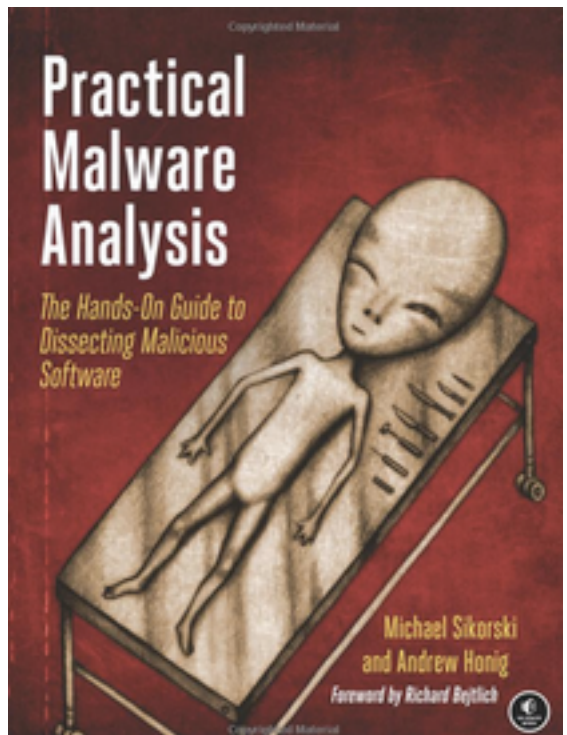
Practical Malware Analysis

Ch 10: Kernel Debugging with WinDbg



Updated 10-29-19

WinDbg v. OllyDbg

- OllyDbg is the most popular user-mode debugger for malware analysts
- WinDbg can be used in either user-mode or kernel-mode
- This chapter explores ways to use WinDbg for kernel debugging and rootkit analysis

Drivers and Kernel Code

Device Drivers

- Windows device **drivers** allow third-party developers to run code in the Windows kernel
- Drivers are difficult to analyze
  - They load into memory, stay resident, and respond to requests from applications
- Applications don't directly access kernel drivers
  - They access *device objects* which send requests to particular devices

Devices

- **Devices** are not physical hardware components
  - They are software representations of those components
- A **driver** creates and destroys **devices**, which can be accessed from user space

Example: USB Flash Drive

- User plugs in flash drive
- Windows creates the ***F****:* drive device object
- Applications can now make requests to the ***F****:* drive (such as read and write)
  - They will be sent to the driver for that USB flash drive
- User plugs in a second flash drive
  - It may use the same driver, but applications access it through the ***G****:* drive

Loading DLLs (Review)

- DLLs are loaded into processes
  - DLLs export functions that can be used by applications
  - Using the **export table**
  - When a function loads or unloads the library, it calls **DLLMain**
  - Link Ch 10n

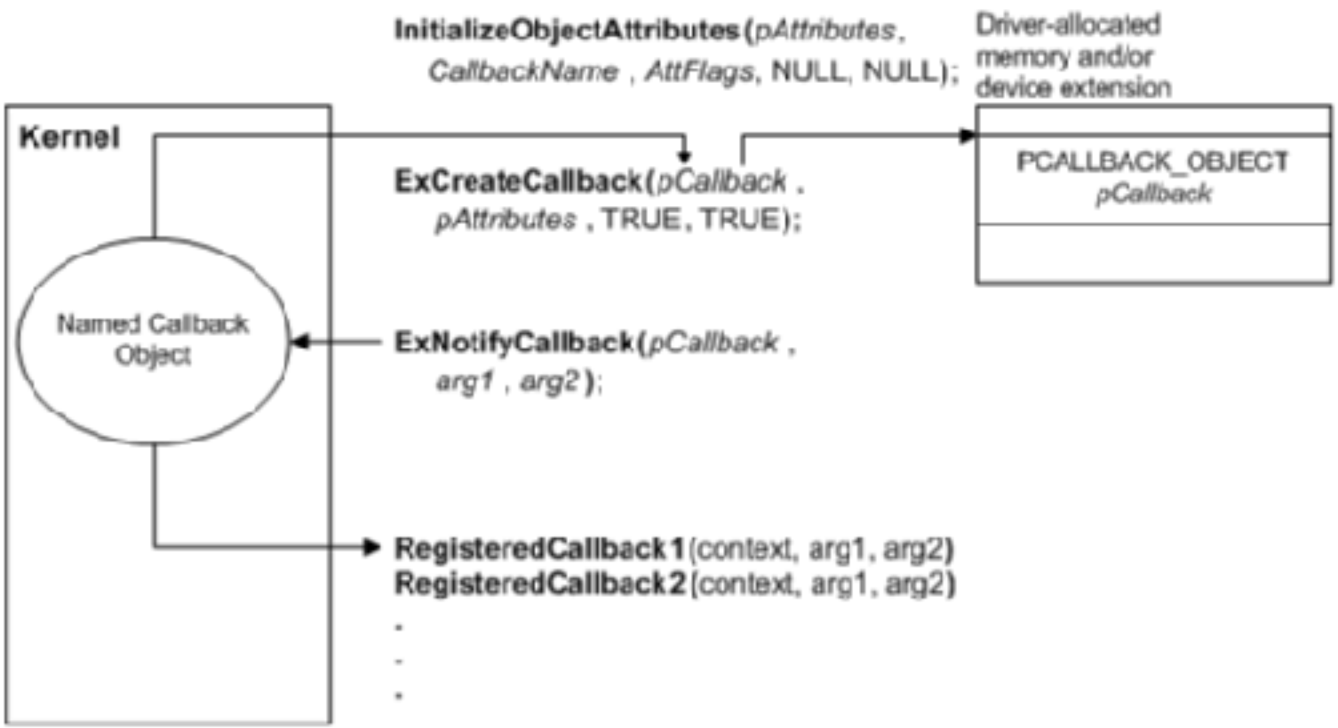
Loading Drivers

- Drivers must be loaded into the kernel
  - When a driver is first loaded, its **DriverEntry** procedure is called
  - To prepare **callback objects**
  - Just like **DLLMain** for DLLs
  - Links Ch 10n, 10o, 10p

Defining a Callback Object

06/15/2017 • 2 minutes to read • Contributors

A driver can create a callback object, through which other drivers can request notification of conditions defined by the creating driver. The following figure shows the steps involved in defining a callback object.



DLLs v. Drivers

- DLL
  - Loads into memory when a process is launched
  - Executes **DLLMain** at loadtime
  - Prepares the **export table**
- Driver
  - Loads into **kernel** when hardware is added
  - Executes **DriverEntry** at loadtime
  - Prepares **callback functions** and **callback objects**

DriverEntry

- DLLs expose functionality through the export table; drivers don't
- Drivers must register the address for callback functions
  - They will be called when a user-space software component requests a service
  - **DriverEntry** routine performs this registration
  - Windows creates a *driver object* structure, passes it to **DriverEntry** which fills it with callback functions
  - **DriverEntry** then creates a device that can be accessed from user-land

Example: Normal Read

- Normal read request
  - User-mode application obtains a file handle to device
  - Calls **ReadFile** on that handle
  - Kernel processes **ReadFile** request
  - Invokes the driver's callback function handling I/O

Malicious Request

- Most common request from malware is **DeviceIoControl**
  - A generic request from a user-space module to a device managed by a driver
  - User-space program passes in an arbitrary-length buffer of input data
  - Received an arbitrary-length buffer of data as output

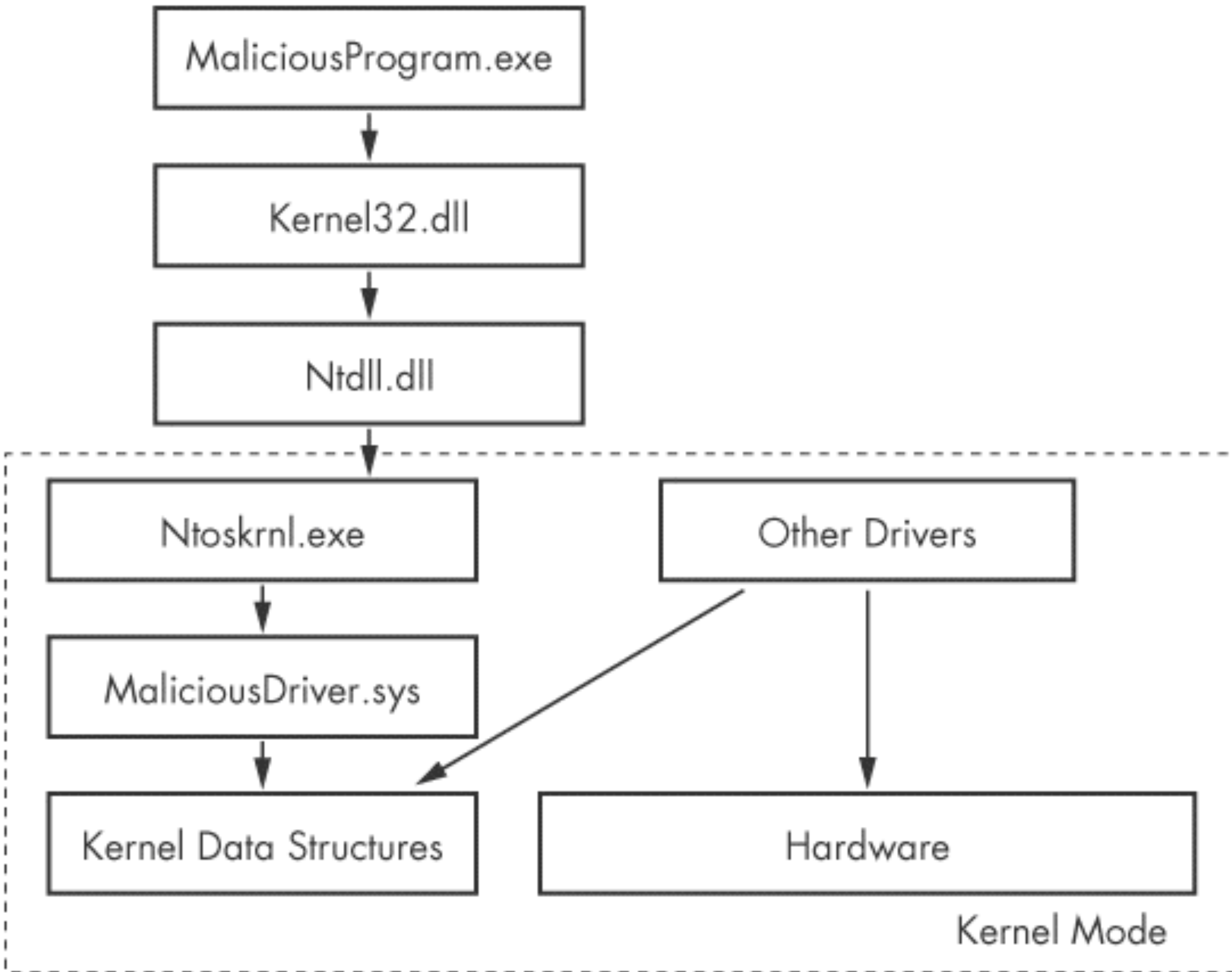


Figure 11-1. How user-mode calls are handled by the kernel

NOTE

Some kernel-mode malware has no significant user-mode component. It creates no device object, and the kernel-mode driver executes on its own.

Ntoskrnl.exe & Hal.dll

- Malicious drivers rarely control hardware
- They interact with *Ntoskrnl.exe* & *Hal.dll*
  - *Ntoskrnl.exe* has code for core OS functions
  - *Hal.dll* has code for interacting with main hardware components
- Malware will import functions from one or both of these files so it can manipulate the kernel

Kahoot!

Setting Up Kernel Debugging

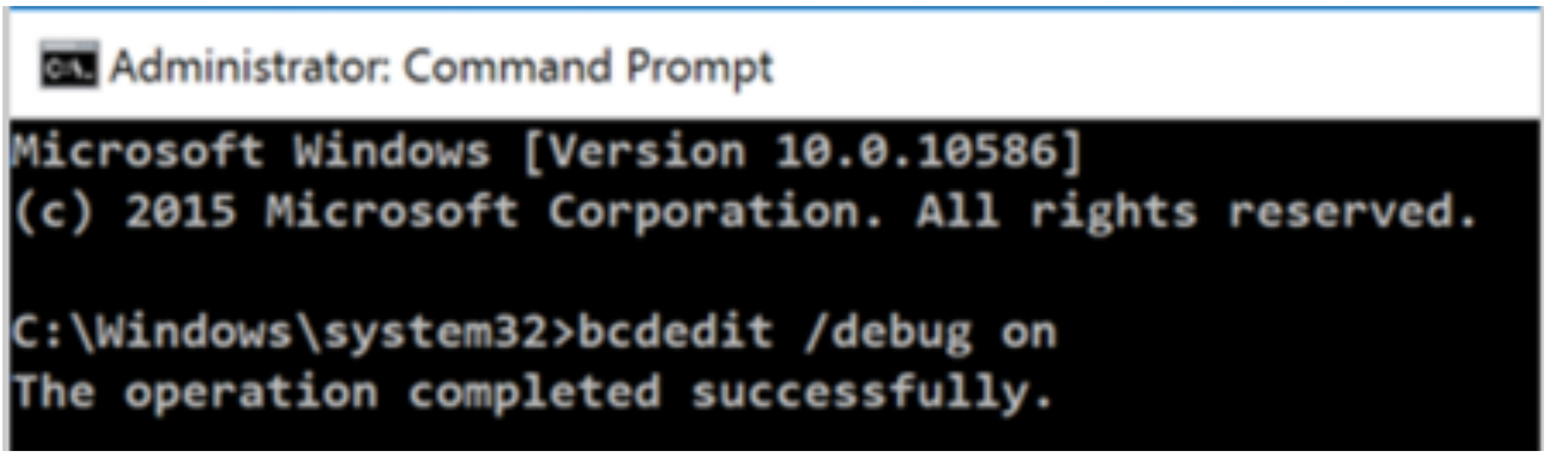
VMware

- In the virtual machine, enable kernel debugging
- Configure a virtual serial port between VM and host
- Configure WinDbg on the host machine

Boot.ini

- The book activates kernel debugging by editing Boot.ini
- But Microsoft abandoned that system after Windows XP
- The new system uses **bcdedit**

bcdedit


Administrator: Command Prompt
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.
C:\Windows\system32>bcdedit /debug on
The operation completed successfully.

Installing WinDbg

Debugging Tools for Windows (WinDbg, KD, CDB, NTSD)

02/21/2017 • 3 minutes to read • Contributors

Start here for an overview of Debugging Tools for Windows. This tool set includes WinDbg and other debuggers.

3 ways to get Debugging Tools for Windows

- **As part of the WDK**

  Debugging Tools for Windows is included in the WDK. You can get the WDK here.

- **As a standalone tool set**

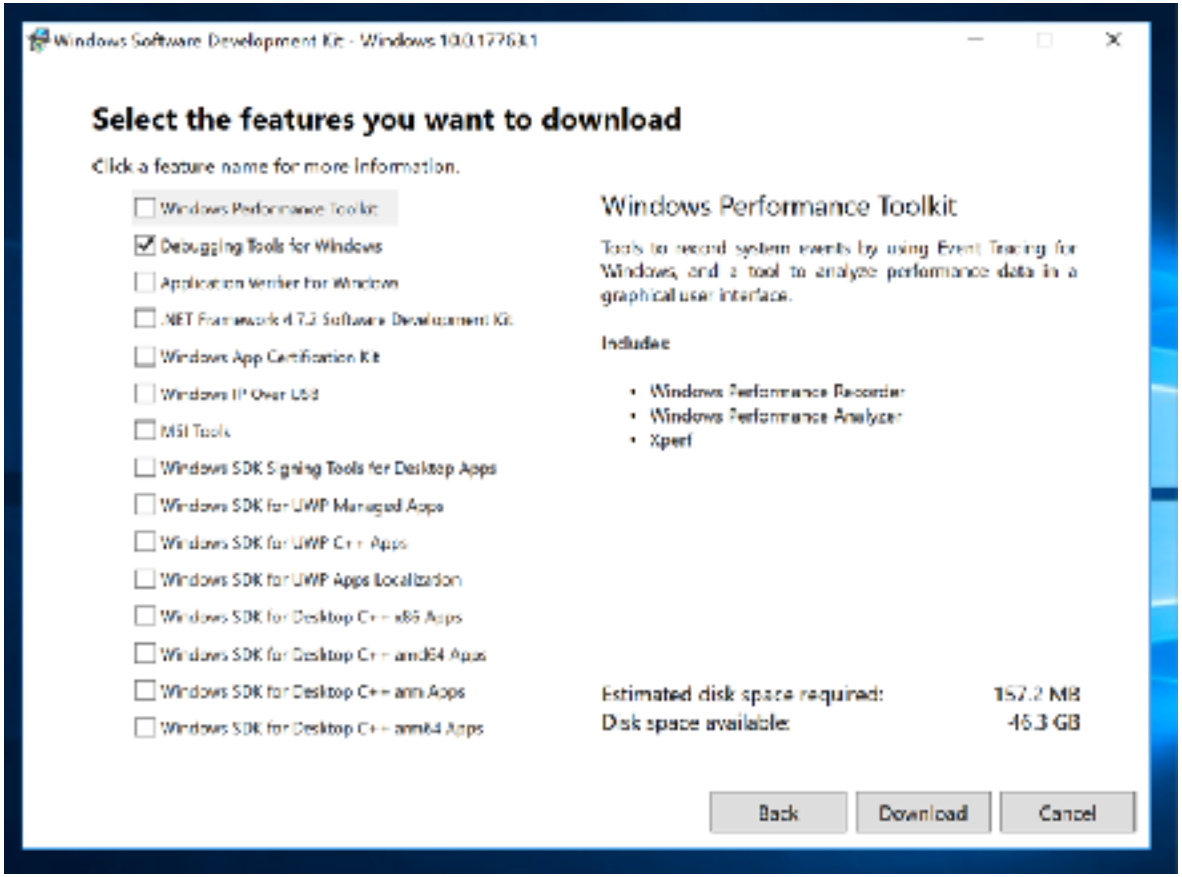
  If you want to download only Debugging Tools for Windows, install the Windows SDK, and, during the installation, select the **Debugging Tools for Windows** box and clear all the other boxes.

- **As part of the Windows SDK**

  Install the complete Windows Software Development Kit (SDK). Debugging Tools for Windows is included in the Windows SDK. You can get the Windows SDK here.

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/

Installing WinDbg



Run LiveKD

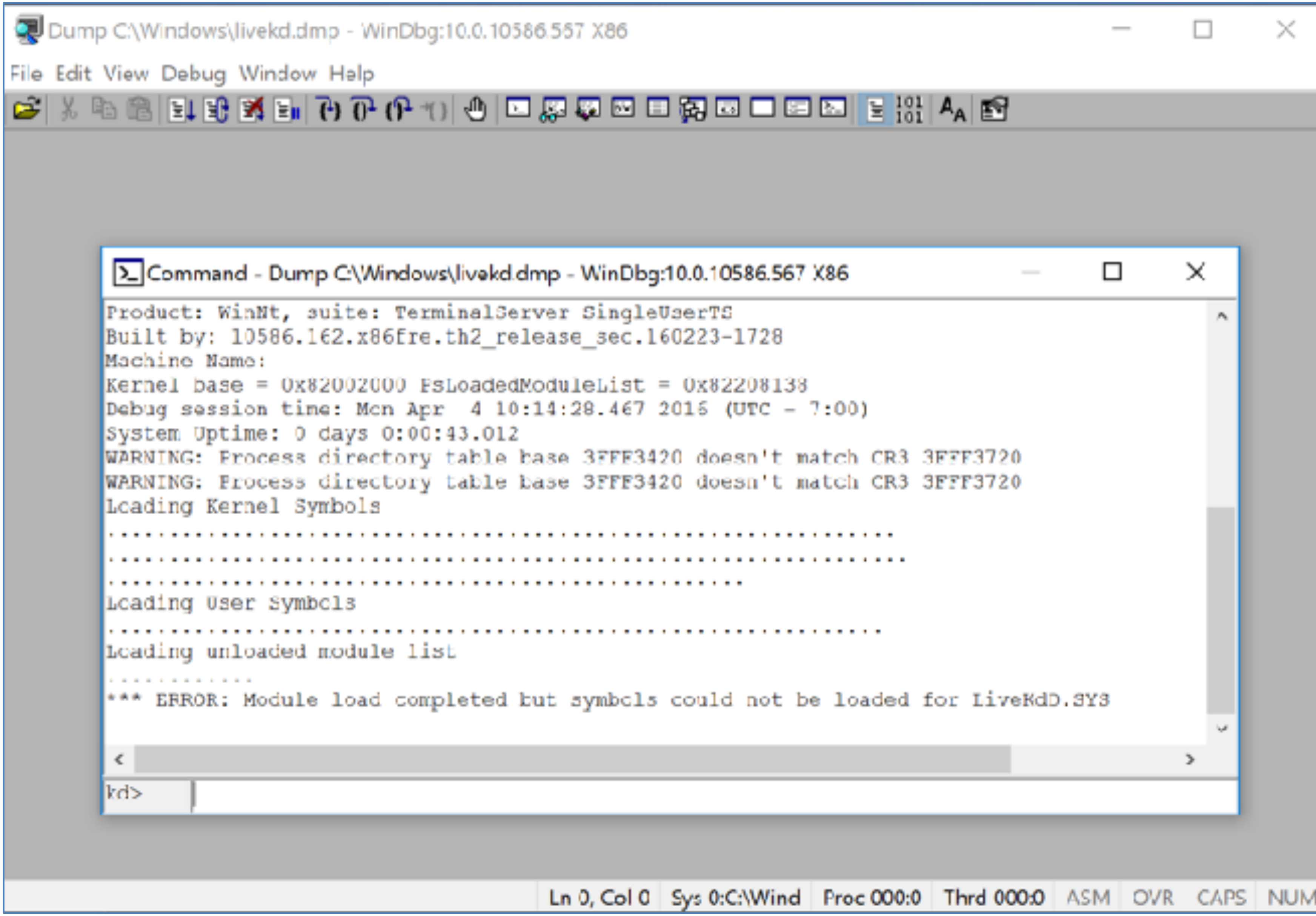
```
C:\Windows\system32>livekd -w

LiveKd v5.40 - Execute kd/windbg on a live system
Sysinternals - www.sysinternals.com
Copyright (C) 2000-2015 Mark Russinovich and Ken Johnson


Symbols are not configured. Would you like LiveKd to set the _NT_SYMBOL_PATH
directory to reference the Microsoft symbol server so that symbols can be
obtained automatically? (y/n)
```

PMA 410c: Kernel Debugging Windows 2016 Server (15 pts)

PMA 410: Kernel Debugging on Windows 2008 Server (15 pts)



Dump C:\Windows\livekd.dmp - WinDbg:10.0.10586.567 X86
File Edit View Debug Window Help
Command - Dump C:\Windows\livekd.dmp - WinDbg:10.0.10586.567 X86
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 10586.162.x86fre.th2_release_sec.160223-1728
Machine Name:
Kernel base = 0x82002000 PsLoadedModuleList = 0x82208138
Debug session time: Mon Apr 4 10:14:28.467 2016 (UTC - 7:00)
System Uptime: 0 days 0:00:43.012
WARNING: Process directory table base 3FFF3420 doesn't match CR3 3FFF3720
WARNING: Process directory table base 3FFF3420 doesn't match CR3 3FFF3720
Loading Kernel Symbols
....
....
....
Loading User Symbols
....
Loading unloaded module list
....
*** ERROR: Module load completed but symbols could not be loaded for LiveKdD.SYS
kd>
Ln 0, Col 0
Sys 0:C:\Wind
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM

Using WinDbg

- Command-Line Commands

Reading from Memory

- d*x* *addressToRead*
- *x* can be
  - da Displays as ASCII text
  - du Displays as Unicode text
  - dd Displays as 32-bit double words
- da 0x401020
  - Shows the ASCII text starting at 0x401020

Editing Memory

- e*x addressToWrite dataToWrite*
- *x* can be
  - ea Writes as ASCII text
  - eu Writes as Unicode text
  - ed Writes as 32-bit double words

Using Arithmetic Operators

- Usual arithmetic operators + - / *
- **dwo** reveals the value at a 32-bit location pointer
- **du dwo (esp+4)**
  - Shows the first argument for a function, as a wide character string

Setting Breakpoints

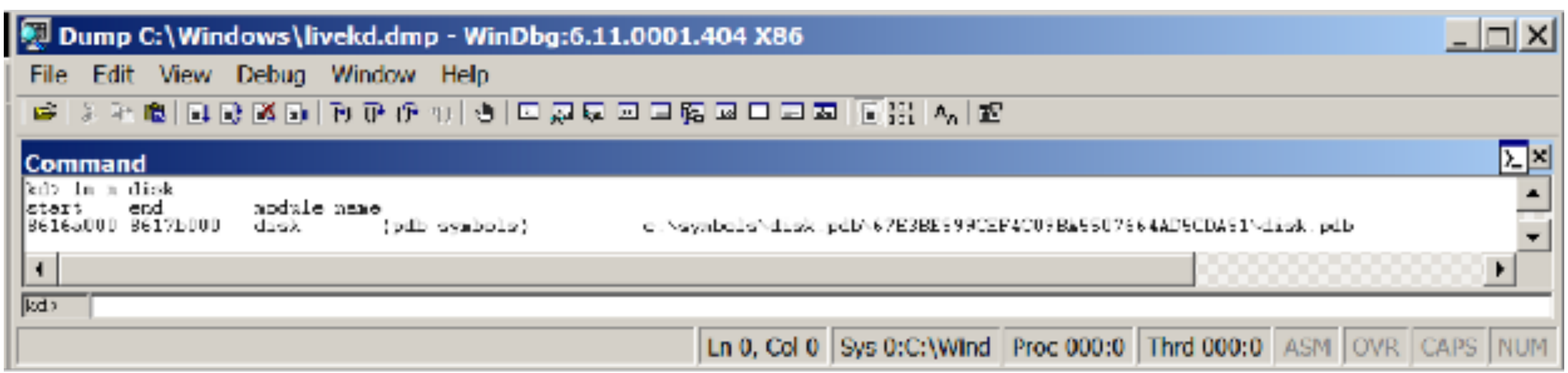
- **bp** sets breakpoints
- You can specify an action to be performed when the breakpoint is hit
- **g** tells it to resume running after the action
- **bp GetProcAddress "da dwo(esp+8); g"**
  - Breaks when GetProcAddress is called, prints out the second argument, and then continues
  - The second argument is the function name

No Breakpoints with LiveKD

- LiveKD works from a memory dump
- It's read-only
- So you can't use breakpoints

Listing Modules

- **lm**
  - Lists all modules loaded into a process
    - Including EXEs and DLLs in user space
    - And the kernel drivers in kernel mode
  - As close as WinDbg gets to a memory map
- **lm m disk**
  - Shows the disk driver



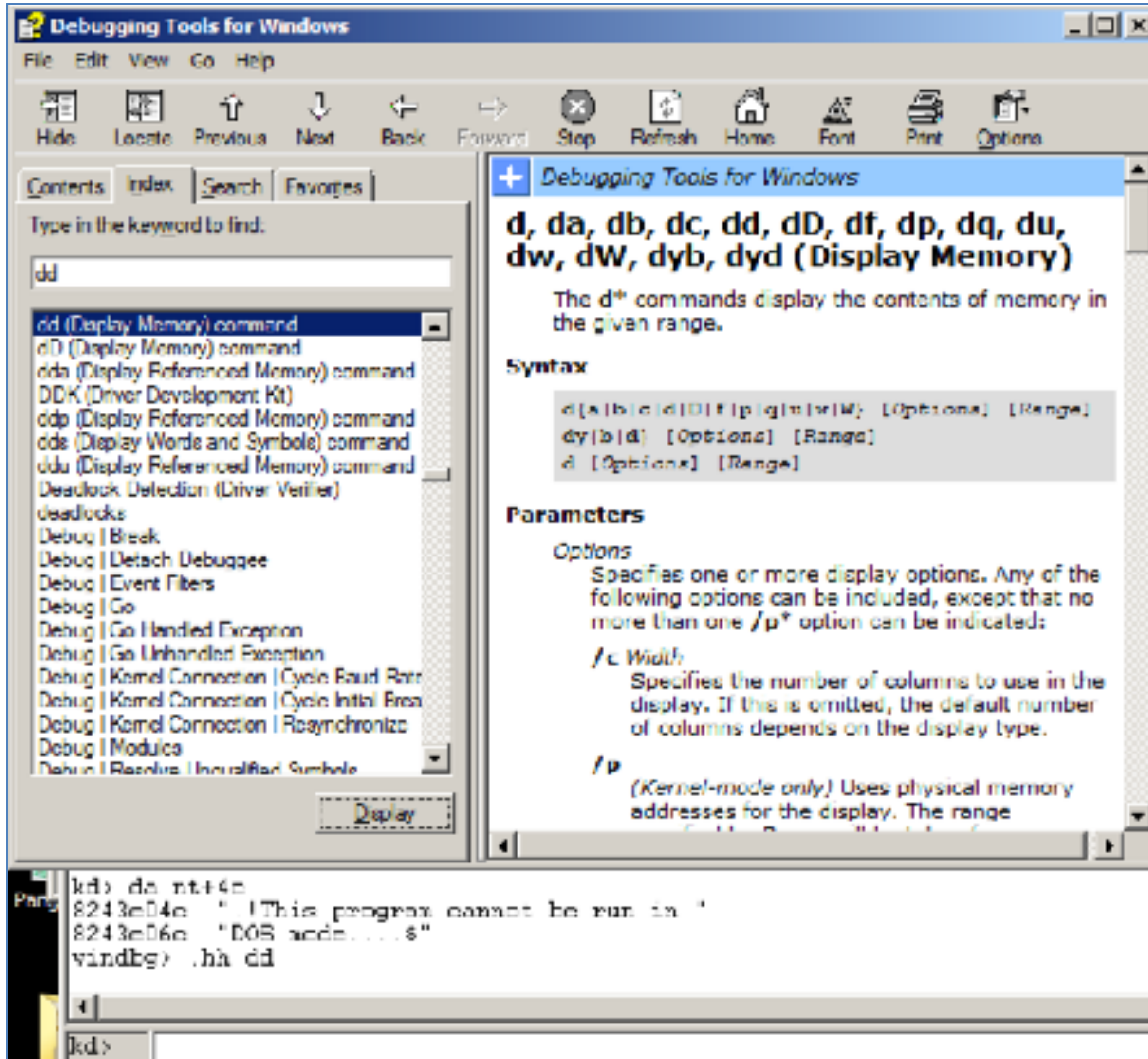
Reading from Memory

- **dd nt**
  - Shows the start of module "nt"
- **dd nt L10**
  - Shows the first 0x10 words of "nt"

```
kd> dd nt
8243e000  00905a4d 00000003 00000004 0000ffff
8243e010  000000b8 00000000 00000040 00000000
8243e020  00000000 00000000 00000000 00000000
8243e030  00000000 00000000 00000000 00000268
8243e040  0eba1f0e cd09b400 4c01b821 685421cd
8243e050  70207369 72676f72 63206d61 6f6e6e61
8243e060  65622074 6e757220 206e6920 20534f44
8243e070  65646f6d 0a0d0d2e 00000024 00000000
kd> dd nt L10
8243e000  00905a4d 00000003 00000004 0000ffff
8243e010  000000b8 00000000 00000040 00000000
8243e020  00000000 00000000 00000000 00000000
8243e030  00000000 00000000 00000000 00000268
```

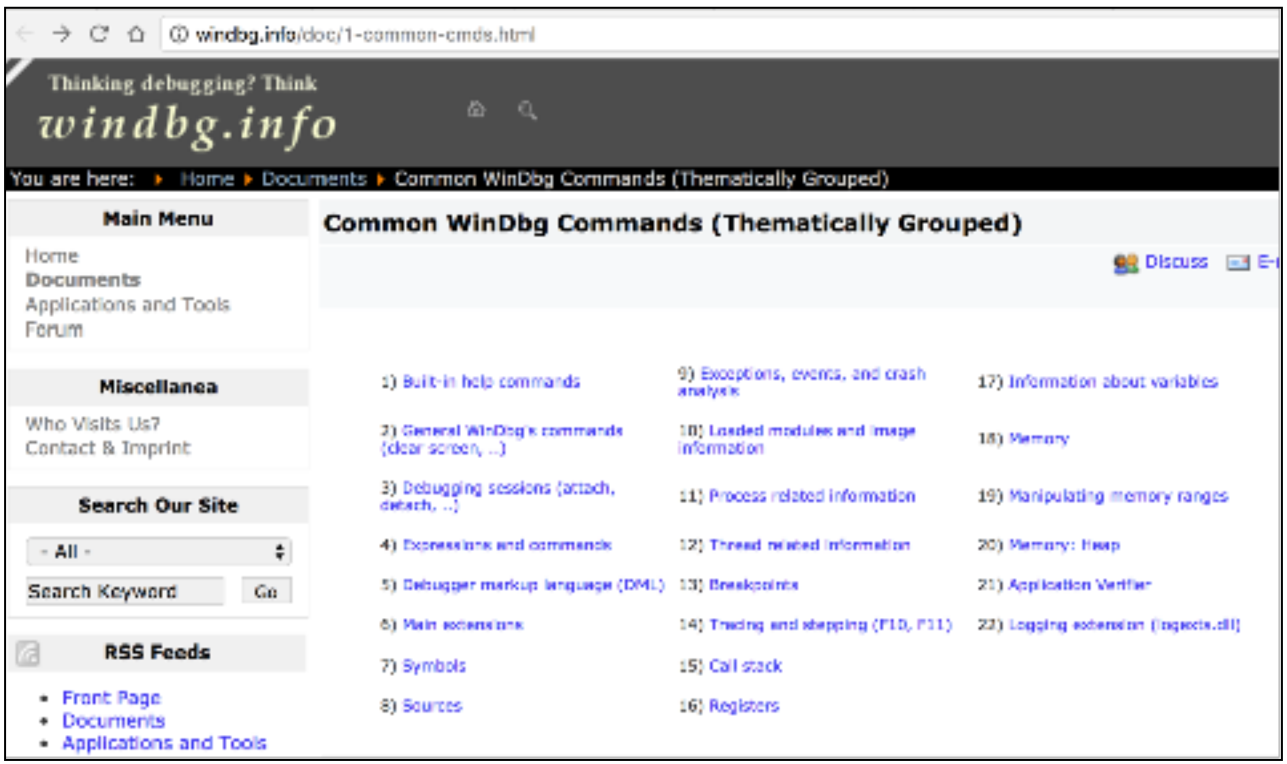
Online Help

- **.hh dd**
  - Shows help about "dd" command
  - But there are no examples



More Commands

- r
  - Dump all registers
    - Link Ch 10m



Kahoot!

Microsoft Symbols

Symbols are Labels

- Including symbols lets you use
  - **MmCreateProcessAddressSpace**
- instead of
  - **0x8050f1a2**

Searching for Symbols

- ***moduleName!symbolName***
  - Can be used anywhere an address is expected
- ***moduleName***
  - The EXE, DLL, or SYS filename (without extension)
- ***symbolName***
  - Name associated with the address
- ntoskrnl.exe is an exception, and is named **nt**
  - Ex: **u nt!NtCreateProcess**
    - Unassembles that function (disassembly)

Demo

- Try these
  - **u nt!ntCreateProcess**
  - **u nt!ntCreateProcess L10**
  - **u nt!ntCreateProcess L20**

```
kd> u nt!ntCreateProcess
nt!NtCreateProcess:
826d1f9f 8bff            mov     edi,edi
826d1fa1 55              push    ebp
826d1fa2 8bec            mov     ebp,esp
826d1fa4 33c0            xor     eax,eax
826d1fa6 f6451c01        test    byte ptr [ebp+1Ch],1
826d1faa 7401            je      nt!NtCreateProcess+0xe (826d1fad)
826d1fac 40              inc     eax
826d1fad f6452001        test    byte ptr [ebp+20h],1
```

Deferred Breakpoints

- **bu *newModule*!*exportedFunction***
  - Will set a breakpoint on *exportedFunction* as soon as a module named *newModule* is loaded
- $iment
  - Function that finds the entry point of a module
- **bu $iment(*driverName*)**
  - Breaks on the entry point of the driver before any of the driver's code runs

Searching with x

- You can search for functions or symbols using wildcards
- **x nt!*CreateProcess***
  – Displays exported functions & internal functions

```
0:003> x nt!*CreateProcess*
805c736a nt!NtCreateProcessEx = <no type information>
805c7420 nt!NtCreateProcess = <no type information>
805c6a8c nt!PspCreateProcess = <no type information>
804fe144 nt!ZwCreateProcess = <no type information>
804fe158 nt!ZwCreateProcessEx = <no type information>
8055a300 nt!PspCreateProcessNotifyRoutineCount = <no type information>
805c5e0a nt!PsSetCreateProcessNotifyRoutine = <no type information>
8050f1a2 nt!MmCreateProcessAddressSpace = <no type information>
8055a2e0 nt!PspCreateProcessNotifyRoutine = <no type information>
```

Listing Closest Symbol with **ln**

- Helps in figuring out where a call goes
- **ln *address***
  - First lines show two closest matches
  - Last line shows exact match

```
  0:002> ln 805717aa
  kd> ln ntreadfile
1 (805717aa)   nt!NtReadFile   |  (80571d38)   nt!NtReadFileScatter
  Exact matches:
2     nt!NtReadFile = <no type information>
```

Viewing Structure Information with **dt**

- Microsoft symbols include type information for many structures
  - Including undocumented internal types
  - They are often used by malware
- **dt *moduleName!symbolName***
- **dt *moduleName!symbolName address***
  - Shows structure with data from *address*

Example 11-2. Viewing type information for a structure

```
0:000> dt nt!_DRIVER_OBJECT
kd> dt nt!_DRIVER_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Flags            : Uint4B
1  +0x00c DriverStart      : Ptr32 Void
   +0x010 DriverSize       : Uint4B
   +0x014 DriverSection    : Ptr32 Void
   +0x018 DriverExtension  : Ptr32 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING
   +0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
   +0x028 FastIoDispatch   : Ptr32 _FAST_IO_DISPATCH
   +0x02c DriverInit       : Ptr32     long
   +0x030 DriverStartIo    : Ptr32     void
   +0x034 DriverUnload     : Ptr32     void
   +0x038 MajorFunction    : [28] Ptr32     long
```

Demo

- Try these
  - **dt nt!_DRIVER_OBJECT**
  - **dt nt!_DEVICE_OBJECT**

```
kd> dt nt!_DEVICE_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Uint2B
   +0x004 ReferenceCount   : Int4B
   +0x008 DriverObject     : Ptr32 _DRIVER_OBJECT
   +0x00c NextDevice       : Ptr32 _DEVICE_OBJECT
   +0x010 AttachedDevice   : Ptr32 _DEVICE_OBJECT
   +0x014 CurrentIrp       : Ptr32 _IRP
   +0x018 Timer            : Ptr32 _IO_TIMER
   +0x01c Flags            : Uint4B
   +0x020 Characteristics  : Uint4B
   +0x024 Vpb              : Ptr32 _VPB
   +0x028 DeviceExtension  : Ptr32 Void
   +0x02c DeviceType       : Uint4B
   +0x030 StackSize        : Char
   +0x034 Queue            : <unnamed-tag>
   +0x05c AlignmentRequirement : Uint4B
   +0x060 DeviceQueue      : _KDEVICE_QUEUE
   +0x074 Dpc              : _KDPC
   +0x094 ActiveThreadCount : Uint4B
   +0x098 SecurityDescriptor : Ptr32 Void
   +0x09c DeviceLock       : _KEVENT
   +0x0ac SectorSize       : Uint2B
   +0x0ae Spare1           : Uint2B
   +0x0b0 DeviceObjectExtension : Ptr32 _DEVOBJ_EXTENSION
   +0x0b4 Reserved         : Ptr32 Void
```

Show Specific Values for the "Beep" Driver

Example 11-3. Overlaying data onto a structure

```
kd> dt nt!_DRIVER_OBJECT 828b2648
   +0x000 Type             :  4
   +0x002 Size             :  168
   +0x004 DeviceObject     :  0x828b0a30 _DEVICE_OBJECT
   +0x008 Flags            :  0x12
   +0x00c DriverStart      :  0xf7adb000
   +0x010 DriverSize       :  0x1080
   +0x014 DriverSection    :  0x82ad8d78
   +0x018 DriverExtension  :  0x828b26f0 _DRIVER_EXTENSION
   +0x01c DriverName       :  _UNICODE_STRING "\Driver\Beep"
   +0x024 HardwareDatabase :  0x80670ae0 _UNICODE_STRING
"\REGISTRY\MACHINE\
HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   :  (null)
   +0x02c DriverInit       : 10xf7adb66c     long  Beep!DriverEntry+0
   +0x030 DriverStartIo    :  0xf7adb51a     void  Beep!BeepStartIo+0
   +0x034 DriverUnload     :  0xf7adb620     void  Beep!BeepUnload+0
   +0x038 MajorFunction    :  [28] 0xf7adb46a     long  Beep!BeepOpen+0
```

Initialization Function

- The **DriverInit** function is called first when a driver is loaded
  - See labelled line in previous slide
- Malware will sometimes place its entire malicious payload in this function

Configuring Windows Symbols

- If your debugging machine is connected to an always-on broadband link, you can configure WinDbg to automatically download symbols from Microsoft as needed
- They are cached locally
- **File**, **Symbol File Path**
  - **`SRC*c:\websymbols*http://msdl.microsoft.com/download/symbols`**

Manually Downloading Symbols

- Link Ch 10a

Kahoot!

Kernel Debugging in Practice

Kernel Mode and User Mode Functions

- We'll examine a program that writes to files from kernel space
  - An unusual thing to do
  - Fools some security products
  - Kernel mode programs cannot call user-mode functions like **`CreateFile`** and **`WriteFile`**
  - Must use **`NtCreateFile`** and **`NtWriteFile`**

User-Space Code

Example 11-4. Creating a service to load a kernel driver

```
04001B3D  push    esi                     ; lpPassword
04001B3E  push    esi                     ; lpServiceStartName
04001B3F  push    esi                     ; lpDependencies
04001B40  push    esi                     ; lpdwTagId
04001B41  push    esi                     ; lpLoadOrderGroup
04001B42  push    [ebp+lpBinaryPathName] ; lpBinaryPathName
04001B45  push    1                       ; dwErrorControl
04001B47  push    3                       ; dwStartType
04001B49  push   1 1                      ; dwServiceType
04001B4B  push    0F01FFh                 ; dwDesiredAccess
04001B50  push    [ebp+lpDisplayName] ; lpDisplayName
04001B53  push    [ebp+lpDisplayName] ; lpServiceName
04001B56  push    [ebp+hSCManager] ; hSCManager
04001B59  call    ds:__imp__CreateServiceA@52
```

Creates a service with the `CreateService` function

`dwServiceType` is 0x01 (Kernel driver)

User-Space Code

Example 11-5. Obtaining a handle to a device object

```
04001893                xor     eax, eax
04001895                push    eax                ; hTemplateFile
04001896                push    80h                ; dwFlagsAndAttributes
0400189B                push    2                  ; dwCreationDisposition
0400189D                push    eax                ; lpSecurityAttributes
0400189E                push    eax                ; dwShareMode
0400189F                push    ebx                ; dwDesiredAccess
040018A0               2push    edi                ; lpFileName
040018A1               1call    esi ; CreateFileA
```

- Not shown: edi being set to
  - \\.\FileWriter\Device

User-Space Code

Once the malware has a handle to the device, it uses the `DeviceIoControl` function at 1 to send data to the driver as shown in Example 11-6.

Example 11-6. Using `DeviceIoControl` to communicate from user space to kernel space

```
04001910  push    0                   ; lpOverlapped
04001912  sub     eax, ecx
04001914  lea     ecx, [ebp+BytesReturned]
0400191A  push    ecx                 ; lpBytesReturned
0400191B  push    64h                 ; nOutBufferSize
0400191D  push    edi                 ; lpOutBuffer
0400191E  inc     eax
0400191F  push    eax                 ; nInBufferSize
04001920  push    esi                 ; lpInBuffer
04001921  push    9C402408h           ; dwIoControlCode
04001926  push    [ebp+hObject]       ; hDevice
0400192C  call    ds:DeviceIoControl 1
```

Kernel-Mode Code

- Set WinDbg to Verbose mode (View, Verbose Output)
  - Doesn't work with LiveKD
- You'll see every kernel module that loads
- Kernel modules are not loaded or unloaded often
  – Any loads are suspicious

In the following example, we see that the *FileWriter.sys* driver has been loaded in the kernel debugging window. Likely, this is the malicious driver.

```
ModLoad: f7b0d000 f7b0e780   FileWriter.sys
```

NOTE

When using VMware for kernel debugging, you will see KMixer.sys *frequently loaded and unloaded. This is normal and not associated with any malicious activity.*

Kernel-Mode Code

- **!drvobj** command shows driver object

Example 11-7. Viewing a driver object for a loaded driver

```
kd> !drvobj FileWriter
Driver object (1827e3698) is for:
Loading symbols for f7b0d000    FileWriter.sys ->   FileWriter.sys
*** ERROR: Module load completed but symbols could not be loaded for
FileWriter.sys
 \Driver\FileWriter
Driver Extension List: (id , addr)

Device Object list:
826eb030
```

Kernel-Mode Code

- **dt** command shows structure

Example 11-8. Viewing a device object in the kernel

```
kd>dt nt!_DRIVER_OBJECT 0x827e3698
nt!_DRIVER_OBJECT
   +0x000 Type             : 4
   +0x002 Size             : 168
   +0x004 DeviceObject     : 0x826eb030 _DEVICE_OBJECT
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xf7b0d000
   +0x010 DriverSize       : 0x1780
   +0x014 DriverSection    : 0x828006a8
   +0x018 DriverExtension  : 0x827e3740 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\FileWriter"
   +0x024 HardwareDatabase : 0x8066ecd8 _UNICODE_STRING
"\REGISTRY\MACHINE\
                             HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xf7b0dfcd     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0xf7b0da2a     void  +0
   +0x038 MajorFunction    : [28] 0xf7b0da06     long  +0
```

Kernel-Mode Filenames

- Tracing this function, it eventually creates this file
  - \DosDevices\C:\secretfile.txt
- This is a *fully qualified object name*
  - Identifies the root device, usually \DosDevices

Finding Driver Objects

- Applications work with *devices*, not drivers
- Look at user-space application to identify the interesting *device object*
- Use *device object* in User Mode to find *driver object* in Kernel Mode
- Use **!devobj** to find out more about the *device object*
- Use **!devhandles** to find application that use the driver

Rootkits

Rootkit Basics

- Rootkits modify the internal functionality of the OS to conceal themselves
  - Hide processes, network connections, and other resources from running programs
  - Difficult for antivirus, administrators, and security analysts to discover their malicious activity
- Most rootkits modify the kernel
- Most popular method:
  - **System Service Descriptor Table (SSDT) hooking**

System Service Descriptor Table (SSDT)

- Used internally by Microsoft
  - To look up function calls into the kernel
  - Not normally used by third-party applications or drivers
- Only three ways for user space to access kernel code
  - **SYSCALL**
  - **SYSENTER**
  - **INT 0x2E**

SYSENTER

- Used by modern versions of Windows
  - Function code stored in EAX register
- More info about the three ways to call kernel code is in links Ch 10j and 10k

Example from ntdll.dll

Example 11-11. Code for `NtCreateFile` function

```
7C90D682 1mov     eax, 25h        ; NtCreateFile
7C90D687  mov     edx, 7FFE0300h
7C90D68C  call    dword ptr [edx]
7C90D68E  retn    2Ch
```

The call to `dword ptr[edx]` will go to the following instructions:

```
7c90eb8b 8bd4  mov     edx,esp
7c90eb8d 0f34  sysenter
```

- EAX set to 0x25
- Stack pointer saved in EDX
- SYSENTER is called

SSDT Table Entries

Example 11-12. Several entries of the SSDT table showing `NtCreateFile`

```
 SSDT[0x22] = 805b28bc (NtCreateaDirectoryObject)
 SSDT[0x23] = 80603be0 (NtCreateEvent)
 SSDT[0x24] = 8060be48 (NtCreateEventPair)
1SSDT[0x25] = 8056d3ca (NtCreateFile)
 SSDT[0x26] = 8056bc5c (NtCreateIoCompletion)
 SSDT[0x27] = 805ca3ca (NtCreateJobObject)
```

- Rootkit changes the values in the SSDT so rootkit code is called instead of the intended function
- 0x25 would be changed to a malicious driver's function

Hooking NtCreateFile

- Rootkit calls the original NtCreateFile, then removes files it wants to hide
  - This prevents applications from getting a handle to the file
- Hooking **`NtCreateFile`** alone won't hide a file from DIR, however

Rootkit Analysis in Practice

- Simplest way to detect SSDT hooking
  - Just look at the SSDT
  - Look for values that are unreasonable
  - In this case, *ntoskrnl.exe* starts at address 804d7000 and ends at 806cd580
  - *ntoskrnl.exe* is the Kernel!
- **lm m nt**
  - Lists modules matching "nt" (Kernel modules)
  - Shows the SSDT table (not in Win 2008 in LiveKD)

Win 2008

- **lm m nt** failed on my Win 2008 VM
- This command shows the SSDT
- **dps nt!KiServiceTable L poi nt!KiServiceLimit**
  - Link Ch 10l

```
kd> dps nt!KiServiceTable L poi nt!KiServiceLimit
824c8970  825ca949 nt!NtAcceptConnectPort
824c8974  8243701f nt!NtAccessCheck
824c8978  825fe9bd nt!NtAccessCheckAndAuditAlarm
824c897c  8243c181 nt!NtAccessCheckByType
824c8980  825fe8dd nt!NtAccessCheckByTypeAndAuditAlarm
824c8984  824f0ba0 nt!NtAccessCheckByTypeResultList
824c8988  826b1845 nt!NtAccessCheckByTypeResultListAndAuditAlarm
824c898c  826b188e nt!NtAccessCheckByTypeResultListAndAuditAlarmByHandle
824c8990  825ccba9 nt!NtAddAtom
824c8994  826c6836 nt!NtAddBootEntry
824c8998  826c7ada nt!NtAddDriverEntry
824c899c  825f48ea nt!NtAdjustGroupsToken
824c89a0  825f5885 nt!NtAdjustPrivilegesToken
824c89a4  826a5757 nt!NtAlertResumeThread
```

SSDT Table

Example 11-13. A sample SSDT table with one entry overwritten by a rootkit

```
kd> lm m nt
...
8050122c  805c9928 805c98d8 8060aea6 805aa334
8050123c  8060a4be 8059cbbc 805a4786 805cb406
8050124c  804feed0 8060b5c4 8056ae64 805343f2
8050125c  80603b90 805b09c0 805e9694 80618a56
8050126c  805edb86 80598e34 80618caa 805986e6
8050127c  805401f0 80636c9c 805b28bc 80603be0
8050128c  8060be48 1f7ad94a4 8056bc5c 805ca3ca
8050129c  805ca102 80618e86 8056d4d8 8060c240
805012ac  8056d404 8059fba6 80599202 805c5f8e
```

- Marked entry is hooked
- To identify it, examine a clean system's SSDT

Finding the Malicious Driver

- **lm**
  - Lists open modules
  - In the kernel, they are all drivers

Example 11-14. Using the `lm` *command to find which driver contains a particular address*

```
kd>lm
...
f7ac7000 f7ac8580   intelide   (deferred)
f7ac9000 f7aca700   dmload     (deferred)
f7ad9000 f7ada680   Rootkit    (deferred)
f7aed000 f7aee280   vmmouse    (deferred)
...
```

Example 11-16. Listing of the rootkit hook function

```
000104A4  mov     edi, edi
000104A6  push    ebp
000104A7  mov     ebp, esp
000104A9  push    [ebp+arg_8]
000104AC  call   1sub_10486
000104B1  test    eax, eax
000104B3  jz      short loc_104BB
000104B5  pop     ebp
000104B6  jmp     NtCreateFile
000104BB -----------------------------
000104BB                ; CODE XREF: sub_104A4+F j
000104BB  mov     eax, 0C0000034h
000104C0  pop     ebp
000104C1  retn    2Ch
```

The hook function jumps to the original `NtCreateFile` function for some requests and returns to 0xC0000034 for others. The value `0xC0000034` corresponds to `STATUS_OBJECT_NAME_NOT_FOUND`. The call at 1 contains

Interrupts

- Interrupts allow hardware to trigger software events
- Driver calls `IoConnectInterrupt` to register a handler for an interrupt code
- Specifies an Interrupt Service Routine (ISR)
  - Will be called when the interrupt code is generated
- Interrupt Descriptor Table (IDT)
  - Stores the ISR information
  - **!idt** command shows the IDT

Example 11-17. A sample IDT

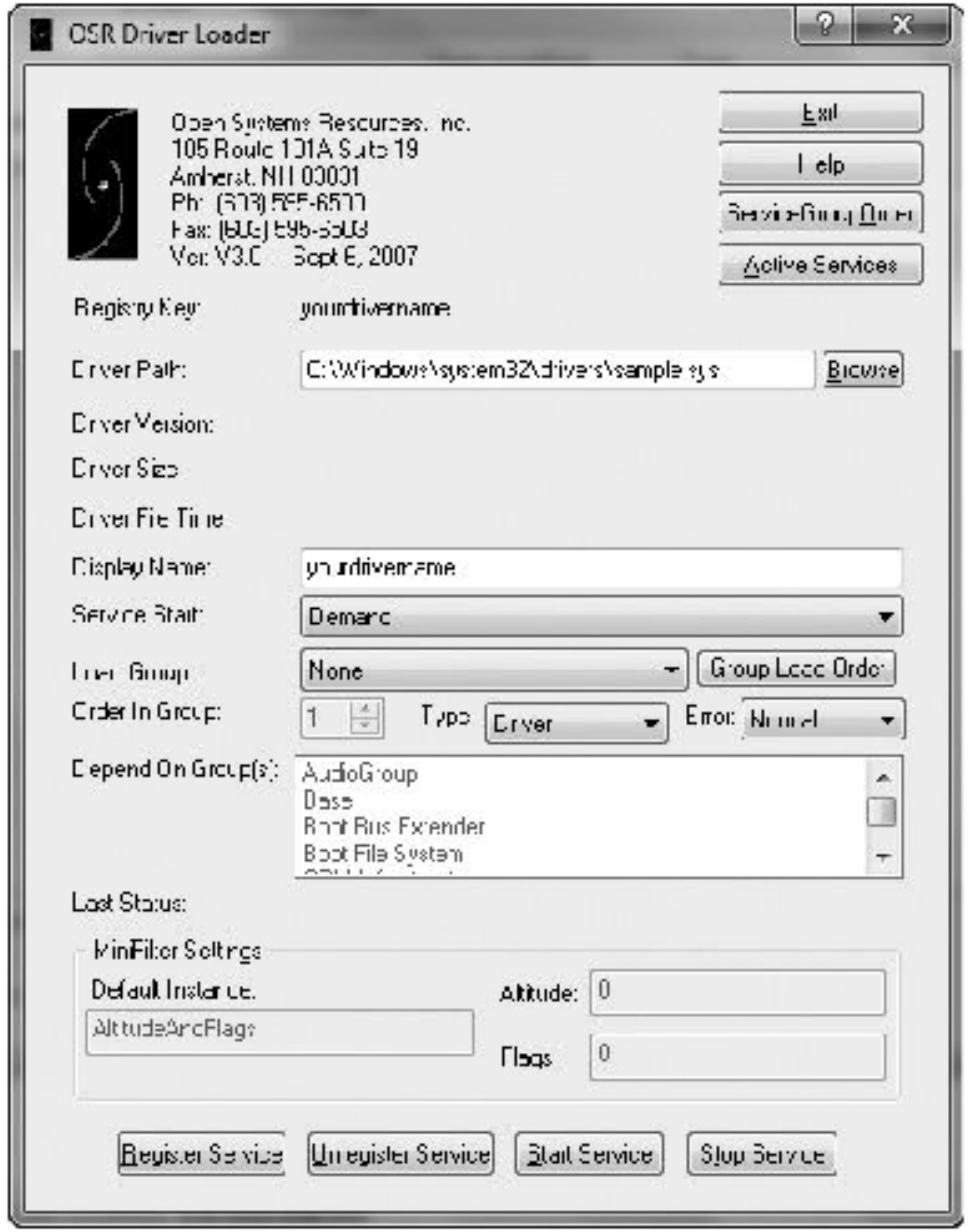
```
kd> !idt

37:    806cf728 hal!PicSpuriousService37
3d:    806d0b70 hal!HalpApcInterrupt
41:    806d09cc hal!HalpDispatchInterrupt
50:    806cf800 hal!HalpApicRebootService
62:    8298b7e4 atapi!IdePortInterrupt (KINTERRUPT 8298b7a8)
63:    826ef044 NDIS!ndisMIsr (KINTERRUPT 826ef008)
73:    826b9044 portcls!CKsShellRequestor::`vector deleting destructor'+0x26
       (KINTERRUPT 826b9008)
             USBPORT!USBPORT_InterruptService (KINTERRUPT 826df008)
82:    82970dd4 atapi!IdePortInterrupt (KINTERRUPT 82970d98)
83:    829e8044 SCSIPORT!ScsiPortInterrupt (KINTERRUPT 829e8008)
93:    826c315c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 826c3120)
a3:    826c2044 i8042prt!I8042MouseInterruptService (KINTERRUPT 826c2008)
b1:    829e5434 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 829e53f8)
b2:    826f115c serial!SerialCIsrSw (KINTERRUPT 826f1120)
c1:    806cf984 hal!HalpBroadcastCallService
d1:    806ced34 hal!HalpClockInterrupt
e1:    806cff0c hal!HalpIpiHandler
e3:    806cfc70 hal!HalpLocalApicErrorService
fd:    806d0464 hal!HalpProfileInterrupt
fe:    806d0604 hal!HalpPerfInterrupt
```

Interrupts going to unnamed, unsigned, or suspicious drivers could indicate a rootkit or other malicious software.

Loading Drivers

- If you want to load a driver to test it, you can download the OSR Driver Loader tool



Kernel Issues for Windows Vista, Windows 7, and x64 Versions

- Uses *BCDedit* instead of *boot.ini*
- x64 versions starting with XP have **PatchGuard**
  - Prevents third-party code from modifying the kernel
  - Including kernel code itself, SSDT, IDT, etc.
  - Can interfere with debugging, because debugger patches code when inserting breakpoints
- There are 64-bit kernel debugging tools
  - Link Ch 10c

Driver Signing

- Enforced in all 64-bit versions of Windows starting with Vista
- Only digitally signed drivers will load
- Effective protection!
- Kernel malware for x64 systems is practically nonexistent
  - You can disable driver signing enforcement by specifying **`nointegritychecks`** in *BCDEdit*

Kahoot!