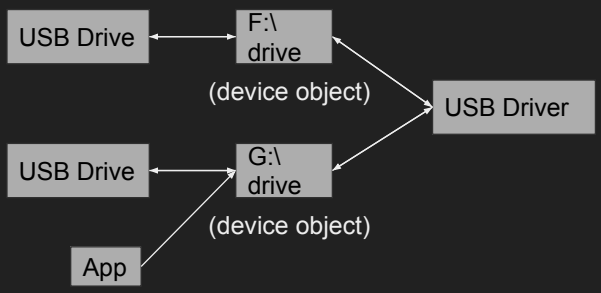
Practical Malware Analysis

Lecture 10 | WinDbg

Drivers and Kernel Code

- Drivers allow 3rd-party devs to run code in the kernel
  - Resident in memory
  - Device <-> device object <-> driver
    - Driver creates/destroys devices accessible from user space



In Windows, device drivers allow third party developers to run code in the kernel. Device drivers may be difficult to analyze, as they are loaded into and live in memory. Applications do not interact directly with drivers, but instead must interface with a device object that sends requests to devices (hardware or virtual - created/destroyed by the driver) which are accessible from user space. For example, when a USB drive is plugged into a Windows computer, a logical drive letter is assigned to that device, with the drive representing the device object which accepts requests from applications and sends requests to the USB driver. One driver may handle requests from multiple device objects, in this case from another USB device for example.

Loading Drivers

Loading into kernel

- Create driver object structure
  - Call *DriverEntry* procedure (like *DllMain*)
    - Driver registers address of callback functions (adds to struct)
      - Called by software in user space
    - Creates device accessible from user space
- *DeviceIoControl*
  - Common call for malicious kernel component
  - Arbitrary-length buffer of data in/out

Loading into kernel

Create driver object structure
Call *DriverEntry* procedure (like *DllMain*)
Driver registers address of callback functions (adds to struct)
Called by software in user space
Creates device accessible from user space
DeviceIoControl
Common call for malicious kernel component
Arbitrary-length buffer of data in/out

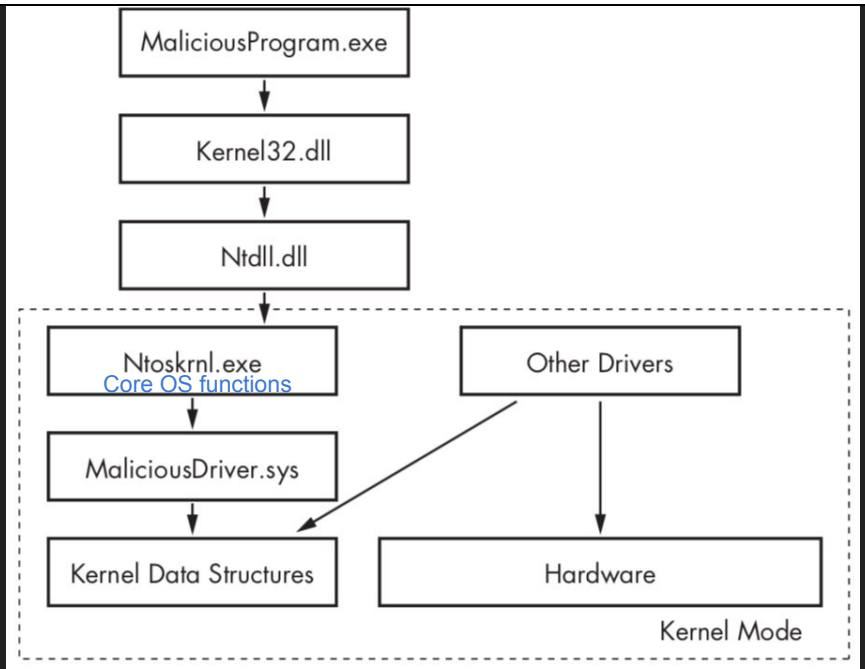


Figure 10-1: How user-mode calls are handled by the kernel

The above illustrates the layers that a user mode request may traverse to reach the kernel. Some requests are sent to drivers that control hardware while others only modify the kernel's state. Some malicious code may never interact with user space, and only executes within the kernel. Malicious drivers generally will not control hardware, but instead interact with Windows kernel components *ntoskrnl.exe* (core OS functions) and *hal.dll* (code for interfacing with main hardware components)

Setting up Kernel Debugging

- Can’t debug on the same machine
  - Kernel has one process context, paused for debugging
- See the book / the Internet for setup w/ WinDbg

Can’t debug on the same machine
    Kernel has one process context, paused for debugging
See the book / the Internet for setup w/ WinDbg

Using WinDbg

Reading from memory [***d****x*], write with [***e****x*]

Table 10-1: WinDbg Reading Options

Option	Description
da	Reads from memory and displays it as ASCII text
du	Reads from memory and displays it as Unicode text
dd	Reads from memory and displays it as 32-bit double words

```
ex addressToWrite dataToWrite
```

Reading from memory [**d**x], write with [**e**x]

(see help for more options)

Using WinDbg

Arithmetic on memory / registers

- + - * /
  - Useful for conditional breakpoints

Dereferencing

- [*dwo*]
  - Dereference a 32-bit pointer
  - Ex. *du dwo (esp+4)* to view value at esp+4

Arithmetic on memory / registers
+ - * /
Useful for conditional breakpoints
Dereferencing
[dwo]
Dereference a 32-bit pointer
Ex. du dwo (esp+4) to view value at esp+4

Using WinDbg

Breakpoints

- [*bp*]
  - Specify cmd to run at breakpoint
    - Can include conditionals like *.if* and *.while*
  - Execute cmd and continue execution with *go* [*g*]
    - Ex. *bp GetProcAddress "da dwo(esp+8); g"* to print function called for every *GetProcAddress* w/o halting execution

Listing Modules

- [lm] lists all user/kernel space modules w/ start and end address

Breakpoints

[bp]

Specify cmd to run at breakpoint

Can include conditionals like .if and .while

Execute cmd and continue execution with go [g]

Ex. bp GetProcAddress "da dwo(esp+8); g" to print function called for every GetProcAddress w/o halting execution

Listing Modules

[lm] lists all user/kernel space modules w/ start and end address

Microsoft Symbols

Names for functions and variables (named memory addresses)

- Search with *moduleName!symbolName*
  - *moduleName* = name of .exe, .dll, or .sys (w/o extension)
    - Except *notskrnl.exe* which is just *nt*
  - *symbolName* = name of address
    - Ex. *u nt!NtCreateProcess* (*u* = unassemble)
    - No module specified = search all modules
- [*bu*]
  - Set a deferred breakpoint (break when module is loaded)
    - Ex. *bu newModule!exportedFunction*
  - *$iment* determines entry point of module
    - Ex. *bu $iment(driverName)*

Names for functions and variables (named memory addresses)

Search with moduleName!symbolName
moduleName = name of .exe, .dll, or .sys (w/o extension)
Except notskrnl.exe which is just nt
symbolName = name of address
Ex. u nt!NtCreateProcess (u = unassemble)
No module specified = search all modules
[bu]
Set a deferred breakpoint (break when module is loaded)
Ex. bu newModule!exportedFunction
$iment determines entry point of module
Ex. bu $iment(driverName)

Microsoft Symbols

- [*x*] search for functions/symbols w/ wildcards
  - Ex. *x nt!*CreateProcess**
- [*ln*] list closest symbol to <address>
  - Ex. *ln 805717aa*

Structures

- *dt nt!_DRIVER_OBJECT <memory_address>*

[x] search for functions/symbols w/ wildcards
	Ex. x nt!*CreateProcess*
[ln] list closest symbol to <address>
	Ex. ln 805717aa
Structures
	dt nt!_DRIVER_OBJECT <memory_address>

```
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       : ❶0xf7adb66c     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
```

The above represents the WinDbg output of a query for information on the beep driver object structure. This driver allows Windows to make a beeping noise indicating a malfunction. At (1) the initialization function is called when the driver is loaded, is located at 0xf7adb66c.

An initialization function is only called when the driver is loaded, and can be an area of interested for malicious drivers, sometimes holding the entire payload.

Configuring Windows Symbols

- Symbols are specific to file versions
  - Can change with an update or hotfix
    - WinDbg can get correct symbols from Microsoft's server
      - File -> Symbol File Path
        - Online: *SRV*c:\websymbols*http://msdl.microsoft.com/download/symbols*
    - Download symbols specific to OS, service pack, architecture

Symbols are specific to file versions
Can change with an update or hotfix
WinDbg can get correct symbols from Microsoft's server
File -> Symbol File Path
Online:
SRV*c:\websymbols*http://msdl.microsoft.com/download/symbols
Download symbols specific to OS, service pack, architecture

Kernel Debugging: Writing a File (User-Space Code)

Create driver -> get handle to device object -> send data to driver

The call at the bottom of this user-space code (service creation to load a kernel driver) indicates that the driver is being created with a call to CreateService. The parameter pushed onto the stack at (1 left) indicates a ServiceType of 0x01 indicating a SERVICE_KERNEL_DRIVER type.

On the top right, a file is being created to get a handle to the device with CreateFileA at (1), and the name of the file at (2) corresponding to \\.\FileWriterDevice (not pictured), the name of the device object created by the driver for user-space applications to access.

On the bottom right, a call to DeviceIoControl is used with the device handle to send data to the driver.

Kernel Debugging: Writing a File (Kernel-Space Code)

1. Kernel modules are not loaded often (except *KMixer.sys*), WinDbg will alert you `ModLoad: f7b0d000 f7b0e780   FileWriter.sys`
2. Find driver object

```
kd> !drvobj FileWriter
Driver object (❶827e3698) 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
```

1. Kernel modules are not loaded often (except KMixer.sys), WinDbg will alert you
2. Finding the driver object is simple when the name is known, using *!drvobj <name>*
   a. You can also browse objects with *!object \Driver* to list all objects in \Driver namespace (a root namespace)
   b. This command revealed that the driver object is stored at 0x827e3698

Kernel Debugging: Writing a File (Kernel-Space Code)

3. View driver object structure

```
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
```

After locating the driver object, we can view its structure with *dt nt!_DRIVER_OBJECT 0x827e3698*

The final entry for *MajorFunction* is a pointer to the first entry of the MajorFunction table, which shows what is executed when the driver is called from user space.

Each index of the MajorFunction table contains different functions, and each index represents a different type of request (indices are found in the *wdm.h* file and start with *IRP_MJ_*

In this scenario, we want to look for the functions associated with *DeviceIoControl* which is called from user space, so we would investigate the *IRP_MJ_DEVICE_CONTROL* index, found at 0xE. The major function table is at offset 0x038 from the beginning of the driver object, so the function that will handle the *DeviceIoControl* request will be found with *dd 827e3698+0x38+e*4 L1*

0xE is multiplied by 4 bytes, as each pointer is 4 bytes long

L1 specifies only one DWORD as output

The output of this command is on the next slide

Kernel Debugging: Writing a File (Kernel-Space Code)

4. Find the function executed by the user-space call

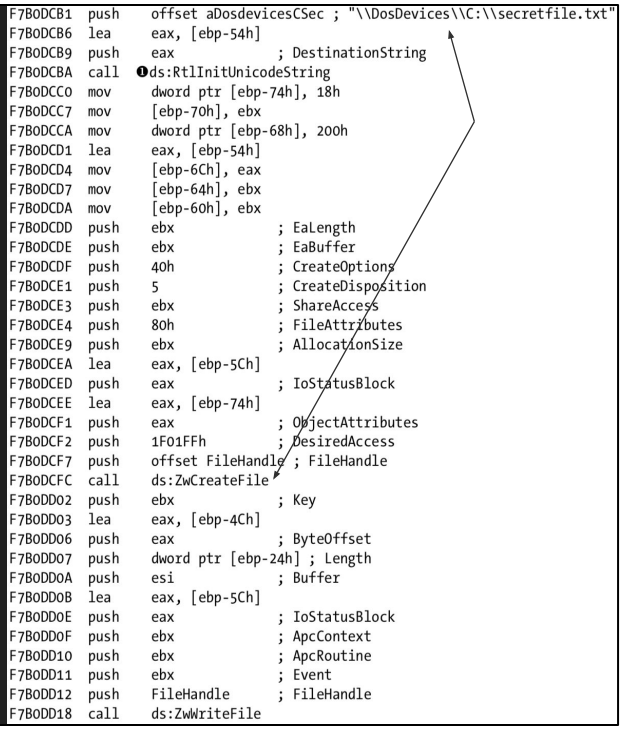
```
kd> dd 827e3698+0x38+e*4 L1
827e3708  f7b0da66
kd> u f7b0da66
FileWriter+0xa66:
f7b0da66 6a68            push    68h
f7b0da68 6838d9b0f7      push    offset FileWriter+0x938 (f7b0d938)
f7b0da6d e822faffff      call    FileWriter+0x494 (f7b0d494)
```

The function called in the kernel is located at 0xf7b0da66, disassembling this address shows the instructions appear valid (no errors in our calculation). At this point the kernel driver can be loaded into IDA or further analysed with a breakpoint in WinDbg.

Kernel Debugging: Writing a File (Kernel-Space Code)

5. Analyse the function call (IDA, probably)

- Create/Read/WriteFile all used to send data from user space to kernel drivers



The above shows IDA Pro output for the malicious driver.

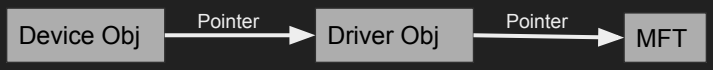
Observe that the Windows kernel uses a UNICODE_STRING structure, instead of the wide character strings in user space. The function called at (1) creates kernel strings. The function accepts a DestinationString parameter (a pointer to the UNICODE_STRING structure to be initialized), and a SourceString (a pointer to a null-terminated wide-character string).

Later, the call to ZwCreateFile uses a fully qualified object name (identifies the root device involved), in this case *\DosDevices\C:\secretfile.txt*, to create a file from within the kernel. For most devices, this name will be *\DosDevices\<object_name>*.

Create/Read/WriteFile can all be used to send data from user space to kernel drivers. Consider an application call to ReadFile with a handle to a device, which calls the IRP_MJ_READ function.

Finding Driver Objects

Application <-> device object <-> driver object

- *!devobj* to get device object info (use name of device specified by user space CreateFile call
- 

- See which application is using the driver with *!devhandles*
  - Iterates through every handle table for every process

```
kd> !devobj FileWriterDevice
Device object (826eb030) is for:
 Rootkit \Driver\FileWriter DriverObject 827e3698
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e13deedc DevExt 00000000 DevObjExt 828eb0e8
ExtensionFlags (0000000000)
Device queue is not busy.
```

```
kd>!devhandles 826eb030
...
Checking handle table for process 0x829001f0
Handle table at e1d09000 with 32 Entries in use

Checking handle table for process 0x8258d548
Handle table at e1cfa000 with 114 Entries in use

Checking handle table for process 0x82752da0
Handle table at e1045000 with 18 Entries in use
PROCESS 82752da0  SessionId: 0  Cid: 0410    Peb: 7ffd5000  ParentCid: 075c
    DirBase: 09180240  ObjectTable: e1da0180  HandleCount:  18.
    Image: FileWriterApp.exe

07b8: Object: 826eb0e8  GrantedAccess: 0012019f
```

From the user space call for CreateFile with the device name as the file being created, we can use the *!devobj <name>* command to list information of the device object being accessed by the application. The device object gives us a pointer to the drive object, and from there we can find the major function table as seen earlier. To see which applications are using the malicious driver, the *!devhandles* command can be used to iterate through the handle table for every process (slow!) and reveal which application is using the driver.

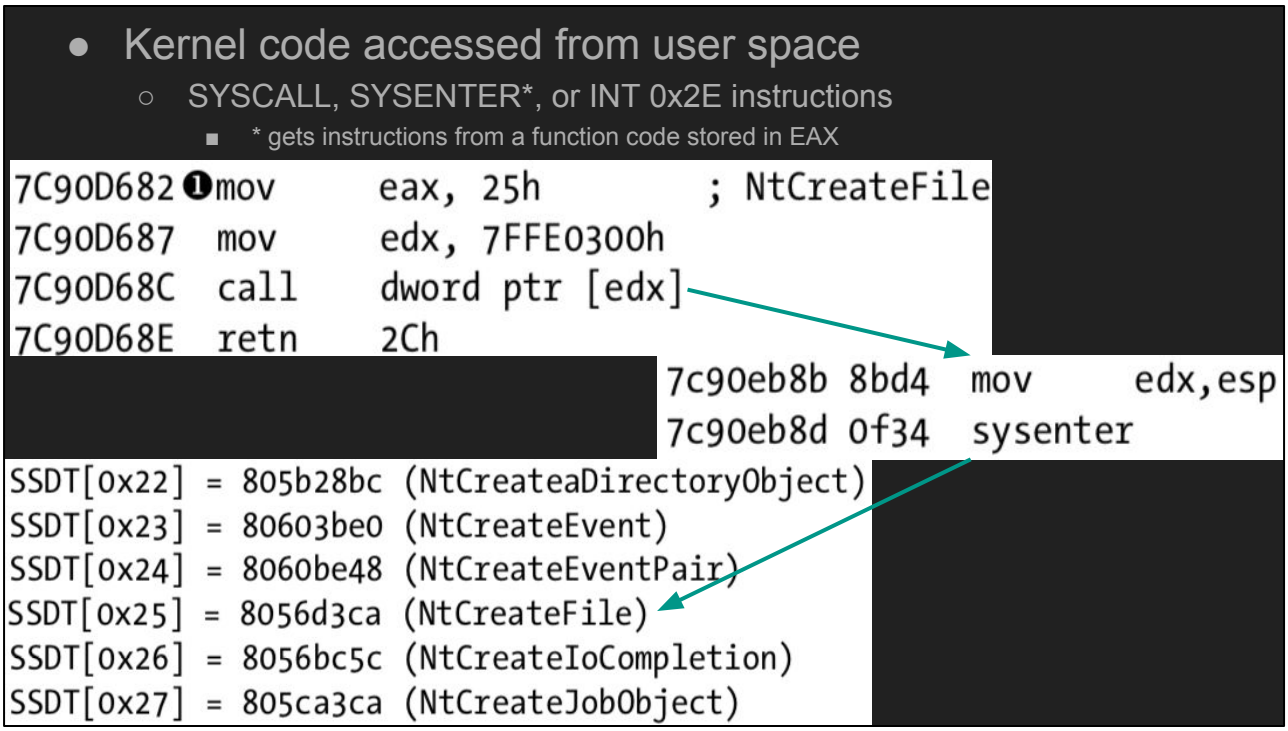
Rootkits

Modify OS to hide their existence, most by modifying kernel

- Most commonly with *System Service Descriptor Table hooking* (SSDT)
  - Easy to implement, easy to detect
  - SSDT used by Microsoft to look up function calls into the kernel
    - Not accessed by third-party code

Rootkits modify the OS to hide their existence, most by modifying kernel

Most commonly with System Service Descriptor Table hooking (SSDT)
Easy to implement, easy to detect
SSDT used by Microsoft to look up function calls into the kernel
Not usually accessed by third-party code



Kernel code accessed from user space

SYSCALL, SYSENTER*, or INT 0x2E instructions

* gets instructions from a function code stored in EAX

At (1) the value 0x25 is moved into EAX, the stack pointer is saved in EDX and then the *sysenter* instruction is called. The value in EAX will be used by *NtCreateFile* as an index into the SSDT when the code enters the kernel (the address at 0x25 in the SSDT will be called).

Hooking

A rootkit can change the value in the SSDT to point to rootkit code instead of the original function

- Normally will call original function and filter results
  - Remove handles to hide files, etc
- View SSDT with offset from *nt!KeServiceDescriptorTable*
  - Check for offsets outside of NT module boundaries
    - Where *ntoskrnl.exe* starts and ends

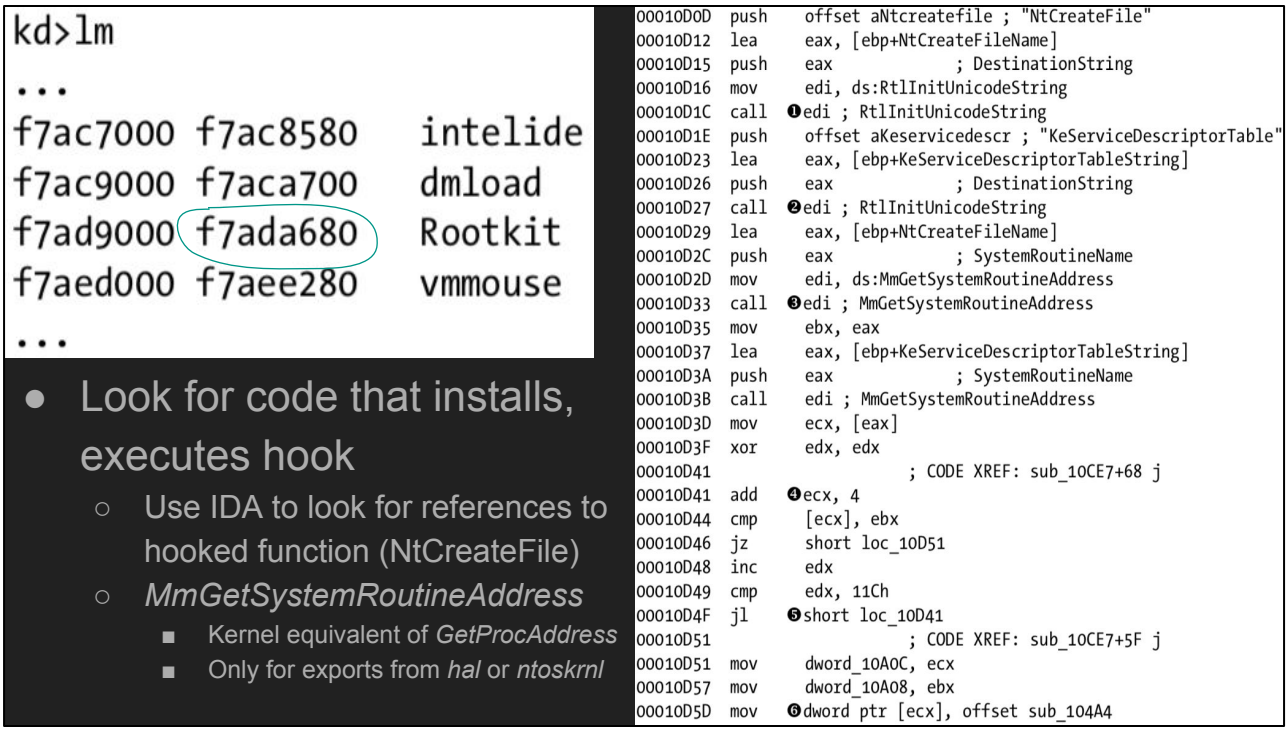
A rootkit can change the value in the SSDT to point to rootkit code instead of the original function
Normally will call original function and filter results
Remove handles to hide files, etc

Example

- NT module boundaries are 804d7000 and 806cd580

```
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 ❶f7ad94a4 8056bc5c 805ca3ca
8050129c  805ca102 80618e86 8056d4d8 8060c240
805012ac  8056d404 8059fba6 80599202 805c5f8e
```

In this example, our NT module boundaries are at 804d7000 and 806cd580, and viewing the SSDT shows one offset that is different from its surroundings.



Using the lm command to investigate open modules, we can find the malicious module that contains the offset in the previous output (0xf7ad94a4), and see that it is the ‘Rootkit’ module.

In the IDA disassembly of the rootkit on the right, the functions at (1) and (2) are used to create strings for *NtCreateFile* and *KeServiceDescriptorTable* which will be used to find the address of these functions as exported by *ntoskrnl.exe* and imported by kernel drivers (can also be loaded at runtime). The *MmGetSystemRoutineAddress* function is the kernel equivalent of *GetProceAddress*, but can only get the address of export from *hal* and *ntoskrnl* modules. At (3) a call to *MmGetSystemRoutineAddress* finds the address for *NtCreateFileName* in order to overwrite its address in the SSDT. A second call to *MmGetSystemRoutineAddress at* 0x010D3B finds the address of the SSDT.

Between (4) and (5), a loop iterates through each entry in the SSDT and compares the current address against the address for *NtCreateFile* until it is found. Once found, the instruction at (6) moves the hook’s procedure address into the space previously occupied by the *NtCreateFile* address.

The hook function:

```
000104A4  mov     edi, edi
000104A6  push    ebp
000104A7  mov     ebp, esp
000104A9  push    [ebp+arg_8]
000104AC  call    ❶sub_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
```

Depending on the value returned by the call at (1), the function will call the original *NtCreateFile* function or return 0xC0000034 (STATUS_OBJECT_NAME_NOT_FOUND). *sub_10486* checks the *ObjectAttributes* such as the filename of the file requested by the user-space program, giving a non-zero value to jump to *NtCreateFile* or 0 for an error telling the user-space program that the file does not exist.

Interrupts

Way for hardware to interfere with system events

- Driver calls *IoConnectInterrupt* to register a handle for an interrupt code
  - Specifies interrupt service routine (ISR) for OS to call on interrupt code
    - ISR info stored in Interrupt Descriptor Table (IDT)
      - View with *!idt* in WinDbg
      - Look for unnamed, unsigned drivers

```
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
```

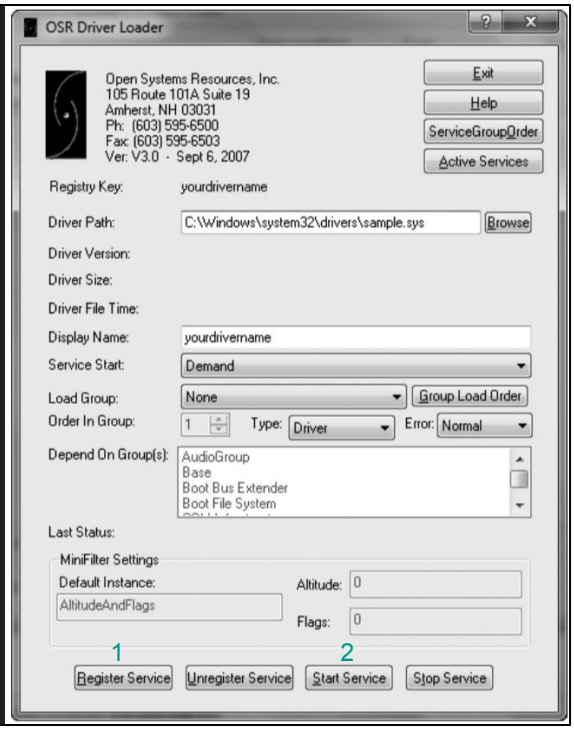
Way for hardware to interfere with system events

Driver calls *IoConnectInterrupt* to register a handle for an interrupt code
Specifies interrupt service routine (ISR) for OS to call on interrupt code
ISR info stored in Interrupt Descriptor Table (IDT)
View with !idt in WinDbg
Look for unnamed, unsigned drivers

Loading Drivers

Drivers typically loaded by user-space component

- Use OSR Driver Loader tool if no user-space component



If you are analyzing a malicious driver with no user-space component, you can manually load the driver with the OSR Driver Loader tool pictured above by specifying the driver and then registering and starting the service.

Differences in Vista, 7, and x64

- *boot.ini* -> BCDEdit program
- PatchGuard kernel protection (x64)
  - No third-party code modifies kernel
    - Code, SSDT, IDT, etc
  - Debugger breakpoints will cause crash if attached after boot
- Driving signing enforced (x64, Vista+)
  - Use BCDEdit to modify boot options for *nointegritychecks* to load unsigned drivers

boot.ini -> BCDEdit program
PatchGuard kernel protection (x64)
No third-party code modifies kernel
Code, SSDT, IDT, etc
Debugger breakpoints will cause crash if attached after boot
Driving signing enforced (x64, Vista+)
Use BCDEdit to modify boot options for nointegritychecks to load unsigned drivers

Resources

VM to VM Kernel Debugging for Fusion (Kernel debug using two Windows VMs on a Mac host)

OSR Driver Loader