>>> From the basement to the clouds
>>> The underpinnings of the internet

Name: Jacob Torrey[†]
Thinkst Labs (the canary folks)
Date: February 23, 2022

[†]jacob@thinkst.com (@JacobTorrey)

>>> TL;DR

Into the clouds

While modern software is increasingly SaaS-based, there are familiar features underpinning the cloud that are important to understand--both how they are similar to your PCs, and where they differ.

This lecture aims to provide a guide to these differences to accelerate researchers performing their research by reducing duplication of effort and provide a reference to the existing project landscape.

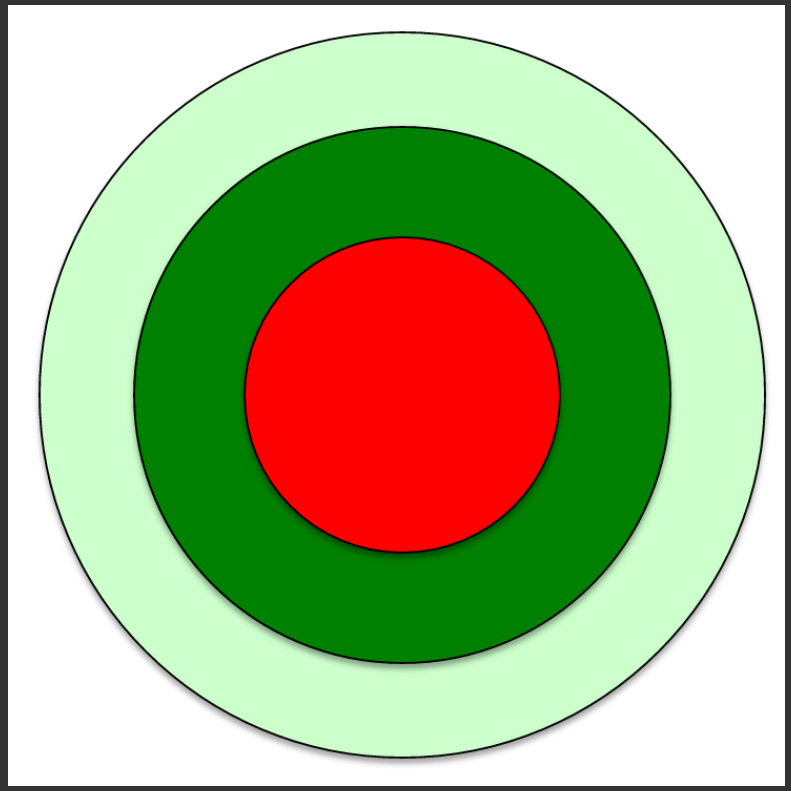
>>> Who am I?

* Head of Thinkst Labs
  * Research arm of Thinkst
  * Looking to collaborate with university students to make the internet a safer place
* Much of past research was in low-level x86 environments
* Spent some time securing the cloud @ AWS

>>> Outline

>>> Privilege rings in x86

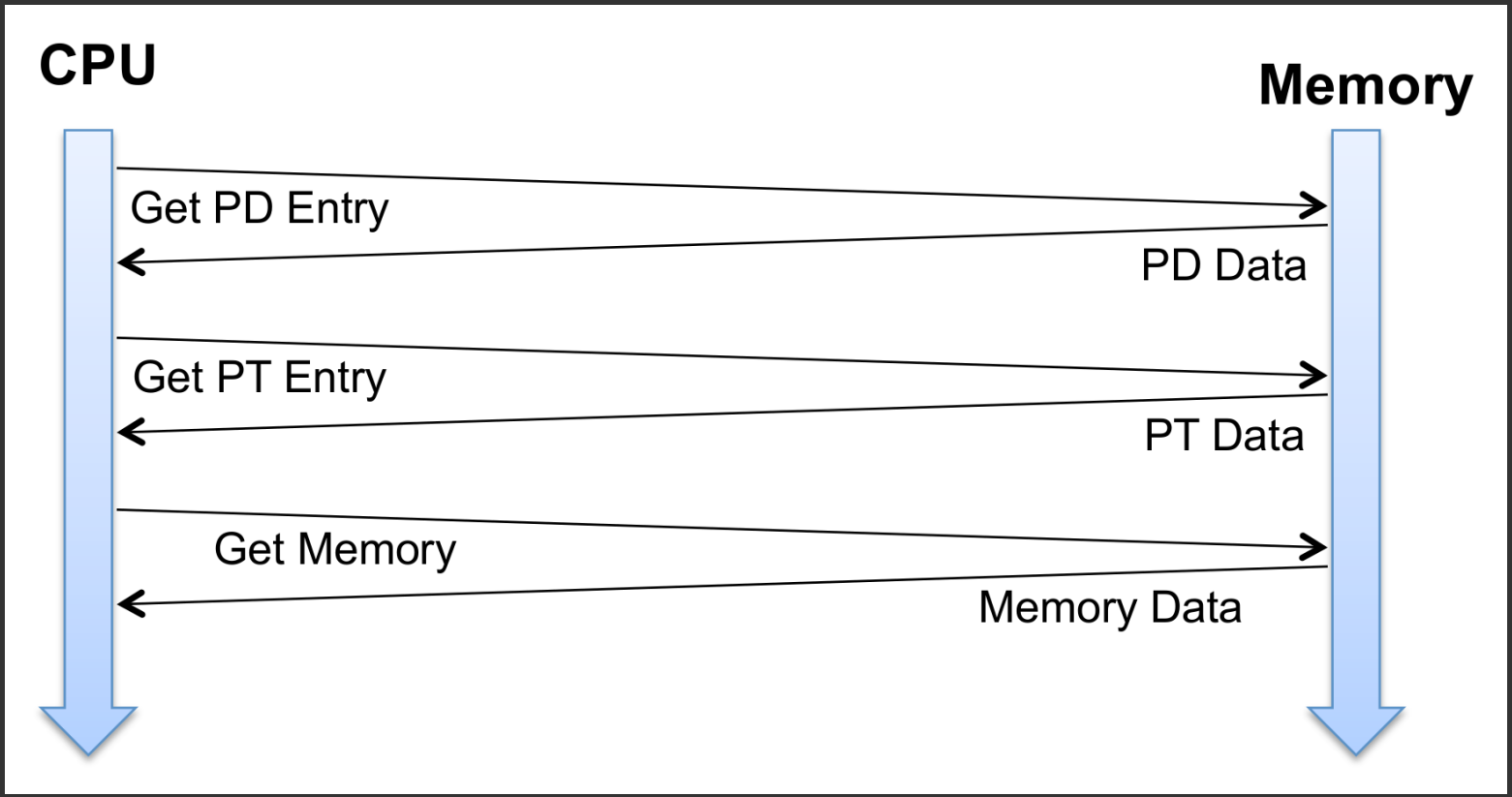
* Intel 386 added kernel protection and separation of processes; officially rings 0-3, unofficially -1, -2, & -3 (higher is less-privileged)
* Need to access to proper ring depending on what architectural features you require
* Once you know what level of access is needed, easier to pair with tool(s) to boot-strap research



* Typically, the *higher* the ring, the easier development is (e.g., much simpler to develop in user-space than risk a system crash in the kernel)

>>> Paging

* One of the most powerful features implemented in the 386 is paging and the concept of *virtual memory*
* Allows more privileged code to isolate and manage less privileged processes (e.g., OS multi-plexing applications or VMM managing OSes)



>>> Cache

* Memory access is slow, processors aim to cache as much as possible to minimize latency hits
* CPU defaults to accessing cache, if a *miss* occurs, caching *hierarchy* will fill the correct line; CPUs have multiple levels L1, L2 & L3
* L3 is a shared resource, providing side-channel opportunities[1]
* Cloud environments work very hard to eliminate these types of side-channels, either through aggressive cache clearing on context switch, or isolated allocation of resources.

[1] Newer CPUs have cache allocation technology which purports to assign L3 regions to core or VM

>>> IVT/IDT

* Main mechanism for the SW to respond to hardware events is through the interrupt handling process
* Interrupt Descriptor Table in protected mode, Interrupt Vector Table in real mode
* OS fills a table in memory and a register pointer (IDTR) with functions to handle different types of events
* In protected mode can also provide mechanism to change privilege rings (CPL)

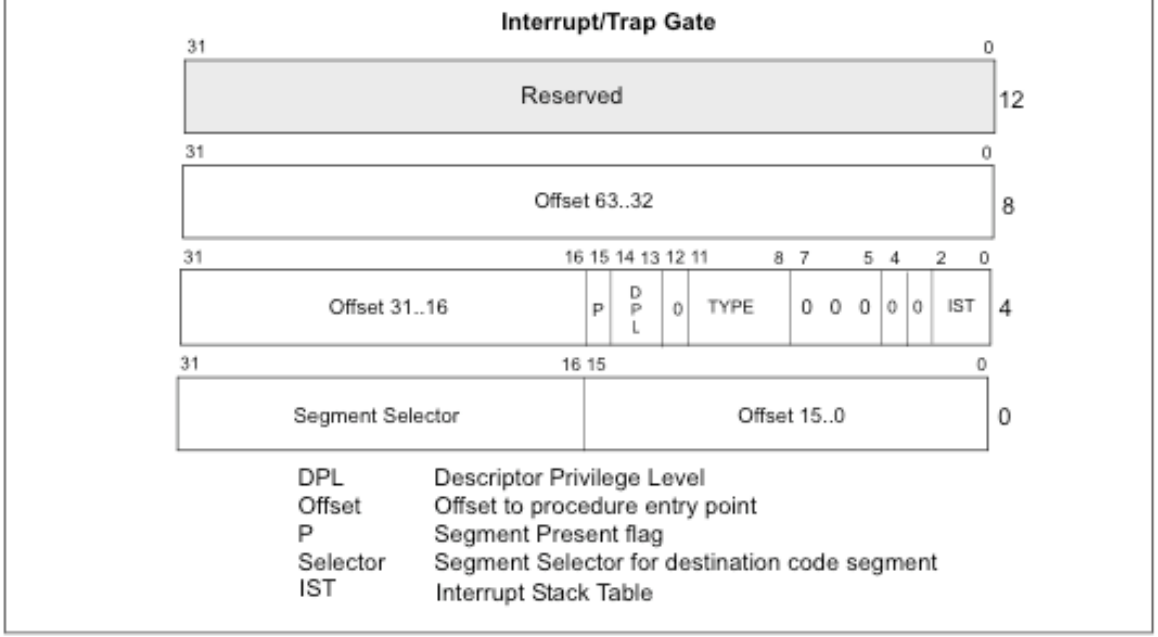
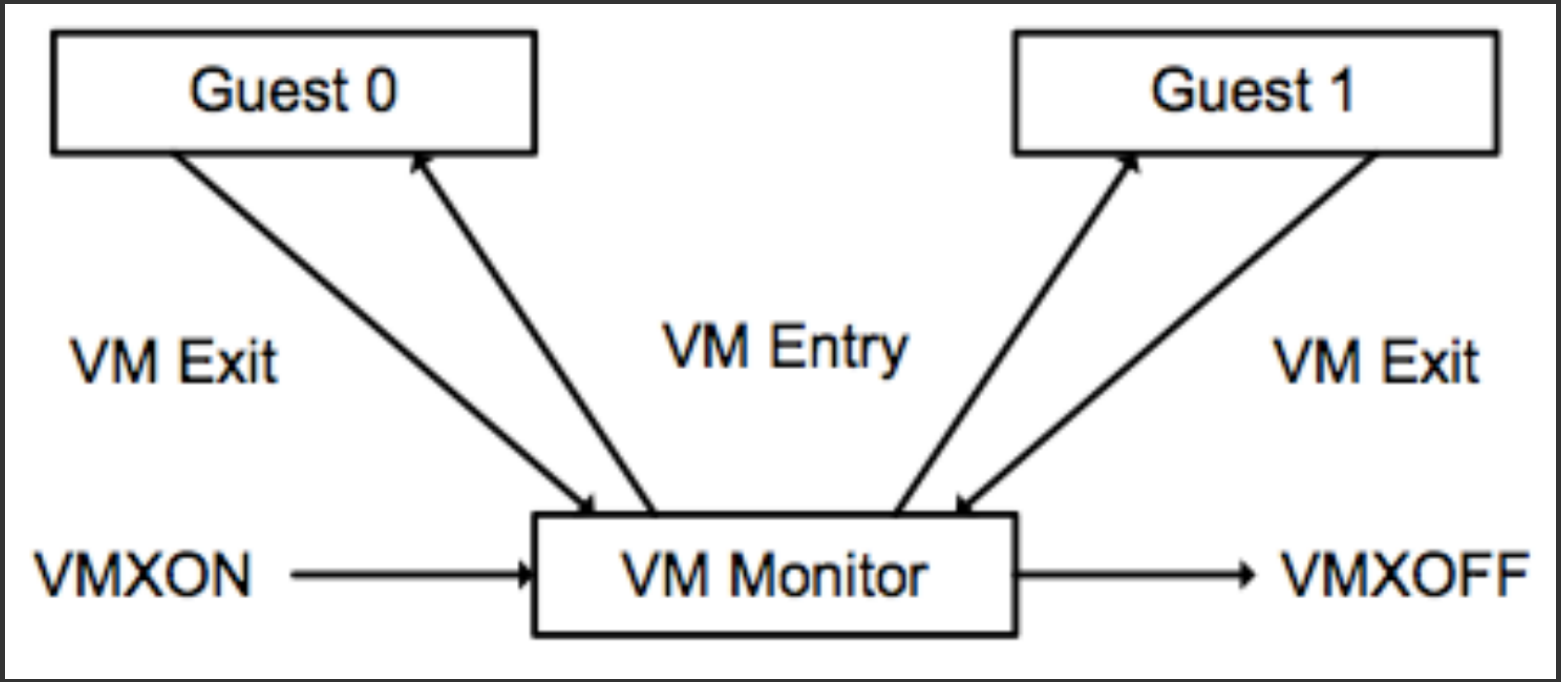


Figure 5-7. 64-Bit IDT Gate Descriptors

>>> Virtualization

* "Ring -1"
* Provides many of the same features available to OS to multiplex/isolate applications for a virtual machine manager (VMM) to manage OSes
* Originally done in a "hack-y" way through software or modified guest OS, now can boot unmodified OS with VT-x, which extends the architecture to support hardware-assisted virtualization



>>> Virtualization is key to the cloud

* Modern VMMs have co-evolved with CPUs to reduce context switching, improve security, and performance
* Examples include:
  * Extended page tables: Paging support for VMMs
  * Tagged TLBs: Per-VM caches of memory lookups to prevent clearing on switch
  * IOMMU: Allows for a VM to directly interface with hardware devices
  * SR-IOV: Hardware feature where one device can simulate multiple to be allocated to a VM each
* These allow cloud providers to offer multiple tenants access to HW resources with minimal interference between customers.

>>> Boot process

* Insight into how the system is loaded helps research if you want to preempt certain processes
* System begins in 16-bit real mode to support backwards compatibility for legacy OSes such as DOS
* Legacy BIOS (or UEFI compatibility mode) continues in real mode
* Modern UEFI systems quickly transition to protected mode for performance reasons and additional features
* Cloud environments have BIOS blob they inject into memory, or can launch directly into UEFI if needed. These are read-only to prevent corruption or compromise.

>>> Standard BIOS

* Boot ROM is loaded into segmented 16-bit mode memory and executed
* Loads BIOS from SPI flash (usually) and initializes system hardware (POST) as well as IVT
* Configures system management mode and (hopefully) locks it with write-once lock bits
* Executes PCI option ROMs to configure hardware devices (which may *hook* IVT entries)
* Executes OS boot-loader which calls BIOS services through IVT calls (some IVT entries are designed to be hooked by OS for periodic alerting)

>>> Standard UEFI

* Boot ROM is loaded into segmented 16-bit mode memory and executed
* Loads UEFI from SPI flash (usually) and initializes system hardware (POST) then transitions to protected mode and configures identity-mapped page tables as well as IDT
* Configures system management mode (called UEFI Runtime Services) and (hopefully) locks it with write-once lock bits
* Executes PCI option ROMs to configure hardware devices in DXE: Driver Execution Environment
* Executes UEFI application(s) (PE-format) to load OS or boot-loader, passing system table structure of function pointers for OS/boot-loader to call

>>> Hooking Boot Process

* Starting with simple boot-loader skeleton, easy to hook boot process and gain insight into OS boot-process

* For legacy BIOS, a simple IVT hook will allow you to be alerted and optionally alter BIOS calls (real mode memory segmentation takes a bit to wrap your head around)

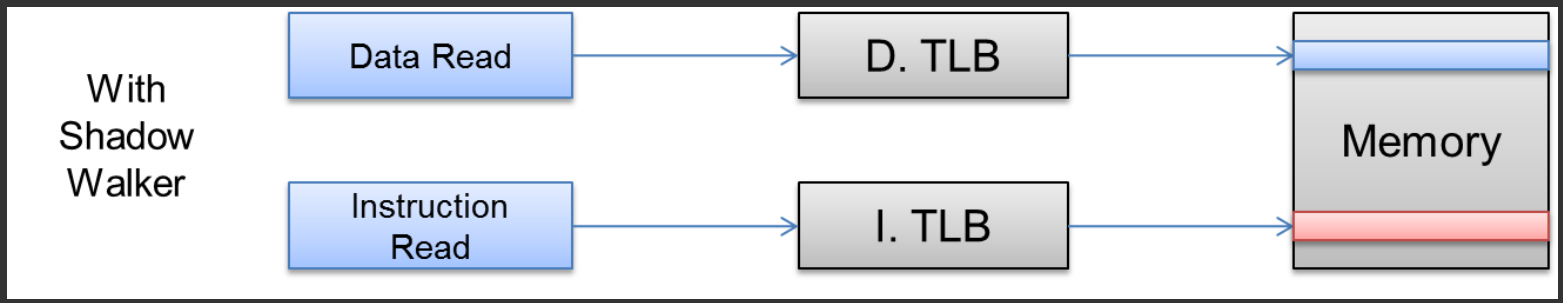
* For UEFI, develop application and use the UEFI LoadImage()/StartImage() boot services to start OS boot-loader, hooking system table structure as desired

>>> Interrupt hooking

* Much harder with Patch Guard, using Windows XP or 7 preferable
* Need to make sure compiler doesn't destroy register state based on incorrect calling conventions
* *VMM can also configure to trap on kernel-level interrupts, may be easier to implement in thin-VMM over patching kernel*
* Following slides have a couple examples of what can be done with this type of event hooking

>>> Page faults (#PF)

* Triggered whenever a mapping from virtual-physical memory is marked as non-present
* If mapping is cached in TLB, may not trigger #PF, must use INVLPG instruction to flush TLB
* Example usage: Shadow-Walker memory-hiding root-kit hooked the #PF-handler to overload a single virtual address to point to different physical addresses depending on type of access (code vs. data)



>>> General Protection Faults

* Used by PAX/GRSecurity to emulate the NX-bit, GPFs occur when the paging structure indicate the mapping is valid, but permissions are wrong
* Set the User/Supervisor (U/S) bit on the page table entry to prevent access:
* If the type of access was data access, set the bit to allow, prime TLB and then reset the U/S bit *without INVLPG*
* If the type of access was execution (instruction fetch), alert or terminate (enforcing NX)
* Maintains a TLB-split to minimize performance impact

>>> VM Exits

* Analogous to interrupts, but allow the VMM to be notified when certain architectural events occur
* Some events are mandatory to trigger VM Exit, many are configurable
* Without VPID, TLBs may be flushed
* A few interesting events that can be triggered on:
  * RDRAND instruction
  * MOV to control registers
  * Reading/writing to MSRs
  * Reading CPUID
  * I/O to CPU ports and hardware devices
  * Reading time-stamp counter
  * Trap-flag for single-stepping
  * ...

>>> EPT faults

* Analogous to interrupts for paging violations, allows VMM to manage guest OS's view of memory
* Hardware-assisted to minimize performance impact
* VMM is notified when there is any form of violation
* Can also trap on the OS #PF interrupt and chose to inject to guest or silently squash to manage memory preemptively

>>> Bigger, faster, cheaper... more accelerators

Besides the obvious drive for more vCPU and memory, there is also interesting use cases that drive the cloud in new directions. While currently many workloads are whole OS and stack (VMs or containers), there is a push for accelerated instance types for offloading GPU work for gaming, video processing, or better VDI solutions; and for AI/ML tasks.

As you dig deeper into the environment, you may find yourself looking at optimized code for an uncommon environment.

>>> Splitting up the system

* With the release of the Nitro framework, AWS moved to further reduce attack surface, and improve performance
* Nitro removes the VMM's role in handling IO -- a common reason for context switching to the VMM
* Nitro cards are PCIe devices that pretend to be a SSD or network card on PCI, then interface through the cloud. Using directed IO, all IO can be handled without VM exit.
* This allows for an unmodified OS to access elastic block storage (EBS) without knowing what it is.
* There are special drivers for high-performance networking to communicate with the Nitro card[2]

[2] https://github.com/amzn/amzn-drivers

Bareflank

"The Bareflank Hypervisor is an open source, lightweight hypervisor... that provides the scaffolding needed to rapidly prototype new hypervisors... *"*

* Open-source: https://github.com/Bareflank/hypervisor
* Lightweight: 10k SLOC (majority if which is testing code to maintain high test coverage
* Scaffolding: If you are not researching *how* VT-x works, use a tool to rapidly focus on your research hypothesis
* Support: Linux, Windows and OS X (expected by year end)

>>> Bareflank II

Bareflank

"...users can leverage inheritance to extend every part of the hypervisor to provide additional functionality *above and beyond what is already provided."*

* Adding VPID support: https://github.com/Bareflank/hypervisor_example_vpid
  * < 10 SLOC for a basic case
* Adding selective MSR trapping: https://github.com/Bareflank/hypervisor_example_msr_bitmap
  * < 25 SLOC for a basic case

>>> LibVMI

* Abstraction layer for performing virtual-machine introspection, if your goal is to monitor a process or OS, use LibVMI[3]
* Provides simple user-space API to trace/modify/trap on execution of software from another guest
* Supports multiple VMMs, OSes and architectures
* Example use-cases from training at TROOPERS[4] provide good jumping-off point

[3] http://libvmi.com/
[4] https://github.com/tklengyel/troopers-training

>>> SimpleVisor

* SimpleVisor[5] provides a very stripped-down VMM that can support Windows 64-bit
* 10 SLOC in assembly, 500 SLOC in C
* If you are engaging in a VT-x specific research effort and want ground-truth for how things *actually* work instead of reading the Intel manuals (though you should have read them already), use this as a self-documenting manual
* Can load/unload while Windows is executing, providing ability to introspect on the host OS without more complex VMM configuration
* HyperPlatform[6] is similar to SimpleVisor, but more robust and extensible for Windows virtualization

[5] https://github.com/ionescu007/SimpleVisor
[6] https://github.com/tandasat/HyperPlatform

>>> Skeleton kernel driver

* Many features are available in ring-0, need access to make use of
* A skeleton kernel module[7] can help serve as boiler-plate
* Linux is easier due to the driver signing hurdles for Windows
* Windows drivers must be signed by a trusted certificate or signature verification disabled[8] in order to easily execute

[7] http://courses.linuxchix.org/kernel-hacking-2002/10-your-first-kernel-mod
[8] https://msdn.microsoft.com/en-us/windows/hardware/drivers/install/test-si

>>> UEFI Tools

* GNU-EFI provides library for doing EFI application development[9]

* The open-source UEFI reference implementation[10] is also available for use, though more difficult to use initially

* The shim[11] Linux loader is a great place to start to see how to inject code into the boot process and load another image with modified Boot Services table

[9] https://github.com/vathpela/gnu-efi/
[10] http://www.tianocore.org/edk2/
[11] https://github.com/rhinstaller/shim

>>> Possible research angles

Much of the cloud is opaque and can only be determined via exploration or RE. Below are a few research projects that may be of interest to jump-start cloud RE exploration:

* ENA protocol analysis/fuzzer: Using the open-source drivers as a framework, understand the protocol from OS to Nitro card for networking, and build a protocol-aware fuzzer to explore hidden states
* Metal exploration: Clouds offer "metal" hosts which give you full access, explore the differences between bare-metal instances and physical machines, explore how they give you full control of a system while ensuring they can recover it and reset to a known-good state.
* Nested virtualization for fuzzing: Learn about hypervisors than running one as a guest, building on KF/x[12] with nested support, you could fuzz a VMM!

[12] https://github.com/intel/kernel-fuzzer-for-xen-project

>>> Conclusions and where to go for help

* Once a research question is posed, rapid determination of what introspective features are needed, what privilege level needed and what tools are available to assist
* There is a wealth of interesting research projects in this low-level space; increasing number of tools to assist with research
* IRC[13] and Twitter[14] a good resource for getting another perspective
* I hope this helped to share my experiences as I did things the not-so-great way to aid you in doing things the way I wish I had/could

[13] #osdev and #bareflank on Freenode
[14] https://twitter.com/JacobTorrey/lists/firmware-security

>>> Questions?

* Thank you for listening!

* Please don't hesitate to reach out with questions and/or comments!