

```
>>> Bootstrapping an architectural research platform
>>> From 0 to hero in 60 min
```

```
Name: Jacob Torrey†
Assured Information Security, Inc.
Date: October 18, 2016
```

[†]torreyj@ainfosec.com (@JacobTorrey)



>>> TL;DR

Raison d'être

The intricacies of low-level architectural analysis have yielded significant findings vis-à-vis security in recent years. As this area of research expands, so too does the duplication of effort to implement "boiler-plate" software needed to gain access or introspective ability required for the end research goal(s).

This talk aims to provide a road-map to accelerate researchers performing their research by reducing duplication of effort and provide a reference to the existing project landscape¹.

¹This is a different format from usual talks; feedback welcomed!



Disclaimer

This talk does not describe new research I've performed; a survey of tools available to help you jump-start your research in this area stemming from the consistent emails I get asking for help. Slides are verbose to serve as a reference.²



²Also, that tweet is fake.
[1. Front matter]\$_

>>> Who am I?

- * Advising research engineer
@ Assured Information
Security in Denver, CO
(words are my own)
- * Leads low-level Computer
Architectures research
group
- * Plays in Intel privilege
rings ≤ 0



>>> Outline

1. Front matter

2. High-level x86

Boot Process

3. Kernel-viewable events

4. VMM-viewable events

5. SMM

6. Case study: TLB-splitting with MoRE

7. Tools

VMM

OS

UEFI

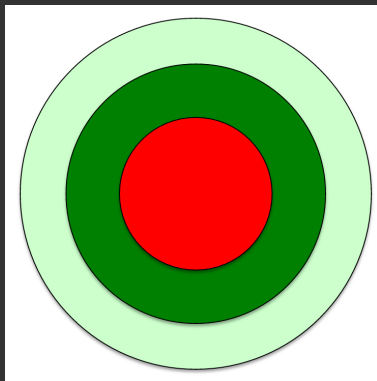
Other

8. Back matter



>>> Privilege rings in x86

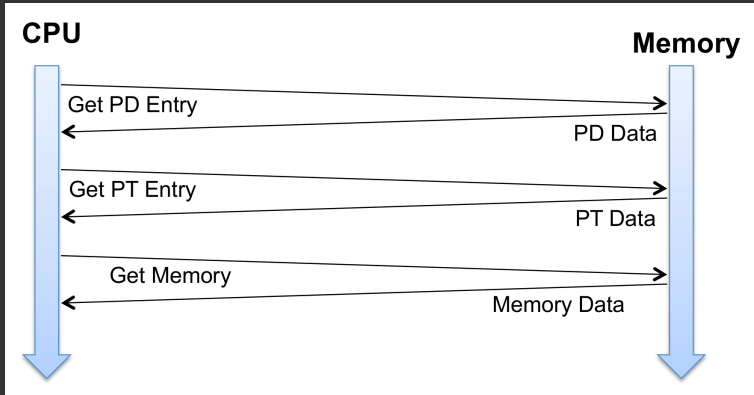
- * Intel 386 added kernel protection and separation of processes; officially rings 0-3, unofficially -1, -2, & -3 (higher is less-privileged)
- * Need to access to proper ring depending on what architectural features you require
- * Once you know what level of access is needed, easier to pair with tool(s)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



- * Typically, the *higher* the ring, the easier development is (e.g., much simpler to develop in user-space than SMM)

>>> 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

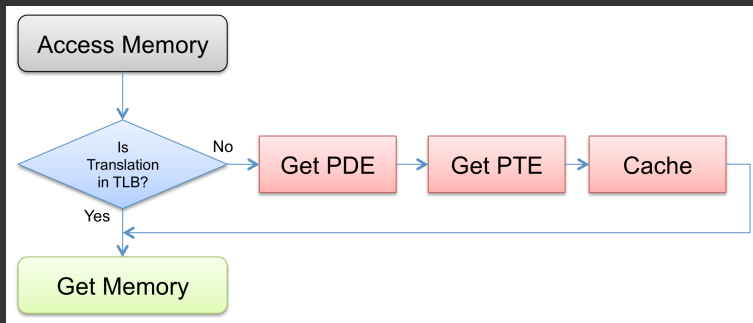
- * Memory access is slow, processors aim to cache as much as possible to minimize latency hits
- * CPU defaults to accessing cache, if a *miss* occurs, caching *hierarchy* will fill the correct line; CPUs have multiple levels L1, L2 & L3
- * L3 is a shared resource, providing side-channel opportunities³
- * Caching type is determined by a combination of control register bits as well as bits in the paging structures and the MTRRs⁴

³Newer CPUs have cache allocation technology which purports to assign L3 regions to core or VM

⁴Invisible Things Labs showed SMM attack where the MCH determined if SMRAM was accessible, but if there was not a cache miss, CPU fetch would not reach MCH; fixed now by BIOS locking MTRRs for certain regions  ais
memory

>>> TLB

- * Even memory accesses to look up virtual-to-physical translations are cached in the translation lookaside buffer (TLB)
- * In silicon, there are 2-3, I-TLB, D-TLB and on newer systems, S-TLB



>>> 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)

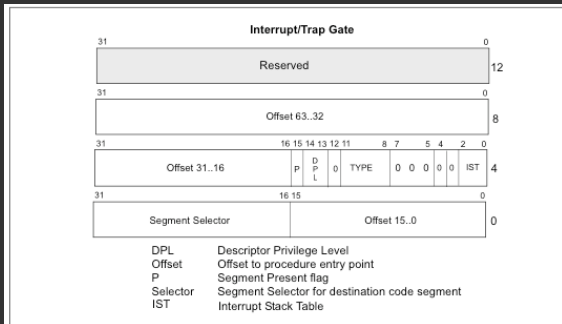
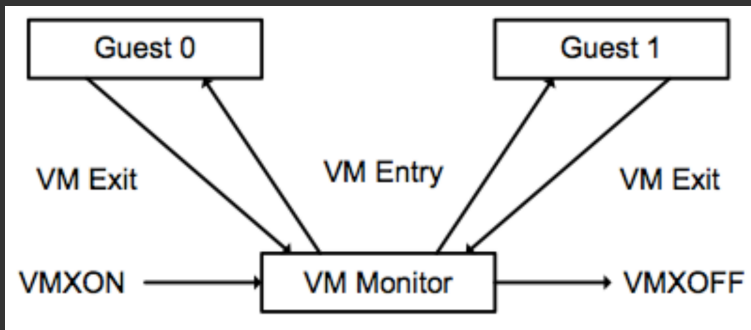


Figure 5-7. 64-Bit IDT Gate Descriptors

>>> 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



>>> 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

>>> 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)

>>> 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

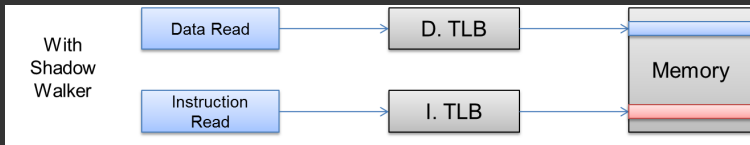
- * Starting with simple boot-loader skeleton, easy to hook boot process and gain insight into OS boot-process
- * For legacy BIOS, a simple IVT hook will allow you to be alerted and optionally alter BIOS calls (real mode memory segmentation takes a bit to wrap your head around)
- * For UEFI, develop 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



>>> Performance Counters

- * Designed for use to optimize performance-critical code, now have been found to be useful for many other interesting purposes
- * Accessible from ring 0, many APIs to export them to user-space (of varying quality)
- * Provide access to information about CPU behavior, few examples:
 - * LBR: Last Branch Record
 - * LLC_MISS: Cache miss counter⁵
 - * EPT: EPT directory look-ups
 - * D/ITLB_MISS: Number of misses in TLB⁶

⁵Used by Herath & Fogh to detect Rowhammer attack (BH'15)

⁶Could probably be used to detect Shadow Walker-type root-kit (if any of you are looking for a research topic)

>>> Branch Tracing

- * Originally, the last-branch-record MSR would records a few previous branches (low overhead, low power)
- * Branch Trace Store (BTS) provided a much higher amount of granularity and more details traces of control-flow (high overhead, high power)
- * Newer CPUs will support Intel Processor Tracing⁷ that can log control-flow information via a ring-buffer (low overhead, high power)

⁷<https://github.com/01org/processor-trace>

>>> 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

>>> SMM overview

- * "Ring -2"
- * Vestigial mode of CPU operation designed for chip-set manufacturers to run privileged code transparently to OS
- * Lots of research in this area, from 2006-now is protected
- * Modern-day systems use this mode of execution to provide the UEFI Runtime Services for UEFI management after system boot
- * Highest-privilege on the system, has full access to system memory and other than through side-channels, very hard to detect its execution



>>> Protections in place

- * Before 2006, the region of memory dedicated for SMM (SMRAM) was unprotected
- * Modifications to the MCH block access to SMRAM unless executing in SMM
- * Invisible Things Labs showed caching attack against SMM by changing the MTRRs caching region and executing code directly from cache while in SMM
- * Modern BIOSes use lockable registers to prevent changing the caching behavior of the SMRAM region

- * SMM is entered through a SMM interrupt (SMI)
- * The SMM handler will handle the interrupt and return to normal execution with the RSM instruction
- * SMM can support the execution of a second hypervisor to contain SMM handler and work in concert with normal executive mode: SMM Transfer Monitor (STM)
- * Intel released open-source reference implementation of STM⁸, though deployment is rare

⁸<https://firmware.intel.com/content/smi-transfer-monitor-stm>

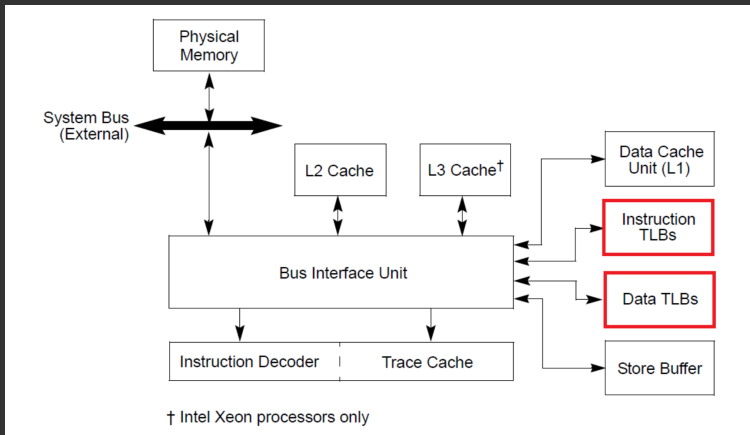
>>> Case study

- * Problem
- * Needs
- * How it was solved⁹
- * How to solve it today (reducing duplication of work)

⁹<http://github.com/ainfosec/MoRE>

>>> TLB-Splitting with S-TLB

- * Challenge: Simulate the TLB-splitting performed by Shadow Walker and PAX/GRsecurity on modern CPUs (Core-i series) with S-TLB



>>> Architectural needs

1. Ability to trap on memory accesses and differentiate between code and data fetch
2. Ability to manage memory without OS interference
3. Minimize performance impact
4. No application source code knowledge or access

>>> VMM design

- * Thin VMM that supports modern OS
- * Support VPID to prevent TLB flushing during VM Exit
- * Ability to use the EPT structures' more granular execute-only permissions

Critical Need

Small code-base to understand within limited time-frame, Xen, KVM, etc. too much time to spend learning code rather than testing hypothesis



>>> Tools I used

- * Began with a skeleton kernel driver for Windows 7 32-bit no PAE
- * Ran into STLB issues and Windows 7 would BSOD when paging structures updated by third-party
- * Switched to using cleaned-up PoC VMX root-kit as template minimal VMM
- * Had to add and debug support for EPT and VPID
- * Added kernel callbacks for new process creation
- * Added *ad hoc* hyper-call API (read insecure)
- * Limited to 32-bit no PAE older OS due to hard-coded elements of the base root-kit that I didn't have time to rewrite



>>> Tools I'd use if I were doing it again



Bareflank Hypervisor™

Bareflank

"The Bareflank Hypervisor is an open source, lightweight hypervisor... that provides the scaffolding needed to rapidly prototype new hypervisors... Existing open source hypervisors that are written in C are difficult to modify, and spend a considerable amount of time re-writing similar functionality instead of focusing on what matters most: hypervisor technologies. Furthermore, users can leverage inheritance to extend every part of the hypervisor to provide additional functionality above and beyond what is already provided."



>>> Bareflank I

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 100% 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

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 LibVMI¹⁰
- * 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 TROOPERS¹¹ provide good jumping-off point

¹⁰<http://libvmi.com/>

¹¹<https://github.com/tklengyel/troopers-training>

>>> SimpleVisor

- * SimpleVisor¹² 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
- * HyperPlatform¹³ is similar to SimpleVisor, but more robust and extensible for Windows virtualization

¹²<https://github.com/ionescu007/SimpleVisor>

¹³<https://github.com/tandasat/HyperPlatform>

>>> Skeleton kernel driver

- * Many features are available in ring-0, need access to make use of
- * A skeleton kernel module¹⁴ 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 disabled¹⁵ in order to easily execute

¹⁴<http://courses.linuxchix.org/kernel-hacking-2002/10-your-first-kernel-module.html>

¹⁵<https://msdn.microsoft.com/en-us/windows/hardware/drivers/install/verify-signing>

>>> UEFI Tools

- * GNU-EFI provides library for doing EFI application development¹⁶
- * The open-source UEFI reference implementation¹⁷ is also available for use, though more difficult to use initially
- * The shim¹⁸ 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

¹⁶<https://github.com/vathpela/gnu-efi/>

¹⁷<http://www.tianocore.org/edk2/>

¹⁸<https://github.com/rhinstaller/shim>

>>> PUFLib

- * Physically Uncloneable Functions expose manufacturing variance to software for device-specific responses
- * Very hardware specific, vary with temperature and hardware age
- * Mostly academic research field thus far; PUFLib¹⁹ will provide abstraction layer and error-correction
- * Hopefully releasing early November with first ubiquitous source of PUFs found in almost all systems
- * Provides simple seal()/unseal() API to lock data to a specific hardware device

¹⁹<https://github.com/ainfosec/puflib>

>>> 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
- * IRC²⁰ and Twitter²¹ 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

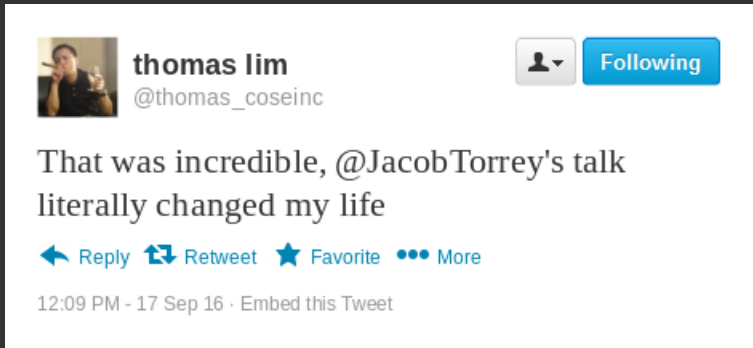
²⁰#osdev and #bareflank on Freenode

²¹<https://twitter.com/JacobTorrey/lists/firmware-security>

>>> Questions?

* Thank you for listening!

* Please don't hesitate to reach out with questions and/or comments!²²



²²Another fake tweet