Analyzing UEFI BIOS from Attacker & Defender Viewpoints

Xeno Kovah	@xenokovah
John Butterworth	@jwbutterworth3
Corey Kallenberg	@coreykal
Sam Cornwell	@ssc0rnwell

__DRAFT!__ Go look for the final version on the intertubes!

MITRE

BIOS is dead, long live UEFI!

- **"Slow down fatty, we're not to the moon yet!"**
- **We'll never be rid of certain elements of legacy BIOS on x86**
- **The initial code will always be hand-coded assembly (or at least C with lots of inline asm), because C doesn't have semantics for setting architecture-dependent registers.**
- **On all modern systems Intel makes extensive use of PCI internal to their own CPUs, therefore early in system configuration there will always be plenty of port IO access to PCI configuration space, where you're going to be at a loss for what is happening to what, until you do extensive looking up of things in manuals**
  - Add to that plenty of port IO to devices where you have no idea what's being talked to, since there's no documentation
- **The bad old days live on, and you still have to learn them…**
- **But there's a whole lot more new interesting and juicy bits added in to the system to be explored**

BIOS/UEFI Commonalities

- **BIOS and UEFI share 2 common traits:**

1. **CPU entry vector on the SPI flash chip is the same**
2. **They sufficiently configure the system so that it can support the loading & execution of an Operating System**
   - They go about it in different ways
   - call it different names: POST/BIOS vs. Platform Initialization
   - This should include properly locking down the platform for security
   - Where software meets bare metal the machine instructions are the same (i.e.: PCI configuration, MTRRs, etc…)

- **UEFI, however, is a publically documented, massive framework**
- **Has an open-source reference implementation called the EDK2**
- **The UDK (UEFI Development Kit) is analogous to a “stable branch” of the “cutting edge” EDK2 (EFI Development Kit)**

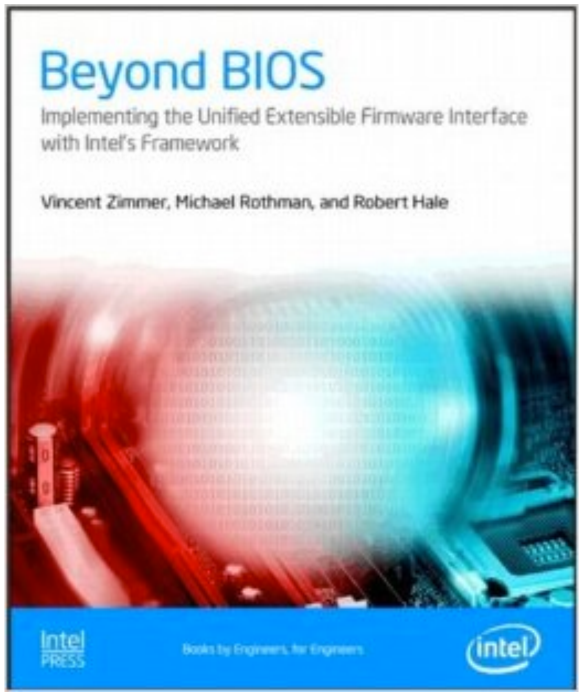
About UEFI

- UEFI = Unified Extensible Firmware Interface
- As the name implies, it provides a software interface between an Operating System and the platform firmware
- The "U" in UEFI is when many other industry representatives became involved to extend the original EFI
  - Companies like AMD, American Megatrends, Apple, Dell, HP, IBM, Insyde, Intel, Lenovo, Microsoft, and Phoenix Technologies
- Originally based on Intel's EFI Specification (1.10)
- Does provide support for some legacy components via the Compatibility Support Module (CSM)
  - Helps vendors bridge the transition from legacy BIOS to UEFI
- It's much larger than a legacy BIOS
  - (And the attackers rejoiced!)

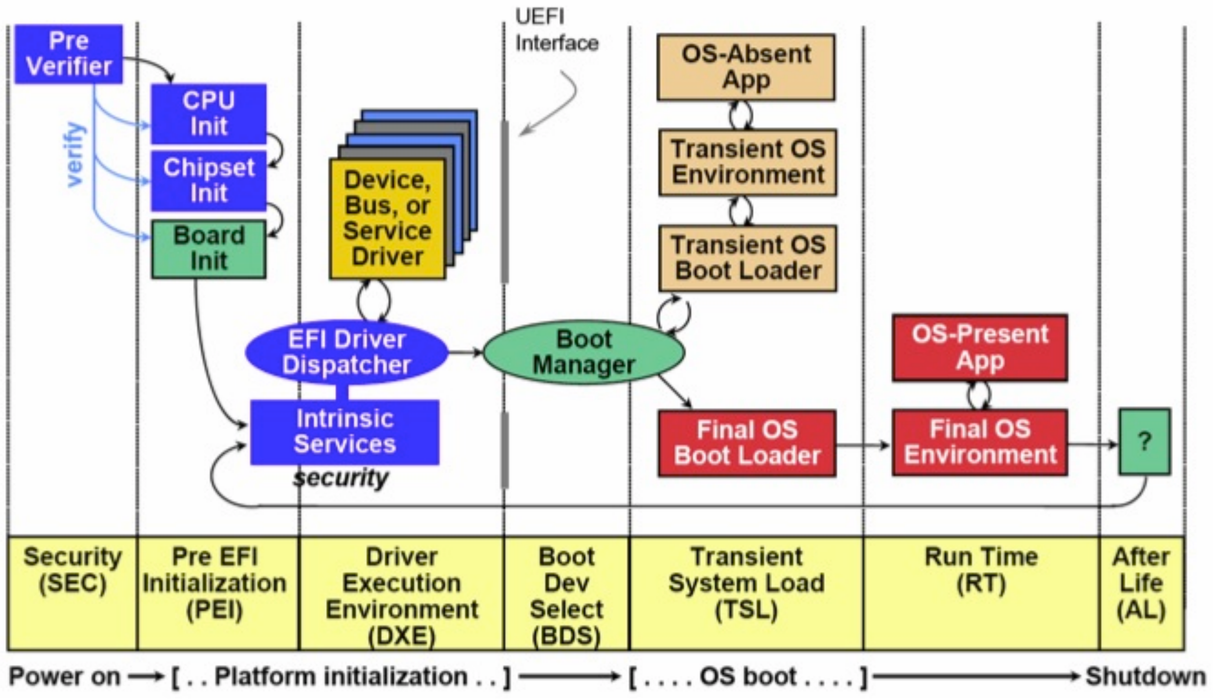
Something you may want to read

- If you don't want to just dive into the thousands of pages of UEFI specifications, a good overview is also given in Beyond BIOS: Developing with the Unified Extensible Firmware Interface 2nd Edition by Zimmer et al.

- Otherwise go enjoy the specs here: http://www.uefi.org/specifications


Beyond BIOS
Implementing the Unified Extensible Firmware Interface with Intel's Framework
Vincent Zimmer, Michael Rothman, and Robert Hale
Intel PRESS
Books by Engineers, for Engineers
intel

UEFI Differences: Boot Phases

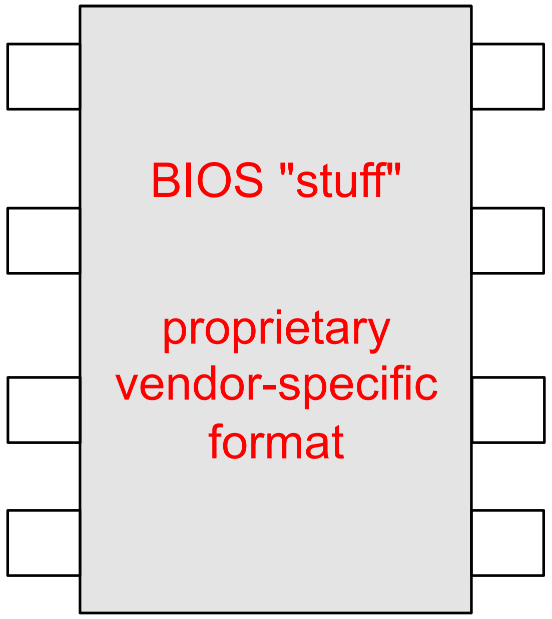


- 7 Phases total
- Each phase is defined via specification

In the beginning

- Let's start with the hardware, rather than the software architecture

Legacy BIOS Firmware Storage



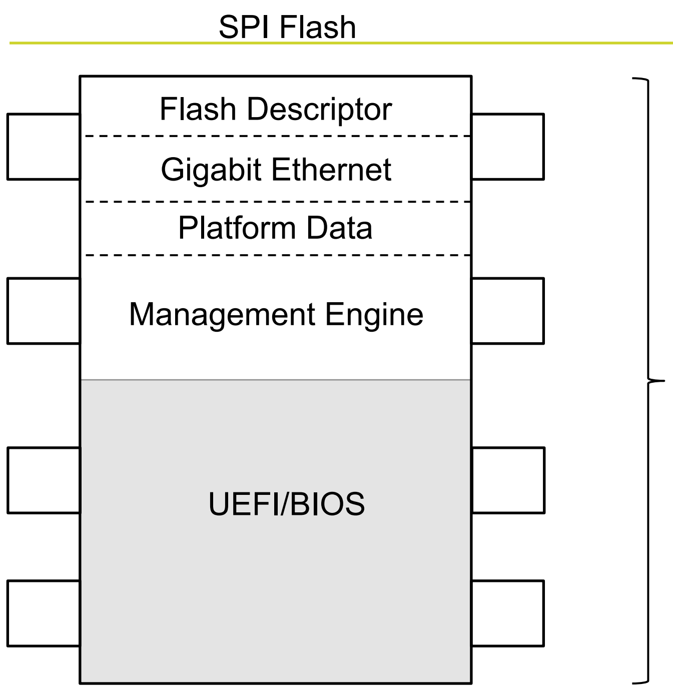
SPI Flash

or

Intel Firmware Hub (on older (meaning like 2007) systems)

- **While there was some semblance of structure and sanity to the contents stored on the chip, it was in some vendor-specific format, which people had to reverse engineer (usually people interested in decomposing the BIOS to suck out a Windows license key sub-file and stick it in some other machine)**
- **To save space, it's probably structured like a "packed" file, with some small decompressor stub which expands compressed modules into memory before executing them**

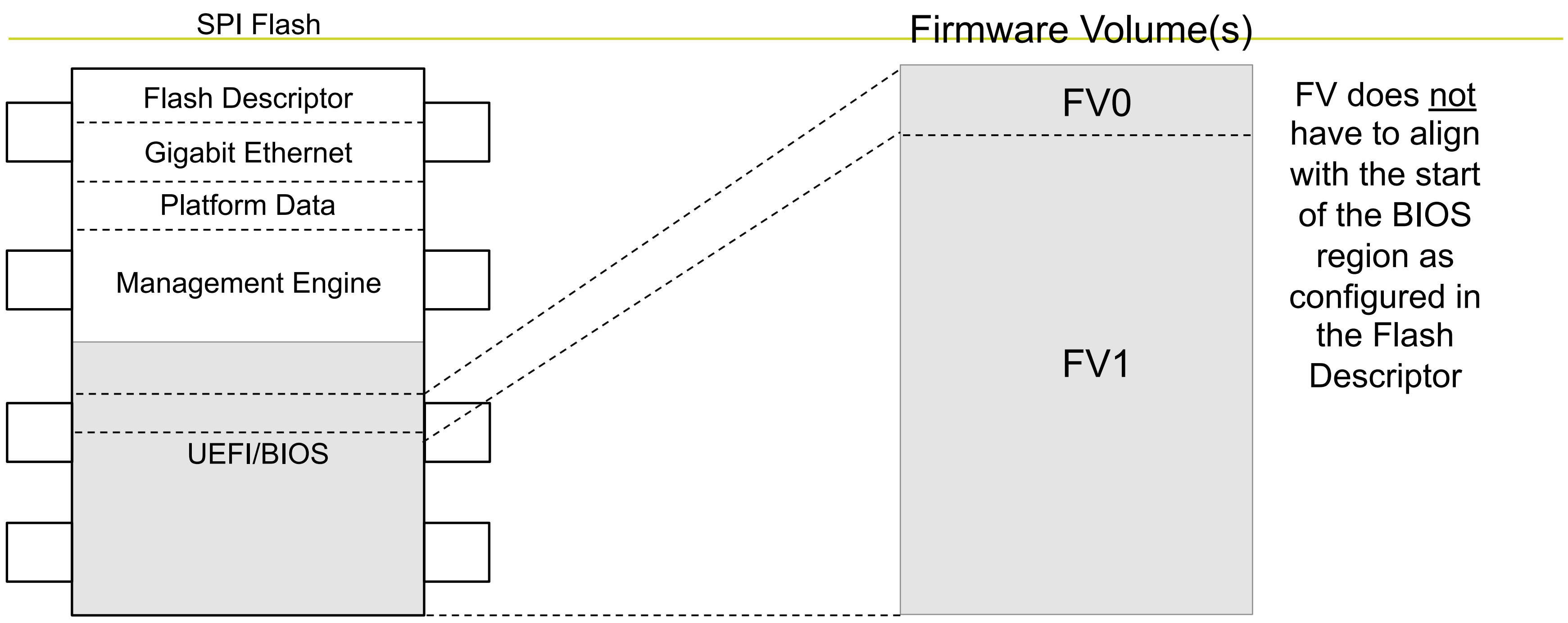
UEFI Firmware Storage



Firmware Device refers to the flash chip

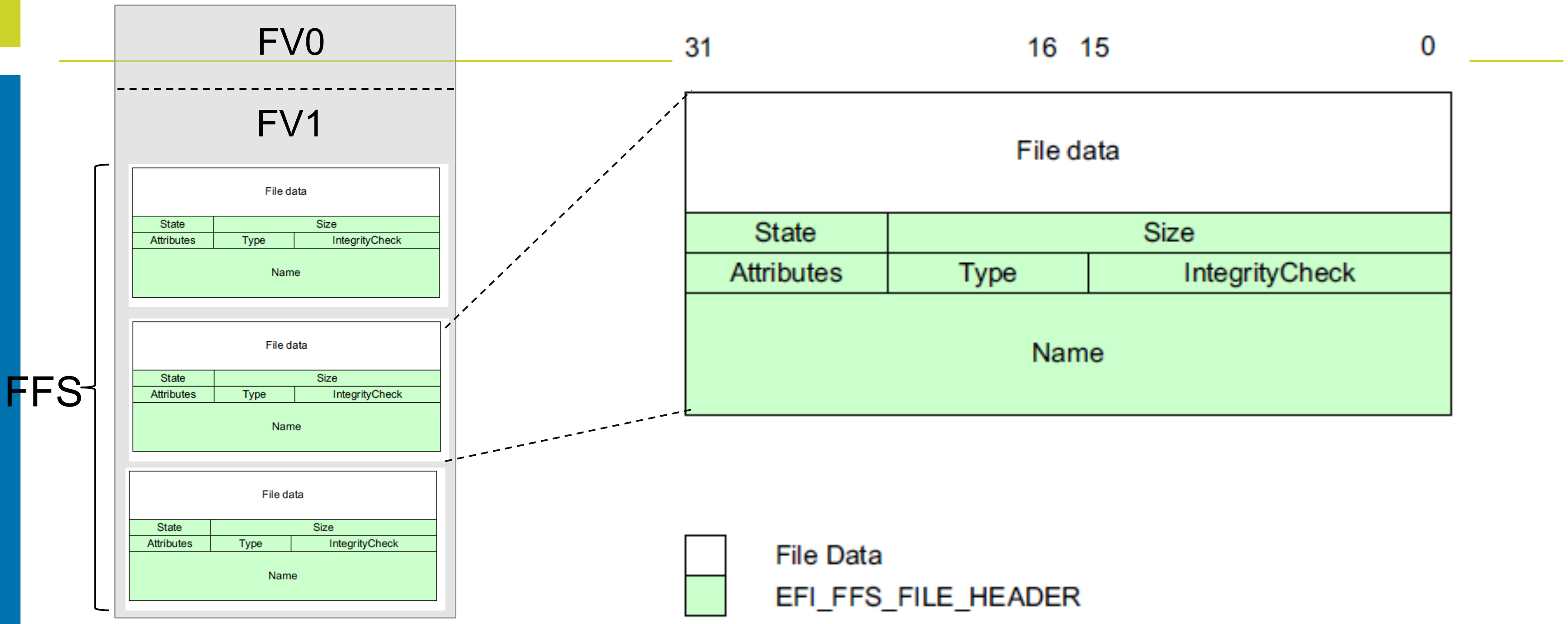
- **UEFI utilizes the physical flash device as a storage repository, with 5 currently defined regions (with space for more), each with differing purposes and access controls**
- **The contents of the "BIOS region" is what we're most interested in**

Firmware Volumes (FVs)



- **A Firmware Device is a physical component such as a flash chip. But we mostly care about Firmware Volumes(FVs)**
- **FVs are logical firmware devices that can contain multiple firmware volumes (nesting)**
  - We often see separate volumes for PEI vs. DXE code
- **FVs are organized into a Firmware File System (FFS)**
- **The base unit of a FFS is a file**

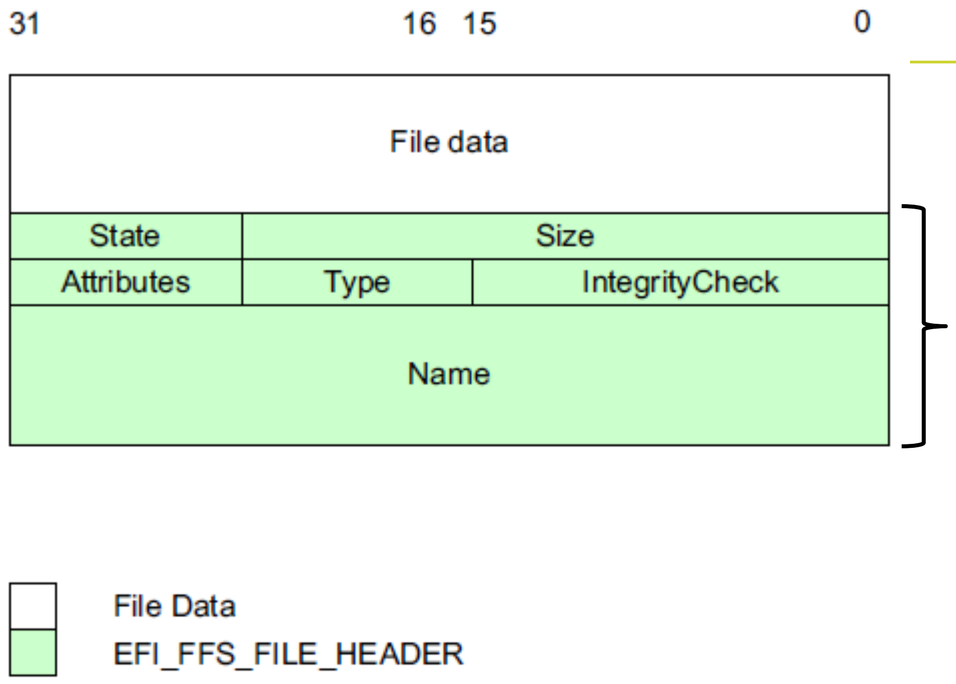
Firmware File System (FFS)



- FVs are organized into a Firmware File System (FFS)
- A FFS describes the organization of files within the FV
- The base unit of a FFS is a file
- Files can be further subdivided into sections

MITRE

Firmware Files



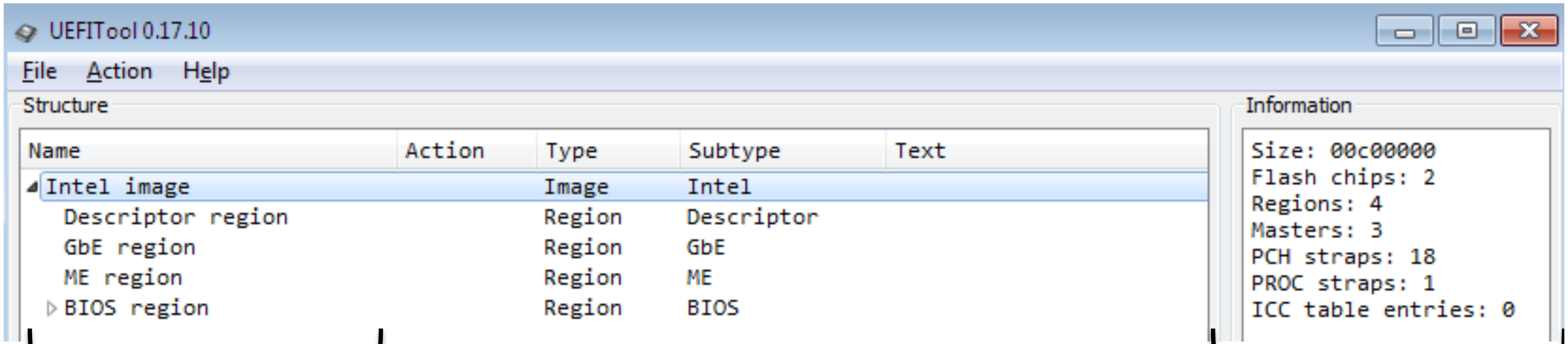
```
typedef struct {
  EFI_GUID                  Name;
  EFI_FFS_INTEGRITY_CHECK   IntegrityCheck;
  EFI_FV_FILETYPE           Type;
  EFI_FFS_FILE_ATTRIBUTES   Attributes;
  UINT8                     Size[3];
  EFI_FFS_FILE_STATE        State;
} EFI_FFS_FILE_HEADER;
```

- **PE (Portable Executable) file format**
  – Alternatively can be a TE (Terse Executable) which is a “minimalist” PE
  – Oh, how interesting! My BIOS uses "Windows" executables? I know how to analyze those!

Options for Parsing FFS

- **EFIPWN**
  - was the first one, so it's what we started from, but it's not actively maintained, and it's known to not handle some vendor-specific foibles, so we're moving away from it
  - https://github.com/G33KatWork/EFIPWN
- **UEFITool**
  - A nice GUI way to quickly walk through the information, with a UEFIExtract command line version for extracting all the files
  - https://github.com/LongSoft/UEFITool
- **UEFI Firmware Parser**
  - Ted Reed is very responsive when files are found that can't be parsed with this. We're probably moving to using it in the future
  - https://github.com/theopolis/uefi-firmware-parser

Go to File->Open and select the file dump (I selected the "e6430A03.bin")

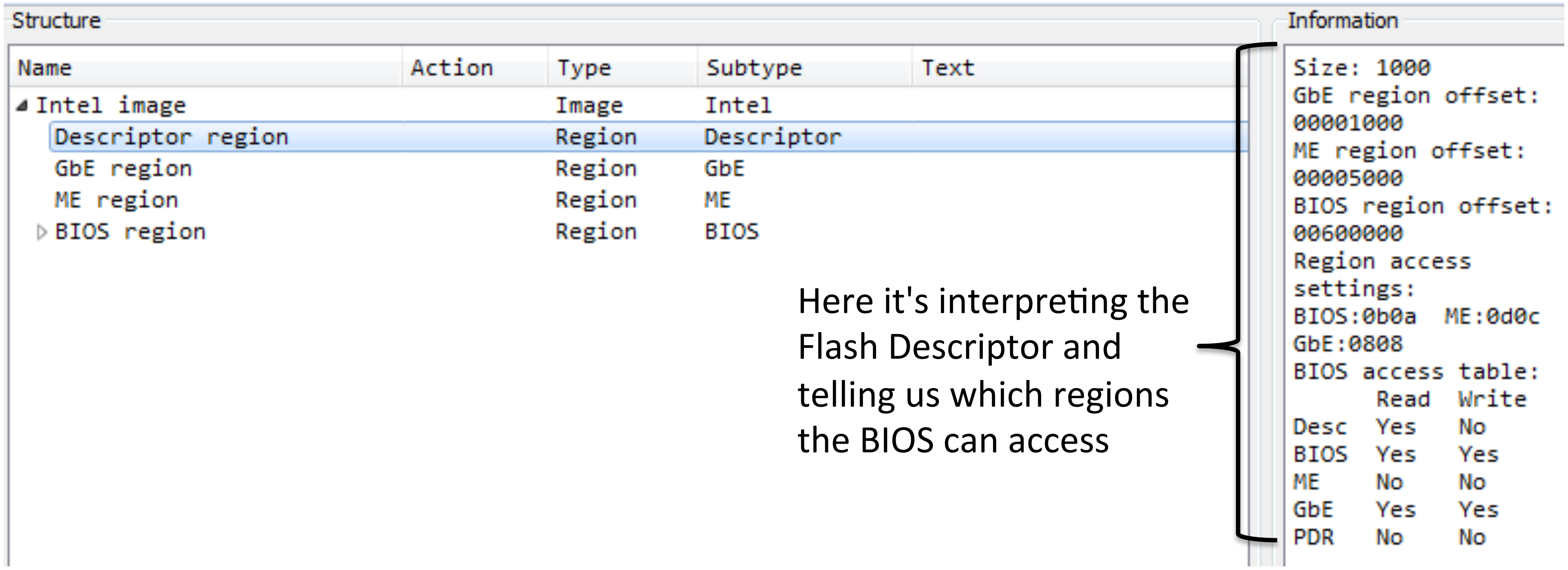


Name	Action	Type	Subtype	Text
Intel image		Image	Intel	
Descriptor region		Region	Descriptor	
GbE region		Region	GbE	
ME region		Region	ME	
BIOS region		Region	BIOS	

```
Size: 00c00000
Flash chips: 2
Regions: 4
Masters: 3
PCH straps: 18
PROC straps: 1
ICC table entries: 0
```

Navigation by expanding portions here

Parsed metadata here



Name	Action	Type	Subtype	Text
Intel image		Image	Intel	
Descriptor region		Region	Descriptor	
GbE region		Region	GbE	
ME region		Region	ME	
BIOS region		Region	BIOS	

```
Size: 1000
GbE region offset:
00001000
ME region offset:
00005000
BIOS region offset:
00600000
Region access
settings:
BIOS:0b0a  ME:0d0c
GbE:0808
BIOS access table:
      Read  Write
Desc  Yes   No
BIOS  Yes   Yes
ME    No    No
GbE   Yes   Yes
PDR   No    No
```

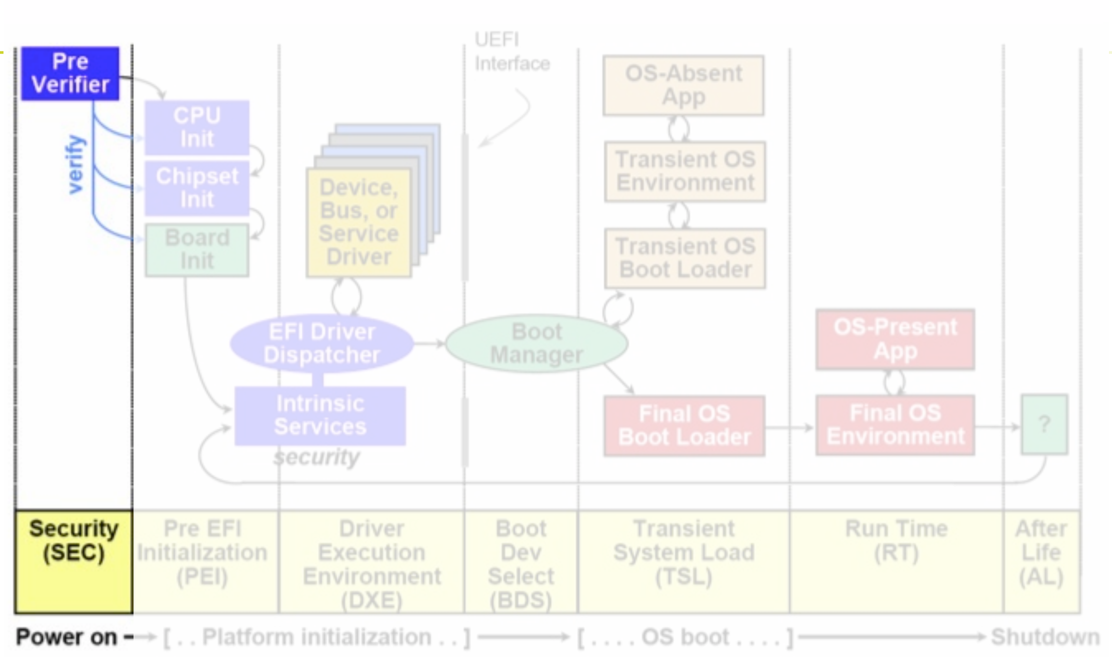
Here it's interpreting the Flash Descriptor and telling us which regions the BIOS can access

A standard way of putting together the firmware filesystem, with nice human readable names, makes it easier for me to find my way around to the likely locations I want to attack

A standard way of putting together the firmware filesystem, with nice human readable names, makes it easier for me to understand the context of what might have been attacked if I see a difference there

MITRE

Security (SEC) Phase

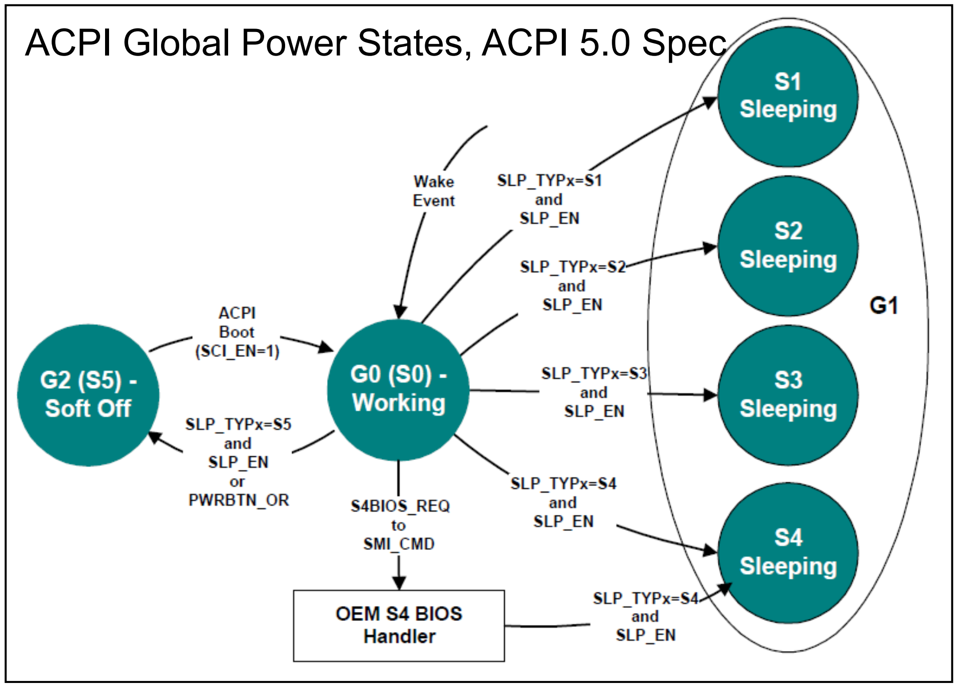


- **The SEC phase is the first phase in the PI architecture**
- **Contains the first code that is executed by the CPU**
- **Environment is basically that of legacy:**
  - Small/minimal code typically hand-coded assembly so architecturally dependent and not portable
  - Executes directly from flash
  - Will be uncompressed code

Platform Initialization Spec Vol. 1, Version 1.3, Sec. 13

SEC Responsibilities 1 of 2

- **Name is a misnomer, as most security-critical things happen later (though of course if the system is compromised this early, the attacker definitely wins)**
- **This is where architecturally the core (read-only) security-critical code *should* go, but doesn't…**
- **The SEC phase handles all platform reset events**
  - All system resets start here (power on, wakeup from sleep, etc)



System boot will follow a different path based on what power state its in on startup!

Platform Initialization Spec Vol. 1, Version 1.3, Sec. 13

Quick ACPI Note: Sleep Modes

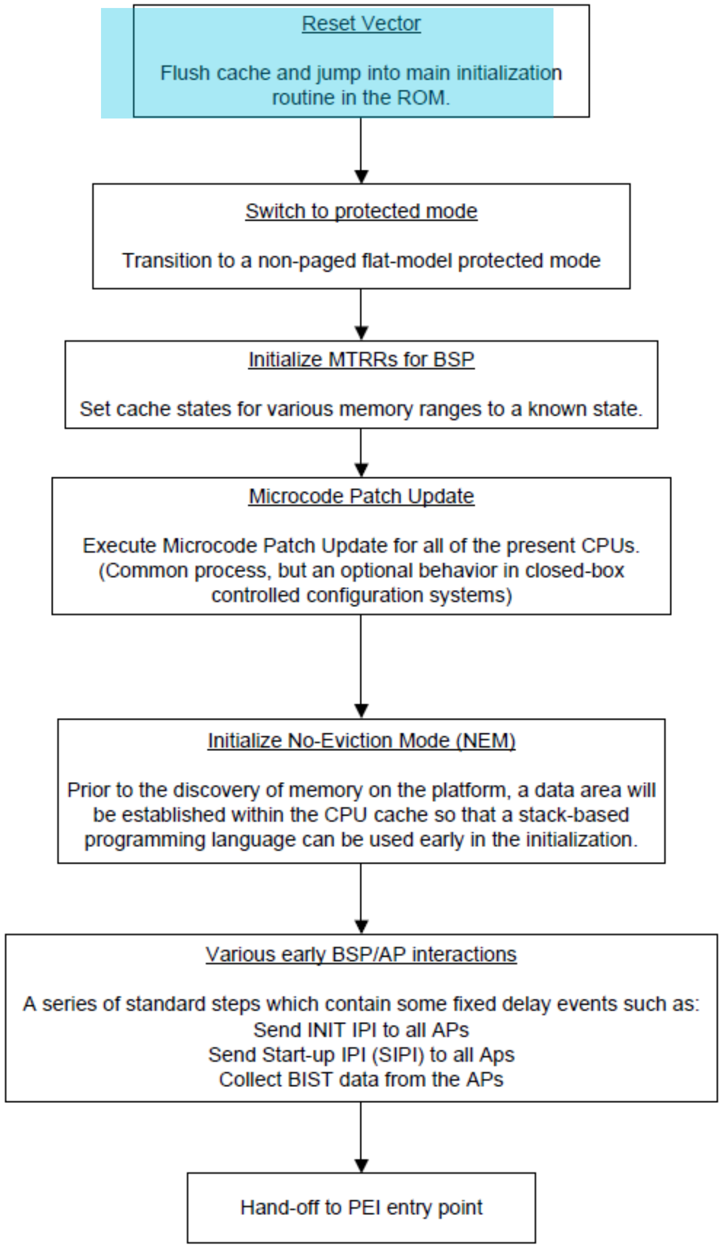
- This isn't a discussion of ACPI (big topic[1]), but it's important to note that alternate boot paths as determined by sleep mode could make the BIOS vulnerable
- A system that awakes from Sleep mode will follow a different path to boot
- This different code path may not lock down the system the same way as when the system boots from power down (or vice versa)
- i.e. your BIOS may be locked down when powered on from shutdown, but not when waking up from sleep
  - Found on real Dell systems. Patched. (And you all run out and apply the latest patches whenever they're released, right?) To be talked about at some point in the future.
  - Other new sleep issues coming soon too

[1]http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf

SEC Responsibilities 2 of 2

- **Implements a temporary memory store by configuring the CPU Cache as RAM (CAR)**
  - Also called "no evictions mode"
- **Memory has not yet been configured, so all read/writes must be confined to CPU cache**
- **A stack is implemented in CAR to pave the way for a C execution environment**
- **The processor active at boot time (Boot Strap Processor) is the one whose cache is used**
- **If you are interested in CAR, more info can be found here:**
  - http://www.coreboot.org/images/6/6c/LBCar.pdf

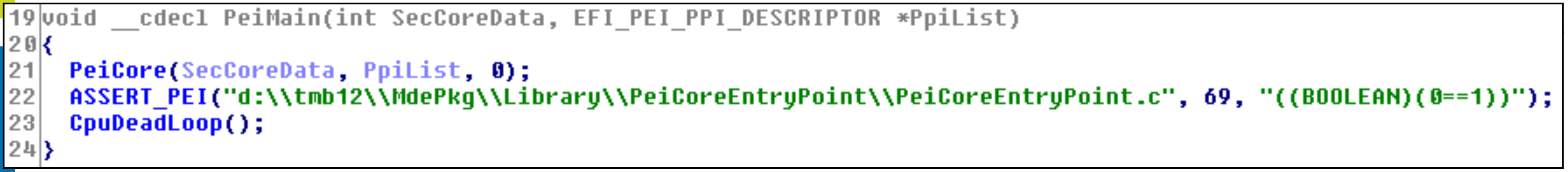
SEC Phase


Reset Vector
Flush cache and jump into main initialization routine in the ROM.
Switch to protected mode
Transition to a non-paged flat-model protected mode
Initialize MTRRs for BSP
Set cache states for various memory ranges to a known state.
Microcode Patch Update
Execute Microcode Patch Update for all of the present CPUs. (Common process, but an optional behavior in closed-box controlled configuration systems)
Initialize No-Eviction Mode (NEM)
Prior to the discovery of memory on the platform, a data area will be established within the CPU cache so that a stack-based programming language can be used early in the initialization.
Various early BSP/AP interactions
A series of standard steps which contain some fixed delay events such as:
Send INIT IPI to all APs
Send Start-up IPI (SIPI) to all Aps
Collect BIST data from the APs
Hand-off to PEI entry point

Intel whitepaper: Reducing Platform Boot Times, Rothman, Figure 1

- **Upon entry the environment is the same as on a legacy platform**
  - Hardware settings, not BIOS settings
- **Processor is in Real Mode**
- **Segment registers are the same**
  - CS:IP = F000:FFF0
  - CS.BASE = FFFF_0000h
- **Entry vector is still a JMP**
- **Note that microcode update is here, which could potentially mitigate exploitable microcode errata, by getting it patched early…assuming it is the kind of errata which gets a patch and assuming you have the latest BIOS w/ the latest microcode…**

SEC Hand-off to PEI Entry Point

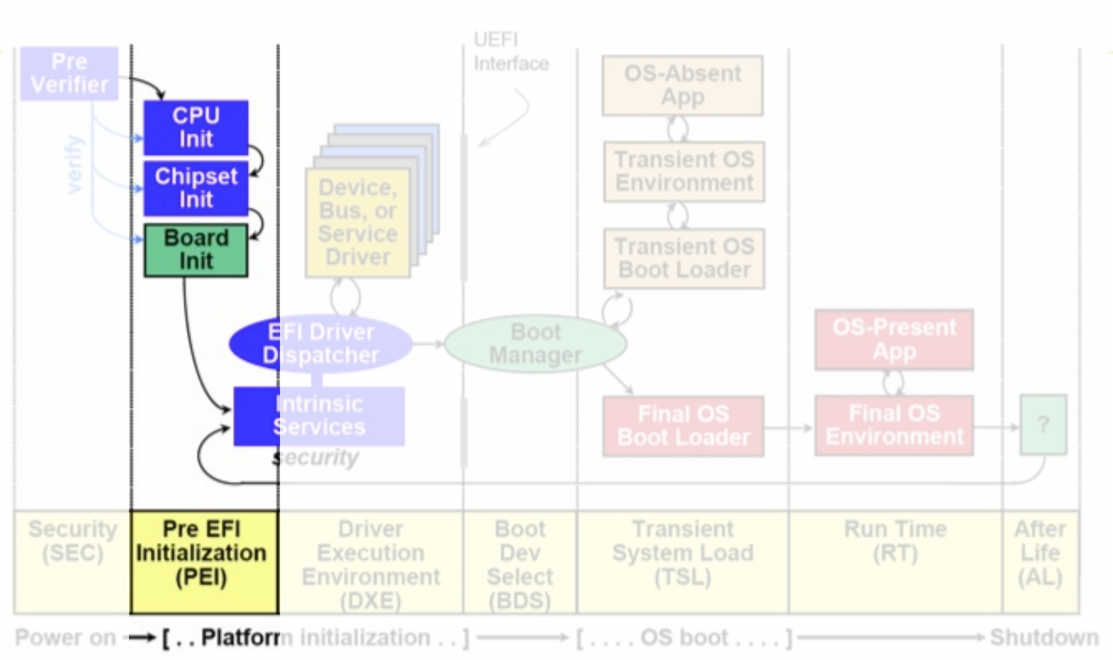


```
void __cdecl PeiMain(int SecCoreData, EFI_PEI_PPI_DESCRIPTOR *PpiList)
{
  PeiCore(SecCoreData, PpiList, 0);
  ASSERT_PEI("d:\\tmb12\\MdePkg\\Library\\PeiCoreEntryPoint\\PeiCoreEntryPoint.c", 69, "((BOOLEAN)(0==1))");
  CpuDeadLoop();
}
```

- **Passing handoff information to the PEI phase (to PeiCore):**
- **SEC Core Data**
  - Points to a data structure containing information about the operating environment:
  - Location and size of the temporary RAM
  - Location of the stack (in temp RAM)
  - Location of the Boot Firmware Volume (BFV)
    - Located in flash file system by its GUID
    - GUID: 8C8CE578-8A3D-4F1C-3599-35896185C32DD3
    - If not found, system halts
- **PPI List (defined in the upcoming PEI section)**
  - A list of PPI descriptors to be installed initially by the PEI Core
- **A void pointer for vendor-specific data (if any)**
- **Execution never returns to SEC until the next system reset**

Specified in Platform Initialization Spec Vol. 1, Version 1.3, Sec. 13
but the names are derived from the EDK2/UDK

PEI (Pre-EFI) Phase



- **The PEI phase primary responsibilities:**
  - Initialize permanent memory
  - Describe the memory to DXE in Hand-off-Blocks (HOBs)
  - Describe the firmware volume locations in HOBs
  - Pass control to DXE phase
  - Discover boot mode and, if applicable, resume from Sleep state
    - Code path will differ based on waking power state (S3, etc.)
    - Power states: http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf

Components of PEI

- **Pre-EFI Initialization Modules (PEIMs)**
  - A modular unit of code and/or data stored in a FFS file
  - Discover memory, Firmware Volumes, build the HOB, etc.
  - Can be dependent on PPIs having already been installed
    - Dependencies are inspected by the PEI Dispatcher
- **PEIM-to-PEIM Interface (PPI)**
  - Permit communication between PEIMs
    - So PEIMs can work with other PEIMs to achieve tasks and to enable code reuse
  - Contained in a structure EFI_PEI_PPI_DESCRIPTOR containing a GUID and a pointer
  - There are *Architectural PPIs* and *Additional PPIs*
  - Architectural PPIs: those which are known to the PEI Foundation (like that which provides the communication interface to the ReportStatusCode() PEI Service)
  - Additional PPIs: those which are not depended upon by the PEI Foundation.

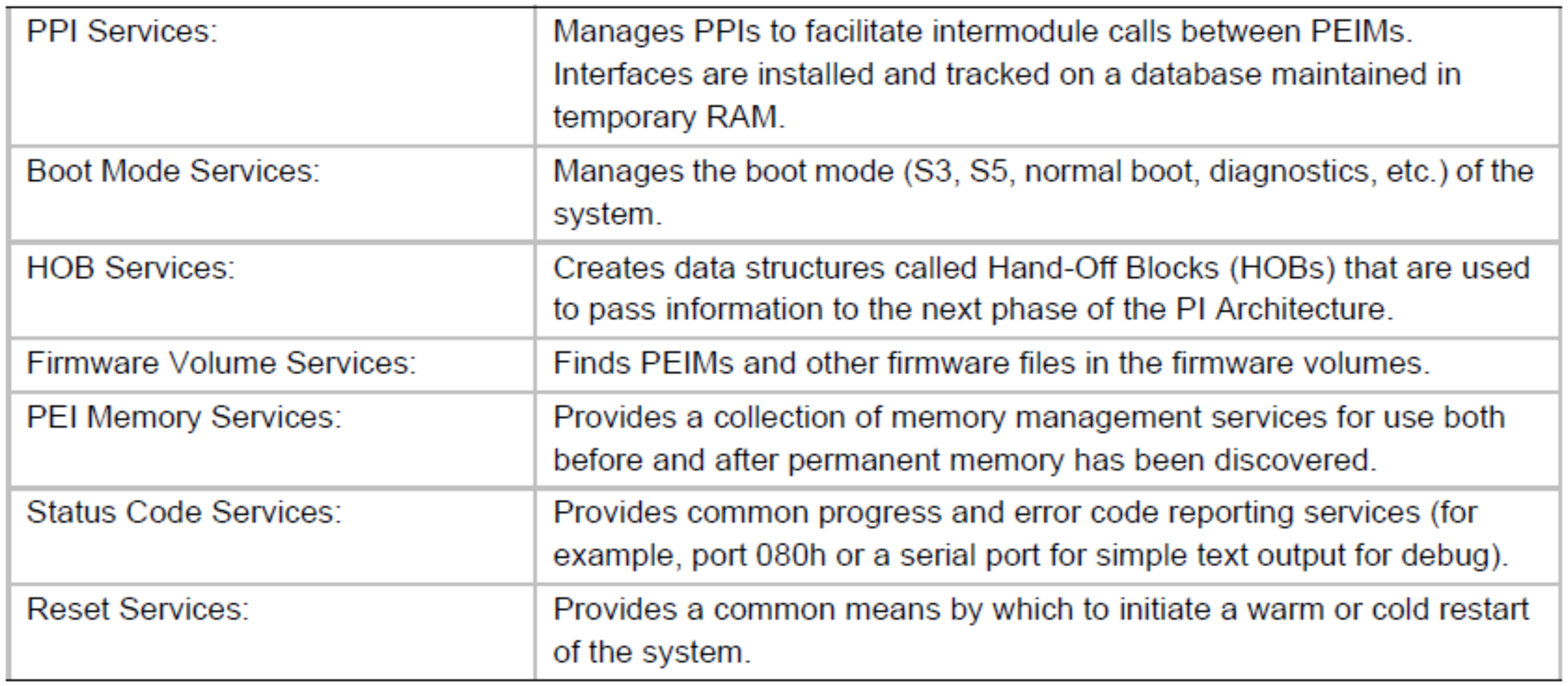
Components of PEI

- **PEI Dispatcher**
  - Evaluates the dependency expressions in PEIMs and, if they are met, installs them (and executes them)
- **Dependency Expression(DEPEX)**
  - Basically GUIDs of PPIs that must have already been dispatched before a PEIM is permitted to load/execute
- **Firmware Volumes**
  - Storage for the PEIMs, usually not compressed in this phase (but will be by DXE)

Components of PEI

- **PEI Services**
  - Available for use to all PEIMs and PPIs as well as the PEI foundation itself
  - Wide variety of services provided (InstallPpi(), LocateFv(), etc.)

Table 4. PEI Foundation Classes of Service



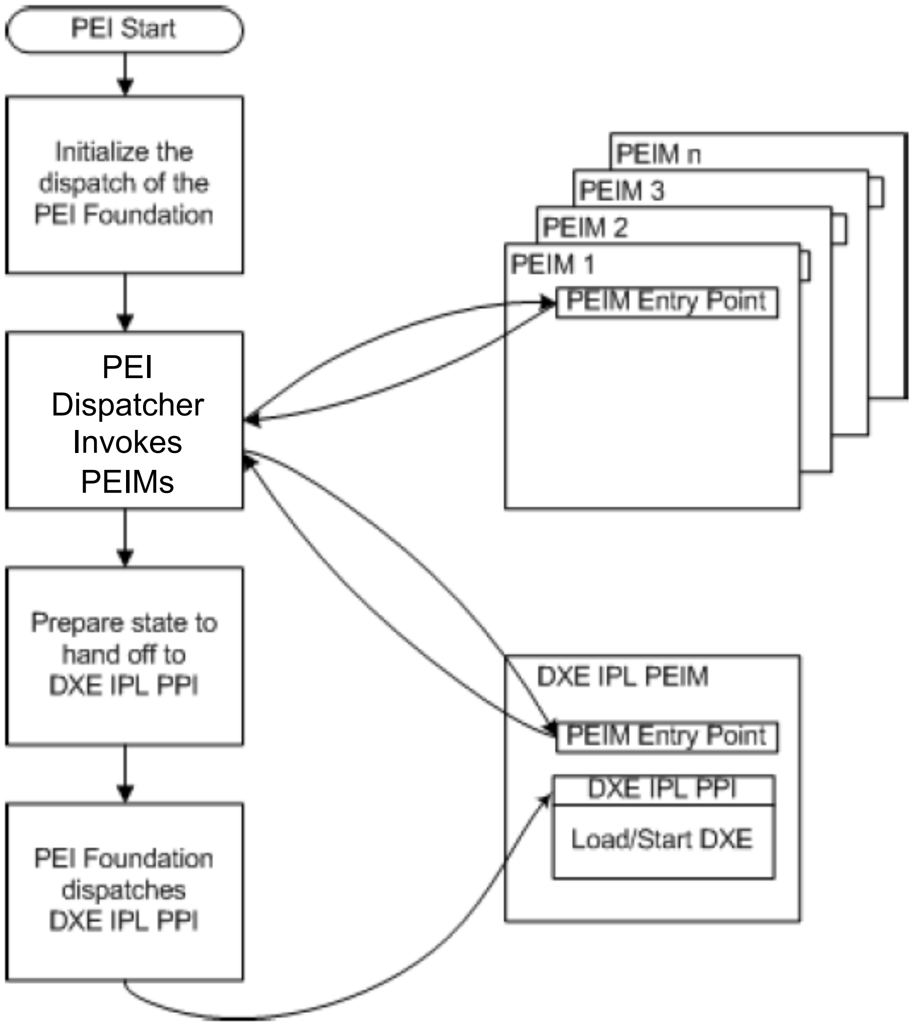
PPI Services:	Manages PPIs to facilitate intermodule calls between PEIMs. Interfaces are installed and tracked on a database maintained in temporary RAM.
Boot Mode Services:	Manages the boot mode (S3, S5, normal boot, diagnostics, etc.) of the system.
HOB Services:	Creates data structures called Hand-Off Blocks (HOBs) that are used to pass information to the next phase of the PI Architecture.
Firmware Volume Services:	Finds PEIMs and other firmware files in the firmware volumes.
PEI Memory Services:	Provides a collection of memory management services for use both before and after permanent memory has been discovered.
Status Code Services:	Provides common progress and error code reporting services (for example, port 080h or a serial port for simple text output for debug).
Reset Services:	Provides a common means by which to initiate a warm or cold restart of the system.

Extensive list of all PPIs can be found in Platform Initialization Spec Vol. 1, Version 1.3, Section 3.1

As the tables turn…

- Talk about the PEI services table

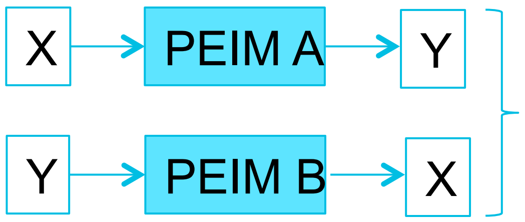
PEI Phase


PEI Start
Initialize the dispatch of the PEI Foundation
PEI Dispatcher Invokes PEIMs
Prepare state to hand off to DXE IPL PPI
PEI Foundation dispatches DXE IPL PPI
PEIM n
PEIM 3
PEIM 2
PEIM 1
PEIM Entry Point
DXE IPL PEIM
PEIM Entry Point
DXE IPL PPI
Load/Start DXE

- **This is a basic diagram of the PEI operations performed by the PEI Foundation**
- **The PEI foundation builds the PEI Services table**
- **The core of it centers around the PEI Dispatcher which locates and executes PEIMs**
  - Initializing permanent memory, etc.
- **The last PEIM to be dispatched will be the DXE IPL (Initial Program Load) PEIM, which will perform the transition to the DXE phase**

PEI Dispatcher

- **The PEI Dispatcher is basically a state machine and central to the PEI phase**
- **Evaluates each dependency expressions (list of PPIs) of PEIMs which are evaluated**
- **If the DEPEX evaluates to True, the PEIM is invoked, otherwise the Dispatcher moves on to evaluate the next PEIM**



UEFI will prevent both PEIMs A and B in this endless cycle from executing.
X and Y are PPIs

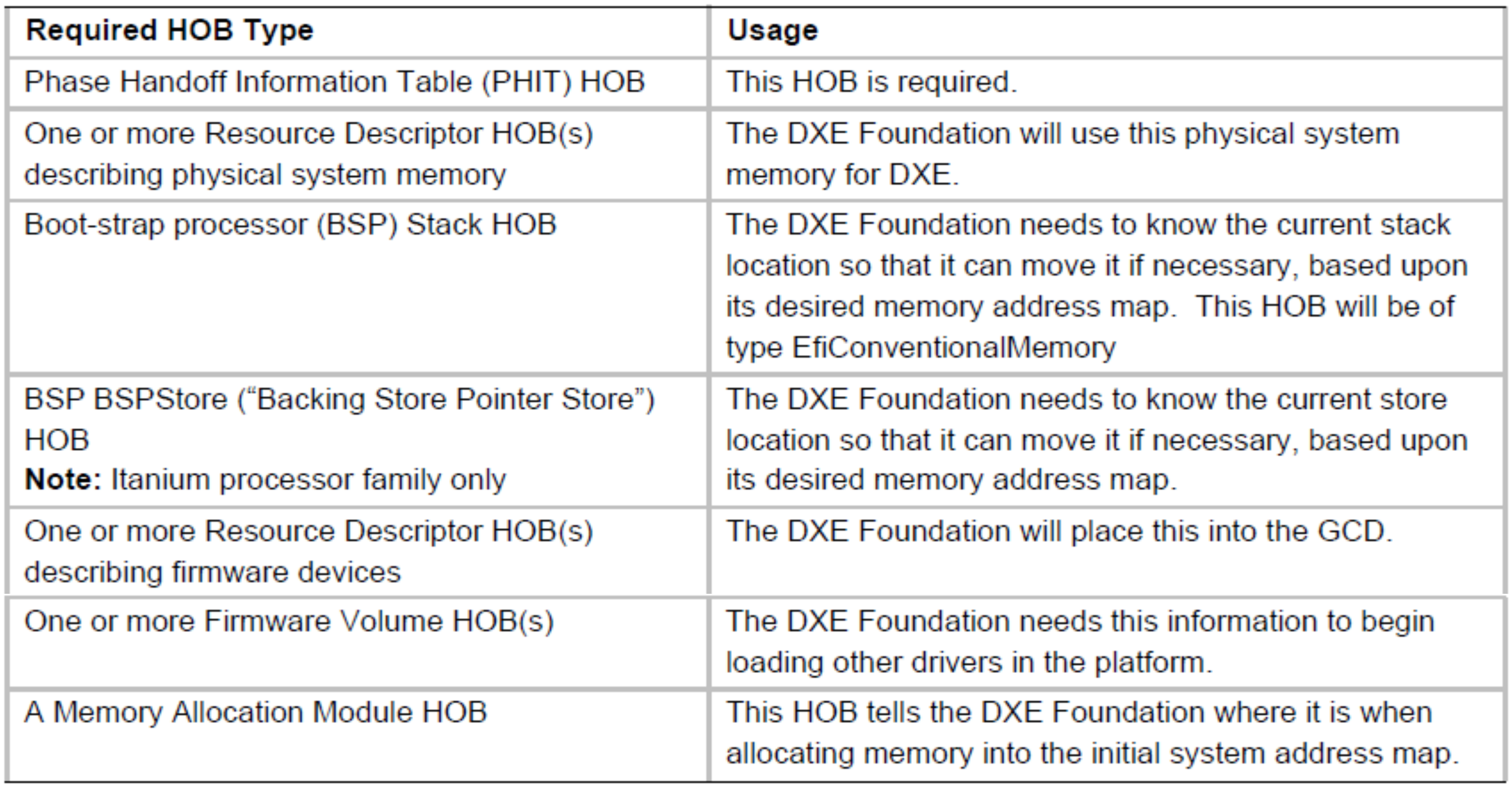
- **One PPI is EFI_FIND_FV_PPI so every PEIM on every Firmware Volume can be invoked**
- **Once all PEIMs that can execute have been, the last PEIM executed is the DXE IPL PEIM which hands off to DXE phase**

DEMO:

- **Showing our scripts reconstructing which PEIMs will be loaded in what order**
- **Don't forget the caveats:**

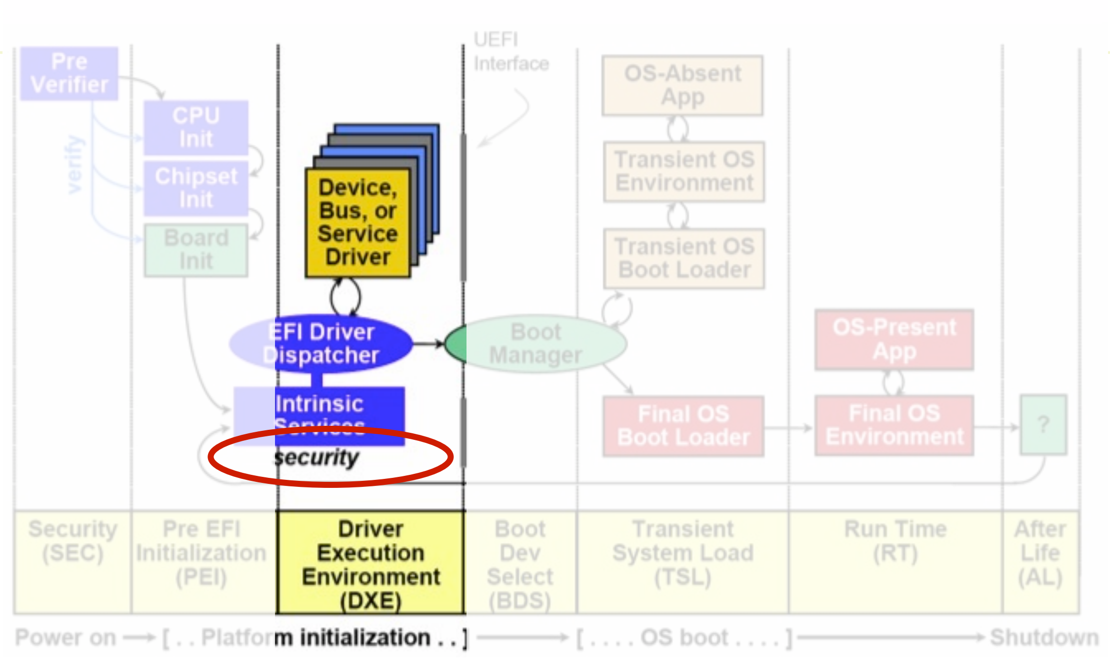
Exit conditions for handoff to DXE

- **The HOB List must contain the following HOBs:**



Required HOB Type	**Usage**
Phase Handoff Information Table (PHIT) HOB	This HOB is required.
One or more Resource Descriptor HOB(s) describing physical system memory	The DXE Foundation will use this physical system memory for DXE.
Boot-strap processor (BSP) Stack HOB	The DXE Foundation needs to know the current stack location so that it can move it if necessary, based upon its desired memory address map. This HOB will be of type EfiConventionalMemory
BSP BSPStore ("Backing Store Pointer Store") HOB **Note:** Itanium processor family only	The DXE Foundation needs to know the current store location so that it can move it if necessary, based upon its desired memory address map.
One or more Resource Descriptor HOB(s) describing firmware devices	The DXE Foundation will place this into the GCD.
One or more Firmware Volume HOB(s)	The DXE Foundation needs this information to begin loading other drivers in the platform.
A Memory Allocation Module HOB	This HOB tells the DXE Foundation where it is when allocating memory into the initial system address map.

Driver Execution Environment (DXE)

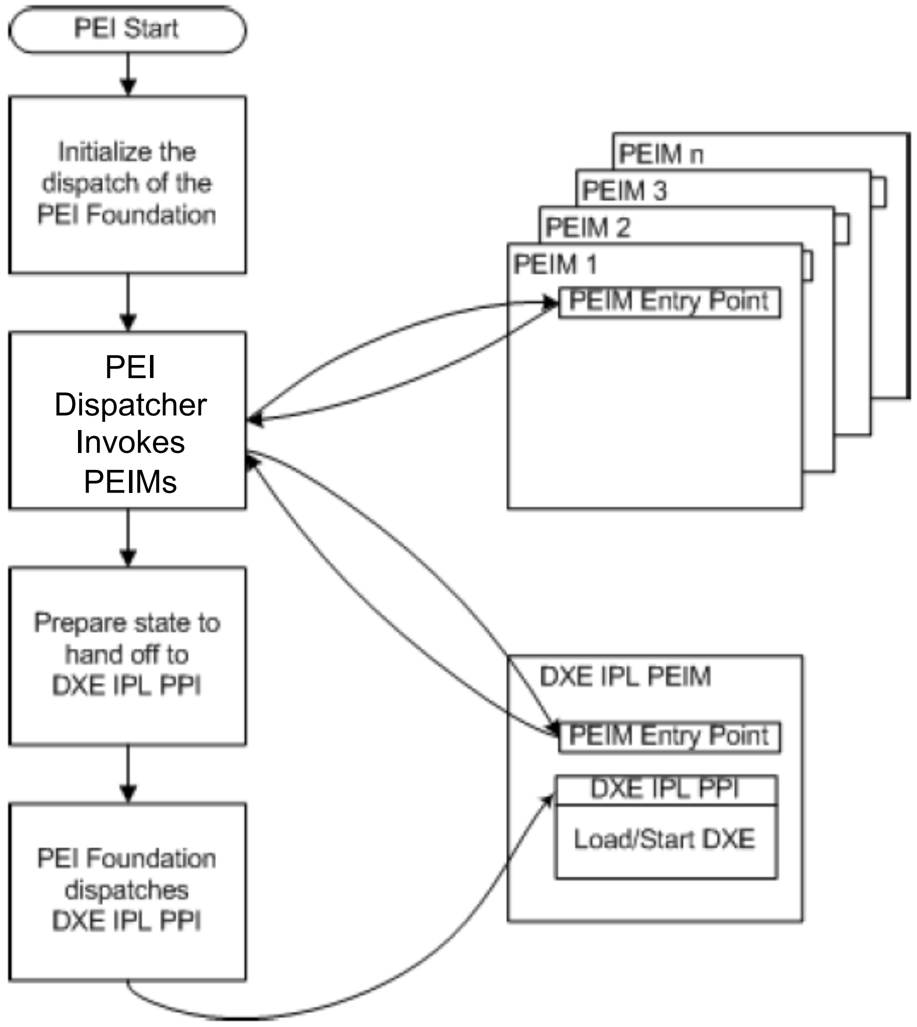


- **The DXE phase is designed to be executed at a high-enough level where it is independent from architectural requirements**
- **Similar to PEI from a high-level PoV: creates services used only by DXE, has a dispatcher that finds and loads DXE drivers, etc.**
- **System Management Mode set up, Secure Boot enforcement and BIOS update signature checks are typically implemented in this phase. Therefore it is the most security-critical.**

As the tables turn…

- Talk about the DXE services tables

DXE Phase


PEI Start
Initialize the dispatch of the PEI Foundation
PEI Dispatcher Invokes PEIMs
Prepare state to hand off to DXE IPL PPI
PEI Foundation dispatches DXE IPL PPI
PEIM n
PEIM 3
PEIM 2
PEIM 1
PEIM Entry Point
DXE IPL PEIM
PEIM Entry Point
DXE IPL PPI
Load/Start DXE

- Use this for mental visualization, but
- s/PEI/DXE/g
- s/PEIM/DXE Driver/g
- s/DXE IPL/BDS IPL/g

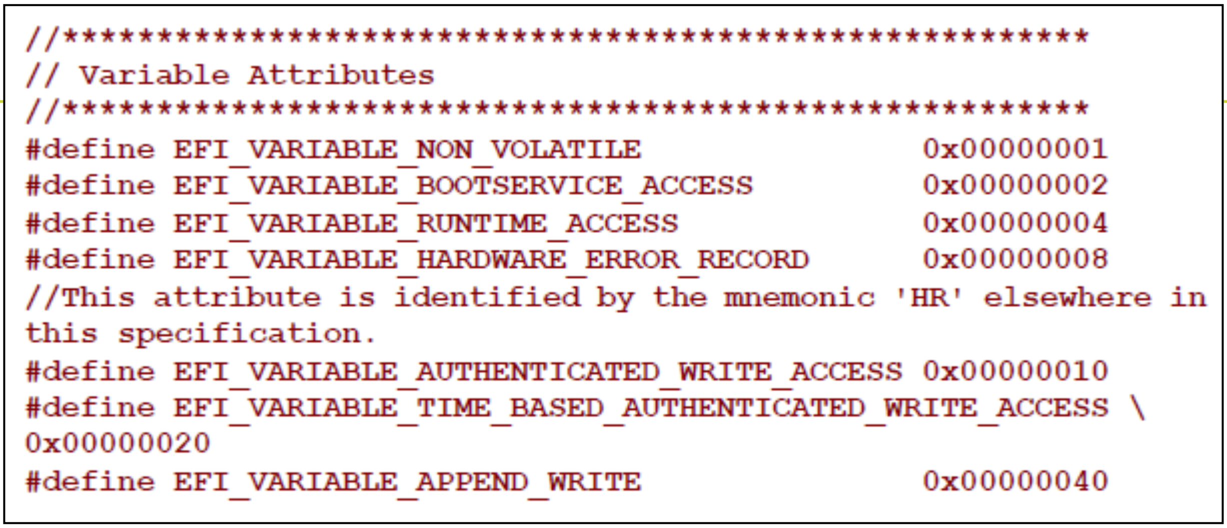
MITRE

Relative magnitude

- Lenovo X1 Carbon: 73 PEIMs, 334 DXE drivers
- Lenovo X240: 80 PEIMs, 352 DXE drivers
- HP Elitebook 2540p (201?): 42 PEIMs, 164 DXE drivers
- HP Elitebook 850 G1 (2014): 117 PEIMs, 392 DXE drivers
- Dell Latitude E6410: 32 PEIMs, ? DXE drivers
- Dell Latitude E6440: ? PEIMs, ? DXE drivers

- DXE has a whole lot of stuff going on!

EFI Variable Attributes



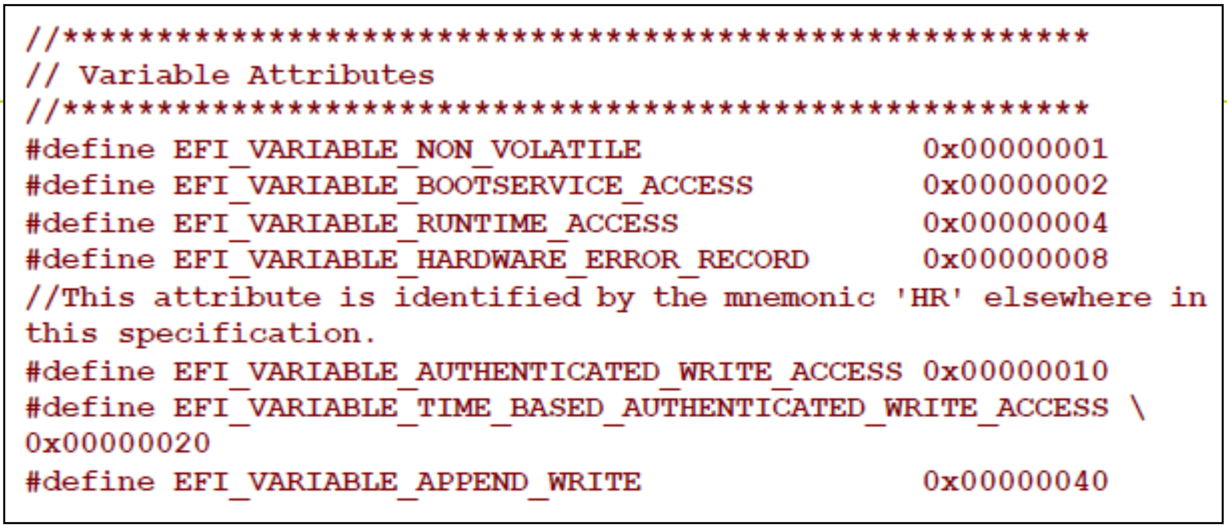
```
//*********************************************************
// Variable Attributes
//*********************************************************
#define EFI_VARIABLE_NON_VOLATILE                0x00000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS          0x00000002
#define EFI_VARIABLE_RUNTIME_ACCESS              0x00000004
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD       0x00000008
//This attribute is identified by the mnemonic 'HR' elsewhere in
this specification.
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS \
0x00000020
#define EFI_VARIABLE_APPEND_WRITE                0x00000040
```

- **Each UEFI variable has attributes that determine how the firmware stores and maintains the data:**
- **‘Non_Volatile’**
  – The variable is stored on flash
- **‘Bootservice_Access’**
  – Can be accessed/modified during boot. Must be set in order for Runtime_Access to also be set

* UEFI 2.3.1 Errata C Final

MITRE

EFI Variable Attributes

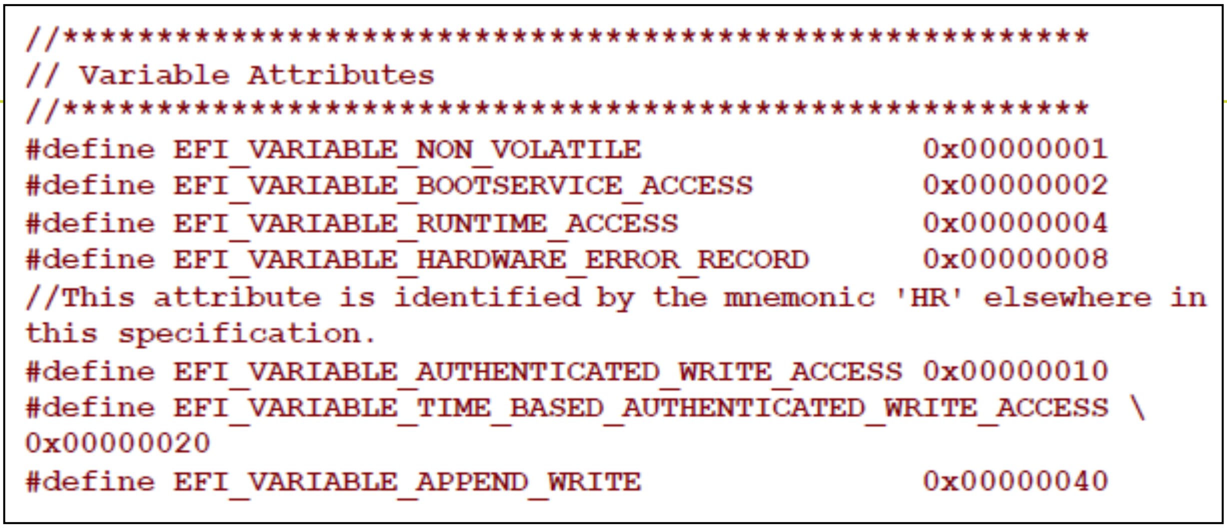


```
//*******************************************************
// Variable Attributes
//*******************************************************
#define EFI_VARIABLE_NON_VOLATILE                    0x00000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS              0x00000002
#define EFI_VARIABLE_RUNTIME_ACCESS                  0x00000004
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD           0x00000008
//This attribute is identified by the mnemonic 'HR' elsewhere in
this specification.
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS \
0x00000020
#define EFI_VARIABLE_APPEND_WRITE                    0x00000040
```

- **‘Runtime_Access’**
  – The variable can be accessed/modified by the Operating System or an application
- **‘Hardware_Error_Record’**
  – Variable is stored in a portion of NVRAM (flash) reserved for error records

* UEFI 2.3.1 Errata C Final

EFI Variable Attributes

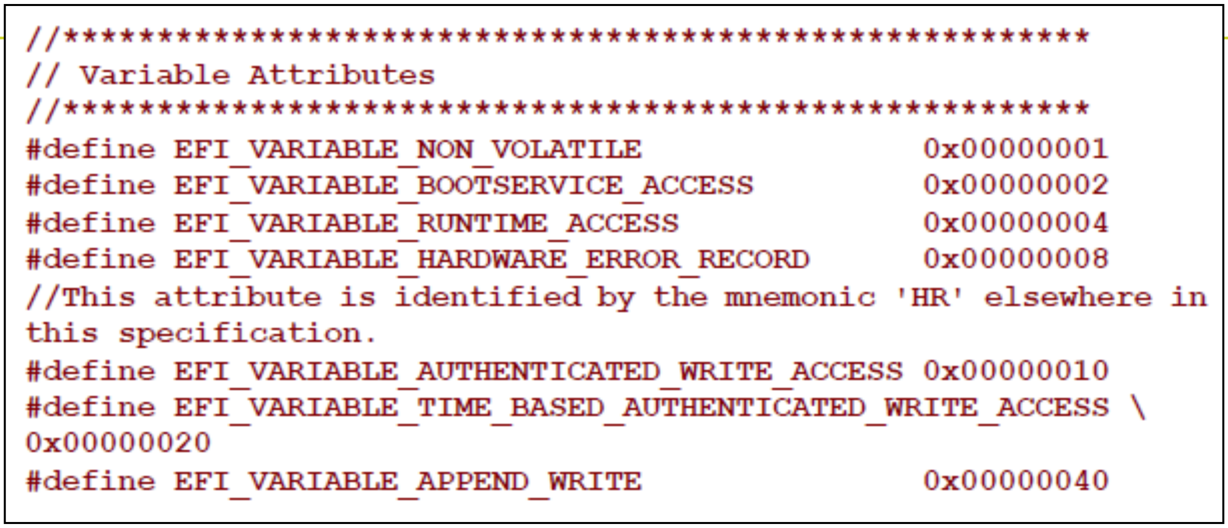


```
//*********************************************************
// Variable Attributes
//*********************************************************
#define EFI_VARIABLE_NON_VOLATILE                  0x00000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS            0x00000002
#define EFI_VARIABLE_RUNTIME_ACCESS                0x00000004
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD         0x00000008
//This attribute is identified by the mnemonic 'HR' elsewhere in
this specification.
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS \
0x00000020
#define EFI_VARIABLE_APPEND_WRITE                  0x00000040
```

- **‘Authenticated_Write_Access’**
  - The variable can be modified only by an application that has been signed with an authorized private key (or by present user)
  - KEK and DB are examples of Authorized variables
- **‘Time_Based_Authenticated_Write_Access’**
  - Variable is signed with a time-stamp
- **‘Append_Write’**
  - Variable may be appended with data

* UEFI 2.3.1 Errata C Final

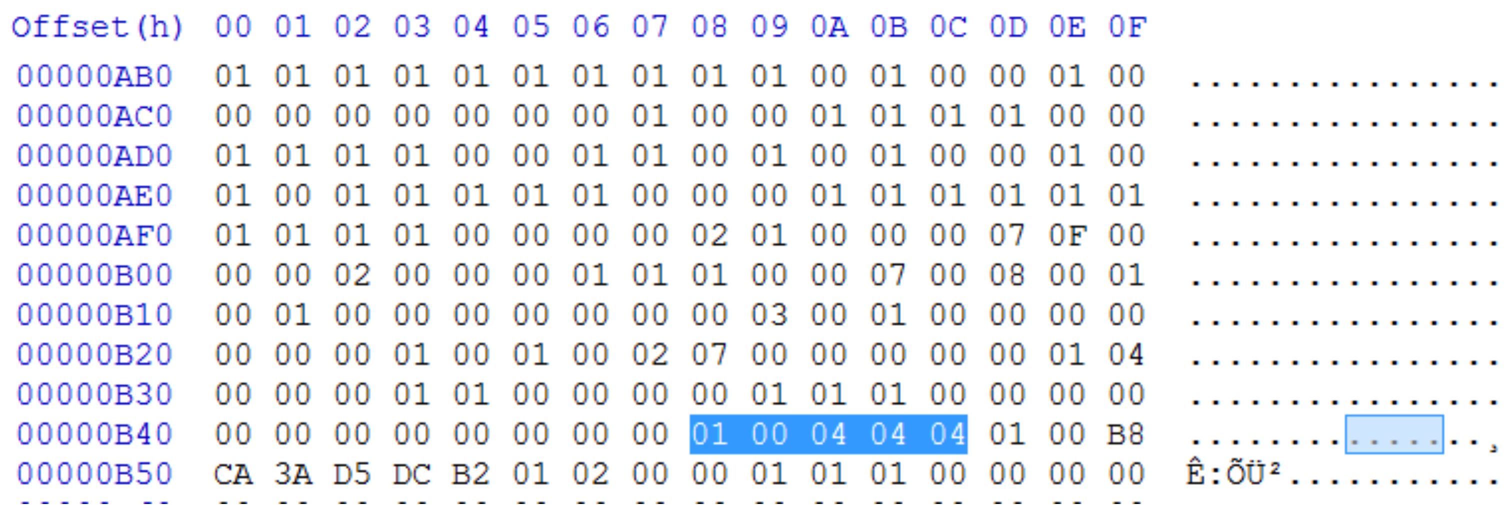
EFI Variable Attributes Combinations



```
//*********************************************************
// Variable Attributes
//*********************************************************
#define EFI_VARIABLE_NON_VOLATILE                  0x00000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS            0x00000002
#define EFI_VARIABLE_RUNTIME_ACCESS                0x00000004
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD         0x00000008
//This attribute is identified by the mnemonic 'HR' elsewhere in
this specification.
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS 0x00000010
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS \
0x00000020
#define EFI_VARIABLE_APPEND_WRITE                  0x00000040
```

- **If a variable is marked as both Runtime and Authenticated, the variable can be modified only by an application that has been signed with an authorized key**
- **If a variable is marked as Runtime but not as Authenticated, the variable can be modified by any application**
  – The Setup variable is marked like this

EFI Setup Variable Data



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000AB0  01 01 01 01 01 01 01 01 01 01 00 01 00 00 01 00  ................
00000AC0  00 00 00 00 00 00 00 01 00 00 01 01 01 01 00 00  ................
00000AD0  01 01 01 01 00 00 01 01 00 01 00 01 00 00 01 00  ................
00000AE0  01 00 01 01 01 01 01 00 00 00 01 01 01 01 01 01  ................
00000AF0  01 01 01 01 00 00 00 00 02 01 00 00 00 07 0F 00  ................
00000B00  00 00 02 00 00 00 01 01 01 00 00 07 00 08 00 01  ................
00000B10  00 01 00 00 00 00 00 00 00 03 00 01 00 00 00 00  ................
00000B20  00 00 00 01 00 01 00 02 07 00 00 00 00 00 01 04  ................
00000B30  00 00 00 01 01 00 00 00 00 01 01 01 00 00 00 00  ................
00000B40  00 00 00 00 00 00 00 00 01 00 04 04 04 01 00 B8  ...............¸
00000B50  CA 3A D5 DC B2 01 02 00 00 01 01 01 00 00 00 00  Ê:ÕÜ²...........
```

- **Using the offsets we calculated earlier we can locate the secure boot policy settings in the Setup variable**
- **The Secure Boot policy settings started at offset 0xB49 from the start of the Setup variable data**
- **Byte B49 contains the “IMAGE_FROM_FV” policy and is set to ALWAYS_EXECUTE (0x00)**
- **Bytes B4A-B4C contain the policies pertaining to Option ROMs, Removable Storage, and Fixed Storage, respectively. All are set to “DENY_EXECUTE_ON_SECURITY_VIOLATION**
  – We can change these to ALWAYS_EXECUTE (00)
- **Byte B48 contains the Secure Boot on/off value (on)**

UEFI Variables (Keys and Key Stores)

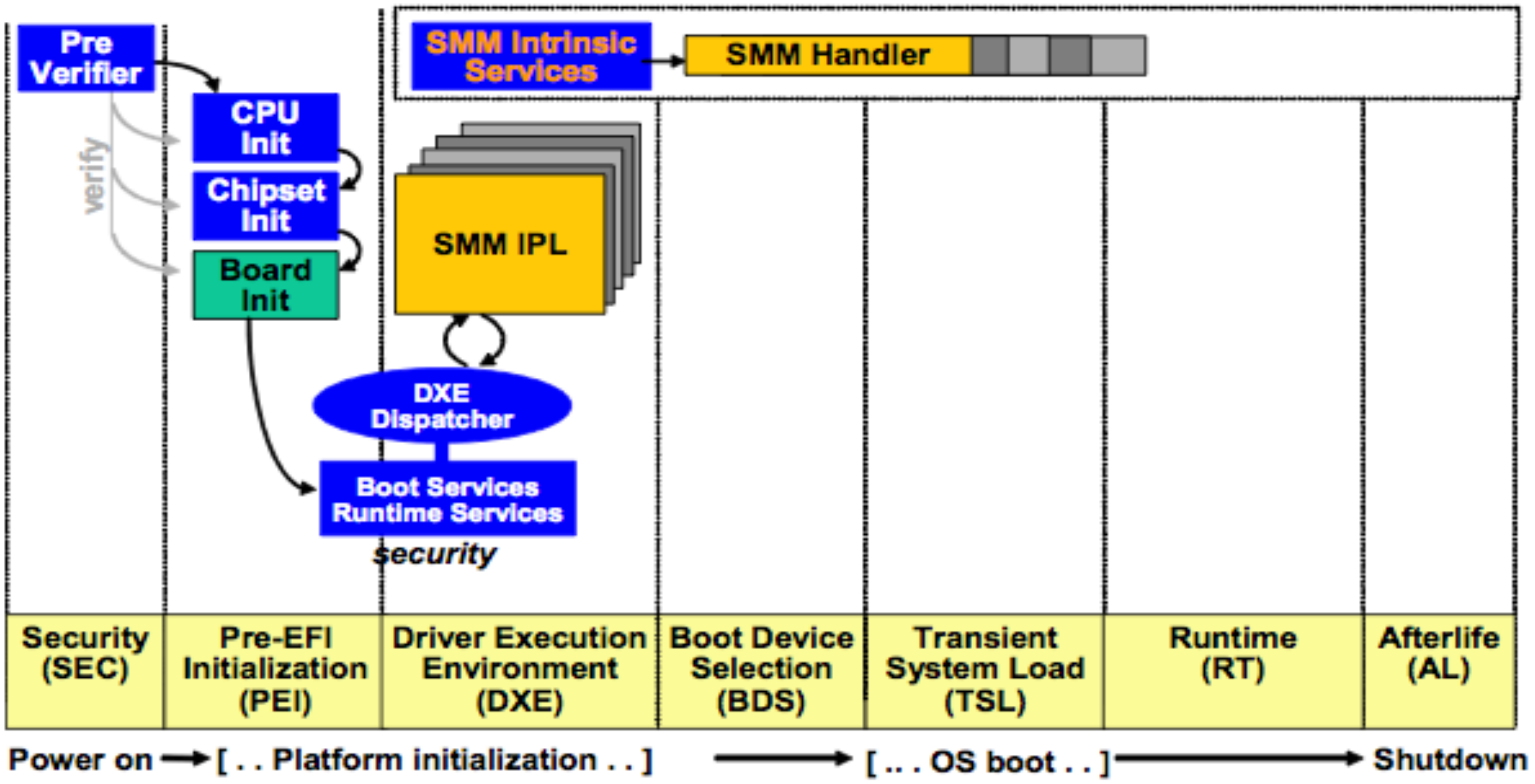
- **UEFI implements 4 “variables” which store keys, signatures, and/or hashes:**
- **Platform Key (PK)**
  - "The platform key establishes a trust relationship between the platform owner and the platform firmware."
  - Controls access to itself and the KEK variables
  - Only a physically present user or an application which has been signed with the PK is supposed to be able to modify this variable
  - Required to implement Secure Boot, otherwise the system is in Setup Mode where keys can be trivially modified by any application
- **Key Exchange Key (KEK)**
  - "Key exchange keys establish a trust relationship between the operating system and the platform firmware."
  - Used to update the signature database
  - Used to sign .efi binaries so they may execute
- **Signature Database (DB)**
  - A whitelist of keys, signatures and/or hashes of binaries
- **Forbidden Database (DBX)**
  - A blacklist of keys, signatures, and/or hashes of binaries

UEFI Version 2.3.1, Errata C

UEFI Variables (Keys and Key Stores)

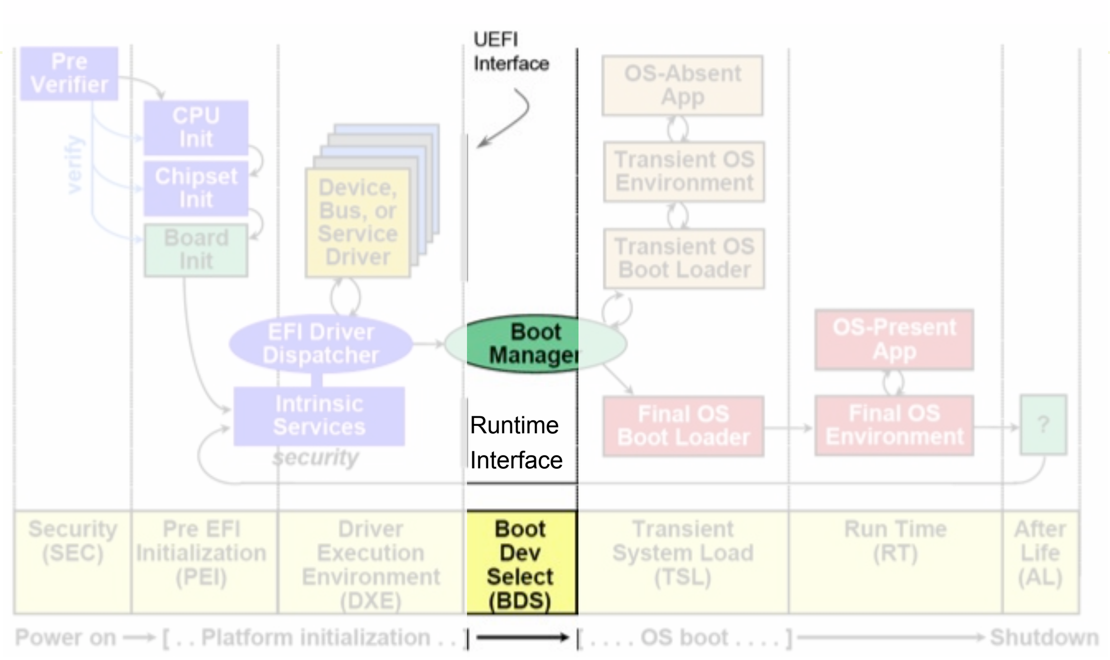
- As stated earlier, these variables are stored on the Flash file system
- Thus, if the SPI flash isn’t locked down properly, these keys/hashes can be overwritten by an attacker
- The problem is, the UEFI variables must rely solely on SMM to protect them!
- The secondary line of defense, the Protected Range registers cannot be used
- The UEFI variables must be kept writeable because at some point the system is going to need to write to them
- We saw this yesterday in the Charizard video where my colleague Sam suppressed SMI and wrote directly to the flash BIOS to add the hash of a malicious boot loader to the DB whitelist

DXE & SMM, BFF 4EVA!



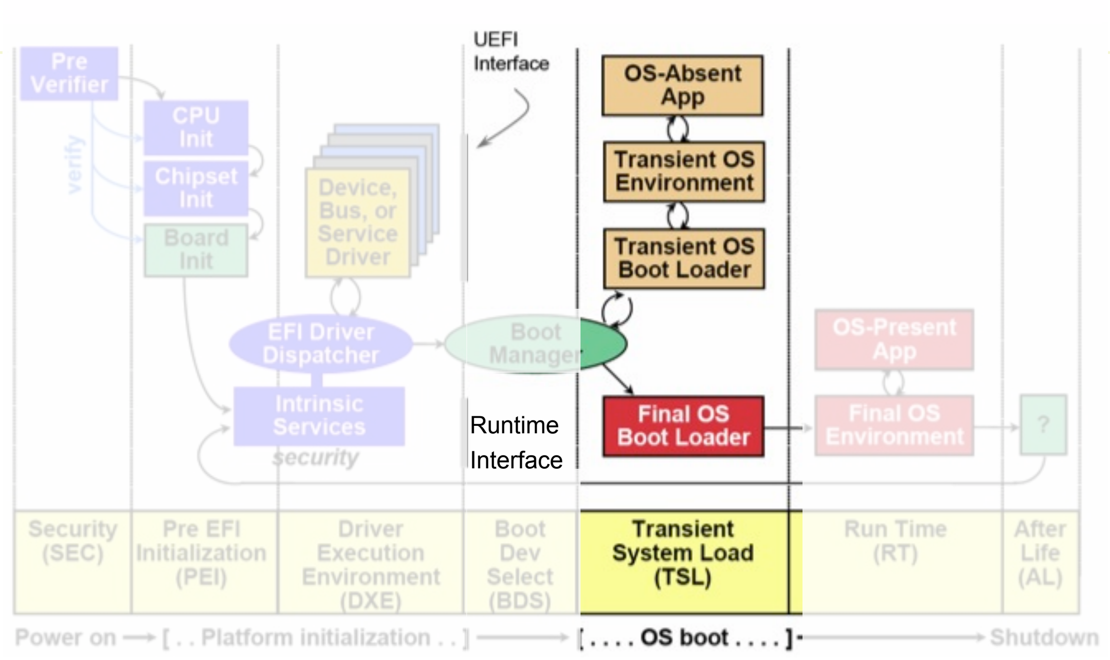
- DXE loads SMM IPL
- SMM IPL loads SMM Core
- SMM Core loads SMM drivers

Boot Device Selection (BDS)



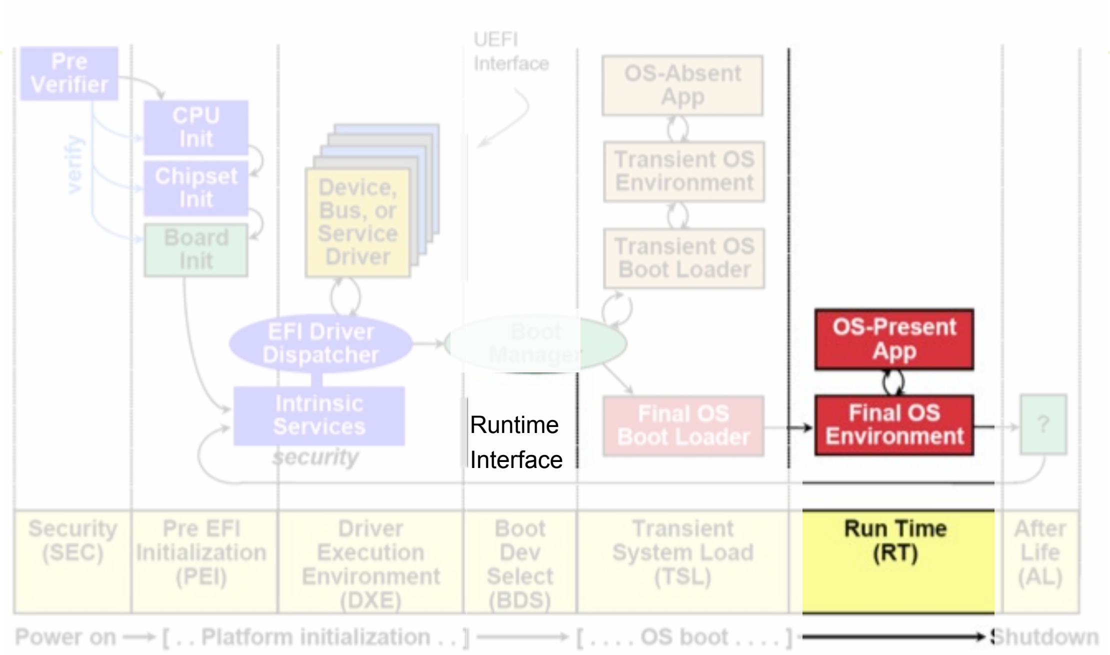
- **The BDS will typically be encapsulated into a single file loaded by the DXE phase.**
- **It consults the configuration information to decide whether you're going to boot an OS or "something else"**
- **It has access to the full UEFI Boot Services Table of services that DXE set up. E.g. HD filesystem access to find an OS boot loader**
  – So that should tell you an attacker in DXE gets that capability at some point

Transient System Load (TSL)



- This is the point where we hand off from firmware-derived code, to typically HD-stored code.
- If the system is running with SecureBoot turned on, the BDS will have checked the signature before loading code in this phase, and denied anything un-signed (e.g. a super lame "Oooh look at me, I made the first UEFI bootkit!!1" bootkit ;))

Run Time (RT)

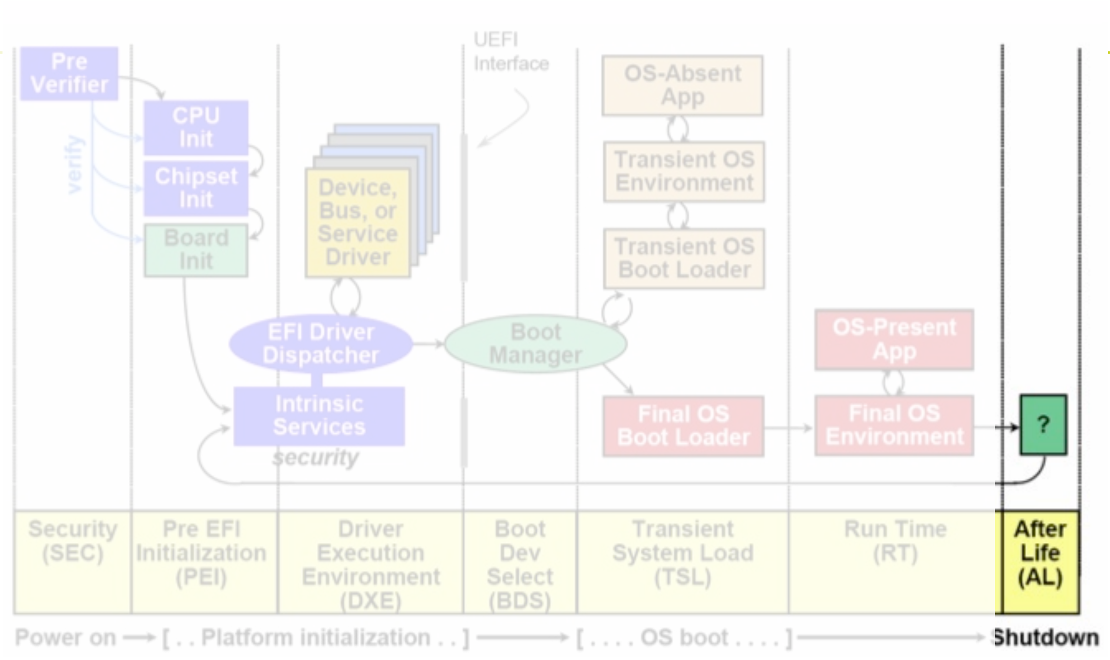


- Typically when the OS boot loader is done, it will call ExitBootServices() in the UEFI Boot Services table. This will reclaim the majority of UEFI memory so the OS can use it
- However some memory is retained, to be used for the Runtime Services Table. This is an interface that the firmware provides to the OS.

As the tables turn…

- Talk about the runtime services table

After Life (AL)



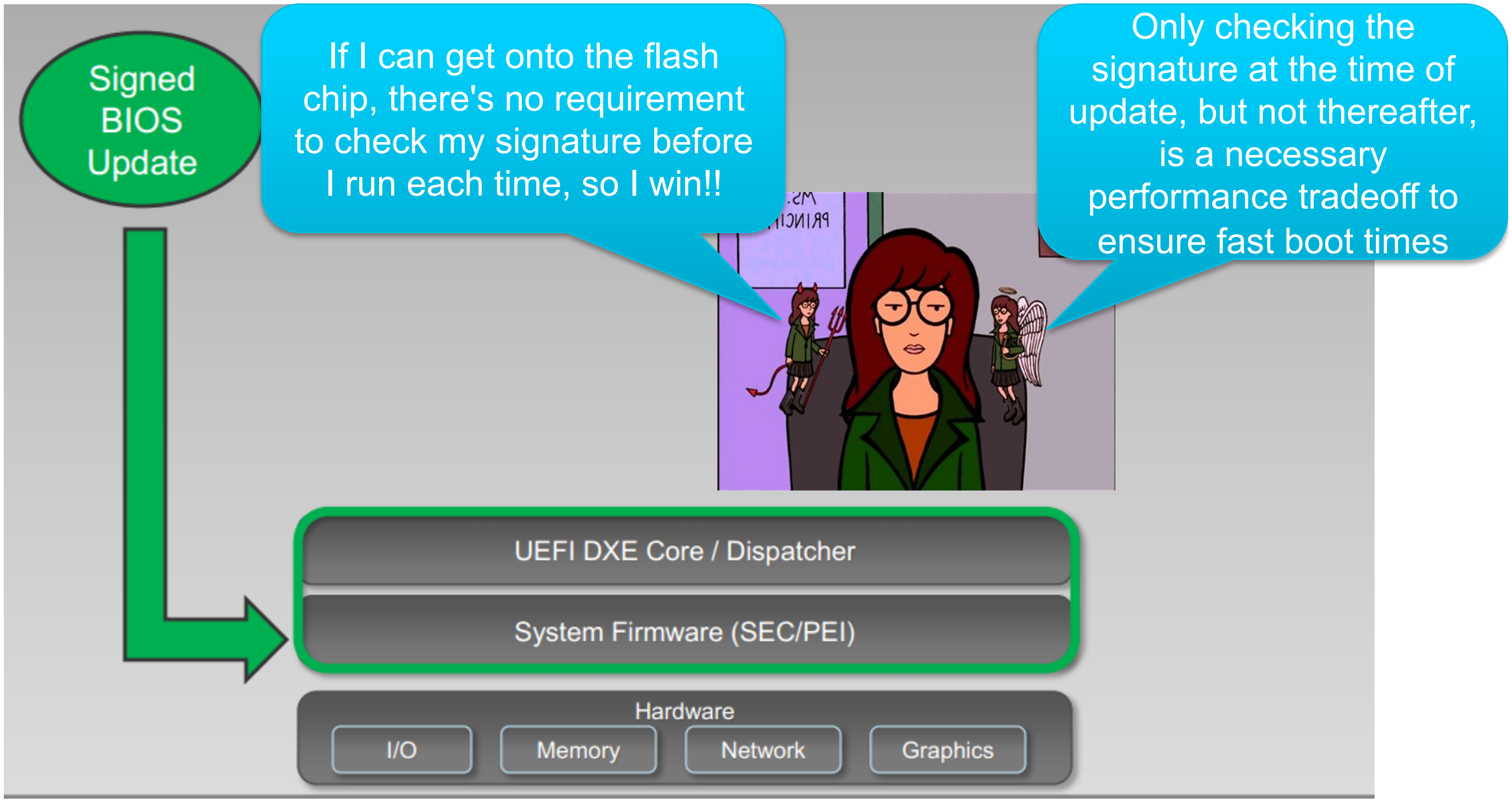
- We haven't checked extensively, but we don't think anyone is doing anything with this right now
- We think it's just something put there so that architecturally they would have the option to do "stuff" upon graceful shutdown (e.g. clearing secrets?)

Intro to UEFI Secure Boot

Intro to UEFI Secure Boot

- **Verifies whether an executable is permitted to load and execute during the UEFI BIOS boot process**
- **When an executable like a boot loader or Option ROM is discovered, the UEFI checks if:**
  - The executable is signed with an authorized key, or
  - The key, signature, or hash of the executable is stored in the authorized signature database
- **UEFI components that are flash based (SEC, PEI, DXECore) are not verified for signature**
  - The BIOS flash image has its signature checked during the update process (firmware signing)
- **Yuriy Bulygin, Andrew Furtak, and Oleksandr Bazhaniuk have the best slides that describe the Secure Boot process**
  - http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf (Black Hat USA 2013)

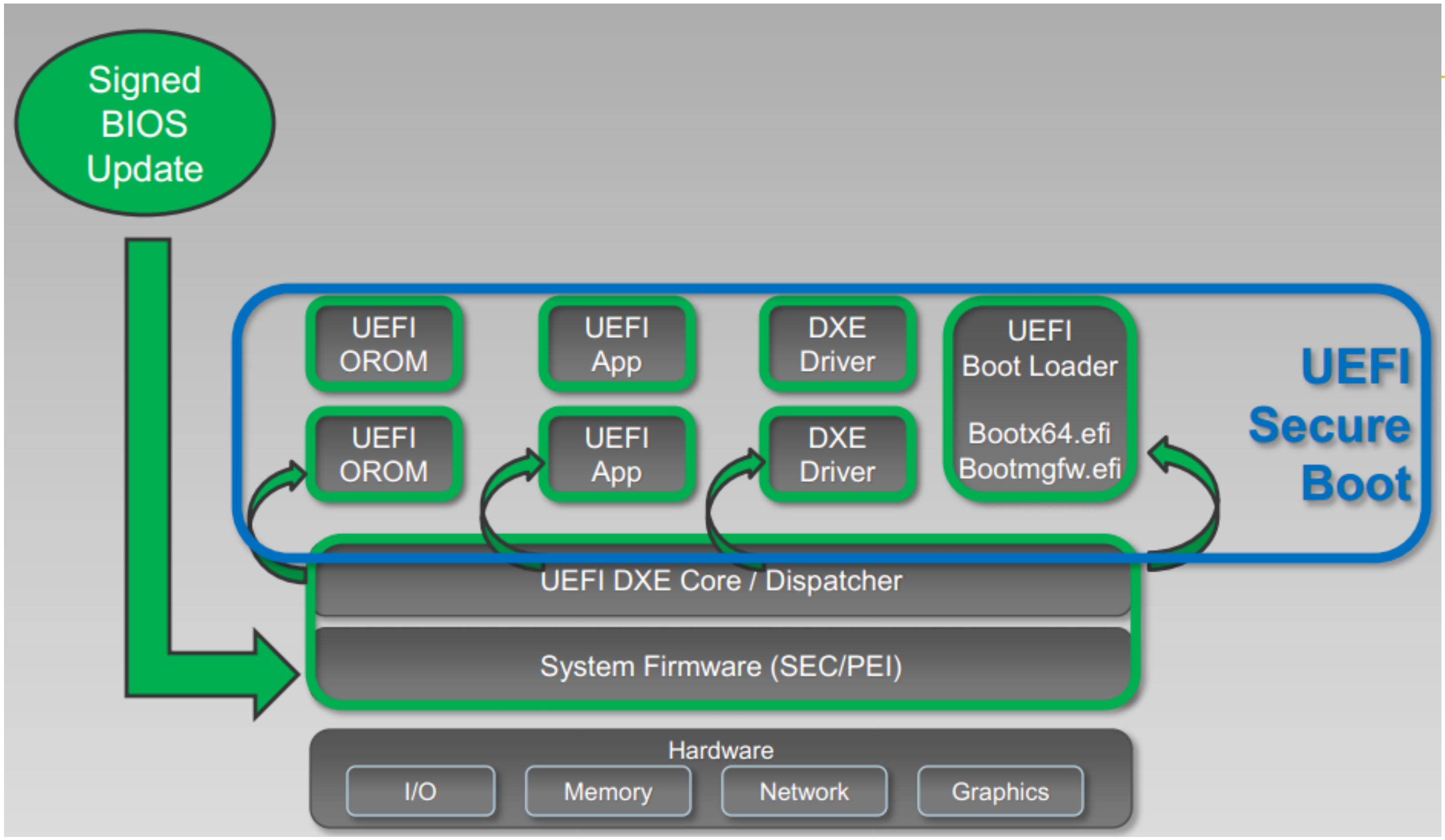
Firmware Signing


Signed BIOS Update
If I can get onto the flash chip, there's no requirement to check my signature before I run each time, so I win!!
Only checking the signature at the time of update, but not thereafter, is a necessary performance tradeoff to ensure fast boot times
UEFI DXE Core / Dispatcher
System Firmware (SEC/PEI)
Hardware
I/O
Memory
Network
Graphics

- **Flash-based UEFI components are verified only during the update process when the whole BIOS image has its signature verified**

http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf

UEFI Secure Boot

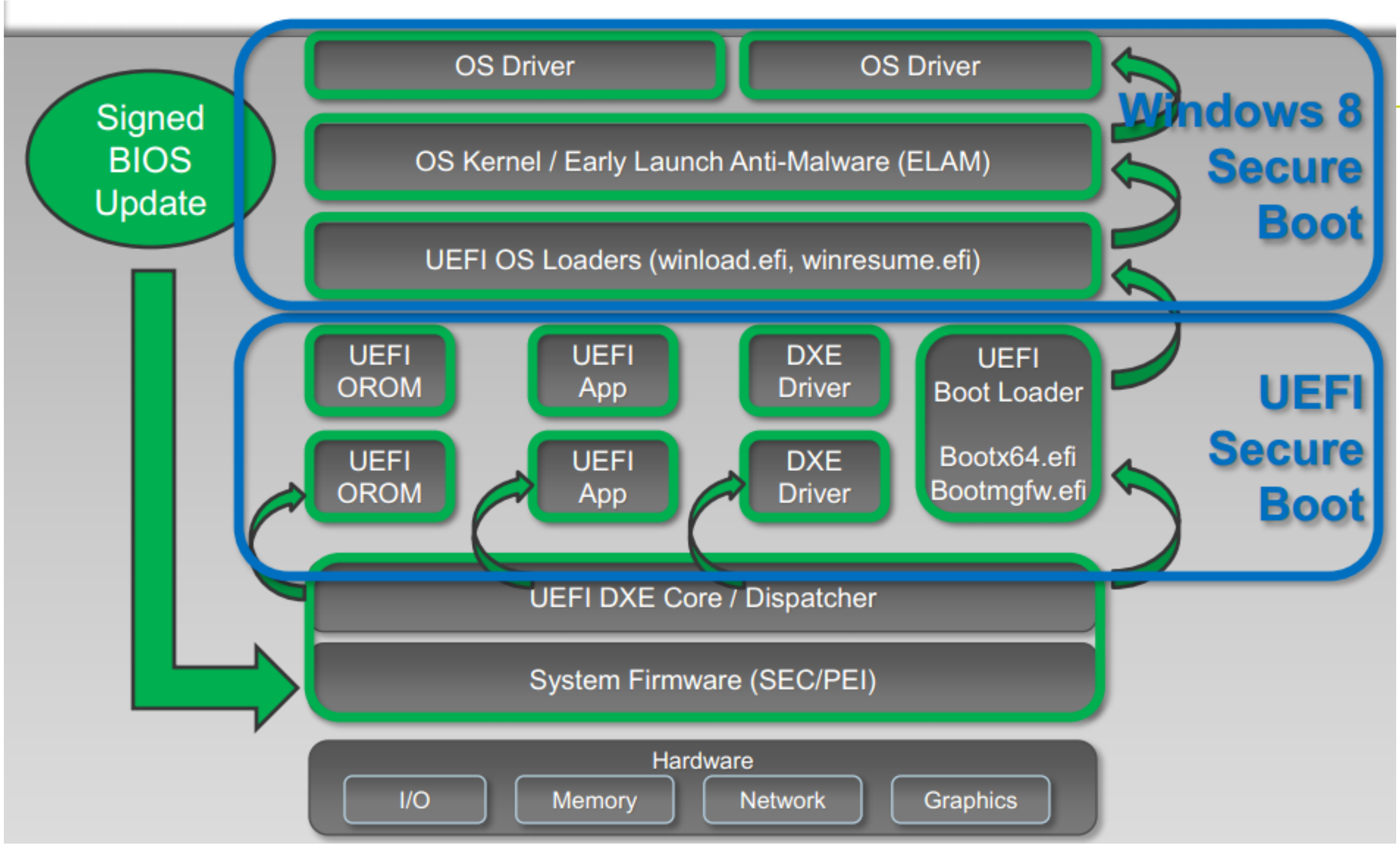


- **DXE verifies non-embedded XROMs, DXE drivers, UEFI applications and boot loader(s)**
- **This is the UEFI Secure Boot process**

http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf

MITRE

Windows 8 Secure Boot



- Microsoft Windows 8 adds to the UEFI secure boot process
- Establishes a chain of verification
- UEFI Boot Loader -> OS Loader -> OS Kernel -> OS Drivers

http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf

MITRE

UEFI SecureBoot makes it easier to keep out low-level attackers like bootkits, which are a serious threat that undercuts OS security…

When I inevitably break SecureBoot (like I've broken every other access control mechanism ever), they won't think to look for me down there, because they'll be complacent and think it's already secured

MITRE

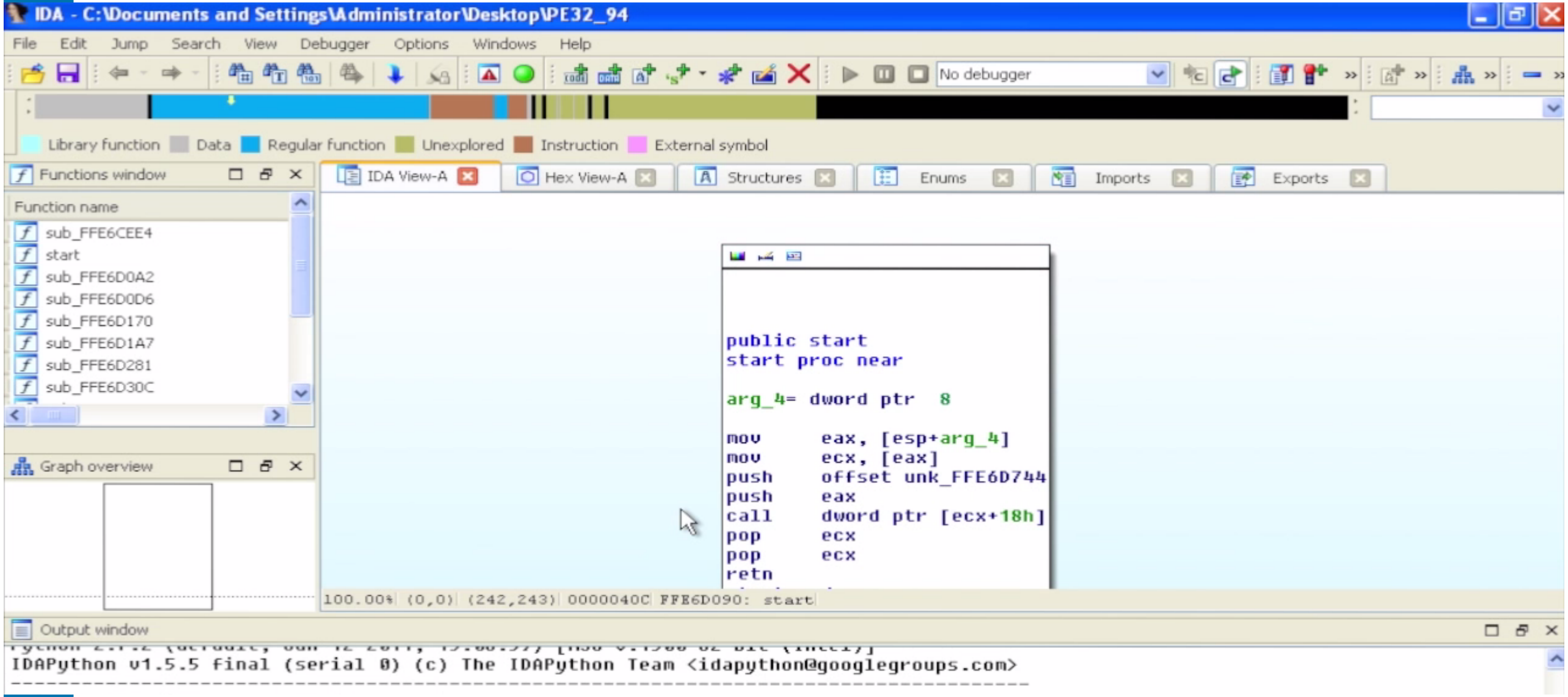
Analyzing UEFI Files with IDA

Demo: Back to the FFS!

- Now that you know a bit more, let's look a bit more

Making sense of UEFI PE files in IDA Pro

- You can watch a 15 minute example of super basic analysis here
- https://www.youtube.com/watch?v=R-5UO6jLkEI



```
public start
start proc near

arg_4= dword ptr  8

mov     eax, [esp+arg_4]
mov     ecx, [eax]
push    offset unk_FFE6D744
push    eax
call    dword ptr [ecx+18h]
pop     ecx
pop     ecx
retn
```

```
100.00% (0,0) (242,243) 0000040C FFE6D090: start
```

```
IDAPython v1.5.5 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
-----------------------------------------------------------------------------------
```

MITRE

Demo: You don't *have* to have private 1337sauce… but it helps :)

- **Thanks to my wife Veronica Kovah who put together these scripts for us in her free time**
- **Some example IDA scripts:**
  - Finding where PEIM "PPIs" are registered/used
  - Finding where DXE "Protocols" are registered/used
  - Other…

Questions?

- Thanks for listening!

- Email contact:

{xkovah, ckallenberg, jbutterworth, scornwell, rheinemann} at mitre dot org

- Twitter contact:

@xenokovah, @coreykal, @jwbutterworth3, @ssc0rnwell

Obligatory "Check out OpenSecurityTraining.info" plug :)

MITRE

References

- **The best place to look: our timeline bibliography:** http://timeglider.com/timeline/5ca2daa6078caaf4
- **[1] Attacking Intel BIOS – Alexander Tereshkin & Rafal Wojtczuk – Jul. 2009** http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf
- **[2] TPM PC Client Specification - Feb. 2013** http://www.trustedcomputinggroup.org/developers/pc_client/specifications/
- **[3] Evil Maid Just Got Angrier: Why Full-Disk Encryption With TPM is Insecure on Many Systems – Yuriy Bulygin – Mar. 2013** http://cansecwest.com/slides/2013/Evil%20Maid%20Just%20Got%20Angrier.pdf
- **[4] A Tale of One Software Bypass of Windows 8 Secure Boot – Yuriy Bulygin – Jul. 2013** http://blackhat.com/us-13/briefings.html#Bulygin
- **[5] Attacking Intel Trusted Execution Technology - Rafal Wojtczuk and Joanna Rutkowska – Feb. 2009** http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf
- **[6] Another Way to Circumvent Intel® Trusted Execution Technology - Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin – Dec. 2009** http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf
- **[7] Exploring new lands on Intel CPUs (SINIT code execution hijacking) - Rafal Wojtczuk and Joanna Rutkowska – Dec. 2011** http://www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf

MITRE

References 2

- [7] Meet 'Rakshasa,' The Malware Infection Designed To Be Undetectable And Incurable - http://www.forbes.com/sites/andygreenberg/2012/07/26/meet-rakshasa-the-malware-infection-designed-to-be-undetectable-and-incurable/
- [8] Implementing and Detecting an ACPI BIOS Rootkit – Heasman, Feb. 2006 http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf
- [9] Implementing and Detecting a PCI Rookit – Heasman, Feb. 2007 http://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf
- [10] Using CPU System Management Mode to Circumvent Operating System Security Functions - Duflot et al., Mar. 2006 http://www.ssi.gouv.fr/archive/fr/sciences/fichiers/lti/cansecwest2006-duflot-paper.pdf
- [11] Getting into the SMRAM:SMM Reloaded – Duflot et. Al, Mar. 2009 http://cansecwest.com/csw09/csw09-duflot.pdf
- [12] Attacking SMM Memory via Intel® CPU Cache Poisoning – Wojtczuk & Rutkowska, Mar. 2009 http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf
- [13] Defeating Signed BIOS Enforcement – Kallenberg et al., Sept. 2013
- http://www.syscan.org/index.php/download/get/6e597f6067493dd581eed737146f3afb/SyScan2014_CoreyKallenberg_SetupforFailureDefeatingSecureBoot.zip

References 3

- [14] Mebromi: The first BIOS rootkit in the wild – Giuliani, Sept. 2011 http://www.webroot.com/blog/2011/09/13/mebromi-the-first-bios-rootkit-in-the-wild/
- [15] Persistent BIOS Infection – Sacco & Ortega, Mar. 2009 http://cansecwest.com/csw09/csw09-sacco-ortega.pdf
- [16] Deactivate the Rootkit – Ortega & Sacco, Jul. 2009 http://www.blackhat.com/presentations/bh-usa-09/ORTEGA/BHUSA09-Ortega-DeactivateRootkit-PAPER.pdf
- [17] Sticky Fingers & KBC Custom Shop – Gazet, Jun. 2011 http://esec-lab.sogeti.com/dotclear/public/publications/11-recon-stickyfingers_slides.pdf
- [18] BIOS Chronomancy: Fixing the Core Root of Trust for Measurement – Butterworth et al., May 2013 http://www.nosuchcon.org/talks/D2_01_Butterworth_BIOS_Chronomancy.pdf
- [19] New Results for Timing-based Attestation – Kovah et al., May 2012 http://www.ieee-security.org/TC/SP2012/papers/4681a239.pdf

MITRE

References 4

- [20] Low Down and Dirty: Anti-forensic Rootkits - Darren Bilby, Oct. 2006 http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Bilby-up.pdf
- [21] Implementation and Implications of a Stealth Hard-Drive Backdoor – Zaddach et al., Dec. 2013 https://www.ibr.cs.tu-bs.de/users/kurmus/papers/acsac13.pdf
- [22] Hard Disk Hacking – Sprite, Jul. 2013 http://spritesmods.com/?art=hddhack
- [23] Embedded Devices Security and Firmware Reverse Engineering - Zaddach & Costin, Jul. 2013 https://media.blackhat.com/us-13/US-13-Zaddach-Workshop-on-Embedded-Devices-Security-and-Firmware-Reverse-Engineering-WP.pdf
- [24] Can You Still Trust Your Network Card – Duflot et al., Mar. 2010 http://www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf
- [25] Project Maux Mk.II, Arrigo Triulzi, Mar. 2008 http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf

MITRE

References 5

- [26] Copernicus: Question your assumptions about BIOS Security – Butterworth, July 2013
  http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/copernicus-question-your-assumptions-about
- [27] Copernicus 2: SENTER the Dragon – Kovah et al., Mar 2014
  https://cansecwest.com/slides/2014/Copernicus2-SENTER_the-Dragon-CSW.pptx
- [28] Playing Hide and Seek with BIOS Implants – Kovah, Mar 2014
  http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/playing-hide-and-seek-with-bios-implants
- [29] Setup for Failure: Defeating UEFI – Kallenberg et al., Apr 2014
  http://syscan.org/index.php/download/get/6e597f6067493dd581eed737146f3afb/SyScan2014_CoreyKallenberg_SetupforFailureDefeatingSecureBoot.zip
- [30] SENTER Sandman: Using Intel TXT to Attack BIOSes – Kovah et al., June 2014
  slides not posted anywhere yet
- [31] Extreme Privilege Escalation on UEFI Windows 8 Systems – Kallenberg et al., Aug 2014 -
  https://www.blackhat.com/docs/us-14/materials/us-14-Kallenberg-Extreme-Privilege-Escalation-On-Windows8-UEFI-Systems.pdf

MITRE