恶意代码分析与防治技术

第9章 WinDBG内核调试

王志
zwang@nankai.edu.cn

updated on 2022-11-13

南开大学 网络空间安全学院
2022/2023

知识点

- Drivers and Kernel Code
  - 难点：Device、Driver、Physical Device
- Setting Up Kernel Debugging
- Using WinDbg
- Microsoft Symbols
- Kernel Debugging in Practice
- Rootkits
  - 难点：SSDT、IDT

Drivers and Kernel Code

WinDbg vs. OllyDbg

- What is the difference between WinDbg and OllyDbg?
- Which type of malware we should use WinDbg for analysis?

WinDbg vs. OllyDbg

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

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
- Driver creates and destroys devices, which can be accessed from user space

USB Flash Drive

- User plugs in flash drive
- Windows creates the "F: drive" *device object*
- Applications can now make requests to the F: drive
  - They will be sent to the driver for USB flash drives

Loading Drivers

- Drivers must be loaded into the kernel
  - Just as DLLs are loaded into processes
- When a driver is first loaded, its **DriverEntry** procedure is called
  - Just like **DLLMain** for DLLs

DriverEntry

- DLLs expose functionality through the export table
- Drivers must register the address for **callback functions**

DriverEntry

- They will be called when a user-space software component requests a service
- DriverEntry 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

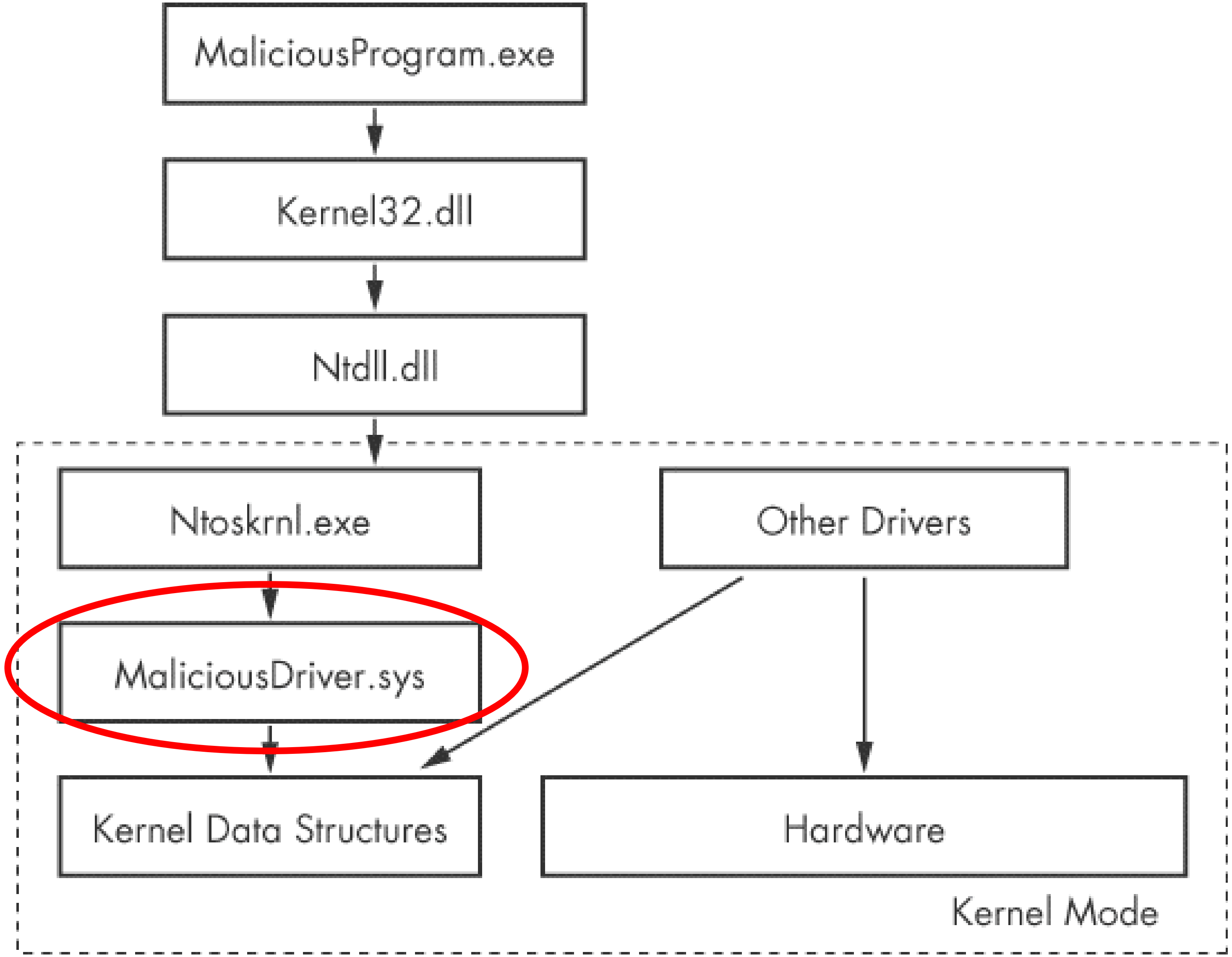


Figure 11-1. How user-mode calls are handled by the kernel

Ntoskrnl.exe & Hal.dll

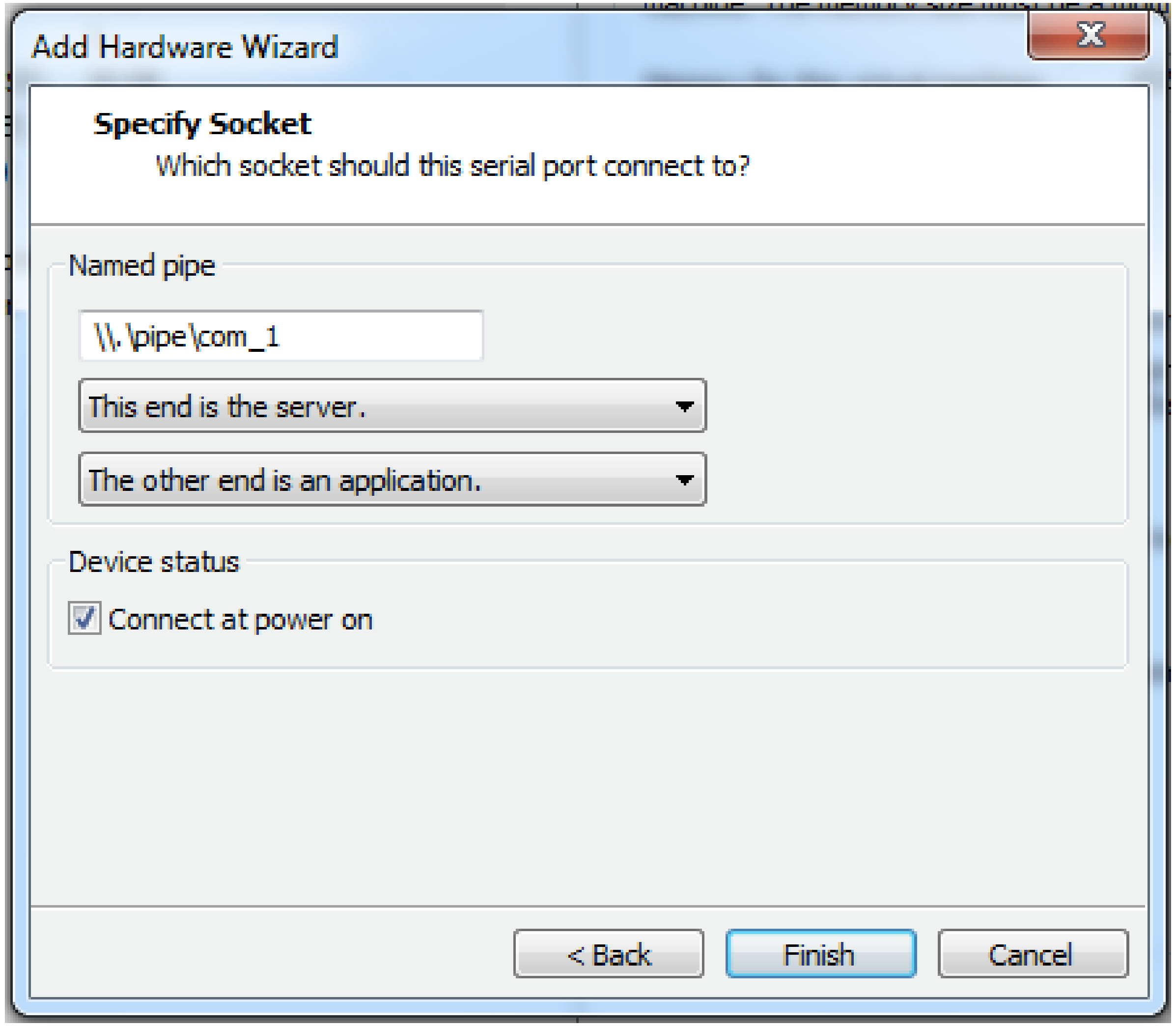
- Malicious drivers rarely control hardware
- They interact with *Ntoskrnl.exe* & *Hal.dll*
  - *Ntoskrnl.exe* has code for core OS functions
  - *Hal.dll* has code for interacting with main hardware components
- Malware will import functions from one or both of these files so it can manipulate the kernel

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

Add a Virtual Serial Port



Using WinDbg

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
  - dereference，dword from specified address
- **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

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

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)

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
```

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
- 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


msdn.microsoft.com/en-us/windows/hardware/gg463028.aspx

Download Windows Symbol Packages

The easiest way to get Windows symbols is to use the Microsoft Symbol Server. The symbol server makes symbols available to your debugging tools as needed. After a symbol file is downloaded from the symbol server it is cached on the local computer for quick access.

If you prefer to download the entire set of symbols for Windows 8.1 Preview, Windows Server 2012 R2 Preview, Windows 8, Windows Server 2012, Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, or Windows 2000, then you can download a symbol package and install it on your computer.

Kernel Debugging in Practice

Kernel Mode and User Mode Functions

- We'll examine a program that writes to files from kernel space
  - 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   11                   ; 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:DeviceIoControl1
```

Kernel-Mode Code

- Set kernel-mode debugger to Verbose mode
- You'll see every kernel module that loads
- Kernel modules are not loaded or unloaded often
  - **Any loads are suspicious**
  - Except **Kmixer.sys** in VMware machines

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
```

Kernel-Mode Code

- **!drvobj** command shows driver object

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

```
kd> !drvobj FileWriter
Driver object (827e3698) 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 *driver 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

Rootkit Basics

- Most rootkits modify the OS 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

SSDT

- 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

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` won't hide a file, 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!

Rootkit Analysis in Practice

- **lm m nt**
  - Lists modules matching "nt" (Kernel modules)
  - Shows the SSDT table

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

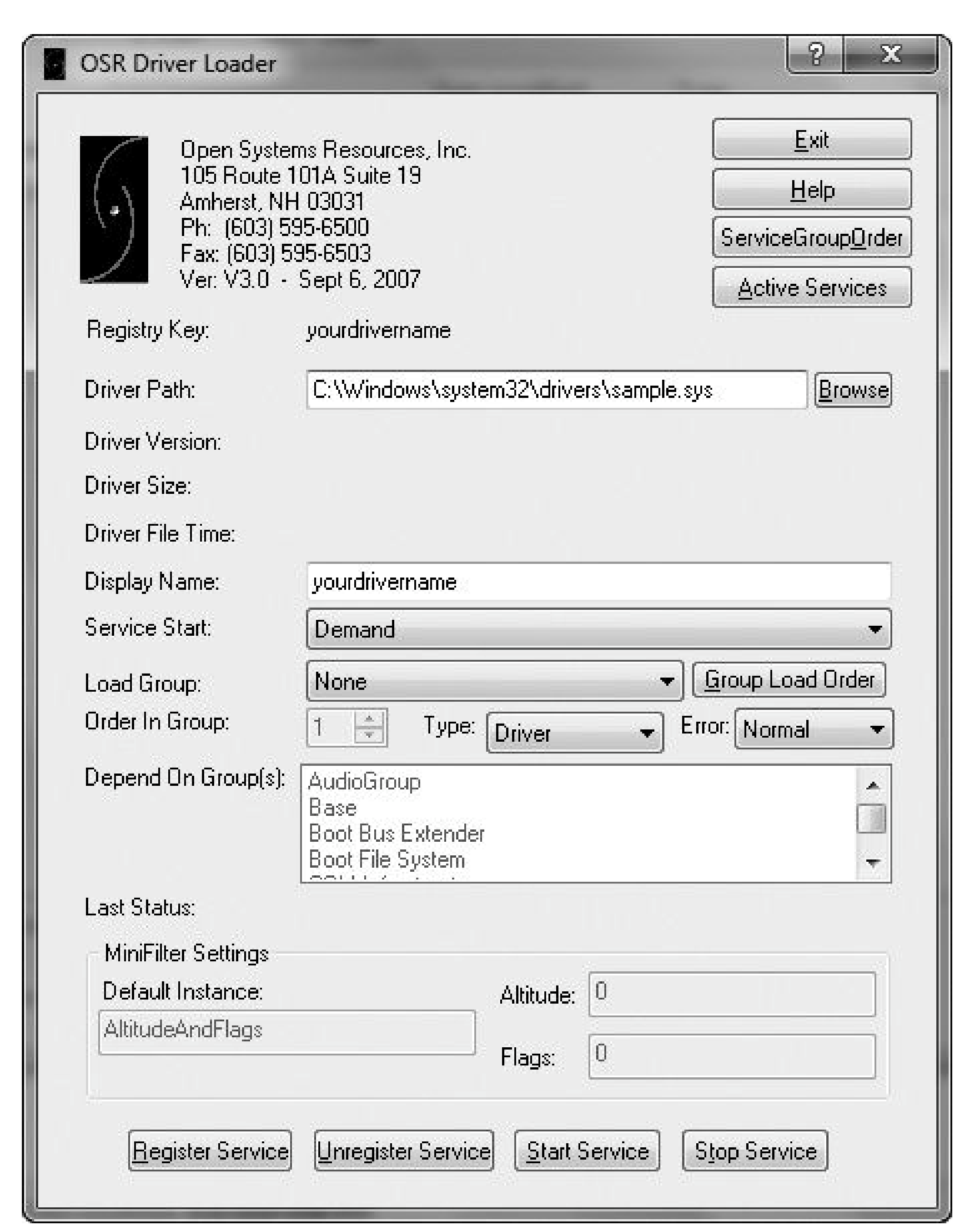
```
kd> !idt

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



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*

知识点

- Drivers and Kernel Code
  - 难点：Device、Device Driver、Physical Device
- Setting Up Kernel Debugging
- Using WinDbg
- Microsoft Symbols
- Kernel Debugging in Practice
- Rootkits
  - 难点：SSDT、IDT

恶意代码分析与防治技术

第9章 WinDBG内核调试

王志
zwang@nankai.edu.cn

updated on 2022-11-13

南开大学 网络空间安全学院
2022/2023