GET OFF THE KERNEL IF YOU CAN'T DRIVE

DEF CON 27

WHO ARE WE

Jesse Michael

@JesseMichael

Mickey Shkatov

@HackingThings

AGENDA

- Beginning
- .
- .
- .
- .
- Conclusions
- Q&A

PRIOR WORK

- **Diego Juarez**
  - https://www.secureauth.com/labs/advisories/asus-drivers-elevation-privilege-vulnerabilities
  - https://www.secureauth.com/labs/advisories/gigabyte-drivers-elevation-privilege-vulnerabilities
  - https://www.secureauth.com/labs/advisories/asrock-drivers-elevation-privilege-vulnerabilities
- **@ReWolf**
  - https://github.com/rwfpl/rewolf-msi-exploit + Blog post link in Readme
- **@NOPAndRoll** (**Ryan Warns**) / **Timothy Harrison**
  - https://downloads.immunityinc.com/infiltrate2019-slidepacks/ryan-warns-timothy-harrison-device-driver-debauchery-msr-madness/MSR_Madness_v2.9_INFILTRATE.pptx
- @SpecialHoang
  - https://medium.com/@fsx30/weaponizing-vulnerable-driver-for-privilege-escalation-gigabyte-edition-e73ee523598b

VS

VS

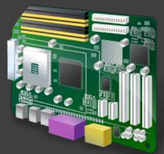
DEF CON 27

BACKGROUND

Application

Windows OS

Driver


Device

User space

Kernel space

BACKGROUND

Application

REQUEST

Windows OS

REQUEST

Driver

MAGIC

Device

DeviceIoControl(dev, ioctl, inbuf, insize, ...)

IOCTL handler in driver called with IRP struct

- contains args passed from userspace

2.3. Windows drivers

- 2.3.1. Signed
- 2.3.2. WHQL signed
- 2.3.3. EV signing cert (A Must for Win10 signing process)

DEF CON 27

HOW IT'S MADE

Briefly explain the process of signing code

KNOWN THREATS

- RWEverything
- LoJax
- Slingshot
- Game Cheats and Anti-Cheats (CapCom and others)
- MSI+ASUS+GIGABYTE+ASROCK

```
Whoami: secret\user
Found wininit.exe PID: 000002D8
Looking for wininit.exe EPROCESS...
EPROCESS: wininit.exe, token: FFFF8A06105A006B, PID: 2D8
Stealing token...
Stolen token: FFFF8A06105A006B
Looking for MsiExploit.exe EPROCESS...
EPROCESS: MsiExploit.exe, token: FFFF8A0642E3B957, PID: CAA8
Reusing token...
Whoami: nt authority\system
```

Read & Write Everything

- Utility to access almost all hardware interfaces via software
- User-space app + signed RwDrv.sys driver
- Driver acts as a privileged proxy to hardware interfaces
- Allows arbitrary access to privileged resources not intended to be available to user-space
- CHIPSEC helper to use RwDrv.sys when available

LoJax

- First UEFI malware found in the wild
- Implant tool includes RwDrv.sys driver from RWEverything
- Loads driver to gain direct access to SPI controller in PCH
- Uses direct SPI controller access to rewrite UEFI firmware

Slingshot

- APT campaign brought along its own malicious driver
- Active from 2012 through at least 2018
- Exploited other drivers with read/write MSR to bypass Driver Signing Enforcement to install kernel rootkit

Motivations

1. Privilege escalation from Userspace to Kernelspace
2. Bypass/disable Windows security mechanisms
3. Direct hardware access
    - Can potentially rewrite firmware

Attack Scenarios

1. Driver is already on system and loaded
   - Access to driver is controlled by policy configured by driver itself
   - Many drivers allow access by non-admin

2. Driver is already on system and not loaded
   - Need admin privs to load driver
   - Can also wait until admin process loads driver to avoid needing admin privs

3. Malware brings driver along with it
   - Need admin privs to load driver
   - Can bring older version of driver
   - Lojax did this for in-the-wild campaign

Finding drivers

1. Signed drivers
2. Focused on drivers from firmware/hardware vendors
3. Size (< 100KB)
4. rdmsr/wrmsr, mov crN, in/out opcodes are big hints
5. Windows Driver Model vs Windows Driver Framework

Finding drivers

Windows Driver Model

```
RtlInitUnicodeString(&DestinationString, L"\\Device\\AsrDrv101");
RtlInitUnicodeString(&SymbolicLinkName, L"\\DosDevices\\AsrDrv101");
result = IoCreateDevice(v1, 0x40u, &DestinationString, 0x22u, 0, 0, &v8);
if ( result >= 0 )
{
  v3 = IoCreateSymbolicLink(&SymbolicLinkName, &DestinationString);
  if ( v3 >= 0 )
  {
    v1->MajorFunction[IRP_MJ_CREATE] = (PDRIVER_DISPATCH)&sub_11008;
    v1->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)&sub_11008;
    v1->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATCH)ioctl_handler;
    v1->DriverUnload = (PDRIVER_UNLOAD)sub_11030;
```

Windows Driver Framework

```
result = WdfVersionBind(DriverObject, &RegistryPath, &WdfVersion, &WdfDriverGlobals);
```

```
WdfVersion      dd 30h                  ; DATA XREF: sub_140001000+4↑o
                                        ; sub_140001000+17↑o ...
                dd 0
                dq offset aKmdflibrary  ; "KmdfLibrary"
                dd 1                    ; WdfMajorVersion
                dd 9                    ; WdfMinorVersion
                dd 1DB0h                ; WdfBuildNumber
                dd 18Ch                 ; NumWdfFunctions
                dq offset WdfFunctions  ; Pointer to array of Functions to be filled by WDF Library
```

Finding drivers

IoCreateDevice vs. WdmlibIoCreateDeviceSecure

Security Descriptor Definition Language (SDDL)

- Used to specify security policy for driver

Example:

- D:P(A;;GA;;;SY)(A;;GA;;;BA)

DACL that allows:

- GENERIC_ALL to Local System
- GENERIC_ALL to Built-in Administrators

Finding drivers

- Spent 2 weeks looking for drivers
- We skimmed though hundreds of files
- At least 42 vulnerable signed x64 drivers
- Found others since ¯_(ツ)_/¯

NOW WHAT

What can we do from user space with a bad driver?

- Physical memory access
- MMIO
- MSR Read & Write
- Control register access
- PCI device access
- SMBUS
- And more...

Arbitrary Ring0 memcpy

- Can be used to patch kernel code and data structures
  - Steal tokens, elevate privileges, etc
  - PatchGuard can catch some modifications, but not all

```
inbuf = (inbuf_memcpy_struct *)a2->AssociatedIrp.SystemBuffer;
a2->IoStatus.Information = 0i64;
if ( inbuf )
{
  dest = inbuf->dest;
  size = inbuf->size;
  src = inbuf->src;
  DbgPrint("Dest=%x,Src=%x,size=%d", inbuf->dest, inbuf->src, (unsigned int)size);
  if ( (_DWORD)size )
  {
    v6 = src - dest;
    v7 = size;
    do
    {
      v8 = (dest++)[v6];
      --v7;
      *(dest - 1) = v8;
    }
    while ( v7 );
  }
  result = 0i64;
}
```

Arbitrary Physical Memory Write

- Another mechanism to patch kernel code and data structures
  - Steal tokens, elevate privileges, etc
  - PatchGuard can catch some modifications, but not all

```
mapped_addr = MmMapIoSpace((PHYSICAL_ADDRESS)ioctl_inbuf->phys_addr, ioctl_inbuf->size, 0);
copy_of_mapped_addr = mapped_addr;
if ( mapped_addr )
{
  src_ptr = (char *)ioctl_inbuf->virt_addr;
  bytes_left = ioctl_inbuf->size;
  dst_ptr = (char *)mapped_addr;          // physical address remapped into virtual address space
  while ( bytes_left )
  {
    item_size = ioctl_inbuf->item_size;   // copy by dwords, words, or bytes
    if ( item_size )                      // item_size = 0 means copy byte-by-byte
    {
      item_size_sub_1 = item_size - 1;
      if ( item_size_sub_1 )              // item_size = 1 means copy word-by-word
      {
        if ( item_size_sub_1 == 1 )       // item_size = 2 means copy dword-by-dword
        {
          dword_val = *(_DWORD *)src_ptr;
          src_ptr += 4;
          *(_DWORD *)dst_ptr = dword_val;
          dst_ptr += 4;
          bytes_left -= 4;
```

- Can also be used to perform MMIO access to PCIe and other devices

Lookup Physical Address from Virtual Address

- Useful when dealing with IOCTLs that provide Read/Write using physical addresses

```
signed __int64 __fastcall ioctl_get_phys_from_virt(__int64 a1, _IRP *a2)
{
  _QWORD *v2; // rbp@1
  _IRP *v3; // rsi@1
  __int64 virt_addr; // rdi@1
  __int64 phys_addr; // rax@1
  unsigned int v6; // ebx@1
  signed __int64 result; // rax@2

  v2 = a2->AssociatedIrp.SystemBuffer;
  a2->IoStatus.Information = 0i64;
  v3 = a2;
  virt_addr = *v2;
  DbgPrint("Default VA=%x", *v2);
  LODWORD(phys_addr) = MmGetPhysicalAddress(virt_addr);
  v6 = phys_addr;
  DbgPrint("Physical Address=%x,dwLins=%x", phys_addr, virt_addr);
  if ( v6 )
  {
    DbgPrint("Physical Address=%x", v6);
    *(_DWORD *)v2 = v6;
    v3->IoStatus.Information = 4i64;
    result = 0i64;
  }
  else
  {
    result = STATUS_INVALID_PARAMETER;
  }
  return result;
}
```

Arbitrary MSR Read

Model Specific Registers

- Originally used for "experimental" features not guaranteed to be present in future processors
- Some MSRs have now been classified as architectural and will be supported by all future processors
- MSRs can be per-package, per-core, or per-thread
- Access to these registers are via rdmsr and wrmsr opcodes
- Only accessible by Ring0

```
if ( ioctl_num == 0x9C402084 )
{
  v11 = readmsr_wrapper(
          irp->AssociatedIrp.SystemBuffer,
          irsp->Parameters.DeviceIoControl.InputBufferLength,
          irp->AssociatedIrp.SystemBuffer,
          irsp->Parameters.DeviceIoControl.OutputBufferLength,
          iostatus_info_ptr);
  goto LABEL_59;
}
```

```
__int64 __fastcall readmsr_wrapper(inbuf_msr_struct *inbuf, __int64 inbuf_size, _QW
{
  unsigned __int64 msr_value; // rax@1

  msr_value = __readmsr(inbuf->msr_addr);
  *outbuf = ((unsigned __int64)HIDWORD(msr_value) << 32) | (unsigned int)msr_value;
  *outbuf_size = 8;
  return 0i64;
}
```

Arbitrary MSR Write

Security-critical architectural MSRs

- STAR (0xC0000081)
  - SYSCALL EIP address and Ring 0 and Ring 3 Segment base
- LSTAR (0xC0000082)
  - The kernel's RIP for SYSCALL entry for 64 bit software
- CSTAR (0xC0000083)
  - The kernel's RIP for SYSCALL entry in compatibility mode

```
if ( ioctl_num == 0x9C402088 )
{
  v11 = writemsr_wrapper(
          irp->AssociatedIrp.SystemBuffer,
          irsp->Parameters.DeviceIoControl.InputBufferLength,
          irp->AssociatedIrp.SystemBuffer,
          irsp->Parameters.DeviceIoControl.OutputBufferLength,
          iostatus_info_ptr);
  goto LABEL_59;
}
```

Entrypoints used in transition from Ring3 to Ring0

```
__int64 __fastcall writemsr_wrapper(inbuf_msr_struct *inbuf, __int64 inbuf_size, void *outbuf, _
{
  unsigned __int64 v5; // rdx@1

  v5 = (unsigned __int64)inbuf->msr_value >> 32;
  __writemsr(inbuf->msr_addr, LODWORD(inbuf->msr_value), HIDWORD(inbuf->msr_value));
  *iostatus_info_ptr = 0;
  return 0i64;
}
```

Arbitrary Control Register Read

CR0 contains key processor control bits:

- PE: Protected Mode Enable
- WP: Write Protect
- PG: Paging Enable

CR3 = Base of page table structures

CR4 contains additional security-relevant control bits:

- UMIP: User-Mode Instruction Prevention
- VMXE: Virtual Machine Extensions Enable
- SMEP: Supervisor Mode Execution Protection Enable
- SMAP: Supervisor Mode Access Protection Enable

```
if ( ioctl_inbuf->which_cr )
{
  switch ( ioctl_inbuf->which_cr )
  {
    case 2:
      cr_value = __readcr2();
      break;
    case 3:
      cr_value = __readcr3();
      break;
    case 4:
      cr_value = __readcr4();
      break;
    default:
      if ( ioctl_inbuf->which_cr != 8 )
      {
        a2->IoStatus.Information = 0i64;
        a2->IoStatus.Status = STATUS_UNSUCCESSFUL;
        goto LABEL_135;
      }
      cr_value = __readcr8();
      break;
  }
}
else
{
  cr_value = __readcr0();
}
ioctl_inbuf->cr_value = cr_value;
```

Arbitrary Control Register Write

CR0 contains key processor control bits:
- PE: Protected Mode Enable
- WP: Write Protect
- PG: Paging Enable

CR3 = Base of page table structures

CR4 contains additional security-relevant control bits:
- UMIP: User-Mode Instruction Prevention
- VMXE: Virtual Machine Extensions Enable
- SMEP: Supervisor Mode Execution Protection Enable
- SMAP: Supervisor Mode Access Protection Enable

```
if ( ioctl_inbuf->which_cr )
{
  switch ( ioctl_inbuf->which_cr )
  {
    case 3:
      __writecr3(ioctl_inbuf->cr_value);
      break;
    case 4:
      __writecr4(ioctl_inbuf->cr_value);
      break;
    case 8:
      __writecr8(ioctl_inbuf->cr_value);
      break;
    default:
      a2->IoStatus.Status = STATUS_UNSUCCESSFUL;
      break;
  }
}
else
{
  __writecr0(ioctl_inbuf->cr_value);
}
```

Arbitrary IO Port Write

- How dangerous this is depends on what's in the system
  - Servers may have ASPEED BMC with Pantdown vulnerability which provides read/write into BMC address space via IO port access
  - Laptops likely have embedded controller (EC) reachable via IO port access
- Can potentially be used to perform legacy PCI access by accessing ports 0xCF8/0xCFC

```
if ( ioctl_num == 0x9C40A0C8 || ioctl_num == 0x9C40A0D8 || ioctl_num
{
  ioctl_inbuf = (inbuf_out_struct *)irp->AssociatedIrp.SystemBuffer;
  port_num = ioctl_inbuf->port_num;
  if ( ioctl_num == 0x9C40A0D8 )
  {
    __outbyte(port_num, ioctl_inbuf->port_value);
    goto LABEL_65;
  }
  if ( ioctl_num == 0x9C40A0DC )
  {
    __outword(port_num, ioctl_inbuf->port_value);
    goto LABEL_65;
  }
  if ( ioctl_num == 0x9C40A0E0 )
  {
    __outdword(port_num, ioctl_inbuf->port_value);
    goto LABEL_65;
  }
```

Arbitrary Legacy PCI Write

- How dangerous this is depends on what's in the system
- Issues with overlapping PCI device BAR over memory regions
  - Overlapping PCI device over TPM region
  - Memory hole attack

```
_disable();
__outdword(
  0xCF8u,
  (unsigned __int8)(ioctl_inbuf->offset & 0xFC)
+ ((ioctl_inbuf->func
  + 8
  * (ioctl_inbuf->dev + 32
                    * (ioctl_inbuf->bus + (((((unsigned int)ioctl_inbuf->offset >> 8) & 0xF) + 128) << 8)))) << 8));
__outdword((ioctl_inbuf->offset & 3) + 0xCFC, ioctl_inbuf->write_value);
```

CAN I DO IT TOO?

- Can we get our own code signing cert?
  - Process and cost.
  - Legality

Putting it all together

High-level steps to escalate from Ring3 to Ring0 via MSR access

- Allocate buffer for Ring0 payload
- Read LSTAR MSR to find address of kernel syscall handler
- Generate payload that immediately restores LSTAR MSR and performs malicious Ring0 actions
- Write address of payload to LSTAR MSR
- Payload immediately executes in Ring0 on next syscall entry

It's a little more complicated than that...

Supervisor Mode Execution Prevention (SMEP)

- Feature added to CPU to prevent kernel from executing code from user pages
- Attempting to execute code in user pages when in Ring0 causes page fault
- Controlled by bit in CR4 register

Need to read CR4, clear CR4.SMEP bit, write back to CR4

  - This can be done via Read/Write CR4 IOCTL primitive or via ROP in payload

It's a little more complicated than that...

- Payload starts executing in Ring0, but hasn't switched to kernelspace yet
  - Need to execute swapgs as first instruction
  - Also need to execute swapgs before returning from kernel payload

- Kernel Page Table Isolation (KPTI)
  - New protection to help mitigate Meltdown CPU vulnerability
  - Separate page tables for userspace and kernelspace
  - Need to find kernel page table base and write that to CR3
  - We can use CR3 read IOCTL to leak Kernel CR3 value when building payload

DEF CON 27

IS THERE HOPE?

- AV industry
  - What good is an AV when you can bypass it, and how can the AV help stop this lunacy.
- Microsoft
  - Virtualization-based Security (VBS)
  - Hypervisor-enforced Code Integrity (HVCI)
  - Device Guard
  - Black List

Automating Detection

- Manually searching drivers can be tedious
- Can we automate the process?
- Symbolic execution with angr framework
  - Got initial script working in about a day
  - Works really well in some cases
  - Combinatorial state explosion in others

Automating Detection

- Testing out the idea...
  - Load the driver into angr
  - Create a state object to start execution at IOCTL handler

```
import angr
import claripy

irp_addr          = 0x3000000
ioctl_inbuf_addr = 0x4000000

ioctl_handler_addr = 0x110d8
wrmsr_addr         = 0x114ac

p = angr.Project("WinRing0x64.sys", auto_load_libs=False)
state = p.factory.call_state(addr=ioctl_handler_addr)
```

Automating Detection

- Testing out the idea...
  - Create symbolic regions for parts of IRP
  - Store those into symbolic memory
  - And set appropriate pointers in execution state

```
irp_buf = claripy.BVS('irp', 8*0xd0).reversed
state.memory.store(irp_addr, irp_buf)

ioctl_inbuf = claripy.BVS('ioctl_inbuf', 1024).reversed
state.memory.store(ioctl_inbuf_addr, ioctl_inbuf)

state.regs.rdx = irp_addr
state.mem[state.regs.rdx+0x18].uint64_t = ioctl_inbuf_addr
```

Automating Detection

- Testing out the idea...
  - Create simulation manager based on state
  - Explore states trying to reach the address of WRMSR opcode
  - If found, show where the WRMSR arguments came from

```
sm = p.factory.simulation_manager(state)
sm.explore(find=wrmsr_addr)

if sm.found:
    f = sm.found[0]

    print("RIP: %x" % f.solver.eval(f.regs.rip))
    print("MSR ADDR: symbolic=%s, value=%s" % (f.regs.ecx.symbolic, f.regs.ecx))
    print("MSR High DWORD: symbolic=%s, value=%s" % (f.regs.edx.symbolic, f.regs.edx))
    print("MSR Low DWORD: symbolic=%s, value=%s" % (f.regs.eax.symbolic, f.regs.eax))
```

Automating Detection

- It worked!
  - Completed in less than five seconds
  - WRMSR address and value are both taken from input buffer

```
(angr) jesse@demo:~$ time python3 wormhole.py
[ ... snipped many angr warnings ... ]
RIP: 114ac
MSR ADDR: symbolic=True, value=<BV32 ioctl_inbuf_2_1024[31:0]>
MSR High DWORD: symbolic=True, value=<BV32 ioctl_inbuf_2_1024[95:64]>
MSR Low DWORD: symbolic=True, value=<BV32 ioctl_inbuf_2_1024[63:32]>

real    0m4.450s
user    0m3.928s
sys     0m0.523s
(angr) jesse@demo:~$
```

Automating Detection

- We can also automatically find IOCTL handler function
  - Set memory write breakpoint on drvobj->MajorFunction[14]
  - Explore states forward from driver entry point

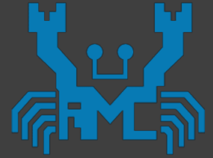
```
def mem_write_hook(state):
    ioctl_handler_addr = state.solver.eval(state.inspect.mem_write_expr)

state = p.factory.entry_state()

drv_obj_buf = claripy.BVS('driver_object', 8*0x150).reversed
state.memory.store(drv_obj_addr, drv_obj_buf)
state.regs.rcx = drv_obj_addr

state.inspect.b('mem_write', mem_write_address=drv_obj_addr+0xe0, when=angr.BP_AFTER, action=mem_write_hook)

sm = p.factory.simulation_manager(state)
sm.explore(n=500)
```

Automating Detection

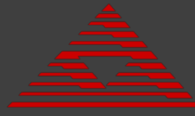
- Problems...
  - Current code only supports WDM drivers
  - Have some ideas how to support WDF drivers
  - Angr uses VEX intermediate representation lifting
    - VEX is part of Valgrind
    - Has apparently never been used to analyze privileged code
    - Decode error on rdmsr/wrmsr, read/write CR, read/write DR opcodes
  - Some drivers cause it blow up and use 64GB of ram

DISCLOSURES

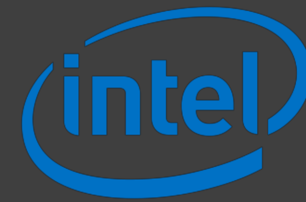
DISCLOSURES



NVIDIA



American
Megatrends



TOSHIBA

Getac

EVGA

REDACTED

DISCLOSURES

- Ask Microsoft what's their policy regarding bad drivers
- Not a security issue, open a regular ticket
- This might be an issue, are you sure?
- Meh, Not an issue
- Are you REALLY, REALLY, sure?
- Ok, let us check
- …
- Ok, We will do something about it
- THANK YOU!

DISCLOSURES

- Sent disclosure Friday 5pm
- Response came back Saturday morning
- Fix ready to start deployment in 6 weeks

DEF CON 27

DISCLOSURES

ASRock®

All the primitives in one driver

- Physical and virtual memory read/write
- Read/Write MSR
- Read/Write CR
- Legacy Read/Write PCI via IN/OUT
- IN/OUT

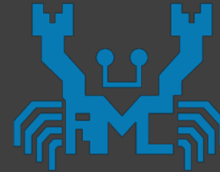
DEF CON 27

ADVISORIES

Vendor	Date	Advisory
Intel	July 9, 2019	https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00268.html
Huawei	July 10, 2019	https://www.huawei.com/fr/psirt/security-advisories/huawei-sa-20190710-01-pcmanager-en
Phoenix	TBD	TBD
REDACTED	Aug 13, 2019	TBD
REDACTED	TBD	TBD

NO RESPONSE

TOSHIBA



Microsoft Statement

Conclusions

- Bad drivers can be immensely dangerous
- Risk remains when old drivers can still be loaded by Windows
- We want to kill off this entire bug class

Code release

- GitHub release of all of our code
  - https://github.com/eclypsium/Screwed-Drivers

Questions?

DEF CON 27