Exploring a New Class of Kernel Exploit Primitive

Security Research & Defense / By Andrew Ruddick / March 22, 2022 / Security Research

The security landscape is dynamic, changing often and as a result, attack surfaces evolve. MSRC receives a wide variety of cases spanning different products, bug types and exploit primitives. One particularly interesting primitive we see is an arbitrary kernel pointer read. These often happen when kernel mode code does not validate that pointers read from attacker-controlled input actually point to the user-mode portion of the Virtual Address Space (VAS). An attacker can utilize such a primitive to supply a kernel mode-pointer which will then be de-referenced by kernel-mode code. Sometimes, there is a further restriction that even though the attacker can cause a read of an arbitrary kernel address / pointer they cannot retrieve the contents of the memory read. If the virtual memory at a given address X contains data Y, then we assume the attacker can cause the kernel to read X but does not have a direct way of obtaining the read data, Y. If the attacker could read Y then they can read memory contents from the VAS and we would generally assess this as an information disclosure vulnerability. However, it isn't always clear how to assess cases where the primitive an attacker has is to cause an arbitrary kernel pointer read but cannot leak the data. Traditionally, these would have an impact of Denial of Service (DoS) or in some cases a second-order Kernel Memory Information Disclosure (where side channels or indirect probing are possible) but we wonder if such a limited primitive could actually be exploited for code execution / privilege escalation?

The idea we wanted to explore when pondering the above question was; can we exploit reads to Memory Mapped I/O (MMIO) ranges of peripheral device drivers? Reads to MMIO ranges are used for two-way communication between the device driver and the IO device and it wouldn't be a stretch to imagine that they would be sensitive to the order, timing and even the size of memory reads issued to their respective MMIO space. We can view the external hardware device logic as an abstract state machine which is affected by the various read and write operations to these MMIO ranges. For example, consider an IO device that implements an “atomic read-and-reset” of a state / last-error register or a serial input device that removes characters from an internal buffer while one reads from it. This could, in theory, cause unexpected behaviors as the IO device would normally expect only a legitimate entity, either the OS kernel or the device driver, to issue the memory reads to its MMIO space. But what if an attacker, using an arbitrary pointer read bug in the kernel, can subvert these legitimate reads and issue them out-of-order and at unexpected times, corrupting the state machine logic for the IO device. Could an attacker exploit this?

In this blog, we will detail the exploratory research we've conducted into this attack vector and walk you through how you as a researcher, can lookout for interesting behaviors with exploit primitives like this.

Locating Target Devices

The first step is to collate a list of candidate system devices for further vulnerability research. Any driver that makes use of MMIO is of interest, but the more complex the ring-0 state-machine logic / decisions made on data originating from those regions, the more likely it is exploitable. To this end, we will discuss the following device and MMIO address range enumeration strategies:

1. Registry / Device Manager Probing
2. ACPI MCFG Table
3. Naïve string scanning
4. MmMapIoSpace Interception

Registry / Device Manager Probing

We can look at device manager to find a list of interesting devices on a system and probe for their associated MMIO ranges. We can also script the extraction of this information using PowerShell:

```
$dnames = Get-WmiObject Win32_PnPEntity -Filter 'DeviceID like "PCI%"'
 $idtable =  @{}
 $rangetable = @{}
 $rangetable2 = @{}


 For ($i=0; $i -lt $dnames.Length; $i++) {
    $name = $dnames[$i].Name;
        $devid = $dnames[$i].DeviceID;
        $idtable[$name] = $devid
        $cis = Get-CimInstance Win32_PNPAllocatedResource | where-
object { $_.Dependent.DeviceID -like "*$devid"} | where Antecedent -
like "Win32_DeviceMemoryAddress*"
        $cis2 = $cis.Antecedent.StartingAddress
        $rangetable[$name] = $cis2
        #$ranges = Get-CimInstance Win32_DeviceMemoryAddress | where
StartingAddress -eq $cis.Antecedent.StartingAddress;
    #$i, $name, $devid, $cis.Antecedent.StartingAddress
    }

foreach ($h in $rangetable.Keys) {
        $qarr = @()
        if ($rangetable[$h] -is [system.array])
        {
                For ($t=0; $t -lt $rangetable[$h].Length; $t++) {
                        $qarr += , ((Get-CimInstance
Win32_DeviceMemoryAddress | where StartingAddress -eq $rangetable[$h]
[$t]).Name);
                }
        }
        else {
                $qarr += , ((Get-CimInstance Win32_DeviceMemoryAddress
| where StartingAddress -eq $rangetable[$h]).Name)
        }
        $rangetable2[$h] = $qarr
}

$endtable = @()
foreach ($h in $idtable.Keys) {
        $obj = [PSCustomObject]@{
                Name = $h
                DeviceID = $idtable[$h]
                Ranges = $rangetable2[$h]
        }
        $endtable += , $obj
}

Write-Output $endtable
```

An example of the MMIO ranges you can see look like these:

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}

The first thing that should be apparent, is that all of these are physical addresses. In the scenario we consider, the attacker can only cause reads of arbitrary Virtual Addresses (VA). For research purposes, we can use a Kernel Debugger (KD), to check if these are mapped to a corresponding VA, but for any practical attack, a given physical MMIO address range would need to be mapped to the kernel VAS, therefore an exploit would also require an additional primitive to bypass Kernel Address Space Layout Randomization (KASLR).

ACPI MCFG Table

The Advanced Configuration and Power Interface (ACPI) is another source we can examine to find interesting devices. ACPI controls at the lowest level, interactions with system hardware devices over the primary and (optionally) the peripheral busses, forming an abstraction layer between hardware device firmware and the Operating System (OS). ACPI brings control of some low-level firmware management operations to the OS, reducing reliance on x86 / x86-64 System Management Mode (SMM) to handle such operations.

ACPI information is organized into a number of tables, which are stored in the registry and can be dumped using open-source tools from the ACPI Component Architecture Project, ACPICA. A number of ACPI tables are required, as defined in the ACPI specification, however, a number of optional, reserved tables also exist. The 'MCFG' table is an optional table which contains Peripheral Component Interconnect (PCI) device configuration information, including registered MMIO ranges and PCI Base Address Registers (BARs). The MCFG table is described by the specification as 'PCI Express memory mapped configuration space base address Description Table'. Details of the PCI configuration entries are detailed in the PCI Firmware Specification, Revision 3.0, Table 4-2. We can dump the registered ACPI tables at runtime using the ACPICA tools and the following commands:

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

Intel ACPI Component Architecture
ACPI Binary Table Extraction Utility version 20210730
Copyright (c) 2000 - 2021 Intel Corporation


 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
```

The MCFG table can be extracted and decompiled using the same tools:

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

Intel ACPI Component Architecture
ACPI Binary Table Extraction Utility version 20210730
Copyright (c) 2000 - 2021 Intel Corporation

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

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

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20210730
Copyright (c) 2000 - 2021 Intel Corporation

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
```

The decompiled result, is ACPI Source Language (ASL / DSL) and provides us the base address (physical address) that contains the MCFG raw table data:

```
/*
* Intel ACPI Component Architecture
* AML/ASL+ Disassembler version 20210730 (32-bit version)
* Copyright (c) 2000 - 2021 Intel Corporation
*
 * Disassembly of mcfg.dat, Wed Sep 22 22:08:07 2021
*
* ACPI Data Table [MCFG]
*
* Format: [HexOffset DecimalOffset ByteLength]  FieldName : FieldValue
(in hex)
*/

[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

Raw Table Data: Length 60 (0x3C)

    0000: 4D 43 46 47 3C 00 00 00 01 84 41 4C 41 53 4B 41
       // MCFG<.....ALASKA
    0010: 41 20 4D 20 49 00 00 00 09 20 07 01 4D 53 46 54
       // A M I.... ..MSFT
    0020: 13 00 01 00 00 00 00 00 00 00 00 00 00 00 00 E0
       // ................
    0030: 00 00 00 00 00 00 00 FF 00 00 00 00
       // ............
```

Given this MCFG physical base address, we can parse the physical PCI bus, device and function numbers from this location for each PCI device. The RWEverything tool contains PCI parsing utility functions to display this information to us simply (though this needs to be ran on a machine that does not have Virtualization Based Security (VBS) enabled). An example of a PCI device, with registered MMIO ranges dumped using this method:

```
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
Class Code      0x060400
Cacheline Size  0x10
Latency Timer   0x00
Interrupt Pin   INTA
Interrupt Line  IRQ11
BAR1            0x00000000
BAR2            0x00000000
Primary Bus#    0x00
Secondary Bus#  0x01
Subordinate Bus#        0x01
IO Range
   0x0000E000 - 0x0000EFFF
Memory Range
   0xFEA00000 - 0xFEAFFFFF
Prefetchable Memory Range
```

```
   0xC0000000 - 0xD07FFFFF
Expansion ROM   0x00000000
Subsystem ID    0x5A141002
```

It is worth mentioning that during our analysis, none of the Hyper-V VM configurations on our test machines implemented this optional MCFG table, therefore this method is not viable in all situations. Equally, the requirement of reading physical memory can prevent dumping of the MCFG configs from devices with the table registered where VBS is enabled.

Naïve String Scanning

Owing to the VBS / HVCI issues with the above noted approaches, we could also use a simple PowerShell script to scan all driver images stored in C:\Windows\System32\Drivers that contain a string 'MMIO'. The located strings can be dumped, along with the driver image names. Some PowerShell:

```
$ascii_grep = Get-ChildItem -Recurse | Select-String "MMIO" -List -
Encoding "ASCII" | Select Path
$mbs_grep = Get-ChildItem -Recurse | Select-String "MMIO" -List -
Encoding "Unicode" | Select Path

$merged_list = & {
    $ascii_grep
    $mbs_grep
}

echo $merged_list
```

This can be executed in the directory you wish to scan:

```
PS C:\Windows\System32\drivers> C:\grep_drivers.ps1

Path
----
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
```

Running the SysInternals 'strings' utility over these will give specific details as to the contents of each identified driver. Using this method, we identified a total of 24 driver files on our lab machines that were selected for further analysis. Largely, this list consists of components related to GPIO, I2C, DirectX / Video, Virtualization (Hyper-V), USB and OEM device-specific hardware.

MmMapIoSpace Interception

The MmMapIoSpace / MmMapIoSpaceEx kernel driver routines are used to map a given physical memory address range to the nonpaged system address space. We can intercept calls to these to gather a list of all the VA that are mapped and unmapped (Note: MmMapIoSpace maps writeable and executable memory (if HVCI is turned off) whilst MmMapIoSpaceEx lets you specify page protections). Both of these functions are exported from ntoskrnl.exe, so are easy to locate with public debug symbols.

To intercept these calls interactively, we can instrument the functions in WinDbg and read the returned values. In a normal flow, the device drivers would have to map the physical address range of the device to a VA to perform any operations, thus we can dump a list of all the VA that device drivers map IO space registers into. To dump all mappings made, the following commands can be used:

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

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

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

Finally, if we locate the return address of the same MmMapIoSpace/Ex functions, we can insert a break to also dump the mapped VA for each entry. This can be done as follows (you may need to slightly adjust the offset for your build):

```
bu nt!MmMapIoSpace+0x34 ".block{ r $t1 = @rax; .printf /D \"[+] Mapped
at Virtual Address: %p\n\", @$t1}; gc"

bu nt!MmMapIoSpaceEx+0x2b ".block{ r $t1 = @rax; .printf /D \"[+]
Mapped at Virtual Address: %p\n\", @$t1}; gc"
```

We observed a large number of hits in the output and realized the common pattern among these was device drivers mapping a physical address of the IO device in the VA and then immediately unmapping it after performing some operation. A large majority of the output we got from hooking these was in chunks like this:

```
[+] 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
```

To exploit an entry such as these, an attacker would have to race this small window. A lot of calls we noted were early on in the boot process, before the system was fully loaded or are issued on events such as a user log on and log off. It may be possible to mount an attack upon such an entry, but winning this race may bring it's own challenges. Our observations here will differ based on the devices present and as we will cover later, certain classes of device may elect to dynamically map MMIO regions based on user-triggerable actions. On our test system, we also noted approximately 15 virtual addresses that IO device drivers mapped that still persisted long after boot.

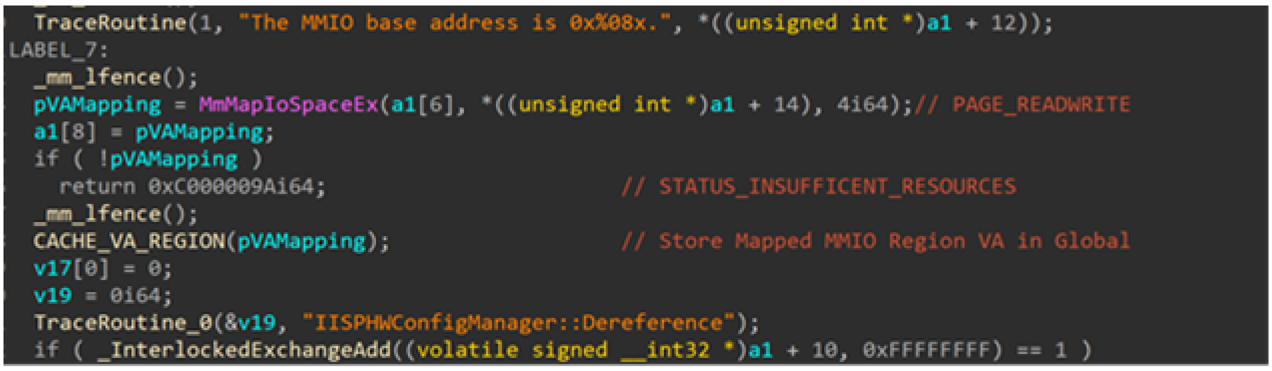
Interesting Device Driver Patterns

Once some target device drivers have been identified, it's time to crack out your favorite decompiler. We started with some driver images located with the naïve string scanning scripts as this had the added benefit of including further context about the code that we are examining. Debug trace and logging strings left in such drivers often give a clue as to their intended purpose, before any detailed analysis needs to be conducted. This allowed us to target our efforts more effectively.

Let's examine a couple of interesting drivers that we spotted and discuss the theoretical attack surface.

Intel AVStream Camera Driver (Iacamera64.sys)

We selected the iacamera64.sys driver early in the process, owing to the interesting string we found: 'The MMIO base address is 0x%08x.'. This string is found in a driver trace routine call:

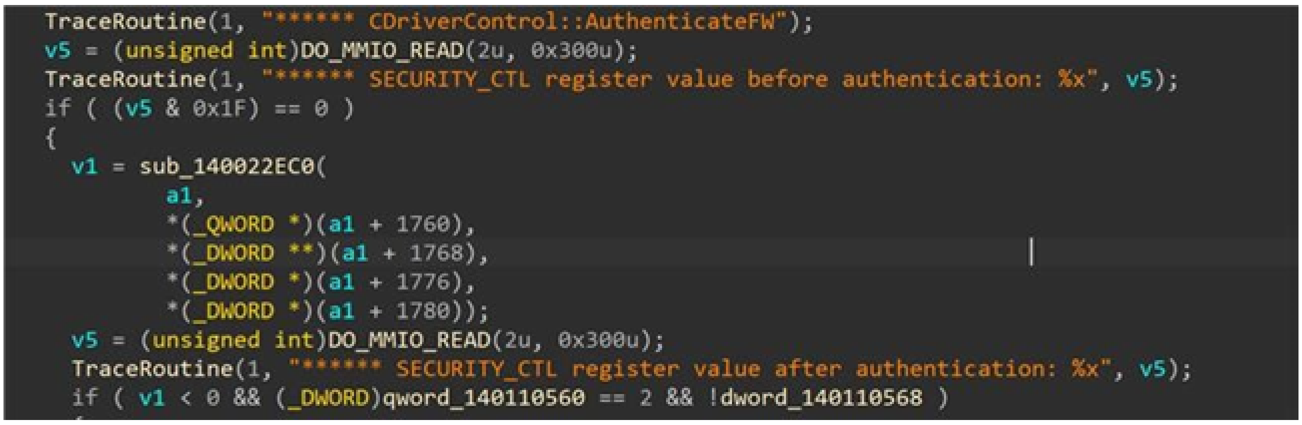


```
  TraceRoutine(1, "The MMIO base address is 0x%08x.", *((unsigned int *)a1 + 12));
LABEL_7:
  _mm_lfence();
  pVAMapping = MmMapIoSpaceEx(a1[6], *((unsigned int *)a1 + 14), 4i64);// PAGE_READWRITE
  a1[8] = pVAMapping;
  if ( !pVAMapping )
    return 0xC000009Ai64;                         // STATUS_INSUFFICENT_RESOURCES
  _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 )
```

Through further analysis of this driver, its purpose appeared to be to support the embedded camera in the laptop we used for testing. A couple of noted locations within this driver use MMIO. Although there was no public symbol information for the binary, helpfully a number of the trace routines leak the function names, so the following analysis can be more specific, without need to resort to binary offsets.

The IspInterfaceNotification routine is registered to intercept PnP notifications for the EventCategoryDeviceInterfaceChange. This event handler is invoked by the PnP manager for future arrivals or removals of device interface instances. The registered PnP operation invokes CISPInterfacedConfigMgr::IaIspArrival on arrival of such a PnP event. This routine sets up an MMIO range for device interface, which it appears to map to some cached globals for the duration of its use (MMIO un-mapping routines are also registered). Wrapper routines that implement spin locks and memory fences on R/W operations wrap accesses to these MMIO regions. Both MMIO Read and write wrappers are present on this cached region.

Examining sub-routine cross-references suggests that there were at least 261 locations in the version of the driver we analyzed that perform MMIO Write operations and another 184 that perform Read operations. A lot of the locations noted were related to performance counter information and device configuration, but in at least a few locations, the device driver makes decisions based upon the result of an MMIO read, including for Authentication of the device FW image:



```
  TraceRoutine(1, "****** CDriverControl::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 + 1780));
    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 )
```

Device drivers such as this, that make numerous decisions based upon the state of data read from the MMIO region would be a good target for attack.

Each MMIO Read operation is synchronized with mem fences / spin locks, but we didn't see any evidence that this code synchronizes in-between the reads themselves. Clearly, this assumption may well be sound as it entirely depends on what the device itself is doing in response to such an operation, however interleaving of operations with interpolated reads certainly could lead to bugs in the state expected by the device driver.

A Note on Memory Barriers and Caching

Where cache write-buffers are employed, the order of two sequential writes to different device addresses may not be observed. This means that MMIO data writes are not guaranteed to reach a peripherals' memory banks or registers in the same order they were issued. Therefore, any driver that is sensitive to the order of operations and does not manually include memory barrier / cache-flushing instructions after sensitive writes could also be a target of interest. Optimization of writes can be made to reduce device redundancy or for memory access latency hiding, without changing the final store state. Usage of a write buffer therefore frees the cache to interleave read requests whilst a write is taking place. This additional complexity adds the potential for race conditions in the order MMIO operations are issued.

Intel Serial IO I2C Driver v2 (iaLPSS2i_I2C.sys)

The Intel Low Power Subsystem Support Integrated Circuit Driver is an example of an entirely different class of device driver that makes use of dynamic MMIO device registration. It appears that this particular device is responsible for the registration of devices onto a PCI bus, containing a linked list of Device Driver object entries that have various event registration handlers hooked up (presumably so power events can be dispatched to devices registered on a specified bus by bus device index). A number of variations of this driver were noted to exist, that do not differ much in terms of the functionality noted here (perhaps different controller drivers exist for different bus types?).

When a device is added, the OnDeviceAdd routine is invoked, this registers a set of fixed functions (some of which look to be later overwritten after function dispatch, though this is not relevant to us here):

```
memset(Dst, 0, sizeof(Dst));
LODWORD(Dst[0]) = 144;
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;
```

This same OnDeviceAdd routine registers both Interrupt Service Routine (ISR) and Deferred Procedure Call (DPC) handlers for the device:

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

The OnPrepareHardware function registers an MMIO range, that is unmapped by the paired OnReleaseHardware routine later:

```
    pVAMapping = MmMapIoSpaceEx(*(_QWORD *)(v15 + 4), *(unsigned int *)
  (v15 + 12), 516i64);
    if ( !pVAMapping )
    {
      v10 = 0xC000009A; // ERROR_INSUFFICENT_RESOURCES
      if ( ((__int64)WPP_MAIN_CB.Queue.Wcb.DeviceObject & 1) != 0 )
        ETWLogThunk(
          (unsigned int)&iaLPSS2_I2C_PROVIDER_Context,
          (unsigned int)&EvtPrepareHardware_MmioMap_Error,
          (_DWORD)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(
        (unsigned int)&iaLPSS2_I2C_PROVIDER_Context,
        (unsigned int)&EvtPrepareHardware_MmioMapped_Info,
        (_DWORD)v9,
        0,
        *(_QWORD *)(v15 + 4),
        *(_DWORD *)(v15 + 12),
        pVAMapping);
```

It is our suspicion that MMIO is employed in this case, to support the registration of an ACPI-compliant device with the PCI bus, so that where (for example), a device has a temperature cut-off sensor, it may be registered by the ACPI subsystem (ACPI.sys) to handle the dispatch of power management events to device firmware handlers. This explanation would make sense given the above noted states D0 / D3 relate to device power states.

It should suffice for our discussion that the ACPI subsystem controls operations including system power, hibernation, sleep states, IO probing and PnP. The subsystem runs in ring 0 and is managed by the ACPI.sys driver. This driver contains an ACPI Machine Language (AML) Interpreter (minimal VM), which is capable of issuing System Control Interrupts (SCI) on events such as a thermal event trigger being exceeded. In such a situation, the run-time can execute a registered ACPI AML control method corresponding to the interrupt. This code is able to dynamically issue device IO. Based on the code here, presumably, this uses the mentioned MMIO region as a target to handle (at least particular classes of) these device IO requests. Discussion of these concepts is beyond the scope of this blog, but information on the ACPI driver power event registration system can be found here, the core ACPI tables are summarized here and the Differentiated System Description Table (DSDT) is defined in section 5.2.11 of the ACPI specification.

The ACPI DSDT contains the entire ACPI implementation, for a given machine, defining objects for all devices rooted on the system bus. An example of some decompiled ACPI Machine Language (AML) code from the DSDT (using the same ACPICA tools noted earlier):

```
Device (S8F0)
{
    Name (_ADR, 0x00070000)  // _ADR: Address
    Name (_SUN, 0x47)  // _SUN: Slot User Number
    OperationRegion (REGS, PCI_Config, 0x00, 0x04)
    Field (REGS, DWordAcc, NoLock, Preserve)
    {
        ID,     32
    }

    Method (_STA, 0, NotSerialized)  // _STA: Status
    {
        Return (BSTA (ID))
    }

    Method (_EJ0, 1, NotSerialized)  // _EJx: Eject Device
    {
        BEJ0 (Arg0, _ADR)
    }

    Method (_DSM, 4, Serialized)  // _DSM: Device-Specific Method
    {
        Return (BDSM (Arg0, Arg1, Arg2, Arg3, _ADR))
    }

    Name (_PRW, Package (0x02)  // _PRW: Power Resources for Wake
    {
        0x03,
        0x03
    })
}
```

This ACPI Source Language (ASL) is just an example of a routine for a random device that has a registered device PCI Configuration Register OperationRegion, but should suffice to show that there is dynamic interaction with the device directly here, on the processing of power management event state handling, in a Virtual Machine, operating in ring 0. An interesting question is, can we now consider registered ACPI code and its interpreter a valid attack surface? In the event that an ACPI SCI Interrupt is triggered, any interpolated MMIO read could cause unintended consequences in the interpreter state, which ultimately is executing a Turing-complete programming language and thus would appear to be vulnerable to the same observations we note in device driver objects.

If our assertions are sound, then any ACPI-compliant device, with dynamic registration handlers may now also be interesting. There is also a potential benefit to the stability of an exploit if it were based upon more generic interactions between a standards-compliant component, such as ACPI and a peripheral device.

Fuzzing MMIO Address Ranges

Using the above noted MmMapIoSpace intercept, we decided to try fuzzing the ~15 MMIO address ranges that we obtained on our test device, and which remained mapped long after boot. To do this, we created multiple threads to continuously read from these addresses, within the size range of each mapping to see if we could trigger any interesting behavior. Some examples of the targeted MMIO addresses are listed below (note: the addresses listed are not fixed and change every boot/every time MmMapIoSpaceEx is called).

```
0xffff67c84600000; // 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
<…>
```

On our test devices we did not observe any noteworthy crashes or behaviors from this rudimentary black-box fuzzing exercise. The few crashes we encountered were the device drivers finally un-mapping or re-mapping the VAS after a while. A better approach would be to target specific MMIO regions, such as those noted during reverse engineering efforts and target those specific instances at particular points of transition within the driver state machine. Adding dynamic breakpoints and scripting dynamically issued reads through the debugger may be enough to trigger this. It is also likely that there could be issues with drivers that won't manifest as an observable crash in the device driver. Maybe the device hardware itself does something interesting that is harder to observe as a consequence of our fuzzing. This again brings the problem of measuring positive results and would require extensive knowledge of the specific IO device. We encourage interested readers to try these techniques on their own setup and peripheral devices. Let us know if you find anything cool!

Parting Thoughts

Programming low-level device interactions are complicated and fraught with complexities. Where there is complexity, there are usually bugs. Hopefully, our explorations will encourage further research into exploiting arbitrary pointer reads via MMIO ranges for something more profitable like RCE or EoP. Afterall, there are a wide variety of IO devices out there. Any exploit is however unlikely to be portable and will be restricted to interactions with a specific piece of hardware. Reachability of such bugs are therefore limited (and their blast-radius reduced). In scenarios where the hardware eco-system is less fragmented than Windows (such as some classes of device in the mobile handset space), this could be less of a concern for a motivated attacker.

From an exploitation research point of view, we believe there is still significant benefit in expanding the state-of-the-art on kernel read exploitation. We barely scratched the surface in this article, and believe more research needs to be done to further understand the practical viability and impact of such attacks. Today, there is much lower-hanging fruit for an attacker to pursue. Finding another more easily exploitable primitive in a 'standard' ring-0 component is also significantly more likely to be portable. Thus, expending effort on weaponizing a bug in the MMIO space is perhaps less attractive. If an attacker were to find a lot of MMIO crashes in varying devices with a custom fuzzer, such efforts may become more palatable. Similarly, if a certain class of IO devices is discovered to have specific behavior (example: unique state machine logic) that is susceptible to exploitation using this primitive, then that would also lower the barrier for entry and provide more value to attackers.

Not all exploitation primitives are created equal. A generic, MMIO blind-read primitive is, based on current knowledge, less exploitable than an Information Disclosure or a blind kernel-write vulnerability, will require another bug for an attacker to exploit it, and a very specific hardware configuration on the target. There is however a theoretical possibility and what could be more cool than RCE from a pointer read? There's a ton of hardware out there, and we only tested a few devices which barely scratches the surface. We would love to hear from you if you find interesting devices and behaviours that can facilitate exploitation using this primitive!

Andrew Ruddick, Rohit Mothe, Carolina Hatanpaa – Microsoft Security Response Center (MSRC)

Acknowledgements

Special thanks to Saar Amar, Joe Bialek, Nicolas Joly, Saruhan Karademir, Nick Stephens, Adrian Tolbaru, Bruce Sherwin, David Zhang and David Hepkin.

Search ...

Categories

BlueHat (181)

Japan Security Team (947)

MSRC (983)

Security Research & Defense (372)

Tags

advisory (60) ANS (47) Attack (43) Attack Vector (68) Black Hat (33) BlueHat Security Briefings (55) Community-based Defense (93) Defense-in-depth (38) EcoStrat (34) EMET (68) Exploitability (77) Internet Explorer (IE) (156) malware (59) Microsoft Office (81) Microsoft Windows (106) Mitigations (127) monthly bulletin release (48) rating (48) Risk Assessment (104) security (80) Security Advisory (134) Security Bulletin (133) security bulletin release (44) Security Bulletins (39) Security Conference Engagement (56) Security Ecosystem (52) Security Engineering (42) Security Research (79) Security Update (139) Security Update Webcast (46) Security Update Webcast Q & A (70) Update Tuesday (63) Webcast (37) Windows Update (68) Workarounds (74) Zero-Day Exploit (36) アドバイザリ (151) セキュリティ (54) セキュリティ情報 (458) セキュリティ更新 (83) ワンポイント (39) 啓発 (45) 展開 (45) 時事ネタ (42) 脆弱性 (246)

Recent Posts

Microsoft's Response to CVE-2022-22965 Spring Framework

Randomizing the KUSER_SHARED_DATA Structure on Windows

On-Premises Servers Products are Here! Introducing the Applications and On-Premises Servers Bug Bounty Program

Increasing Representation of Women in Security Research

Randomizing the KUSER_SHARED_DATA Structure on Windows

Archives

Select Month