BARing the System
New vulnerabilities in Coreboot & UEFI based systems

Presenting: Yuriy Bulygin (@c7zero), Oleksandr Bazhaniuk (@ABazhaniuk)
Andrew Furtak, John Loucaides, Mikhail Gorobets
Agenda

• Recap of SMM Pointer Vulnerabilities
• Intro to Memory-Mapped I/O
• MMIO BAR Issues
• MMIO BAR Issues in UEFI Firmware
• MMIO BAR Issues in Coreboot Firmware
• Limitations
• Mitigations
• Tools
• Conclusion
Recap of SMM pointer vulnerabilities
Pointer Arguments to SMI Handlers

Via ACPI table

“UEFI” ACPI

Comm Buffer

EDKII

Directly in registers

RAX (code)

RBX (pointer)

EDKI

Phys Memory

SMI Handlers in SMRAM

OS Memory

SMM communication buffer
Arguments IN, results OUT

EDKII

Phys Memory

SMI

“UEFI” ACPI

Comm Buffer

EDKII

Directly in registers

RAX (code)

RBX (pointer)

EDKI

SMM communication buffer
Arguments IN, results OUT
Exploiting SMM pointers…

Exploit tricks SMI handler to write to an address in SMRAM (Attacking and Defending BIOS in 2015)
Attacking hypervisors via SMM pointers…

Even though SMI handler check pointers for overlap with SMRAM, exploit can trick it to write to VMM protected page (Attacking Hypervisors via Firmware and Hardware)
Example: SMIFlash SMI Handler

Reported by ATR to BIOS vendor in June 2014

Similar to publication by Sogeti ESEC Lab

```c
VOID SMIFlashSMIHandler (
   ...
   SwSmi = (UINT8)DispatchContext->SwSmiInputValue;
   CpuState = pSmst->CpuSaveState;
   AddrHi = CpuState[Cpu].Ia32SaveState.ECX;
   AddrLo = CpuState[Cpu].Ia32SaveState.EBX;
   Buff = AddrHi;
   Buff = Shl64(Buff, 32);
   Buff += AddrLo;
   ...
   switch (SwSmi) {
   ...
   case 0x21:
   ReadFlashData( (FUNC_BLOCK *)Buff );
   ...
   case 0x25:
   ReadFlashInfo( (INFO_BLOCK *)Buff );
   }
   ...
   EFI_STATUS ReadFlashData(IN OUT FUNC BLOCK *func)
   ...
   sts = Flash->Read(
      (UINT8*)(FlashStart + func->BlockAddr),
      func->BlockSize,
      (UINT8*)func->BufAddr
   );
}
```
SMI handlers now validate input pointers

- SMI handlers now validate pointer + offsets received from the OS for overlap with SMRAM before using it (SmmIsBufferOutsideSmmValid). This does not block exploits using SMI handlers as proxies to attack hypervisor pages (Hyper-V, Windows 10 Virtual Secure Mode)
- Most recently, EDKII implemented `CommBuffer` at fixed memory location to mitigate attacks on hypervisors and reporting to Windows through the Windows SMM Mitigations ACPI Table (WSMT)

<table>
<thead>
<tr>
<th>Length</th>
<th>Bit offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>FIXED_COMM_BUFFERS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If set, expresses that for all synchronous SMM entries, SMM will validate that input and output buffers lie entirely within the expected fixed memory regions.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>COMM_BUFFER_NESTED_PTR_PROTECTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If set, expresses that for all synchronous SMM entries, SMM will validate that input and output pointers embedded within the fixed communication buffer only refer to address ranges that lie entirely within the expected fixed memory regions.</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>SYSTEM_RESOURCE_PROTECTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If set, expresses that firmware has taken steps ensuring that configuration for any system resources that are not configurable through an architectural mechanism (e.g., ACPI or PCI MMIO) must be locked by firmware before transitioning to Windows.</td>
</tr>
</tbody>
</table>
Memory-Mapped I/O (MMIO)
PCI Express

- PCI Express Fabric consists of PCIe components connected over PCIe interconnect in a certain topology (e.g. hierarchy)
- *Root Complex* is a root component in a hierarchical PCIe topology with one or more PCIe root ports
- Components: *Endpoints* (I/O Devices), *Switches*, PCIe-to-PCI/PCI-X *Bridges*
- All components are interconnect via PCI Express Links
- Physical components can have up to 8 physical or virtual *functions*
- Some endpoints are *integrated* into Root Complex
PCle Config Space Layout

Figure 7-3: PCI Express Configuration Space Layout

Source: PCI Express Base Specification Revision 3.0
PCI/PCle Config Space Access

1. Software uses processor I/O ports CF8h (control) and CFCh (data) to access PCI configuration of bus/dev/fun. Address (written to control port) is calculated as:

   \[ \text{bus} \ll 16 \ | \ \text{dev} \ll 11 \ | \ \text{fun} \ll 8 \ | \ \text{offset} \ & \sim 3 \]

   - 32 * 8 * 100h per bus
   - 8 * 100h per device
   - 100h bytes of CFG header

2. Enhanced Configuration Access Mechanism (ECAM) allows accessing PCle extended configuration space (4kB) beyond PCI config space (256 bytes)
   - Implemented as memory-mapped range in physical address space split into 4kB chunks per B:D.F
   - Register address is a memory address within this range

   \[ \text{MMCFG base} + \text{bus} \times 32 \times 8 \times 1000h + \text{dev} \times 8 \times 1000h + \text{fun} \times 1000h + \text{offset} \]
Memory-Mapped I/O

- Devices need more space for registers
- Memory-mapped I/O (MMIO)
- MMIO range is defined by Base Address Registers (BAR) in PCI configuration header
- Access to MMIO ranges forwarded to devices
MMIO vs DRAM

- High DRAM
  - Top of Low DRAM
    - SMM Memory
    - Graphics Memory
    - BAR 1 – BAR n
    - Direct-mapped BIOS, APIC, TPM…
  - Low DRAM
- Low MMIO

Memory
- Low DRAM
- 4GB
## MMIO BARs

<table>
<thead>
<tr>
<th>MMIO Range</th>
<th>BAR</th>
<th>Base</th>
<th>Size</th>
<th>En</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTTMMADR</td>
<td>00:02.0 + 10</td>
<td>00000000F0000000</td>
<td>00400000</td>
<td>1</td>
<td>Graphics Translation Table Range</td>
</tr>
<tr>
<td>SPIBAR</td>
<td>00:1F.0 + F0</td>
<td>00000000FED1F800</td>
<td>00000200</td>
<td>1</td>
<td>SPI Controller Register Range</td>
</tr>
<tr>
<td>HDABAR</td>
<td>00:03.0 + 10</td>
<td>0000007FFFFFF000</td>
<td>00001000</td>
<td>1</td>
<td>HD Audio Register Range</td>
</tr>
<tr>
<td>GMADR</td>
<td>00:02.0 + 18</td>
<td>00000000E0000000</td>
<td>00001000</td>
<td>1</td>
<td>Graphics Aperture</td>
</tr>
<tr>
<td>DMIBAR</td>
<td>00:00.0 + 68</td>
<td>00000000FED18000</td>
<td>00001000</td>
<td>1</td>
<td>Root Complex Register Range</td>
</tr>
<tr>
<td>MMCFG</td>
<td>00:00.0 + 60</td>
<td>00000000F8000000</td>
<td>00001000</td>
<td>1</td>
<td>PCI Express Register Range</td>
</tr>
<tr>
<td>RCBA</td>
<td>00:1F.0 + F0</td>
<td>00000000FED1C000</td>
<td>00004000</td>
<td>1</td>
<td>PCH Root Complex Register Range</td>
</tr>
<tr>
<td>MCHBAR</td>
<td>00:00.0 + 48</td>
<td>00000000FED10000</td>
<td>00008000</td>
<td>1</td>
<td>Memory Controller Register Range</td>
</tr>
</tbody>
</table>
MMIO Range Relocation

• MMIO ranges can be *relocated* at runtime by the OS
  • OS would write new address in BAR registers

• Certain MMIO ranges cannot be relocated at runtime
  • Fixed (e.g. direct-access BIOS range)
  • Or locked down by the firmware (e.g. MCHBAR)
MMIO BAR Issues
Firmware use of MMIO

Firmware configures chipset and devices through MMIO

SMI handlers communicate with devices via MMIO registers
MMIO BAR Issue

Exploit with PCI access can modify BAR register and relocate MMIO range

On SMI interrupt, SMI handler firmware attempts to communicate with device(s)

It may read or write “registers” within relocated MMIO
Examples of MMIO BARs accessed in SMM

- EHCI (USB 2.0) controller MMIO BAR (B0:D26:F0, B0:D29:F0)
- GBe LAN MMIO BAR (B0:D25:F0)
- Root Complex Block Address (RCBA) on earlier platforms (B0:D31:F0)
- SPI BAR on Skylake or later generations (B0:D31:F5)
- AHCI (SATA) controller MMIO BAR (B0:D31:F2, B0:D31:F5)
- xHCI (USB 3.0) controller MMIO BAR (B0:D20:F0)
- Integrated Graphics Device MMIO BAR (B0:D2:F0)
- B1:D0.F0 MMIO BAR
- …
SPI Controller MMIO BAR (Access to SPI Flash)

# chipsec_util.py uefi var-write B 55555555-4444-3333-2211-000000000000 B.bin
# chipsec_util.py mmio dump SPIBAR
Finding MMIO BAR issues at runtime

Goal: Find all MMIO registers modified by SMI handler

1. Dump MMIO range
2. Trigger SMI
3. Dump MMIO range and compare all registers

Problem: many registers are modified by devices all the time! Up to 30,000 registers change in Graphics Device MMIO
Finding MMIO BAR issues at runtime

Goal: Find all MMIO registers modified by SMI handler

1. Dump MMIO range multiple times
2. Find all registers which frequently change without SMM
3. Dump MMIO range
4. Trigger SMI
5. Dump MMIO range and compare all registers
6. Find registers which don’t normally change
7. Repeat this multiple times to confirm suspected registers are actually being modified in SMM
8. Copy original contents of MMIO range to memory
9. Relocate MMIO range (change its base address) to this memory
10. Generate SMI
11. Monitor changes in memory at suspected offsets
Monitoring changes in USB MMIO BAR

Module: Monitors MMIO changes done by SMI handlers

Configuration:
- MMIO BAR names: ['USBBAR']
  - Generate SMI: True
  - SMI codes: [0x00:0x00]
- SMM comm buffer (EBX): 0x00000000D9469000
- MMIO BAR 'USBBAR': base = 0x00000000F063C000, size = 0x00001000
- reading contents of MMIO BARs ['USBBAR']
  - reading 'USBBAR'
- calculating normal MMIO BAR differences..
- 'USBBAR' normal difference (5 diffs):
  - diff0: 0 regs []
  - ...
  - diff19: 2 regs [70, 74]
  - 2 regs changed: [70, 74]

Fuzzing SMIs..

SMI# 00: data 00, func (ECX) 0x00000000
- reading 'USBBAR'
- generating SMI
- reading 'USBBAR'
- differ 'USBBAR' (1024 regs)
  - 2 regs changed: [70, 77]
  - new regs: [77]

New changes found!
- repeating SMI
- reading 'USBBAR'
- differ 'USBBAR' (1024 regs)
  - 2 regs changed: [70, 74]
  - new regs: []
MMIO BAR Issues in UEFI Firmware
Finding MMIO BAR issues in binaries

1. Consider MMIO BAR (MBARA) of GBe LAN device (B0:D25:F0) at offset \texttt{0x10}.

2. Legacy PCIe config address is
   \[(25<<11) + 0x10 = \texttt{0xC810}\]
   \[0x8000\texttt{C810} \text{ if with Enable bit (31) set}\]

3. Memory-mapped ECAM address is ECAM base + offset to 4kB page of B0:D25:F0 + BAR register offset
   \[0x\texttt{F8000000} + 0xC8010 = \texttt{0xF80C8010}\]

4. Look for these constants in the binaries of SMI handlers.
**GBe LAN MMIO BAR (B0:D25:F0)**

```c
__int16 GBE_MMIO_access_func()
{
    signed int v0; // ebp@1
    unsigned int *v1; // rax@1
    __int64 MBARA_MMIO; // rbx@3
    unsigned int v3; // er12@4
    int v4; // er14@4
    unsigned int v5; // edi@4
    int v6; // er13@5
    int v7; // er8@10
    int v8; // edx@10
    int *v9; // rdi@18

    v0 = 0;
    LOWORD(v1) = MEMORY[0xF800F048];
    if ( !(*(__WORD*)((MEMORY[0xF800F048] & 0xFFFFFFFF) & 0x20) )
        { 
            LOWORD(v1) = MEMORY[0xF80C80CC];
            if ( !(MEMORY[0xF80C80CC] & 3) )
                { 
                    MBARA_MMIO = MEMORY[0xF80C8010];
                    LOWORD(v1) = *(__DWORD*)((MEMORY[0xF80C8010] + 0x5800i64);
                    if ( (unsigned __int8)v1 & 1 )
                        { 
                            v3 = *(__DWORD*)((MEMORY[0xF80C8010] + 0x5400i64);
                            v4 = *(__DWORD*)((MEMORY[0xF80C8010] + 0x5404i64);
                            v5 = 0;
                            *(__DWORD*)((MEMORY[0xF80C8010] + 0x500i64) |= 0x20u;
```
GBe LAN MMIO BAR (B0:D25:F0)

Access to unchecked MBARA MMIO
EHCI MMIO BAR (B0:D29:F0)

```c
v0 = 0;
dev1 = 0x1D;  // EHCI controller
dev2 = 0x1A;  // EHCI controller
v9 = 0;
v11 = 0;
result = sub_180001878();
if ( (unsigned __int8)result > 8u )
{
    do
    {
        v2 = (unsigned __int8)*(&v9 + 2 * v0) << 12;
        v3 = ((unsigned __int8)&(dev1 + 2 * v0) << 15) + 0xF8000000164;
        int64_t Mem_BAR = *(unsigned int *)(v3 + v2 + 0x10);
        v5 = (__int16 *)(v3 + v2 + 0x54);
        v6 = *(__WORD *)(v3 + v2 + 0x54);
        v7 = &v5;
        if ( (__DWORD)Mem_BAR != -1 )
        {
            if ( (*(__WORD *)(((unsigned __int8)*(&dev1 + 2 * v0) << 15)
                      + 0xF8000000164
                      + ((unsigned __int8)*(&v9 + 2 * v0) << 12)
                      + 0x54) & 3) == 3 )
            {
                *(__WORD *)(((unsigned __int8)*(&dev1 + 2 * v0) << 15)
                             + 0xF8000000164
                             + ((unsigned __int8)*(&v9 + 2 * v0) << 12)
                             + 0x54) = v6 & 0xFFFC;
                *(__DWORD *)(v3 + v2 + 0x10) = Mem_BAR;
                sub_180001A28(v3 + v2 + 0x54);
            }
            if ( *(__DWORD *)(Mem_BAR + 0x20) & 1 )
            {
                MemorySetResetValues32((unsigned int *)(Mem_BAR + 0x20), 0, 48);
                MemorySetResetValues32((unsigned int *)(Mem_BAR + 0x20), 1, 0);
                v5 = (__int16 *)(v3 + v2 + 0x54);
            }
        }
        v4 = v3;
    }
    while (result);
}
```

Calculate EHCI BAR register address in MMCFG:
0xF8000000 + 0x1D<<15 + 0x10

Read EHCI MMIO BAR

Modify MMIO register 0x20 in EHCI MMIO range
Finding MMIO BAR issues in binaries

Identify functions reading PCI config registers via legacy or ECAM access and find ones reading BAR registers using above constants

Legacy PCI config read

```asm
pci_dword_read proc near
    ; CODE XREF: sub_1000850+2E_tp
    ; sub_100085C4+82_tp
    sub    rsp, 28h
    mov    edx, ecx
    mov    ecx, 0CF8h
    call   outdword
    mov    ecx, 0CFCh
    add    rsp, 28h
    jmp    indword
pci_dword_read endp
```

Extended PCIe config access through MMCFG

```c
union __int32 __fastcall pcie_read_dword(__int64 a1, __int8 a2, __int8 a3, __int16 a4)
{
    if ( a4 >= 0x100u )
        return *(DWORD *)(a3 << 12) + (a2 << 15) + ((unsigned __int8)a1 << 20) + (unsigned int)a4 - 0x8000000;
    outdword(0xCF8u, a4 & 0xFC | ((a3 | 8 * (a2 | 32 * (unsigned __int8)a1)) << 8) | 0x80000000);
    return indword(0xCFCu);
}
```
MMIO BAR Issues in Coreboot Firmware
Finding MMIO BAR issues in the source code

1. Find functions within SMI handlers which read MMIO BAR PCI config registers (offsets $0x10\text{--}0x24$ or chipset specific offsets for integrated devices)
   - BAR registers can be read using memory-mapped config reads (offsets in ECAM memory space). In this case, normal memory reads will be used

   ```c
   reg_base = (void *)((uintptr_t)pci_read_config32(SA_DEV_IGD, PCI_BASE_ADDRESS_0) & ~0xf);
   ```

2. Find all memory accesses to offsets off of BAR addresses within SMI handlers

   ```c
   write32(reg_base + PCH_PP_CONTROL, pp_ctrl); // << memory write of modified pp_cntrl value
   read32(reg_base + PCH_PP_CONTROL);
   ```

3. Can often search the names of the BAR registers and MMIO ranges (e.g. SPI_BAR0, RCBA/RCRB, PCI_BASE_ADDRESS etc.)
mainboard_io_trap_handler SMI handler

static void mainboard_smi_brightness_down(void)
{
  u8 *bar;
  if ((bar = (u8 *)pci_read_config32(PCl_DEV1, 0, 0, 0x10))) {
    printk(BIOS_DEBUG, "bar: %08X, level %02X\n", (unsigned int)bar,
           *(bar+LVTMA_BL_MOD_LEVEL) &= 0xf0;
    if (*(bar+LVTMA_BL_MOD_LEVEL) > 0x10)
      *(bar+LVTMA_BL_MOD_LEVEL) -= 0x10;
  }
}

static void mainboard_smi_brightness_up(void)
{
  ...
  if (*(bar+LVTMA_BL_MOD_LEVEL) < 0xf0)
    *(bar+LVTMA_BL_MOD_LEVEL) += 0x10;
  ...
  int mainboard_io_trap_handler(int smif)
{
  ...
    switch (smif) {
      ...
      case SMI_BRIGHTNESS_UP:
        mainboard_smi_brightness_up();
        break;
      ...
      case SMI_BRIGHTNESS_DOWN:
        mainboard_smi_brightness_down();
  }
southbridge_smi_sleep SMI handler

```c
static void backlight_off(void)
{
    void *reg_base;
    uint32_t pp_ctrl;
    uint32_t bl_off_delay;

    reg_base = (void*)((uintptr_t)pci_read_config32(SA_DEV_ID, PCI_BASE_ADDRESS_0) & ~0xf);

    /* Check if backlight is enabled */
    pp_ctrl = read32(reg_base + PCH_PP_CONTROL);
    if (!(pp_ctrl & EDP_BLC_ENABLE))
        return;

    /* Enable writes to this register */
    pp_ctrl |= ~PANEL_UNLOCK_MASK;
    pp_ctrl |= ~PANEL_UNLOCK_REGS;

    /* Turn off backlight */
    pp_ctrl &= ~EDP_BLC_ENABLE;
    write32(reg_base + PCH_PP_CONTROL, pp_ctrl);
    read32(reg_base + PCH_PP_CONTROL);

    * Figure out SLP_TYP */
    reg32 = inl(ACPI_BASE_ADDRESS + PM1_CNT);
    printk(BIOS_SPEW, "SMI#: SLP = 0x%x\n", reg32);
    slp_typ = (reg32 >> 10) & 7;
    switch (slp_typ) {
        ...
        case SLP_TYP_S5:
            printk(BIOS_DEBUG, "SMI#: Entering S5 (Soft Power off)\n");
            /* Turn off backlight if needed */
            backlight_off();
    }
}
```
Limitations

1. Exploit can overwrite specific offsets off of aligned addresses
   - MMIO ranges are typically normally (size) aligned
   - Most MMIO ranges are 4kB large (Graphics MMIO is 2-4MB)
   - Example: 16kB aligned Root Complex Base + 0x38xx (SPI registers)
   - PCI architecture allows MMIO ranges as small as 16 bytes

2. Exploit may not be able to control values written
   - Firmware SMI handlers typically write specific values to MMIO registers
   - Often do Read-Modify-Write: \( \text{reg} \pm 0x10 \)
   - Certain SMI handler may write attacker-supplied data
   - `SetVariable` SMI handler write contents of UEFI variable supplied by the OS to SPI_DATAx registers in SPIBAR MMIO range
Limitations

1. Many conditions for SMI handler to start communicating with I/O device/controller
   - Device present/enabled, mode/feature supported
   - Is platform in ACPI mode?
   - Other SMM code may also use fake MMIO (and hang)
   - SMI may get triggered on difficult events – power button, on S3 resume, etc.

2. Often, SMI handlers implement protocol rather than just reading or writing to MMIO registers
   - IF Bit X in Reg1 is set THEN Write to Reg2
   - Poll until certain bits are set/cleared in MMIO register (wait until SPI cycle complete)
   - When SMI handler waits for the device to respond or cycle to complete then it’ll hang after MMIO BAR is relocated

3. Non PCI-architectural BAR registers are locked down by boot firmware and cannot be relocated (MCHBAR, DMIBAR etc.)
Mitigations

Option 1. SMI handlers can verify MMIO BAR doesn’t overlap with SMRAM

Option 2. Firmware can verify that MMIO BAR is not in DRAM (e.g. between TOLUD and 4GB or above TOUUD). This would ensure all BARs used by firmware are within MMIO

Option 3. Firmware can reserve default MMIO range for all BARs. Before accessing MMIO range, SMI handlers can relocate BARs to the default range if they point to somewhere else
SPIBAR Mitigation Example

- On latest platforms, SPI MMIO is a separate 4kB range rather than a part of Root Complex MMIO.
- Firmware reserves `0xFE010000` page for SPI MMIO and programs `SPI_BAR0` register in SPI controller with this address.
- On any PCH SMI, SMI handler checks `SPI_BAR0` and restores it to `0xFE010000` if it’s been relocated.

```
PCI 00:1F.05 + 0x10: 0xFE010000
CPU0: RDMSR( 0x34 ) = 0000000000000075 (EAX=00000075, EDX=00000000)
write 0x89F50000 to PCI 00:1F.05 + 0x10
PCI 00:1F.05 + 0x10: 0x89F50000
writing EFI variable Name='test' GUID={55555555-4444-3333-2211-000000000000} from 'test.bin'
writing EFI variable was successful
PCI 00:1F.05 + 0x10: 0xFE010000
CPU0: RDMSR( 0x34 ) = 0000000000000079 (EAX=00000079, EDX=00000000)
IN 0x00B2 -> 0x000000EC (size = 0x01)
```

Relocating SPI BAR to memory

**var-write** triggers SMI writing variable to SPI BAR. It succeeded!

SMI handler restored SPI BAR to the original location
Tools to assist in finding/analyzing these issues

tools.smm.rogue_mmio_bar

Attempts to create fake MMIO ranges in memory, relocate hardware MMIO BARs to the fake memory, then observe changes made by SMI handlers in relocated MMIO ranges

tools.smm.bar

Simply monitors changes made by SMI handlers in MMIO registers of specified MMIO BARs
Conclusion

• The root cause is that firmware assumes hardware is trusted
• Hardware registers like PCI Base Address Registers can be modified by runtime software (some are locked down)
• Firmware shouldn’t assume addresses in BAR registers are correct and should treat them as untrusted input
• Boot firmware should also validate contents of BAR registers upon resume from sleep if it restores them from S3 boot script
Thank You!