Hyntrospect: A Fuzzer for Hyper-V Devices

Diane Dubois - didu@google.com - @0xdidu
SSTIC
June 2 2021
Whoami?

- Security Engineer at Google
- 20% with Project Zero
- Passionate about vulnerability research on systems
- @0xdidu
Why Hyper-V?

● Project Zero mission: aims to reduce harm caused by targeted attacks on the Internet

● Hyper-V
  ○ Hypervisor running Azure, Microsoft Cloud
  ○ Modern versions of Windows run it (Virtualization-based security)
  ○ Possibly a high impact if there are 0-days in the wild

● ... and because virtualization is a fun topic
  ○ Spans multiple layers from hardware to high level software
  ○ Some complex implementations
Goals

- **Instrumenting Hyper-V for vulnerability research**
  - A fuzzer called Hyntrospect was developed and open sourced
    - [https://github.com/googleprojectzero/Hyntrospect](https://github.com/googleprojectzero/Hyntrospect)
      - Coverage-guided
      - On closed-source binaries
      - Pragmatic approach, using existing Hyper-V features and Windows tools
      - In a real execution environment
  - Its internals and current results (coverage...) are presented

- **Finding vulnerabilities and reporting them to Microsoft**
Agenda

- Background on Hyper-V
- The Research Target
- Hyntrospect fuzzer
- Current results
- Future endeavours
Background on Hyper-V
Hyper-V Architecture Overview

Source: Microsoft
What is a “Guest to Host Escape”?

- Gaining code execution on one of the hypervisor layers from a virtual machine
- On Hyper-V: ambiguous
  - Hypervisor layer
  - Root partition (kernel / userland)
- Other type of attack: host denial of service
The Hypervisors’ Attack Surface

- As defined by Alisa Esage (Zer0Con 2020):
... and in practice
Hyper-V Attack Surface

- Hypervisor
  - Hypercall handlers
  - Faults
  - Instruction emulation
  - Intercepts
  - Register access (MSRs...)

- Root partition kernel attack surface
  - VMBus

- Root partition userland attack surface
  - Emulated devices
  - Integration components

- ... and this list is not exhaustive

- MSRC: first-steps-in-hyper-v-research
State of the Art on the Research

● MSRC and Microsoft publish on Hyper-V
  ○ Blog posts to help vulnerability researchers
    ■ e.g. First Steps in Hyper-V research
  ○ Posts on Hyper-V components
  ○ Several presentations at conferences on vulnerabilities found internally
    ■ e.g. Breaking VSM by Attacking SecureKernel at BlackHatUSA 2020
  ○ Symbols provided for some key components

● Active external contributors
  ○ @gerhart_x and his dedicated blog
  ○ @alisaesage
  ○ Presentation by Damien Aumaitre on whvp at SSTIC 2020

● And many more (a list can be found on GitHub/gerhart01/Hyper-V-Internals)
The Research Target
The **Emulated Devices Controllers**

- Examples: floppy disks, IDE, PS2
- Called “virtual devices” or “VDEVs” at Microsoft
- Emulation of hardware controller by the hypervisor
  - Real hardware controllers use and access control
  - Resources shared
  - Guest operating systems unmodified
- Implemented for Hyper-V generation 1 VMs
  - Azure mostly uses this generation
- Userland of the root partition
- In DLLs loaded by the worker process
Why Choosing the Emulated Devices?

● Complex (state machines)
  ○ For example: enabling / disabling ports, updating a status register, waiting for a command
● Several bugs on several hypervisors
● Azure mostly uses Generation 1 VMs
● Hyper-V is developed in C++
● Potential “guest to root partition” escapes
Life of a Request

- Communication through IO ports
  - CPU instructions: IN / OUT
  - “IN EAX, DX”: input from I/O port in DX into EAX
  - “OUT DX, EAX”: output in EAX to I/O port address in DX
  - More details and versions in Intel manuals
- Communication through the hypervisor, the VID, and callbacks
- More on MSRC blogpost “Attacking the VM worker process”
- Some VDEVs are more complex with MMIO handling for instance, or the use of the VMBus
Some Reverse Engineering

● DLL implementing the controllers
● Typical IO handlers
  ○ $Device::NotifyIOPortRead
  ○ $Device::NotifyIOPortWrite
● + : Symbols available, no particular difficulty (no obfuscation...)
● - : Reversing C++ and its indirect calls
● Example with VmEmulatedStorage.dll
Hyntrospect: A Fuzzer for the Emulated Devices

https://github.com/googleprojectzero/Hyntrospect
Inspiration

- libFuzzer: coverage-guided approach
- Microsoft publication on their coverage (Keeping Windows Secure - Bluehat IL 2019)
- CVE-2018-0959 + Dedicated MSRC blogpost

How to do the same, closed source?
Existing Tools for Windows Binaries Fuzzing

● Gathering Coverage
  ○ DynamoRIO
  ○ Intel Pin
  ○ Intel PT (though this is not a tool like previous two)
  ○ Mesos
  ○ QDBI for Windows
  ○ TinyInst

● Fuzzers
  ○ WinAFL + DynamoRIO
  ○ Jackalope
  ○ whvp

● Memory Corruption Detection
  ○ PageHeap
So Why Another Toolchain?

● The target is a DLL
  ○ This disqualifies all the fuzzers that only apply to executables
● Emulating only the relevant functions is hard
  ○ All the VM context would be needed
● vmwp binary and the DLL cannot be restarted with instrumentation
  ○ That would mean restarting the whole VM for each run
● The runtime operations are specific
  ○ Injecting / mocking the injection of IOs
● Some tools were developed during Hyntrospect development
● Managing all the blocks with a minimal set of languages is hard
● The fuzzer will be ported to similar use cases
  ○ vSMB, or with some architectural changes the network stack...
Scope

- Windows guest VM
- Intel CPU
- Generation 1 VMs
- Binaries (DLLs/EXEs) in the userland of the root partition
# Design Choices at a Glance

<table>
<thead>
<tr>
<th>Design Choice</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emulation vs execution</td>
<td>Execution of a VM through a debugger (DbgShell) at runtime</td>
</tr>
<tr>
<td>Coverage</td>
<td>Tracked with the int3 technique described by @5aelo for TrapFuzz / @gamozolabs mesos</td>
</tr>
<tr>
<td>Memory corruption detection</td>
<td>Pageheap (gflags)</td>
</tr>
<tr>
<td>Type of bugs</td>
<td>Memory corruption</td>
</tr>
<tr>
<td></td>
<td>State machine logic errors</td>
</tr>
<tr>
<td></td>
<td>[Use after free]</td>
</tr>
<tr>
<td></td>
<td>Race conditions</td>
</tr>
<tr>
<td>Design Choices at a Glance (2)</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Environment reset</strong></td>
<td>Hyper-V checkpoints = <strong>snapshots</strong></td>
</tr>
<tr>
<td><strong>Mutation strategy</strong></td>
<td><strong>Custom</strong></td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td><strong>PowerShell</strong> (except for the IDA scripts)</td>
</tr>
<tr>
<td><strong>External dependencies</strong></td>
<td><strong>DbgShell, CHIPSEC,</strong> [pageheap], [LightHouse], [IDA]</td>
</tr>
</tbody>
</table>
Overview of Hynthrospect

Diagram showing the interaction between host (root partition) and guest (debuggee) components with Fuzzer, Debugger, Target (.dll), Hyper-V drivers, chipsec, and Chipsec driver.
Workflow

Blocks’ addresses

config.json

Main.ps1
Workflow

- Blocks’ addresses
- config.json
- Main.ps1
- fuzzer-master.ps1
Workflow

1. Blocks' addresses
2. config.json
3. Main.ps1
4. helper.psm1
5. fuzzer-master.ps1
6. input-generator.ps1
7. corpus
8. input
Workflow

- Blocks’ addresses
- config.json

Main.ps1 → fuzzer-master.ps1 → helper.psm1 → debugger.ps1 → input-generator.ps1

- corpus
- input

debugs

VM

vmwp.exe
Workflow

Blocks’ addresses

Main.ps1

helper.psm1

debbuger.ps1

vm-monitoring.ps1

input-generator.ps1

corpus

input

config.json

vmwp.exe

VM
Coverage collection and guidance

- Block coverage
Coverage collection and guidance

- Block coverage
Coverage collection and guidance

- Block coverage
Coverage collection and guidance

• Block coverage
  ○ Versus edge coverage: easier to implement but does not promote rare paths
Coverage collection and guidance

● Block coverage
  ○ Versus edge coverage: easier to implement but does not promote rare paths
  ○ No counter
Coverage collection and guidance

● Block coverage
  ○ Versus edge coverage: easier to implement but does not promote rare paths
  ○ No counter

● int3 technique
  ○ Pre-compute the list of targeted blocks’ addresses
  ○ Set int3 at the beginning of each block
  ○ Each int3 reached = coverage increase
  ○ The int3 is removed, input file handled, execution resumes
  ○ Faster over time
Generation of the Input File

- Record of seeds at the beginning [optional]
  - Record of legitimate traffic
- Corpus of “interesting files”
  - Corpus files = permanent residents
  - Input files = temporary residents to be tested
- Coverage increase -> truncated input file added to the corpus
  - Will influence future runs
- 3 strategies: mutate, append, generate randomly

- Format of input files
  - Byte 0 % 2 -> IN / OUT operation
  - Byte 1 % (number of ports) -> selected IO port
  - Byte 2 % 3 -> length
  - If OUT and based on length -> value
Crash Qualification

- 2 levels of monitoring: debugger level + monitoring process
  - Tip: the monitoring process can track the VM uptime
    - avoid while(true)
    - avoid missing quick status change
      (up-down-up again) ... as if it was blinking

- Crash folder created with logs and artefacts to re-run the case
Coverage visualization in IDA

● Optionally, using a helper and IDA+LightHouse
Current Results
Local Runs

● First targets: i8042 (PS/2), videoS3, floppy, IDE
  ○ Example: I8042 device with IO ports 0x60, 0x61, 0x62, 0x64

● Local setup: dedicated workstation with 32 GB RAM and Intel Core i9 CPU
  ○ 8 GB per VM, 1 or 2 vCPUs

● Speed limitation
  ○ Main factor: number of breakpoints
  ○ Time to set them / update them in DbgShell
  ○ Not linear

<table>
<thead>
<tr>
<th>Number of breakpoints</th>
<th>Time to set up the breakpoints in DbgShell at each iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>immediate</td>
</tr>
<tr>
<td>500</td>
<td>6 seconds</td>
</tr>
<tr>
<td>1000</td>
<td>20 seconds</td>
</tr>
<tr>
<td>2000</td>
<td>1 minute 15 seconds</td>
</tr>
</tbody>
</table>

● Next goal: port the fuzzer to GCP
Coverage (3 days run)

- Start / init / stop functions not called
  - Attaching to a running VM
- Debug strings blocks skipped

<table>
<thead>
<tr>
<th>DLL</th>
<th>Current coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmemulateddevices.dll</td>
<td></td>
</tr>
<tr>
<td>VideoS3Device</td>
<td>42.7%</td>
</tr>
<tr>
<td>i8042Device</td>
<td>40%</td>
</tr>
<tr>
<td>VmEmulatedStorage.dll</td>
<td></td>
</tr>
<tr>
<td>FloppyControllerDevice</td>
<td>43.3%</td>
</tr>
<tr>
<td>IdeControllerDevice</td>
<td>28.8%</td>
</tr>
</tbody>
</table>
Guest VM Crash Found

- On i8042 device
- Reproducible
- BSOD of the VM with different error messages at each run
  - SYSTEM_SERVICE_EXCEPTION (0x3b)
  - PFN_LIST_CORRUPT (4e)
  - ATTEMPTED_WRITE_TO_READONLY_MEMORY (0xbe)
  - KERNEL_SECURITY_CHECK_FAILURE (0x139)
- Memory corruption error
Some More Investigation

- Narrowed down the case
  - Sequence of 2 OUT operations

- State machine, path accessible in 2 steps

- PciBusDevice::HandleA20GateChange
  - Legacy A20 device
  - Updates the memory mapping on the host
  - ... but the guest keeps the same mapping

- Question: possible compromise of VBS?
Follow-up

- In practice, impossible to exploit
- Not a security bug
- Shared with MSRC

- Validates the behavior of the fuzzer, crash handling and reproduction scripts
Future endeavours
Design Limitations

● Restricted to the userland of the root partition
  ○ Limits the attack surface as parts of the virtualized stack are in the root partition kernel and hypervisor

● Not optimized for speed
  ○ More expensive in Cloud as more cycles are needed
Future Work

● **Development of the fuzzer internals**
  ○ Mutation strategy
  ○ Userland vs kernel
  ○ Speed-related updates: minimal debugger?

● **Porting to GCP**
  ○ Port to new devices
  ○ Run faster and longer

● **Adapting to other root partition targets**
  ○ Keeping the frame and “basic blocks”
  ○ Changing the commands and input consumption
Conclusion