Exploring a New Class of Kernel Exploit Primitive

Andrew Ruddick @arudd1ck

Microsoft Security Response Center (MSRC)
Vulnerabilities & Mitigations

Who Are We?

Andrew Ruddick

Security Researcher @ MSRC Vulnerabilities & Mitigations

8 years experience in low-level Windows internals, kernel development, VR, exploit development & mitigation techniques

Red Member of MSRC Purple Team

Prior Conferences: USENIX WOOT '16, 44Con `22

Rohit Mothe

Security Researcher @ MSRC Vulnerabilities & Mitigations

10 years experience in VR and exploit development on Windows platforms.

MSRC OS Mitigations Lead

Prior Conferences: BlackHat USA '16, RECON '16

Motivations

- MSRCs handling of kernel bugs
  - The bug is often clear, but the exploitability is not always.
  - We don’t require an exploit from finders, just a crashing PoC.
    - It’s sometimes hard to prove exploitability without investing longer than it would take to just fix it.
    - We don’t want to ‘put finders off’ submitting issues to us. We *want* to patch the OS.
- Proving the Negative is Hard
  - What do we do with an OOB-R where the attacker can’t retrieve the data? DoS? Info Disclosure? Is that worth us patching down-level?
  - Not all OOB-Rs are equal. An MSRC we handled got us talking. Is it possible to do better than DoS with some of these reads?

Agenda

- Blind Arbitrary Read Primitive
- Memory Mapped I/O (MMIO)
- Targeting Drivers that use MMIO
  - Enumeration / Windbg Scripting
- Reverse Engineering Drivers
- Interesting MMIO Patterns / Primitives
- What’s Next?

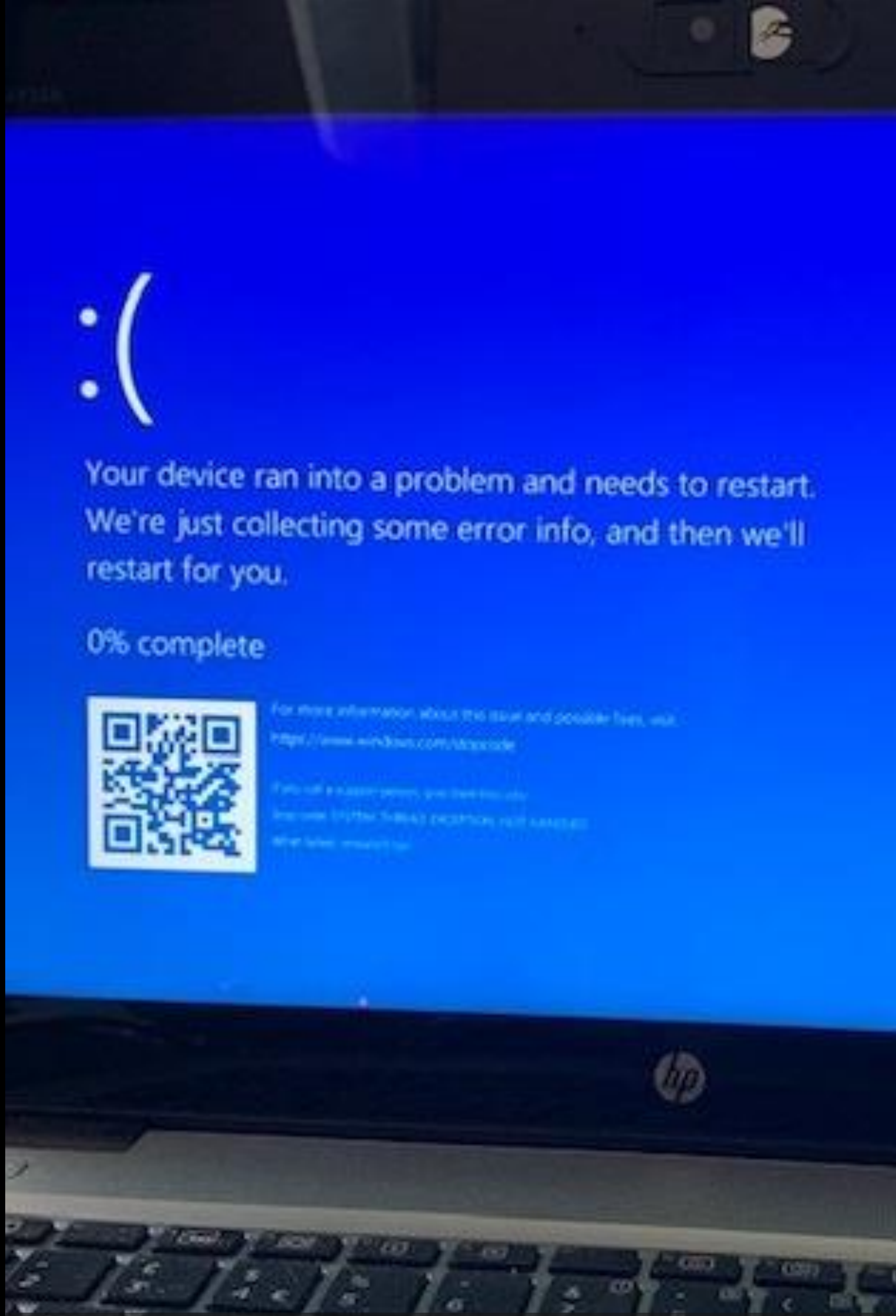
Blind Arbitrary Read Primitive

Hyper-V Host Arbitrary Read (CVE-2021-28476)

- Hyper-V Guest can cause Host to de-reference arbitrary pointer for read
- Reported independently by more than one finder
- Presented at BH USA 2021 (hAFL1: Our Journey of Fuzzing Hyper-V and Discovering an 0-day)

Acknowledgements

- Leo Adrien (@australeo)
- Daniel Fernandez Kuehr of Blue Frost Security GmbH
- Peleg Hadar (@peleghd) of SafeBreach Labs and Ophir Harpaz (@OphirHarpaz) of Guardicore Labs



MSRC 64478 Reproduction

Hyper-V Host Arbitrary Read (CVE-2021-28476)

- **Bug in Hyper-V virtual networking switch driver (vmswitch.sys)**
  - Provides virtual ethernet services to guest VMs over vmbus
  - Processes RNDIS packets from guest
- **Kernel Panic due to invalid pointer de-reference for read in vmswitch!VmsIfrInfoParams_OID_SWITCH_NIC_REQUEST**
  - For all requests to the physical NIC, this routine is called to log them
  - Logging routine accepts an NDIS_SWITCH_NIC_OID_REQUEST structure
- **If OidRequest is not NULL, it is de-referenced to access information in the request**
  - Attacker can forge this structure

Hyper-V Host Arbitrary Read (CVE-2021-28476)

_NDIS_SWITCH_NIC_OID_REQUEST

```
typedef struct _NDIS_SWITCH_NIC_OID_REQUEST {
  NDIS_OBJECT_HEADER     Header;
  ULONG                  Flags;
  NDIS_SWITCH_PORT_ID    SourcePortId;
  NDIS_SWITCH_NIC_INDEX  SourceNicIndex;
  NDIS_SWITCH_PORT_ID    DestinationPortId;
  NDIS_SWITCH_NIC_INDEX  DestinationNicIndex;
  PNDIS_OID_REQUEST      OidRequest;
} NDIS_SWITCH_NIC_OID_REQUEST, *PNDIS_SWITCH_NIC_OID_REQUEST;
```

Hyper-V Host Arbitrary Read (CVE-2021-28476)

VmsIfrInfoParams_OID_SWITCH_NIC_REQUEST

```
Static VOID VmsIfrInfoParams_OID_SWITCH_NIC_REQUEST (
    …
    _In_ PVOID Data,
    …
    )
{
    PNDIS_SWITCH_NIC_OID_REQUEST switchNicOid = (PNDIS_SWITCH_NIC_OID_REQUEST)Data;

    if ((DataLength > 0) && (DataLength >= SIZEOF_NDIS_SWITCH_NIC_OID_REQUEST))
    {
        if (switchNicOid->OidRequest != NULL)
        {
            VmsIfrLogRoutine(
                <irrelevant args>,
                "    --> Params: InnerOID=%!NDIS_OID!",
                switchNicOid->OidRequest->DATA.QUERY_INFORMATION.Oid); // Crash
```

Hyper-V Host Arbitrary Read (CVE-2021-28476)

```
1: kd> !analyze -v
…
EXCEPTION_RECORD:  fffff60589adf568 -- (.exr 0xfffff60589adf568)
ExceptionAddress: fffff8056ffd1a63 (vmswitch!VmsIfrInfoParams_OID_SWITCH_NIC_REQUEST+0x00000000000000fb)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 0000000000000000
   Parameter[1]: ffffffffffffffff
Attempt to read from address ffffffffffffffff

CONTEXT:  fffff60589adeda0 -- (.cxr 0xfffff60589adeda0)
rax=fffff80570157214 rbx=fffff805701862a0 rcx=0000000000000000
rdx=000000008f112807 rsi=fffff60589adf960 rdi=ffff8082815c0700
rip=fffff8056ffd1a63 rsp=fffff60589adf7a0 rbp=ffff8082815c07c0
 r8=0000000000000000  r9=000000000000013b r10=4141414141414141
r11=fffff60589adf770 r12=fffff805701573d0 r13=00000000c0000001
r14=00000000000021f0 r15=fffff80570157360
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0000  ds=002b  es=002b  fs=0053  gs=002b             efl=00050246
vmswitch!VmsIfrInfoParams_OID_SWITCH_NIC_REQUEST+0xfb:
fffff805`6ffd1a63 418b4a20        mov     ecx,dword ptr [r10+20h] ds:002b:41414141`41414161=????????
```

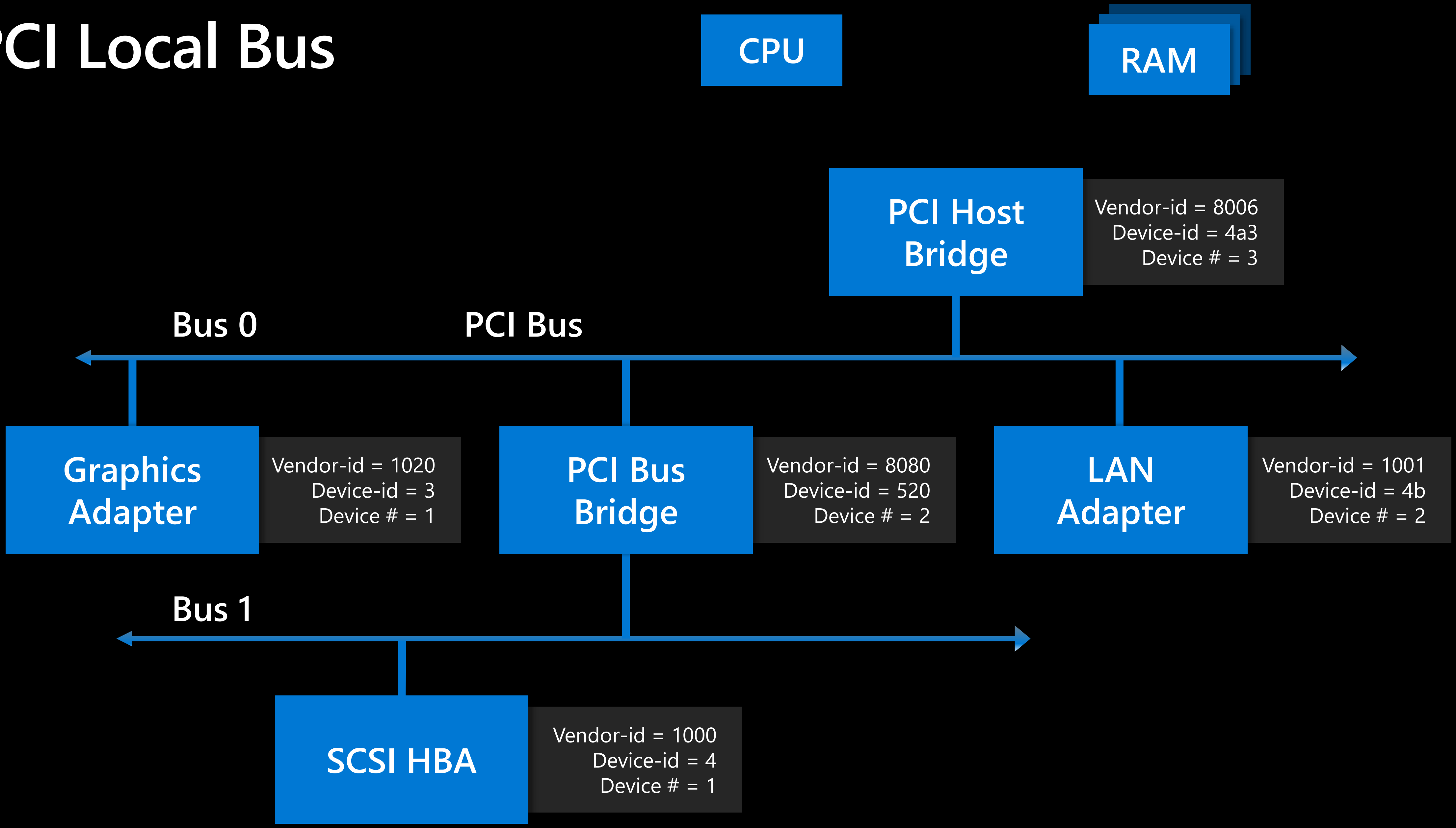
Memory Mapped I/O (MMIO)

Why MMIO?

- **Peripheral device drivers use MMIO for 2-way device communications with Firmware over the peripheral buses**
  - These are mapped (often transiently) to the Kernel Virtual Address Space (VAS)

- **Assuming an attacker knew the location of such an address, could it theoretically be targeted to corrupt the device driver ‘state machine’?**
- Could such corruptions lead to EoP / RCE?

PCI Local Bus



Programmed Input Output (PIO)

- **Two methods of I/O between CPU and PCI hardware devices**
  - **Memory Mapped I/O (MMIO)** uses general purpose memory and is accessed using the same CPU instructions
  - **Port Mapped I/O (PMIO)** uses a special class of CPU instructions (e.g. x86 in/out) and operates on a different address space. Copies bytes between EAX register and specified I/O port

- **PCI Device Registers**
  - PCI devices have a set of registers (configuration space registers) mapped to main memory
  - Base Address Registers (BARs) are mapped to an I/O memory region
    - The length of each BAR is defined by the hardware and communicated to software via the configuration registers.

"The size of the access could therefore be thought of as additional bits feeding into the device's state-change logic."

Semantics of MMIO mapping attributes across architectures (2016)

On the Linux Kernel implementation of MMIO (https://lwn.net/Articles/698014)

"For example, the MMIO read operation that reads a character from a serial input device would be expected to also remove that character from the device's internal queue, so that the next MMIO read would read the next input character. As with writes, the size of the MMIO read is significant."

Semantics of MMIO mapping attributes across architectures (2016)

On the Linux Kernel implementation of MMIO (https://lwn.net/Articles/698014)

Prior MMIO Bugs

Google P0 AMD EPYC SEV-SNP: Firmware accepts malleable MMIO Pages (here)

Intel: Processor MMIO Stale Data Vulnerabilities (here)

CanSecWest 2022: Matryoshka Trap: Recursive MMIO Flaws Lead to VM Escape (here)

Targeting Drivers that use MMIO

Enumerating Target Drivers

- Registry / Device Manager Probing
  - Script available on the MSRC blog, here
- Naïve String Scanning
  - Script is available on the MSRC blog, here
- ACPI MCFG Table
- MmMapIoSpace Interception

Registry / Device Manager Probing

- **We can manually examine device manager to find interesting devices on the system**
- **Can script extraction of MMIO ranges using WMI**
  - Win32_DeviceMemoryAddress / Win32_PNPAllocatedResource
  - Our script is available on the MSRC blog, here

Registry / Device Manager Probing

Name	DeviceID	Physical Address
Mobile 6th/7th Generation Intel(R) Processor Family I/O PCI Express Root Port #1 – 9D10	PCI\VEN_8086&DEV_9D10&SUBSYS_72708086&REV_F1\3&11583659&0&E0	{0xD4400000-0xD45FFFFF}
Intel(R) UHD Graphics 620	PCI\VEN_8086&DEV_5917&SUBSYS_00271414&REV_07\3&11583659&0&10	{0xD3000000-0xD3FFFFFF, 0xB0000000-0xBFFFFFFF}
Marvell AVASTAR Wireless-AC Network Controller	PCI\VEN_11AB&DEV_2B38&SUBSYS_045E0009&REV_00\4&32FA7CC7&0&00E0	{0xD4500000-0xD45FFFFF, 0xD4400000-0xD45FFFFF}
Intel(R) Management Engine Interface #1	PCI\VEN_8086&DEV_9D3A&SUBSYS_72708086&REV_21\3&11583659&0&B0	{0xDFBDF000-0xDFBDFFFF}

Naïve String Scanning

- **Simplest manner to locate drivers is to scan for strings**
- **Search for any driver on a system containing the string 'MMIO'**
- **A second script is available on the MSRC blog, here that can be executed in any directory to dump this list of drivers**
- **Running the Sysinternals 'strings' utility over the output gives further context on each driver**
- **Using this method, we identified 24 drivers for further analysis on our lab machines**
  - Includes components related to GPIO, I2C, DirectX / Video, Virtualization (Hyper-V), USB and OEM device-specific hardware

Naïve String Scanning

Identified Driver Images

```
C:\Windows\System32\drivers\iaStorAVC.sys
C:\Windows\System32\drivers\USBXHCI.SYS
C:\Windows\System32\drivers\Vid.sys
C:\Windows\System32\drivers\vmbkmcl.sys
C:\Windows\System32\drivers\vmbus.sys
C:\Windows\System32\drivers\dxgkrnl.sys
C:\Windows\System32\drivers\iaLPSS2i_GPIO2.sys
C:\Windows\System32\drivers\iaLPSS2i_GPIO2_BXT_P.sys
C:\Windows\System32\drivers\iaLPSS2i_GPIO2_CNL.sys
C:\Windows\System32\drivers\iaLPSS2i_GPIO2_GLK.sys
C:\Windows\System32\drivers\iaLPSS2i_I2C.sys
C:\Windows\System32\drivers\iaLPSS2i_I2C_BXT_P.sys
C:\Windows\System32\drivers\iaLPSS2i_I2C_CNL.sys
C:\Windows\System32\drivers\iaLPSS2i_I2C_GLK.sys
C:\Windows\System32\drivers\iaLPSSi_I2C.sys
```

What is ACPI?

- **Advanced Configuration & Power Interface (ACPI)**
  - Controls at the lowest level, interactions with system hardware over primary and peripheral busses
  - Brings control of firmware management operations to the OS, reducing reliance on SMM
  - Organized into tables which are stored in the registry
    - \REGISTRY\MACHINE\HARDWARE\ACPI\DSDT\A_DEVICE\A_THING\00000000
  - ACPI tables can be dumped using open-source tools from ACPI Component Architecture (ACPICA) project
  - The (optional) 'MCFG' table contains PCI config. Information, including registered MMIO ranges and PCI BARs

ACPI MCFG Table

- **ACPI Specification says:**
  - 'PCI Express memory mapped configuration space base address Description Table'
- **Contains a physical base address that details the PCI bus, device and function numbers for each PCI device on the system**
  - Can extract and decompile the MCFG table to get this physical base address
  - Output can be parsed manually, but we used the RWEverything tool to do this for us
- **Hyper-V VMs examined (at least under default configuration) don't have this registered**

ACPI Binary Table Extraction

Intel ACPICA: ACPI Binary Table Extraction Utility

```
C:\Workspace\ACPI\iasl-win-20210730>acpidump.exe > acpitabl.dat
C:\Workspace\ACPI\iasl-win-20210730>acpixtract.exe -l acpitabl.dat

 Signature  Length     Version Oem       Oem         Oem          Compiler
                               Id        TableId     RevisionId   Name
 _________  __________ ____    ________  __________  __________   _______
 01)  MCFG  0x0000003C 0x01    "ALASKA"  "A M I    " 0x01072009   "MSFT"
 02)  FACP  0x000000F4 0x04    "ALASKA"  "A M I    " 0x01072009   "AMI "
 03)  APIC  0x0000009E 0x03    "ALASKA"  "A M I    " 0x01072009   "AMI "
 04)  HPET  0x00000038 0x01    "ALASKA"  "A M I    " 0x01072009   "AMI "
 05)  FPDT  0x00000044 0x01    "ALASKA"  "A M I    " 0x01072009   "AMI "
 06)  SSDT  0x00001714 0x01    "AMD    " "POWERNOW"  0x00000001   "AMD "
 07)  XSDT  0x00000054 0x01    "ALASKA"  "A M I    " 0x01072009   "AMI "
 08)  DSDT  0x00005BC1 0x02    "ALASKA"  "A M I    " 0x00000000   "INTL"

Found 8 ACPI tables in acpitabl.dat
```

Disassembling An ACPI Table

Intel ACPICA: ACPI Binary Table Extraction Utility

```
C:\Workspace\ACPI\iasl-win-20210730>acpixtract -s MCFG acpitabl.dat

  MCFG -      60 bytes written (0x0000003C) - mcfg.dat

C:\Workspace\ACPI\iasl-win-20210730>iasl mcfg.dat

File appears to be binary: found 38 non-ASCII characters, disassembling
Binary file appears to be a valid ACPI table, disassembling
Input file mcfg.dat, Length 0x3C (60) bytes
ACPI: MCFG 0x0000000000000000 00003C (v01 ALASKA A M I    01072009 MSFT 00010013)
Acpi Data Table [MCFG] decoded
Formatted output:  mcfg.dsl - 1568 bytes
```

MCFG Table Disassembly (ASL / DSL)

Intel ACPICA: AML/ASL+ Disassembler

```
[000h 0000   4]                    Signature : "MCFG"    [Memory Mapped Configuration Table]
[004h 0004   4]                 Table Length : 0000003C
[008h 0008   1]                     Revision : 01
[009h 0009   1]                     Checksum : 84
[00Ah 0010   6]                       Oem ID : "ALASKA"
[010h 0016   8]                 Oem Table ID : "A M I"
[018h 0024   4]                 Oem Revision : 01072009
[01Ch 0028   4]              Asl Compiler ID : "MSFT"
[020h 0032   4]        Asl Compiler Revision : 00010013
[024h 0036   8]                     Reserved : 0000000000000000
[02Ch 0044   8]                 Base Address : 00000000E0000000
[034h 0052   2]         Segment Group Number : 0000
[036h 0054   1]             Start Bus Number : 00
[037h 0055   1]               End Bus Number : FF
[038h 0056   4]                     Reserved : 00000000
```

PCI Device Dump

RWEverything: PCI Device Dump

```
Bus 00, Device 02, Function 00 - ATI Technologies Inc. PCI-to-PCI Bridge (PCIE)
 ID=5A161002, SID=5A141002, Int Pin=INTA, IRQ=0B, PriBus=00, SecBus=01, SubBus=01
 MEM=FEA00000-FEAFFFFF C0000000-D07FFFFF  IO=0000E000-0000EFFF

Device/Vendor ID 0x5A161002
Revision ID      0x00
<snip>
IO Range
   0x0000E000 - 0x0000EFFF
Memory Range
   0xFEA00000 - 0xFEAFFFFF
Prefetchable Memory Range
   0xC0000000 - 0xD07FFFFF
Expansion ROM    0x00000000
Subsystem ID     0x5A141002
```

Reading / Writing to Device Registers

- **Ntoskrnl exports public interfaces from Mm executive**
  - MmMapIoSpace(Ex)
    - Maps a given physical address range to a non-paged system address space
    - Device drivers can access device registers through this mapping
    - MmMapIoSpace maps WX memory (if HVCI is turned off)
    - MmMapIoSpaceEx allows caller to specify page protections

```
PVOID MmMapIoSpace(
    [in] PHYSICAL_ADDRESS PhysicalAddress,
    [in] SIZE_T NumberOfBytes,
    [in] MEMORY_CACHING_TYPE CacheType
);
```

Reading / Writing to Device Registers

- **Ntoskrnl exports public interfaces from Mm executive**
  - MmUnmapIoSpace
    - Unmaps a specified range of physical addresses previously mapped by MmMapIoSpace

```
PVOID MmUnMapIoSpace(
    [in] PVOID BaseAddress,
    [in] SIZE_T NumberOfBytes
);
```

MmMapIoSpace(Ex) Interception

- **We can hook calls to MmMapIoSpace(Ex) to gather a list of all Physical to Virtual mappings made on the system**
- **Also hook releases via MmUnmapIoSpace**
- **Ntoskrnl.exe exports these routines, easy to locate with public symbols**
- **We provide Windbg scripted breakpoints to do this**

MmMapIoSpace Interception

Windbg MASM Scripted Breakpoints

```
bu nt!MmMapIoSpace ".block{ r $t1 = @rcx; r $t2 = @rdx; r $t3 =
@r8; .printf \"[+] MmMapIoSpace - Physical Address: %p, Size: %p,
Cache Type: %p)\\n\", @$t1, @$t2, @$t3}; gc"

bu nt!MmMapIoSpaceEx ".block{ r $t1 = @rcx; r $t2 = @rdx; r $t3 =
@r8; .printf \"[+] MmMapIoSpaceEx - Physical Address: %p, Size:
%p, Protect: %p)\\n\", @$t1, @$t2, @$t3}; gc"

bu nt!MmUnmapIoSpace ".block{ r $t1 = @rcx; .printf \"[-]
Unmapped at Virtual Address: %p\\n\", @$t1}; gc"
```

MmMapIoSpace Interception

Windbg MASM Scripted Breakpoints #2

```
bu nt!MmMapIoSpace+0x22 ".block{ r $t1 = @rax; .printf \"[+]
Mapped at Virtual Address: %p\\n\", @$t1}; gc"

bu nt!MmMapIoSpaceEx+0x30 ".block{ r $t1 = @rax; .printf \"[+]
Mapped at Virtual Address: %p\\n\", @$t1}; gc"
```

MmMapIoSpace Interception

Windbg Script Output

[+] MmMapIoSpaceEx - Physical Address: 00000000000f93d0,
Size: 000000000000439b, Protect: 0000000000000004)

[+] Mapped at Virtual Address: ffffb980cff123d0

[-] Unmapped at Virtual Address: ffffb980cff123d0

[+] MmMapIoSpaceEx - Physical Address: 00000000f7ff0300,
Size: 0000000000000024, Protect: 0000000000000204)

[+] Mapped at Virtual Address: ffffb980d04da300

[-] Unmapped at Virtual Address: ffffb980d04da300

MMIO Ranges Remaining After Boot

MMIO Range Interception

```
0xfffff67c84600000 // nt  size 0x300000
0xffffe68063040000 // BOOTVID  size 0x20000
0xffffe68063616000 // BasicDisplay  size 0x20000
0xffffe6806343e4f0 // vmgencounter  size 0x10
0xffffe680631e8064 // ACPI size 0xff
0xffffe680631ff000 // winhv size 0x1000
0xffffe6806336c000 // fvevol size 0x4000
0xfffff67c89000000 // DXGKrnl   size 0x300000
<…>
```

Reverse Engineering

Drivers Noted

- **Some we confirmed to Contain MMIO**
  - Iacamera64.sys
  - SurfaceHotPlug.sys
  - USBXHCI.sys
    - Reads MMIO range values, then immediately resets that range location.
    - Could be susceptible to double-fetch, if on a valid device stack?
  - vid.sys (MmioGpaRange for emulation of MMIO device registers)
  - iaLPSS2i_GPIO2.sys + variants
  - iaLPSS2i_I2C.sys + variants
  - Dxgkrnl.sys (vGPU uses MMIO – Hyper-V)
- **Some candidates not looked at (Hyper-V):**
  - vmbusproxy.sys, Vmbususr.sys, Vpcivsp.sys (SR-IOV), vmbkmcl.sys, vmbus.sys, vmswitch.sys

Iacamera64.sys

- Intel AVStream Camera Driver
  - Used for SurfaceBook embedded camera
- Naïve String scanning returned an 'interesting' string:

```
   TraceRoutine(1, "The MMIO base address is 0x%08x.", *((unsigned int *) a1 + 12));
LABEL_7:
   _mm_lfence();
   pVAMapping = MmMapIoSpaceEx(a1[6], *((unsigned int *)a1 + 14), 4i64); // PAGE_READWRITE
   a1[8] = pVAMapping;
   if ( !pVAMapping )
     return 0xC000009Ai64;                           // STATUS_INSUFFICENT_RESOURCES
   _mm_lfence();
   CACHE_VA_REGION(pVAMapping);                      // Store Mapped MMIO Region VA in Global
   v17[0] = 0;
   v19 = 0i64;
   TraceRoutine_0(&v19, "IISPHWConfigManager::Dereference");
   if ( _InterlockedExchangeAdd((volatile signed __int32 *)a1 + 10, 0xFFFFFFFF) == 1 )
```

Iacamera64.sys

- **Plug and Play Device Management**
  - IspInterfaceNotification routine is registered to intercept PnP notifications for EventCategoryDeviceInterfaceChange events.
  - The PnP manager calls registered callbacks for GUID_DEVICE_INTERFACE_ARRIVAL or GUID_DEVICE_INTERFACE_REMOVAL events
  - CISPInterfacedConfigMgr::IaIspArrival (registered arrival callback) sets up an MMIO range for device interface, that is mapped to some cached globals for the duration of its use
  - MMIO unmapping routines registered for device removal
- **Wrapper routines implement SpinLocks and memory fences on R/W operations to MMIO regions**

lacamera64.sys

DO_MMIO_READ Routine

```
__int64 __fastcall DO_MMIO_READ(unsigned int mmioReadBase, unsigned int readLength)
{
  __int64 result; // rax
  KIRQL v5; // bl
  unsigned int v6; // edi

  if (  sub_14004A400(mmioReadBase, readLength)  )
    return sub_14004A4C0(mmioReadBase, readLength);
  v5 = KeGetCurrentIrql( );
  if (  v5 >= 2u  )
  {
    if (  v5 == 2  )
      KeAcquireSpinLockAtDpcLevel(&SpinLock);
  }
  else
  {
    _mm_lfence( );
    KfRaiseIrql(2u);
    while (  !KeTryToAcquireSpinLockAtDpcLevel(&SpinLock) )
    {
      KeLowerIrql(v5);
      KfRaiseIrql(2u);
    }
  }
  result = READ_MMIO_OFFSET(mmioReadBase, readLength);
  v6 = result;
  if ( v5 >= 2u )
  {
    if ( v5 == 2 )
    {
      KeReleaseSpinLockFromDpcLevel(&SpinLock);
      result = v6;
    }
  }
  else
  {
    _mm_lfence( );
    KeReleaseSpinLock(&SpinLock, v5);
    result = v6;
  }
  return result;
}
```

Iacamera64.sys

READ_MMIO_OFFSET,
READ_MMIO_RANGE_OFFSET
CacheMMIORegionBase
CACHE_VA_REGION
Routines

```
__int64 __fastcall READ_MMIO_OFFSET(unsigned int al, int a2)
{
  __int64 result; // rax

  if ( al < 3 )
    result = (unsigned int) GET_MMIO_READ_RANGE_OFFSET(a2 + dword_14010ED60[a1]);
  else
    result = 0xFFFFFFFFi64;
  return result;
}

__int64 __fsatcall GET_MMIO_READ_RANGE_OFFSET(unsighed int readIndex)
{
  __int64 result; // rax

    if ( g_MappedMMIORangeReadAddress && *(_QWORD *)g_MappedMMIORangeReadAddress )
      result = *(unsigned int *)(*(_QWORD *)g_MappedMMIORangeReadAddress + readIndex);
    else
      result = 0xFFFFFFFFi64;
    return result;
}

__int64 *__fastcall CacheMMIORegionBase(__int64 a1)
{
  __int64 *result; // rax

    result = &g_MappedMIORegionbase;
    g_MappedMIOIRegionbase = a1;
    g_MappedMMIORangeReadAddress = (__int64)&g_MappedMIORegionbase;
    return result;
}

__int64 *__fastcall CACHE_VA_REGION(__int64 a1)
{
  KeInitializeSpinLockThunk();
  return CacheMMIORegionBase(a1);
}
```

Iacamera64.sys

CISPInterfacedConfigMgr::IaIspArrival Routine

```
pVAMapping = MmMapIoSpaceEx(*((_QWORD *)a1 + 6), a1[14], 4i64); // PAGE_READWRITE
V20 = a1[14];
V21 = a1[12];
*((_QWORD *)a1 + 8) = pVAMapping;
if ( pVAMapping )
{
   LODWORD(StartingOffset) = v20;
   TraceRoutine(
     1,
     "%s: (m_Base %#x, m_Length %#x) map to %#x",
     "CISPInterfacedConfigMgr: :IaIspArrival",
     v21,
     StartingOffset,
     pVAMapping);
   CACHE_VA_REGION(*((_QWORD *)a1 + 8));        // StoreMMIO Region in Global
```

Iacamera64.sys

- Pivoting on these DO_MMIO_READ wrappers
  - 261 locations perform Writes,
  - 184 locations perform Reads
- A lot of these related to perf. counters, but not all

```
TraceRoutine(1, "****** CDrivferControl : : AuthenticateFW");
v5 = (unsigned int)DO_MMIO_READ(2u, 0x300u);
TraceRoutine(1, "****** SECURITY_CTL register value before authentication: %x", v5);
If ( (v5 & 0x1F) == 0 )
{
   v1 = sub_140022EC0(
        a1,
        *(_QWORD *)(a1 + 1760),
        *(_DWORD **)(a1 + 1768),
        *(_DWORD *)(a1 + 1776),
        *(_DWORD *)(a1 + 1760),
   v5 = (unsigned int)DO_MMIO_READ(2u, 0x300u);
   TraceRoutine(1, "****** SECURITY_CTL register value after authentication: %x", v5);
   if ( v1 < 0 && (_DWORD)qword_140110560 == 2 && !dword_140110568 )
```

iaLPSS2i_I2C.sys

- **Intel Low Power Subsystem Support Integrated Circuit Driver**
  - Responsible for registration of devices onto a PCI bus
  - Holds a linked list of Device Driver object entries, each with various registration handlers hooked up
  - Presumably, this is to allow dispatch of power events to devices registered on a bus by device Index
  - We suspect this driver employs MMIO to support registration of an ACPI-compliant device with the PCI bus
  - Several variants of this driver exist, that look very similar in terms of offered functionality
    - iaLPSS2i_I2C_BXT_P.sys, iaLPSS2i_I2C_CNL.sys, iaLPSS2i_I2C_GLK.sys, iaLPSSi_I2C.sys

iaLPSS2i_I2C.sys

OnDeviceAdd

```
Dst[5] = (__int64)OnPrepareHardware;
Dst[6] = (__int64)OnReleaseHardware;
Dst[1] = (__int64)OnD0Entry;
Dst[3] = (__int64)OnD0Exit;
Dst[9] = (__int64)OnSelfManagedIoInit;
Dst[7] = (__int64)OnSelfManagedIoCleanup;
Dst[14] = (__int64)OnQueryStop;

Dst[3] = (__int64)OnInterruptIsr;
Dst[4] = (__int64)OnInterruptDpc;
Dst[1] = v20;
```

iaLPSS2i_I2C.sys

OnPrepareHardware

```
pVAMapping = MmMapIoSpaceEx(*(_QWORD *)(v15 + 4), *(unsigned int *)(v15 + 12), 0x204i64);
If ( !pVAMapping )
{
   status = 0xC000009A;            // STATUS_INSUFFICENT_RESOURCES
   if ( ((__int64)WPP_MAIN_CB.Queue.Wcb.DeviceObject & 1) != 0 )
     ETWLogThunk(
        &iaLPSS2_I2C_PROVIDER_Context,
        &EvtPrepareHardware_MmioMap_Error,
        v9,
        0,
        *(_QWORD *)(v15 + 4),
        *(_DWORD *)(v15 + 12),
        154);
     goto LABEL_53;
}
*(_QWORD *)(v8 + 32) = pVAMapping;
*(_QWORD *)(v8 + 24) = *(_QWORD *)(v15 + 4);
*(_DWORD *)(v8 + 40) = *(_DWORD *)(v15 + 12);
*(_QWORD *)(v8 + 56) = pVAMapping + 512;
*(_QWORD *)(v8 + 272) = pVAMapping + 2048;
*(_QWORD *)(v8 + 48) = pVAMapping;
If ( ((__int64)WPP_MAIN_CB.Queue.Wcb.DeviceObject & 2) != 0 )
   ETWLogThunk2(
     (__int64)&iaLPSS2_I2C_PROVIDER_Context,
     (__int64)&EvtPrepareHardware_MmioMapped_Info,
     (__int64)v9,
     0,
     *(_QWORD *)(v15 + 4),
     *(_DWORD *)(v15 + 12),
     pVAMapping);
```

iaLPSS2i_I2C.sys

- **We suspect MMIO is used here to support registration of ACPI-compliant devices with the PCI bus**
- **ACPI supports thermal event triggers**
  - Managed and issued by ACPI subsystem to support device temperature cut-off
  - Managed using System Control Interrupts (SCI)
- **Could we cause an interpolated MMIO read during ACPI SCI Interrupt handling?**
  - If so, we could consider the ACPI code, and its interpreter state a valid attack surface
  - AML is a Turing-complete language, running in a VM, in ring-0
  - It would appear the same observations we apply to device drivers could apply here too
- **This would make any ACPI-compliant device with dynamic registration handlers a target**

MMIO Double-Fetch

- **Where a device performs:**
  - MmMapIoSpace / ExAllocatePool / RtlCopyMemory / MmUnmapIoSpace
  - Some action relying on the state
  - MmMapIoSpace / ExAllocatePool / RtlCopyMemory / MmUnmapIoSpace
- **There may be opportunities to exploit an MMIO double-fetch**

- **We saw this pattern in drivers examined**
  - Often the mapping operations will be wrapped to include memory barriers and caching (maybe also logging), so pattern becomes:
  - ReadMMIO() / Some action / ReadMMIO()
- **Hypervisor attack primitive?**
  - Virtualized devices?
  - Hyper-V?

MMIO Double-Fetch

At least one person doesn't think we're crazy…

meysam
@R00tkitSMM

Seriously this is a new attack scenario, we always used device MMIO/PortIO to attack virtual machine's device implementations, but this time we attack kernel from real device, any type of wrong assumptions in MMIO processing.

> **meysam** @R00tkitSMM · Mar 22
> So if a device driver do double fetch like:
> p = MmMapIoSpace()
> ExAllocatePool(..,*p); RtlCopyMemory(...,*p);
> MmUnmapIoSpace()
> then theoretically we can have a device to exploit it :)))
> twitter.com/JosephBialek/s...

MMIO Operation Race Conditions

- **Ordering of operations can be extremely important**
  - Uncached (UC) and Write Combining (WC) are most common types of MMIO

- **Any driver sensitive to the order of MMIO operations not using memory fences, barriers and cache flushing may be a target**
  - This complexity adds to the potential for bugs

Fuzzing MMIO Ranges

Fuzzing MMIO addresses - Idea

- **Intercept MmMapIoSpaceEx function to obtain mapped virtual address and size.**
- **Obtain the MMIO address ranges that are mapped long after boot.**
- **Create multiple threads**
  - Read from these addresses, within the size range of each mapping
  - Essentially simulate a “blind read” across all the MMIO ranges.

Fuzzing MMIO addresses - Problems

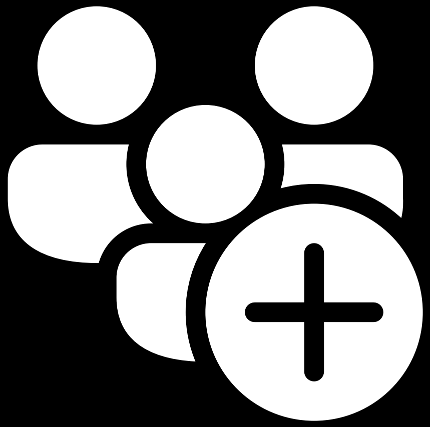
- **IO drivers constantly map/unmap/remap the MMIO ranges.**
  - Example for reading a status register on the device on the fly.
  - Accessing an unmapped address will bugcheck; restart fuzzing setup
  - Solution: Track and update the latest mapped regions by intercepting MMapIoSpaceEx and MmUnmapIoSpace.
  - Observing "weird" behavior
- What are we looking for?
  - Crashes?
  - Freezes?
  - Something else....?
- **Not all exploitable behaviors will manifest as an observable crash by fuzzing.**
- **Behavior is very specific to each IO device and the associated driver**

Conclusions

Parting Thoughts

- **Programming low-level device interactions is complicated and fraught with complexities**
  - Where there is complexity, there is usually bugs
- **Many classes of device exist out there, we only have a small subset**
  - barely scratches the surface
- **Not all exploit primitives are created equal**
  - MMIO exploit is unlikely to be portable (device specific)
  - Will likely require another bug to exploit (e.g. VA leak)
  - Not a lot of low-hanging-fruit
    - Perhaps a better avenue to attack the Hypervisor than ring-0 devices
- **Theoretically it looks possible**
  - What could be cooler than RCE from a pointer read?

Call to Arms



We’d love for the external research community to build on this idea.

Tell us if you find some interesting devices and behaviors that can facilitate exploitation!

Questions?

- Andrew Ruddick @arudd1ck
- Rohit Mothe @rohitwas

Microsoft

Thank you