HyperDbg Debugger

A debugger designed for analyzing, fuzzing and reversing

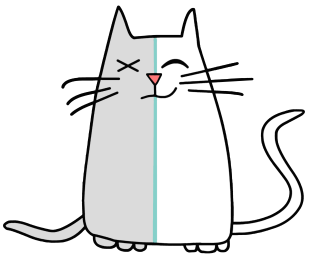
version 0.2.0.0

Chasing Bugs with/in Hypervisors

Author
Sina Karvandi

Research
https://research.hyperdbg.org

April 16, 2023



ZeroCon

Chasing Bugs with/in Hypervisors

This article discusses HyperDbg and its associated methods for compromising hypervisors, which were originally presented at Zer0Con 2023 (`https://zer0con.org`).

---

1 Abstract

Hypervisors are an indispensable component of contemporary software systems. While the primary purpose of hypervisors is to virtualize the system resources. There are various other applications for hypervisors besides their conventional use, and our focus lies in employing them for security and reverse engineering purposes; thus, this presentation is divided into two parts. The first part is about how hypervisors and solutions derived from hypervisors can help us in finding bugs in kernel-mode and user-mode routines as well as discussing the possibilities of using hypervisor debuggers in reverse engineering. The second part is about finding different types of bugs within the hypervisors (type 1 and type 2).

The study involves using various bug-finding techniques, including static analysis, dynamic analysis, and finding attack vectors, to identify vulnerabilities in both types of hypervisors. The presentation also highlights the importance of bug finding in hypervisors and the potential consequences of leaving vulnerabilities unaddressed. The findings can inform developers and security professionals in their efforts to improve the security of hypervisors and mitigate the risks associated with virtualization.

2 Introduction

A hypervisor (also known as a virtual machine monitor or VMM) is software that makes virtualization possible by virtually sharing the resources, such as memory and processor. It abstracts guest machines and the operating system from the actual hardware and runs virtual machines (VMs). There are two types of hypervisors. Type 1 are those hypervisors that runs before operating system and manage the entire system. E.g., KVM, VMware ESXi, Hyper-V, Xen. Type 2 are those hypervisors that run after the operating system and virtualize the virtual machines. E.g., VMware Workstation (and player), VirtualBox, Hyper-V, Parallels Desktop.

Type 2 hypervisors are also used for different purposes, with different ones being designed for different requirements. For instance, there are currently hypervisors created by Kaspersky, Avast, and several game cheating protection providers that serve the purpose of integrity, and security, while also providing entertainment value.

In this article, we are going to learn how we can use hypervisors for our security research and ease reverse engineering as well as finding the caveats within the design of common hypervisors.

First, we introduce some of the possible scenarios in which a hypervisor will be facilitative for reverse engineering and creating security solutions. Then we dig into the design of hypervisors to reveal attacks against hypervisors.

3 Hypervisors, Reversing, and Bug Finding

Hypervisors are able to fully control the system including the user-mode applications and kernel-mode operating systems, thus, it is a place of interest for security researchers. The most incredible part of a hypervisor is its ability to virtualize the memory (RAM) by utilizing EPT (Extended Page Table) tables. A researcher with transparency concerns can easily use EPT to monitor the execution of memory pages without modifying anything in the user-mode module or kernel codes.

A vast number of security bugs are caused by handling memory improperly which EPT can be employed to ease the discovery of them. Furthermore, it provides a valuable opportunity to conduct reverse engineering

and analyze binary modules due to its advanced monitoring capabilities. HyperDbg Debugger [2] is an example of a hypervisor-assisted tool that tries to export these hypervisor features in a way that is suitable for reversing and security research.

4 HyperDbg Debugger

HyperDbg Debugger [4] is an open-source [3], hypervisor-assisted, user-mode, and kernel-mode Windows debugger with a focus on using hypervisors for debugging, and reversing.

HyperDbg is a powerful debugger that offers several unique features, such as hidden hooks that are both fast and stealthy. This is achieved by mimicking hardware debug registers, which allow for reading and writing to specific locations without being detected by either the Windows kernel or programs. Additionally, it can measure code coverage and monitor all MOV(s) to and from memory by a function.

However, what makes HyperDbg particularly unique is its ability to maintain its stealthiness. It does not rely on traditional debugging APIs, which means that it is not detectable by typical anti-debugging methods. Additionally, it is resistant to time delta exploitation methods such as RDTSC/RDTSCP, making it extremely challenging for malware, anti-cheat engines, protectors, and other applications to discover the debugger.

4.1 Transparency

HyperDbg is transparent by its nature. It also provides other methods to make it even more transparent. For example, it tries to hide itself from being detected by timing attacks against hypervisors.

4.2 EPT and SYSCALL Hooks

Besides having classic EPT hooks, HyperDbg has the implementation of hidden in-line hooks. These hooks are substantially faster than classic EPT hooks as they won't cause VM-exits.

4.3 I/O Debugging

You can debug an unlimited number of Port Mapped I/O (PMIO) and Memory Mapped I/O (MMIO) ports.

4.4 Monitoring Memory or Emulating Hardware Debug Registers

Using this method, one can create a log from the "MOV" instructions that a function is performed during its execution. Thus, one can monitor the different places where a function tries to read or write. You've probably encountered the limitations of having only four Hardware Debug Registers. In HyperDbg, this limitation is removed by simulating debug registers using Intel EPT.

4.5 Directly Stepping from User-mode to Kernel-Mode

HyperDbg is the only debugger that can trace instructions from user-mode to kernel-mode and kernel-mode to user-mode. For example, when you press step after a SYSCALL instruction, then the next instruction is in the kernel mode, or if the debuggee executes an IRET or SYSRET instruction, then the execution is traced to the user-mode. It also guarantees that won't run (continue) the entire system while the user is stepping. There is also another implementation (command) for stepping like regular debuggers, but the user might choose to avoid running other cores, processes, or threads while is stepping through the instructions in both user-mode and kernel-mode.

4.6 A Fast Scripting Interface

This is what makes HyperDbg a different debugger. Everything is done in the kernel, and there is no break and return to the debugger; thus, its scripting engine substantially faster than WinDbg.

5 Hypervisor Vulnerabilities

Hypervisors are widely used in both security research as well as security products. However, implementing these monitoring features comes with plenty of compatibility issues. In many of these cases, developers of hypervisors may overlook design limitations to prevent compatibility issues. Of course, this approach comes with the price of rising new attack vectors. Here we categorize hypervisor attacks into two types. The first type is attacks that need physical access to the computer and the second type of attacks are those which can be carried even without physical access. Finally, we introduce a complicated type of attack in which the attacker forces the processor to bypass the security measurements made by the hypervisor developer to protect the hypervisor from tampering and present how we can extend this attack to entirely compromise the system.

6 Common Techniques with Physical Access

Here we introduce different techniques that need physical access to the computer or physical devices to perform the attack.

6.1 Bypassing Hypervisor with SMM Patches

System Management Mode or SMM (also known as ring -2) is an execution mode in Intel x86 processors that provides low-level system management functions which are used by firmware of devices. Whenever the processor loads the SMM firmware it sets a lock on the physical memory range that the firmware is loaded and the processor block all of the memory accesses to this range of memory where the SMM module is loaded. In order to run your code in SMM, you can either change the SMM firmware, or just patch the SMM firmware to avoid setting the protection bit, so you can view and modify the memory at the target address.

Most of the time, it needs physical access to the system. Microsoft made some protections [5] for checking the integrity of the system, however, once you have physical access to the target machine, you can change these protections and also an SPI programmer can be used to directly modify the ROM memory. A practical example would be to bypass *SMM_CODE_CHK_EN* as explained at [1].

6.2 DMA Attacks

There are other possible scenarios when you have physical access to the computer. PCILeech is your friend [6]. You can use PCILeech to perform Direct Memory Access (DMA) attacks.

All of these techniques need special devices and physical access to the computer. For the SMM you need an SPI programmer to interact with the memory chip, and for the second technique, you need a device (e.g., an expensive FPGA) to perform DMA attacks.

Now, let's investigate other possible scenarios which don't need physical access (devices).

7 Techniques Based on Design Flaws

Let's start with techniques, based on hypervisor design flaws.

7.1 Using an IDT Accessible from Kernel Pages

By default, a hypervisor should use a separate Interrupt Descriptor Table (IDT) for the VM-root mode. If not? We can use NMIs to exploit these hypervisors.

NMIs are typically indicative of hardware errors. However, with the use of event injection in the vmx-root, it is possible to virtually introduce them to the guest or even create them from the vmx non-root using the *x2APIC* or *XAPIC* (older) ICR register. In the vmx-root, interrupts are disabled due to the *RFLAGS.IF* flag, but this flag does not affect NMIs. Thus, if an NMI occurs while in the vmx-root, the NMI handler is called with the vmx-root permission and can disable the hypervisor.

Windows has a function called *KeRegisterNmiCallback*. It allows one to register a callback that will be executed in the event of an NMI. If the guest can be forced unconditionally to go on vmx-root (for example, using the CPUID instruction) and an NMI is fired simultaneously, the callback will be executed in vmx-root. Using a separate *HOST_IDTR_BASE* (not the same as *GUEST_IDTR_BASE*) will solve this problem, but it is crusial to ensure that the *HOST_IDTR_BASE* is not accessible from vmx non-root, or it will be bypassed. Let's elabortate it.

Most of the time, pass-through hypervisors, use a different *HOST_IDT* for the hypervisor and handle interrupts by themselves. But, is it safe? It depends. If the page table where *HOST_IDT* is located in the memory where a kernel-mode code can modify this page, then it could easily modify the IDT table, change the entry related to the NMI handler, and in the NMI handler, deactivate the hypervisor.

What's the solution to this attack? These attacks are typically prevented by protecting the page table related to *HOST_IDT* by using EPT protection bits. If a kernel-mode (or user-mode) code wants to modify a physical address (and of course a virtual address) of the memory where *HOST_IDT* is located, then an *#EPT* Violation will happen and the hypervisor is notified about this modification and could easily prevent the modification by ignoring the memory writes.

Is the hypervisor still safe? No, let's see other attack vectors in which we can change the hypervisor developers' assumptions to attack the hypervisor.

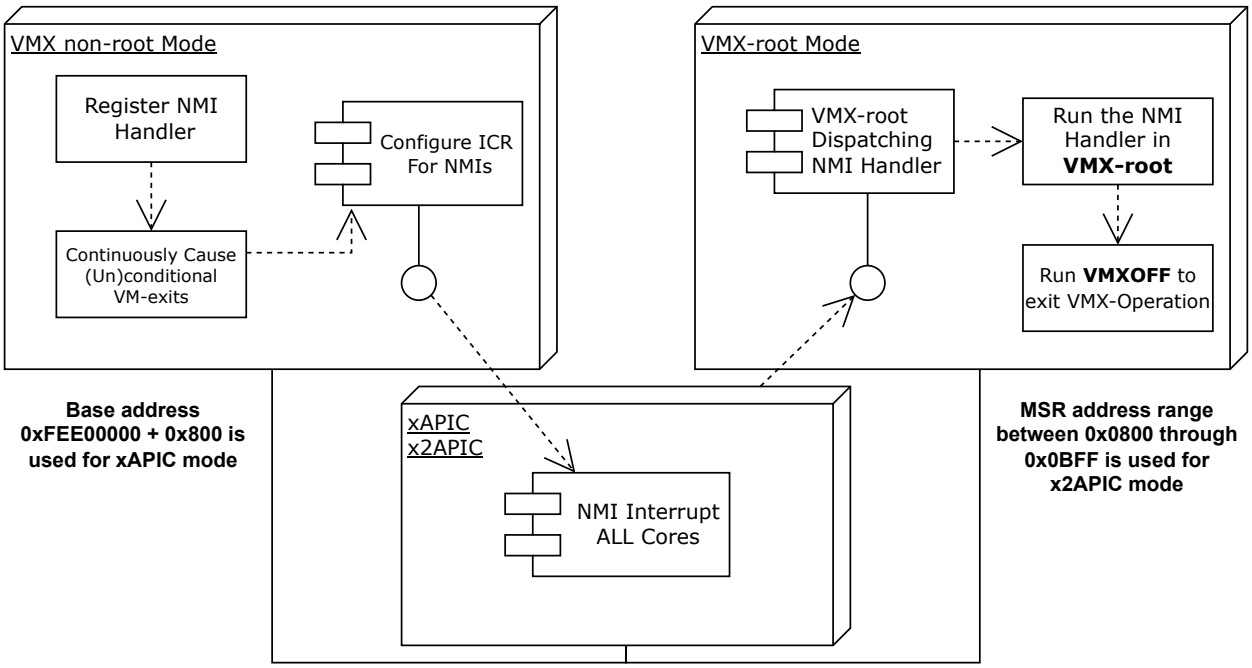


Figure 1: Registering NMI Handler in VMX-root Mode

7.2 Changing Control Registers

In the x86 processor, control registers are special-purpose registers that are used to control the operation, and enable/disable features of the processor. Intel processors have several control registers that are used for various purposes. For example, *CR0*, *CR2*, *CR3*, *CR4*, *CR8*.

7.3 Using the Same Page Table

In most of the security or commercial products where hypervisors are used as a solution, they use the same allocation routines that are used by Windows. After allocating memory, they try to protect the physical address from being modified by kernel-mode and user-mode codes. In those cases, it is not simply possible

to protect the kernel-level page table itself, because the page tables are accessed by different parts of the Windows memory manager and to avoid messing with the Windows memory manager, they won't protect the page table itself. This can leave an attack vector, where the attacker is able to modify the target page tables' physical address and map another physical address instead of a critical page table of the target hypervisor.

If we can simply change the *HOST_IDT*'s page table, then we could easily compromise the entire system.

Some hypervisors try to protect the page tables, by only allowing some special operations on the page-table and disallowing other operations (like modifying the page-table physical address or write enable bits).

Let's find another way of compromising these types of hypervisors!

7.4 CPU Features to Rescue

There are plenty of processor features, in which the logical processor will write into the memory. The problem comes from the fact that the MSRs should not be directly modified from the hypervisor. If the non-privileged code wants to modify critical hypervisor structures, codes, and protections, then it should just configure the Model Specific Registers (MSRs) related to that feature and after that, it easily manipulates the hypervisor page tables. For example, Branch Trace Store (BTS) is a processor feature that might be used for this purpose.

One might ask, okay, we'll blacklist this MSR from the hypervisor by avoiding running WRMSR on this MSR. But there are tons of these MSRs available on the system. We can easily find other features that can be used for the same purpose and this way, we could compromise the hypervisor.

7.5 What's Next?

You may consider where to proceed from this point? After corrupting a critical hypervisor structure, we have several options here that will lead to fully compromising the hypervisor. A general approach will be to set the memory write bit in the EPT page-table. Once we have access to modify the hypervisor codes, we can change the hypervisor code to run the *VMXOFF* instruction in VMX-root mode. Note that, we have to gracefully return the system state so, we won't end up in a kernel crash. Once the *VMXOFF* instruction is executed, the hypervisor is turned off (disabled), and the system is no longer on VMX Operation. Also, an attacker may fully compromise the system by utilizing techniques such as bypassing certain checks within the hypervisor or modifying the Virtual Machine Control Structure (VMCS) structure.

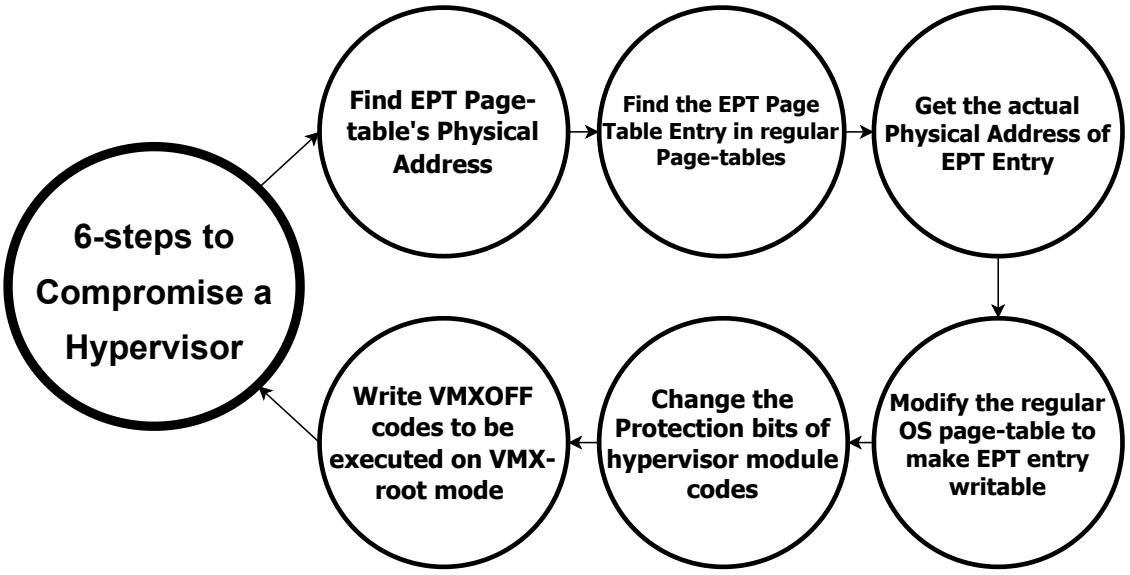


Figure 2: Six Steps to Compromise Hypervisors

8 Why These Bugs, Are NOT Easy to Fix?

The worst thing about these bugs comes from the fact that these bugs are not easily fixable. Implementing an entirely independent memory manager for type 2 hypervisors needs a significant code base that handles every condition and manages the entire memory. Due to the complexity of implementing a memory manager for the entire system and dozens of compatibility issues, most of the custom type 2 hypervisors prefer to skip the memory manager and use the Windows (or Linux) memory manager for allocating critical structures and code base of the hypervisor. This brings a gap between security and compatibility where an attacker can easily manipulate the memory manager of the operating system and gain privilege over the hypervisor. On the other hand, simple tricks to avoid manipulating page tables are proved to be ineffective as the operating systems might change the behavior of their memory manager in different versions (or even different updates) and it leaves the hypervisor incompatible with OS changes and ends up a kernel panic or a blue screen of death (BSOD).

On the other hand, fixing the above issue might not solve all problems as there are plenty of features in the processor that are not known to the hypervisor developer (or even, Intel might (and will) add new features to the newer generation of its processors that will make a new bypass for the previously developed hypervisors.

One might think that block listing all MSRs and only allowing some of them might be the solution, but it's also not the best case as it causes lots of compatibility issues with different drivers that try to use a special processor feature and even the operating system itself might use some different MSR registers that are black-listed in the hypervisor and eventually cause trouble for the entire system.

Because of that, most of the hypervisor developers (especially type 2), try to pass MSRs directly to the processor (or to the top-level hypervisor) and this brings a big hole in the security of the solutions based on hypervisors.

9 Conclusion

In conclusion, hypervisors play a crucial role in virtualizing system resources and have become an indispensable component of modern software systems. Hypervisors not only have conventional use but also have applications in reverse engineering and security. This article discussed the two main parts of using hypervisors for bug finding and identifying different types of bugs within hypervisors. It also highlighted the importance of bug finding in hypervisors and the potential risks associated with leaving vulnerabilities unaddressed.

References

[1] *CODE CHECK(MATE) IN SMM*. https://www.synacktiv.com/en/publications/code-checkmate-in-smm.html. 2018.

[2] *Hyperdbg.org*. https://docs.hyperdbg.org/. 2023.

[3] *HyperDbg/HyperDbg: State-of-the-art native debugging tool*. https://github.com/HyperDbg/HyperDbg. 2023.

[4] Mohammad Sina Karvandi et al. "HyperDbg: Reinventing Hardware-Assisted Debugging". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 1709–1723.

[5] *System Guard Secure Launch and SMM protection (Windows 10)*. https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-system-guard/system-guard-secure-launch-and-smm-protection. Feb. 2023.

[6] *ufrisk/pcileech: direct memory access (dma) attack software*. https://github.com/ufrisk/pcileech. 2023.