Malware Functionality

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
  - 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: device object

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 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

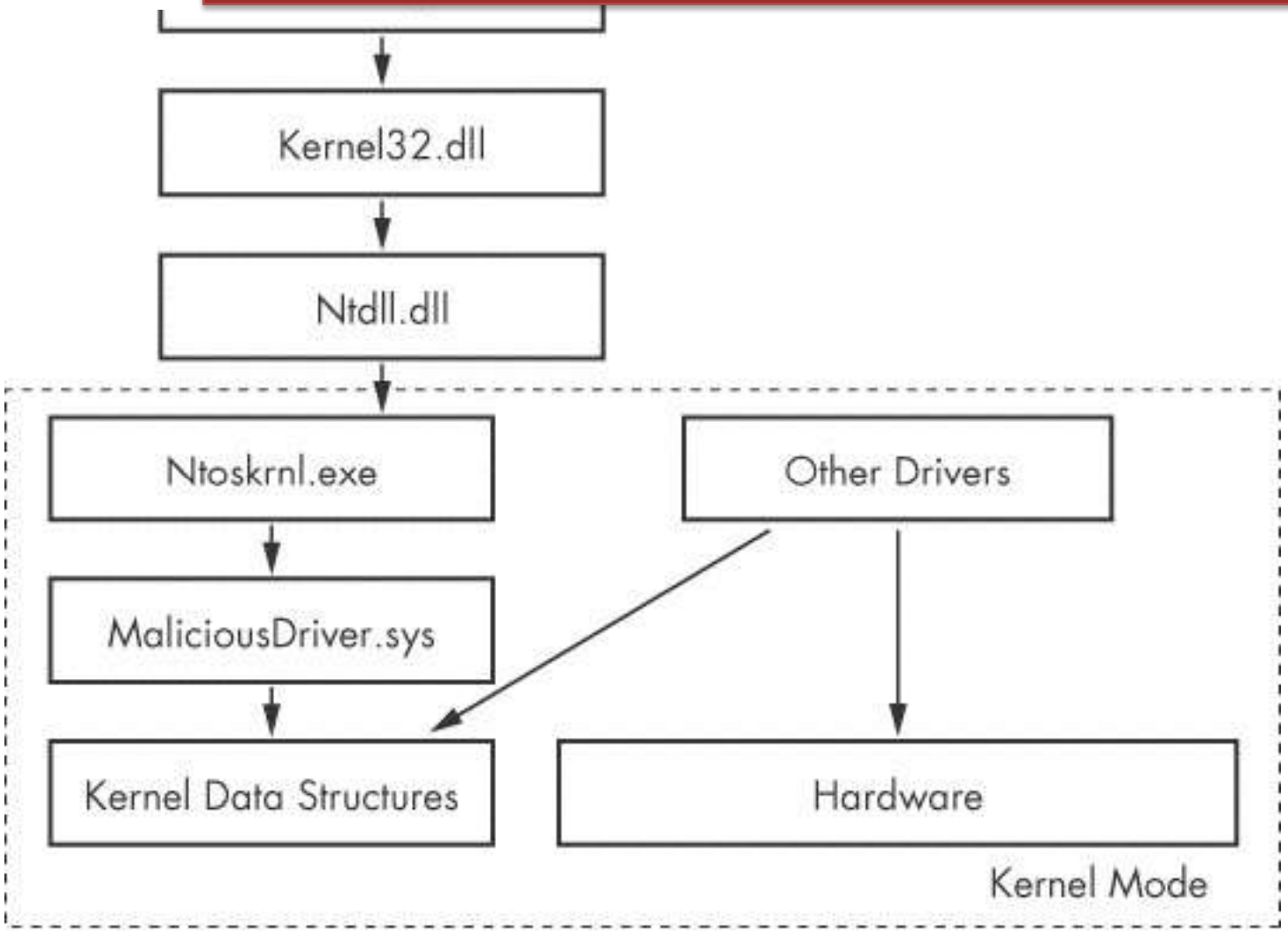


Figure 11-1. How user-mode calls are handled by the kernel

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

Setting Up Kernel Debugging

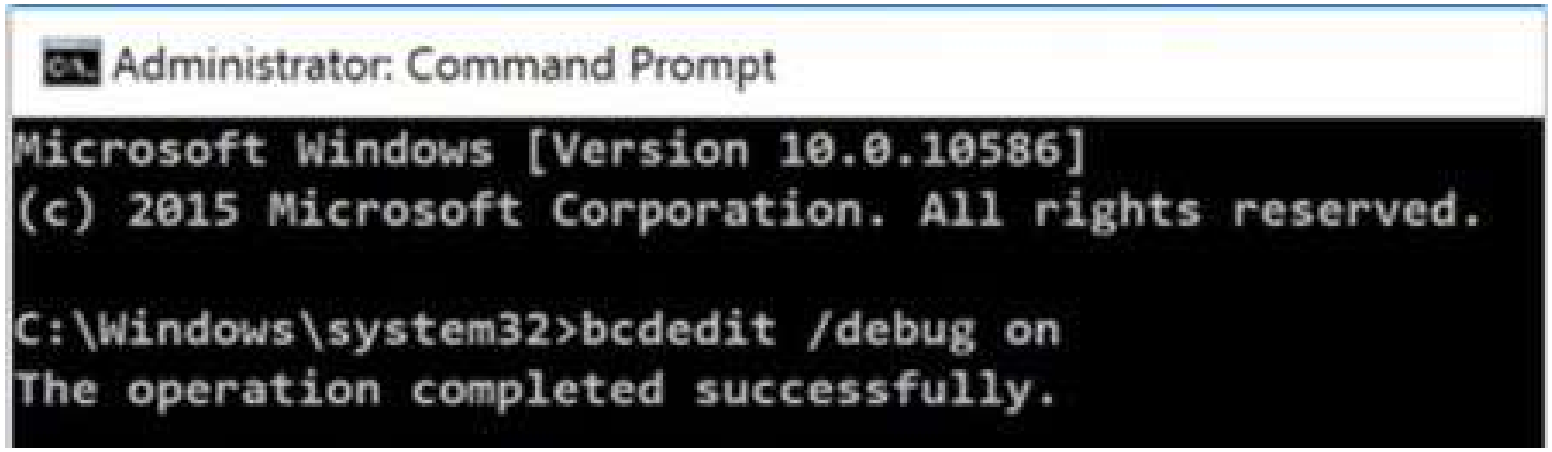
VMware

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

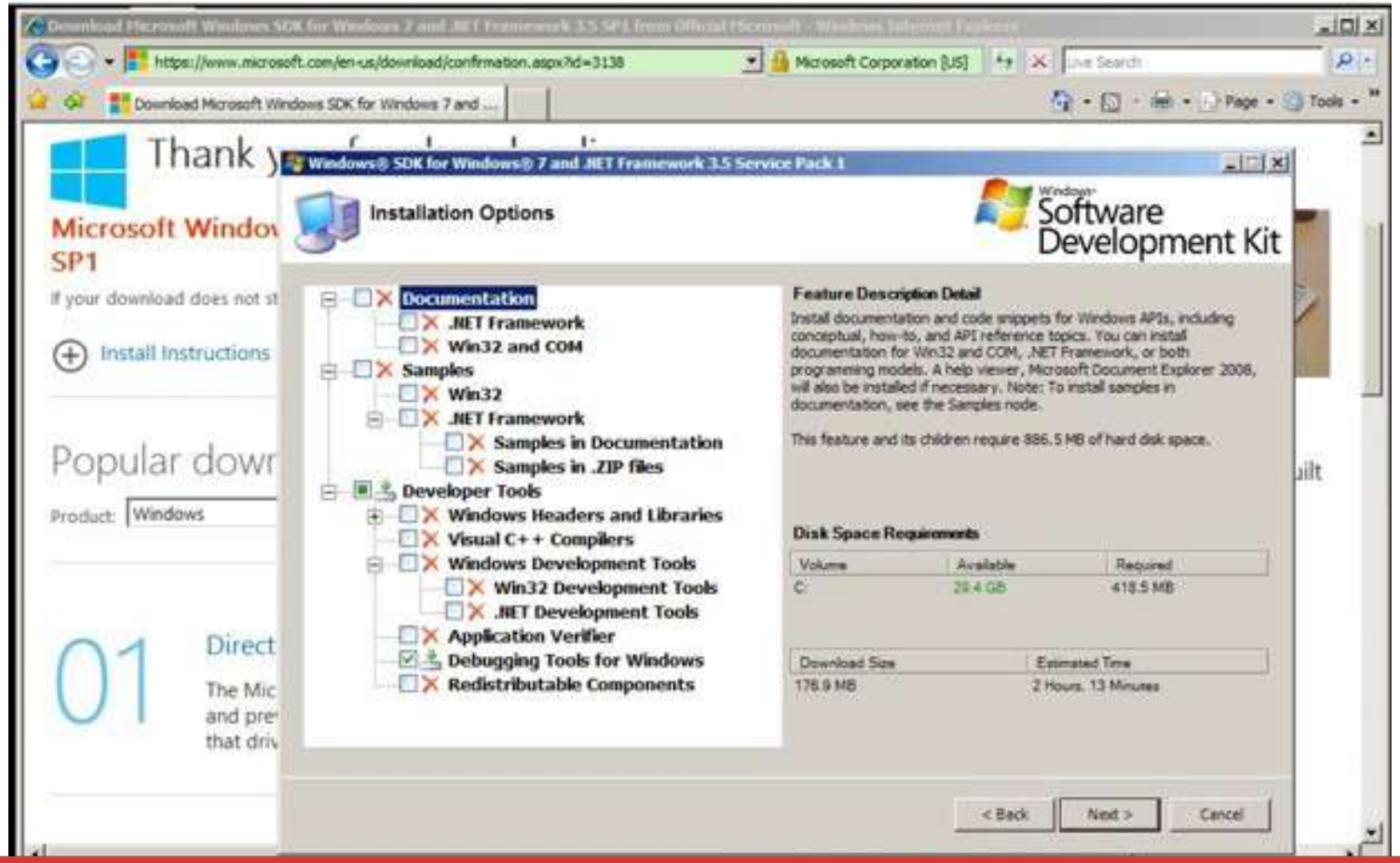
Boot.ini

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

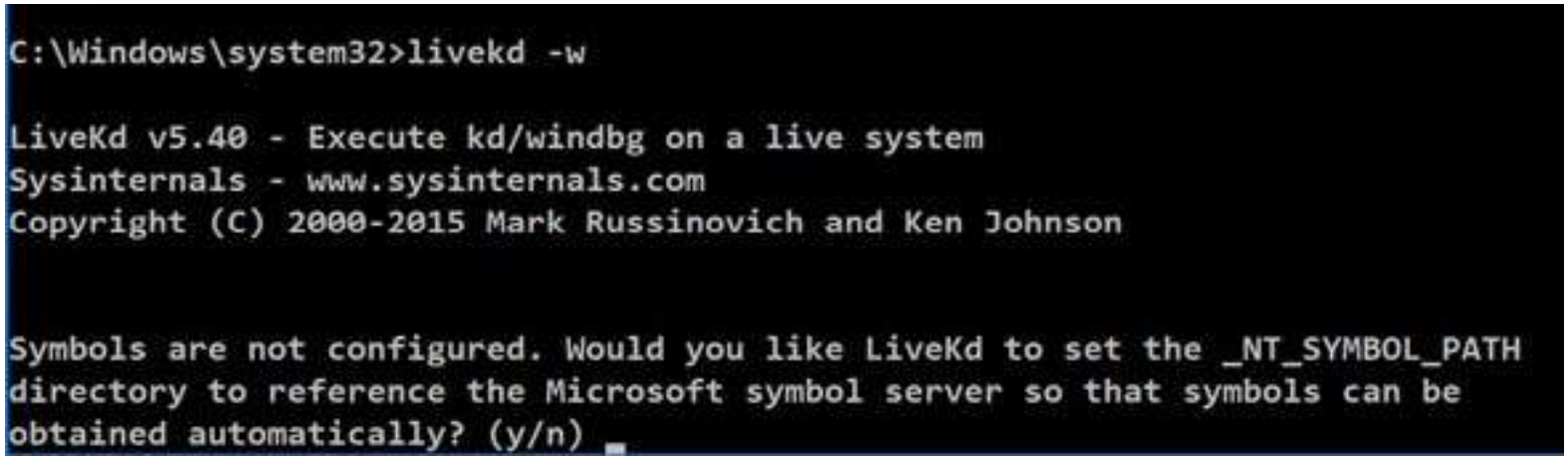
bcdedit

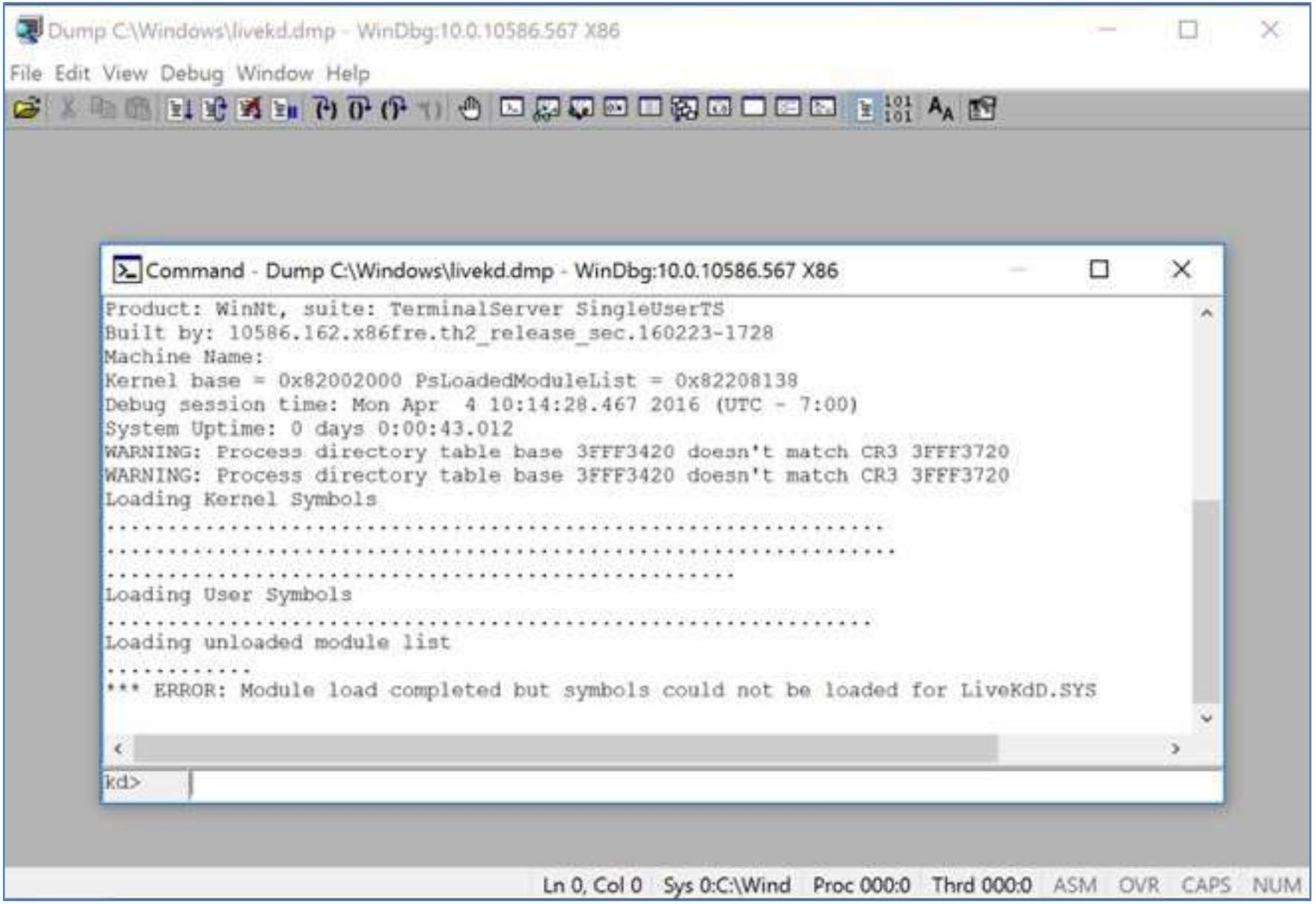

Administrator: Command Prompt
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.
C:\Windows\system32>bcdedit /debug on
The operation completed successfully.

Get WinDbg


https://www.microsoft.com/en-us/download/confirmation.aspx?id=3138
Microsoft Corporation [US]
Download Microsoft Windows SDK for Windows 7 and ...
Page
Tools
Thank y
Microsoft Windov
SP1
If your download does not st
Install Instructions
Popular downr
Product:
Windows
01
Direct
The Mic
and pre
that driv
Windows® SDK for Windows® 7 and .NET Framework 3.5 Service Pack 1
Installation Options
Windows Software Development Kit
Documentation
.NET Framework
Win32 and COM
Samples
Win32
.NET Framework
Samples in Documentation
Samples in .ZIP files
Developer Tools
Windows Headers and Libraries
Visual C++ Compilers
Windows Development Tools
Win32 Development Tools
.NET Development Tools
Application Verifier
Debugging Tools for Windows
Redistributable Components
Feature Description Detail
Install documentation and code snippets for Windows APIs, including conceptual, how-to, and API reference topics. You can install documentation for Win32 and COM, .NET Framework, or both programming models. A help viewer, Microsoft Document Explorer 2008, will also be installed if necessary. Note: To install samples in documentation, see the Samples node.
This feature and its children require 886.5 MB of hard disk space.
Disk Space Requirements
Volume
Available
Required
C:
28.4 GB
418.5 MB
Download Size
Estimated Time
176.9 MB
2 Hours, 13 Minutes
< Back
Next >
Cancel

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)


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

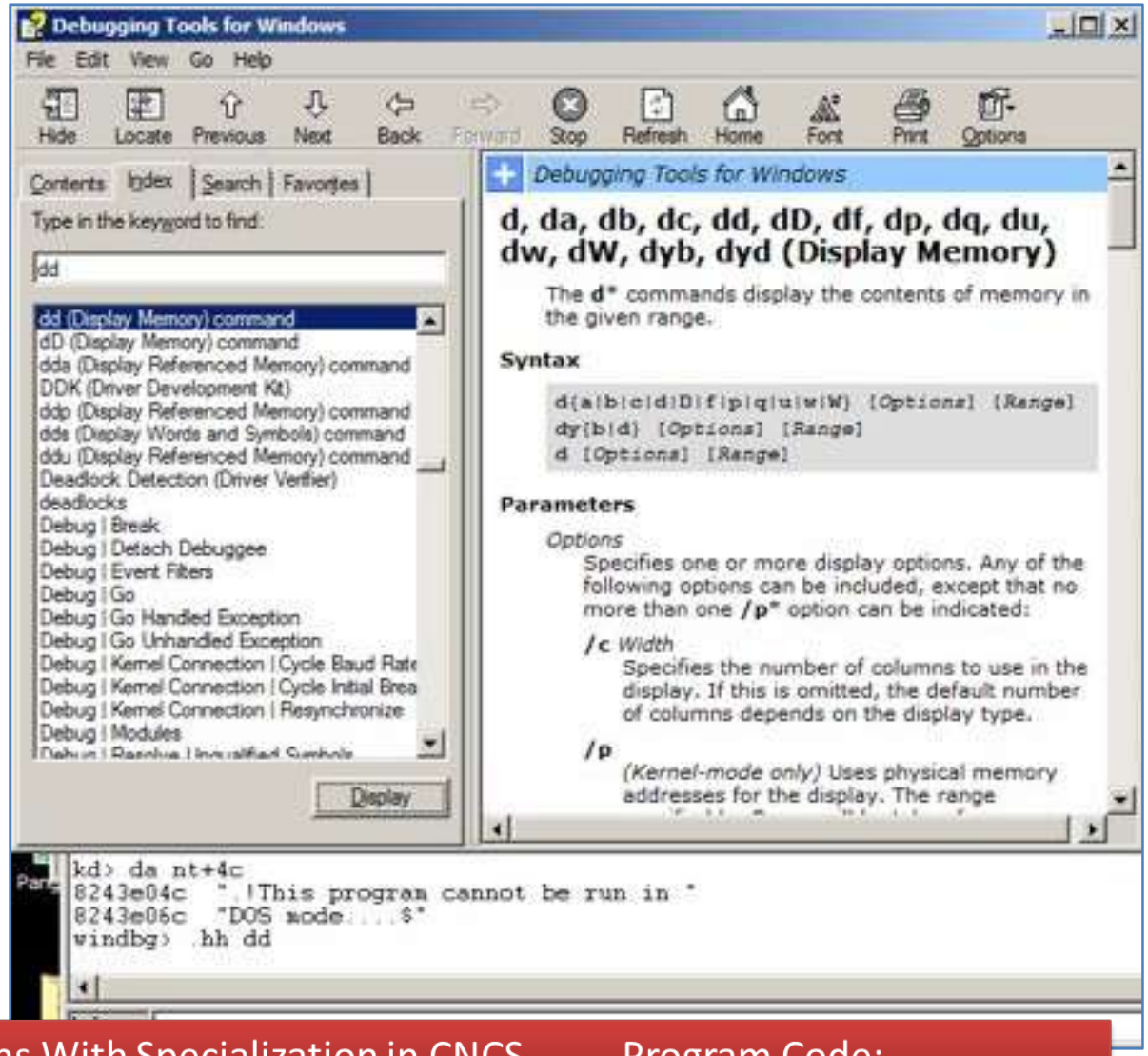
Reading from Memory

- **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



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>
```

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
```

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       :  0xf7adb66c      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

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                   ; 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

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 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 (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 *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 mov     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)
SSDT[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

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
```

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

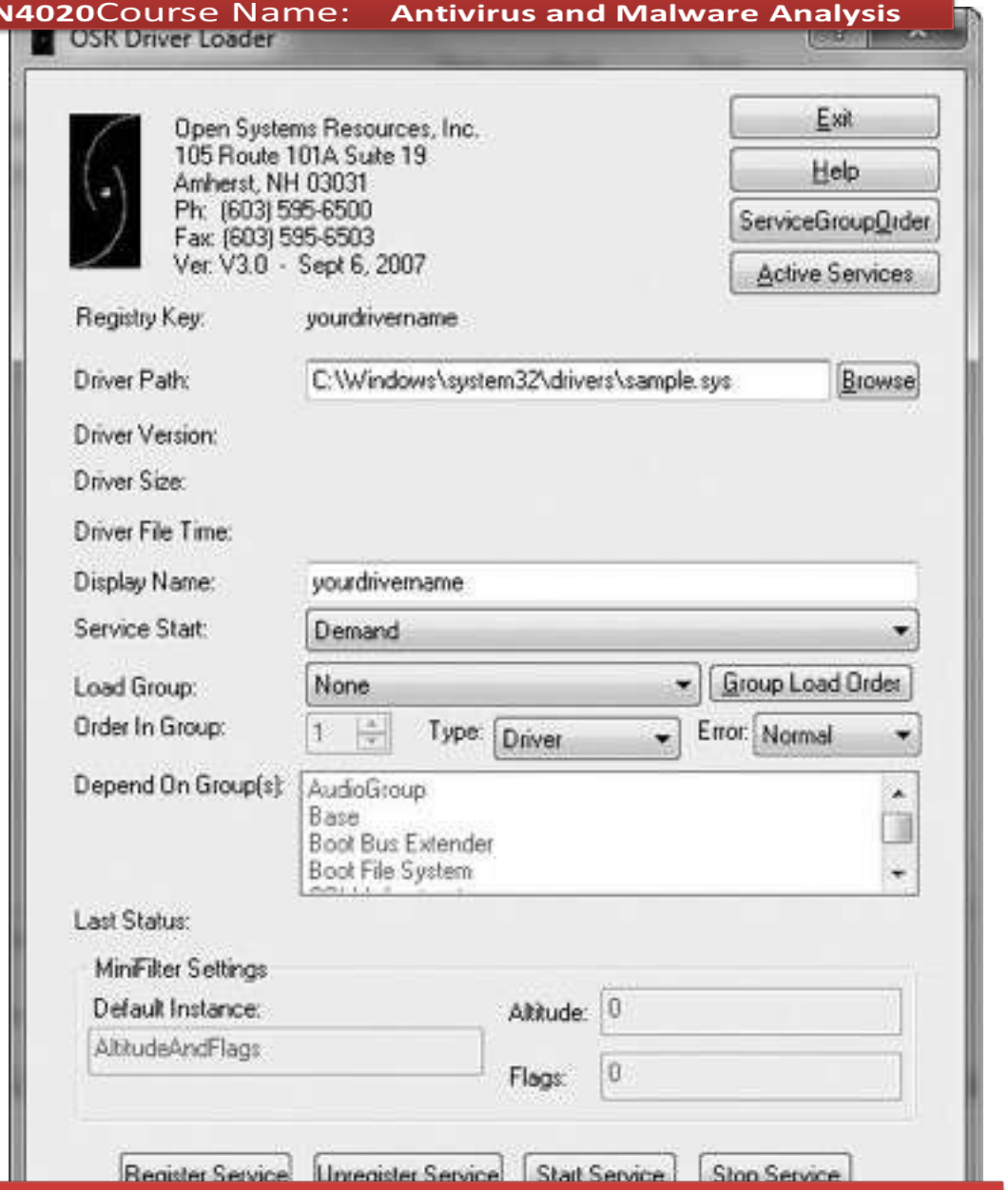
```
kd> !idt

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

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*

Downloaders and Launchers

Downloaders

- Download another piece of malware
  - And execute it on the local system
- Commonly use the Windows API **`URLDownloadtoFileA`**, followed by a call to **`WinExec`**

Launchers (aka Loaders)

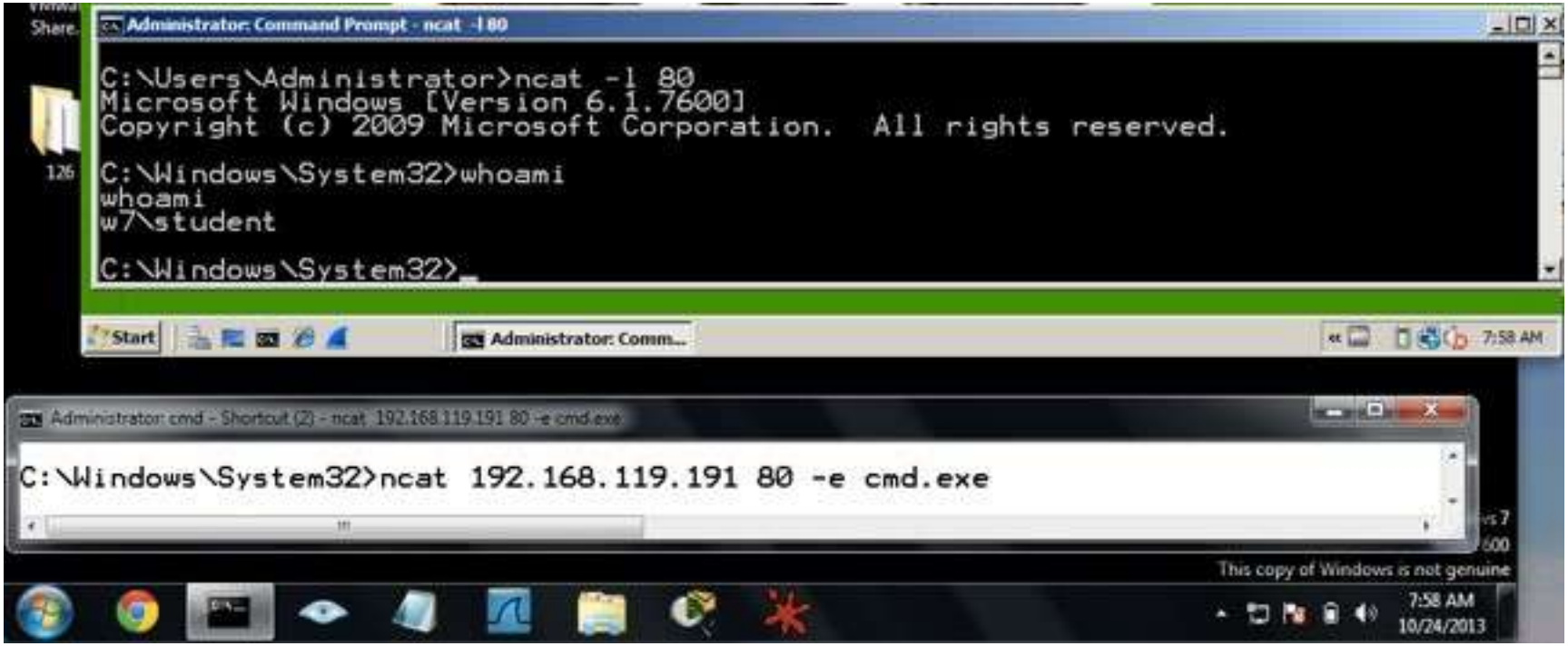
- Prepares another piece of malware for covert execution
  - Either immediately or later
  - Stores malware in unexpected places, such as the **.rsrc** section of a PE file

Backdoors

- Provide remote access to victim machine
- The most common type of malware
- Often communicate over HTTP on Port 80
  - Network signatures are helpful for detection
- Common capabilities
  - Manipulate Registry, enumerate display windows, create directories, search files, etc.

Reverse Shell

- Infected machine calls out to attacker, asking for commands to execute



Windows Reverse Shells

- Basic
  - Call **`CreateProcess`** and manipulate STARTUPINFO structure
  - Create a socket to remote machine
  - Then tie socket to standard input, output, and error for cmd.exe
  - **`CreateProcess`** runs cmd.exe with its window suppressed, to hide it

Windows Reverse Shells

- Multithreaded
  - Create a socket, two pipes, and two threads
  - Look for API calls to **`CreateThread`** and **`CreatePipe`**
  - One thread for stdin, one for stdout

RATs
(Remote Administration Tools)

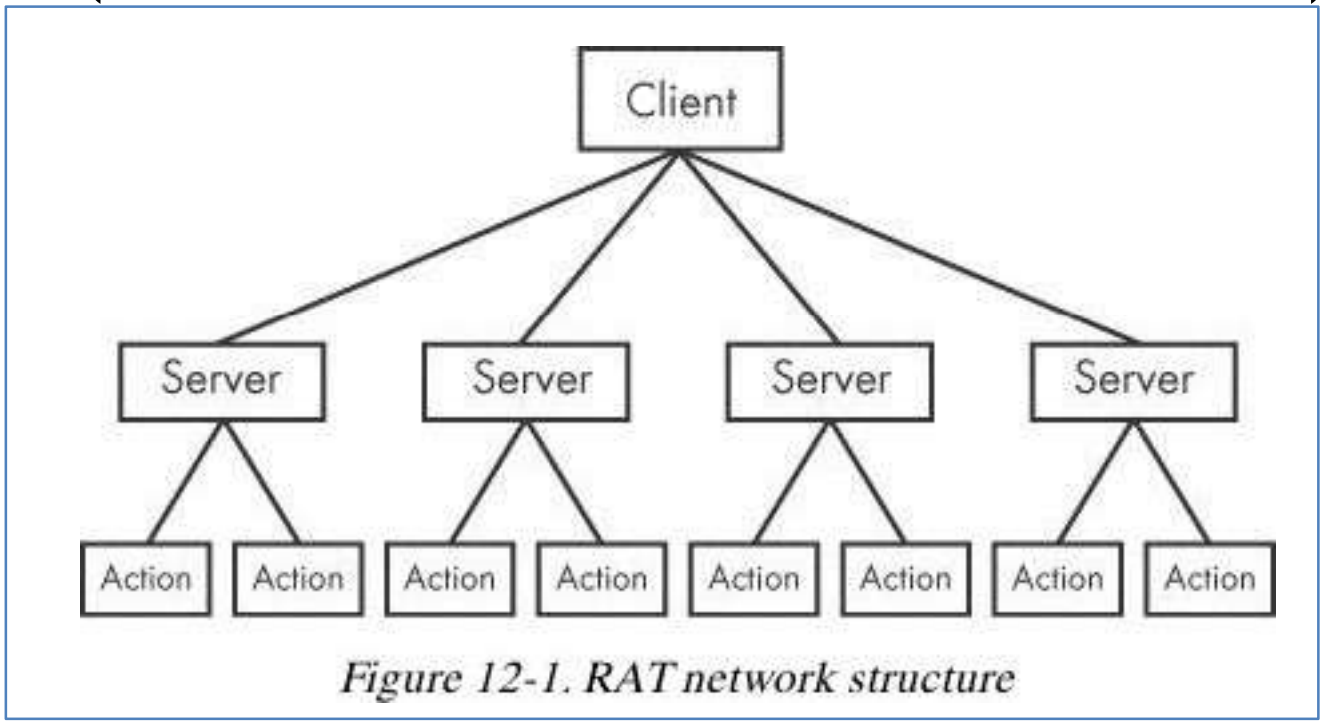


Figure 12-1. RAT network structure

- Ex: Poison Ivy

Botnets

- A collection of compromised hosts
  - Called *bots* or *zombies*

Botnets v. RATs

- Botnet contain many hosts; RATs control fewer hosts
- All bots are controlled at once; RATs control victims one by one
- RATs are for targeted attacks; botnets are used in mass attacks

Credential Stealers

- Three types
  - Wait for user to log in and steal credentials
  - Dump stored data, such as password hashes
  - Log keystrokes

GINA Interception

- Windows XP's Graphical Identification and Authentication (GINA)
  - Intended to allow third parties to customize logon process for RFID or smart cards
  - Intercepted by malware to steal credentials
- GINA is implemented in **msgina.dll**
  - Loaded by WinLogon executable during logon
- WinLogon also loads third-party customizations in DLLs loaded between WinLogon and GINA

GINA Registry Key

- HKLM\SOFTWARE\Microsoft\Windows NT \CurrentVersion\Winlogon\GinaDLL
- Contains third-party DLLs to be loaded by WinLogon

Figure 12-2. Malicious fsgina.dll sits in between the Windows system files to capture data.

MITM Attack

- Malicious DLL must export all functions the real *msgina.dll* does, to act as a MITM
  - More than 15 functions
  - Most start with **`Wlx`**
  - Good indicator
  - Malware DLL exporting a lot of **`Wlx`** functions is probably a GINA interceptor

WlxLoggedOutSAS

- Most exports simply call through to the real functions in *msgina.dll*
- At 2, the malware logs the credentials to the file %SystemRoot%\system32\drivers\tcpudp.sys

Example 12-1. GINA DLL `WlxLoggedOutSAS` export function for logging stolen credentials

```
100014A0 WlxLoggedOutSAS
100014A0         push    esi
100014A1         push    edi
100014A2         push    offset aWlxloggedout_0 ; "WlxLoggedOutSAS"
100014A7         call    Call_msgina_dll_function 1
...
100014FB         push    eax ; Args
100014FC         push    offset aUSDSPSOpS ;"U: %s D: %s P: %s OP: %s"
10001501         push    offset aDRIVERS ; "drivers\tcpudp.sys"
10001503         call    Log_To_File 2
```

GINA is Gone

- No longer used in Windows Vista and later
- Replaced by Credential Providers
  - Link Ch 11c

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
\Authentication\Credential Providers\{ACFC407B-266C-4085-8DAE-
F3E276336E4B}]
@="SampleWrapExistingCredentialProvider"

[HKEY_CLASSES_ROOT\CLSID\{ACFC407B-266C-4085-8DAE-F3E276336E4B}]
@="SampleWrapExistingCredentialProvider"

[HKEY_CLASSES_ROOT\CLSID\{ACFC407B-266C-4085-8DAE-F3E276336E4B}
\InprocServer32]
@="SampleWrapExistingCredentialProvider.dll"
"ThreadingModel"="Apartment"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
\Authentication\Credential Provider Filters\
{ACFC407B-266C-4085-8DAE-F3E276336E4B}]
@="SampleWrapExistingCredentialProvider"
```

Custom Credential Provider Rootkit on Windows 7

- Two sets of login buttons
- Only steals passwords from second set
- Code is provided to filter out the original set

Hash Dumping

- Windows login passwords are stored as LM or NTLM hashes
  - —Hashes can be used directly to authenticate (pass-the-hash attack)
  - – Or cracked offline to find passwords
- Pwdump and Pass-the-Hash Toolkit
  - – Free hacking tools that provide hash dumping
  - – Open-source
  - – Code re-used in malware
  - – Modified to bypass antivirus

- Injects a DLL into LSASS (Local Security Authority Subsystem Service)
  - To get hashes from the SAM (Security Account Manager)
  - Injected DLL runs inside another process
  - Gets all the privileges of that process
  - LSASS is a common target
    - High privileges
    - Access to many useful API functions

- Injects *lsaext.dll* into *lsass.exe*
  - Calls **`GetHash`**, an export of *lsaext.dll*
  - Hash extraction uses undocumented Windows function calls
- Attackers may change the name of the **`GetHash`** function

Pwdump Variant

- Uses these libraries
  - *samsrv.dll* to access the SAM
  - *advapi32.dll* to access functions not already imported into *lsass.exe*
  - Several **`Sam`** functions
  - Hashes extracted by **`SamIGetPrivateData`**
  - Decrypted with **`SystemFunction025`** and **`SystemFunction027`**
- All undocumented functions

Example 12-2. Unique API calls used by a pwdump variant's export function `GrabHash`

```
1000123F        push    offset LibFileName      ; "samsrv.dll" 1
10001244        call    esi ; LoadLibraryA
10001248        push    offset aAdvapi32_dll_0  ; "advapi32.dll" 2
...
10001251        call    esi ; LoadLibraryA
...
1000125B        push    offset ProcName         ; "SamIConnect"
10001260        push    ebx                     ; hModule
10001265        call    esi ; GetProcAddress
...
10001281        push    offset aSamrqu ; "SamrQueryInformationUser"
10001286        push    ebx                     ; hModule
1000128C        call    esi ; GetProcAddress
...
100012C2        push    offset aSamigetpriv ; "SamIGetPrivateData"
100012C7        push    ebx                     ; hModule
100012CD        call    esi ; GetProcAddress
...
100012CF        push    offset aSystemfuncti  ; "SystemFunction025" 3
100012D4        push    edi                     ; hModule
100012DA        call    esi ; GetProcAddress
100012DC        push    offset aSystemfuni_0  ; "SystemFunction027" 4
100012E1        push    edi                     ; hModule
100012E7        call    esi ; GetProcAddress
```

Pass-the-Hash Toolkit

- Injects a DLL into *lsass.exe* to get hashes
  - Program named **whosthere-alt**
- Uses different API functions than Pwdump

Example 12-3. Unique API calls used by a whosthere-alt variant's export function TestDump

```
10001119        push    offset LibFileName ; "secur32.dll"
1000111E        call    ds:LoadLibraryA
10001130        push    offset ProcName ; "LsaEnumerateLogonSessions"
10001135        push    esi             ; hModule
10001136        call    ds:GetProcAddress 1
...
10001670        call    ds:GetSystemDirectoryA
10001676        mov     edi, offset aMsv1_0_dll ; \\msv1_0.dll
...
100016A6        push    eax             ; path to msv1_0.dll
100016A9        call    ds:GetModuleHandleA 2
```

Keystroke Logging

- Kernel-Based Keyloggers
  - Difficult to detect with user-mode applications
  - Frequently part of a rootkit
  - Act as keyboard drivers
  - Bypass user-space programs and protections

Keystroke Logging

- User-Space Keyloggers
  - Use Windows API
  - Implemented with *hooking* or *polling*
- Hooking
  - Uses **`SetWindowsHookEx`** function to notify malware each time a key is pressed
  - Details in next chapter
- Polling
  - Uses **`GetAsyncKeyState`** & **`GetForegroundWindow`** to constantly poll the state of the keys

Polling Keyloggers

- **`GetAsyncKeyState`**
  - Identifies whether a key is pressed or unpressed
- **`GetForegroundWindow`**
  - Identifies the foreground window
- Loops through all keys, then sleeps briefly
- Repeats frequently enough to capture all keystrokes

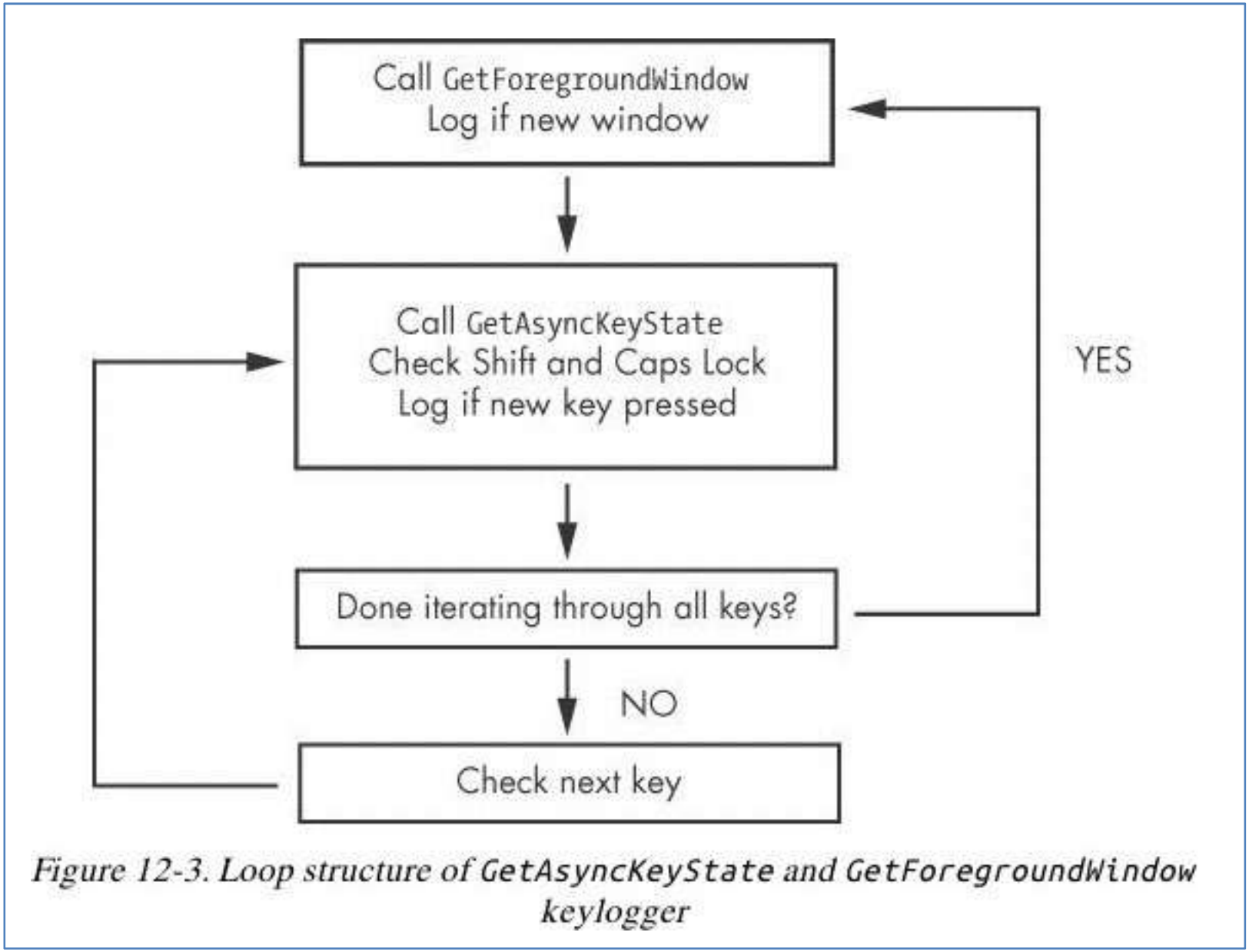


Figure 12-3. Loop structure of `GetAsyncKeyState` and `GetForegroundWindow` keylogger

Identifying Keyloggers in Strings Listings

- Run Strings
- Terms like these will be visible

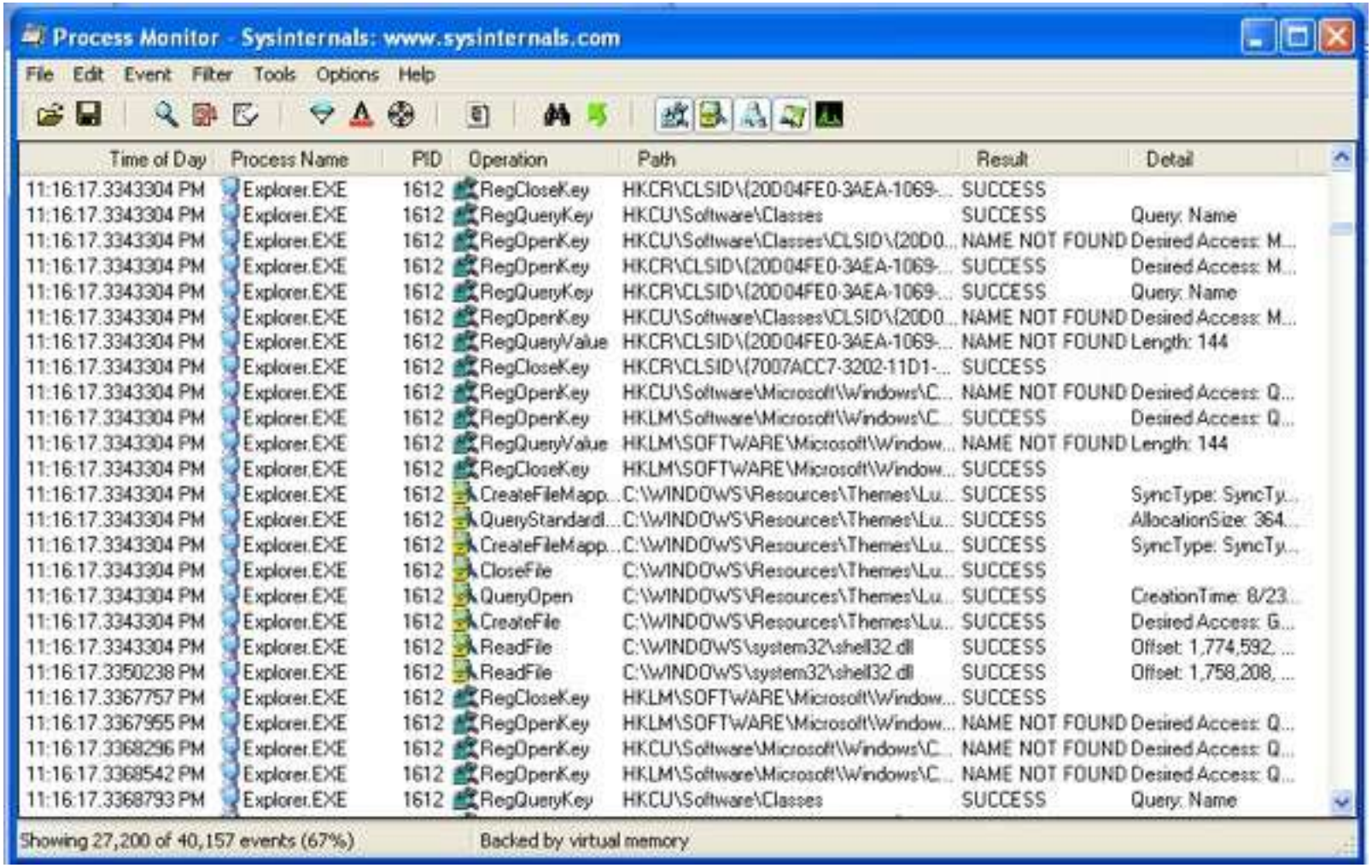
```
[Up]
[Num Lock]
[Down]
[Right]
[UP]
[Left]
[PageDown]
```

Three Persistence Mechanisms

1. Registry modifications, such as Run key
   - Other important registry entries:
     - AppInit_DLLs
     - Winlogon Notify
     - ScvHost DLLs
2. Trojanizing Binaries
3. DLL Load-Order Hijacking

Registry Modifications

- Run key
  - HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ Windows\ CurrentVersion\ Run
  - Many others, as revealed by Autoruns
- ProcMon shows all registry modifications when running malware (dynamic analysis)
  - Can detect all these techniques


Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help
Time of Day Process Name PID Operation Path Result Detail
11:16:17.3343304 PM Explorer.EXE 1612 RegCloseKey HKCR\CLSID\{20D04FE0-3AEA-1069-... SUCCESS
11:16:17.3343304 PM Explorer.EXE 1612 RegQueryKey HKCU\Software\Classes SUCCESS Query: Name
11:16:17.3343304 PM Explorer.EXE 1612 RegOpenKey HKCU\Software\Classes\CLSID\{20D0... NAME NOT FOUND Desired Access: M...
11:16:17.3343304 PM Explorer.EXE 1612 RegOpenKey HKCR\CLSID\{20D04FE0-3AEA-1069-... SUCCESS Desired Access: M...
11:16:17.3343304 PM Explorer.EXE 1612 RegQueryKey HKCR\CLSID\{20D04FE0-3AEA-1069-... SUCCESS Query: Name
11:16:17.3343304 PM Explorer.EXE 1612 RegOpenKey HKCU\Software\Classes\CLSID\{20D0... NAME NOT FOUND Desired Access: M...
11:16:17.3343304 PM Explorer.EXE 1612 RegQueryValue HKCR\CLSID\{20D04FE0-3AEA-1069-... NAME NOT FOUND Length: 144
11:16:17.3343304 PM Explorer.EXE 1612 RegCloseKey HKCR\CLSID\{7007ACC7-3202-11D1-... SUCCESS
11:16:17.3343304 PM Explorer.EXE 1612 RegOpenKey HKCU\Software\Microsoft\Windows\C... NAME NOT FOUND Desired Access: Q...
11:16:17.3343304 PM Explorer.EXE 1612 RegOpenKey HKLM\Software\Microsoft\Windows\C... SUCCESS Desired Access: Q...
11:16:17.3343304 PM Explorer.EXE 1612 RegQueryValue HKLM\SOFTWARE\Microsoft\Window... NAME NOT FOUND Length: 144
11:16:17.3343304 PM Explorer.EXE 1612 RegCloseKey HKLM\SOFTWARE\Microsoft\Window... SUCCESS
11:16:17.3343304 PM Explorer.EXE 1612 CreateFileMapp... C:\WINDOWS\Resources\Themes\Lu... SUCCESS SyncType: SyncTy...
11:16:17.3343304 PM Explorer.EXE 1612 QueryStandardI... C:\WINDOWS\Resources\Themes\Lu... SUCCESS AllocationSize: 364...
11:16:17.3343304 PM Explorer.EXE 1612 CreateFileMapp... C:\WINDOWS\Resources\Themes\Lu... SUCCESS SyncType: SyncTy...
11:16:17.3343304 PM Explorer.EXE 1612 CloseFile C:\WINDOWS\Resources\Themes\Lu... SUCCESS
11:16:17.3343304 PM Explorer.EXE 1612 QueryOpen C:\WINDOWS\Resources\Themes\Lu... SUCCESS CreationTime: 8/23...
11:16:17.3343304 PM Explorer.EXE 1612 CreateFile C:\WINDOWS\Resources\Themes\Lu... SUCCESS Desired Access: G...
11:16:17.3343304 PM Explorer.EXE 1612 ReadFile C:\WINDOWS\system32\shell32.dll SUCCESS Offset: 1,774,592, ...
11:16:17.3350238 PM Explorer.EXE 1612 ReadFile C:\WINDOWS\system32\shell32.dll SUCCESS Offset: 1,758,208, ...
11:16:17.3367757 PM Explorer.EXE 1612 RegCloseKey HKLM\SOFTWARE\Microsoft\Window... SUCCESS
11:16:17.3367955 PM Explorer.EXE 1612 RegOpenKey HKLM\SOFTWARE\Microsoft\Window... NAME NOT FOUND Desired Access: Q...
11:16:17.3368296 PM Explorer.EXE 1612 RegOpenKey HKCU\Software\Microsoft\Windows\C... NAME NOT FOUND Desired Access: Q...
11:16:17.3368542 PM Explorer.EXE 1612 RegOpenKey HKLM\Software\Microsoft\Windows\C... NAME NOT FOUND Desired Access: Q...
11:16:17.3368793 PM Explorer.EXE 1612 RegQueryKey HKCU\Software\Classes SUCCESS Query: Name
Showing 27,200 of 40,157 events (67%)
Backed by virtual memory

APPINIT DLLS

- AppInit_DLLs are loaded into every process that loads User32.dll
  - This registry key contains a space-delimited list of DLLs
  - HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ Windows NT\ CurrentVersion\ Windows
  - Many processes load them
  - Malware will call DLLMain to check which process it is in before launching payload

Winlogon Notify

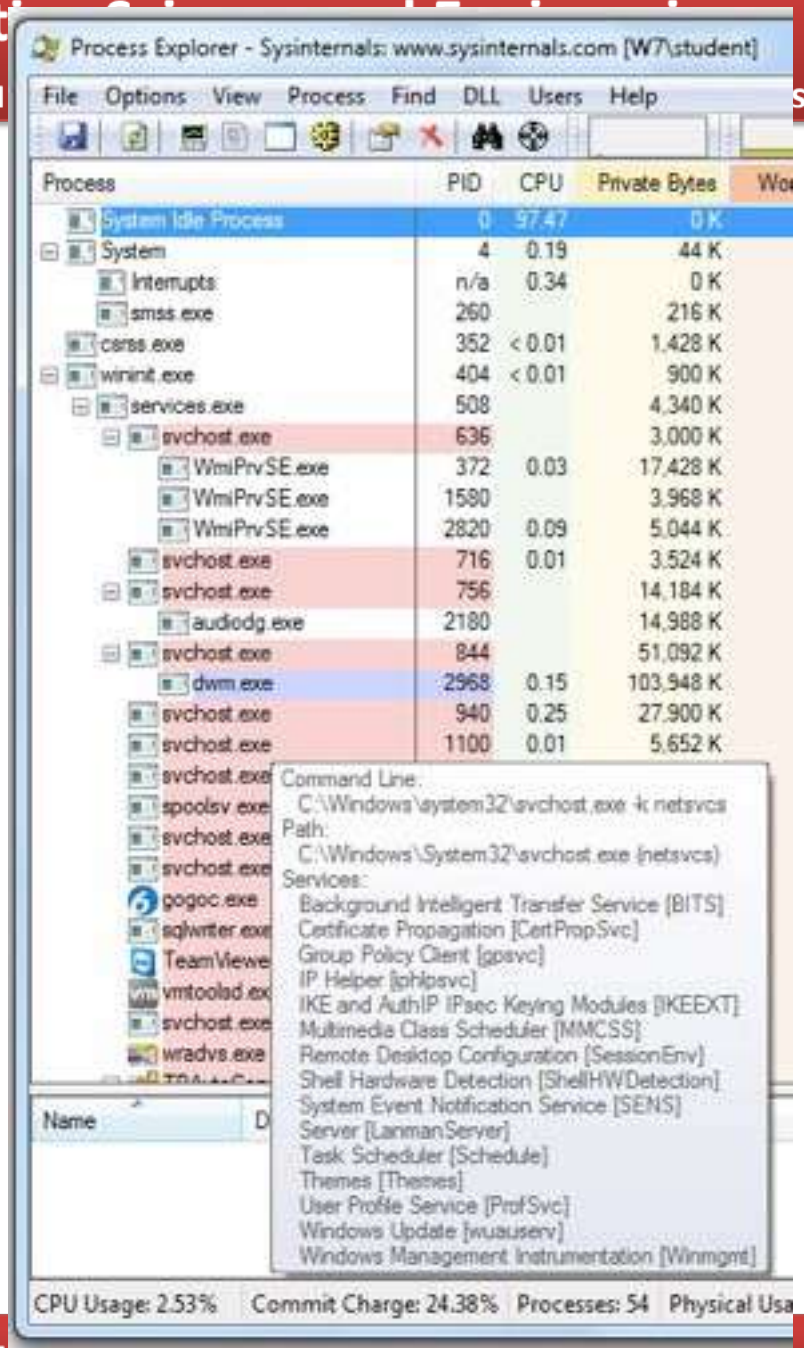
- Notify value in
  - HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ Windows
  - These DLLs handle *winlogon.exe* events
  - Malware tied to an event like logon, startup, lock screen, etc.
  - It can even launch in Safe Mode

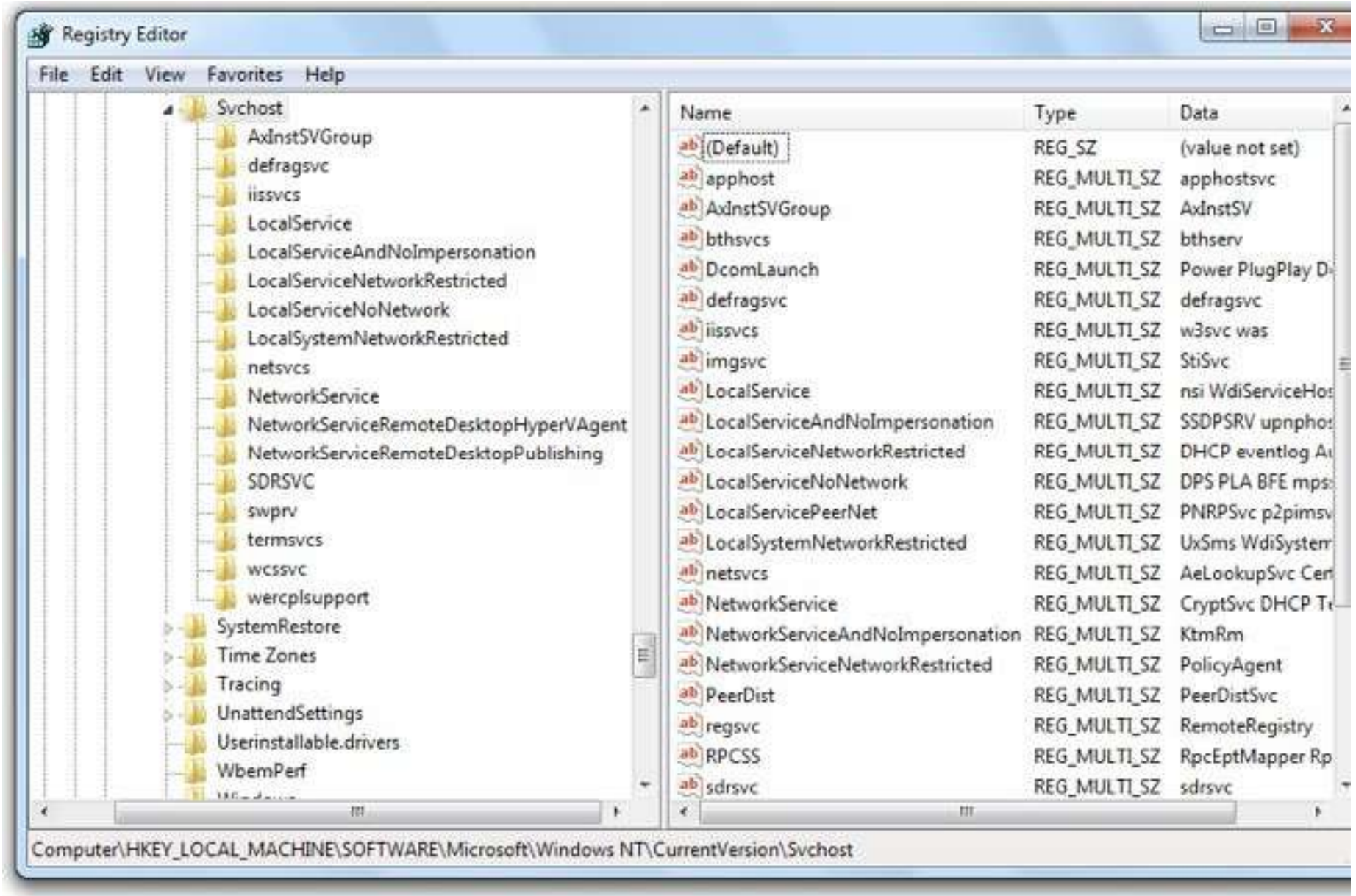
SvcHost DLLs

- Svchost is a generic host process for services that run as DLLs
- Many instances of Svchost are running at once
- Groups defined at
  - HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\ Windows NT\ CurrentVersion\ Svchost
- Services defined at
  - HKEY_LOCAL_MACHINE\ System\ CurrentControlSet\ Services\ ServiceName

Process Explorer

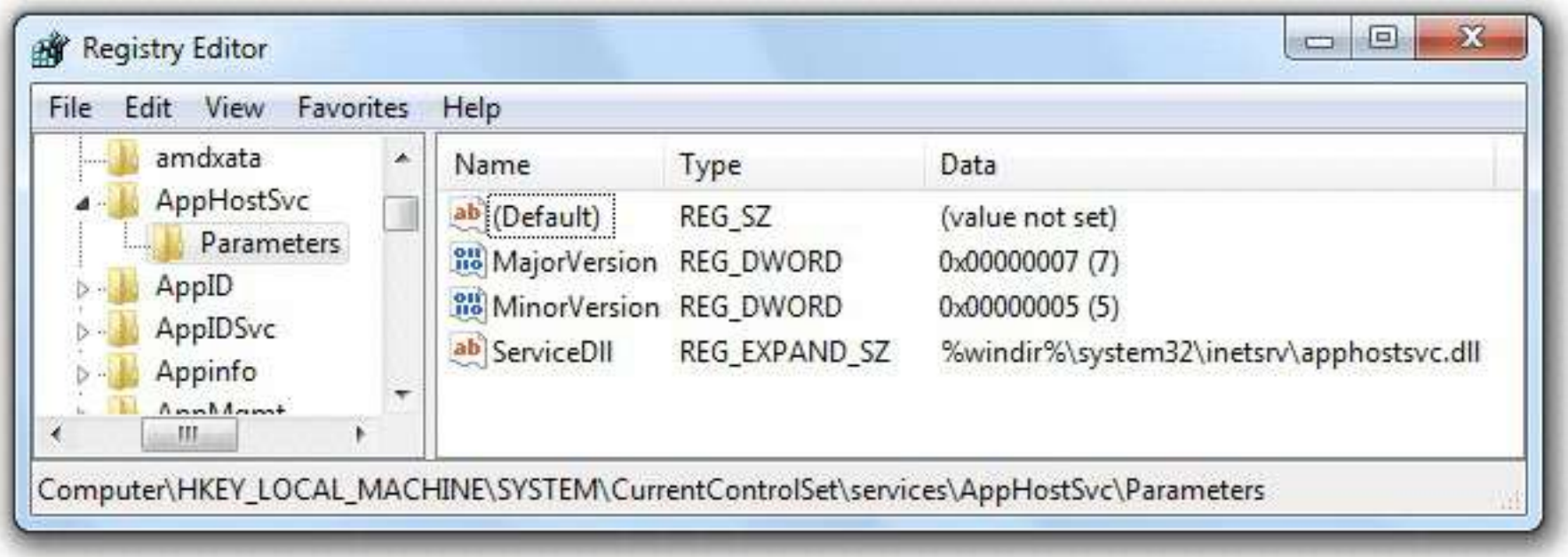
- Shows many services running in one svchost process
- This is the netsvcs group




Registry Editor
File Edit View Favorites Help
Svchost
AxInstSVGroup
defragsvc
iissvcs
LocalService
LocalServiceAndNoImpersonation
LocalServiceNetworkRestricted
LocalServiceNoNetwork
LocalSystemNetworkRestricted
netsvcs
NetworkService
NetworkServiceRemoteDesktopHyperVAgent
NetworkServiceRemoteDesktopPublishing
SDRSVC
swprv
termsvcs
wcssvc
wercplsupport
SystemRestore
Time Zones
Tracing
UnattendSettings
Userinstallable.drivers
WbemPerf
Name
Type
Data
(Default)
REG_SZ
(value not set)
apphost
REG_MULTI_SZ
apphostsvc
AxInstSVGroup
REG_MULTI_SZ
AxInstSV
bthsvcs
REG_MULTI_SZ
bthserv
DcomLaunch
REG_MULTI_SZ
Power PlugPlay D
defragsvc
REG_MULTI_SZ
defragsvc
iissvcs
REG_MULTI_SZ
w3svc was
imgsvc
REG_MULTI_SZ
StiSvc
LocalService
REG_MULTI_SZ
nsi WdiServiceHos
LocalServiceAndNoImpersonation
REG_MULTI_SZ
SSDPSRV upnphos
LocalServiceNetworkRestricted
REG_MULTI_SZ
DHCP eventlog Au
LocalServiceNoNetwork
REG_MULTI_SZ
DPS PLA BFE mps
LocalServicePeerNet
REG_MULTI_SZ
PNRPSvc p2pimsv
LocalSystemNetworkRestricted
REG_MULTI_SZ
UxSms WdiSystem
netsvcs
REG_MULTI_SZ
AeLookupSvc Cer
NetworkService
REG_MULTI_SZ
CryptSvc DHCP Te
NetworkServiceAndNoImpersonation
REG_MULTI_SZ
KtmRm
NetworkServiceNetworkRestricted
REG_MULTI_SZ
PolicyAgent
PeerDist
REG_MULTI_SZ
PeerDistSvc
regsvc
REG_MULTI_SZ
RemoteRegistry
RPCSS
REG_MULTI_SZ
RpcEptMapper Rp
sdrsvc
REG_MULTI_SZ
sdrsvc
Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost

ServiceDLL

- All *svchost.exe* DLL contain a Parameters key with a ServiceDLL value
  - Malware sets ServiceDLL to location of malicious DLL



Groups

- Malware usually adds itself to an existing group
  - Or overwrites a non-vital service
  - Often a rarely used service from the netsvcs group
- Detect this with dynamic analysis monitoring the registry
  - Or look for service functions like **`CreateServiceA`** in disassembly

Trojanized System Binaries

- Malware patches bytes of a system binary
  - To force the system to execute the malware the next time the infected binary is loaded
- DLLs are popular targets
- Typically the entry function is modified
- Jumps to code inserted in an empty portion of the binary
- Then executes DLL normally

Table 12-1. rtutils.dll's DLL Entry Point Before and After Trojanization

Original code	Trojanized code

`DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)`
 	
`mov edi, edi`
`push ebp`
`mov ebp, esp`
`push ebx`
`mov ebx, [ebp+8]`
`push esi`
`mov esi, [ebp+0Ch]`
	`DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)`
	
	`jmp DllEntryPoint_0`

The default search order for loading DLLs on Windows XP is as follows:

1. The directory from which the application loaded
2. The current directory
3. The system directory (the `GetSystemDirectory` function is used to get the path, such as *.../Windows/System32/*)
4. The 16-bit system directory (such as *.../Windows/System/*)
5. The Windows directory (the `GetWindowsDirectory` function is used to get the path, such as *.../Windows/*)
6. The directories listed in the `PATH` environment variable

KnownDLLs Registry Key

- Contains list of specific DLL locations
- Overrides the search order for listed DLLs
- Makes them load faster, and prevents load-order hijacking
- DLL load-order hijacking can only be used
  - On binaries in directories other than System32
  - That load DLLs in System32
  - That are not protected by KnownDLLs

Example: *explorer.exe*

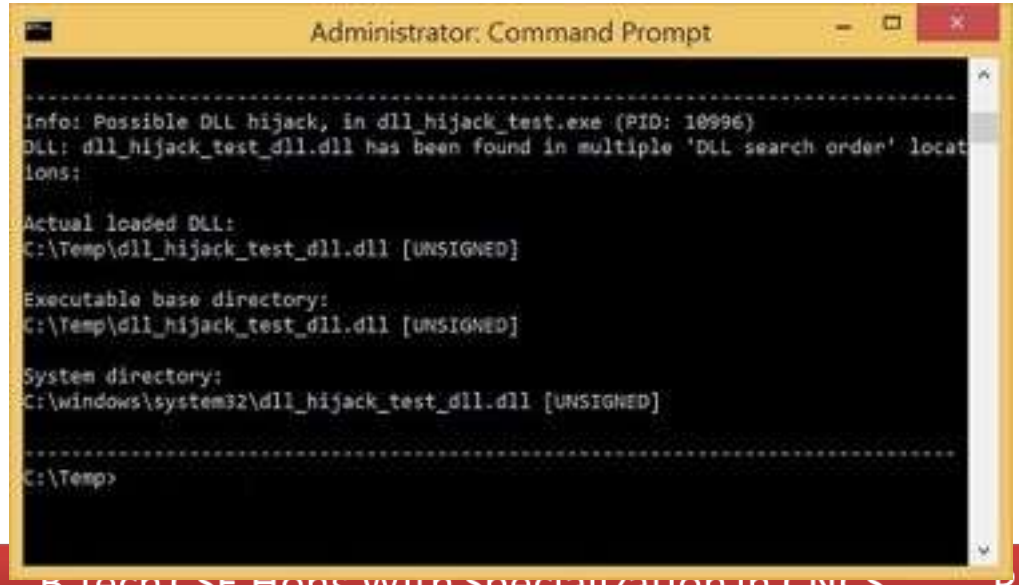
- Lives in /Windows
- Loads *ntshrui.dll* from System32
- *ntshrui.dll* is not a known DLL
- Default search is performed
- A malicious *ntshrui.dll* in /Windows will be loaded instead

Many Vulnerable DLLs

- Any startup binary not found in /System32 is vulnerable
- *explorer.exe* has about 50 vulnerable DLLs
- Known DLLs are not fully protected, because
  - Many DLLs load other DLLs
  - Recursive imports follow the default search order

DLL Load-Order Hijacking Detector

- Searches for DLLs that appear multiple times in the file system, in suspicious folders, and are unsigned
- From SANS (2015) (link Ch 11d)



No User Account Control

- Most users run Windows XP as Administrator all the time, so no privilege escalation is needed to become Administrator
- Metasploit has many privilege escalation exploits
- DLL load-order hijacking can be used to escalate privileges

Using SeDebugPrivilege

- Processes run by the user can't do everything
- Functions like **`TerminateProcess`** or **`CreateRemoteThread`** require System privileges (above Administrator)
- The **`SeDebugPrivilege`** privilege was intended for debugging
- Allows local Administrator accounts to escalate to System privileges

Example 12-6 shows how malware enables its SeDebugPrivilege.

Example 12-6. Setting the access token to SeDebugPrivilege

```
00401003  lea     eax, [esp+1Ch+TokenHandle]
00401006  push    eax                     ; TokenHandle
00401007  push    (TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY)
; DesiredAccess
00401009  call    ds:GetCurrentProcess
0040100F  push    eax                     ; ProcessHandle
00401010  call    ds:OpenProcessToken 1
00401016  test    eax, eax
00401018  jz      short loc_401080
0040101A  lea     ecx, [esp+1Ch+Luid]
0040101E  push    ecx                     ; lpLuid
0040101F  push    offset Name             ; "SeDebugPrivilege"
00401024  push    0                       ; lpSystemName
00401026  call    ds:LookupPrivilegeValueA
0040102C  test    eax, eax
0040102E  jnz     short loc_40103E
```

- 1 obtains an access token

```
...
0040103E  mov     eax, [esp+1Ch+Luid.LowPart]
00401042  mov     ecx, [esp+1Ch+Luid.HighPart]
00401046  push    0                       ; ReturnLength
00401048  push    0                       ; PreviousState
0040104A  push    10h                     ; BufferLength
0040104C  lea     edx, [esp+28h+NewState]
00401050  push    edx                     ; NewState
00401051  mov     [esp+2Ch+NewState.Privileges.Luid.LowPt], eax 3
00401055  mov     eax, [esp+2Ch+TokenHandle]
00401059  push    0                    ; DisableAllPrivileges
0040105B  push    eax                  ; TokenHandle
0040105C  mov    [esp+34h+NewState.PrivilegeCount], 1
00401064  mov    [esp+34h+NewState.Privileges.Luid.HighPt], ecx 4
00401068  mov    [esp+34h+NewState.Privileges.Attributes],
SE_PRIVILEGE_ENABLED 5
00401070  call   ds:AdjustTokenPrivileges 2
```

- 2 AdjustTokenPrivileges raises privileges to System

Covering Its Tracks—
User-Mode Rootkits

Launchers

Purpose of a Launcher

- Sets itself or another piece of malware
  - For immediate or future covert execution
- Conceals malicious behavior from the user
- Usually contain the malware they're loading
  - An executable or DLL in its own resource section
- Normal items in the resource section
  - Icons, images, menus, strings

Encryption or Compression

- The resource section may be encrypted or compressed
- Resource extraction will use APIs like
  - **`FindResource`**
  - **`LoadResource`**
  - **`SizeofResource`**
- Often contains privilege escalation code

Process Injection

- The most popular covert launching process
- Injects code into a running process
- Conceals malicious behavior
- May bypass firewalls and other process-specific security mechanisms
- Common API calls:
  - **`VirtualAllocEx`** to allocate space
  - **`WriteProcessMemory`** to write to it

DLL Injection

- The most commonly used covert launching technique
- Inject code into a remote process that calls **`LoadLibrary`**
- Forces the DLL to load in the context of that process
- On load, the OS automatically calls **`DLLMain`** which contains the malicious code

Gaining

- Malware code has the same privileges as the code it is injected into

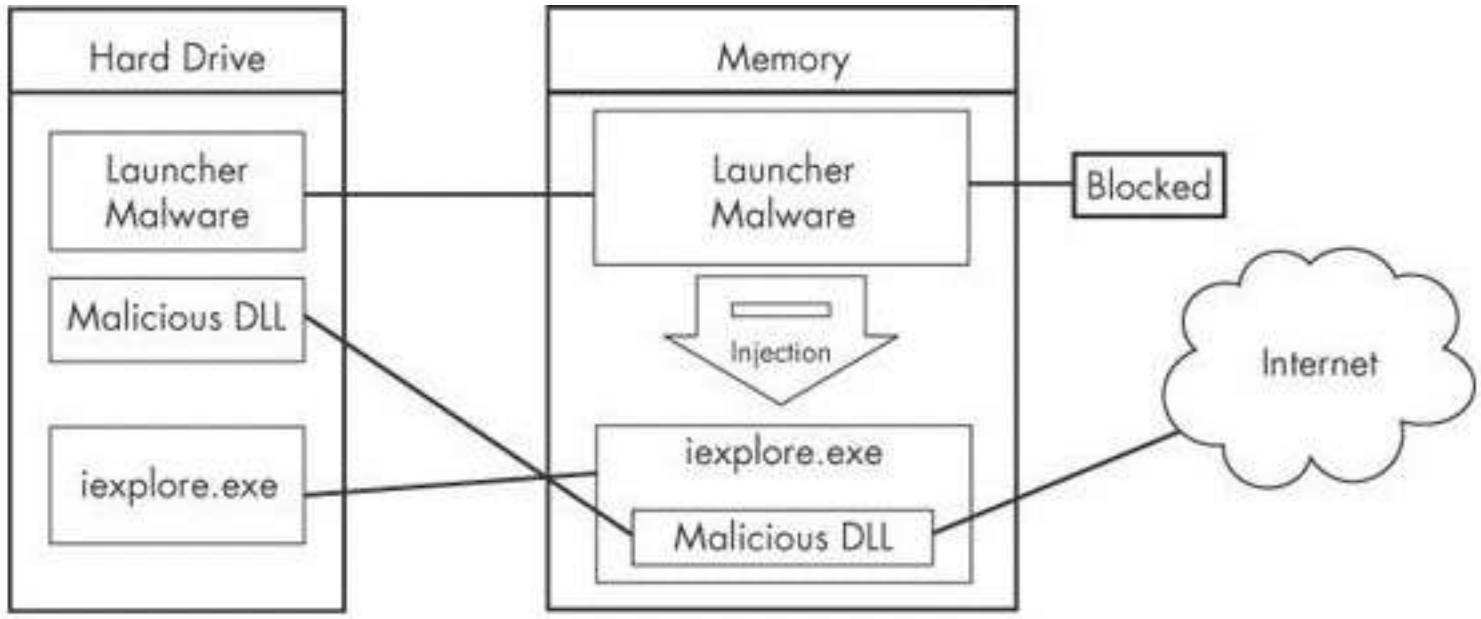


Figure 13-1. DLL injection—the launcher malware cannot access the Internet until it injects into iexplore.exe.

Example 13-1. C Pseudocode for DLL injection

```
hVictimProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, victimProcessID 1);

pNameInVictimProcess = VirtualAllocEx(hVictimProcess,...,sizeof(maliciousLibraryName),...,...);
WriteProcessMemory(hVictimProcess,...,maliciousLibraryName, sizeof(maliciousLibraryName),...);
GetModuleHandle("Kernel32.dll");
GetProcAddress(...,"LoadLibraryA");
2 CreateRemoteThread(hVictimProcess,...,...,LoadLibraryAddress,pNameInVictimProcess,...,...);
```

- **`CreateRemoteThread`** uses 3 parameters
  - Process handle **`hProcess`**
  - Starting point **`lpStartAddress`** (LoadLibrary)
  - Argument **`lpParameter`** Malicious DLL name

Direct Injection

- Injects code directly into the remote process
- Without using a DLL
- More flexible than DLL injection
- Requires a lot of customized code
  - To run without negatively impacting the host process
- Difficult to write

Process Replacement

- Overwrites the memory space of a running object with malicious code
- Disguises malware as a legitimate process
- Avoids risk of crashing a process with process injection
- Malware gains the privileges of the process it replaces
- Commonly replaces *svchost.exe*

Suspended State

- In a *suspended state*, the process is loaded into memory but the primary thread is suspended
  - So malware can overwrite its code before it runs
- This uses the **`CREATE_SUSPENDED`** value
- in the **`dwCreationFlags`** parameter
- In a call to the **`CreateProcess`** function

Example 13-2. Assembly code showing process replacement

```
00401535        push    edi                     ; lpProcessInformation
00401536        push    ecx                     ; lpStartupInfo
00401537        push    ebx                     ; lpCurrentDirectory
00401538        push    ebx                     ; lpEnvironment
00401539        push    CREATE_SUSPENDED ; dwCreationFlags
0040153B        push    ebx                     ; bInheritHandles
0040153C        push    ebx                     ; lpThreadAttributes
0040153D        lea     edx, [esp+94h+CommandLine]
00401541        push    ebx                     ; lpProcessAttributes
00401542        push    edx                     ; lpCommandLine
00401543        push    ebx                     ; lpApplicationName
00401544        mov     [esp+0A0h+StartupInfo.dwFlags], 101h
0040154F        mov     [esp+0A0h+StartupInfo.wShowWindow], bx
00401557        call    ds:CreateProcessA
```

Example 13-3. C pseudocode for process replacement

```
CreateProcess(...,"svchost.exe",...,CREATE_SUSPEND,...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(...,ImageBase,SizeOfImage,...);
WriteProcessMemory(...,headers,...);
for (i=0; i < NumberOfSections; i++) {
  WriteProcessMemory(...,section,...);
}
SetThreadContext();
...
ResumeThread();
```

- **`ZwUnmapViewOfSection`** releases all memory pointed to by a section
- **`VirtualAllocEx`** allocates new memory
- **`WriteProcessMemory`** puts malware in it

Example 13-3. C pseudocode for process replacement

```
CreateProcess(...,"svchost.exe",...,CREATE_SUSPEND,...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(...,ImageBase,SizeOfImage,...);
WriteProcessMemory(...,headers,...);
for (i=0; i < NumberOfSections; i++) {
  WriteProcessMemory(...,section,...);
}
SetThreadContext();
...
ResumeThread();
```

- **`SetThreadContext`** restores the victim process's environment and sets the entry
- **`ResumeThread`** runs the malicious code

Hooks

- Windows hooks intercept messages destined for applications
- Malicious hooks
  - Ensure that malicious code will run whenever a particular message is intercepted
  - Ensure that a DLL will be loaded in a victim process's memory space

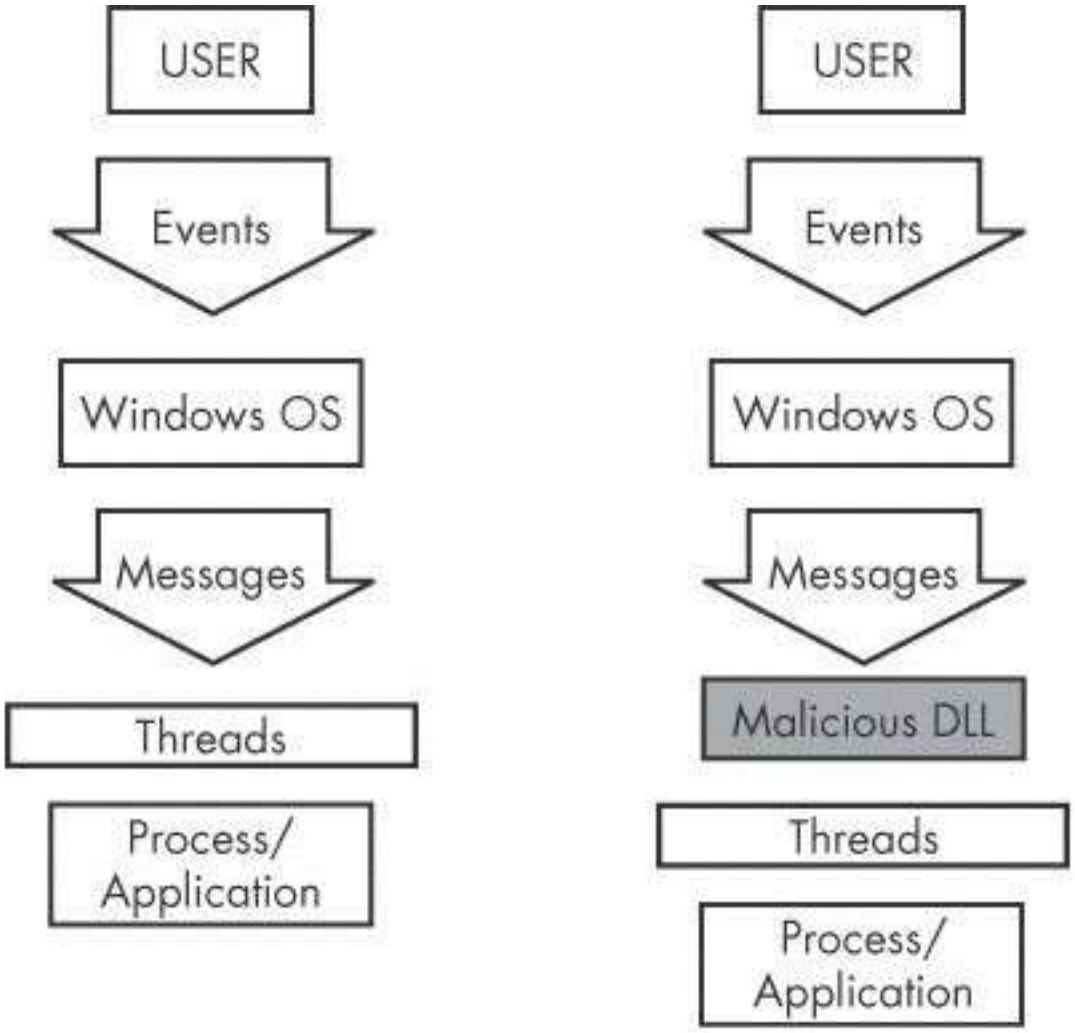


Figure 13-3. Event and message flow in Windows with and without hook injection

Local and Remote Hooks

- *Local hooks* observe or manipulate messages destined for an internal process
- *Remote hooks* observe or manipulate messages destined for a remote process (another process on the computer)

- *High-level remote hooks*
  - Require that the hook procedure is an exported function contained in a DLL
  - Mapped by the OS into the process space of a hooked thread or all threads
- *Low-level remote hooks*
  - Require that the hook procedure be contained in the process that installed the hook

Keyloggers Using Hooks

- Keystrokes can be captured by high-level or low-level hooks using these procedure types
  - **`WH_KEYBOARD`** or **`WH_KEYBOARD_LL`**

- Parameters
  - **idHook** - type of hook to install
  - **lpfn** - points to hook procedure
  - **dwThreadId** – thread to associate the hook with. Zero = all threads
  - **hMod** - handle to DLL, or local module, in which the **lpfn** procedure is defined
- The hook procedure must call **CallNextHookEx** to pass execution to the next hook procedure so the system continues to run properly

Thread Targeting

- Loading into all threads can degrade system performance
- May also trigger an IPS
- Keyloggers load into all threads, to get all the keystrokes
- Other malware targets a single thread
- Often targets a Windows message that is rarely used, such as **`WH_CBT`** (a computer-based training message)

Explanation

- Malicious DLL *hook.dll* is loaded
- Malicious hook procedure address obtained
- The hook procedure calls only **`CallNextHookEx`**
- A **`WH_CBT`** message is sent to a Notepad thread
- Forces *hook.dll* to be loaded by Notepad
- It runs in the Notepad process space

Example 13-4. Hook injection, assembly code

```
00401100        push    esi
00401101        push    edi
00401102        push    offset LibFileName ; "hook.dll"
00401107        call    LoadLibraryA
0040110D        mov     esi, eax
0040110F        push    offset ProcName ; "MalwareProc"
00401114        push    esi             ; hModule
00401115        call    GetProcAddress
0040111B        mov     edi, eax
0040111D        call    GetNotepadThreadId
00401122        push    eax             ; dwThreadId
00401123        push    esi             ; hmod
00401124        push    edi             ; lpfn
00401125        push    WH_CBT   ; idHook
00401127        call    SetWindowsHookExA
```

A Microsoft Product

- Detours makes it easy for application developers to modify applications and the OS
- Used in malware to add new DLLs to existing binaries on disk
- Modifies the PE structure to create a `.detour` section
- Containing original PE header with a new import address table

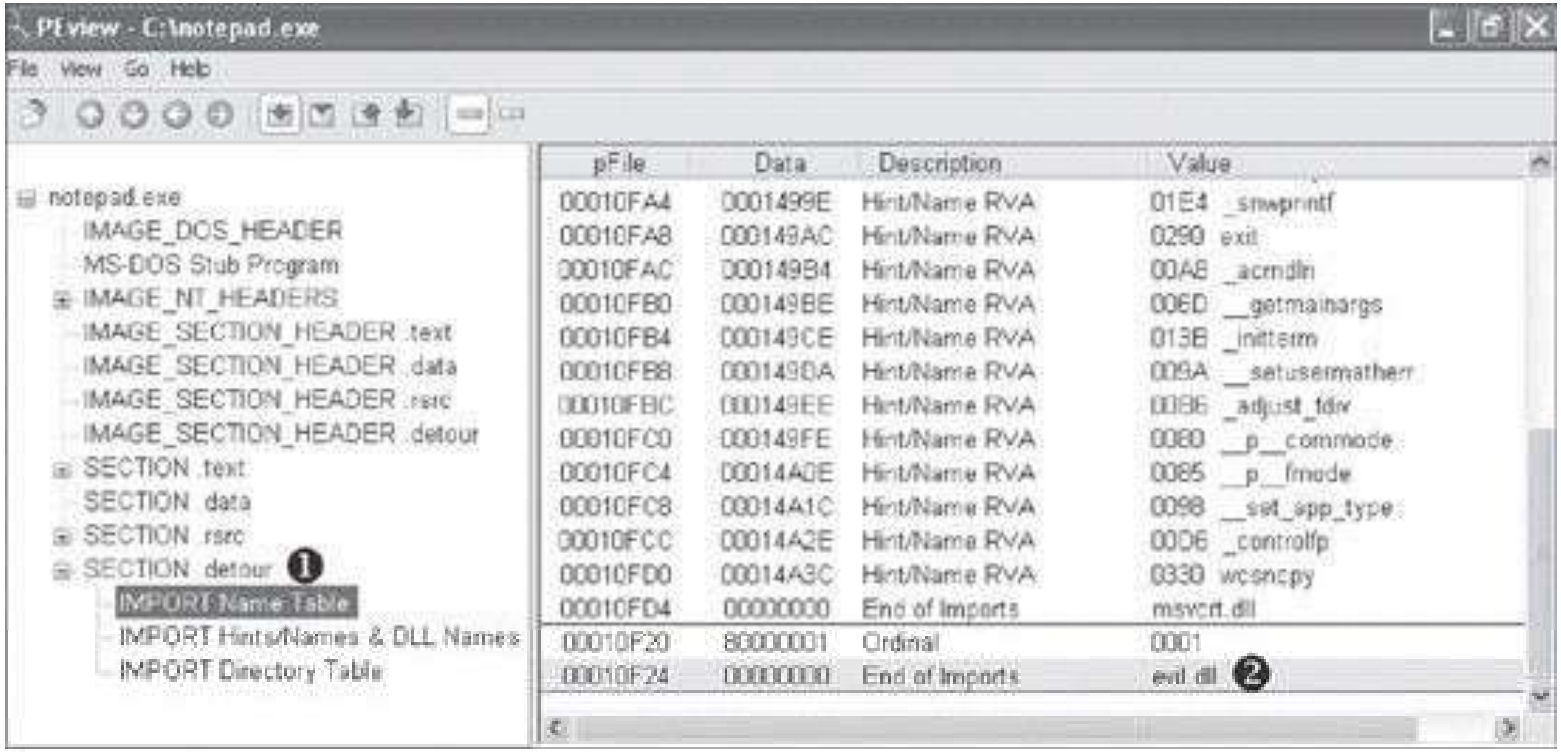


Figure 13-4. A PEview of Detours and the evil.dll

- **`setdll`** is the Microsoft tool used to point the PE to the new import table
- There are other ways to add a **`.detour`** section

- Directs a thread to execute other code prior to executing its regular path
- Every thread has a queue of APCs attached to it
- These are processed when the thread is in an alterable state, such as when these functions are called
  - **`WaitForSingleObjectEx`**
  - **`WaitForMultipleObjectsEx`**
  - **`Sleep`**

Two Forms of APCs

- Kernel-Mode APC
  - Generated for the system or a driver
- User-Mode APC
  - Generated for an application
- APC Injection is used in both cases

APC Injection from User Space

- Uses API function **`QueueUserAPC`**
- Thread must be in an alterable state
- **`WaitForSingleObjectEx`** is the most common call in the Windows API
- Many threads are usually in the alterable state

QueueUserAPC Parameters

- **hThread** handle to
- **pfnAPC** defines the function to run
- **dwData** parameter for function

Example 13-5. APC injection from a user-mode application

```
00401DA9        push    [esp+4+dwThreadId]      ; dwThreadId
00401DAD        push    0                       ; bInheritHandle
00401DAF        push    10h                     ; dwDesiredAccess
00401DB1        call    ds:OpenThread 1
00401DB7        mov     esi, eax
00401DB9        test    esi, esi
00401DBB        jz      short loc_401DCE
00401DBD        push    [esp+4+dwData]          ; dwData = dbnet.dll
00401DC1        push    esi                     ; hThread
00401DC2        push    ds:LoadLibraryA 2       ; pfnAPC
00401DC8        call    ds:QueueUserAPC
```

- 1: Opens a handle to the thread
- 2: **`QueueUserAPC`** is called with **`pfnAPC`** set to **`LoadLibraryA`** (loads a DLL)
- **`dwData`** contains the DLL name (*dbnet.dll*)
- *Svchost.exe* is often targeted for APC injection

APC Injection from Kernel Space

- Malware drivers and rootkits often want to execute code in user space
- This is difficult to do
- One method is APC injection to get to user space
- Most often to *svchost.exe*
- Functions used:
  - **`KeInitializeApc`**
  - **`KeInsertQueueApc`**

Example 13-6. User-mode APC injection from kernel space

```
000119BD        push    ebx
000119BE        push    1 1
000119C0        push    [ebp+arg_4] 2
000119C3        push    ebx
000119C4        push    offset sub_11964
000119C9        push    2
000119CB        push    [ebp+arg_0] 3
000119CE        push    esi
000119CF        call    ds:KeInitializeApc
000119D5        cmp     edi, ebx
000119D7        jz      short loc_119EA
000119D9        push    ebx
000119DA        push    [ebp+arg_C]
000119DD        push    [ebp+arg_8]
000119E0        push    esi
000119E1        call    edi         ;KeInsertQueueApc
```

User-Mode Rootkits

- Modify internal functionality of the OS
- Hide files, network connections, processes, etc.
- Kernel-mode rootkits are more powerful
- This section is about User-mode rootkits

IAT (Import Address Table) Hooking

- May modify
  - IAT (Import Address Table) or
  - EAT (Export Address Table)
- Parts of a PE file
- Filled in by the loader
  - Link Ch 11a

– This technique is old and easily detected

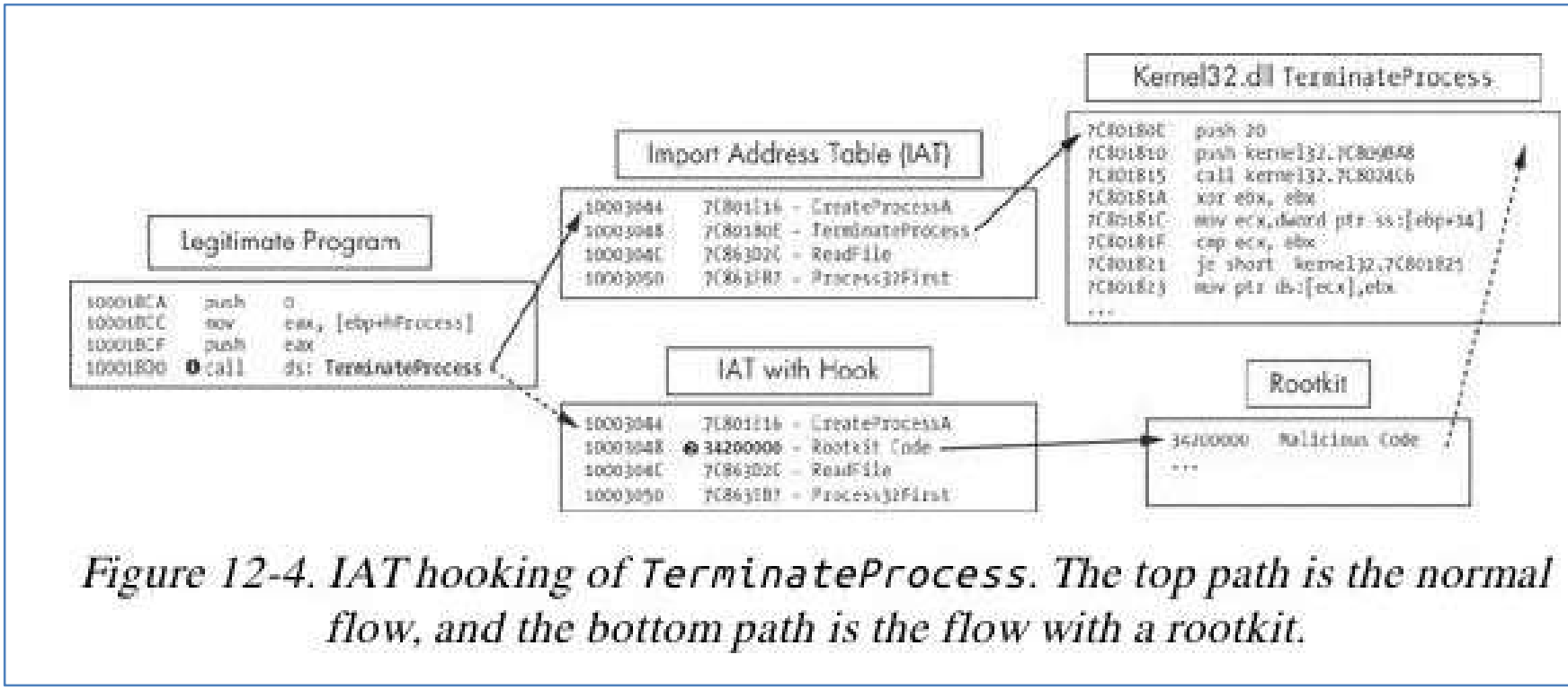


Figure 12-4. IAT hooking of `TerminateProcess`. The top path is the normal flow, and the bottom path is the flow with a rootkit.

Inline Hooking

- Overwrites the API function code
- Contained in the imported DLLs
- Changes actual function code, not pointers
- A more advanced technique than IAT hooking

Purpose of a Launcher

- Sets itself or another piece of malware
  - For immediate or future covert execution
- Conceals malicious behavior from the user
- Usually contain the malware they're loading
  - An executable or DLL in its own resource section
- Normal items in the resource section
  - Icons, images, menus, strings

Encryption or Compression

- The resource section may be encrypted or compressed
- Resource extraction will use APIs like
  - **`FindResource`**
  - **`LoadResource`**
  - **`SizeofResource`**
- Often contains privilege escalation code

Process Injection

- The most popular covert launching process
- Injects code into a running process
- Conceals malicious behavior
- May bypass firewalls and other process-specific security mechanisms
- Common API calls:
  - **`VirtualAllocEx`** to allocate space
  - **`WriteProcessMemory`** to write to it

DLL Injection

- The most commonly used covert launching technique
- Inject code into a remote process that calls **`LoadLibrary`**
- Forces the DLL to load in the context of that process
- On load, the OS automatically calls **`DLLMain`** which contains the malicious code

Gaining Privileges

- Malware code has the same privileges as the code it is injected into

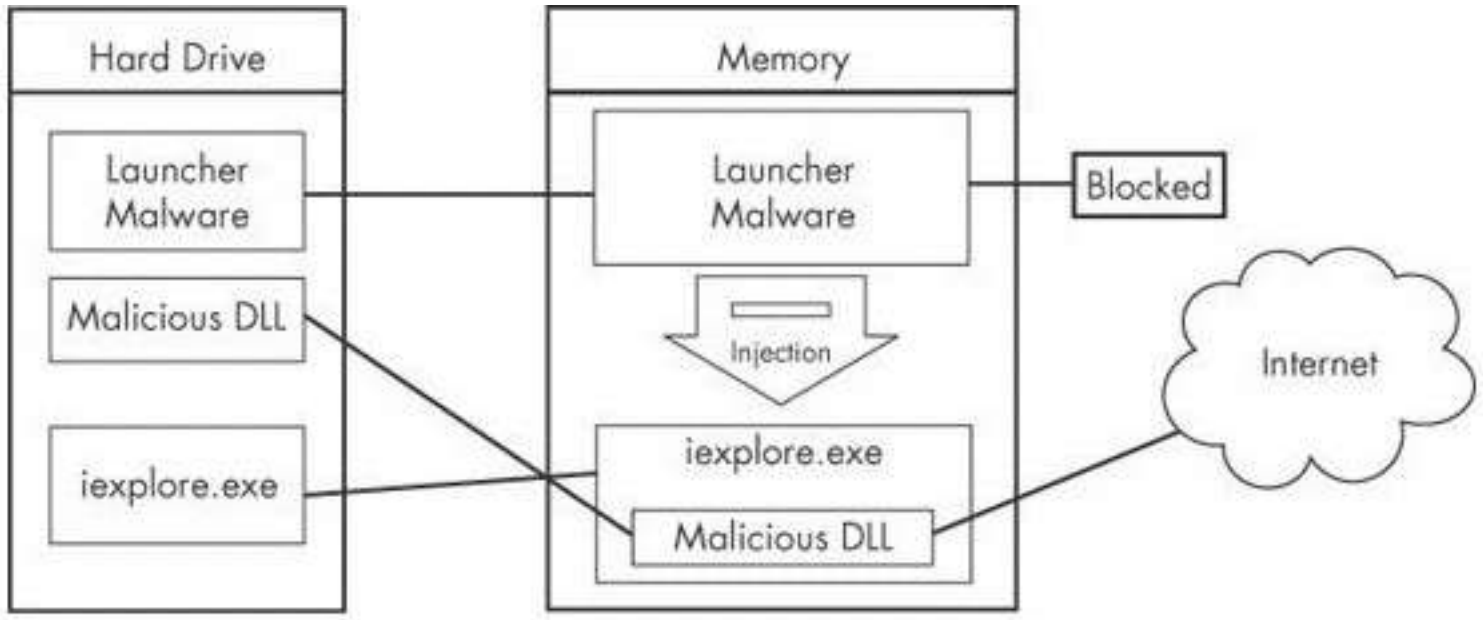


Figure 13-1. DLL injection—the launcher malware cannot access the Internet until it injects into iexplore.exe.

Example 13-1. C Pseudocode for DLL injection

```
hVictimProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, victimProcessID 1);

pNameInVictimProcess = VirtualAllocEx(hVictimProcess,...,sizeof(maliciousLibraryName),...,...);
WriteProcessMemory(hVictimProcess,...,maliciousLibraryName, sizeof(maliciousLibraryName),...);
GetModuleHandle("Kernel32.dll");
GetProcAddress(...,"LoadLibraryA");
2 CreateRemoteThread(hVictimProcess,...,...,LoadLibraryAddress,pNameInVictimProcess,...,...);
```

- **`CreateRemoteThread`** uses 3 parameters
  - Process handle **`hProcess`**
  - Starting point **`lpStartAddress`** (LoadLibrary)
  - Argument **`lpParameter`** Malicious DLL name

Direct Injection

- Injects code directly into the remote process
- Without using a DLL
- More flexible than DLL injection
- Requires a lot of customized code
  - To run without negatively impacting the host process
- Difficult to write

Process Replacement

- Overwrites the memory space of a running object with malicious code
- Disguises malware as a legitimate process
- Avoids risk of crashing a process with process injection
- Malware gains the privileges of the process it replaces
- Commonly replaces *svchost.exe*

Suspended State

- In a *suspended state*, the process is loaded into memory but the primary thread is suspended
  - So malware can overwrite its code before it runs
- This uses the **`CREATE_SUSPENDED`** value
- in the **`dwCreationFlags`** parameter
- In a call to the **`CreateProcess`** function

Example 13-2. Assembly code showing process replacement

```
00401535        push    edi                     ; lpProcessInformation
00401536        push    ecx                     ; lpStartupInfo
00401537        push    ebx                     ; lpCurrentDirectory
00401538        push    ebx                     ; lpEnvironment
00401539        push    CREATE_SUSPENDED ; dwCreationFlags
0040153B        push    ebx                     ; bInheritHandles
0040153C        push    ebx                     ; lpThreadAttributes
0040153D        lea     edx, [esp+94h+CommandLine]
00401541        push    ebx                     ; lpProcessAttributes
00401542        push    edx                     ; lpCommandLine
00401543        push    ebx                     ; lpApplicationName
00401544        mov     [esp+0A0h+StartupInfo.dwFlags], 101h
0040154F        mov     [esp+0A0h+StartupInfo.wShowWindow], bx
00401557        call    ds:CreateProcessA
```

Example 13-3. C pseudocode for process replacement

```
CreateProcess(...,"svchost.exe",...,CREATE_SUSPEND,...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(...,ImageBase,SizeOfImage,...);
WriteProcessMemory(...,headers,...);
for (i=0; i < NumberOfSections; i++) {
  WriteProcessMemory(...,section,...);
}
SetThreadContext();
...
ResumeThread();
```

- **`ZwUnmapViewOfSection`** releases all memory pointed to by a section
- **`VirtualAllocEx`** allocates new memory
- **`WriteProcessMemory`** puts malware in it

Example 13-3. C pseudocode for process replacement

```
CreateProcess(...,"svchost.exe",...,CREATE_SUSPEND,...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(...,ImageBase,SizeOfImage,...);
WriteProcessMemory(...,headers,...);
for (i=0; i < NumberOfSections; i++) {
  WriteProcessMemory(...,section,...);
}
SetThreadContext();
...
ResumeThread();
```

- **`SetThreadContext`** restores the victim process's environment and sets the entry
- **`ResumeThread`** runs the malicious code

Hooks

- Windows hooks intercept messages destined for applications
- Malicious hooks
  - Ensure that malicious code will run whenever a particular message is intercepted
  - Ensure that a DLL will be loaded in a victim process's memory space

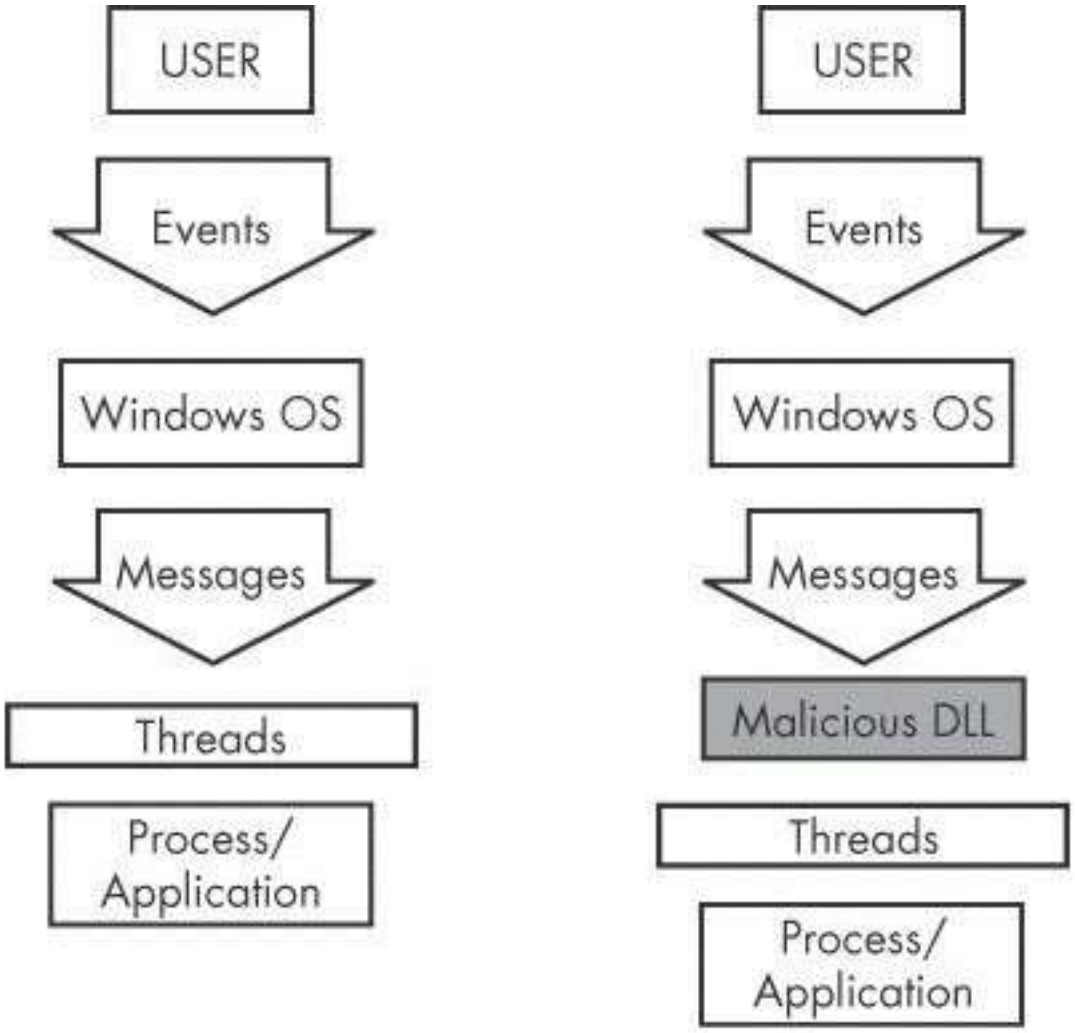


Figure 13-3. Event and message flow in Windows with and without hook injection

Local and Remote

- *Local hooks* observe or manipulate messages destined for an internal process
- *Remote hooks* observe or manipulate messages destined for a remote process (another process on the computer)

- *High-level remote hooks*
  - Require that the hook procedure is an exported function contained in a DLL
  - Mapped by the OS into the process space of a hooked thread or all threads
- *Low-level remote hooks*
  - Require that the hook procedure be contained in the process that installed the hook

Keyloggers Using Hooks

- Keystrokes can be captured by high-level or low-level hooks using these procedure types
  - **`WH_KEYBOARD`** or **`WH_KEYBOARD_LL`**

- Parameters
  - **idHook** - type of hook to install
  - **lpfn** - points to hook procedure
  - **dwThreadId** – thread to associate the hook with. Zero = all threads
  - **hMod** - handle to DLL, or local module, in which the **lpfn** procedure is defined
- The hook procedure must call **CallNextHookEx** to pass execution to the next hook procedure so the system continues to run properly

Thread Targeting

- Loading into all threads can degrade system performance
- May also trigger an IPS
- Keyloggers load into all threads, to get all the keystrokes
- Other malware targets a single thread
- Often targets a Windows message that is rarely used, such as **`WH_CBT`** (a computer-based training message)

Explanation

- Malicious DLL *hook.dll* is loaded
- Malicious hook procedure address obtained
- The hook procedure calls only **`CallNextHookEx`**
- A **`WH_CBT`** message is sent to a Notepad thread
- Forces *hook.dll* to be loaded by Notepad
- It runs in the Notepad process space

Example 13-4. Hook injection, assembly code

```
00401100        push    esi
00401101        push    edi
00401102        push    offset LibFileName ; "hook.dll"
00401107        call    LoadLibraryA
0040110D        mov     esi, eax
0040110F        push    offset ProcName ; "MalwareProc"
00401114        push    esi             ; hModule
00401115        call    GetProcAddress
0040111B        mov     edi, eax
0040111D        call    GetNotepadThreadId
00401122        push    eax             ; dwThreadId
00401123        push    esi             ; hmod
00401124        push    edi             ; lpfn
00401125        push    WH_CBT   ; idHook
00401127        call    SetWindowsHookExA
```

A Microsoft Product

- Detours makes it easy for application developers to modify applications and the OS
- Used in malware to add new DLLs to existing binaries on disk
- Modifies the PE structure to create a `.detour` section
- Containing original PE header with a new import address table

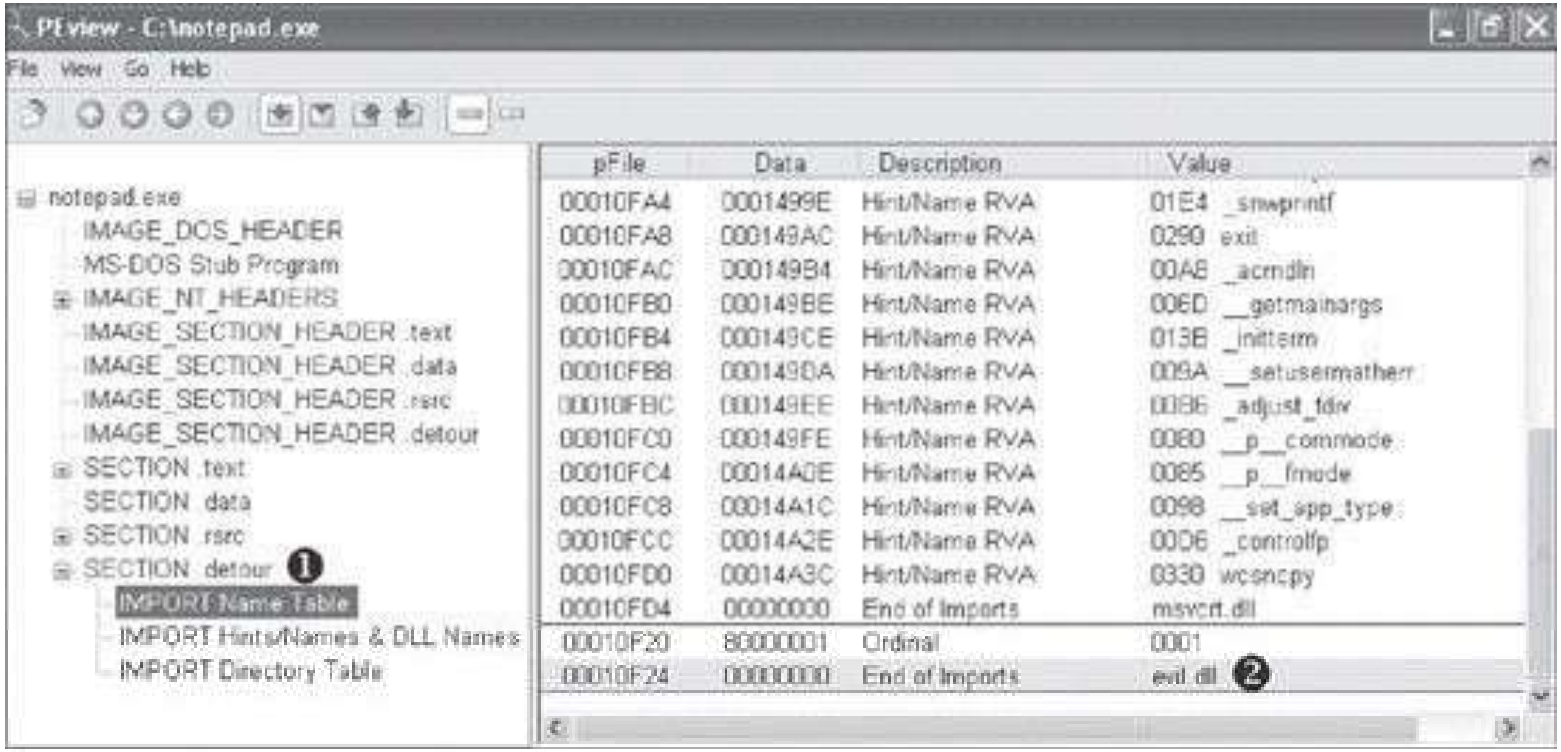


Figure 13-4. A PEview of Detours and the evil.dll

- **`setdll`** is the Microsoft tool used to point the PE to the new import table
- There are other ways to add a **`.detour`** section

- Directs a thread to execute other code prior to executing its regular path
- Every thread has a queue of APCs attached to it
- These are processed when the thread is in an alterable state, such as when these functions are called
  - **`WaitForSingleObjectEx`**
  - **`WaitForMultipleObjectsEx`**
  - **`Sleep`**

Two Forms of APCs

- Kernel-Mode APC
  - Generated for the system or a driver
- User-Mode APC
  - Generated for an application
- APC Injection is used in both cases

APC Injection from User Space

- Uses API function **`QueueUserAPC`**
- Thread must be in an alterable state
- **`WaitForSingleObjectEx`** is the most common call in the Windows API
- Many threads are usually in the alterable state

QueueUserAPC Parameters

- **hThread** handle to
- **pfnAPC** defines the function to run
- **dwData** parameter for function

Example 13-5. APC injection from a user-mode application

```
00401DA9        push    [esp+4+dwThreadId]      ; dwThreadId
00401DAD        push    0                       ; bInheritHandle
00401DAF        push    10h                     ; dwDesiredAccess
00401DB1        call    ds:OpenThread 1
00401DB7        mov     esi, eax
00401DB9        test    esi, esi
00401DBB        jz      short loc_401DCE
00401DBD        push    [esp+4+dwData]          ; dwData = dbnet.dll
00401DC1        push    esi                     ; hThread
00401DC2        push    ds:LoadLibraryA 2       ; pfnAPC
00401DC8        call    ds:QueueUserAPC
```

- 1: Opens a handle to the thread
- 2: **`QueueUserAPC`** is called with **`pfnAPC`** set to **`LoadLibraryA`** (loads a DLL)
- **`dwData`** contains the DLL name (*dbnet.dll*)
- *Svchost.exe* is often targeted for APC injection

APC Injection from Kernel Space

- Malware drivers and rootkits often want to execute code in user space
- This is difficult to do
- One method is APC injection to get to user space
- Most often to *svchost.exe*
- Functions used:
  - **KeInitializeApc**
  - **KeInsertQueueApc**

Example 13-6. User-mode APC injection from kernel space

```
000119BD        push    ebx
000119BE        push    1 1
000119C0        push    [ebp+arg_4] 2
000119C3        push    ebx
000119C4        push    offset sub_11964
000119C9        push    2
000119CB        push    [ebp+arg_0] 3
000119CE        push    esi
000119CF        call    ds:KeInitializeApc
000119D5        cmp     edi, ebx
000119D7        jz      short loc_119EA
000119D9        push    ebx
000119DA        push    [ebp+arg_C]
000119DD        push    [ebp+arg_8]
000119E0        push    esi
000119E1        call    edi         ;KeInsertQueueApc
```