Satoshi's notes About

Hunting down the HVCI bug in UEFI

Jan 15, 2024

- Discovery to reporting
  - Discovery
  - Exploitation
  - Partial root causing
  - Reporting
- Technical details and fixes
  - Intel VT-x and its limitation
  - Intel VT-d
  - DMA remapping
  - DMAR ACPI table and RMRR structure
  - Interaction with Windows, and the bug
  - Fixes
- Summary

This post was coauthored with Andrea Allievi (@aall86), a Windows Core OS engineer who analyzed and fixed the issue.

This post details the story and technical details of the non-secure Hypervisor-Protected Code Integrity (HVCI) configuration vulnerability disclosed and fixed with the January 9th update on Windows. This vulnerability, CVE-2024-21305, allowed arbitrary kernel-mode code execution, effectively bypassing HVCI within the root partition.

While analysis of the HVCI bypass bug alone can be interesting enough, I and Andrea found that the process of root causing and fixing it would also be fun to detail and decided to write this up together. The first half of this article was authored by me, and the second half was by Andrea. Readers can expect a great deal of Windows internals and x64 architecture details thanks to Andrea's contribution!

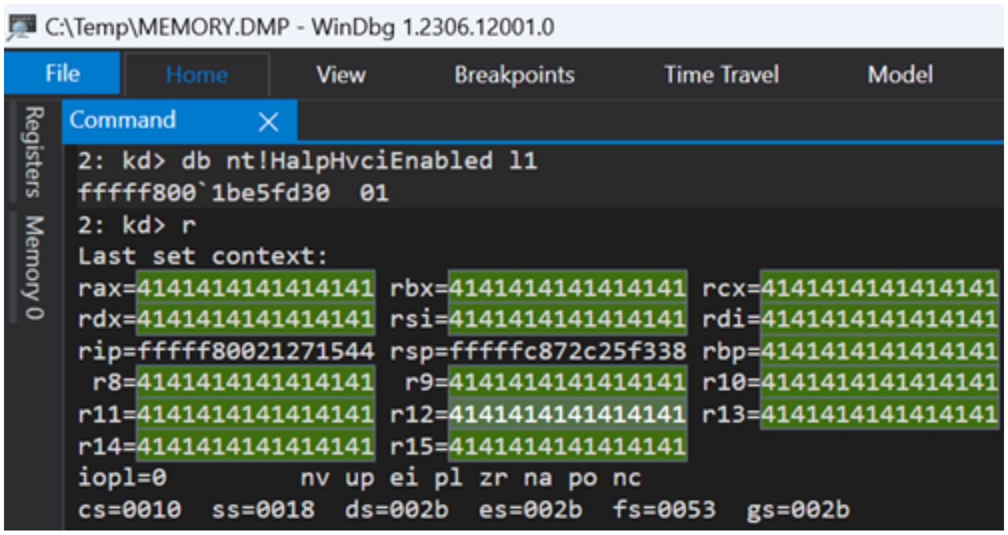
Discovery to reporting

Discovery

The discovery of the bug was one of the by-products of hvext.js, the Windbg extension for studying the implementation of Hyper-V on Intel processors. With the extension, I dumped EPT on a few devices to better understand the implementation of HVCI, and one of them showed readable, writable, and kernel-mode executable (later referred to as RWX) guest physical addresses (GPAs). When HVCI is enabled, such GPAs should not exist as it would allow generation and execution of arbitrary code in kernel-mode. Eventually, out of 7 Intel devices I had, I found 3 devices with this issue, ranging from 6th to 10th generation processors.

Exploitation

Exploiting this issue for a verification purpose was trivial as the RWX GPAs did not change across reboot or when test-signing was enabled. I wrote the driver that remapped a choice of linear address onto one of RWX GPAs and placed shellcode there, and was able to execute the shellcode as expected! If HVCI were working as intended, the PoC driver would have failed to write shellcode and caused a bug check. For more details on the PoC, see the report on GitHub.



```
2: kd> db nt!HalpHvciEnabled l1
fffff800`1be5fd30  01
2: kd> r
Last set context:
rax=4141414141414141 rbx=4141414141414141 rcx=4141414141414141
rdx=4141414141414141 rsi=4141414141414141 rdi=4141414141414141
rip=fffff80021271544 rsp=fffffc872c25f338 rbp=4141414141414141
 r8=4141414141414141  r9=4141414141414141 r10=4141414141414141
r11=4141414141414141 r12=4141414141414141 r13=4141414141414141
r14=4141414141414141 r15=4141414141414141
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b
```

I asked Andrea about this and was told it could be a legit issue.

Partial root causing

I was curious why the issue was seen on only some devices and started to investigate what the RWX GPAs were.

Contents of those GPAs all seemed zero during runtime, and RamMap indicated it was outside NTOS-managed memory. I dumped memory during the Winload debug session, but they were still vastly zero. It was the same even during the UEFI shell phase.

At this point, I thought it might be UEFI-reserved regions. First, I realized that the RWX GPAs were parts of Reserved regions but did not exactly match, per the output of the `memmap` UEFI shell command. Shortly after, I discovered the regions exactly corresponded to the ranges reported by the Reserved Memory Region Reporting (RMRR) structure in the DMAR ACPI table.

I spent more time trying to understand why they were marked as RWX and why it occurred on only some machines. Eventually, I could not get the answers, but I was already reasonably satisfied with my findings and decided to hand this over to MSFT.

Reporting

I sent an initial write-up to Andrea, then, an updated one to MSRC a few days later. Though, it turned out that Andrea was the engineer in charge of this case. Such a small world.

Nothing much happened until mid-October when Andrea privately let me know he root caused and fixed it, and also offered to write up technical details from his perspective.

So the following is his write-up with a lot of technical details!

Technical details and fixes

Intel VT-x and its limitation

So what is the DMAR table and why was important in this bug?

To understand it we should take a step back and briefly introduce one of the first improvements of the Intel Virtualization Extension (Intel VT-**x**). Indeed, Intel VT-x was introduced back around the year 2004 and, in its initial implementation, it misses some parts of the technology that are currently used in modern Operating Systems (in 2023). In particular:

1. The specifications did not include a hardware Stage-2 MMU able to perform the translation of the Guest physical addresses (GPAs) to System physical addresses (SPAs). First Hypervisors (like VmWare) were using a technique calling Memory Shadowing
2. Similarly, the specification did not protect devices performing DMA to system memory addresses.

As the reader can imagine, this was not compatible with the Security standard required nowadays, so multiple "addendums" were added at the first implementation. While in this article we are not talking about #1 (plenty of articles are available online, like this one), we will give a short introduction and description of the Intel VT-**d** technology, which aims at protecting Device data transfer initiated via DMA.

Intel VT-d

Intel maintains the VT-d technology specifications at the following URL: https://www.intel.com/content/www/us/en/content-details/774206/intel-virtualization-technology-for-directed-i-o-architecture-specification.html

The document is updated quite often (at the time of this writing, we are at revision 4.1) and explains how an I/O memory management unit (IOMMU) can now protect devices to access memory that belongs to another VM or is reserved for the the host Hypervisor or OS.

A device can be exposed by the Hypervisor in different ways:

- **Emulated** devices always cause a VMEXIT and they are emulated by a component in the Virtualization stack.
- **Paravirtualized** devices are synthetic devices that communicate with the host device through a technology implemented in the Host Hypervisor (VmBus in case of HyperV).
- **Hardware** accelerated devices are mapped directly in the VM. (readers who want to know more can check Chapter 9 of the Windows Internals book).

All the hardware devices are directly mapped in the root partition by the HV. To correctly support Hardware accelerated devices in a child VM the HV needs an IOMMU. But what exactly is an IOMMU? To be able to isolate and restrict device accesses to just the resource owned by the VM (or by the root partition), an IOMMU should provide the following capabilities:

- I/O device assignment
- **DMA remapping** to support address translations for Direct Memory Accesses (DMA) initiated by the devices
- Interrupt remapping and posting for supporting isolation and routing of interrupts to the appropriate VM

DMA remapping

The DMA remapping capability is the feature related to the bug found in the Hypervisor. Indeed, to properly isolate DMA requests coming from hardware devices, an IOMMU must translate request coming from the endpoint device attached to the Root Complex (which, in its simplest form, a DMA request is composed of a target DMA address/size and originating device ID specified as Bus/Dev/Function - BDF) to its corresponding Host Physical Address

(HPA).

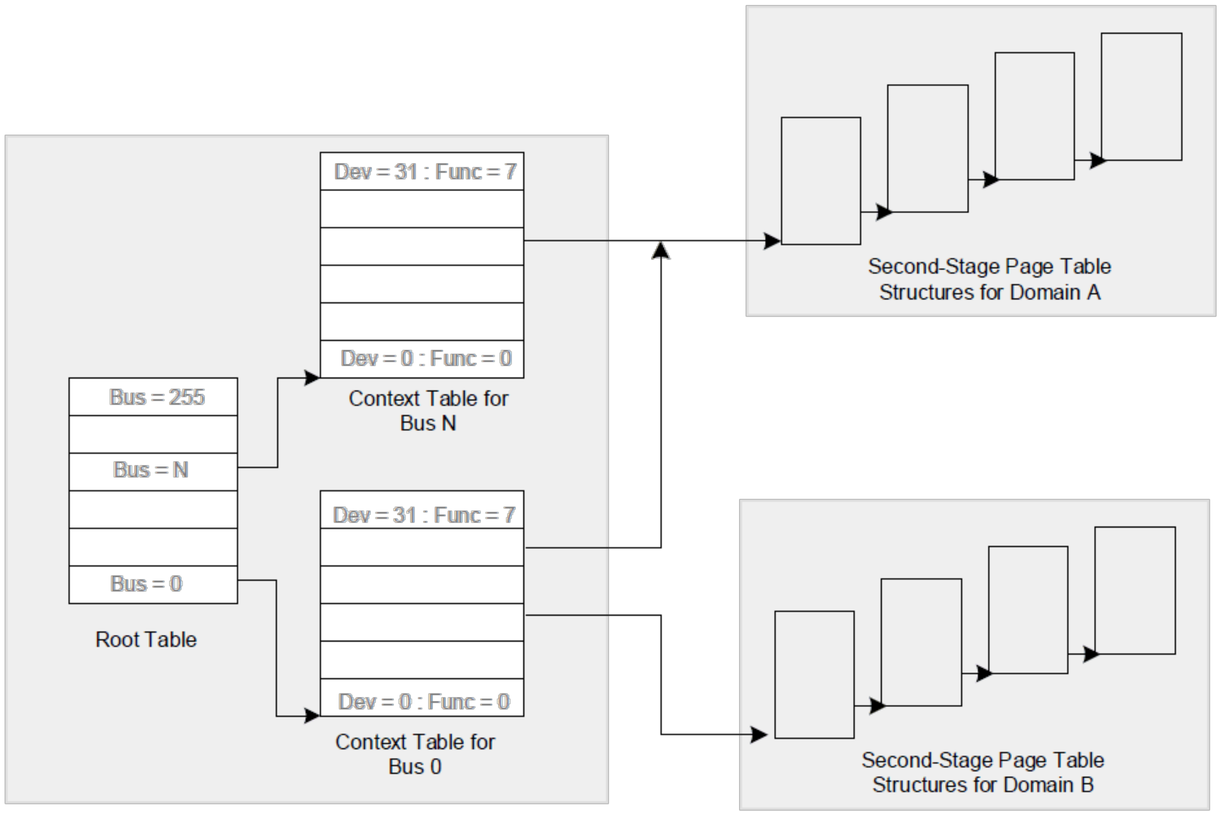
Note that readers that do not know what a Root Complex is or how the PCI-Ex devices interact with the system memory bus can read the excellent article by Gbps located here (he told me that a part 2 is coming soon :-)).

The IOMMU defines the **Domain** concept, such an isolated environment in the platform for which a subset of host physical memory is allocated (basically a bunch of isolated physical memory pages). The isolation property of a domain is achieved by blocking access to its physical memory from resources **not** assigned to it. Software creates and manages domains, allocates the backing physical memory (SPAs), and sets up the DMA address translation function using "Device-to-Domain Mapping" and "Hierarchical Address translation" structures.

Skipping a lot of details, both structures can be thought as "Special" page tables:

- Device–to-Domain Mapping structures are addressed by the BDF of the source device. In the Intel manual this is called "Source ID" and yield backs the domain ID **and** the root Address Translation structures for the domain (yes, entries in this table are 128 bits indeed, and not 64).
- Hierarchical Address translation structures are addressed by the source DMA address, which is treated as GPA, and outputs the final Host Physical address used as target for the DMA transfer.

The concepts above are described by the following figure (source: Intel Manual):



DMAR ACPI table and RMRR structure

The architecture defines that any IOMMU present in the system must be detected by the BIOS and announced via an ACPI table, called DMA Remapping Reporting (DMAR). The DMAR is composed of multiple remapping structures. For example, an IOMMU is reported with the DMA Remapping Unit Definition (DRHD) structure. Describing all of them is beyond the scope of this article.

What if a device always needs to perform DMA transfer with specific memory regions? Certain devices, like the Network controller, when used for debugging (for example in KDNET), or the USB controller, when used for legacy Keyboard emulation in the BIOS, should always be able to perform DMA both before and after setting up IOMMU. For these kinds of devices, the Reserved Memory Region Reporting (**RMRR**) structure is used by the BIOS to describe regions of memory where the DMA should always be possible.

Two important concepts described in the Intel manual regarding the RMRR structure:

1. The BIOS should report physical memory described in the RMRR as **Reserved** in the UEFI memory map.
2. When the OS enables DMA remapping, it should set up the Second-stage address translation structures for mapping the physical memory described by the RMRR using the "identity mapping" with **read and write** (RW) permission (meaning that GPA X is

mapped to HPA X).

Interaction with Windows, and the bug

In some buggy machines, consideration #1 was not happening, meaning that neither the HV nor the Secure Kernel know about this memory range from the UEFI memory map.

When booting, the Hypervisor initializes its internal state, creates the Root partition (again, details are in the Windows Internals book) and performs the IOMMU initialization in multiple phases. On AMD64 machines, one of these phases requires parsing the RMRR. Note that the HV still has *no idea* whether the system will enable VBS/HVCI or not, so it has no options other than applying the full identity mapping to the range (which implies RWX protection).

When the Secure Kernel later starts and determines that HVCI should be enabled, it will set the new "default VTL permission" to be RW (but *not* Execute) and will inform the hypervisor by setting the public HvRegisterVsmPartitionConfig synthetic MSR (documented in the Hypervisor TLFS). When VTL 1 of the target partition sets the default VTL protection and writes to the HvRegisterVsmPartitionConfig MSR, it causes a VMEXIT to the Hypervisor, which cycles between each valid Guest physical frame described in the **UEFI memory map** and mapped in the VTL 0 SLAT, removing the "Execute" permission bit (as dictated by the "DefaultVtlProtectionMask" field of the synthetic register).

Mindful readers can already understand what is going wrong here. In buggy firmware, where the RMRR is *not* set in the UEFI memory map, leaves the "Execute" protection of the described region on, producing a HVCI violation (thanks Satoshi).

Fixes

MSFT has fixed (thanks Andrea) the issue working on two separate sides:

1. Fixing the firmware in all the commercial devices MSFT released, forcing the RMRR memory region to be included in the UEFI memory map
2. Implementing a trick in the HV. Since the architecture requires that the RMRR memory region must be mapped in the IOMMU (via the Hierarchical Address translation structures as described above) using identity map with RW access permission (but no X - Execute), we decided to perform some compatibility tests and see what happen if the HV protects all the initial PFNs for RMRR memory regions in the SLAT by stripping the X bit. Indeed, the OS always needs to read or write to those regions, so programming the SLAT is needed.

Tests for fix 2 worked and produced almost 0 compatibility issue, so MSFT decided also to increase the protection and remove the X permission on all RMRR memory region by default on ALL systems, also increasing the protection when the firmware is bugged.

Summary

Hope you enjoyed this jointly written post with both bug reporter's and developer's perspectives and a great deal of details on the interaction of VT-d and Hyper-V by Andrea.

To summarize, the combination of buggy UEFI that did not follow one of the requirements by the Intel VT-d specification and permissive default EPT configuration caused unintended RWX GPAs under HVCI. MSFT resolved the issue by correcting the default permission and their UEFI and released the fix on January 9. Not all devices are vulnerable to this issue. However, you may identify vulnerable devices by checking the `memmap` UEFI shell command not showing the exact RMRR memory regions as Reserved.

Satoshi's notes

Satoshi's notes
tanda.sat@gmail.com

tandasat
standa_t

Thoughts and notes about platform security, reverse engineering, system programming and other low-level stuff.