Hyntrospect: a fuzzer for Hyper-V devices

Diane Dubois
didu@google.com

Google

Abstract. Hypervisors are complex software which may require the reimplementation of legacy stacks. On Microsoft Hyper-V virtual machines (generation 1), some devices are emulated in the userland of its root partition. To explore this attack surface, a specifically crafted open source toolchain called Hyntrospect has been developed. It aims at helping find vulnerabilities in a pragmatic way: by taking benefits of existing Hyper-V and Windows capabilities and tools to perform coverage-guided fuzzing on Hyper-V closed-source binaries. That approach was inspired by previous experiences with libFuzzer, a publication by Microsoft on their fuzzing campaign, and other research conducted on the topic. The specificity of that tool is to rely on debugging and as a consequence to run in a real environment. It was also written in the perspective of putting together techniques that could be ported in the future to other Hyper-V root partition's userland targets.

After covering Hyper-V and the state of the art on its instrumentation and research, this paper introduces Hyntrospect and its associated design choices, and finally describes the outcome of the first runs and future endeavors.

1 Introduction

Hyper-V is the hypervisor developed by Microsoft which runs Microsoft Cloud Azure. Modern versions of Windows also run on top of Hyper-V to enhance their virtualization-based security [36]. Defeating Hyper-V security could lead to compromising local security policies, servers or exposing customer data. Hyper-V is, as a consequence, a complex but also interesting target for vulnerability researchers. A bug bounty is offered by Microsoft, the rewards go up to `$250,000`.

On Hyper-V, the virtualization stack is mostly implemented in a virtual machine called the root partition which has a different status than the other virtual machines, and a good chunk of it "lives" in userland such as the worker process which holds the VMs (cf first steps in Hyper-V research and Hyper-V architecture and vulnerabilities [29, 41]). Having execution in the root partition enables controlling the virtualization stack. As a consequence, a "ret to the root partition" is an interesting target for

https://t.me/learningnets

a bug hunter. Such an approach already enabled Joe Bialek to find a bug in the IDE bus stack (CVE-2018-0959, BlackHat USA 2019) [11].

The peripherals of a virtual machine, such as the network interface card, can be handled in different ways. One way is to mimic a real controller in dedicated code which enables keeping the guest VM unaware of the virtualization. This is called emulation. The implementation of those controllers is common and not straight-forward, and has proven to be an area prone to bugs (on Hyper-V: CVE-2018-0888, CVE-2018-0959, but also in other hypervisors). On Hyper-V, the emulated devices' controllers are implemented in the root partition. They offer a great target that can be reached from the VM. This technical stack has already been covered in this blog post published by MSRC [42].

The goal of this paper is not to cover that topic again but instead to cover how the following challenge was resolved: how to instrument Hyper-V to test that attack surface? Or in more detail: how to do coverage-guided fuzzing on components of the userland of the root partition of Hyper-V VMs without having access to the source code? After introducing some architectural concepts and some previous work done in the area, the strategy deployed for this project and resulting tool will be presented here. Finally, some results will be presented. The goal of open-sourcing the fuzzer is rather to share and get contributions from the security community as - with some adaptations - the use of the fuzzer can be broadened.

This project was done during my 20% project (an initiative where Google employees can spend 20% of their time on a project of their choosing) with the Project Zero team.

2 Hyper-V and the state of the art

2.1 What is Hyper-V?

Hypervisors are pieces of software that enable several operating systems (the virtual machines) to run concurrently on the same machine (the host) and manage the host's resources in a process called virtualization. Examples of uses for hypervisors are to maximize the usage of resources or to containerize.

There are 2 types of hypervisors. Type 1 (or bare-metal) hypervisors lie between the hardware layer and the operating systems, like Xen or VMWare ESXi. Type 2 hypervisors are embedded within one operating system to run other operating systems on top of that operating system like any other program, such as VMWare Workstation or Oracle VirtualBox. Hyper-V is a type 1 hypervisor.

https://t.me/learningnets

Hyper-V is Microsoft's hypervisor. The Microsoft Cloud called Azure runs Hyper-V. Modern versions of Windows also run on top of Hyper-V as it enables enhancing their level of security through virtualization-based security [36].

3 Overview of Hyper-V architecture

Figure 1 summarizes Hyper-V architecture (source: Microsoft documents [37]).

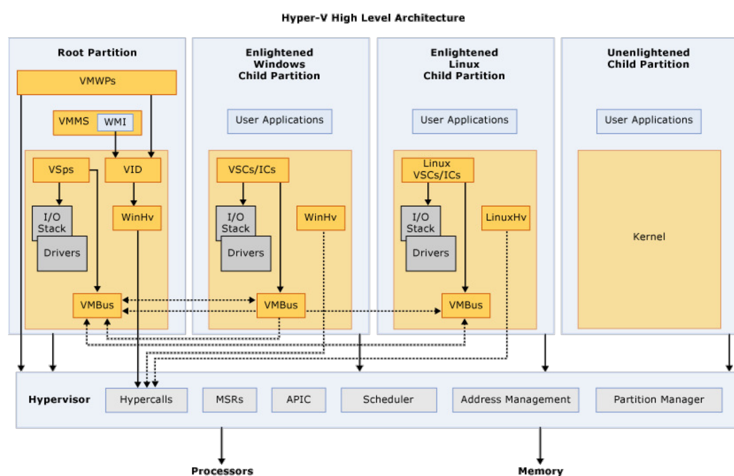

Fig. 1. Hyper-V architecture overview

Microsoft published a post about Hyper-V architecture [37]. The key components for this analysis are the root partition and the VM worker process, which will be introduced here.

The **root partition** is privileged compared to the other partitions and is sometimes referred to as the parent partition. It is the only partition that has direct access to physical memory and devices. VMs on Azure or VMs started on a Windows machine inside Hyper-V Manager are **children partitions**. They communicate with the root partition either through the VMBus (logical communication channel) or through the hypervisor.

One **VM worker process (vmwp)** is spawned per child partition in the root partition's userland. It provides the virtual machine management services.

In this paper, "guest", "VM" and "virtual machine" will refer interchangeably to "virtual machine". Host is more ambiguous on Hyper-V as

https://t.me/learningnets

it could refer to either the hypervisor or the root partition (as a matter of interpretation) so this term will be avoided unless it refers to both these surfaces together.

3.1 VM generations

Hyper-V has 2 different generations of VMs: generation 1 (with more legacy components, leaning towards compatibility) and generation 2 (next generation, with a focus on performance and newer technologies). In this post [32], Microsoft presents the differences between generation 1 and generation 2 VMs. Among the numerous differences, the following ones can be called out: UEFI versus BIOS, the introduction of SecureBoot enabled by default, booting from SCSI instead of IDE, and the removal of all the legacy emulated controllers.

3.2 Hyper-V attack surface

In the Microsoft MSRC blog post "First steps in Hyper-V research" [41], the attack surface for **guest-to-host escape** is detailed as follows: the hypercalls handlers, the faults (triple fault, EPT page faults, etc.), the instruction emulation, the intercepts, and register access (control registers, MSRs).

This is also a topic that was presented by Alisa Esage (slides 41 and 42 of her presentation on hypervisor security research [21]).

Microsoft has decided to push several components out of the hypervisor layer to the root partition in order to reduce the attack surface and complexity of the hypervisor layer. In the root partition, some components such as the memory management run in the kernel, and some such as the devices have been developed in userland. As a consequence, a good portion of the attack surface such as virtualization code and VM memory are in the root partition which makes it an interesting target. Across this large attack surface, the emulated devices were chosen for this analysis.

3.3 The target: the emulated devices on generation 1 VMs

What is an emulated device? In a non virtualized environment, the operating system can have direct access to the hardware. Virtualization adds complexity as several operating systems can run in parallel, need access to resources concurrently, and in the meantime should not be granted the same level of privilege as the "master operating system" (the hypervisor) for security reasons as well as stability. The hypervisor has

https://t.me/learningnets

to orchestrate and check the access to resources and operations on the devices such as a disk write operation or sending packets over the network. On virtualized platforms, there are three ways to deal with the access to those devices: emulation, paravirtualization, and pass-through.

Emulation is a technique that consists of imitating the behavior of a real hardware controller through software within the hypervisor's code. The operating system of the virtual machine runs unmodified. With paravirtualization, the hypervisor exposes a modified version of the physical hardware interface to the guest VM. Its operating system is as a consequence modified, with the benefits of enhanced performance. Pass-through gives direct access to the real hardware underneath.

Why this target? As emulation consists of rewriting the controllers' logic at the hypervisor level, implementation is not straight-forward. It has proven in the past to be an area prone to bugs (on Hyper-V: CVE-2018-0888 [6], CVE-2018-0959 [7]; but also in other hypervisors, for example QEMU with CVE-2015-3456 [5]).

Hyper-V supports emulated devices. They are only implemented on generation 1 VMs in the userland of the root partition. Several of them are legacy devices. They are implemented in C++ which is a memory unsafe language. Gaining code execution from a VM would mean executing code in the root partition which is an interesting target. Even if this area has already been investigated by Microsoft and very likely by other researchers, it looked like an interesting entry point.

Only the implementation for Windows VMs and for machines running Intel CPUs will be covered in this analysis.

How are they implemented? The clients located in the guest and the controllers located in the root partition need to communicate. This is done through the use of IO ports and MMIOs. Specifically for the IO ports, the virtual machines send Intel's IN and OUT instructions in assembly from privileged code. The handling of such instructions are deferred to the worker process on the root partition (`C:\Windows\vmwp.exe`) which transfers it to the relevant controller implemented in DLLs loaded by vmwp.exe such as `C:\Windows\System32\VmEmulatedStorage.dll` or `C:\Windows\System32\vmemulateddevices.dll` (more details on the whole stack in this blog post [42]). The controllers consume these requests in dedicated functions: NotifyIoPortRead for IN operations and NotifyIoPortWrite for OUT operations, which are implemented for each emulated device. For example:

https://t.me/learningnets

`vmemulatedstorage!IdeControllerDevice::NotifyIoPortWrite`, `vmemulatedstorage!FloppyControllerDevice::NotifyIoPortRead`. When analyzing the emulated devices, these are the entry points that can be looked at.

Reversing the binaries enabled understanding these entry points. NotifyIoPortRead takes 3 arguments: the device ("this" in C++, in register rcx), the IO port (uint16, stored in register dx) and the access size (uint16, which should be equal to 1, 2 or 4, stored in register r8). NotifyIoPortWrite takes 4 arguments: the same arguments as NotifyIOPortRead plus the value (uint stored in register r9).

Figure 2 is a snapshot of the first block of `FloppyControllerDevice::NotifyIoPortWrite`, in `vmemulatedstorage.dll`.

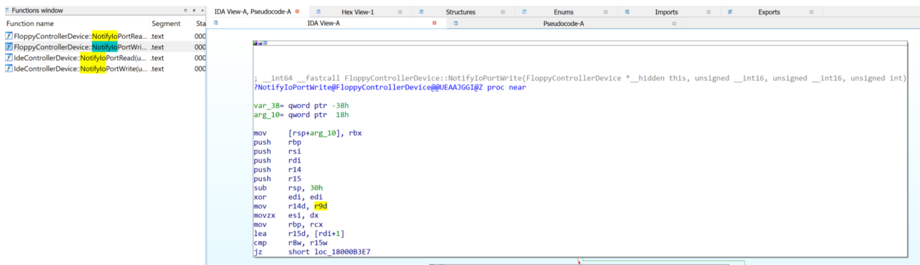

Fig. 2. IDA screenshot of NotifyIOPortWrite

Those functions were key in the pre-analysis that was necessary to fuzz each device.

3.4 Hyper-V tools and features

This section introduces the key features that are mentioned later in this paper.

Debugging Windows Hyper-V is a component of Windows, which offers some native debugging features. WinDbg is the API published by Microsoft, which relies underneath on a debugger engine [34]. Those debugging capabilities apply to Hyper-V: the hypervisor, the root partition kernel and the worker process can be debugged. DbgShell [31] is an open source project written by Microsoft which offers a PowerShell front-end for the Windows debugger engine. It only applies to userland binaries.

https://t.me/learningnets

Availability of the symbols Hyper-V and Windows code is not available but the analysis can be done by reversing the binaries using the symbols. Those symbols can also be used when debugging.

The symbols of the root partition (such as `C:\Windows\System32\vmwp.exe` and related DLLs like `C:\Windows\System32\vmemulateddevices.dll`) are provided by Microsoft. The symbols of the hypervisor layer are not available.

More details can be found on this page [38].

Hyper-V user capabilities Windows offers a wide range of capabilities to manage the Hyper-V server and the VMs [39]. Some of these capabilities will be presented here as they will be mentioned later.

Hyper-V management through scripts Hyper-V can be accessed and managed via a UI but it can also be managed through PowerShell scripts. In other words, there is a Windows feature called "Hyper-V Management Tools" which has 2 subcomponents: "Hyper-V GUI Management Tools" and "Hyper-V Module for Windows PowerShell". Some additional "VMIntegrationService" capabilities can also be enabled on top of that to add options such as the ability to copy a file from the host to the VM.

Powershell direct Among the management capabilities offered by scripting, PowerShell direct [33] enables sending commands from the root partition to the guest through authenticated sessions. It relies on the Hyper-V VMBus.

Snapshot Another interesting feature provided by the hypervisor is the ability to take snapshots (in Hyper-V words "checkpoints") of the VM and revert it to a stable state.

State of the art on Hyper-V vulnerability research

Vulnerability research techniques There are several ways to approach vulnerability research, the main techniques being either analyzing the source code or assemblies, or instrumenting the target for dynamic analysis or fuzzing. One way to instrument it is by writing a program to inject user controlled input and test the robustness and correctness of the target. That program is called a fuzzer.

Multiple implementations and concepts have already been developed. One interesting technique is coverage-guided fuzzing: it consists in monitoring the coverage of the target and updating the fuzzer input based

https://t.me/learningnets

on that feedback loop. The goal is to reach more paths or focus on the paths that are not often taken. This technique has been proved to substantially enhance the performance of the fuzzer [26]. It is implemented in libFuzzer [30] for example.

Several well known fuzzers or binary instrumentations already exist for Windows binaries as black boxes (this list is not exhaustive):

- WinAFL [22], a variant of AFL, which can be associated to DynamoRIO to perform coverage analysis [23]
- Intel Pin [28]
- Intel Processor Tracing ("Intel PT") [27]
- Jackalope [24]
- QDBI for Windows [43]
- Mesos [16]

Microsoft presented on their fuzzer [15]. It is coverage-guided. The major difference is their access to sources.

Publications by MSRC Microsoft MSRC is a major actor for Hyper-V vulnerability research.

They made available blog posts to help vulnerability researchers get started on Hyper-V:

- First steps in Hyper-V research [41]: `https://msrc-blog.microsoft.com/2018/12/10/first-steps-in-hyper-v-research/`;
- Attacking the VM worker process [42]: `https://msrc-blog.microsoft.com/2019/09/11/attacking-the-vm-worker-process/`.

They also presented at conferences:

- A Dive in to Hyper-V Architecture and Vulnerabilities at BlackHat USA 2018 [29];
- Exploiting the Hyper-V IDE Emulator to Escape the Virtual Machine at BlackHat USA 2019 [11];
- Breaking VSM by Attacking SecureKernel at BlackHat USA 2020 [8].

Previous external work and publications on Hyper-V The security community has also published on Hyper-V:

- `@gerhart_x` is an active contributor. He posts content and resources on his dedicated blog [17, 18].
- Jordan Rabet (Microsoft OSR) presented Hardening Hyper-V through offensive security research [20]

https://t.me/learningnets

— Alisa Esage is also active in the area, as reflected on her analysis of the hypervisors' attack surface and state of the art [21].
— Damien Aumaitre (Quarkslab) published a tool based on Windows Hypervisor Platform [10] and presented it at SSTIC 2020 [9].

Based on the analysis of the state-of-the-art, the next section will cover the motivations for a new approach.

4 Hyntrospect fuzzer

The source code is located in this repository [12]:
`https://github.com/googleprojectzero/Hyntrospect`.

4.1 So why another toolchain?

The goal was to reproduce a similar structure as Microsoft fuzzer [15] from an outsider perspective. The major differences when assessing Hyper-V as an outsider is not having access to the source code. Indeed, having the source code enables recompiling the code with instrumentation for fuzzing such as ASAN for memory error detection. The material in this case is limited to the assembly - which offers less possibilities for coverage-guided fuzzing.

As stated above, several tools already enable black-box fuzzing. The motivation to rewrite a fuzzer came from different factors:

— The target can be either a DLL or an executable, which disqualifies all the fuzzers instrumenting only executables.
— Hyper-V is a complex target. Running a module separately (through emulation) may hide some side effects coming from real runs. Also, emulating only the target functions outside of their context would require providing relevant context for the execution, such as the devices set in a correct state, which would have implied a lot of reverse engineering and development. As a consequence, the strategy of running a full VM and instrumenting the target binary in an environment as close as possible to real runs was preferred. This was done through debugging.
— Another option offered by several existing tools is starting the target binary with the instrumentation set, which would mean starting vmwp.exe which implies starting the VM itself at each iteration. This technique would be slower as booting a VM is time consuming. As a consequence, the fuzzer should attach to a running instance.

https://t.me/learningnets

— The possibility to port the fuzzer to other Hyper-V use cases in the future, as some basics and core structure will already be implemented.
— As the need is a bit specific (injecting IOs in a VM and monitoring some specific APIs), none of the public tools seemed to match the need without heavy modifications. Performing coverage-guided fuzzing on specific input in this environment poses some challenges. As many existing capabilities as possible were reused.
— Managing all the blocks with only one language makes interoperability easier.

As a consequence, it was decided to tailor a fuzzer to this specific need. Some previous work with libFuzzer [30] was a source of inspiration to achieve coverage-guided fuzzing.

4.2 The challenges and high level overview

The main questions to answer were:

— How to instrument a VM? Where to position the fuzzer in Hyper-V architecture?
— How to guide fuzzing with coverage (both structurally and content-wise)?
— How much structure should be introduced to reach more paths?
— What should be the mutation strategy?
— How to trace without too much latency as single-step tracing is slow?
— How to detect memory corruptions?
— How to monitor crashes and reproduce cases?

These questions will be answered in the following sections.

Design choices at a glance The key design choices of the fuzzer are summarized here:

Emulation vs execution	Execution of a VM through a debugger
Coverage	Tracked with the int3 technique described in this blogpost [14]
Memory corruption detection	Pageheap (gflags) [35]
Environment reset	Hyper-V checkpoints
Mutation strategy	Custom

Existing tools and capabilities that were used: IDA [2], DbgShell [31], CHIPSEC [1], LightHouse [25], gflags/pageheap [35], snapshot capability on Hyper-V, PowerShell libraries for Hyper-V including PowerShell direct [33].

https://t.me/learningnets

Overview of the architecture Figure 3 gives an overview of Hyntrospect architecture.

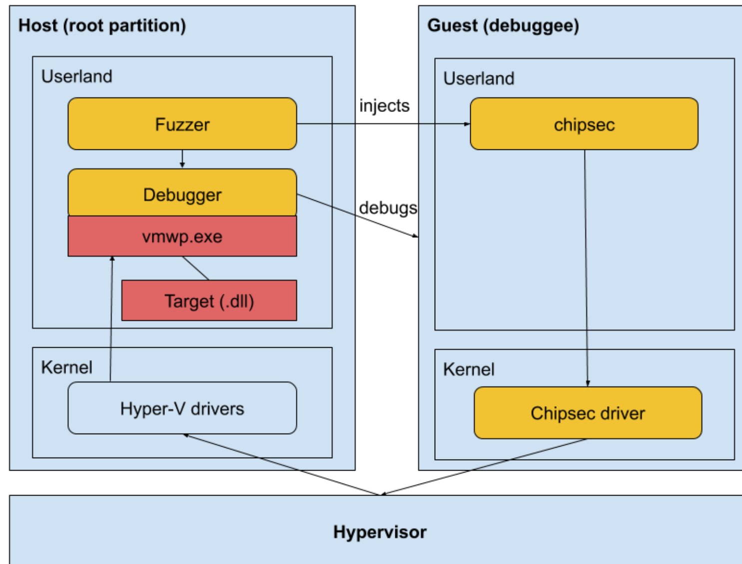

Fig. 3. Overview of Hyntrospect architecture

Workflow There are several goals that need to be dealt with in parallel:

- Instrumenting the VM to trigger the desired action (fuzzer-master)
- Running a debugger attached to the worker process (debugger)
- Monitoring the state (vm-monitoring)
- Adding meaningful input relying on the coverage update (fuzzer-master and input-generator).

The fuzzer master deals with the overall logic and spawns different processes to debug, monitor, and fill the input folder.

The overview of the Hyntrospect's subcomponents is shared in figure 4.

4.3 Design choices in more detail

PowerShell as the implementation language Modern Windows systems are manageable through PowerShell which offers numerous benefits:

https://t.me/learningnets

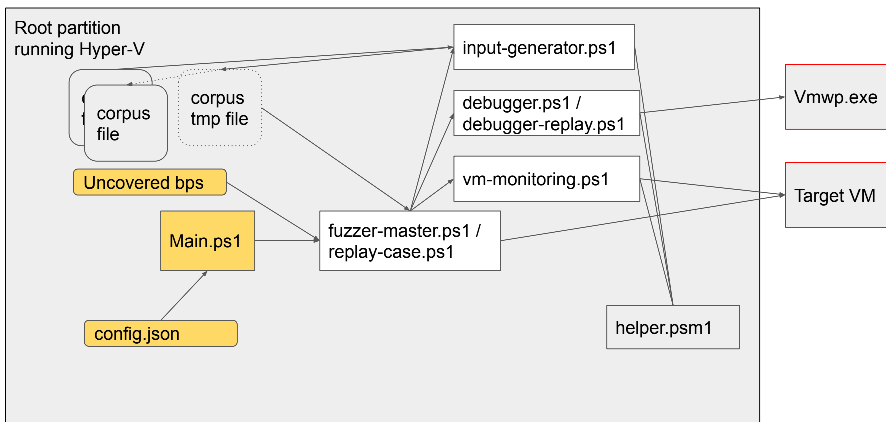

Fig. 4. Hyntrospect workflow

— Hyper-V APIs
— Integration with other Windows components like the event logs
— Direct calls to .NET APIs or even embedding `C#`
— Plug-ins to Windows debugging engine.

This choice comes at the cost of speed.

Work environment

Overview of the working environment The fuzzer has been written to run **elevated in the root partition in userland**: that enables injecting commands in the debuggee VM and monitoring in real time what happens in the userland of the root partition. The pitfalls of such an approach are that the hypervisor layer is not monitored by the fuzzer. It is an accepted risk: if the hypervisor crashes, the case is deemed interesting enough to be worth a full case examination and wasting some cycles (as the machine hence the fuzzer would be down). This can be partially mitigated by running the hypervisor inside another hypervisor, which is called nested virtualization, and by attaching a debugger to it.

Nested virtualization The current implementation of the fuzzer lets it run **either from a nested root partition (level 1) or directly in the root partition of the machine (level 0)**.

https://t.me/learningnets

When nesting Hyper-V, Hyper-V runs on a Windows machine (level 0), which contains the targeted Hyper-V server and root partition (level 1), which runs a debug VM (level 2), as illustrated on figure 5. This infrastructure and debugging method are described by MSRC [41] and `@gerhart_x` [18].

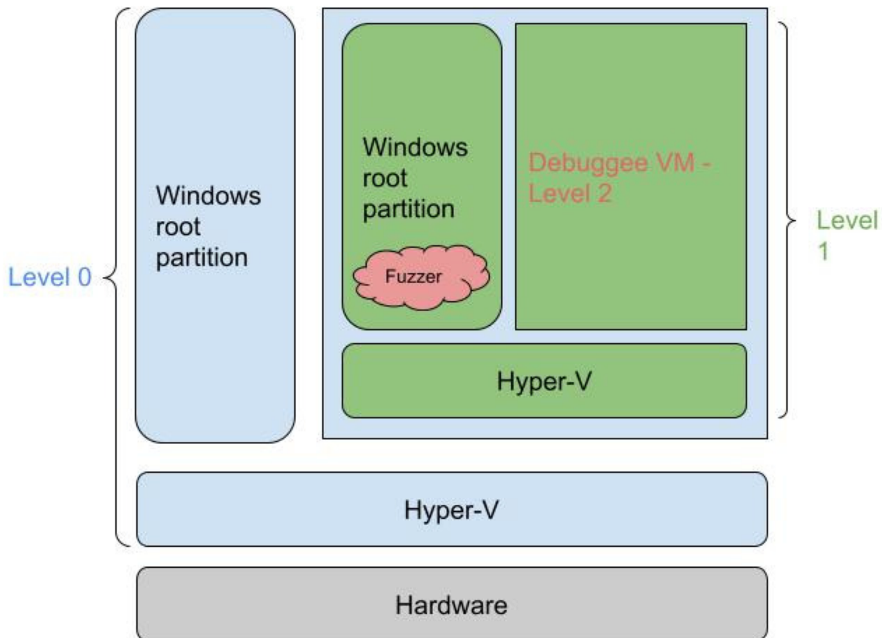

Fig. 5. Nested hypervisor setup

With this representation, the fuzzer runs in level 1 and monitors a level 2 VM.

The **benefits** of nesting the fuzzing environment is:

— In the case of a root partition corruption or hypervisor failure, to be able to take a snapshot of that state thanks to the checkpoint capability, and after the analysis the possibility to restart easily from a clean state without needing to reimage the machine.

— To attach a debugger in the host to the level 1 hypervisor as presented in MSRC blog post - which partially mitigates the fact that the hypervisor layer is not monitored by the fuzzer (but would still require manual investigation in case of a crash).

https://t.me/learningnets

The **drawbacks** of nesting are mostly tied to performance as it adds one more level of indirection, but it could also possibly impact the behavior of the hypervisor (virtualized hypervisor versus running on real hardware).

For the current analysis, the hypervisor **was run nested**.

An entire process orchestration initiated by a fuzzer master The fuzzer master deals with the overall logic and spawns different processes to debug, monitor, and fill the input folder. Each process then follows its own logic and execution. This design raised some challenges:

- Implementing a state machine is needed in case of crashes to halt the execution, but the code is split in between different processes. The solution was to send signals through the creation and the deletion of temporary files.
- Monitoring the state of the VM is not trivial as it may change and resume too fast to be detected by the monitoring process. The solution was to monitor the uptime of the VM: it should only grow after the snapshot is reapplied until the end of each case.

Sending VM commands through PowerShell direct and using CHIPSEC PowerShell direct [33] is used to send IOs directly from the VM and extract data from the VM.

As the communication between the devices and their controllers happens through IO ports, the VM has to simply issue IO ports IN [3] and OUT [4] operations. This needs to be done from the kernel. Reimplementing a specific driver would have been an option. For ease, CHIPSEC [1] drivers and wrappers were used. Indeed, CHIPSEC is a framework for analyzing the security of PC platforms including low level layers which provides APIs to interact with those layers.

By default with PowerShell Direct, the commands are executed in an unelevated shell. As the CHIPSEC IO commands need to be sent from an elevated shell, the default configuration raises issues and a registry key has to be modified to enable elevated execution by default [40].

Reset of the initial state of the VM Starting every new iteration in a clean state is critical. To achieve that, the checkpoint feature of Hyper-V was leveraged. Every single run starts after resetting the image and restarting the instrumentation on it. Time-wise, that choice is costly. Consistency was favored.

This also implies that the user needs to prepare a VM snapshot before running the fuzzer. That VM snapshot requires a certain state described in the README file of Hyntrospect's source repository.

https://t.me/learningnets

Concurrent runs The behavior of a fuzzer needs to be deterministic in order to be reproducible. Fuzzing the same VM with different threads may prevent that. As a consequence, each process created by the fuzzer is single-threaded for now. Multi-threading the fuzzer to have it concurrently deal with several VMs will not be implemented: those multiple VMs would be too heavy for most environments. However, it is possible to start several instances of the fuzzer on the same machine against different VMs.

The target's instrumentation

Fuzzing through the debugger Different approaches have been considered: emulating the code or monitoring existing running code. For the latter, the follow up question is whether a VM should be started on purpose or be monitored while running.

As the virtualization stack is complex, it was chosen to try to get as close as possible to a real execution context and attach to that running environment. This led to an **instrumentation based on debugging**.

Windows offers debugging engines that could be leveraged.

Binaries can act differently when attached to a debugger. This was taken into account but the target binaries do not seem to implement anti-debug features.

By design, as the fuzzer is positioned in userland, this restricts the debugging capabilities to the userland of the root partition. Porting it to kernel monitoring would require a modification of the fuzzer architecture (by adding components to the L0 layer, then injecting into the L1 layer which would then inject into the L2 layer).

DbgShell at the core and technical problems solved The code of the fuzzer relies on DbgShell [31] to instrument the target binaries.

Several difficulties were encountered (only the main ones are listed here):

— Some DbgShell commands are blocking ("g"): it is not possible to take over the debugger execution in an automated way until it reaches a breakpoint. The commands of the script after "g" will only be executed once a breakpoint is reached. The code had to be designed around that constraint. The debugger is a script on its own, launched in a separate process by the fuzzer master which deals with the overall logic and kills the debugger after use. The flow is like a ping pong game between the master which forces the VMs into sending IO commands and the debugger which handles breakpoints and lets the VM go.

https://t.me/learningnets

— DbgShell is not a real shell: not all PowerShell commands can be used. The blockers during the implementation were Start-Job and Get-Credential. These had to be externalized and dealt with by the fuzzer main. Also, Write-Host and Write-Output are not printed at the same time: Write-Host had to be used.
— Launching a script in DbgShell which takes arguments required some experimentation. The arguments cannot be PowerShell objects, only strings that will be interpreted or numerals. Also, DbgShell needs to receive fullpaths.
— Restore-VMSnapshot is blocked (and blocking) when the VM is being debugged. This led to killing the debugger process (after killing the monitoring process) before applying the snapshot.

The approach to perform coverage-guided fuzzing In the first implementation, the trace was recorded for every instruction ("t" on WinDbg). This was extremely slow and the mutations needed to be computed post runs to not slow it down even more. Also, with that first naive approach, the fuzzer created a random file in real time and consumes it for each iteration, with no feedback loop.

On a second implementation, conditional breakpoints were set. This also massively slowed down the flow.

A more refined strategy has then been implemented, which was inspired by Samuel Groß (@5aelo) technique on ImageIO [14], also developed by Brandon Falk (@gamozolabs) for mesos [16]. The current schema is already leveraging a debugger so a shadow mapping of int3 is already "available for free".

The **approach**:

— If the corpus is empty, a **seed** made of a record of **legitimate traffic** for that device is generated.
— If the breakpoint list does not exist yet, **all basic blocks' addresses are precomputed** through IDA [2]. That list can be consumed and updated in the script.
— The fuzzer **master** starts the input generator, consumes the unreached addresses, sets static breakpoints (with Intel assembly instruction "int3") on those, and consumes the oldest temporary file among all temporary files in the corpus directory. Every time the debugger engine reaches one of the int3 instructions, the script:
  — updates the corpus with the input file truncated at the largest offset triggering a breakpoint; it copies that truncated file to

https://t.me/learningnets

  a permanent file in that same corpus, and as a consequence guides the fuzzer towards that new coverage
  - removes the breakpoint from the breakpoints list
  - removes the breakpoint in the debugger and completes the run (in case something interesting comes out of it). When reaching the end of an input file, that file is deleted.
- **Input generator**: the script is started at each iteration (which is less resource intensive than a while loop). It feeds the corpus folder with up to n new samples labelled as tmp files. It applies different strategies to diversify the input.
- **Coverage**: the coverage can be computed by subtracting the unreached breakpoints to the list of blocks addresses that were retrieved from IDA. The coverage is LightHouse [25] compliant. That operation is done offline in a dedicated script.

The input files are a sequence of IO operations. They are made of sequences of bytes that are translated into an operation (read or write), a compatible IO port, length, and IO value (in case of a write operation). The list of acceptable values for each of the parameters is listed in the configuration file. For each of them, the raw byte of the input file is read, reduced modulo the number of possible parameter values and used as an index to get the value. For example, if [0x60, 0x61, 0x62] are the possible values for the IO ports, and 5 is read, it will point to the index 5 % 3 = 2, which is port 0x62. When reaching the end of the input file, the current input is padded with zeros to cover all the arguments needed.

After each coverage increase, the file is truncated to the offset following the operation responsible for the increase and added to the corpus file. This means that the new corpus file is a sequence of IO operations which increased the coverage and which is used as a base for future file generation.

Pre-generation of block's addresses This point raises one design choice: edge coverage (monitoring the different possible paths in the binary) versus block coverage (independently of the caller). For instance, AFL checks information about the origin and the destination (edge) in the flow graphs (as expressed in this whitepaper [19]).

The problem of only breaking on the beginning of blocks is to get there through a given edge (certain IO), remove the breakpoint, expand the corpus, and ignore edges to that address from different origins (which would have been interesting to add to the corpus). Finding a bug through an undiscovered edge to a known block would still be possible when

https://t.me/learningnets

elaborating on top of existing samples but less likely as it would not be seen as a new case (marked and added to the corpus).

Both approaches could be implemented here, covering edges is more complex. Covering blocks was chosen for ease.

As a consequence, the addresses of all targeted basic blocks have to be extracted from the target binary using IDA. A helper already written by Samuel Groß (@5aelo) for TrapFuzz [13] was modified to be runnable on Windows and from the script. It can be either called from the script or called standalone by the user.

This led to an intriguing technical issue: when running a python script in IDA from the command line, the list of breakpoints was different from the one computed when running the same script from IDA GUI. The explanation is IDA autoanalysis which needs to be complete before starting any code analysis. This can be done using: Ida `auto_wait()`.

Input generation For coverage-guided fuzzing, the input generation strategy is a key component.

Strategy for the new input files' generation Samples from the corpus are picked and modified in the following ways:

— Append: n bytes are appended at the end of the sample.
— Mutation: random bits of the sample are flipped with a notion of density. The first mutations can be done at minimal density, starting with only 1 bit per sample. The density can then be increased once the coverage plateaus.
— Introduction of new input: once in a while, some new input is created from scratch. New random input files can be interesting to find edge cases.

The maximal mutation rate is controlled by the user. The weight of each of these 3 alternatives has been defined empirically, is hard-coded, and may need to be adjusted.

Strategy for the seeds' generation Having meaningful seeds can speed up the discovery process of the coverage guided fuzzer as the first sets of input are mostly derived from these seeds. A script enables the user to record legitimate traffic on the device of interest. That traffic is then converted to consumable input.

This requires access by the script to the symbols to put breakpoints on the entry points (NotifyIOPortRead and NotifyIOPortWrite of the relevant controller). Some prior reversing work is needed to know the exact

https://t.me/learningnets

device handler's name, and to get an approximate understanding of the used IO ports. The unexpected IO port reads / writes on the device are signaled as errors and blocking during the seed generation.

Crash qualification Another key component of coverage-guided fuzzers is the qualification of unexpected behaviors.

Classes of vulnerabilities considered The classes of vulnerabilities considered are memory corruption bugs (through pageheap), state machine logic errors, and parsing errors. Use-after-frees may get detected if the VM crashes. Race conditions can not be detected in most cases.

Memory corruption detection Ideally, the goal would be to reproduce ASAN behavior. However, this is not a feature that is directly available through the fuzzer. A workaround is to set pageheap through gflags on the target DLL prior to runs. No crash has been reported so far, so there is no evidence this strategy works. An initial idea was to add DR breakpoints around sensitive buffers, if any - but that does not scale.

Crash handling When there is a crash or a VM reset, 2 different mechanisms handle it:

- DbgShell catches an exception
- And/Or the monitoring process detects that the VM has stopped running.

In both cases, it triggers a timestamped crash folder containing:

- The configuration file
- The input file(s)
- The host event logs (channels: "Microsoft-Windows-Hyper-V-Hypervisor-Admin", "Microsoft-Windows-Hyper-V-Hypervisor-Analytic", "Microsoft-Windows-Hyper-V-Hypervisor-Operational", "Microsoft-Windows-Hyper-V-Worker-Admin", "Microsoft-Windows-Hyper-V-Worker-Analytic", "Microsoft-Windows-Hyper-V-Worker-Operational")
- The Error and Critical logs from the System event log of the guest
- The latest error.

The main script can then be started in repro mode with the crash folder as an argument. Figure 6 shows a typical crash folder.

The oldest file from the input file is always the first one consumed. In the unlikely case where both DbgShell and the monitoring process would crash, the input responsible for that crash could be retrieved as the oldest

https://t.me/learningnets

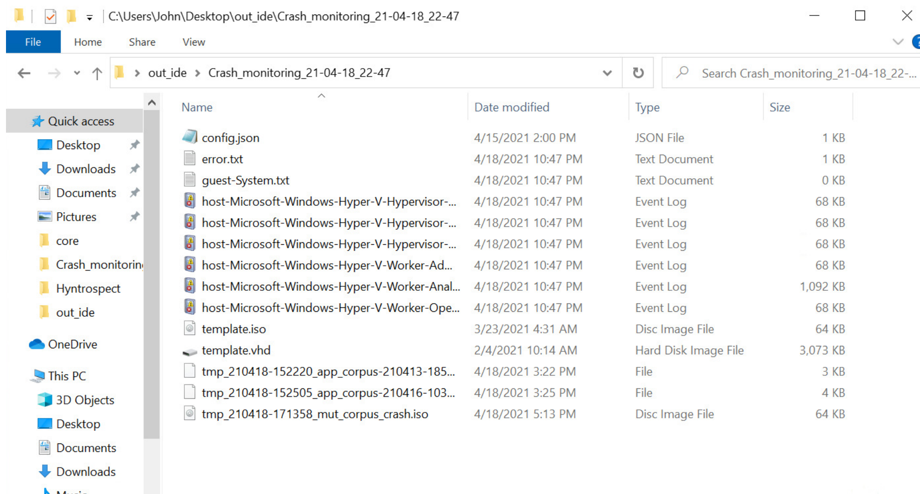

Fig. 6. Screenshot of a crash folder

temporary file in the input folder. That case would need to be handled by hand.

Some perturbations to that flow could be added by pageheap. That case has not been witnessed yet.

At this point, all the core components needed for Hyntrospect's coverage-guided fuzzing have been presented.

Additional features

Visualization of the coverage in IDA Pro IDA Pro or Binary Ninja are needed to use that feature as it relies on Lighthouse [25]. Generally speaking, IDA was arbitrarily chosen for this project (the breakpoint list initialization script is also written for IDA specifically).

The helper Create-CoverageFile.ps1 creates a Lighthouse compatible coverage file to display the coverage from the initial breakpoint list (saved by the fuzzer) and the updated breakpoint list.

Syntax: vmemulateddevices+offset

Figure 7 shows a screen capture of IDA (when all instructions are traced).

Additional helpers 4 more helpers are available for the user:

— Create-CorpusSeed.ps1: it prepares a seed in the fuzzer compliant format for the targeted emulated device.

https://t.me/learningnets

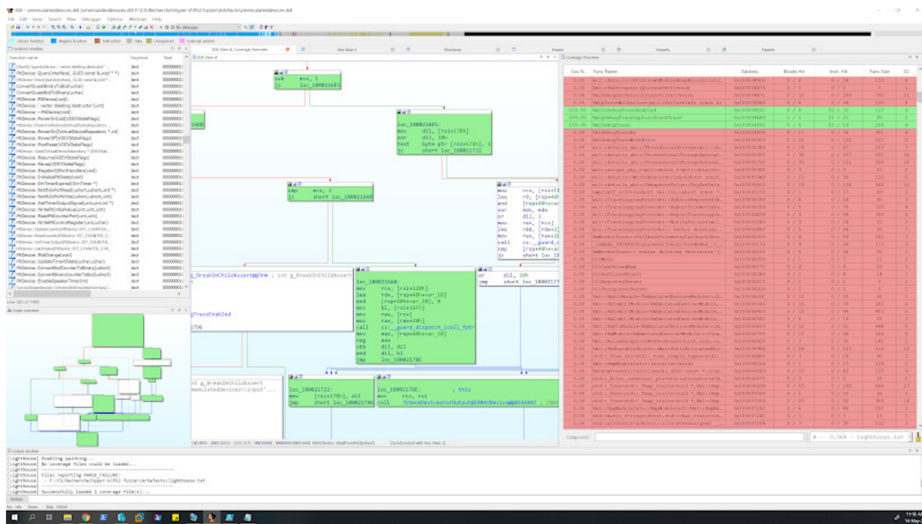

Fig. 7. Screenshot of coverage in IDA

- Translate-InputBytesToFulltext.ps1: it translates an input file in the transcript of the corresponding IO operations.
- findPatchPoints.py: IDA script that outputs a list of the blocks' addresses of the binary.
- findPatchPointsWithKeyword.py: this is the same IDA script as findPatchPoints.py except that it filters the function names on a given keyword (to restrict the basic blocks to a relevant subset). The keyword is currently hardcoded.

The last 2 files are written in Python to be executable on IDA Pro.

5 Current results

This section exposes the results obtained by running Hyntrospect against some first targets. As of today, no security bugs have been found, though one guest VM crash was discussed with MSRC.

5.1 The first targets, runtime environment and performance

The very first implementation of the fuzzer was tested against the i8042 device which handles for example PS/2 mouse and keyboard. There was no particular reason to pick that specific device except that it was a legacy feature, so there was hope that some legacy code could have been simply ported. Once the fuzzer was implemented, some more devices were

https://t.me/learningnets

tested: the floppy controller, the IDE controller, and the VideoS3 device. Fuzzing each of these targets required locating the code responsible for these controllers, and then reversing it to find the entry points and the expected IO values.

The fuzzer so far has only been run locally on a dedicated machine. That machine is a 32 GB RAM workstation equipped with an Intel Core i9 CPU. 2 VMs (with 8 GB RAM each) ran concurrently on it for the tests.

Figure 8 is a snapshot of a run.

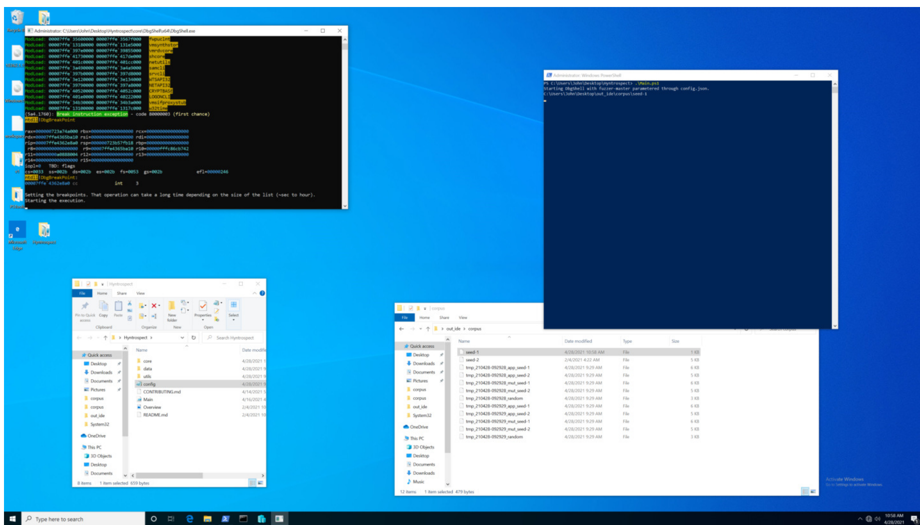

Fig. 8. Screenshot of a run

The main focus when designing the fuzzer was the execution in a native context with minor modification, and the first runs highlighted that it comes at the cost of performance (speed first but also memory and CPU consumption) - which was accepted in this analysis as long runtime on Cloud architectures are doable.

Regarding speed, three factors and bottlenecks have been identified:

- The reset of the VM and other setup on the order of 10 seconds, constant
- The number of breakpoints given and set within DbgShell: highly variable, decreasing over time when the coverage expands. It impacts both the setup and the run (when the breakpoints are removed).

https://t.me/learningnets

— The size of the input file: the more operations, the longer it takes. Files below 4 KB are advised.

These 3 factors apply for each new input case.

The **runtime per case** depends on the number of breakpoints hit and on the size of the input file: it spans from a few seconds for 1 KB input files with no breakpoint removed to several minutes for the first run on large seed files. (For example, for a first run with a 6 KB seed which removed about 450 breakpoints, the execution time went up to 28.5 minutes.)

The most important factor is the size of the list of breakpoints. Here are figures retrieved from the local setup:

Breakpoints file size	Number of breakpoints	Time to set up the breakpoints in DbgShell at each iteration
2 KB	150	immediate
5 KB	500	6 seconds
9 KB	1000	20 seconds
18 KB	2000	1 minute 15 seconds
42 KB	4751	˜ 9.5 minutes
46 KB	5175	˜ 13.5 minutes

The reader will notice that **the ratio time / number of breakpoints is not linear**. This is why it is highly recommended to **provide a minimized list of breakpoints** (focusing on one device or a subset of functions of interest). An additional helper script filtering IDA functions on a keyword was created.

As a note, the original size of the DLL used for this test was 611 KB which resulted in a breakpoints list of 132 KB containing 15242 breakpoints. It was shrunk to 5175 breakpoints to only cover some of the code (a third of the DLL).

Also, **the longer the fuzzer runs, the faster the runs are** as there are fewer breakpoints set and fewer breakpoints reached during execution.

Setting a maximum for the size of the input file is also an efficient technique to keep the runs short.

Size of the input file	Duration of the run (with no bp removed)
480 bytes	30 seconds
960 bytes	1 minute
1900 bytes	2 minutes
3840 bytes	3 minutes and 15 seconds

https://t.me/learningnets

Unlike the number of breakpoints, the size of the input seems to have an almost linear impact on the time it takes to run the case.

For the current version of Hyntrospect, **speed is as a consequence the major drawback. Some optimizations in future versions of the tool could enhance its speed** and enable the use of a larger input. The most promising idea would be to replace DbgShell by either another solution or to reimplement a minimal debugger that would insert the int3 instructions and keep a shadow table of the addresses and initial instructions.

5.2 Coverage of these targets

The coverage displayed here is defined per basic blocks covered. First, the breakpoint list of the DLLs has been trimmed to keep only the breakpoints that are related to the device and the coverage was calculated on that subset.

It has been computed on the local runtime environment. The fuzzer has run a maximum of 3 days for each case. Porting it to Google Cloud Platform is currently work in progress.

vmemulateddevices.dll	Access with NotifyIO-PortRead/Write	Current coverage
I8042Device Ps2Keyboard Ps2Mouse	IO ports 0x60, 0x61, 0x62, 0x64	40%
VideoS3Device VideoDevice VideoDirt	IO port 0x3B0 -> 0x3DF, 0x4AE0-> 0x4AEF	42.7%

VmEmulatedStorage.dll	Access with NotifyIO-PortRead/Write	Current coverage
FloppyControllerDevice	IO ports 0x3F0 -> 0x3F5, 0x3F7	43.3%
IdeControllerDevice	IO ports 0x1F0->0x1F7, 0x170->0x177, 0x3F6, 0x376 + for write: 0x1E0->0x1EF, 0x160->0x16F	28.8%

A certain number of blocks were not reached for 2 reasons:

https://t.me/learningnets

- the setup functions are not called as the fuzzer attaches to an already running VM.
- the debug strings blocks are skipped.

Porting the code to Cloud is the next goal to enhance that coverage.

In parallel, Microsoft has published their own coverage of Hyper-V [15, slide 24].

5.3 Guest VM crash caused by memory mapping

This first short fuzzing campaign discovered one bug. That issue was found when testing the very first device: the I8042 device. This was not considered as a security bug or a usable primitive.

The fuzzer reported several input files that led to crashes. The behavior could be reproduced consistently for these files. Every run led to a BSOD of the VM, and even more interestingly to different error messages and stacks at each run:

- `SYSTEM_SERVICE_EXCEPTION` (0x3b);
- `PFN_LIST_CORRUPT` (4e);
- `PAGE_FAULT_IN_NONPAGED_AREA` (0x50);
- `ATTEMPTED_WRITE_TO_READONLY_MEMORY` (0xbe);
- `KERNEL_SECURITY_CHECK_FAILURE` (0x139);
- ...

When analyzing the set and minimizing the input, 2 instructions seemed responsible: OUT 0x64 0xd1 followed (not necessarily immediately) by OUT 0x60 <value>.

Isolating those 2 instructions enabled more targeted debugging using WinDbg by attaching it to vmwp.exe and running the commands from the guest. This also enabled in parallel understanding the root cause better in IDA.

When passing OUT 0x64 0xd1 and then OUT 0x60 <value with bit 0 set to 1 and bit 1 set to 0>, the execution flow leads to "PciBusDevice::HandleA20GateChange" which updates the memory addressing and thus mapping on the host, but does not seem to update the guest with that information.

This has not been investigated in much depth after that point. Indeed, as the commands on the VM are executed as administrator, this could not lead to an elevation inside the VM. The host is not affected by the bug. A potential interest in remapping the memory would have been to circumvent virtualization-based security measures by leaking memory information that should not be readable by the VM kernel. This however

https://t.me/learningnets

seems extremely hard to accomplish as the first mis-allocation makes the kernel panic when the VM execution resumes. In theory, the only way to work around this would be to set beforehand a bunch of samples of kernel code that would dump memory executing in high priority (... but dump where? disk accesses are too slow). And even if it were to work, it would be extremely unstable. All in all: this does not seem doable.

This was shared with Saar Amar from MSRC in January 2021.

Even if the outcome cannot be used, this validates the behavior of the fuzzer, crash handling and reproduction scripts.

6 Future endeavours

6.1 Fuzzer internals

Some trade-offs were mentioned when explaining the design choices. The main ideas for future improvements are:

- Refining the mutation strategy, possibly by leveraging existing engines or their logic. Machine learning could help. Here [44] is a paper focusing on that topic.
- The fuzzer is currently targeting only userland, not the kernel of the root partition. DbgShell cannot do kernel debugging. A workaround is doing nested virtualization and debugging the VM kernel and hypervisor with WinDbg instances for the fuzzer lifetime. If any of those happened to crash, it would be non automated but certainly worth the investigation. The material is saved before being used so the latest state when rebooting would reflect what led to the crash. A heavier modification to adapt the fuzzer would be to monitor level 0 which would inject into level 1 which would inject into level 2.
- Speed-wise, as expressed before, two main factors reduce the performance of the fuzzer: the restoration of a snapshot for the debuggee VM, and the number of breakpoints set within DbgShell. It is strongly recommended to sanitize the list of breakpoints' addresses manually before running the fuzzer for large targets by only keeping the sections of interest. The next step would be to replace DbgShell.
- Minimization of the cases.

6.2 Porting to GCP

The goal of porting to Google Cloud Platform is to scale in 2 different ways:

https://t.me/learningnets

— Port the fuzzer to new devices and have more fuzzers run in parallel.
— Run the fuzzers faster and for a longer time, which will very likely result in extending the coverage.

This is currently work in progress. The environment as defined before can be pushed to GCP.

6.3 Porting the fuzzer to other userland targets

The first goal will be to keep covering IO-based targets that are in userland.

6.4 Broader cases with some PowerShell development

Beyond that, the goal will be to port the fuzzer to other targets in userland by passing different sets of commands than the IO port commands. This part can be achieved by modifying the expected configuration file and the fuzzer engine (mostly Main.ps1 and fuzzer-master.ps1). Some additional files would also require adjustments. The goal would be to keep the structure of the fuzzer with a Main calling a fuzzer master which spawns input generators, VM monitoring... and use it as a frame for a whole range of commands that could be executed within the VM.

7 Conclusion

Hyntrospect is a new tool that enables fuzzing Hyper-V emulated devices' controllers using code coverage to guide the generation of the fuzzer input. The motivation was to get code coverage in a similar way as Microsoft did with their fuzzer, but without access to the sources. The core of its design is based on running the binaries in their real environment through debugging to preserve all the aspects and side effects of this complex stack, the main drawback being performance (speed). The coverage results presented here and the guest VM crash found during the runs validated its behavior. No security vulnerability has been found yet on local runs (up to 3 days per run on 4 different devices). Porting it to Cloud might enhance the results. The fuzzer was opened to the community to share and get contributions. One of the main goals for future development will be to port it to broader use cases in the userland of the root partition.

https://t.me/learningnets

References

1. CHIPSEC GitHub repository. https://github.com/chipsec/chipsec.
2. IDA Pro page. https://www.hex-rays.com/products/ida/.
3. x86 Instruction Set Reference IN. https://c9x.me/x86/html/file_module_x86_id_139.html.
4. x86 Instruction Set Reference OUT. https://c9x.me/x86/html/file_module_x86_id_222.html.
5. CVE-2015-3456, 2015. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-3456.
6. CVE-2018-0888, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0888.
7. CVE-2018-0959, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0959.
8. Saar Amar and Daniel King. Breaking VSM by Attacking SecureKernel - BlackHat USA 2020, 2020. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2020_08_BlackHatUSA/Breaking_VSM_by_Attacking_SecureKernel.pdf.
9. Damien Aumaitre. Fuzz and Profit with WHVP - SSTIC 2020, 2020. https://www.sstic.org/2020/presentation/fuzz_and_profit_with_whvp/.
10. Damien Aumaitre. WHVP GitHub repository, 2020. https://github.com/quarkslab/whvp.
11. Joe Bialek. Exploiting the Hyper-V IDE Emulator to Escape the Virtual Machine - BlackHat USA 2019, 2019. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_08_BlackHatUSA/BHUSA19_Exploiting_the_Hyper-V_IDE_Emulator_to_Escape_the_Virtual_Machine.pdf.
12. Diane Dubois. Hyntrospect GitHub repository, 2021. https://github.com/googleprojectzero/Hyntrospect.
13. Samuel Groß (*5aelo*). TrapFuzz GitHub repository. https://github.com/googleprojectzero/p0tools/tree/master/TrapFuzz.
14. Samuel Groß (*5aelo*). Fuzzing ImageIO, 2020. https://googleprojectzero.blogspot.com/2020/04/fuzzing-imageio.html.
15. David *dwizzzle* Weston. Keeping Windows Secure - Bluehat IL 2019, 2019. https://github.com/dwizzzle/Presentations/blob/master/David%20Weston%20-%20Keeping%20Windows%20Secure%20-%20Bluehat%20IL%202019.pdf.
16. Brandon Falk (*gamozolabs*). mesos GitHub repository. https://github.com/gamozolabs/mesos.
17. Arthur Khudyaev (*gerhart_x*). gerhart_x GitHub page. https://github.com/gerhart01.
18. Arthur Khudyaev (*gerhart_x*). Hyper-V Internals blog by gerhart_x, 2021. https://hvinternals.blogspot.com/.
19. Michał Zalewski (*lcamtuf*). Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt.
20. Jordan Rabet (*smealum*). Hardening Hyper-V through offensive security research - BlackHat USA 2018, 2018. https://i.blackhat.com/us-18/Thu-August-9/us-18-Rabet-Hardening-Hyper-V-Through-Offensive-Security-Research.pdf.

https://t.me/learningnets

21. Alisa Esage. Hypervisor Vulnerability Research State of the Art - Zer0Con 2020, 2020. https://alisa.sh/slides/HypervisorVulnerabilityResearch2020.pdf.
22. Ivan Fratric. winafl GitHub repository. https://github.com/googleprojectzero/winafl.
23. Ivan Fratric. winafl with Dynamorio Instrumentation mode. https://github.com/googleprojectzero/winafl/blob/master/readme_dr.md.
24. Ivan Fratric. Jackalope GitHub repositoty, 2020. https://github.com/googleprojectzero/Jackalope.
25. gaasedelen. Lighthouse GitHub repository. https://github.com/gaasedelen/lighthouse.
26. Google. Coverage guided vs blackbox fuzzing. https://google.github.io/clusterfuzz/reference/coverage-guided-vs-blackbox/.
27. Intel. Processor Tracing, 2013. https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html.
28. Intel. Pin - A Dynamic Binary Instrumentation Tool, 2018. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.
29. Nicolas Joly and Joe Bialek. A Dive in to Hyper-V Architecture and Vulnerabilities - BlackHat USA 2018, 2018. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2018_08_BlackHatUSA/A%20Dive%20in%20to%20Hyper-V%20Architecture%20and%20Vulnerabilities.pdf.
30. LLVM. libFuzzer. https://llvm.org/docs/LibFuzzer.html.
31. Microsoft. DbgShell GitHub repository. https://github.com/microsoft/DbgShell.
32. Microsoft. Generation 2 Virtual Machine Overview, 2016. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/dn282285(v=ws.11).
33. Microsoft. Virtual Machine automation and management using PowerShell, 2016. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/user-guide/powershell-direct.
34. Microsoft. Debugger Engine Introduction, 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/introduction.
35. Microsoft. GFlags and PageHeap, 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap.
36. Microsoft. Virtualization-based Security (VBS), 2017. https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.
37. Microsoft. Hyper-V Architecture, 2018. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture.
38. Microsoft. Hyper-V symbols for debugging, 2018. https://docs.microsoft.com/en-us/virtualization/community/team-blog/2018/20180425-hyper-v-symbols-for-debugging.
39. Microsoft. Manage Hyper-V on Windows Server, 2018. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-on-windows-server.

https://t.me/learningnets

40. Microsoft. Description of User Account Control and remote restrictions in Windows Vista, 2020. https://docs.microsoft.com/en-US/troubleshoot/windows-server/windows-security/user-account-control-and-remote-restriction.

41. MSRC. First Steps in Hyper-V Research, 2018. https://msrc-blog.microsoft.com/2018/12/10/first-steps-in-hyper-v-research/.

42. MSRC. Attacking the VM Worker Process, 2019. https://msrc-blog.microsoft.com/2019/09/11/attacking-the-vm-worker-process/.

43. Quarkslab. QuarkslaB Dynamic binary Instrumentation. https://qbdi.quarkslab.com/.

44. Luping Liu Cheng Huang Yan Wang, Peng Jia and Zhonglin Liu. A systematic review of fuzzing based on machine learning techniques. *PloS one*, 15(8):e0237749, 2020.

https://t.me/learningnets