GET OFF THE KERNEL IF YOU CAN'T DRIVE

DEF CON

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
- @FuzzySec
  - https://www.fuzzysecurity.com/tutorials/expDev/23.html

REFERENCES

- @matrosov
  - https://medium.com/@matrosov/dangerous-update-tools-c246f7299459
- Matt Graeber
  - https://posts.specterops.io/threat-detection-using-windows-defender-application-control-device-guard-in-audit-mode-602b48cd1c11
- Dave Weston
  - https://github.com/dwizzzle/Presentations/blob/master/Bluehat%20Shanghai%20-%20Advancing%20Windows%20Security.pdf
- Gal Diskin
  - https://media.paloaltonetworks.com/lp/endpoint-security/blog/a-brief-analysis-of-microsoft-patchguard-msr-protection.html
- Cr4sh
  - https://github.com/Cr4sh/fwexpl

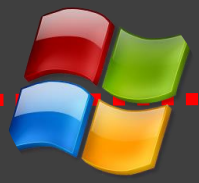
VS

VS

DEF CON

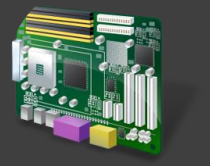
BACKGROUND

Application



Windows OS

Driver



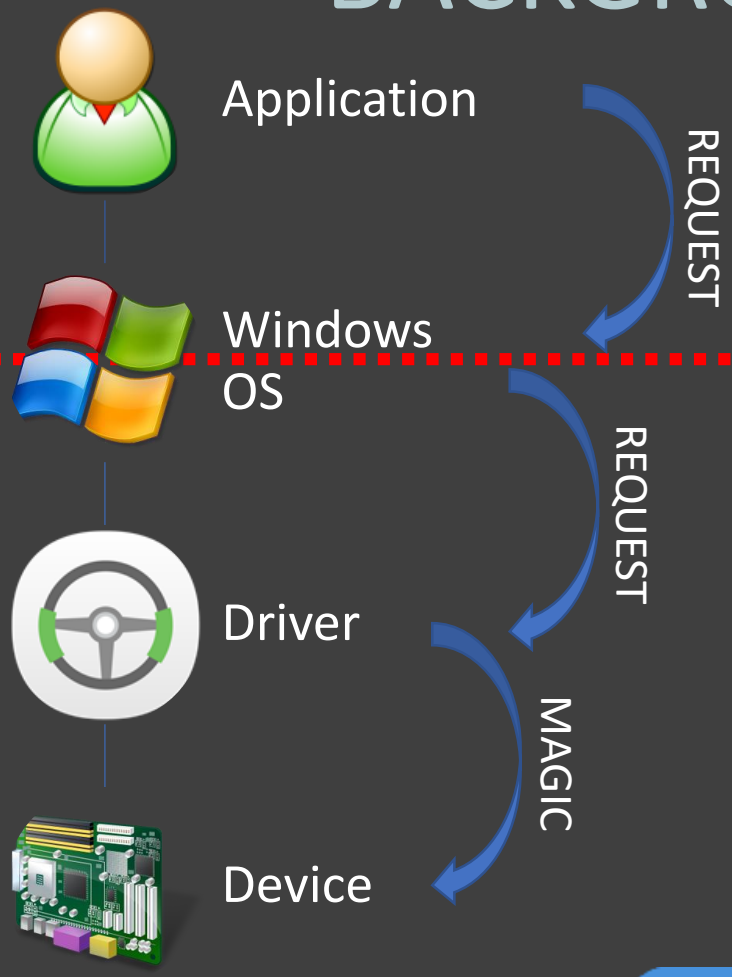
Device

User space

Kernel space

DEF CON

BACKGROUND



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

IOCTL handler in driver called with IRP struct

- contains args passed from userspace

- Windows drivers
  - Signed
  - WHQL signed
  - New EV signing cert (A Must for Win10 signing process)

New

Microsoft Signatures for kernel mode drivers

New requirements

- During Windows 10 all kernel mode drivers need to be signed by Microsoft
  - These signatures are only available by submitting to Sysdev
- Extended Validation (EV) certificates are required to create new submissions
  - It's easy for the "bad guys" to sign kernel-mode code today; we're raising the bar
  - EV certificates better validate your identity and are much harder to steal
    - this means **less malware** on our end-user's machines

http://video.ch9.ms/sessions/winhec/2015/files/DDF202%20-%20Introduction%20to%20Windows%20Driver%20Signing,%20Publishing,%20Distribution%20and%20Servicing.pptx

New

Microsoft Signatures for kernel mode drivers

New requirements

- During Windows 10 all kernel mode drivers need to be signed by Microsoft
  - These signatures are only available by submitting to Sysdev
- Extended Validation (EV) certificates are required to create new submissions
  - It's easy for the "bad guys" to sign kernel-mode code today; we're raising the bar
  - EV certificates better validate your identity and are much harder to steal
    - this means **less malware** on our end-user's machines

Only New drivers are affected

- Drivers which are signed prior to Windows 10 RTM will ignore this change
- Drivers for all previous releases of Windows will be unaffected
- User mode drivers are unaffected

http://video.ch9.ms/sessions/winhec/2015/files/DDF202%20-%20Introduction%20to%20Windows%20Driver%20Signing,%20Publishing,%20Distribution%20and%20Servicing.pptx


DRIVER SIGNING
CERTIFICATE EXPIRED
LOADS IT ANYWAY
imgflip.com

DEF CON

GETTING OUR OWN

Get started with the Hardware Developer Program

The Windows Hardware Developer Program allows you to certify your hardware for Windows and sign and publish your drivers to Windows Update.

- You must have an Extended Validation (EV) code signing certificate. Please check whether your company already has a code signing certificate. If your company already has a certificate, have the certificate available. You will need the certificate to sign files. If your company does not have a certificate, you will need to buy one as part of the registration process.

- You will need to sign in as a global administrator in your organization's Azure Active Directory. If you do not know whether your organization has an Azure Active Directory, contact your IT department. If your organization does not have an Azure Active Directory, you will be able to create one for free in the next step.

- You must have the authority to sign legal agreements on behalf of your organization.

GETTING OUR OWN

Get started with the Ha[illegible]er Program

The Windows Hardware Developer Program allow[illegible] your hardware for Windows and sign and publish your drivers to Windows Update.

- You must have an Extended Validation (EV) [illegible]ate. Please check whether your company already has a code signing certificate. If your [illegible]as a certificate, have the certificate available. You will need the certificate to sign files. If y[illegible]ot have a certificate, you will need to buy one as part of the registration process.

- You will need to sign in as a global adm[illegible]s Azure Active Directory. If you do not know whether your organization has an [illegible]act your IT department. If your organization does not have an Azure Active Directory, [illegible]one for free in the next step.

- You must have the authority to sign legal agreements on behalf of your organization.

DEF CON

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

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

- Privilege escalation from Userspace to Kernelspace
- Bypass/disable Windows security mechanisms
- Direct hardware access
  - Can potentially modify system and device firmware
  - Still have lots of issues with unsigned firmware

Attack Scenario #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

Attack Scenario #2

Driver is already on system and not loaded

- Need admin privileges to load driver
- Load driver via signed app with UAC from trusted vendor
- Can also wait until admin process loads driver to avoid needing admin privileges

Attack Scenario #3

Malware brings driver along with it

- Need admin privileges to load driver
- Load driver via signed app with UAC from trusted vendor
- Can bring older version of driver
- LoJax did this for in-the-wild campaign
  - Modified UEFI firmware to install persistent rootkit

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?

- Kernel virtual memory access
- Physical memory access
- MMIO access
- MSR access
- Control Register access
- PCI device access
- SMBUS access
- 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 )
  {
    src_dst_delta = src - dest;
    bytes_left = size;
    do
    {
      byte_val = (dest++)[src_dst_delta];
      --bytes_left;
      *(dest - 1) = byte_val;
    }
    while ( bytes_left );
  }
  result = 0i64;
}
```

Arbitrary Physical Memory Write

- Can perform MMIO access to PCIe and other devices
- Another mechanism to patch kernel code and data structures
  - Steal tokens, elevate privileges, etc
  - PatchGuard can catch some modifications, but not all
  - Partial mitigation in Win 10 1803

```
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;
```

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

- Impact is platform dependent
  - Can potentially be used to modify UEFI and device firmware
  - Servers may have ASPEED BMC with Pantdown vulnerability which provides read/write into BMC address space
  - 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

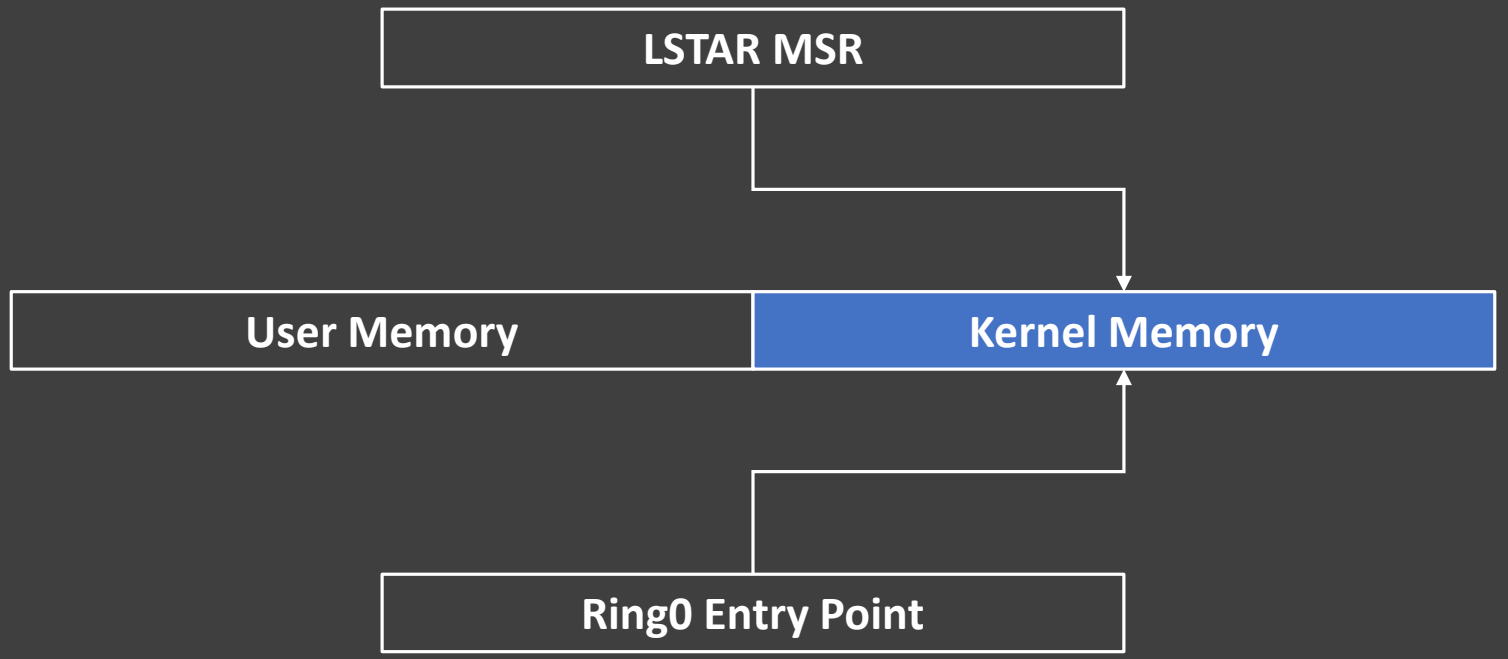
```
if ( ioctl_num == 0x9C40A0C8 || ioctl_num == 0x9C40A0D8 || ioctl_num
{
  ioctl_inbuf = (inbuf_out_struct *)irp->AssociatedIrp.SystemBuffer;
  port_num = ioctl_inbuf->port_num;
  if ( ioctl_num == 0x9C40A0D8 )
  {
    __outbyte(port_num, ioctl_inbuf->port_value);
    goto LABEL_65;
  }
  if ( ioctl_num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

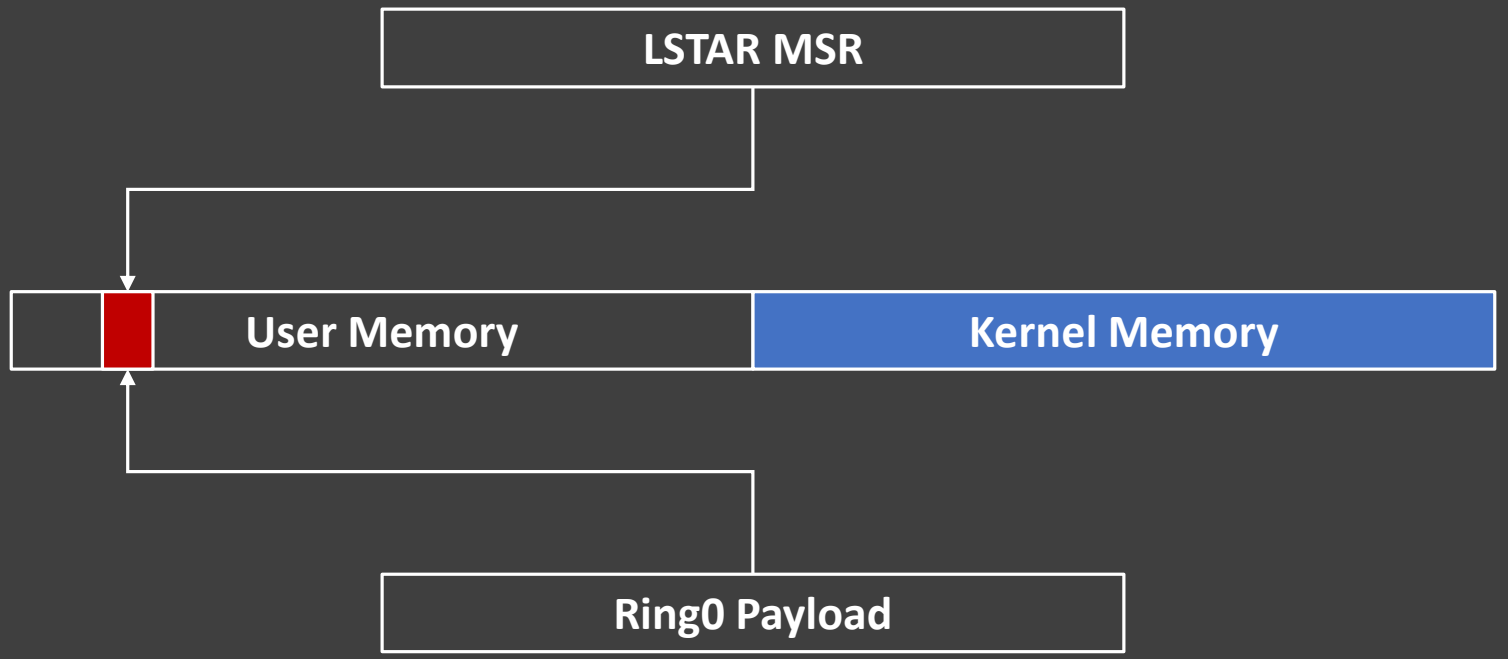
- Impact is platform dependent
  - Can potentially be used to modify UEFI and device firmware
- Issues with overlapping PCI device BAR over memory regions
  - Overlapping PCI device over TPM region
  - Memory hole attack

```
_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);
```

Kernel Code Execution via MSR



Kernel Code Execution via MSR



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

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 WDM 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

- Automatically find IOCTL number and other constraints
  - IOCTL num is at known offset in IRP
  - Constraint tracking is very useful
    - Can get spammed with overly complex constraints
    - Angr can simplify constraints for you

```
[AsrDrv101.sys] Attempting to find path from 110a8 to WrCR at 11731
[AsrDrv101.sys] Found path from 110a8 to 11731
RIP: 11731
IOCTL NUM: 222870 from <BV32 irsp_params_ioctl_num_337_32>
Found write to control register with arbitrary value!
Write CR: target=cr4, symbolic=True, value=<BV64 ioctl_inbuf_328_8192[127:64]>
Constraints:
  Input Buffer: <Bool ioctl_inbuf_328_8192[31:0] != 0x0>
  Input Buffer: <Bool ioctl_inbuf_328_8192[31:0] != 0x3>
  Input Buffer: <Bool (0xfffffffd + ioctl_inbuf_328_8192[31:0]) == 0x1>
```

Automating Detection

- Problems...
  - Angr uses VEX intermediate representation lifting
    - Has apparently never been used to analyze privileged code
    - Decode error on rdmsr/wrmsr, read/write CR, read/write DR opcodes
    - Can implement missing opcodes with Gymrat spotter

Automating Detection

- Problems...
  - Current code only supports WDM drivers
  - Have some ideas how to find WDF ioctl handlers
    - Hook WdfVersionBind to fill WdfFunctions[]
    - Hook WdfFunctions[WdfIoQueueCreate]
  - Some drivers cause it to blow up and run out of memory

DISCLOSURES

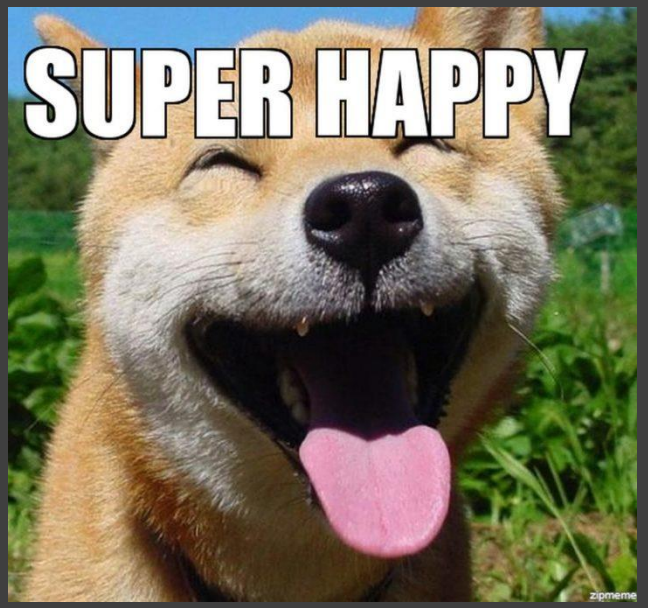
DISCLOSURES

DEF CON

DISCLOSURES

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

"Phoenix Technologies Ltd. has made available to its customers an updated version of its WinFlash driver, revoked prior certificates and assigned new certificates."



DISCLOSURES

soc@us-cert.gov
cert@cert.org

DISCLOSURES

Microsoft

- 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!

DEF CON

DISCLOSURES

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

DISCLOSURES

NO RESPONSE

TOSHIBA

ADVISORIES

Vendor	Date	Advisory
Phoenix	Jun 21, 2019	TBD
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
Insyde	Aug 10, 2019	TBD
REDACTED	Aug 13, 2019	TBD
REDACTED	TBD	TBD

Statements

To: Eclypsium
From: Insyde Software
Date: **Embargoed until 3pm PST August 10, 2019**

DEF CON 27 Aug. 8-11, 2019 Las Vegas

At Insyde Software, we applaud Eclypsium for their efforts to identify vulnerable firmware in the supply chain of enterprise servers and work with suppliers and industry partners to mitigate these issues.

In the specific case of the "wormhole" vulnerability, Insyde appreciates Eclypsium's responsible reporting of this issue and allowing us necessary time to prepare our resolutions and disclosure.

After receiving Eclypsium's report, our engineers reviewed the issue and started a fresh study of our drivers and applications that use the impacted drivers. We followed Microsoft's updated Windows driver guidelines to redesign our applications and drivers. We also reduced the overall access requirements of our applications. New versions of our application packages with these and other security enhancements were released to our customers starting last month. We continue to work towards a full resolution for all platforms impacted.

Insyde Software takes the responsibility of the security of our firmware technology very seriously and encourages all security researchers to responsibly report security issues directly to security.report@insyde.com

The Insyde Software Security Team

Statements

- Microsoft has a strong commitment to security and a demonstrated track record of investigating and proactively updating impacted devices as soon as possible. For the best protection, we recommend using Windows 10 and the Microsoft Edge browser.

- In order to exploit vulnerable drivers, an attacker would need to have already compromised the computer. To help mitigate this class of issues, Microsoft recommends that customers use Windows Defender Application Control to block known vulnerable software and drivers.

- Customers can further protect themselves by turning on memory integrity for capable devices in Windows Security.

- Microsoft works diligently with industry partners to address to privately disclose vulnerabilities and work together to help protect customers.

Conclusions

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

Code release

- GitHub Repo Contents:
  - Angr script to find wormhole drivers
  - Example code in C#, C++ and PowerShell
  - Latest slides
  - Demo videos
  - All our links to Drivers and tools

https://github.com/eclypsium/Screwed-Drivers

We will be taking questions outside