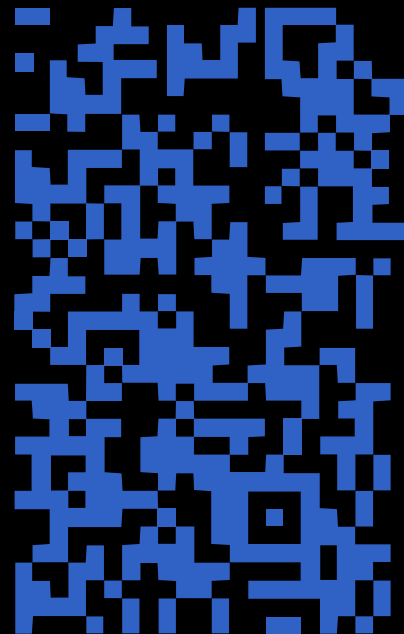




Chasing **Bugs** with/in Hypervisors

ZEROCON





Who am I?

Sina Karvandi [Sinaei]

Chosun University

Windows Internals enthusiast, Interested in microarchitecture, digital design and low-level programming...

- A developer of **HyperDbg** Debugger



<https://rayanfam.com>



sina@hyperdbg.org



[@Intel80x86](https://twitter.com/Intel80x86)

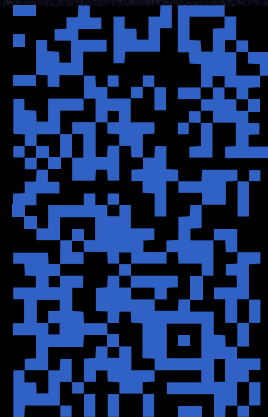
Outline

- Story Time
- Hypervisors & Reverse Engineering
- Hypervisor Vulnerabilities
- Common Techniques with Physical Access
- Design Flaws
- Conclusion



01

Story Time



Motivation

Why hypervisors?



Reverse Eng. & Bug Finding

As hypervisors provide a higher privilege level and control over the main components of the system, it is actively used for reverse engineering and bug finding



Industry

Hypervisors are utilized in the component of Digital Rights Management (DRM) and Anti-tampering systems as well anti-viruses like **Kaspersky**, Avast, and several game cheating protection providers



Virtualization Solutions

Hypervisors are often the focus of security research due to their widespread use in cloud servers and desktop solutions, including **Virtualization Based Security (VBS)** and **Hyper-V** API-based products

Background (terms)

01

Rings

Different levels of execution
in x86 processors

02

Hypervisor [VMM]

Virtual Machine Monitor
(VMM), manages the
virtualized cores

03

Types

Two types of
hypervisor. Type-1 and
Type-2 hypervisors

04

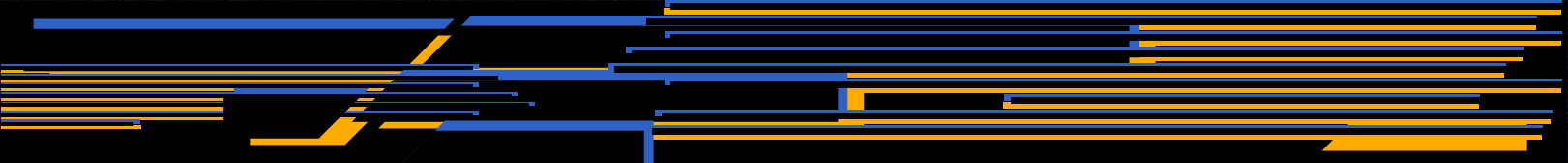
Modes

Two execution modes,
VMX-root Mode,
VMX non-root Mode

05


Nested Paging

Second Level Address Translation (SLAT)
implementation in x64 processors, EPT
(Extended Page Table)



Hypervisors & Reverse Engineering

02



Control Over Memory

A key aspect of hypervisors is their ability to control memory, which empowers control over program flow and enables analysis of memory accesses made by programs in both kernel-mode and user-mode.

Extended Page Table [EPT]

This control over memory is a fundamental feature of **Second Level Address Translation (SLAT)**, allowing for powerful monitoring and analysis of program behavior at the memory level.

Based on these two main demands, HyperDbg debugger is made.

Reason 1

First, we needed to control memory which was later implemented as different types of EPT hook.

Reason 2

We needed to find a way of connecting user-mode to kernel-mode and kernel-mode to user-mode in order to ease the reverse engineering.

HyperDbg Debugger

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.

See !measure and !hide commands.

I/O Debugging

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

See !ioin and !ioout commands.

For more info:

<https://github.com/HyperDbg/HyperDbg>

HyperDbg Hooks

!epthook/!epthook2

Hidden hooks

!syscall/!sysret

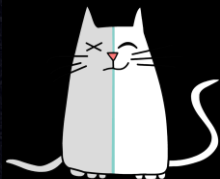
System-call hooks

!monitor

Hardware Debug Register Emulation

EPT 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.



System-Call Hooks

Both **SYSCALL** and **SYSRET** instructions can be hook in HyperDbg.

Debug Registers Emulation

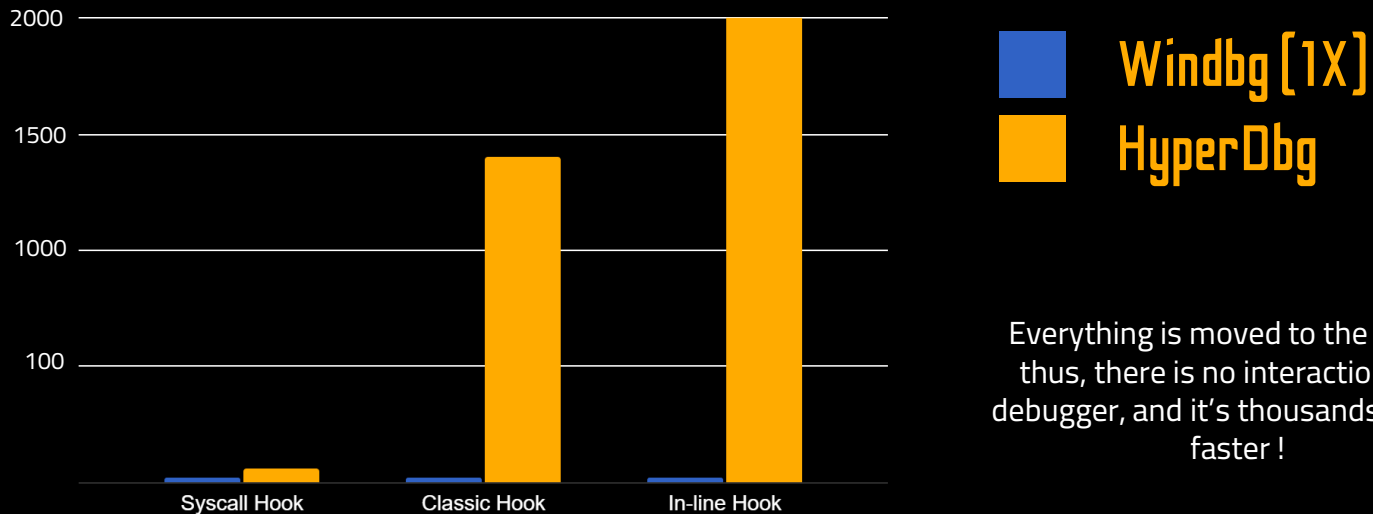
Using this method, one can create a log from the "**MOV**" instructions that a function is performed during its execution. Thus, one can monitor the different places where a function tries to read or write.

Limitless Emulation

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.

WinDbg vs. HyperDbg

Speedup (X)



Everything is moved to the kernel,
thus, there is no interaction with
debugger, and it's thousands of time
faster !

HyperDbg is **2.98** (SYSCALL hooks), **1319** (classic hidden EPT hooks), and **2018** (in-line EPT hooks) times faster!

HyperDbg



hundreds of breakpoints
tracing from user mode to kernel mode
I/O Debugging
transparent
open-source and community aware

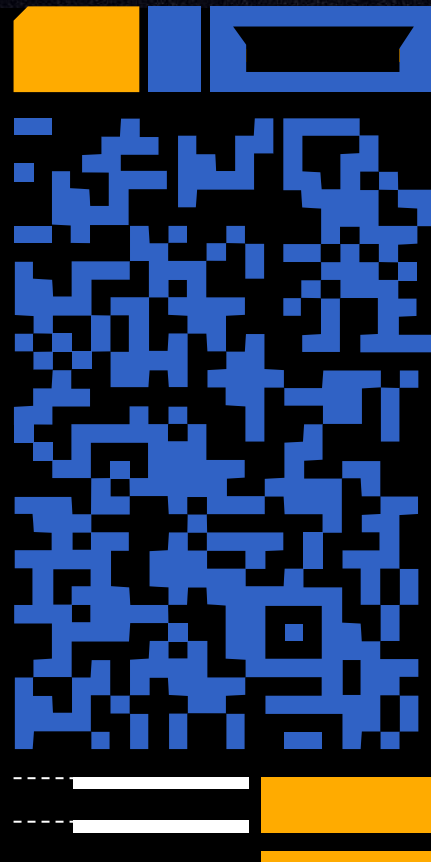
WinDbg



only one breakpoint can halt the system
very basic stepping
not open-source but the source code
leaked multiple times, waah!
what's transparency?

Directly Stepping from User-mode to Kernel-Mode

- You can trace instructions from user-mode to kernel-mode and kernel-mode to user-mode.
- Like after a **SYSCALL**, then the next instruction is in the kernel mode, or if the debuggee executes an **IRET** or **SYSRET** instruction, then the execution is in the user-mode.
- It also guarantees that won't run (continue) the entire system while the user is stepping.
- Have you ever visited **Heaven's Gate**?





03

Hypervisor Vulnerabilities





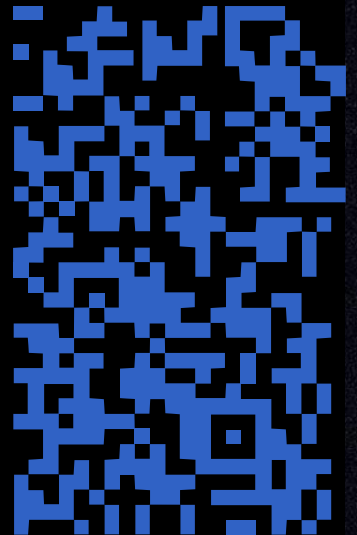
- After having looked at some examples of how hypervisors are used in reverse engineering, it's time to delve into the potential pitfalls and weaknesses associated with hypervisors.
- Let's see how we can compromise a hypervisor in two main categories.

1. **Common Techniques with Physical Access**
2. **Design Flaws**

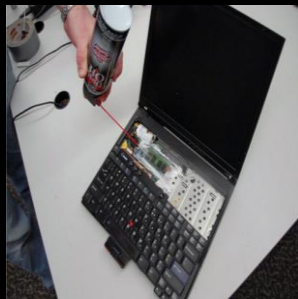


04

Common Techniques with Physical Access



How can we compromise a hypervisor with physical access?



Cold Boot Attack

Liquid nitrogen, freeze spray or compressed air cans

Cooling memory modules in order to slow down the degradation of volatile memory



DMA Attacks

An FPGA or other DMA capable devices

Using an FPGA along with **PCILeech** to perform Direct Memory Access attack



SMM Patches

An SPI Programmer

Using SPI programmer to patch the BIOS memory chip



Design Flaws 05

Techniques Based on Design Flaws

Now let get down to the business...

Assumptions

- It is presumed that the attacker has authority over the computer, such as root privileges in Linux or elevated admin credentials in Windows.
- Hypervisor attacks are platform (Window/Linux) independent.

Goal

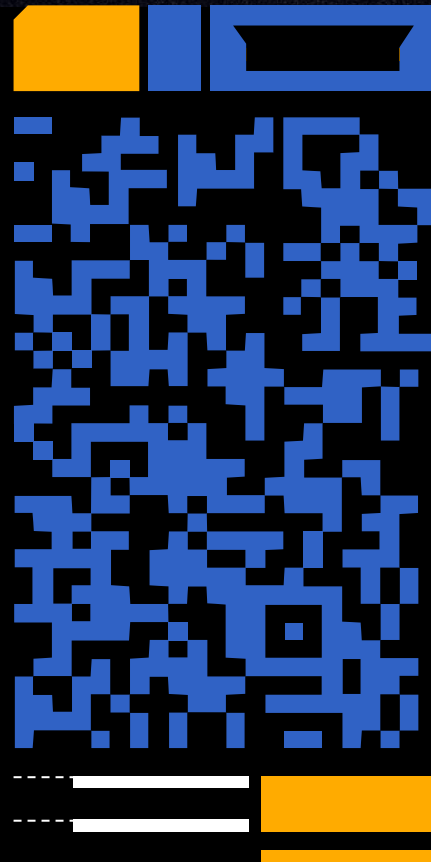
- Gain control over the top-level hypervisor.



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.
- Using a separate **HOST_IDTR_BASE** (not the same as **GUEST_IDTR_BASE**) will solve this problem.
- But, it is crucial to ensure that the **HOST_IDTR_BASE** is not accessible from VMX non-root, or it will be bypassed.

What are NMIs?



Non-Maskable Interrupt (NMI)

Hardware Error

NMIs are typically indicative of hardware errors



Cannot be Masked

NMIs cannot be masked, even when interrupts are disabled due to the **RFLAGS.IF** flag



Used by the OS

Windows/Linux use NMIs typically for watchdog purposes



VMX non-root

We can create NMIs from the VMX non-root using Interrupt Command Register (ICR) register (**x2APIC & xAPIC**)

The scenario (Attacking Hypervisor Using NMI)

- Windows has a function called *KeRegisterNmiCallback*. It allows us 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, and an **NMI** is fired simultaneously; the callback will be executed in VMX-root.
- As the callback function will be called in VMX-root mode, we could fully get the control over the hypervisor.



One more thing...

- There are some instructions that cause VM-exit unconditionally.
- Here's a list of these instructions:

<u>Regular Instructions</u>	CPUID, GETSEC, INVD, XSETBV, INVEPT, INVVPID
<u>VMX Instructions</u>	VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMRESUME, VMXOFF, and VMXON



One more thing...

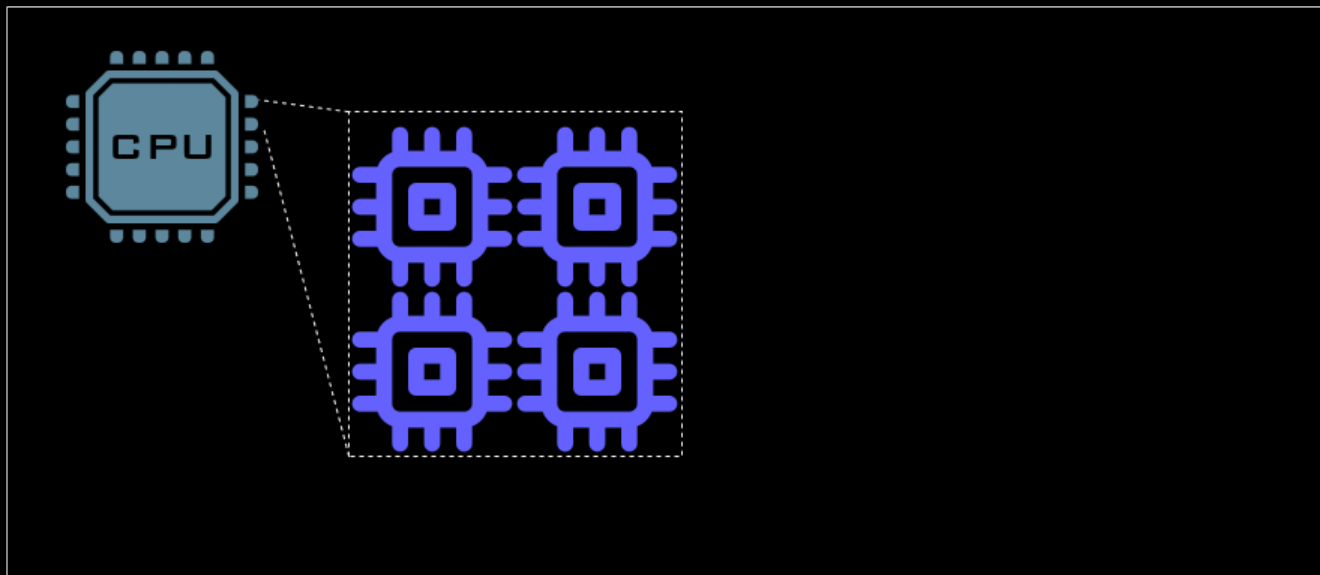
- There are some instructions that cause VM-exit conditionally.
- Some of which :

INVLPG	LMSW	RDTSC/P	MOVE to CR8
RDPMC	RSM	IN, OUT	ENCLS
INVPCID	MONITOR	CLTS	MOV DR
RDSEED	MOV from CR3	MOV to CR3	ENCLV
LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR	MOV from CR8	XRSTORS	MWAIT
	MOV CR0, CR3, CR4	XRSAVES	HLT



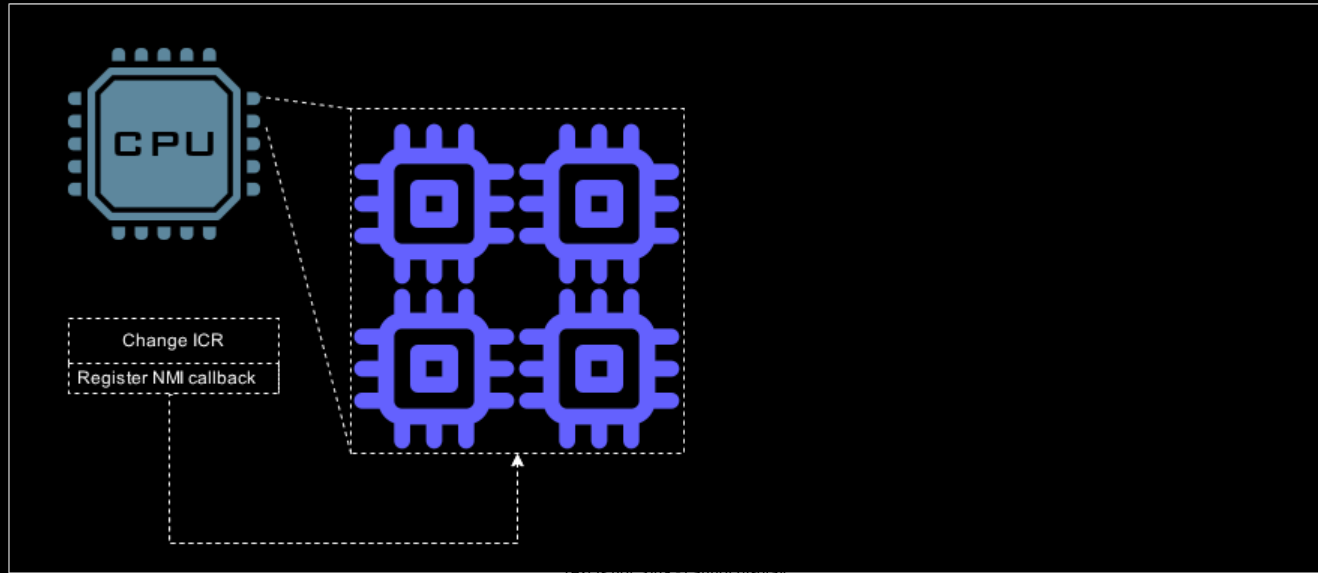
Attacking Hypervisor Using NMI

We need to control one core, to attack the neighbor cores.



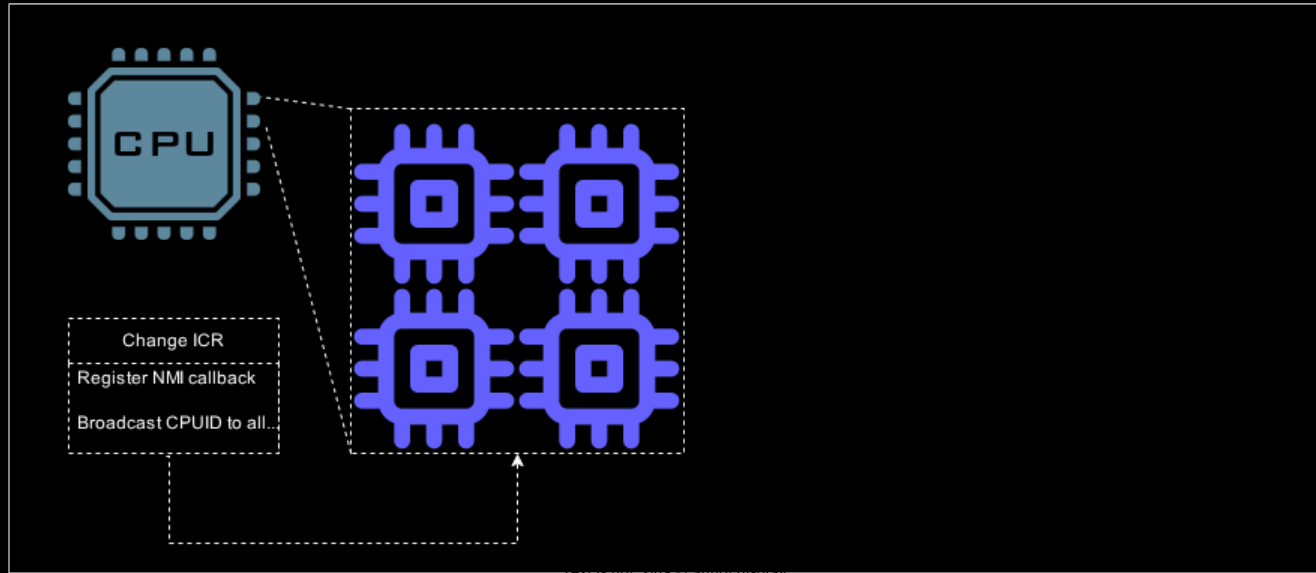
Attacking Hypervisor Using NMI

First of all, we register an NMI handler using *KeRegisterNmiCallback* function. The callback will point to an attacker controlled code.



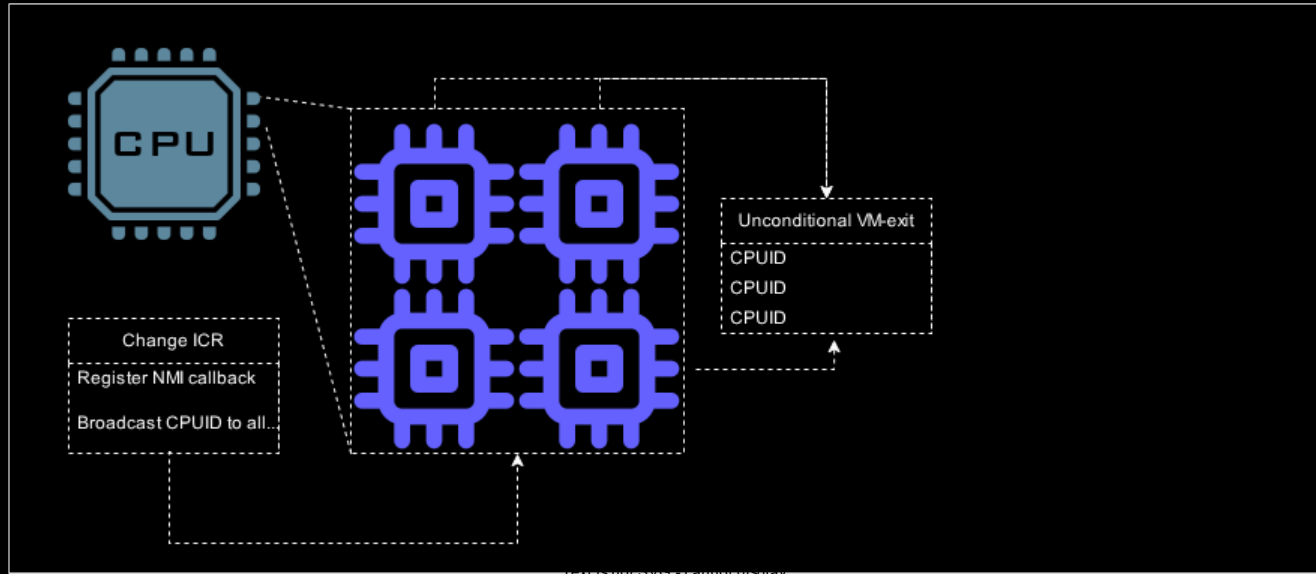
Attacking Hypervisor Using NMI

We need to control one core, in order to perform our attack.



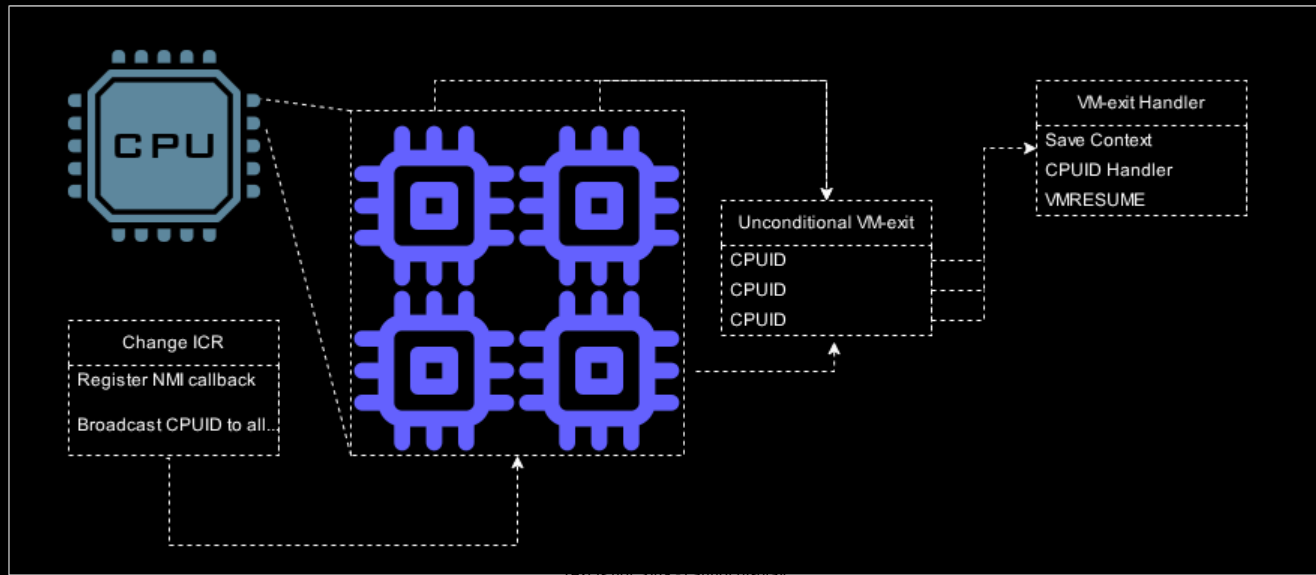
Attacking Hypervisor Using NMI

Then, we broadcast a function that consists of infinite loops of an instruction that triggers an unconditional VM-exit on all adjacent cores (E.g., by using **DPC**).



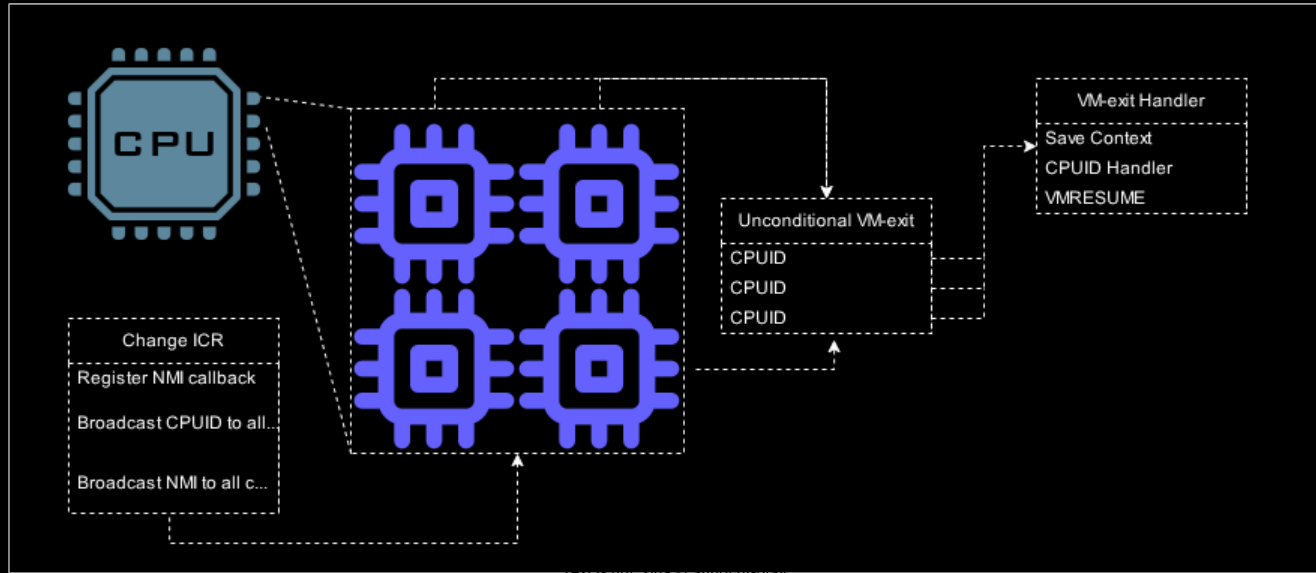
Attacking Hypervisor Using NMI

All the neighbor cores are forced with a VM-exit and the VM-exit handler is called in VMX-root mode.



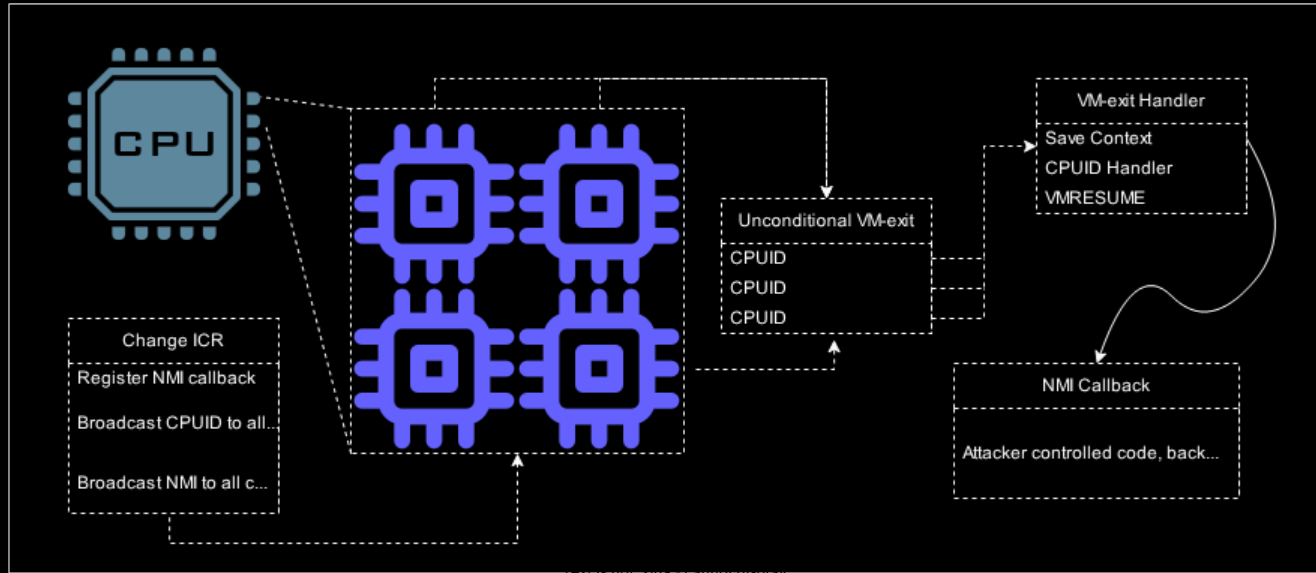
Attacking Hypervisor Using NMI

Now, we broadcast NMIs to the adjacent cores and hope that the NMI will be received while the core is operating in VMX-root mode.



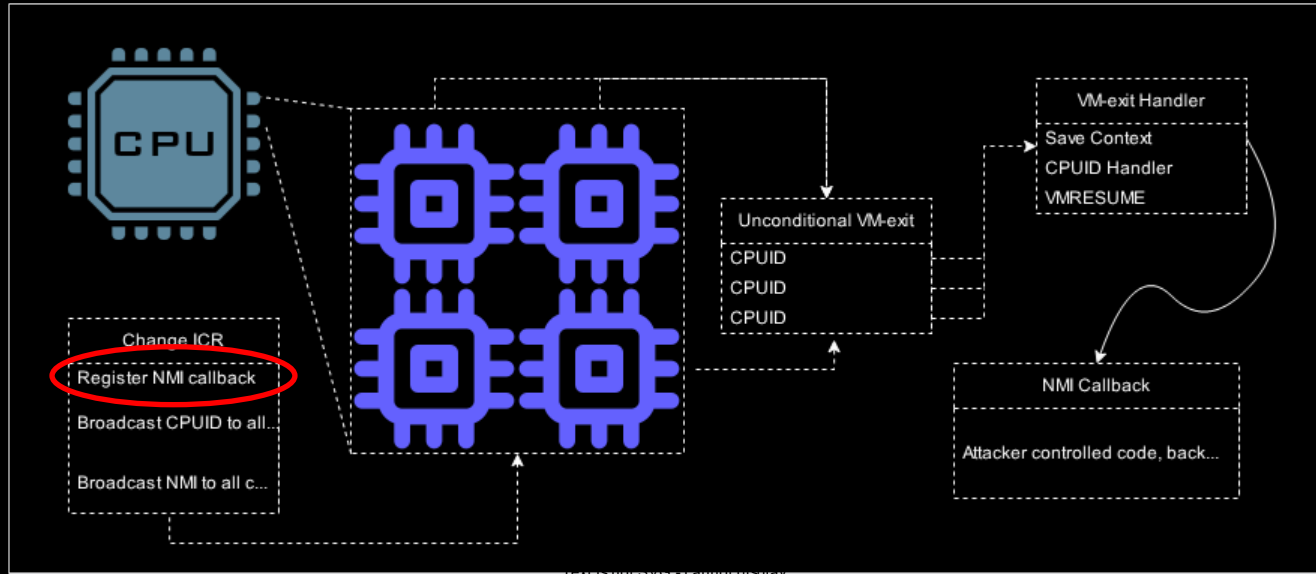
Attacking Hypervisor Using NMI

After trying multiple times (usually less than 5 NMI broadcasts), we'll end up running our codes in the privileged mode as the NMI callback that is previously registered is now called.



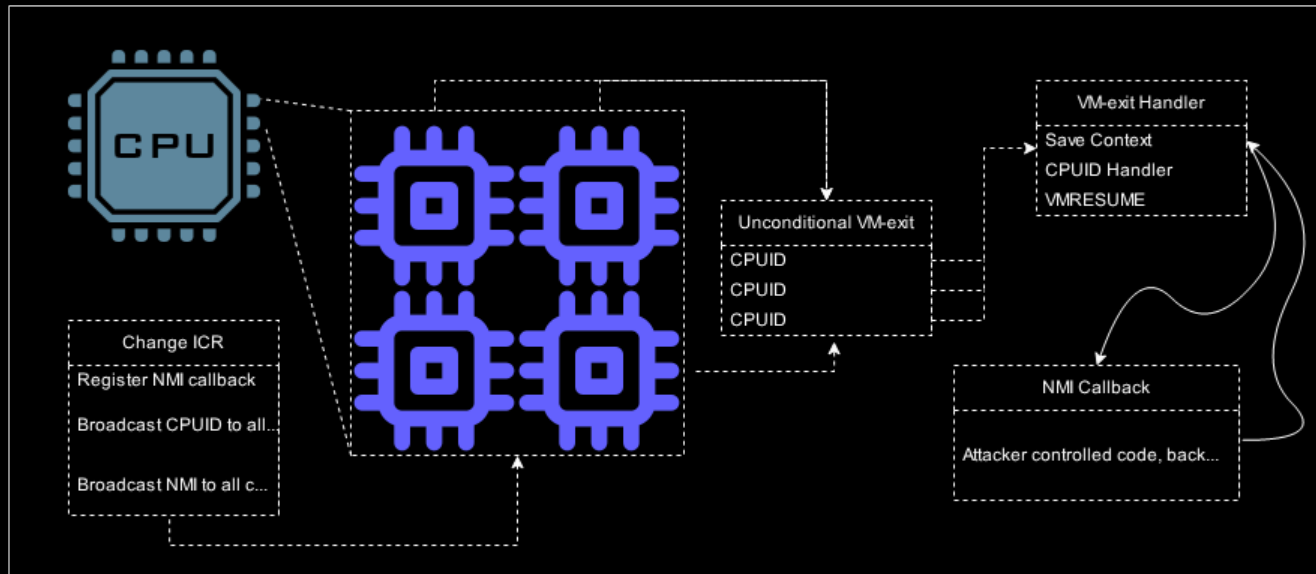
Attacking Hypervisor Using NMI

After trying multiple times (usually less than 5 NMI broadcasts), we'll end up running our codes in the privileged mode as the NMI callback that is previously registered is now called.



Attacking Hypervisor Using NMI

Now, it is important to return gracefully to the VMX-exit handler in order

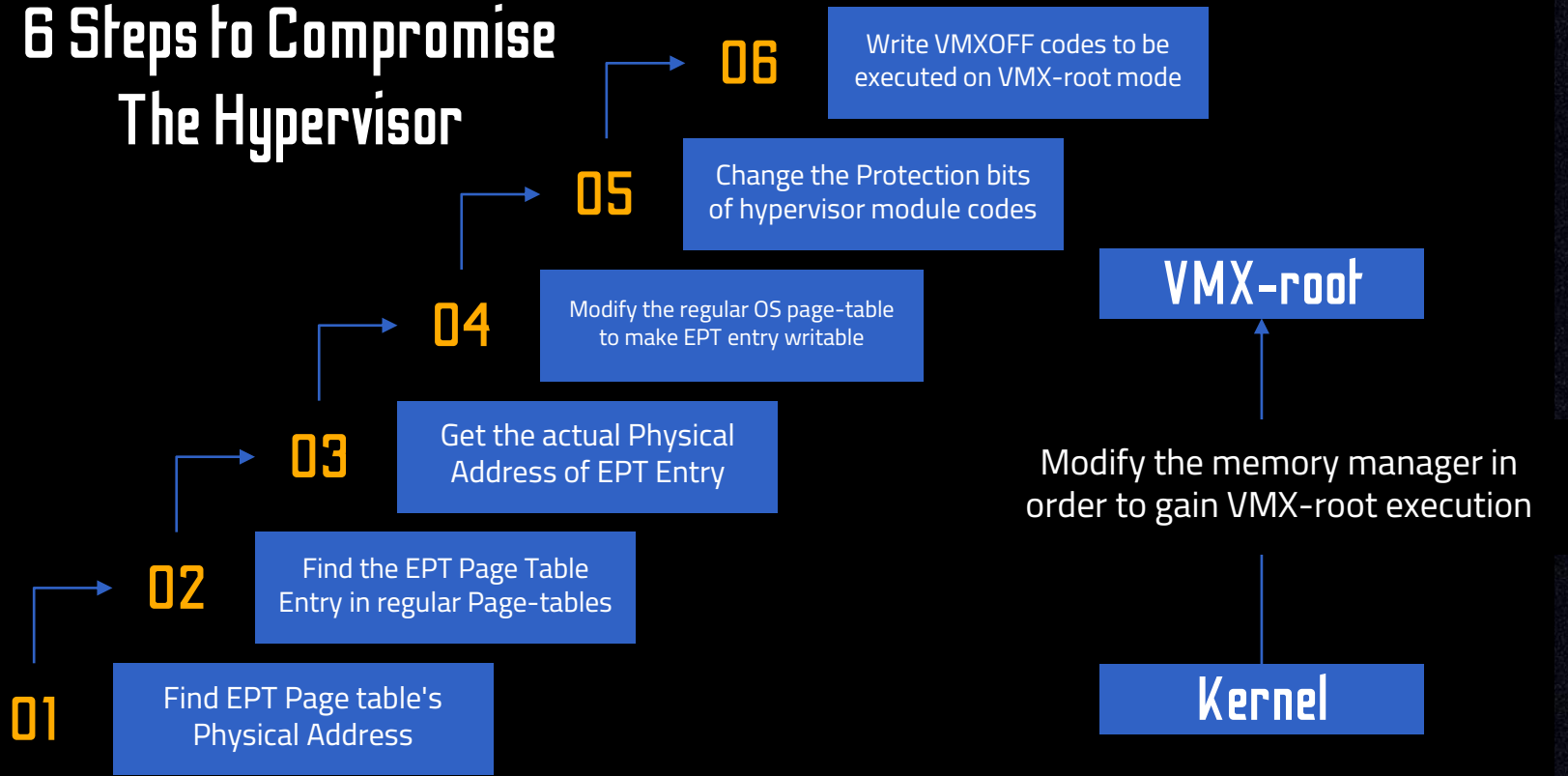


Altering Host-IDT Memory

- Let's use the previous attack as a starting point.
- There are numerous situations where we can challenge the assumptions made by the hypervisor designer, even if they have defined a distinct **HOST_IDT_BASE**.
- If the hypervisor and operating system share the same memory manager, which is quite common, what would happen?
- Yes; we could trick the memory manager to modify the memory where the **HOST_IDT_BASE** is located.



6 Steps to Compromise The Hypervisor



Mitigating Memory Attacks Using EPT

- What's the solution to this attack?
- These attacks are supposed to be prevented by protecting the page table related to **HOST_IDT_BASE** by using EPT protection bits.
- If a kernel-mode (or user-mode) code wants to modify a physical address of where **HOST_IDT_BASE** is located, then an **#EPT Violation** will happen
- The hypervisor is notified about this modification and could easily prevent the modification by ignoring the memory writes.
- Most of the time, left without protection!



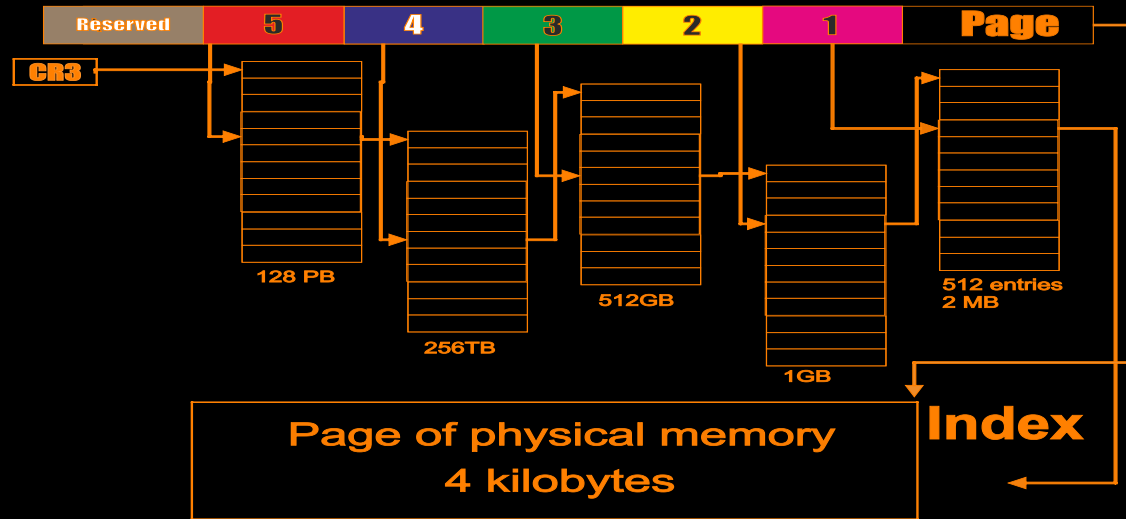
Using the Same Page Table

- Is the hypervisor safe now? No, let's see a new bypass!
- 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/Linux.
- After allocating memory, they try to protect the physical address from being modified by kernel-mode and user-mode codes.
- The problem here is that, it is not simply possible to protect the kernel-level page table itself!



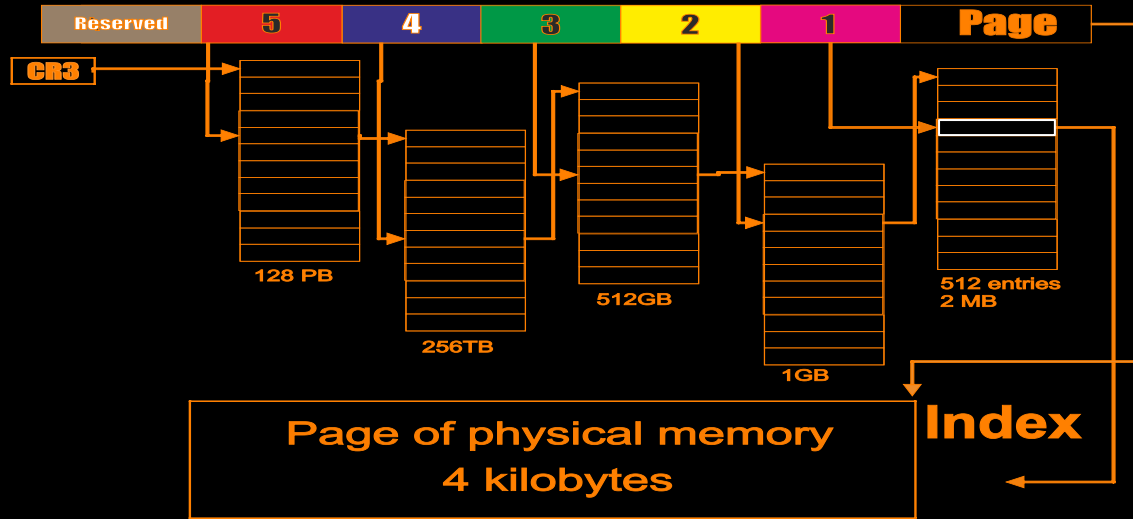
Protecting OS page-tables by utilizing EPT

A hypervisor designer might must protect the regular operating system page-table entries by using EPT page-table entries.



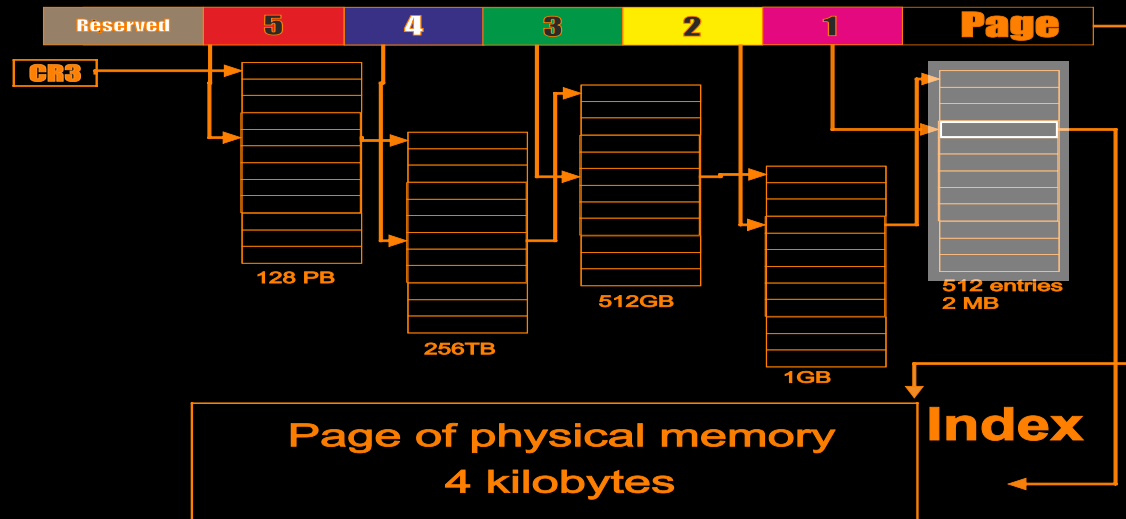
Protecting OS page-tables by utilizing EPT

It is not possible to protect a single entry in the **EPT (Extended Page Table)** since the minimum granularity for **EPT PTE (Page Table Entry)** at the last level is **4 kilobytes**.



Protecting OS page-tables by utilizing EPT

Protecting an entire page table will bring enormous challenges. The operating system never supposed that we're preventing it from modifying its page tables, thus it wants to freely alter it.



Protecting OS page-tables by utilizing EPT

- Even though it is possible, but lots of considerations should be made to protect a page-table.
- The hypervisor developer should verify that only benign OS memory manager codes are able to modify the page-table and it brings a new attack vector!
- It's a security with the cost of breaking stability.
- Assume that the protections to EPT are successfully implemented, **are we safe now?**



Protecting OS page-tables by utilizing EPT

- Even though it is possible, but lots of considerations should be made to protect a page-table.
- The hypervisor developer should verify that only benign OS memory manager codes are able to modify the page-table and it brings a new attack vector!
- It's a security with the cost of breaking stability.
- Assume that the protections to EPT are successfully implemented, **are we safe now?**

Of course, No. We always find ways through it :)





BTS To Rescue!

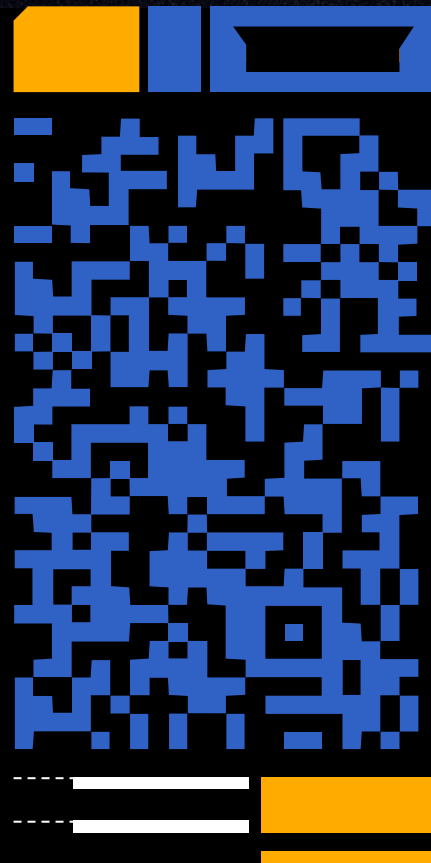
Here's how we can bypass the mitigation made by protecting OS page-tables by using EPT page-tables

Wait wait, Not this BTS, of course :D



Branch Trace Store (BTS)

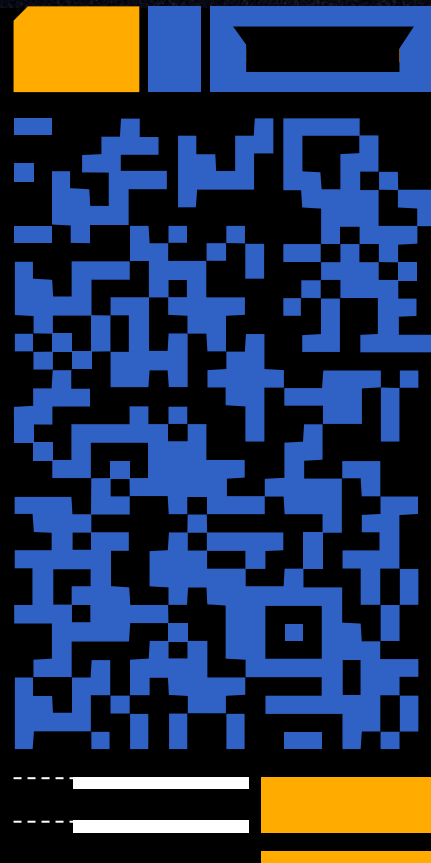
- **Branch Trace Store (BTS)** is a performance monitoring feature of Intel processors that enables tracing and profiling of software execution by collecting information about branch instructions.
- It is mainly designed to identify performance bottlenecks, debug issues, and optimize software.
- **BTS** works by capturing the addresses of branch instructions executed by the CPU and storing them in a circular buffer called the **BTS** buffer.



Branch Trace Store (BTS)

- **Branch Trace Store (BTS)** is a performance monitoring feature of Intel processors that enables tracing and profiling of software execution by collecting information about branch instructions.
- It is mainly designed to identify performance bottlenecks, debug issues, and optimize software.
- **BTS** works by capturing the addresses of branch instructions executed by the CPU and storing them in a circular buffer called the **BTS** buffer.

So, What?



Modifying Critical Hypervisor Buffers Using BTS

- As it's possible to specify the target buffer, the memory manager will be tricked into modifying critical hypervisor structures.
- For example, **Virtual Machine Control Structure (VMCS)** can be directly modified with using privileged **VMREAD/VMWRITE** instructions.
- As memory allocation branch addresses are also under the control of a kernel-mode module, the memory content will be customized and even the VT-x module can be modified directly.
- Can **MSR Bitmaps** solve this issue ?



Modifying Critical Hypervisor Buffers Using BTS

- As it's possible to specify the target buffer, the memory manager will be tricked into modifying critical hypervisor structures.
- For example, **Virtual Machine Control Structure (VMCS)** can be directly modified with using privileged **VMREAD/VMWRITE** instructions.
- As memory allocation branch addresses are also under the control of a kernel-mode module, the memory content will be customized and even the VT-x module can be modified directly.
- Can **MSR Bitmaps** solve this issue ? **NO, what do you do with other variants?**



Why These Bugs, Are NOT Easy to Fix?

These bugs are not easily fixable. Why?

Memory Manager

- An independent memory manager requires a significant code base.
- Due to the complexity of implementing a memory manager, most custom type 2 hypervisors **skip** it and use the **Windows** or **Linux** memory manager.
- **EPT (Access Bits)** tricks tables are ineffective because the OS wants to modify its page tables according to its own requirements (updates may change it).

Model-Specific Registers

- Keeping the compatibility while not over-limiting the guest.
- Whitelisting won't work. A Windows/Linux update may break the hypervisor.
- Blocklisting also won't work. The CPU is too complex to mitigate all MSRs. Also, in each new generation of processor, new features will be added.

Before Finishing...

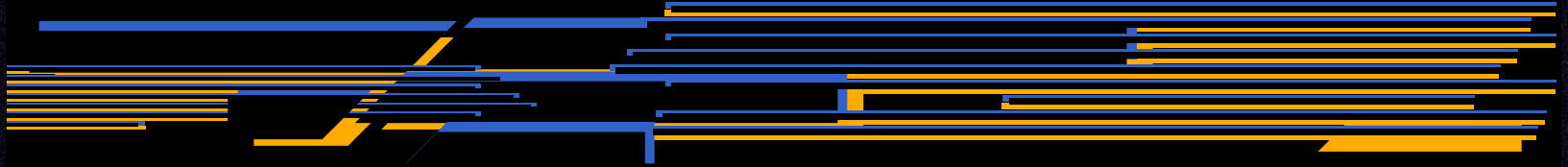
Conclusion

- Designing hypervisors is challenging, and even more difficult by the constant advancements in Intel processors with each generation.
- Relying on custom-made hypervisors can be problematic, as these hypervisors are often plagued with inherent design issues and should not be considered as a definitive security boundary.
- Use HyperDbg, it makes your life easier :)





Questions?





Thanks!

Feel free to reach me if you have any question!

ZEROCON

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon** and infographics & images by **Freepik**