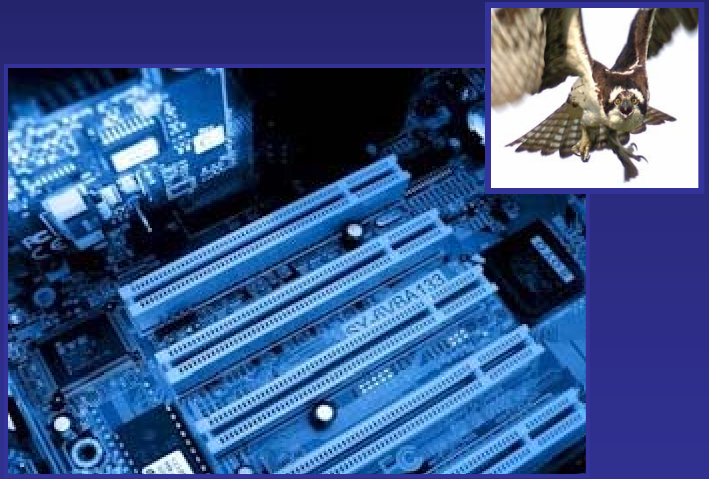
Hacking the Extensible Firmware Interface



John Heasman, Director of Research

NGS Consulting

Agenda

- The role of the BIOS
- Attacking a legacy BIOS
- Limitations of the legacy BIOS
- Introduction to the EFI environment
- Attacking the EFI environment
- UEFI, summary and conclusions

Some Caveats…

- This talk is about rootkit persistence
  - i.e. how to deploy a rootkit from the BIOS/EFI

- This talk is not about Trusted Computing

- Some attacks may require physical access
  - And most require root access
  - Could be deployed as a blended attack
  - e.g. browser bug -> escalation to kernel -> deploy rootkit

- Parts of this research are still work in progress…

The Role of the BIOS

- Test and initialise the hardware
  - Configure Northbridge and Southbridge
- Locate and execute options ROMs
  - Scan PCI buses
  - Copy option ROMs to RAM
  - Scan RAM for options ROMs and execute
- Provide means of user configuration
  - User can select boot device priority and configure hw
  - Persists settings to CMOS
- Launch bootloader

Attacking a Legacy BIOS

- #1 - Modify BIOS code and reflash firmware
- #2 - Modify PCI Option ROM and reflash device
- #3 - Modify ACPI tables and reflash firmware
- #4 - Non-persistent warm reboot attacks

1. Patching the BIOS

- Many places that we can insert code
  - Ultimately we want to subvert the bootloader
  - The bootloader relies on the Interrupt Vector Table
  - The IVT is created dynamically
- BIOS calls int 19h (“the bootstrap loader” vector)
  - Append code before this call after IVT is built
  - Rewrite IVT to hook desired interrupt
- Caveats:
  - May require physical access (write protect jumper)
  - Secure Flash may prevent unsigned updates

2. PCI Option ROMs

- ROM on PCI card holding initialisation code
- Can be for any platform but typically holds x86 code
- Copied to RAM and executed by BIOS
- Stored in EPROM or EEPROM
- Example: EEPROM on your PCIe graphics card:
  - Hooks int 10h in real mode IVT
  - Implements VGA/VBE BIOS functions

Attacking Option ROMs

- Obtain option ROM and flash tool

- Patch option ROM
  - Add code to hook interrupt of choice
  - Gain control when bootloader calls interrupt
  - Patch kernel itself or kernel modules

- Which interrupt to hook?
  - eEye's BootRoot hooked int 13h (disk)
  - Can also hook int 10h (video) on Windows
  - There are likely other candidates

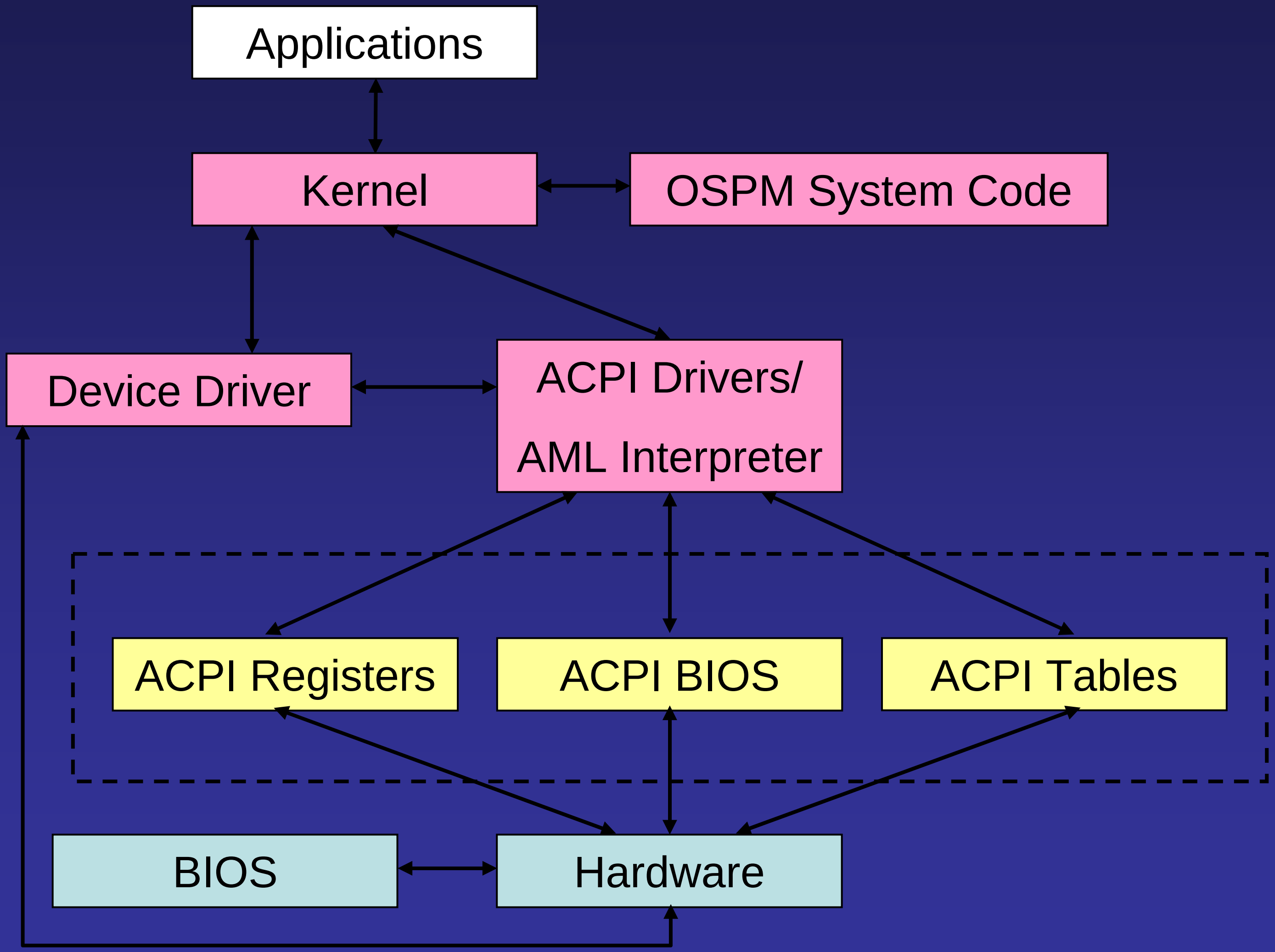
Pros and Cons of Option ROM Attacks

- Typically no jumper on PCI card
  - Flashing is easy – typically just I/O to the card
  - Almost all standalone graphics card can be flashed
  - Network cards with PXE are useful

- Space is typically limited to a few kilobytes
  - Could distribute over multiple PCI devices

- Detection is fairly easy
  - Dump ROM from card and analyse
  - Give aways e.g. presence of protected mode code
  - Detection process could be subverted though

3. Typical ACPI Implementation



ACPI BIOS Rootkits

- ➢ BIOS holds tables containing AML instructions
- ➢ ACPI device driver contains AML interpreter
- ➢ AML instruction set allows us to modify system memory
- ➢ Re-flash BIOS to contain patched ACPI tables
- ➢ AML methods now deploy rootkit from BIOS

Benefits of ACPI Rootkits

- Independent of OS!
  - AML is platform and OS independent
- ASL is a high level language
  - Easy to disassemble AML to ASL and recompile
- Kernel is already loaded when AML is interpreted
  - Modify kernel data structures directly
- Make “smart” decisions before deploying rootkit
  - Future-proof rootkit against service packs/hotfixes

Limitations of ACPI Rootkits

- Must be able to update system BIOS
  - Signed updates prevent attack (Secure Flash)
- OS must have ACPI device driver
  - Stop it loading for cross-view detection
- OS must not sandbox AML interpreter
  - Prevent mapping of kernel address space

4. Warm Reboot Attacks

- ➢ Previous attacks make persistent modifications
  - Makes detection easier
  - Systems with SLAs are not cold booted regularly
  - But might be warm rebooted (to install updates)

- ➢ Persist across reboot by modifying code at reset vector
  - This is copied to shadow RAM during cold boot
  - We must remove write protection then modify

- ➢ Removing write protection is chipset specific
  - Intel: Programmable Attribute Map Registers (PAMs)
  - AMD: Memory Type Range Registers (MTRRs)

Legacy BIOS Limitations

- ➢ BIOS typically written in Assembler
  - Who writes 16-bit real mode assembler?
  - Rooted in x86 Interrupt model

- ➢ Few cleanly defined interfaces exposed by vendors
  - int 15h is the “miscellaneous” interrupt
  - Subfunctions vary from vendor to vendor
  - Interfaces that are defined are clunky

    e.g. the Post Memory Manager (PMM) spec:

Legacy BIOS Limitations Cont.

“A client follows this procedure to locate and access PMM Services:

1. Search for the four-byte “$PMM” string on paragraph boundaries starting at E000h, and ending, if not found, at FFFFh.

2. Verify that the PMM Structure data is valid by performing a checksum. The checksum is calculated by doing a byte-wise sum of the entire PMM Structure and comparing this sum with zero. If the checksum is not zero, then the PMM Structure data is not valid and the EntryPoint field should not be called.

3. Optionally inspect the StructureRevision field to determine the appropriate structure map. The StructureRevision field changes if previously reserved fields in the PMM Structure are redefined to be valid fields.

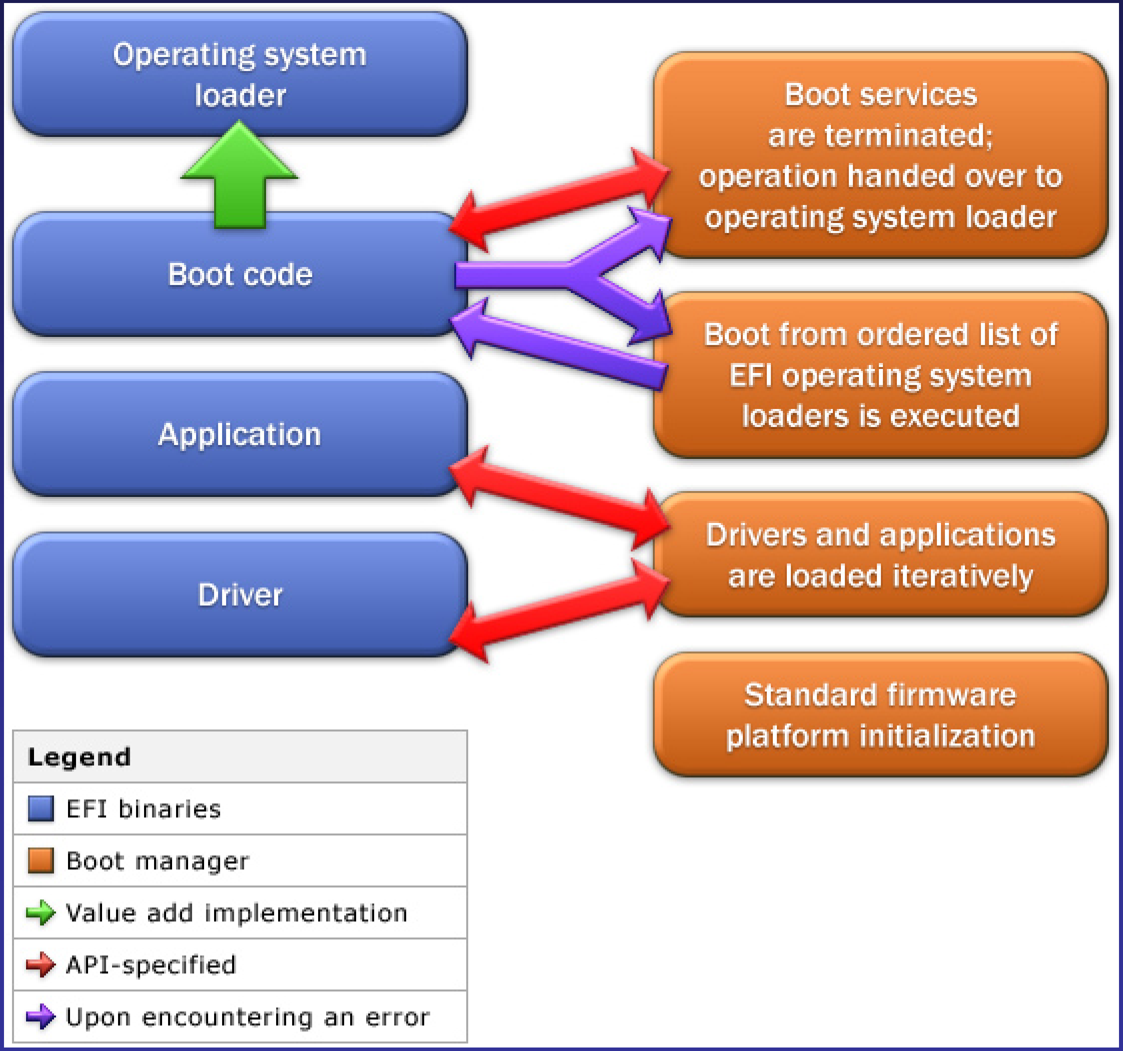
4. Make calls to the EntryPoint field in the PMM Structure to allocate and free memory as desired.”

Introduction to EFI

EFI Design Principles

- ➢ Re-use existing technologies:
  - EFI system partition filesystem is FAT
  - Executables are PE/PE32+
  - ACPI, SMBIOS
- ➢ Extensibility and modularity
  - Core EFI implementation is in firmware
  - Third party drivers can exist on disk or firmware
- ➢ Development in high level language
  - Bootloaders/drivers typically written in C
  - Platform agnostic, spec simply defines interfaces
  - EFI Byte Code (EBC) is interpreted instruction set

A Typical EFI Environment


Operating system loader
Boot code
Application
Driver
Boot services are terminated; operation handed over to operating system loader
Boot from ordered list of EFI operating system loaders is executed
Drivers and applications are loaded iteratively
Standard firmware platform initialization
Legend
🟦 EFI binaries
🟧 Boot manager
➡ Value add implementation
➡ API-specified
➡ Upon encountering an error

Key EFI Definitions

- ➢ Protocol – “drivers” that expose interfaces
  - Each protocol has a GUID
  - A single driver can implement multiple protocols
- ➢ EFI System Table
  - Key EFI data structure handed to every app/driver
  - Provides means of accessing EFI services
- ➢ Boot Services – Services available in EFI environment
  - Event, Timer and Task Priority Services
  - Memory Allocation Services
  - Protocol Handler Services
  - Images Services

Key EFI Definitions Cont.

- Runtime Services – Services available post EFI
  - Variable Services
  - Time Services
  - Virtual Memory Services

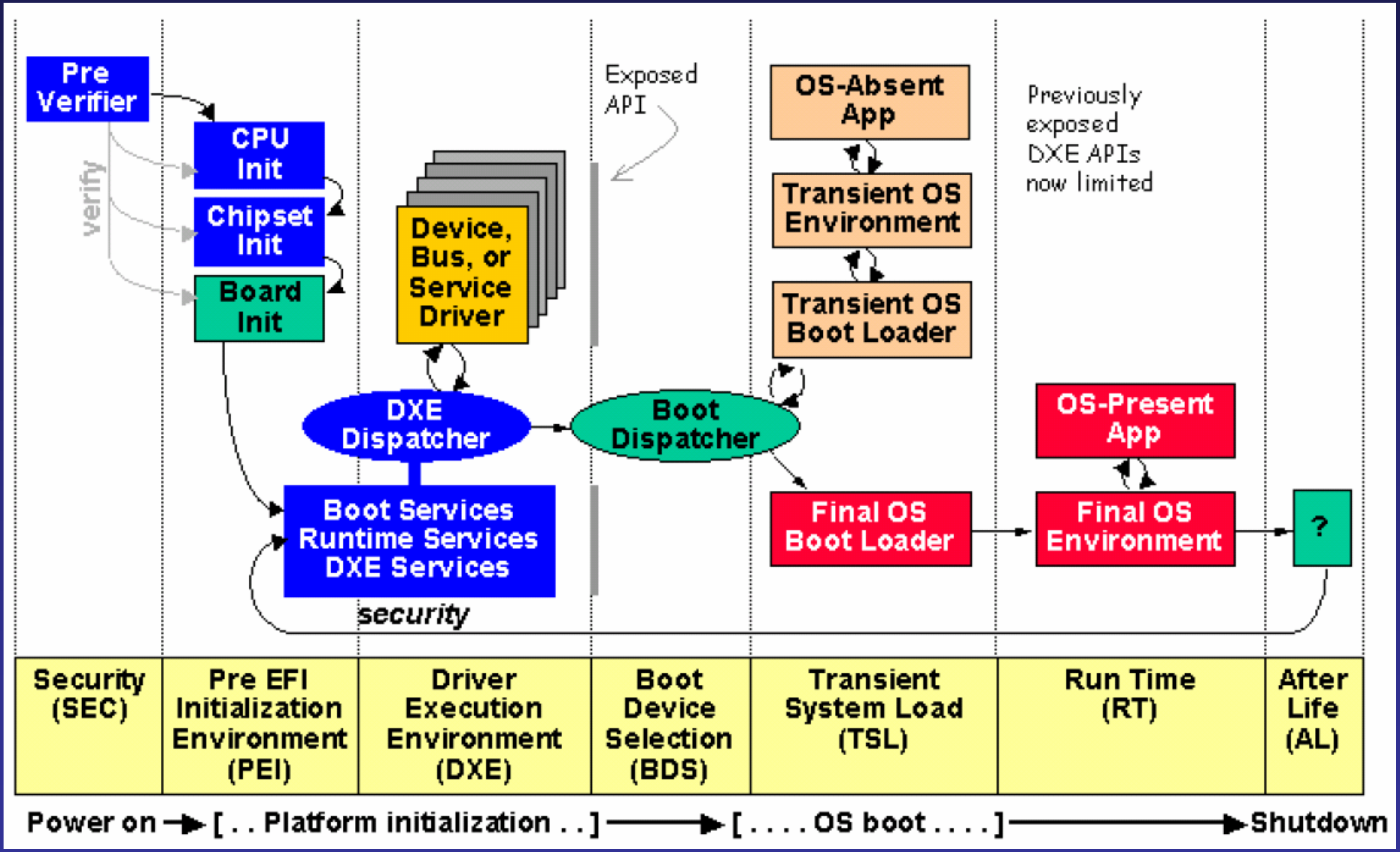
- “The Framework” – Intel’s reference implementation
  - Used by OS X
  - Partially open source as “Tiano”

“Intel views the Framework as the implementation of choice”

EFI Security

- EFI 1.10 spec not focused on security
  - Framework docs elaborate on Security phase
- Security (SEC) is first phase in the Framework:
  - Handles all platform restart events
  - Creates a temporary memory store
  - Serves as the root of trust in the system
  - Passes handoff information to the PEI
- PEI is the Pre-EFI phase:
  - Loads modules specific to low level hardware
  - Maintains root of trust
  - Invokes Driver Execution Environment (DXE) loader

EFI Security Cont.



Pre Verifier
verify
CPU Init
Chipset Init
Board Init
Device, Bus, or Service Driver
DXE Dispatcher
Boot Services Runtime Services DXE Services
security
Exposed API
Boot Dispatcher
OS-Absent App
Transient OS Environment
Transient OS Boot Loader
Previously exposed DXE APIs now limited
OS-Present App
Final OS Boot Loader
Final OS Environment
?
Security (SEC)
Pre EFI Initialization Environment (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TSL)
Run Time (RT)
After Life (AL)
Power on
[. . Platform initialization . .]
[. . . . OS boot . . . .]
Shutdown

Abusing EFI

Objectives

- Get code into the EFI environment
  1. Modify bootloader itself
  2. Modify NVRAM bootloader variable
  3. Modify and reflash platform firmware
  4. Exploit implementation flaw in driver

- Subvert loading of the operating system
  1. Shim a boot service/runtime service
  2. Modify the ACPI tables
  3. Load an SMM driver
  4. Hook interrupt handlers if CSM & legacy bootloader

Modifying the Bootloader

- Modify the bootloader binary itself
  - MacOS X: /System/Library/CoreServices/boot.efi
  - N.B. OS X does not use the EFI system partition

- Not very stealthy
  - Easily detected with system integrity tools
  - Why not just modify the kernel itself?
  - Won’t work if environment enforces driver signing

Modifying NVRAM Variables

- Global variables persisted in NVRAM
  - Specifies which bootloader to use
  - EFI provides interface for reading/writing
  - OS typically provides an ‘nvram’ tool

- Create custom bootloader
  - Can simply patch environment and call original
  - Modify “efi-boot-device” variable

- Stealthier than modifying original bootloader?
  - Leaves original bootloader in tact
  - But obviously requires extra file on disk
  - Won’t work if environment enforces driver signing

Code Injection Attacks

- Important when firmware verifies digital signatures
  - Depends on implementation flaw in driver

- Plenty of targets:
  - File system drivers (e.g. FAT32, HFS+)
  - PE parsing code
  - Crypto code (Data in certs, ASN.1 decoding)
  - Network interaction (PXE)

Shimming Boot Services

- Bootloader must call ExitBootServices()
  - This indicates it is ready to launch kernel
  - Runtime drivers remain
  - Perfect place to hook as kernel is likely in memory

- Create runtime driver that hooks ExitBootServices:
  - Replace ExitBootServices function pointer
  - Function pointer located in EFI System Table
  - Locate kernel, patch to deploy rootkit

- Could alternatively shim a runtime service, if called

Shimming Boot Services Cont.

eLilo loading kernel:

```
1. kernel_load(image, kname, &kd, &imem, &mmem));

/* free resources associated with file accesses
    (before ExitBootServices) */
2. close_devices();

/* terminate bootservices */
3. status = BS->ExitBootServices(image, cookie);

4. start_kernel(kd.kentry, bp);
/* NOT REACHED */
```

System Management Mode

- SMM first introduced in 386SL
  - Entered via SMI
  - May be triggered via external event
  - Or periodically
  - Or on I/O access

- “Get out of jail free card” for platform designers
  - Enable/disable ACPI mode
  - Power button support while not in ACPI mode
  - Error logging for ECC/PERR/SERR in IA-32
  - Protected flash writes on some IA-32 platforms
  - Century rollover bug workaround

Abusing SMM

- Loic Duflot used SMM to bypass BSD Securelevels
  - Hinted at possibility of SMM-based malware

- What does this mean for rootkits/rootkit detection?
  - Hardware breakpoints do not fire in SMM
  - Access to SMM memory blocked if lock bit set
  - SMIs cannot be interrupted, even by NMIs
  - SMM can trap I/O reads/writes

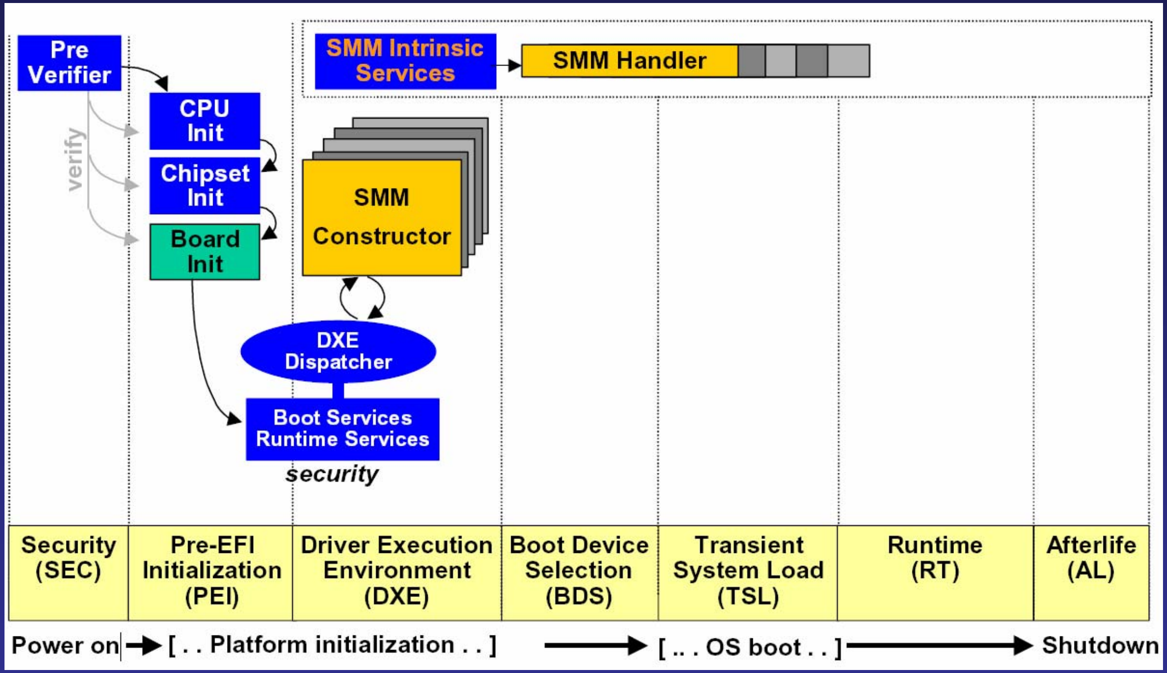
- Why has there been no SMM malware yet?
  - Bar for entry is high: debug with logic analyzer
  - Limited opportunity with SMM lock bit
  - System dependencies make it less attractive

EFI and SMM

- ➢ EFI provides clean, easy to use SMM interfaces
  - Base Protocol for driver registration
  - Access Protocol for setting lock bit
  - Control Protocol for triggering SMI
  - Child Dispatch Protocol for types of SMM event

- ➢ System Management System Table (SMST)
  - Provides set of services to SMM drivers
  - Handles memory allocation/de-allocation
  - Abstracts access to CPU context, memory, and I/O space

EFI and SMM Cont.



"The SMM phase must preserve the chain of trust initiated in the previous phase. To do so, it must validate the modules that it loads for the subsequent dispatcher."

Compatibility Support Modules

- Provide backwards compatibility for legacy bootloader
  - Implements IVT
  - Execute bootloader in 16-bit real mode
  - Interrupt handlers thunk to EFI (32-bit, protected)

- Examples:
  - Xp On Mac (XOM)
  - Apple Bootcamp

- Hook IVT as per legacy BIOS attack

UEFI

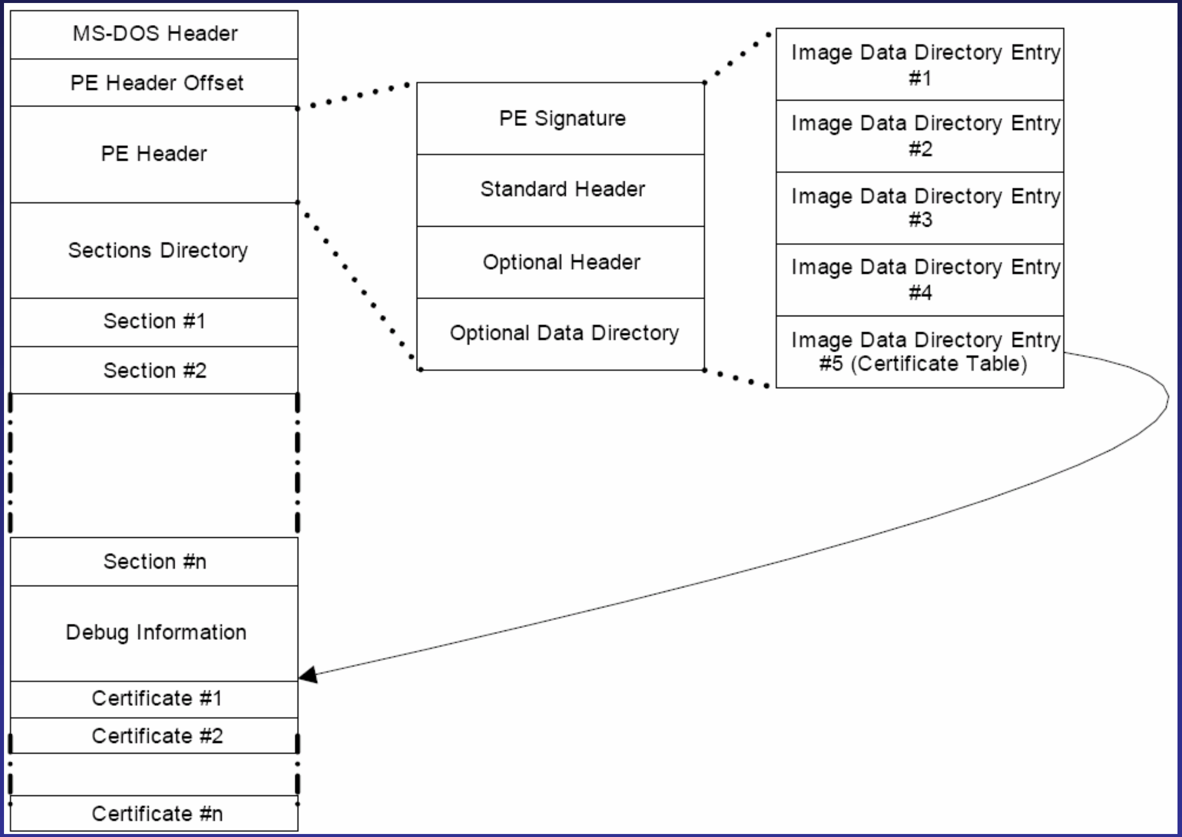
EFI and UEFI

- “Unified EFI” spec originally based on EFI 1.10
  - UEFI is a consortium of major hw/sw vendors
  - Current version is 2.1

- Provides further information on driver signing

- Trusted Computing Group specs:
  - TCG EFI Platform Specification
  - TCG EFI Protocol Specification

EFI and UEFI


MS-DOS Header
PE Header Offset
PE Header
Sections Directory
Section #1
Section #2
Section #n
Debug Information
Certificate #1
Certificate #2
Certificate #n
PE Signature
Standard Header
Optional Header
Optional Data Directory
Image Data Directory Entry #1
Image Data Directory Entry #2
Image Data Directory Entry #3
Image Data Directory Entry #4
Image Data Directory Entry #5 (Certificate Table)

Summary & Conclusions

- ➢ EFI offers a large attack surface
  - High level development tools make it more of a target
  - Third party driver model presents easier target
- ➢ The EFI spec is vague on security
  - Blurred relationship between spec and Framework
  - How is the Sec phase supposed to be implemented?
- ➢ UEFI makes things clearer
  - But plenty of surface for code injection attacks
- ➢ More to come on EFI attacks, stay tuned ☺

References

- EFI - http://www.intel.com/technology/efi/
- UEFI - http://www.uefi.org/specs/
- Tiano - https://www.tianocore.org/
- “Security Issues Relating to System Management Mode” - http://www.cansecwest.com/slides06/csw06-duflot.ppt
- XP On Mac - http://www.onmac.net/

Any Questions?

Thanks!

john at ngssoftware dot com