ERNW
providing security.

ERNW Newsletter 43 / May 2014

Security Assessment of Microsoft Hyper-V

Version:	1.1
Date:	6/2/2014
Author(s):	Felix Wilhelm, Matthias Luft

Table of Content

1 INTRODUCTION

Hyper-V is Microsoft's first bare-metal hypervisor and its first native hypervisor with full support for the x86-64 architecture. It is heavily marketed as a competitive alternative to similar virtualization solutions like VMware ESXi, Xen or KVM and is gaining market share due to its easy integration with other Microsoft solutions. Besides this enterprise orientation, Hyper-V is also used in a variety of other platforms such as Microsoft Azure or the Xbox One gaming console.

Despite the growing importance of Hyper-V, very little research which is publicly available was performed until now. After almost six years on the market, only a handful of Denial-of-Service vulnerabilities were patched. Even though Microsoft's SDL has an impressive track record of producing secure software, this seems like an unrealistic low amount of vulnerabilities for such a complex software.

In this paper we describe our research on the security of Hyper-V against attacks from a malicious unprivileged guest VM. We focused on vulnerabilities that are related to memory corruptions, protocol parsing and design flaws but did not look into issues like side-channel and timing attacks or insecure management of the surrounding environment.

The remainder of this paper is organized as follows: Chapter 2 gives a technical overview about the Hyper-V architecture including its support for device emulation and synthetic devices. In addition we will highlight how Hyper-V is used inside the Azure environment and what security consequences follow out of that. Chapter 3 discusses the attack surface that exists from the perspective of a malicious virtual machine. This surface will be broken down into the separate functionality and described as for the input vectors that are available to an attacker and our approach in assessing those. Chapter 4 discusses the most interesting result of our research: The important vulnerability MS13-092 in Hyper-V's hypercall functionality. Chapter 5 describes interesting targets for further research and is followed by the final conclusion.

2 HYPER-V ARCHITECTURE

The following subsections will describe Hyper-V's internal architecture in detail. The covered terms, components, and internals are required for the analysis of the attack surface in Chapter 3.

2.1 Overview

Hyper-V is a Type 1 hypervisor and thus runs directly on the hardware not relying on an underlying operating system. This may be quite surprising for a number of users, because the installation procedure starts with a regular Windows Server installation and Hyper-V is added as an additional role.

However, during the installation procedure the Windows operating system gets turned into a (highly privileged) Hyper-V partition and is under control of the hypervisor after the next reboot.

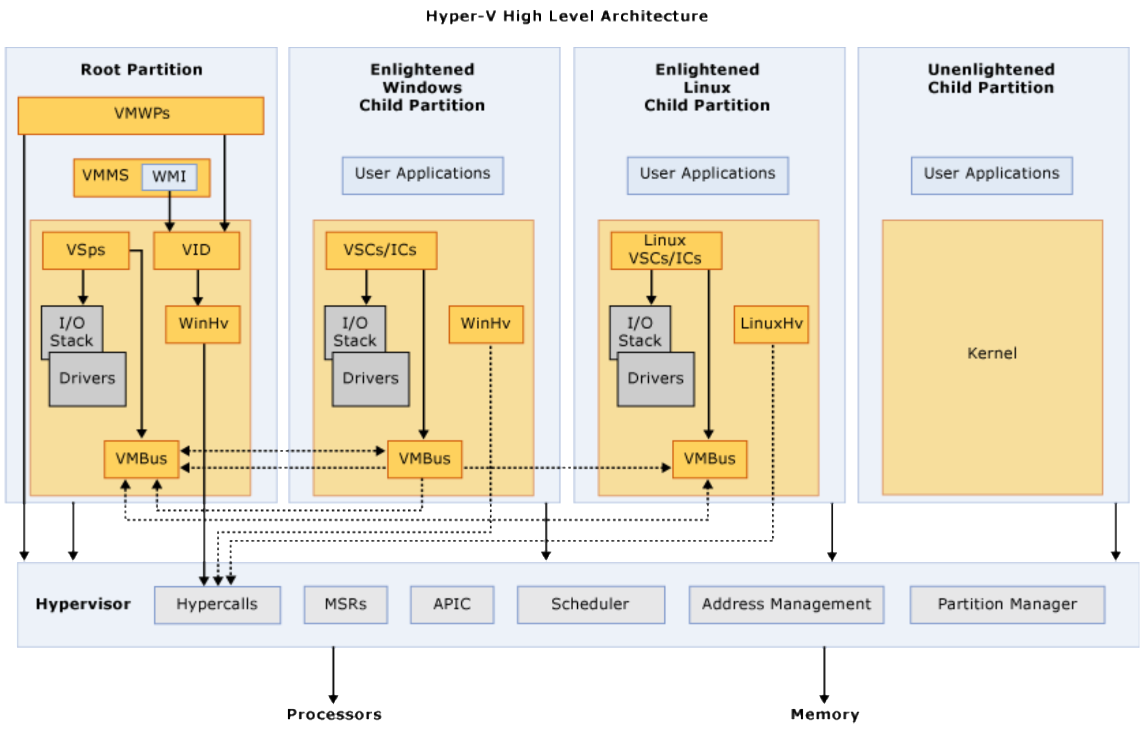


Figure 1: High Level Architecture[1]

Figure 1 shows an overview of the architecture of Hyper-V. Microsoft calls virtualized operating systems *partitions* and distinguishes between the (privileged) root partition and its (unprivileged) child partitions. The root partition is responsible for management and configuration of all other partitions and is fully trusted by the hypervisor (comparable to Xen's Dom0). Hyper-V supports unmodified operating systems, which do not know that they are running in virtual environment and are called *unenlightened*. In order to make this possible the hypervisor transparently emulates certain standard devices that are supported by all modern operating systems.

[1] *http://msdn.microsoft.com/de-de/library/cc768520(en-us).aspx*

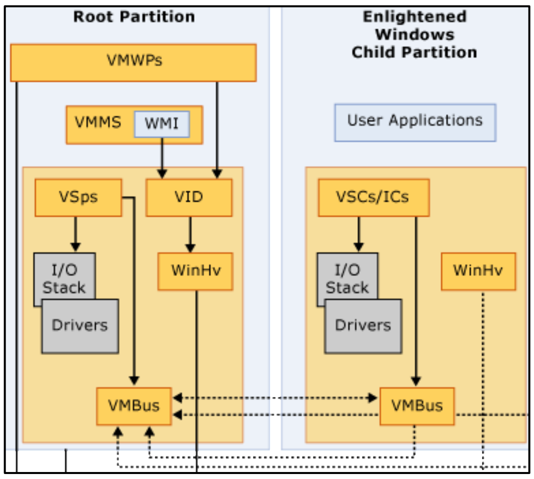


Figure 2: Hyper-V Components

Enlightened operating systems with explicit support for Hyper-V can make use of more advanced features to increase performance. First of all they can use *hypercalls* to communicate directly with the hypervisor, a mechanism quite similar to the standard system call interface of operating system kernels. Hypercalls are one of the publicly documented Hyper-V APIs and are described in Chapter 0. In addition, the VMBus mechanism is used to facilitate direct high-speed communication between child and root partitions. This is mainly used for so-called *synthetic* devices that can be used by all enlightened partitions and are much faster than their emulated counterparts[2].

Because Microsoft decided keep the Hyper-V hypervisor as minimal as possible, large parts of the functionality are outsourced to the root partition. This includes emulated as well synthetic devices and all advanced management interfaces. This decision significantly reduces the complexity of the hypervisor code itself to about 100.000 lines of code. However it has only limited impact on the security of the overall Hyper-V environment, as we will demonstrate later on.

2.2 VMWP, VSC and VSP

Figure 2 shows the core components inside the root partition and enlightened children. Each child partition has a corresponding *virtual machine worker process* (VMWP) that is running as a normal user space process in the root partition. The worker process performs management duties like snapshots or migrations. But it also implements the aforementioned device emulation, as well as a couple of synthetic devices that are not performance critical.

The implementation of these features inside a user space process has a number of advantages: First, bugs that result in a crash or high resource consumption do not affect the stability of the root partition, other VMs, or the overall hypervisor. Second, the worker processes can be executed with low privileges. This means that an attacker who is able to exploit a vulnerability in the device emulation layer still needs to perform an escalation of privilege for a full compromise of the root partition. However, due to the need for additional context switches, a user space implementation is not suitable for performance critical devices like networking and storage.

Such devices are implemented in *Virtualization Service Providers* (VSP). VSPs are drivers running in the kernel of the root partition and are therefore very interesting targets for an attacker. The VSP's counterparts in the child partitions are the *Virtualization Service Clients* (VSC). All modern versions of Windows already come with these kernel drivers included and provide the support of the operating system for the synthetic devices. In addition Linux and FreeBSD include open source implementations of multiple VSC that were provided by Microsoft engineers. For Linux this code is called *Linux Integration Services* and is nowadays directly included in the kernel tree. However, a more current version with additional features can also be found in a Github repository[3].

Device support alone is not sufficient for a virtualization solution and the core task of a hypervisor is the virtualization of CPU and memory. As almost all x86-64 hypervisors, Hyper-V uses hardware-assisted virtualization. This means that

[2] *For example, emulated devices only support 100MBps NICs while synthetic 1000MBps NICs are available.*
[3] *https://github.com/LIS/LIS3.5/tree/master/hv-rhel6.x/hv*

Hyper-V uses the extended virtualization instruction sets Intel VT[4] and AMD-V[5]. Because all of our research was performed on Intel hardware we will use the Intel-specific terminology throughout this paper.

2.3 Hardware-assisted Virtualization

Intel VT adds support for two different processor operating modes: VMX root mode and VMX non-root mode. The hypervisor operates in VMX root mode, while all partitions are running in non-root mode.

VMs operating in non-root mode do not experience any performance impact, but certain events can trigger a context switch back to the hypervisor. These context switches are called VM exits and can occur for a number of reasons like the execution of certain instructions (such as interrupt handling) or access to special system registers.

One of the goals of Intel VT is to be transparent to the guest operating system. Therefore, VM exits triggered by instructions or system register access require a complete emulation of their behavior by the hypervisor. While this is trivial for many exit reasons, complete support for all corner cases of the x86 architecture is difficult and error prone. A complete discussion of security problems of hardware-assisted virtualization is out of the scope of this paper[6], however later sections will describe several properties of the VT architecture in more detail.

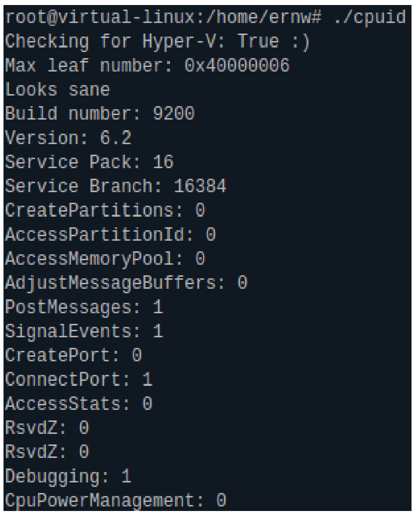
Even though Intel VT is designed to be transparent to the guest operating system, Hyper-V's support for enlightened partition requires that a virtualized operating system can examine whether it is running in Hyper-V or not. This is supported by a dedicated interface discovery mechanism described in the following section.

2.4 Interface Discovery

When the CPUID instruction is executed by a virtualized partition, a VM exit is triggered and Hyper-V modifies the typical CPUID return value by adding Hyper-V specific information. A CPUID call with EAX=1 will result in an ECX register with the MSB set. This indicates that a hypervisor is present. Further information about the Hyper-V version and partition permissions can be queried by executing CPUID with input values between 0x40000000 and 0x40000006. The figure on the right shows the output of a small wrapper utility executed on a Windows Server 2012 Hyper-V system.

Interestingly, the returned information include the exact version of Hyper-V , as well the hypercall permissions of the calling partition. These details can help an attacker to decide if certain vulnerabilities are present and allow better targeting for attacks.

Furthermore, using this Interface discovery mechanism in a Azure VM returns quite interesting results as described in the next section.



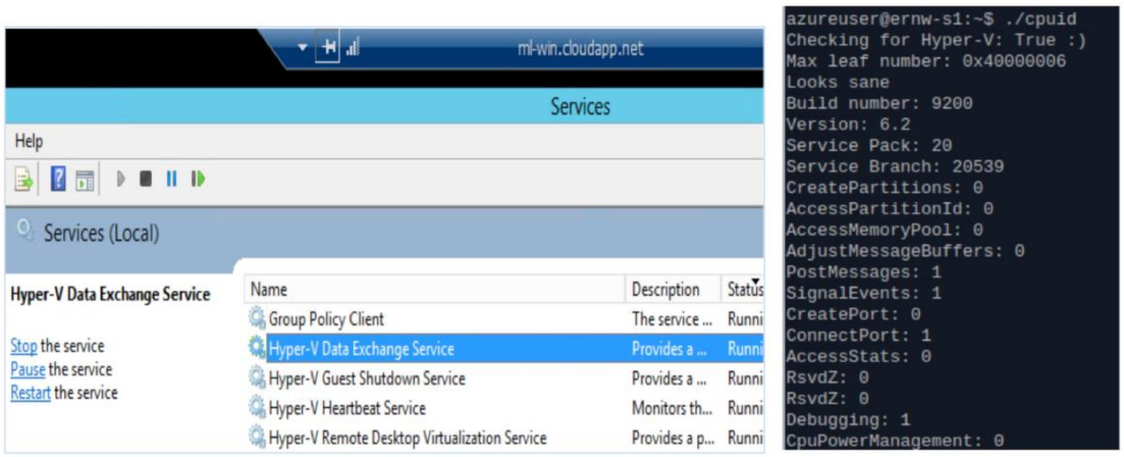
```
root@virtual-linux:/home/ernw# ./cpuid
Checking for Hyper-V: True :)
Max leaf number: 0x40000006
Looks sane
Build number: 9200
Version: 6.2
Service Pack: 16
Service Branch: 16384
CreatePartitions: 0
AccessPartitionId: 0
AccessMemoryPool: 0
AdjustMessageBuffers: 0
PostMessages: 1
SignalEvents: 1
CreatePort: 0
ConnectPort: 1
AccessStats: 0
RsvdZ: 0
RsvdZ: 0
Debugging: 1
CpuPowerManagement: 0
```

Figure 3: CPUID Instruction

[4] *Intel® 64 and IA-32 Architectures Software Developer's Manual : Volume 3 (3A, 3B & 3C): System Programming Guide]*
[5] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*
[6] *Interested readers should watch the 30c3 talk Virtually Impossible by Gal Diskin: https://www.youtube.com/watch?v=GoipioWrzAg*

2.5 Azure Hypervisor = Hyper-V ?

Officially, the Microsoft Azure cloud runs on a hypervisor called the "Azure hypervisor", which is not the same as Hyper-V. However, even a cursory look at an Azure VM shows that the both hypervisors are at least strongly related. Figure 4 shows a screenshot of an Azure VM with the standard Hyper-V VSC services running and the CPUID instruction executed in a Azure VM.



Name	Description	Status
Group Policy Client	The service ...	Runni
Hyper-V Data Exchange Service	Provides a ...	Runni
Hyper-V Guest Shutdown Service	Provides a ...	Runni
Hyper-V Heartbeat Service	Monitors th...	Runni
Hyper-V Remote Desktop Virtualization Service	Provides a p...	Runni

```
azureuser@ernw-s1:~$ ./cpuid
Checking for Hyper-V: True :)
Max leaf number: 0x40000006
Looks sane
Build number: 9200
Version: 6.2
Service Pack: 20
Service Branch: 20539
CreatePartitions: 0
AccessPartitionId: 0
AccessMemoryPool: 0
AdjustMessageBuffers: 0
PostMessages: 1
SignalEvents: 1
CreatePort: 0
ConnectPort: 1
AccessStats: 0
RsvdZ: 0
RsvdZ: 0
Debugging: 1
CpuPowerManagement: 0
```

Figure 4: Hyper-V Integration Services in Azure VM & CPUID Instruction in Azure

In addition, we can use the interface discovery mechanisms discussed in the last section. Figure 5 shows the output of the CPUID command: The only recognizable differences are in the service pack and service branch number. Even the partition privileges are identical to the permission of a default Hyper-V VM.

Sharing a code base between Hyper-V and Azure makes sense from an engineering standpoint. For security researchers or malicious attackers targeting Azure, it has the big advantage of allowing offline analysis of the hypervisor. By concentrating on the Hyper-V attack surface that is also relevant for Azure, the chance to discover vulnerabilities with a serious impact on the Azure cloud is quite high.

2.6 Information Sources

Even though Hyper-V is a proprietary software product, there are several public information sources that can be used to get a better understanding of its implementation:

- **Hypervisor Top Level Functional Specification[7]:** This specification includes a detailed discussion of the Hypercall API and the Interface discovery mechanisms.
- **Patent Applications[8]:** Multiple public patent applications describe implementation details of Hyper-V. This includes the mechanisms used for Child-Parent communication as well as protocol specifications. While most applications are quite hard to read in comparison to normal technical documents, they can still help to gain an understanding about internal nomenclature and high level architecture.

[7] *http://blogs.msdn.com/b/virtual_pc_guy/archive/2014/02/17/updated-hypervisor-top-level-functional-specification.aspx*
[8] *http://www.faqs.org/patents/app/20120084517*

- **Linux Integration Services**[9]**:** As mentioned before, the Linux Integration Services are an open source implementation of VSCs, VMBus and the hypercall API for Linux. They ease the understanding of implementation details in these parts of Hyper-V and are extremely helpful when developing tools to evaluate and attack these APIs.
- **Singularity Header Files**[10]**:** Singularity is an open source research OS developed by Microsoft Research. Its SVN repository contains a number of proprietary Windows header files, including several for the Microsoft Hyper-V guest interface.
- **Common Criteria Certification Documents**[11]**:** The Common Criteria certification requires a comprehensive documentation to be available. This documentation is publicly available.
- **Binaries:** Of course, the most accurate and detailed information can be extracted out of the implementation itself. Table 1 lists several of the files we analyzed during our research. The next section describes some of the challenges involved with reversing these files.

Filename	Description
`hvix64.exe` / `hvax64.exe`	Core Hypervisor executables, for Intel and AMD. Includes all code that runs in VMX root mode after boot process is finished
`vmwp.exe`	Executable for VM worker processes. Includes code for device emulation, as well as several synthetic devices
`vmswitch.sys`	Windows Kernel driver that implements the Networking VSP
`storvsp.sys` / `vhdmp.sys`	Kernel driver for Storage VSP

Table 1: Hyper-V executables

2.7 Reverse Engineering Pitfalls

The reverse engineering of Hyper-V/some of its core functionality, we had to overcome several challenges. Those challenges are very relevant for any security researcher analyzing Hyper-V and we hope the description in the following subsections will provide helpful insights.

2.7.1 Symbol Porting

To understand the implementation of security relevant core functionality like VM exit handling or the hypercall API, reverse engineering of the core hypervisor binaries must be performed. Our research was performed on Intel hardware, so we worked with `hivx64.exe`. When trying to get an initial understanding of the included functionality we have to overcome some challenges.

First, no public debugging symbols are available for the binary. This increases the effort necessary to identify security relevant code regions. Second, the standard method to identify interesting function by searching for debug output and other human readable strings does not work due to the very low amount of such functionality in the hypervisor itself. Finally, we can't rely on the usage of known APIs or libraries. In contrast to user space programs or kernel drivers, the hypervisor is a single statically linked executable.

[9] *https://github.com/LIS*
[10] *https://singularity.svn.codeplex.com/svn/base/Windows/Inc/*
[11] *http://www.commoncriteriaportal.org/files/epfiles/0570b_pdf.pdf*

However, there are some techniques we can use to identify interesting functionality: A researcher named Gerhart describes his approach to symbol porting in a detailed blog post on securitylab.ru[12]: `hvix64.exe` shares a lot code with `winload.exe` and `hvloader.exe`. Public debugging symbols are available for both of them. Using the popular BinDiff[13] software, we can identify shared functions and port the included symbols over to our executable. Unfortunately, the shared functionality mainly concerns networking code, the USB stack as well as the WinDBG debugger stub and is not that interesting for our goal of identifying the attack surface.

2.7.2 VMCS

As mentioned before, the hypervisor cannot rely on any external libraries. The only helpful documented functionality it uses are the Intel VMX instructions used to configure and manage the different virtual machines. As described in Volume 3 of the official Intel Manual[14], the central part of VMX are data structures called VMCS (Virtual-Machine Control Structures). A VMCS is separated into four logical parts:

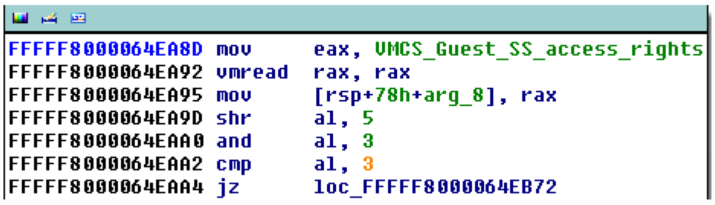
- Guest-state area
- Host-state area
- Control fields for VM-execution, VM-exit and VM-entry
- VM-exit information fields

The Guest-state area is used to store the processor state of the VM when it exits and passes control to the hypervisor. This includes certain control registers, as well as MSRs and segment selectors but most importantly the values of RIP, RSP and RFLAGS. When the VM exit was handled by the hypervisor and the virtual machine is continued, the processor state is loaded from its VMCS and execution can (potentially) continue transparently.

The Host-state area is loaded during a VM-exit and describes the initial state of the hypervisor when handling these exits. This makes it especially interesting for us, because the stored RIP and RSP values allow us to quickly identify the main exit handler as well as its stack location.

Control fields for VM-execution control the operation in VMX non-root mode. These fields control the handling of interrupts and certain types of instruction and decide which actions trigger a VM exit. They are a interesting target for security research, because insecure control field settings can lead to severe logical flaws.

Finally, the VM-exit information fields contain information about the VM-exit reason. These information are used throughout the VM exit handlers and are therefore quite relevant from a reversing standpoint.



```
FFFFF8000064EA8D mov     eax, VMCS_Guest_SS_access_rights
FFFFF8000064EA92 vmread  rax, rax
FFFFF8000064EA95 mov     [rsp+78h+arg_8], rax
FFFFF8000064EA9D shr     al, 5
FFFFF8000064EAA0 and     al, 3
FFFFF8000064EAA2 cmp     al, 3
FFFFF8000064EAA4 jz      loc_FFFFF8000064EB72
```

Figure 5: VMREAD instruction with human-readable field name

[12] *http://www.securitylab.ru/contest/444112.php*
[13] *http://www.zynamics.com/bindiff.html*
[14] *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3 (3A, 3B & 3C): System Programming Guide*

VMCS fields are read and written with dedicated instructions: VMREAD and VMWRITE. Both instruction require an encoded argument that describes the accessed VMCS field[15]. For our research we decided on using this information in a semi-automated approach. Using a IDAPython script, we translated all VMCS accesses into their human-readable version. This eases understanding of all code parts involving VMCS fields and allowed us to identify main functionality like VM exit handling and hypercall handler.

2.7.3 Debugging

Even though static analysis proved to be feasible to gain an initial understanding, we quickly decided that additional dynamic analysis was required for a more comprehensive analysis.

Interestingly, Hyper-V has integrated debugging functionality using Microsoft's WinDBG. Debugging Hyper-V itself works nearly the same way as debugging the Windows kernel, but is not quite as comfortable due to missing symbols and functionality[16].

Different access methods like Firewire, Ethernet and USB are supported, but in practice we had the most reliable results on physical hardware with standard serial ports[17]. Although debugging the hypervisor using physical hardware is definitely possible, a virtualization based method would be much more comfortable. Luckily, current versions VMware Workstation and VMware Fusion support a feature called nested virtualization. Nested virtualization allows the usage of Intel VT inside a virtualized machine and makes it possible to run Hyper-V as a normal VM.

By virtualizing Hyper-V itself we gain a number of advantages. In addition to the mentioned WinDBG interface we can now also used VMware's built-in GDB stub as an alternative debugging environment. In addition, snapshots can be used as an easy and fast way to get a physical memory dump. VMware Workstation/Fusion also provides support for sharing serial ports between virtual machines, which makes it possible to run both the debugger and the debugee in virtual machines[18].

While this still does not lead to a comfortable or feature rich debugging environment, it proved sufficient to perform the analyses required for our research.

[15] *A table containing all VMCS fields and the corresponding value can be found in APPENDIX B of the Intel Manual Volume 3.*
[16] *Hyper-V specific functionality seems to be included inside a WinDBG extension called hvexts.dll. Sadly, it is not publicly available.*
[17] *The actual setup including required commands is described here: http://msdn.microsoft.com/en-us/library/windows/hardware/ff540654(v=vs.85).aspx*
[18] *For VMware Fusion a small workaround is required, which is described here: http://www.insinuator.net/2014/01/serial-port-debugging-between-two-virtual-machines-in-vmware-fusion/*

3 Attack Surface And Testing Methodology

Based on the architectural overview presented in Chapter 2, we identified the following components as potential and most promising targets for VM breakout attacks:

- Device Emulation
- VMBus and Synthetic Devices
- Hypercall API
- (VM Exit handling)

The following sections will describe each of these components in more detail and present our approach at finding security issues. We did not perform more than a cursory assessment of general VM exit handling yet and will therefore not include this topic in this chapter.

3.1 Device Emulation

Device virtualization is one of the core responsibilities of every virtualization solution. While synthetic devices have better performance characteristics and a saner interface, they are not sufficient for general purpose virtualization. Older operating systems without explicit support for these devices are still dependent on the availability of "standard" hardware inside the virtual environment. This problem is normally solved using device emulation, where older hardware with wide support is emulated by the hypervisor.

Due to the high complexity of the emulated devices and the high performance requirements, device emulation is often a weak point in virtualization software. In 2007, Tavis Ormandy describes his approach at fuzzing emulated devices in multiple virtualization solutions, which discovered multiple vulnerabilities. In addition, the device emulation layer is a popular target for the Xen and KVM hypervisors, which use QEMU for the implementation of their devices.

Hyper-V supports a number of emulated devices for normal VMs:

- Network adapter
- S3 Trio graphic card
- Keyboard / Mouse
- IDE Controller

All of them are implemented in `vmwp.exe`, the VM worker process. Because the attack surface of emulated devices is quite well understood and fuzzing them requires no knowledge specific to a single hypervisor, we did not expect many results. Basic fuzzing of all devices triggered permanent crashes of the VM, which were triggered by assertions in the worker process. In addition, we were able to freeze the worker process with 100% CPU consumption, which required a hard kill using the task manager of the parent partition. This might be a relevant issue for some cloud environments but does not influence other VMs in any way and is therefore not critical.

Further static analysis of the `vmwp.exe` binary showed a lot of defensive checks and no obvious security vulnerabilities.

3.2 VMBus and Synthetic Devices

3.2.1 VMBus

As already mentioned, the VMBus is a mechanism used for communication between partitions. In Hyper-V's default configuration, communication is only allowed between child and root partition. The VMBus itself is implemented using memory pages that are mapped into multiple partitions, in the default case into the root partition and the target child partition. This means that data which is sent through the bus does not need to be copied "through" the hypervisor, which reduces the necessary number of context switches which improves performance significantly.

VMBus communication is separated into channels, where every channel consists of an incoming and an outgoing ring buffer. Because the ring buffer space is quite limited, performance critical components use an additional mechanism called *Guest Physical Address Descriptor Lists* (GPADLs), which allows the root partition to directly map additional pages of guest memory into its own address space.

The main consumers of the VMBus infrastructure are synthetic devices. This includes storage and networking, but also video drivers and some additional utility services (such as time synchronization, dynamic memory allocation for VMs and a Key-Value service).

As mentioned in Section 2.2, storage and networking are implemented as kernel drivers. This makes them interesting targets because a vulnerability in those drivers (e.g. parsing the input from attacker-controlled partitions) would result in a direct compromise of the root partition's kernel space and thus the complete root partition. In contrast, the video driver as well as all utility services are implemented as part of the VM worker process.

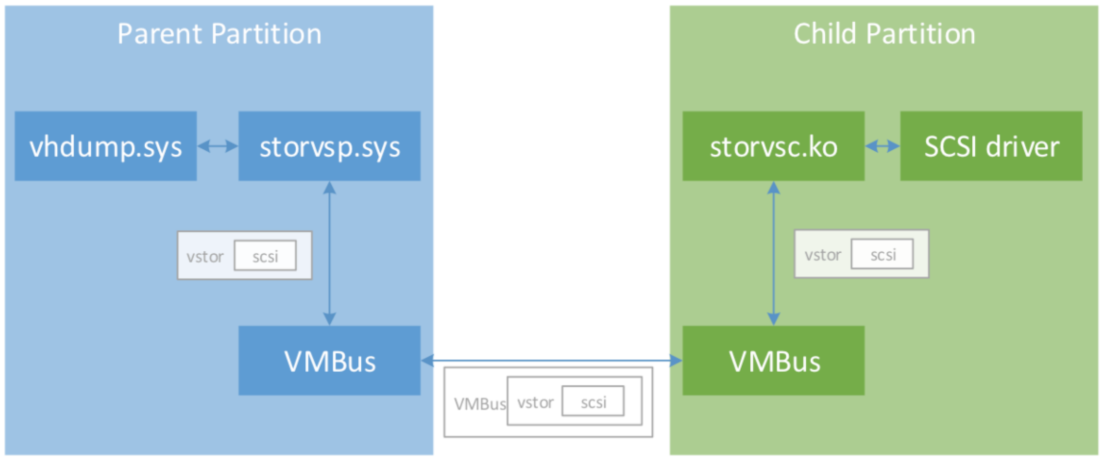


Figure 6: High Level Overview of the synthetic storage device

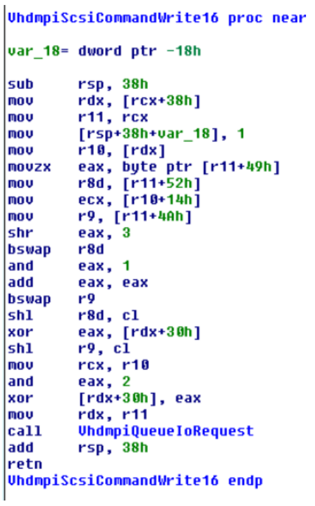
3.2.2 Storage

For our research we first concentrated on the storage VSP. Its design is quite interesting because the VSC sends (amongst other packet types) encapsulated SCSI commands over the VMBus, which are then parsed and transformed into simple file accesses actions by the Storage VSP. Figure 6 highlights the involved components for the implementation of a synthetic storage device on an enlightened Linux guest.

The storage VSC in the child partition encapsulates a SCSI command inside a so-called *vstor* packet. This packet gets encapsulated into a VMBus packet by the VMBus driver and gets transmitted over the aforementioned ring buffer. In the root partition the VMBus driver hands the packet over to the `storvsp.sys` driver, which dispatches it into the corresponding handler function in the `vhdump.sys` driver. Figure 7 shows an example for such an SCSI command handler. Notice the use of the BSWAP instruction, which is used to convert between the big endian SCSI protocol and the little endian system architecture.

In order to assess the security of the storage VSP we again started with basic fuzzing. This naïve approached proved to be infeasible, because fuzzing of vstor and vmbus packets leads to permanent crashes of the storage VSC and therefore to a crash of the whole child partition. To solve this problem, we improved our fuzzing framework to allow fine grained fuzzing of executed SCSI commands: Our fuzzer uses the kprobes[19] interface, which allows the hooking of almost arbitrary kernel functions. By hooking generic VMBus `packet_send` functions and manipulating the passed arguments depending on the callers, we can restrict our fuzzing to data that does not trigger permanent VM crashes. Of course, this reduces the surface that is actively fuzzed and therefore this approach needs to be supplied with manual analysis. f course, this reduces the surface that is actively fuzzed and therefore this approach needs to be supplied with manual analysis.

In the end, we did not discovery any critical vulnerabilities in the storage VSP. However, our analysis was quite limited and due to the ineffectiveness of normal fuzzing manual analysis is definitely required.



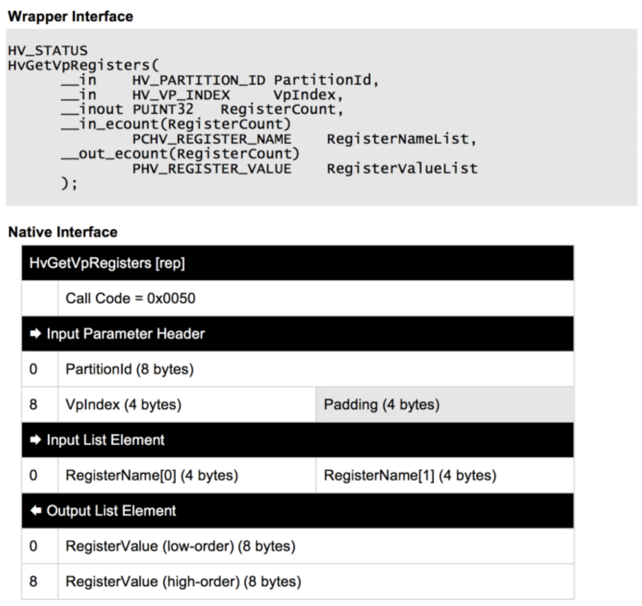
```
VhdmpiScsiCommandWrite16 proc near

var_18= dword ptr -18h

sub     rsp, 38h
mov     rdx, [rcx+38h]
mov     r11, rcx
mov     [rsp+38h+var_18], 1
mov     r10, [rdx]
movzx   eax, byte ptr [r11+49h]
mov     r8d, [r11+52h]
mov     ecx, [r10+14h]
mov     r9, [r11+4Ah]
shr     eax, 3
bswap   r8d
and     eax, 1
add     eax, eax
bswap   r9
shl     r8d, cl
xor     eax, [rdx+30h]
shl     r9, cl
mov     rcx, r10
and     eax, 2
xor     [rdx+30h], eax
mov     rdx, r11
call    VhdmpiQueueIoRequest
add     rsp, 38h
retn
VhdmpiScsiCommandWrite16 endp
```

Figure 7: SCSI Commana Handler

[19] *https://www.kernel.org/doc/Documentation/kprobes.txt*

4 HYPERCALL API

The last component we want to discuss in this paper is the Hypercall API. Hypercalls in Hyper-V are like system calls but operate between the guest kernel and the hypervisor. They are used by enlightened partitions to enhance performance and to enable advanced functionality like the VMBus. In addition, the root partition manages all other partitions using administrative hypercalls. The API itself is very powerful and includes the ability to create and destroy new partitions or to set and read register values of different VMs. Of course such functionality is protected by permission checks and normally restricted to the root partition.



Wrapper Interface

```
HV_STATUS
HvGetVpRegisters(
      __in     HV_PARTITION_ID PartitionId,
      __in     HV_VP_INDEX     VpIndex,
      __inout  PUINT32   RegisterCount,
      __in_ecount(RegisterCount)
              PCHV_REGISTER_NAME     RegisterNameList,
      __out_ecount(RegisterCount)
              PHV_REGISTER_VALUE     RegisterValueList
      );
```

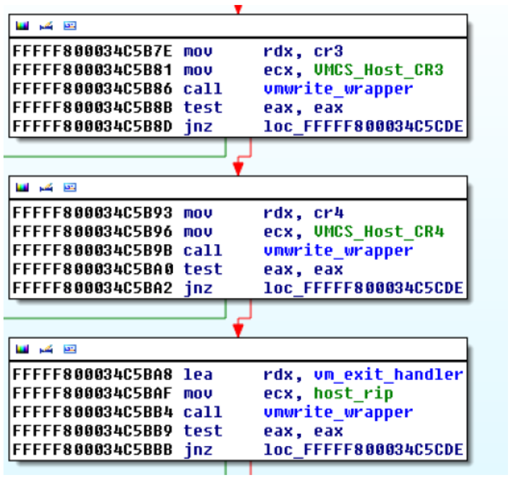
Native Interface

HvGetVpRegisters [rep]		
	Call Code = 0x0050	
➡ Input Parameter Header		
0	PartitionId (8 bytes)	
8	VpIndex (4 bytes)	Padding (4 bytes)
➡ Input List Element		
0	RegisterName[0] (4 bytes)	RegisterName[1] (4 bytes)
⬅ Output List Element		
0	RegisterValue (low-order) (8 bytes)	
8	RegisterValue (high-order) (8 bytes)	

Figure 8: Documentation for HvGetVPRegisters

In contrast to the VMBus and its synthetic devices, the Hypercall API is fully documented by Microsoft in the Hypervisor Top Level Functional Specification. For example, Figure 8 shows the interface description of the `HvGetVPRegisters` hypercall that allows read access to the registers of a virtual machine. All other available hypercalls are documented in the same format.

In order to audit the hypercalls we first have to identify the corresponding handler functions. Using the approach outlined in Section 2.7.2, we can quickly identify the main VM exit handler by searching for the corresponding VMWRITE instruction as shown in Figure 9.



```
FFFFF800034C5B7E mov     rdx, cr3
FFFFF800034C5B81 mov     ecx, VMCS_Host_CR3
FFFFF800034C5B86 call    vmwrite_wrapper
FFFFF800034C5B8B test    eax, eax
FFFFF800034C5B8D jnz     loc_FFFFF800034C5CDE

FFFFF800034C5B93 mov     rdx, cr4
FFFFF800034C5B96 mov     ecx, VMCS_Host_CR4
FFFFF800034C5B9B call    vmwrite_wrapper
FFFFF800034C5BA0 test    eax, eax
FFFFF800034C5BA2 jnz     loc_FFFFF800034C5CDE

FFFFF800034C5BA8 lea     rdx, vm_exit_handler
FFFFF800034C5BAF mov     ecx, host_rip
FFFFF800034C5BB4 call    vmwrite_wrapper
FFFFF800034C5BB9 test    eax, eax
FFFFF800034C5BBB jnz     loc_FFFFF800034C5CDE
```

Figure 9: vmwrite to host_rip VMCS field

The VM exit handlers of all hypervisors follow the same basic structure:

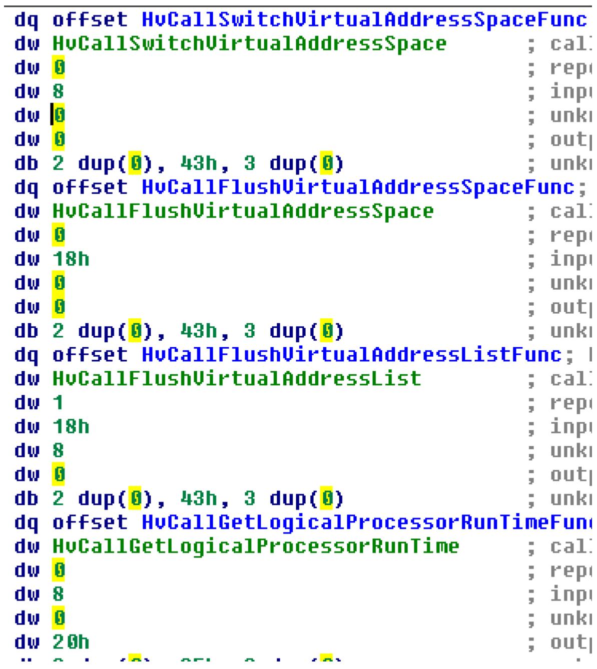
- Store Guest State
- Dispatch to different handler functions depending on the exit reason
- Restore Guest State

The interface for basic hypercalls works like this:

- The call number is stored in RCX,
- the guest physical address pointing to the input is stored in RDX,
- and the GPA pointing to a writable memory region for the output is stored in R8.

Finally, the VMCALL instruction is executed to trigger a VM exit. By identifying the functions that are called when the VMCS exit reason equals VMCALL, we can therefore identify the code responsible for handling hypercalls.

By analyzing this function, we can quickly identify the main data structure for handling hypercalls, which is displayed in Figure 10.



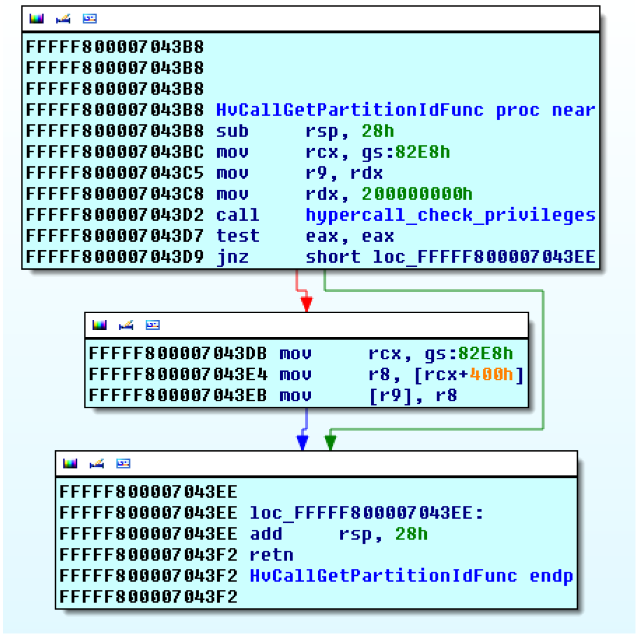
```
dq offset HvCallSwitchVirtualAddressSpaceFunc
dw HvCallSwitchVirtualAddressSpace      ; cal
dw 0                                    ; rep
dw 8                                    ; inp
dw 0                                    ; unk
dw 0                                    ; out
db 2 dup(0), 43h, 3 dup(0)              ; unk
dq offset HvCallFlushVirtualAddressSpaceFunc;
dw HvCallFlushVirtualAddressSpace       ; cal
dw 0                                    ; rep
dw 18h                                  ; inp
dw 0                                    ; unk
dw 0                                    ; out
db 2 dup(0), 43h, 3 dup(0)              ; unk
dq offset HvCallFlushVirtualAddressListFunc;
dw HvCallFlushVirtualAddressList        ; cal
dw 1                                    ; rep
dw 18h                                  ; inp
dw 8                                    ; unk
dw 0                                    ; out
db 2 dup(0), 43h, 3 dup(0)              ; unk
dq offset HvCallGetLogicalProcessorRunTimeFun
dw HvCallGetLogicalProcessorRunTime     ; cal
dw 0                                    ; rep
dw 8                                    ; inp
dw 0                                    ; unk
dw 20h                                  ; out
```

Figure 10: Hypercall Handler Table

Using this data structure, allows a quick identification of all relevant handler functions. When auditing single hypercall handlers for vulnerabilities, we have several advantages: First, the handler functions itself are relatively isolated. They have a single purpose, which makes it easer to identify the functionality of called functions. Second, because the input and output interface are documented by the Top Level Specification, reasoning about the actual code flow is relatively easy.

Figure 11 shows the handler function for the `HvGetPartitionId` hypercall. At the start of each handler function, the RCX register points to the input location, while RDX points to the output. Both memory regions do not directly map into the VM memory. Instead, the input is copied before starting the handler and the output is written to VM memory after the handler finishes. This removes the possibility to manipulate the hypercall input while the handler is actively running and therefore removes a whole vulnerability class.

When actually auditing the hypercall handlers itself for vulnerabilities, we can quickly identify one big issue: Most interesting functionality is only available to the root partition and is guarded by strict and early permission checks. However, there are a lot of checks performed before the actual hypercall handler executes. When we looked at them we quickly identified one critical issue that resulted in MS13-092.



```
FFFFF800007043B8
FFFFF800007043B8
FFFFF800007043B8
FFFFF800007043B8 HvCallGetPartitionIdFunc proc near
FFFFF800007043B8 sub     rsp, 28h
FFFFF800007043BC mov     rcx, gs:82E8h
FFFFF800007043C5 mov     r9, rdx
FFFFF800007043C8 mov     rdx, 200000000h
FFFFF800007043D2 call    hypercall_check_privileges
FFFFF800007043D7 test    eax, eax
FFFFF800007043D9 jnz     short loc_FFFFF800007043EE
```

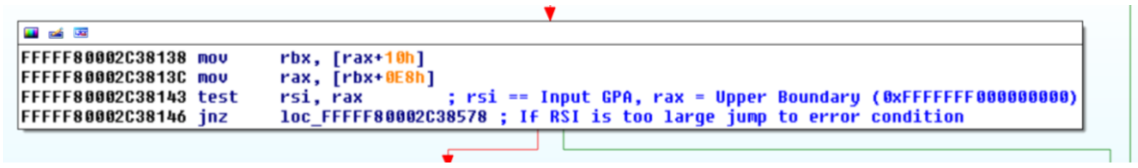
```
FFFFF800007043DB mov     rcx, gs:82E8h
FFFFF800007043E4 mov     r8, [rcx+400h]
FFFFF800007043EB mov     [r9], r8
```

```
FFFFF800007043EE
FFFFF800007043EE loc_FFFFF800007043EE:
FFFFF800007043EE add     rsp, 28h
FFFFF800007043F2 retn
FFFFF800007043F2 HvCallGetPartitionIdFunc endp
FFFFF800007043F2
```

Figure 11 HvGetPartitionId handler

5 MS13-092

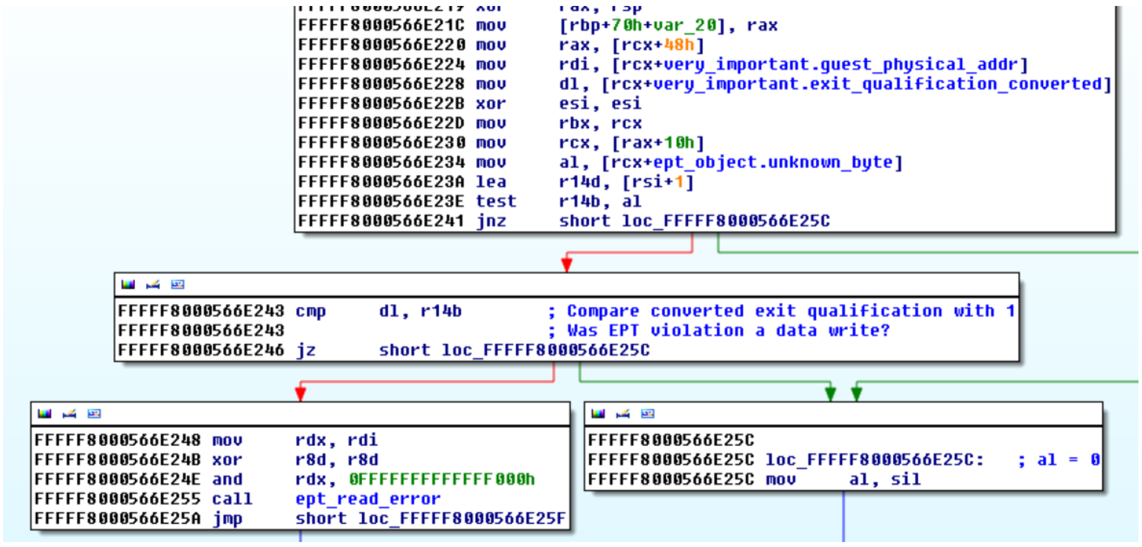
Before calling the actual hypercall handler, multiple sanity checks are performed. For example, it has to be ensured that the hypercall actually comes from ring 0 of the VM. In addition sanity checks of the input and output GPA are performed. Are they correctly aligned? Do they look "sane"?



```
FFFFF80002C38138 mov     rbx, [rax+10h]
FFFFF80002C3813C mov     rax, [rbx+0E8h]
FFFFF80002C38143 test    rsi, rax          ; rsi == Input GPA, rax = Upper Boundary (0xFFFFFFF000000000)
FFFFF80002C38146 jnz     loc_FFFFF80002C38578 ; If RSI is too large jump to error condition
```

Figure 12 Check on Input GPA

One of these checks is shown in Figure 12: The input GPA is stored in the RSI register and is checked against a bitmap. If the input GPA is too large an error condition is raised. The vulnerability lies deeper down the call stack in one of the error handler functions:

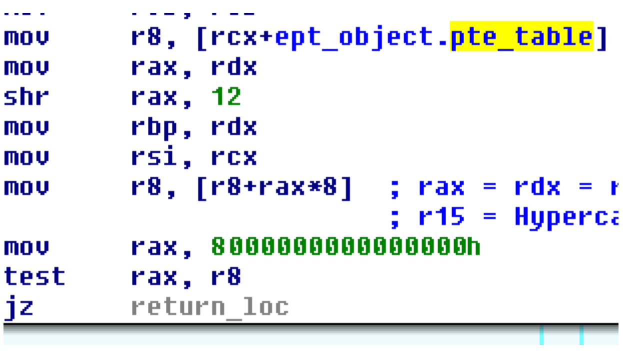


```
FFFFF8000566E21C mov     [rbp+70h+var_20], rax
FFFFF8000566E220 mov     rax, [rcx+48h]
FFFFF8000566E224 mov     rdi, [rcx+very_important.guest_physical_addr]
FFFFF8000566E228 mov     dl, [rcx+very_important.exit_qualification_converted]
FFFFF8000566E22B xor     esi, esi
FFFFF8000566E22D mov     rbx, rcx
FFFFF8000566E230 mov     rcx, [rax+10h]
FFFFF8000566E234 mov     al, [rcx+ept_object.unknown_byte]
FFFFF8000566E23A lea     r14d, [rsi+1]
FFFFF8000566E23E test    r14b, al
FFFFF8000566E241 jnz     short loc_FFFFF8000566E25C

FFFFF8000566E243 cmp     dl, r14b          ; Compare converted exit qualification with 1
FFFFF8000566E243                           ; Was EPT violation a data write?
FFFFF8000566E246 jz      short loc_FFFFF8000566E25C

FFFFF8000566E248 mov     rdx, rdi
FFFFF8000566E24B xor     r8d, r8d
FFFFF8000566E24E and     rdx, 0FFFFFFFFFFFFF000h
FFFFF8000566E255 call    ept_read_error
FFFFF8000566E25A jmp     short loc_FFFFF8000566E25F

FFFFF8000566E25C
FFFFF8000566E25C loc_FFFFF8000566E25C:     ; al = 0
FFFFF8000566E25C mov     al, sil
```

Figure 13: EPT Error Handler

Figure 13 shows a function responsible for the handling of EPT[20] errors. It is called when a normal EPT violation is triggered by a VM but also when the aforementioned error condition is set.

[20] *Extended Page Table*



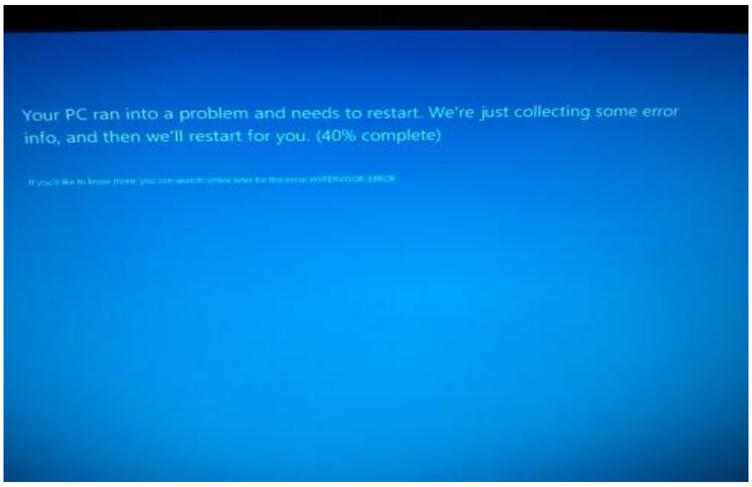
```
mov     r8, [rcx+ept_object.pte_table]
mov     rax, rdx
shr     rax, 12
mov     rbp, rdx
mov     rsi, rcx
mov     r8, [r8+rax*8]   ; rax = rdx = r
                         ; r15 = Hyperca
mov     rax, 8000000000000000h
test    rax, r8
jz      return_loc
```

Figure 14: Out-of-Bound Array Access

If the EPT violation was triggered by a memory read, a function we named `ept_read_error` is called. This is also the case for the code path executing after an invalid input GPA was supplied as a hypercall argument. At the beginning of the function, RDX contains the attacker controlled input GPA. RDX is then copied to RAX, shifted by 12 bits to the right. This value is than used as an index into the page table, containing page table entries (PTEs), of the VM.

The purpose of the page table is to map guest physical pages to system page table entries. There is exactly one such table for each VM and interestingly they are allocated in a constant offset to each other.

Because the input GPA can be almost arbitrarily large[21] the array access can be used to access memory out of bounds of the page table. Crashing the hypervisor is now really easy by triggering an invalid memory access. Any valid hypercall number with the input GPA set to 0x4141414141 will suffice and result in a Hyper-V blue screen and a Denial-of-Service (DoS) situation for the complete hypervisor, of course including all guests operated on it:



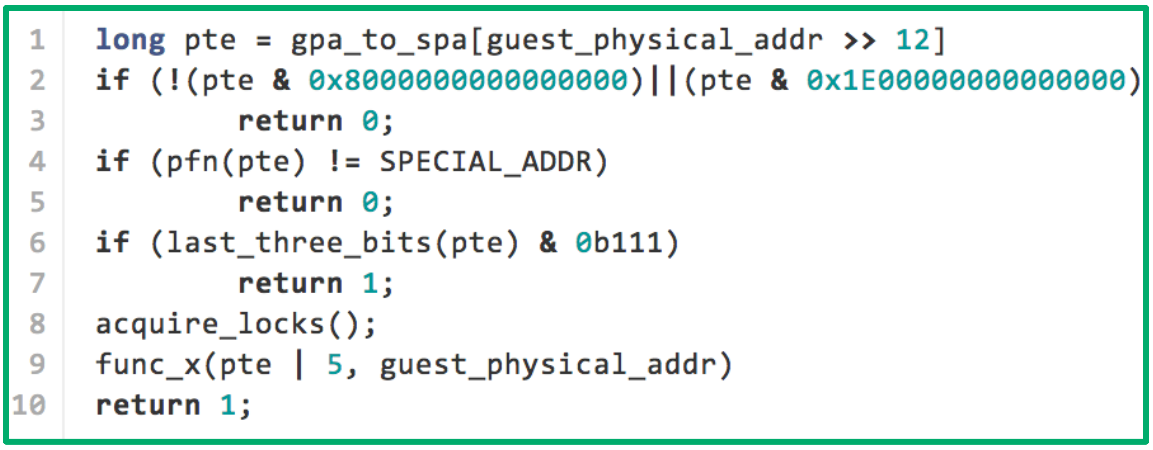
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (40% complete)

[illegible]

Figure 15 Hyper-V Blue-Screen-of-Death

We reported this bug to Microsoft and it was patched in November 2013 in bulletin MS13-092[22]. Interestingly, Microsoft rates this bug as a potential privilege escalation between different partitions. So we decided to dig a bit deeper how this bug can be turned from a DoS to a PrivEsc.

[21] *There is a upper limit of 0x1000000000000*
[22] *https://technet.microsoft.com/en-us/library/security/ms13-092.aspx*



```
long pte = gpa_to_spa[guest_physical_addr >> 12]
if (!(pte & 0x8000000000000000)||(pte & 0x1E00000000000000)
        return 0;
if (pfn(pte) != SPECIAL_ADDR)
        return 0;
if (last_three_bits(pte) & 0b111)
        return 1;
acquire_locks();
func_x(pte | 5, guest_physical_addr)
return 1;
```

Figure 16: Simplified Pseudo-Code

Figure 16 shows a simplified version of the vulnerable function. It will simply return 0 in most cases and only performs relevant operations when the page frame number (PFN) in the PTE equals a certain value.

Analyzing the patch provided by Microsoft reveals as the only obvious change an additional size check in front of the function that force returns 0 when it fails. This means that all code paths executing after the vulnerable function returns 0 are not interesting for the exploitation of the vulnerability.

In conclusion, we have to find a way to read a PTE (from a different VM) that contains the "special" PFN number in order to trigger potentially malicious memory access. Using purely static analysis, we were not able to identify the use case of this PFN and it was not used by the VMs in our lab. This probably means that it is only used in certain circumstances that are possibly configuration dependent. Trying to read a completely attacker controlled value fails due to the large size of the 64bit address space and the aforementioned upper limit on the input GPA value.

If an attacker is able to pass this check, he can reach interesting code. However, the only two values he can influence are the PTE and the input GPA. Those are quite restricted due to the mentioned PFN checks, so he can only minimally influence the next execution steps.

While it is definitely not possible to turn the read violation in some kind of writing memory corruption, the possibility to map a PTE from one VM to another would be a problem as well. While we are positive that there is no trivial way to exploit this specific issue, we will analyze the patch and resulting behavior further and encourage other researches to do so as well.

6 Further Research and Conclusion

This paper summarized our research on Hyper-V security. While most of the time was spent gaining a detailed understanding of the architecture and mapping the attack surface for VM breakout attacks, we also did discover a critical vulnerability in the handling of hypercalls.

Based on this work, we plan to perform a more detailed analysis of other VSPs and develop techniques to improve our fuzzing capabilities (mainly improving crash recovery/reducing the overall amount/need of crashes of the guest machine). In addition, analyzing the different Hyper-V versions for silently patched vulnerabilities seems to be a promising activity as we plan to demonstrate later this year.

Our research shows that hypervisors are large and complex software with a significant attack surface. Even if the term "Virtual Air Gap" is quite popular nowadays, our research shows that this gap is much smaller than a physical one. While Hyper-V is solid software and was developed with security in mind, it still suffers from critical security vulnerabilities. This is supposed to motivate other researchers as well to use our results and step in on analyzing the huge attack surface of Hyper-V, following the old hacker spirit *Make the Theoretical Practical*!

7 APPENDIX: DISCLAIMER

All products, company names, brand names, trademarks and logos are the property of their respective owners.