  7cdb9dc8 00100003 61b7be19 001f0001  ...|.......a....
e1b78850  7ceca3a8 00000001 7cdb9988 00100003  ...|.......|....
e1b78860  613f5ec8 000f003f 6125dcd8 000f001f  .^?a?.....%a....
e1b78870  7cdb9809 001f0003 7cdc3778 0002000f  ...|....x7.|....
```

Now, I dump the address and it looks like there's another address. So, I dump that and looks like there's 3 addresses now, so I dump the first one. What did I find? Well, I found what appears to be the "table". The second DWORD of every pair appears to match the access mask of the objects in order of the dump, if you look above. 000f001f, followed by 00100003, matches exactly. Now, how

do the first numbers relate to the object? If I dump them, they are not valid addresses! There are a few ways we could approach this dilemma. The first would be to debug the kernel debugger and figure out where it's going to translate this. The second would be to make a call into the kernel with one of these objects and figure out what it's doing to translate them. The last would obviously be attempt everything we can think of to translate it back. Well, let's try something. Let's set a "ba r1" on one of those locations and hope Winlogon attempts to use it. We could also write our own application, but I'm a bit lazy. We need to find an object we think Winlogon will attempt to use often though.

I actually think it may be easier to do this with an application like Notepad. Find handles it has open, then set break points on those, and close it. It will need to close those handles!

We have already figured out that if this location is 0, it's a "free" handle.

Now, I've set 4 "BA R1 <address>" in the Notepad object table. We can only set a maximum of 4, so I randomly chose a few handles. I then close Notepad. I end up in a function called "`ExMapHandleToPointer`".

```
eax=e1dfb8f0 ebx=00000000 ecx=e1dfb800 edx=00000000 esi=61caf508 edi=fb6e6bd0
eip=804b3971 esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei pl zr na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
nt!ExLockHandleTableEntry+0xe:
804b3971 85f6             test    esi,esi
kd> p;r

eax=e1dfb8f0 ebx=00000000 ecx=e1dfb800 edx=00000000 esi=61caf508 edi=fb6e6bd0
eip=804b3973 esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000202
nt!ExLockHandleTableEntry+0x10:
804b3973 8975f8           mov     [ebp-0x8],esi     ss:0010:fb6e6b00=e1dfb800
kd>
eax=e1dfb8f0 ebx=00000000 ecx=e1dfb800 edx=00000000 esi=61caf508 edi=fb6e6bd0
eip=804b3976 esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000202
nt!ExLockHandleTableEntry+0x13:
804b3976 0f8449f0ffff     je nt!ExLockHandleTableEntry+0x5c (804b29c5)
kd>
eax=e1dfb8f0 ebx=00000000 ecx=e1dfb800 edx=00000000 esi=61caf508 edi=fb6e6bd0
eip=804b397c esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000202
nt!ExLockHandleTableEntry+0x15:
804b397c 0f8eabdb0000    jle nt!ExLockHandleTableEntry+0x31 (804c152d)
kd>
eax=e1dfb8f0 ebx=00000000 ecx=e1dfb800 edx=00000000 esi=61caf508 edi=fb6e6bd0
eip=804b3982 esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000202
nt!ExLockHandleTableEntry+0x17:
804b3982 8bc6             mov     eax,esi
kd>
eax=61caf508 ebx=00000000 ecx=e1dfb800 edx=00000000 esi=61caf508 edi=fb6e6bd0
eip=804b3984 esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000202
nt!ExLockHandleTableEntry+0x19:
804b3984 0d00000080       or      eax,0x80000000
kd>
eax=e1caf508 ebx=00000000 ecx=e1dfb800 edx=00000000 esi=61caf508 edi=fb6e6bd0
eip=804b3989 esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
nt!ExLockHandleTableEntry+0x1e:
804b3989 8945fc           mov     [ebp-0x4],eax     ss:0010:fb6e6b04=e1dfb800
kd>
eax=e1caf508 ebx=00000000 ecx=e1dfb800 edx=00000000 esi=61caf508 edi=fb6e6bd0
eip=804b398c esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
nt!ExLockHandleTableEntry+0x21:
804b398c 8b45f8           mov     eax,[ebp-0x8]     ss:0010:fb6e6b00=61caf508
kd>
eax=61caf508 ebx=00000000 ecx=e1dfb800 edx=00000000 esi=61caf508 edi=fb6e6bd0
eip=804b398f esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
nt!ExLockHandleTableEntry+0x24:
804b398f 8b4d0c           mov     ecx,[ebp+0xc]     ss:0010:fb6e6b14=e1dfb8f0
kd>
eax=61caf508 ebx=00000000 ecx=e1dfb8f0 edx=00000000 esi=61caf508 edi=fb6e6bd0
eip=804b3992 esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
nt!ExLockHandleTableEntry+0x27:
804b3992 8b55fc           mov     edx,[ebp-0x4]     ss:0010:fb6e6b04=e1caf508
kd>
eax=61caf508 ebx=00000000 ecx=e1dfb8f0 edx=e1caf508 esi=61caf508 edi=fb6e6bd0
eip=804b3995 esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
nt!ExLockHandleTableEntry+0x2a:
804b3995 0fb111           cmpxchg [ecx],edx         ds:0023:e1dfb8f0=61caf508
kd>
Breakpoint 1 hit
eax=61caf508 ebx=00000000 ecx=e1dfb8f0 edx=e1caf508 esi=61caf508 edi=fb6e6bd0
eip=804b3998 esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei pl zr na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
nt!ExLockHandleTableEntry+0x2d:
804b3998 3bc6             cmp     eax,esi
kd>
eax=61caf508 ebx=00000000 ecx=e1dfb8f0 edx=e1caf508 esi=61caf508 edi=fb6e6bd0
eip=804b399a esp=fb6e6af8 ebp=fb6e6b08 iopl=0         nv up ei pl zr na po nc
```

So, what have we accomplished. We notice that we index into the handle table and find that weird number. That weird number is then replaced by the object header "e1caf508". The actual object is at ObjectHeader + 18h, so e1caf508 + 18 = e1caf520.

```
kd> !object e1caf520
Object: e1caf520  Type: (fcebc160) Key
    ObjectHeader: e1caf508
    HandleCount: 1  PointerCount: 1
    Directory Object: 00000000  Name: \REGISTRY\USER
```

Now, what was the calculation to get to this? So simple, "or eax,0x80000000". Let's give it a try now on some of those other objects. We had 7cdee148, and we get fcdee148. Add 18h, and we should be able to !object.

```
kd> !object fcdee160
Object: fcdee160  Type: (fcebf420) Event
    ObjectHeader: fcdee148
    HandleCount: 1  PointerCount: 1
```

So, that's how simple it is.

Heap Information

The heap for a particular process can be displayed from kernel mode. Set your context to that process and dump the PEB. The PEB will point to the number of heaps and a heap list which can be used. Each heap then has a VirtualAlloc list that can be used as a linked list. This can all be done in usermode as well, nothing special here. The only difference is that you may not be able to display all of the information if it's paged out. Anyway, luckily you don't even have to go through all that. You can simply use *!heap* in the kernel! Just set your context and just use *!heap* as normal. You may refer back to Debug Tutorial Part 3 to learn more about heaps.

Pools

Kernel drivers generally use memory from the Paged and Non-Paged pool. You can take an address and do *!pool* to determine its size, tag information, etc. If you develop drivers, you know that when you allocate memory in the kernel, it comes from a global pool, not per-process. So if you corrupt memory, you can end up corrupting other drivers in the system. When a driver allocates memory, they generally specify a "pool tag" which is then associated with the memory. This is how we can tell what driver allocated what memory. There is also *!poolfind* * to find pool based on a pool tag. You can look online or other places to find information about a pool tag and what it relates to.

For example, let's do *!pool* on the objects we were handling before.

```
kd> !pool e1caf520
 e1caf000 size:   60 previous size:    0  (Allocated)  NtFd
 e1caf060 size:   20 previous size:   60  (Free)       ....
 e1caf080 size:   40 previous size:   20  (Allocated)  NtFs
 e1caf0c0 size:   20 previous size:   40  (Free)       Key
 e1caf0e0 size:   20 previous size:   20  (Allocated)  CMVI
 e1caf100 size:   40 previous size:   20  (Allocated)  CMVa
 e1caf140 size:   20 previous size:   40  (Free)       CMVa
 e1caf160 size:   40 previous size:   20  (Lookaside)  Key  (Protected)
 e1caf1a0 size:   20 previous size:   40  (Allocated)  Ntf0
 e1caf1c0 size:   40 previous size:   20  (Allocated)  IoNm
 e1caf200 size:   20 previous size:   40  (Free)       Gla8
 e1caf220 size:   40 previous size:   20  (Allocated)  CMVa
 e1caf260 size:   40 previous size:   40  (Allocated)  CMVa
 e1caf2a0 size:   20 previous size:   40  (Free)       Key
 e1caf2c0 size:   60 previous size:   20  (Allocated)  Ntfo
 e1caf320 size:   40 previous size:   60  (Allocated)  CMNb (Protected)
 e1caf360 size:   40 previous size:   40  (Allocated)  CMIn (Protected)
 e1caf3a0 size:   a0 previous size:   40  (Allocated)  SeSd
 e1caf440 size:   20 previous size:   a0  (Free)       Usqm
 e1caf460 size:   40 previous size:   20  (Allocated)  CMVa
 e1caf4a0 size:   40 previous size:   40  (Allocated)  CMVa
 e1caf4e0 size:   20 previous size:   40  (Allocated)  Ntf0
*e1caf500 size:   40 previous size:   20  (Allocated) *Key  (Protected)
 e1caf540 size:   20 previous size:   40  (Free)       Gla8
 e1caf560 size:   40 previous size:   20  (Allocated)  CMVa
 e1caf5a0 size:   20 previous size:   40  (Free)       SYSA
 e1caf5c0 size:   20 previous size:   20  (Allocated)  CMNb (Protected)
 e1caf5e0 size:   40 previous size:   20  (Allocated)  CMNb (Protected)
 e1caf620 size:   40 previous size:   40  (Allocated)  CMNb (Protected)
 e1caf660 size:   a0 previous size:   40  (Allocated)  CMDa
 e1caf700 size:   40 previous size:   a0  (Allocated)  NtFL
 e1caf740 size:   40 previous size:   40  (Allocated)  CMNb (Protected)
 e1caf780 size:   a0 previous size:   40  (Allocated)  MmSt
 e1caf820 size:   40 previous size:   a0  (Allocated)  CMNb (Protected)
 e1caf860 size:   20 previous size:   40  (Allocated)  ObDi
 e1caf880 size:   20 previous size:   20  (Free)       CMDa
 e1caf8a0 size:   20 previous size:   20  (Allocated)  CMVI
 e1caf8c0 size:   40 previous size:   20  (Allocated)  IoNm
 e1caf900 size:   80 previous size:   40  (Allocated)  FSim
 e1caf980 size:   20 previous size:   80  (Free)       MmSt
 e1caf9a0 size:   60 previous size:   20  (Allocated)  CMDa
 e1cafa00 size:  3e0 previous size:   60  (Allocated)  NtfF
 e1cafde0 size:   60 previous size:  3e0  (Allocated)  CMDa
 e1cafe40 size:   40 previous size:   60  (Allocated)  CMVa
 e1cafe80 size:   40 previous size:   40  (Allocated)  CMVa
 e1cafec0 size:   80 previous size:   40  (Allocated)  FSim
 e1caff40 size:   40 previous size:   80  (Allocated)  Key  (Protected)
 e1caff80 size:   80 previous size:   40  (Allocated)  Gla@
```

The Kernel Debugger will display the allocation with an "*". All the kernel debugger does is it takes the address, and takes off the last 3 numbers and replace them with 0. In this case, e1caf520, became e1caf000. It then simply walks the pool from there. It's a linked list, just like the user mode heap, except it doesn't go by pointer links but rather by size links. The sizes point forward and backward. In the above case, the sizes are shifted by 5, and the sizes are represented by bytes. Take the last allocation for example.

```
kd> db e1caff80
e1caff80  02 81 02 04 47 6c 61 40-59 01 10 08 00 00 00 00  ....Gla@Y.......
e1caff90  00 00 00 80 00 00 00 00-07 00 00 00 00 00 00 00  ................
e1caffa0  00 00 00 00 14 00 00 80-8b 00 00 00 f0 00 75 00  ..............u.
e1caffb0  00 00 00 00 d4 d0 c8 00-00 00 00 00 00 00 00 00  ................
e1caffc0  01 00 00 00 00 00 00 00-ff ff ff 00 04 00 00 00  ................
e1caffd0  08 00 00 00 c8 d0 d4 00-00 00 00 00 d4 d0 c8 00  ................
e1caffe0  01 05 00 00 00 00 00 05-15 00 00 00 85 e7 7e 2f  ..............~/
e1cafff0  23 f3 f6 63 f8 9f b4 74-01 02 00 00 00 00 00 00  #..c...t........
kd> ? 02 << 5
Evaluate expression: 64 = 00000040
kd> ? 04 << 5
Evaluate expression: 128 = 00000080
```

The flags in the middle specify a `SHORT` that tells whether the allocation is free or used, for

example. The tag is the next 4 bytes, and then is the allocated memory. 8 byte header, just like in user mode. (Of course, there is a large header in usermode for large allocations in the "VirtualAlloc" list of the heap.) Of course, the kernel can allocate large blocks of memory as well in the pool, how does that work? You just get a few pages of memory not allocated from a pool, you get your own address that looks like "e1caf000", and unlike doing this in usermode that has a 32 byte header information with size, there is no header information.

Ready Threads

Ready threads are threads that are ready to be executed, in a READY state as opposed to a WAIT state.

```
kd> !ready 0
Ready Threads at priority 0
  THREAD fcebf020  Cid 8.4  Teb: 00000000  Win32Thread: 00000000 READY
```

Just *!ready* will display the entire thread information. In Windows XP/2003, there is also a *!running* which will show the currently running threads on multiprocessor systems. Also, ~<number> <NUMBER>s in the usermode debugger switches between threads, in the kernel debugger it switches between processors.

CPU ID

There are other things you can do like *!cpuid*.

```
kd> !cpuid
CP  F/M/S  Manufacturer     MHz
 0  5,8,12 AuthenticAMD     350
```

Virtual Memory Usage

To display memory usage, you can use *!vm*.

```
kd> !vm

*** Virtual Memory Usage ***
    Physical Memory:    31613   (  126452 Kb)
    Page File: \??\C:\pagefile.sys
       Current:    184320Kb Free Space:    180464Kb
       Minimum:    184320Kb Maximum:       368640Kb
    Available Pages:    17617   (   70468 Kb)
    ResAvail Pages:     26203   (  104812 Kb)
    Modified Pages:       543   (    2172 Kb)
    NonPagedPool Usage:   677   (    2708 Kb)
    NonPagedPool Max:   12528   (   50112 Kb)
    PagedPool 0 Usage:   2993   (   11972 Kb)
    PagedPool 1 Usage:    439   (    1756 Kb)
    PagedPool 2 Usage:    451   (    1804 Kb)
    PagedPool Usage:     3883   (   15532 Kb)
    PagedPool Maximum:  25600   (  102400 Kb)
    Shared Commit:        324   (    1296 Kb)
    Special Pool:           0   (       0 Kb)
    Free System PTEs:   14442   (   57768 Kb)
    Shared Process:      1125   (    4500 Kb)
    PagedPool Commit:    3883   (   15532 Kb)
    Driver Commit:        912   (    3648 Kb)
    Committed pages:    11945   (   47780 Kb)
    Commit limit:       73862   (  295448 Kb)

    Total Private:       5662   (   22648 Kb)
     00a4 winlogon.exe     1358 (    5432 Kb)
     00d8 services.exe      587 (    2348 Kb)
     02e4 explorer.exe      557 (    2228 Kb)
     01b4 SPOOLSV.EXE       503 (    2012 Kb)
     00e4 lsass.exe         446 (    1784 Kb)
     01e8 svchost.exe       412 (    1648 Kb)
     00a8 csrss.exe         329 (    1316 Kb)
     0374 mspaint.exe       303 (    1212 Kb)
     008c smss.exe          273 (    1092 Kb)
     018c svchost.exe       227 (     908 Kb)
     0290 winvnc.exe        164 (     656 Kb)
     0264 winmgmt.exe       154 (     616 Kb)
     02e0 cmd.exe           143 (     572 Kb)
     0228 mstask.exe        141 (     564 Kb)
     0210 regsvc.exe         59 (     236 Kb)
     0008 System              6 (      24 Kb)
```

If you're looking for "is the system out of memory", you basically want to look at the "committed pages" versus the "commit limit". The committed pages are how much memory is currently being used and the commit limit is how much total memory can be used.

The next tool that can be used is *!memusage* which builds a map of all memory mapped files on the system.

```
kd> !memusage
 loading PFN database
loading (99% complete)
             Zeroed:   3163 ( 12652 kb)
               Free:     20 (    80 kb)
            Standby:  14434 ( 57736 kb)
           Modified:    543 (  2172 kb)
    ModifiedNoWrite:      0 (     0 kb)
       Active/Valid:  13567 ( 54268 kb)
         Transition:      0 (     0 kb)
            Unknown:      0 (     0 kb)
              TOTAL:  31727 (126908 kb)
  Building kernel map
  Finished building kernel map


  Usage Summary (in Kb):
Control Valid Standby Dirty Shared Locked PageTables  name
fce7a3c8   468  16776     0      0     0     0    No Name for File
fccf6148     0   2876     0      0     0     0  mapped_file( CIM.REP )
fcd43388     0     96     0      0     0     0  mapped_file( chord.wav )
fcc40568     0     48     0      0     0     0  mapped_file( index.dat )
fcc64308     0   2420     0      0     0     0  mapped_file( mshtml.dll )
fcce8b28     0    188     0      0     0     0  mapped_file( rasppp.dll )
fcc76728     0    208     0      0     0     0  mapped_file( h323.tsp )
fccec3c8   220     92     0      0     0     0  mapped_file( netcfgx.dll )
fce48e88  1152    180     0      0     0     0  mapped_file( win32k.sys )
fcde4f28    16    460     0      0     0     0  mapped_file( wininet.dll )
fcc7f808     0     32     0      0     0     0  mapped_file( wbemcomn.dll )
fcc4d7a8     0    388     0      0     0     0  mapped_file( ntvdm.exe )
fcceb128    80    200     0      0     0     0  mapped_file( rasdlg.dll )
fcde5a08   200    896     0    120     0     0  mapped_file( shell32.dll )
fceca188  3196   1280     0      0     0     0    No Name for File
fccecd88   108     36     0      0     0     0  mapped_file( tapisrv.dll )
fcda5a28     8    204     0      0     0     0  mapped_file( msgina.dll )
fcc83348     0    300     0      0     0     0  mapped_file( wbemess.dll )
fcce9248    40    756     0      0     0     0  mapped_file( shdocvw.dll )
fcd10a68     0     32     0      0     0     0  mapped_file( notepad.exe )
fcd7dd08     0    504     0      0     0     0  mapped_file( jscript.dll )
fcc7cc48     0     32     0      0     0     0  mapped_file( mswsock.dll )
fcc4a4a8     0    116     0      0     0     0  mapped_file( dxtmsft.dll )
fcce95e8    32     40     0      0     0     0  mapped_file( ntmarta.dll )
fcce89a8     0     12     0      0     0     0  mapped_file( ntlsapi.dll )
fcc821a8     0     68     0      0     0     0  mapped_file( modemui.dll )
fcc7a4e8    16    196     0      0     0     0  mapped_file( msi.dll )
fccec688   112     32     0      0     0     0  mapped_file( rasmans.dll )
fcc4ea68   240     64     0      0     0     0  mapped_file( mspaint.exe )
fcd64ba8     0     32     0      0     0     0  mapped_file( tapisrv.dll )
fcde4aa8     0     36     0      0     0     0  mapped_file( mpr.dll )
fcdc33a8   164     64     0      0     0     0  mapped_file( winsrv.dll )
fcceeb08     0     32     0      0     0     0  mapped_file( wkssvc.dll )
fccf3ee8     0     32     0      0     0     0  mapped_file( winlogon.exe )
fcc54a28   112     36     0      0     0     0  mapped_file( cmd.exe )
fcc60408     0    148     0      0     0     0  mapped_file( dxtrans.dll )
fccee4a8     0     32     0      0     0     0  mapped_file( cryptsvc.dll )
fcc7fe08     0     32     0      0     0     0  mapped_file( SPOOLSV.EXE )
fcd7e788    80    512     0      0     0     0  mapped_file( browseui.dll )
fcc7ed28     0      0     4      0     0     0  mapped_file( software.LOG )
fcc7f128    40    264     0      8     0     0  mapped_file( netshell.dll )
fcc51ac8     0     32     0      0     0     0  mapped_file( jscript.dll )
...
--------    12      0     0 ----- -----     8  pagefile section (78a7)
--------   456      0     0 ----- -----    72  process ( mstask.exe )
--------   452    536   116 ----- -----    92  process ( lsass.exe )
--------   140      0     0 ----- -----    28  process ( smss.exe )
--------    88    452     0 ----- -----    52  process ( winmgmt.exe )
--------   236      0     0 ----- -----    36  process ( regsvc.exe )
--------   520      0     0 ----- -----    72  process ( winvnc.exe )
--------    12      0     0 ----- -----     8  pagefile section (6933)
--------  1320    100     0 -----     0 -----  driver ( ntoskrnl.exe )
--------    60     16     0 -----     0 -----  driver ( hal.dll )
...
```

As you can see, all files that are loaded in the system are loaded as memory mapped files. This means the Operating System uses the copy on disk to continuously load it into memory. It doesn't

copy it into memory and then put it in the page file. To get information on one of these memory mapped files, such as how many references are currently being mapped, etc., use *!ca*.

```
kd> !ca fcdee008

ControlArea @fcdee008
  Segment:     e1b764c8    Flink              0   Blink:              0
  Section Ref         1    Pfn Ref           1e   Mapped Views:       1
  User Ref            2    Subsections        4   Flush Count:        0
  File Object fcdbe728     ModWriteCount      0   System Views:       0
  WaitForDel          0    Paged Usage      100   NonPaged Usage      c0
  Flags (90000a0) Image File HadUserReference Accessed

    File: \WINNT\system32\winlogon.exe

Segment @ e1b764c8:
   ControlArea  fcdee008  Total Ptes          0  NonExtendPtes:       2d
   WriteUserRef       2d  Extend Info         0  SizeOfSegment: 2d000
   Image Base          0  Committed    e1b765b8  Based Addr    2d36f438
PTE Template:          3
   Image commit  1000000  Image Info:           0
   ProtoPtes    e1b76500

Subsection 1. @ fcdee040
   ControlArea: fcdee008  Starting Sector 0 Number Of Sectors 2
   Base Pte     e1b76500  Ptes In subsect        1 Unused Ptes           0
   Flags              15  Sector Offset          0 Protection            1
    ReadOnly SubsectionStatic

Subsection 2. @ fcdee060
   ControlArea: fcdee008  Starting Sector 2 Number Of Sectors ff
   Base Pte     e1b76504  Ptes In subsect       20 Unused Ptes           0
   Flags              35  Sector Offset          0 Protection            3
    ReadOnly SubsectionStatic

Subsection 3. @ fcdee080
   ControlArea: fcdee008  Starting Sector 101 Number Of Sectors 11
   Base Pte     e1b76584  Ptes In subsect        3 Unused Ptes           0
   Flags              55  Sector Offset          0 Protection            5
    ReadOnly SubsectionStatic

Subsection 4. @ fcdee0a0
   ControlArea: fcdee008  Starting Sector 112 Number Of Sectors 48
   Base Pte     e1b76590  Ptes In subsect        9 Unused Ptes           0
   Flags              15  Sector Offset          0 Protection            1
    ReadOnly SubsectionStatic
```

You will notice there is only one mapped view for Winlogon on this machine. Don't worry if it says 0 views and the process is running; look on the list, there may be two listings for that file and the other one is the mapview being used.

Conclusion

This is a simple introduction to the kernel debugger and a few things that you can do. While I haven't covered everything, I hope that you now know at least the basics and can navigate it. There's always the help and http://www.google.com/! You learn something new everyday; one person can't know everything, but they can try! I tried to make sure the information presented is as accurate as possible, but things can get through, so let me know if I missed something or misrepresented something. I may add more Kernel Debugging tutorials in the future, with other topics. We didn't even cover IRPs!

You usually don't end up working with every single detail in the listing. You usually end up using a bunch of different commands to get different information, and each command you may only ever use a subset of the data, and this is fine. You may not even care what the other data is or even know what it is. Perhaps even if you know what it is, it would never be useful to you. So, don't get discouraged if you don't know what a piece of displayed information is or if you can't find any references to it online. If you really need to know though, you could always set ba's on the address and track down who's using it for what.

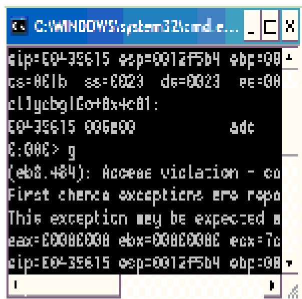
License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found here

About the Author

Toby Opferman



Member

Toby Opferman has worked in just about all aspects of Windows development including applications, services and drivers.

He has also played a variety of roles professionally on a wide range of projects. This has included pure researching roles, architect roles and developer roles. He also was also solely responsible for debugging traps and blue screens for a number of years.

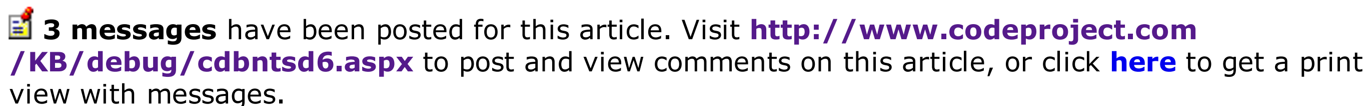
Previously of Citrix Systems he is very experienced in the area of Terminal Services. He currently works on Operating Systems and low level architecture at Intel.

Occupation: Engineer

Company: Intel

Location: United States

Discussions and Feedback



3 messages have been posted for this article. Visit **http://www.codeproject.com/KB/debug/cdbntsd6.aspx** to post and view comments on this article, or click **here** to get a print view with messages.

PermaLink | Privacy | Terms of Use
Last Updated: 7 Aug 2004
Editor: Smitha Vijayan

Copyright 2004 by Toby Opferman
Everything else Copyright © CodeProject, 1999-2009
Web21 | Advertise on the Code Project