Analyzing UEFI BIOS from Attacker & Defender Viewpoints

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

Introduction

- **Who we are:**
  – Trusted Computing and firmware security researchers at The MITRE Corporation

- **What MITRE is:**
  – A not-for-profit company that runs seven US Government "Federally Funded Research & Development Centers" (FFRDCs) dedicated to working in the public interest
  – Technical lead for a number of standards and structured data exchange formats such as *CVE*, CWE, OVAL, CAPEC, STIX, TAXII, etc
  – The first .org, !(.mil | .gov | .com | .edu | .net), on the ARPANET

Introduction 2

- In the talks we've been giving for the last year, we've repeatedly referred to the new UEFI (Unified Extensible Firmware Interface) as a double-edged sword. I.e. there are things about it that help attackers, and things that help defenders.
- This talk is a more thorough examination of that assertion
- It's also to give you more context for if a tool like MITRE Copernicus or Intel Chipsec tells you there is a problem with your BIOS
- To be clear, we *do not* work in the area of attacking BIOSes

MITRE

BIOS is dead, long live UEFI!

- **Not quite**
- **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)**

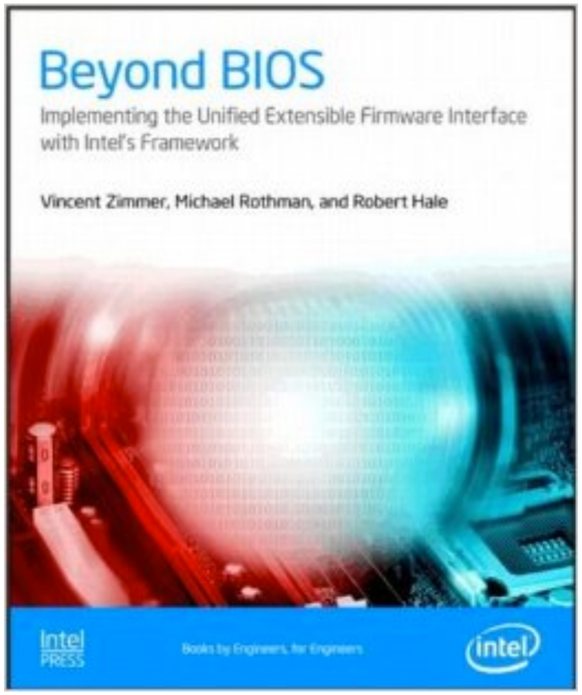
About UEFI

- **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!)

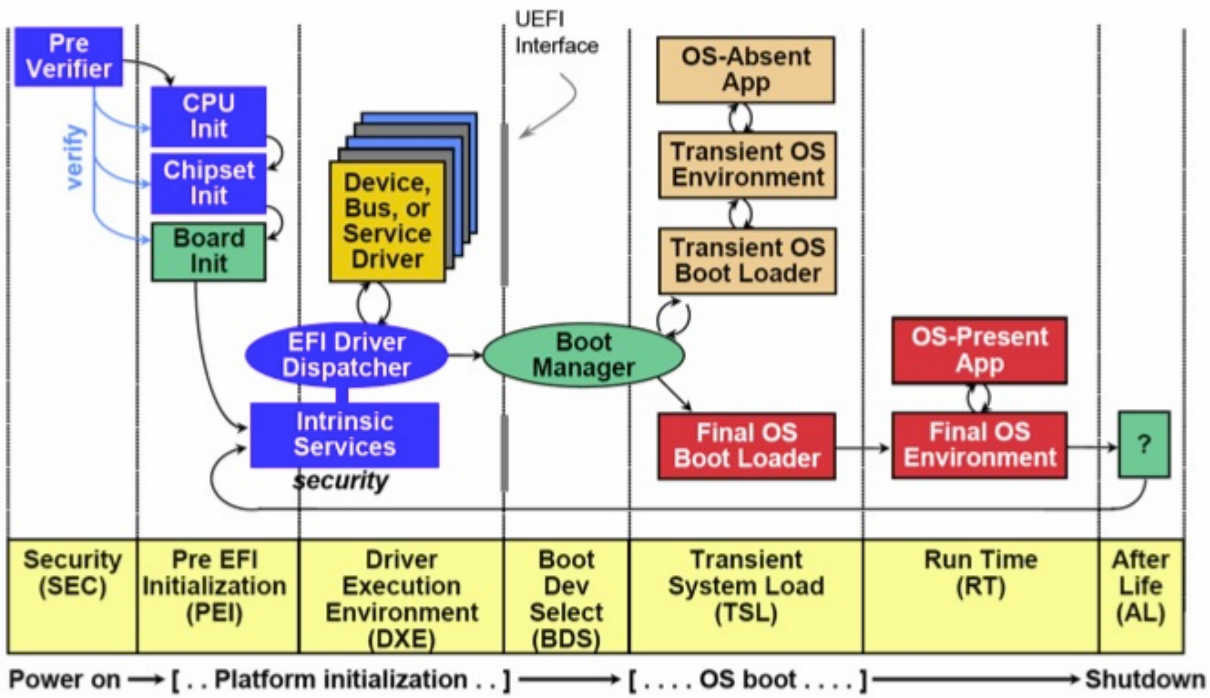
Something you may want to read

- **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**



UEFI Differences: Boot Phases

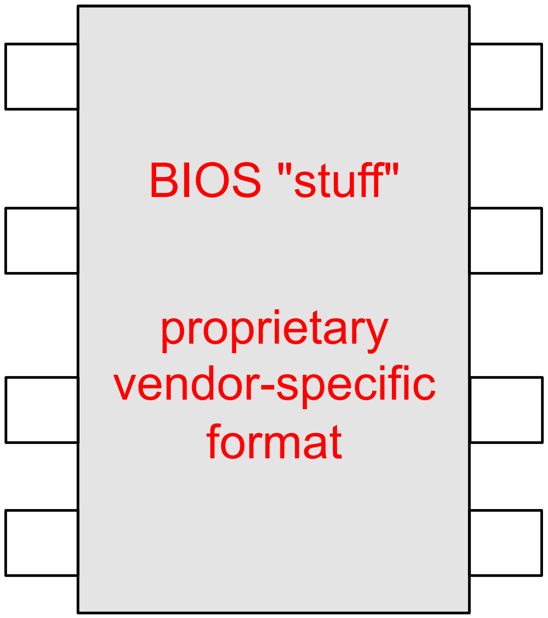


- 7 Phases total
- Phases are defined in the UEFI specification

Bottom up

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

Legacy BIOS Firmware Storage



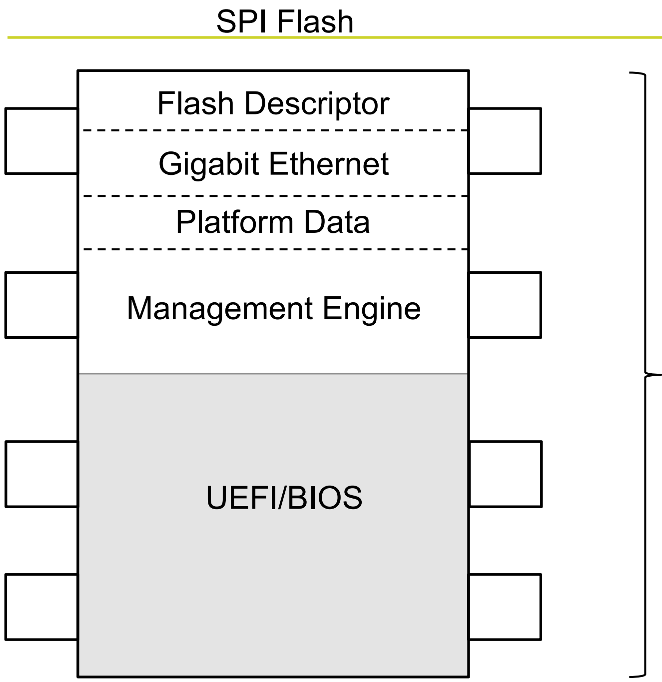
SPI Flash

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**
- **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

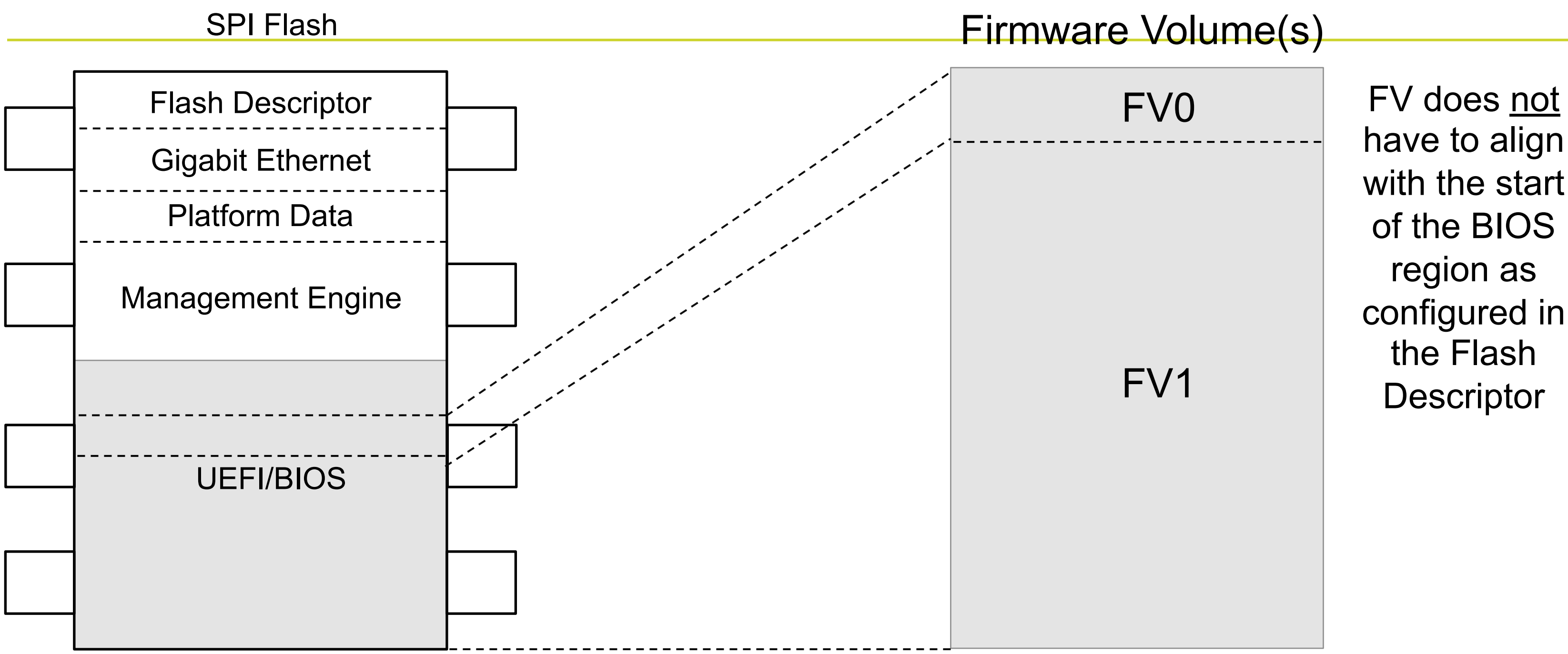


Firmware Device refers to the flash chip

- **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**

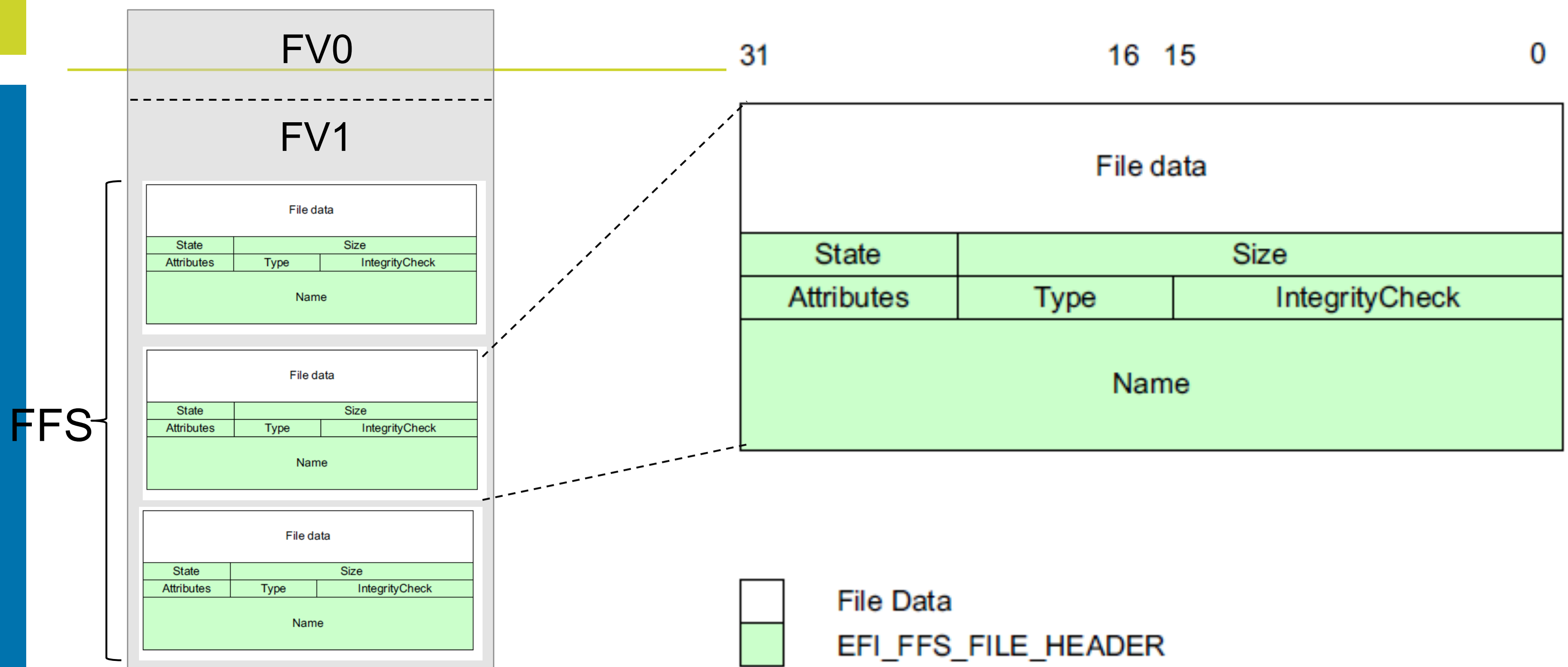
MITRE

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

Firmware Files

31	16 15	0
File data		
State	Size	
Attributes	Type	IntegrityCheck
Name		

```
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;
```

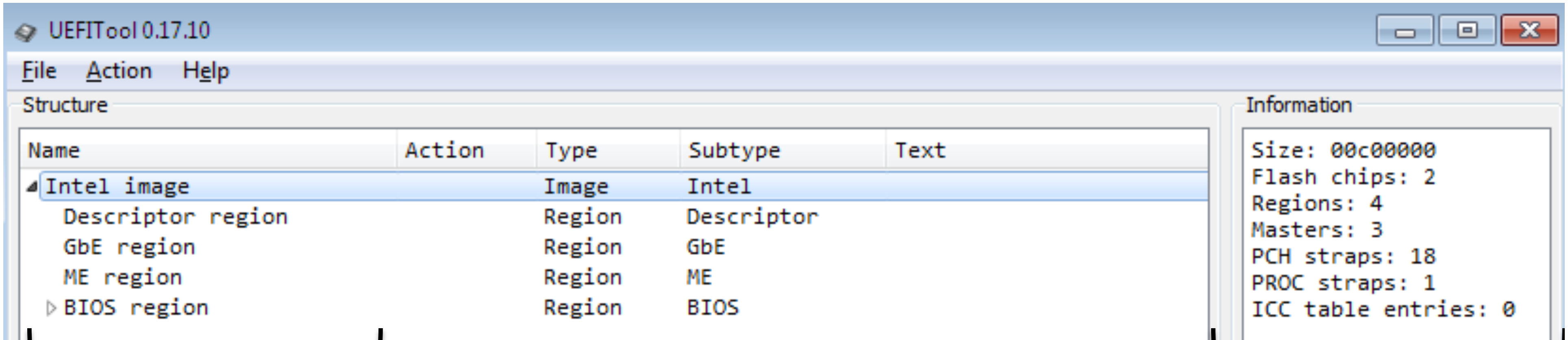
File Data
EFI_FFS_FILE_HEADER

- **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!

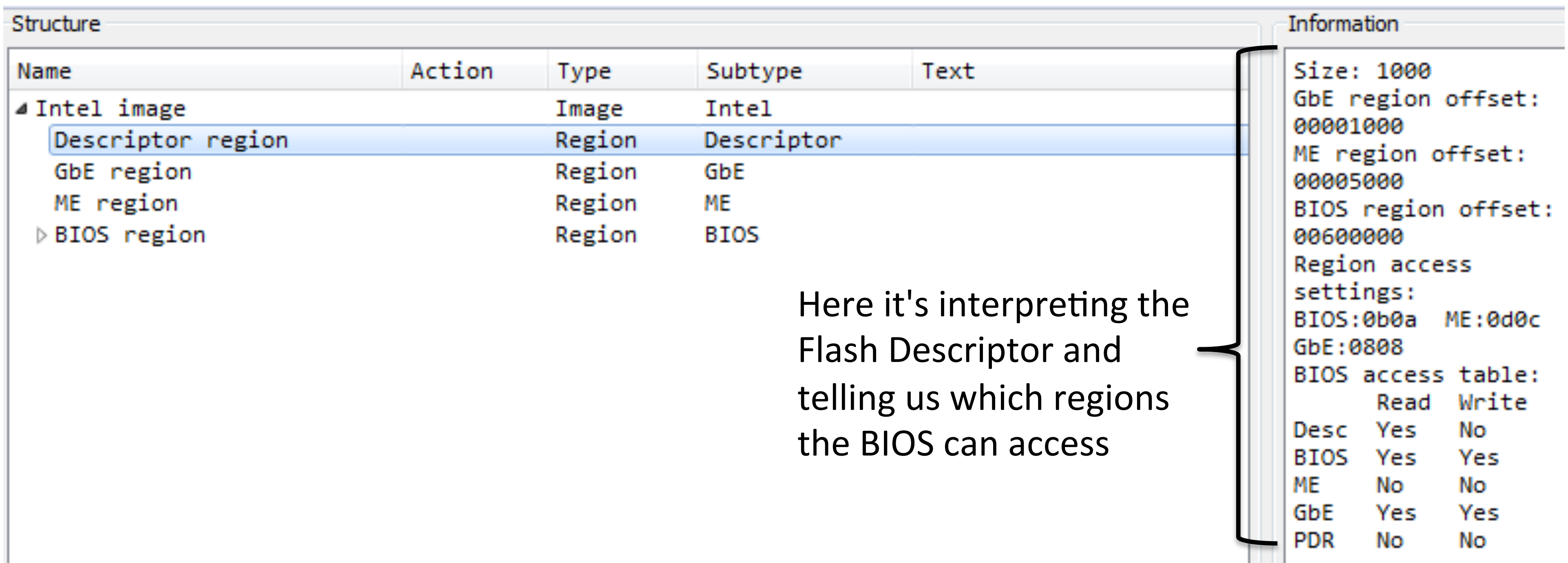
Options for Parsing FFS

- **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



Navigation by expanding portions here

Parsed metadata here



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

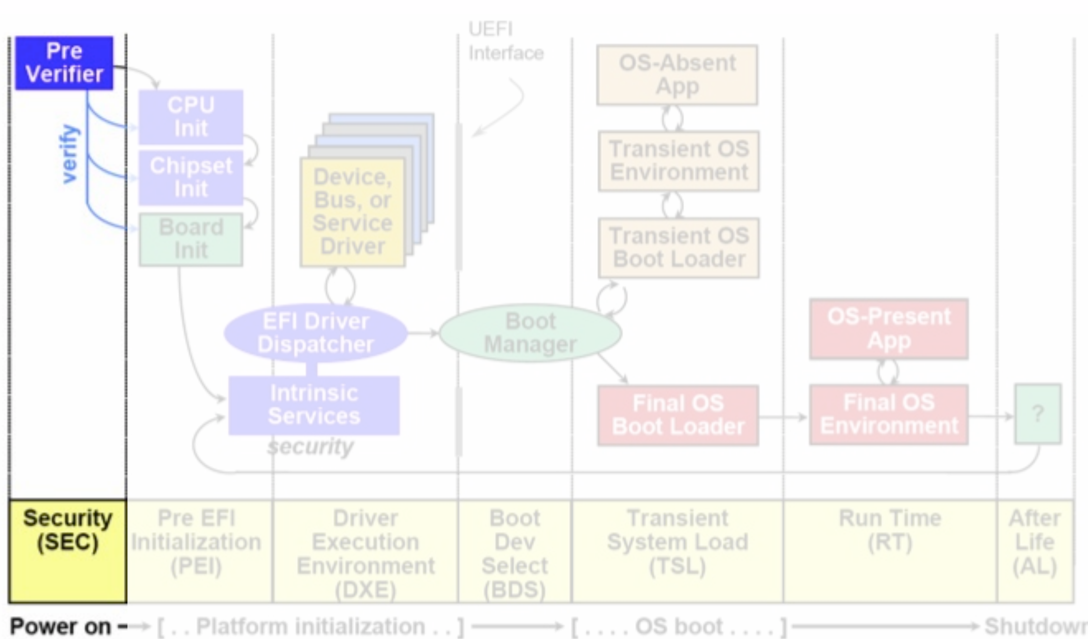
We'll come back to this later after we learn more

Yay Standardization!

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

Security (SEC) Phase



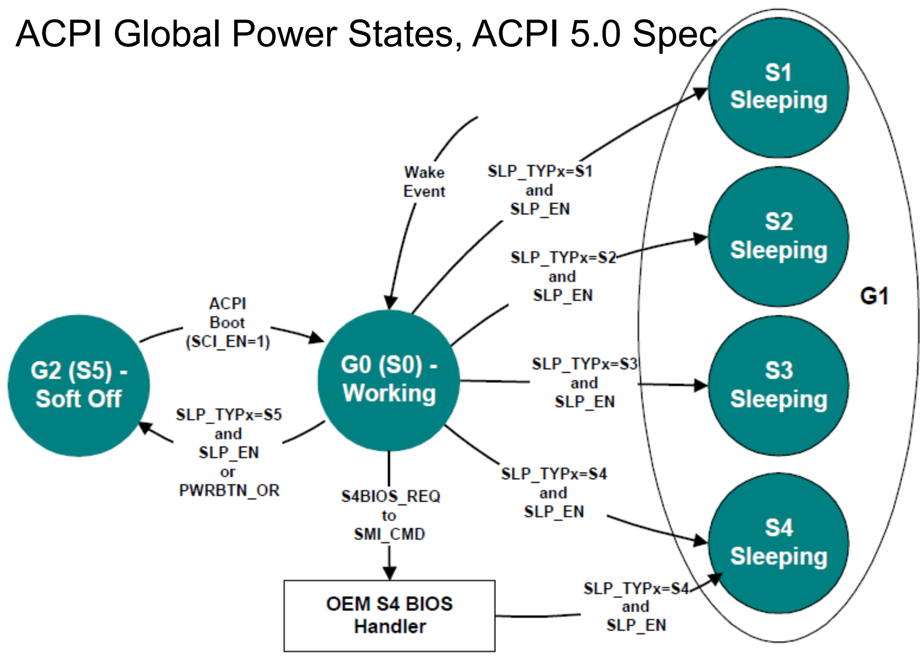
- **The SEC phase is the first phase in the PI architecture**
- **Contains the first code that is executed by the CPU**
- **Environment is basically that of legacy:**
  - Small/minimal code typically hand-coded assembly so architecturally dependent and not portable
  - Executes directly from flash
 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

ACPI Global Power States, ACPI 5.0 Spec



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

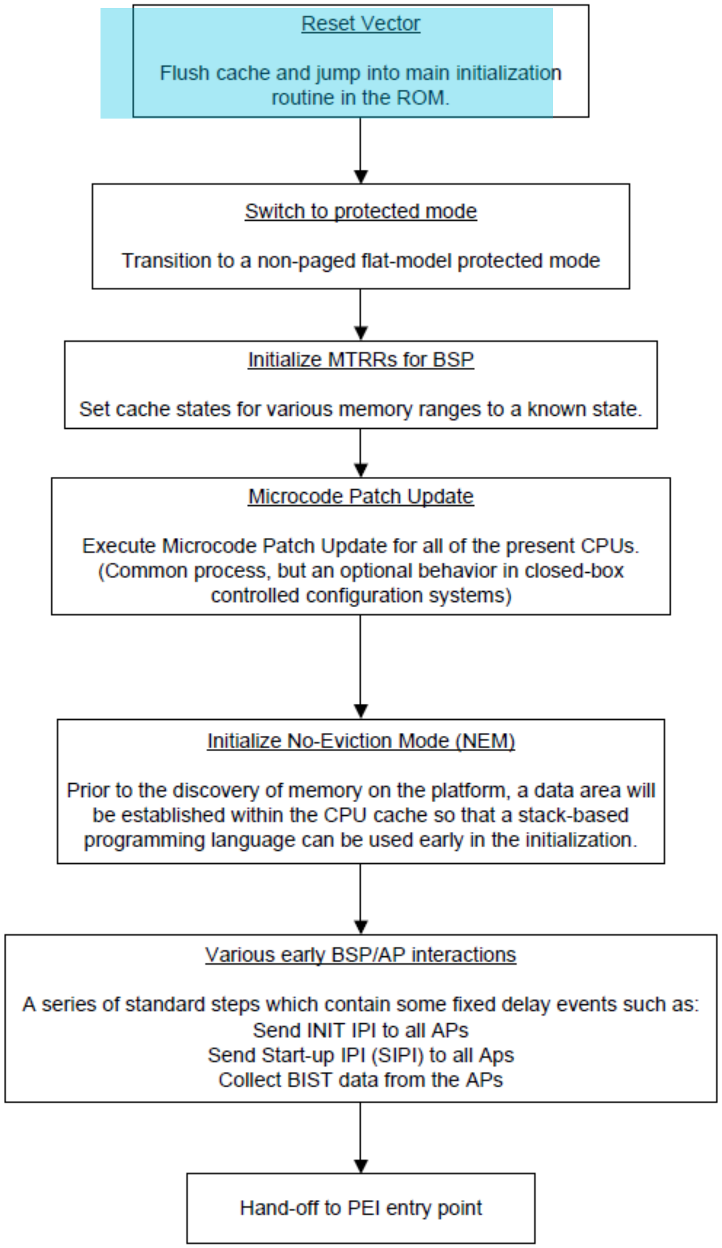
- This isn't a discussion of ACPI (big topic[1]), but it's important to note that alternate boot paths as determined by sleep mode could make the BIOS vulnerable
- A system that awakes from Sleep mode will follow a different path to boot
- This different code path may not lock down the system the same 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.

[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



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…**

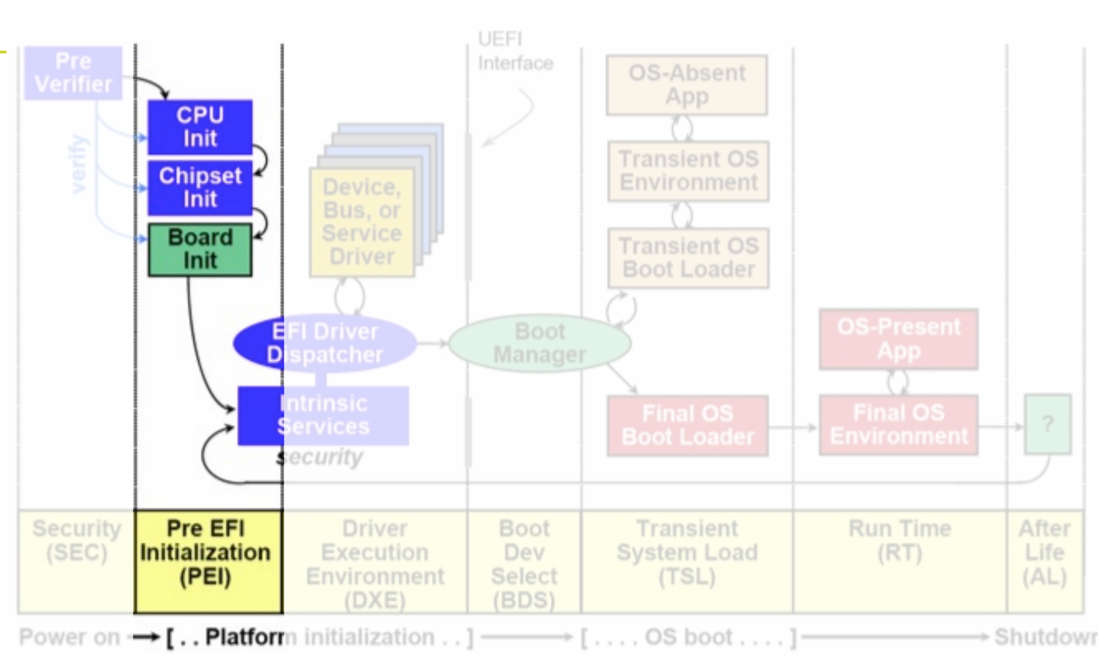
SEC Hand-off to PEI Entry Point

```
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.

Platform Initialization Spec Vol. 1, Version 1.3, Section 2.4

Components of PEI

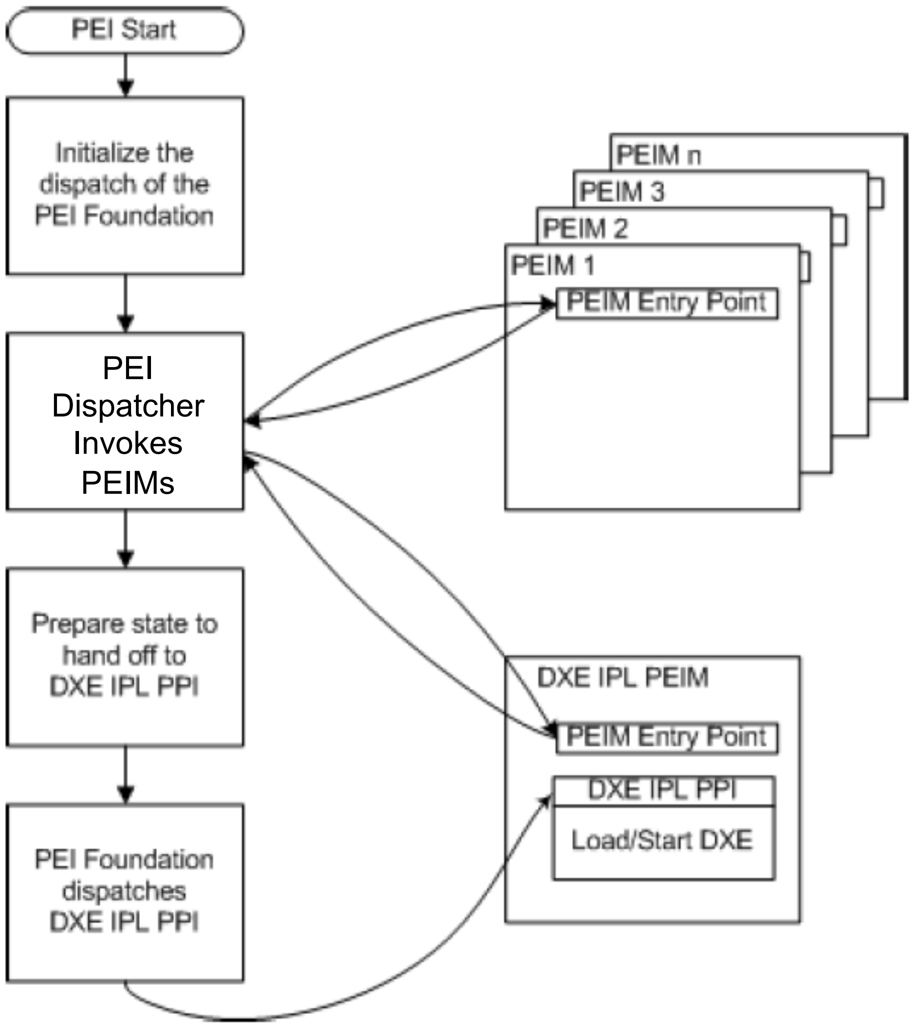
- **PEI Dispatcher**
  - Evaluates the dependency expressions in PEIMs and, if they are met, installs them (and executes them)
- **Dependency Expression(DEPEX)**
  - Basically GUIDs of PPIs that must have already been dispatched before a PEIM is permitted to load/execute
- **Firmware Volumes**
  - Storage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

As the tables turn… PEI Services Table

```
typedef struct _EFI_PEI_SERVICES {
  EFI_TABLE_HEADER                Hdr;
  EFI_PEI_INSTALL_PPI             InstallPpi;
  EFI_PEI_REINSTALL_PPI           ReInstallPpi;
  EFI_PEI_LOCATE_PPI              LocatePpi;
  EFI_PEI_NOTIFY_PPI              NotifyPpi;
  EFI_PEI_GET_BOOT_MODE           GetBootMode;
  EFI_PEI_SET_BOOT_MODE           SetBootMode;
  EFI_PEI_GET_HOB_LIST            GetHobList;
  EFI_PEI_CREATE_HOB              CreateHob;
  EFI_PEI_FFS_FIND_NEXT_VOLUME    FfsFindNextVolume;
  EFI_PEI_FFS_FIND_NEXT_FILE      FfsFindNextFile;
  EFI_PEI_FFS_FIND_SECTION_DATA   FfsFindSectionData;
  EFI_PEI_INSTALL_PEI_MEMORY      InstallPeiMemory;
  EFI_PEI_ALLOCATE_PAGES          AllocatePages;
  EFI_PEI_ALLOCATE_POOL           AllocatePool;
  EFI_PEI_COPY_MEM                CopyMem;
  EFI_PEI_SET_MEM                 SetMem;
  EFI_PEI_REPORT_STATUS_CODE      ReportStatusCode;
  EFI_PEI_RESET_SYSTEM            ResetSystem;
  EFI_PEI_CPU_IO_PPI              CpuIo;
  EFI_PEI_PCI_CFG_PPI             PciCfg;
} EFI_PEI_SERVICES;
```

Phoenix Wiki has good descriptions of what they all do:
http://wiki.phoenix.com/wiki/index.php/EFI_PEI_SERVICES

PEI Phase

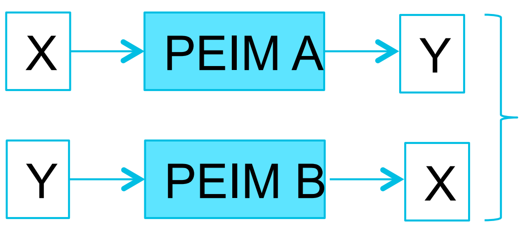


- **This is a basic diagram of the PEI operations performed by the PEI Foundation**
- **The PEI foundation builds the PEI Services table**
- **The core of it centers around the PEI Dispatcher which locates and executes PEIMs**
  - Initializing permanent memory, etc.
- **The last PEIM to be dispatched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

Platform Initialization Spec Vol. 1, Version 1.3, Section 2.4

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

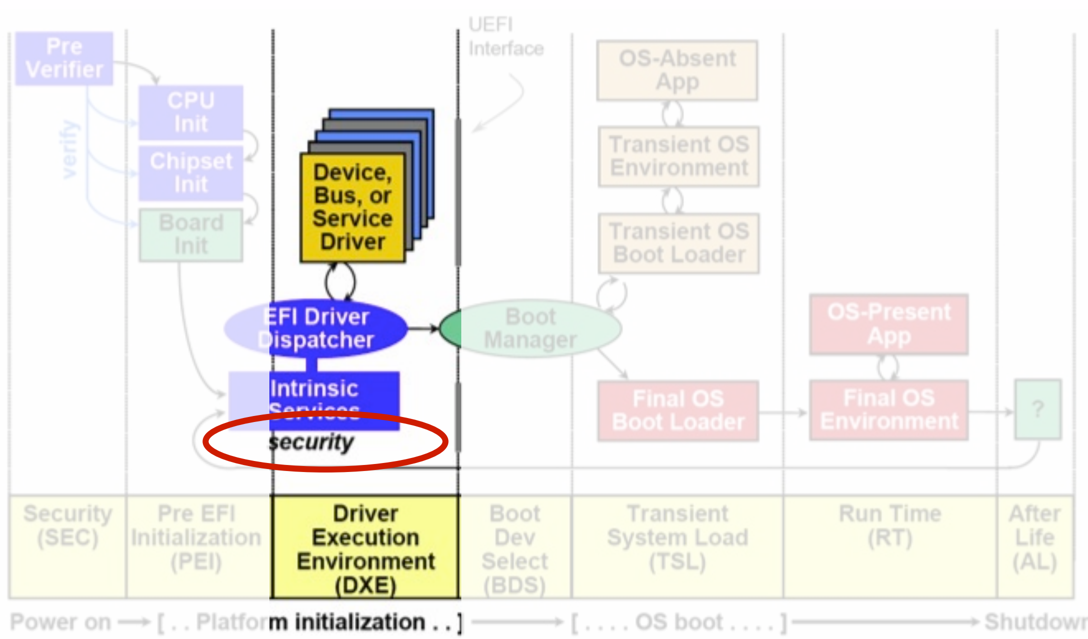
- **One PPI is EFI_FIND_FV_PPI so every PEIM on every Firmware Volume can be invoked**
- **Once all PEIMs that can execute have been, the last PEIM executed is the DXE IPL PEIM which hands off to DXE phase**

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)

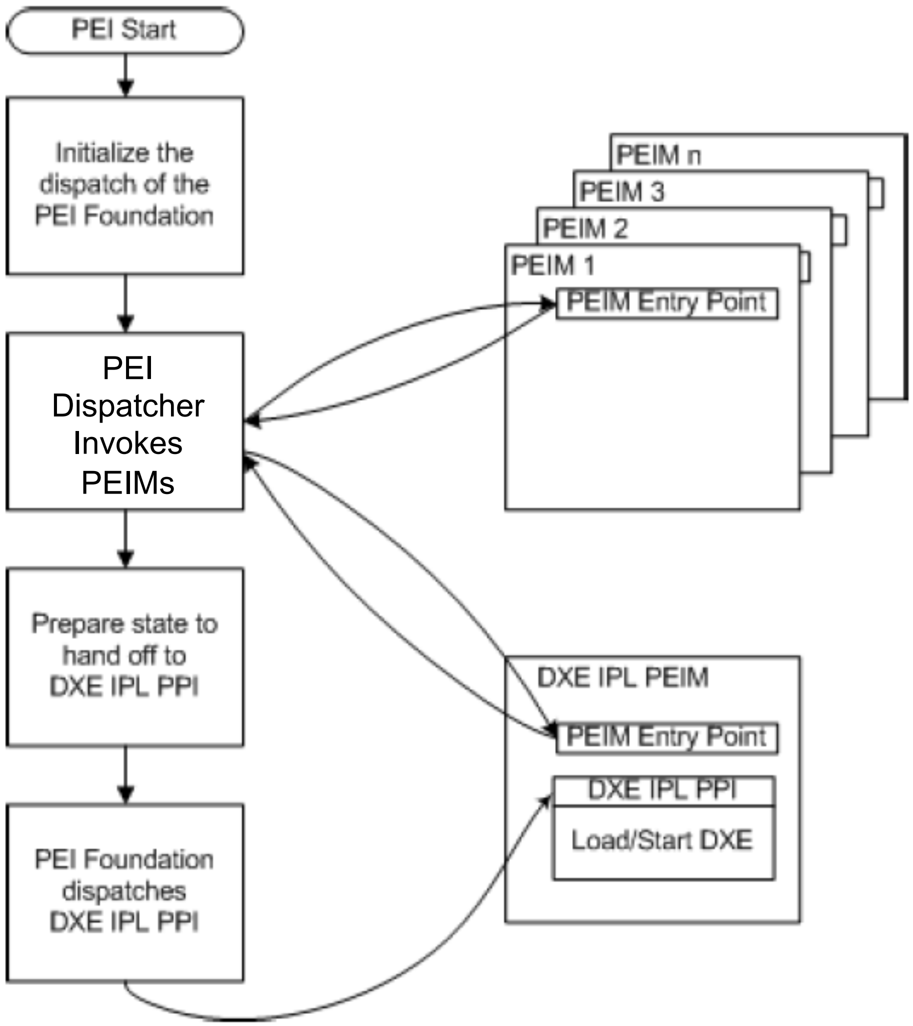


- The DXE phase is designed to be executed at a high-enough level where it is independent from architectural requirements
- Similar to PEI from a high-level PoV: creates services used only by DXE, has a dispatcher that finds and loads DXE drivers, etc.
- System Management Mode set up, Secure Boot enforcement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

PEI is to DXE as…

- **PEIMs are to DXE Drivers**
- **PEI Dispatcher is to DXE Dispatcher**
  – DXE uses an almost identical system as PEI to load and invoke individual units of functionality, as required by the DEPEXs
- **PPI is to Protocol**
  – DXE drivers register and lookup "protocols"
- **Sec Core Data are to HOBs**
  – PEI gets Sec Core Data from SEC, DXE gets HOBs from PEI

DXE Phase



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

Platform Initialization Spec Vol. 1, Version 1.3, Section 2.4

As the tables turn… DXE Services Table

```
typedef struct {
  EFI_TABLE_HEADER                  Hdr;
  EFI_ADD_MEMORY_SPACE              AddMemorySpace;
  EFI_ALLOCATE_MEMORY_SPACE         AllocateMemorySpace;
  EFI_FREE_MEMORY_SPACE             FreeMemorySpace;
  EFI_REMOVE_MEMORY_SPACE           RemoveMemorySpace;
  EFI_GET_MEMORY_SPACE_DESCRIPTOR   GetMemorySpaceDescriptor;
  EFI_SET_MEMORY_SPACE_ATTRIBUTES   SetMemorySpaceAttributes;
  EFI_GET_MEMORY_SPACE_MAP          GetMemorySpaceMap;
  EFI_ADD_IO_SPACE                  AddIoSpace;
  EFI_ALLOCATE_IO_SPACE             AllocateIoSpace;
  EFI_FREE_IO_SPACE                 FreeIoSpace;
  EFI_REMOVE_IO_SPACE               RemoveIoSpace;
  EFI_GET_IO_SPACE_DESCRIPTOR       GetIoSpaceDescriptor;
  EFI_GET_IO_SPACE_MAP              GetIoSpaceMap;
  EFI_DISPATCH                      Dispatch;
  EFI_SCHEDULE                      Schedule;
  EFI_TRUST                         Trust;
  EFI_PROCESS_FIRMWARE_VOLUME       ProcessFirmwareVolume;
} EFI_DXE_SERVICES;
```

Phoenix Wiki has good descriptions of what they all do:
http://wiki.phoenix.com/wiki/index.php/EFI_DXE_SERVICES

As the tables turn… Boot Services Table 1

```
typedef struct {
  EFI_TABLE_HEADER                       Hdr;
  EFI_RAISE_TPL                          RaiseTPL;
  EFI_RESTORE_TPL                        RestoreTPL;
  EFI_ALLOCATE_PAGES                     AllocatePages;
  EFI_FREE_PAGES                         FreePages;
  EFI_GET_MEMORY_MAP                     GetMemoryMap;
  EFI_ALLOCATE_POOL                      AllocatePool;
  EFI_FREE_POOL                          FreePool;
  EFI_CREATE_EVENT                       CreateEvent;
  EFI_SET_TIMER                          SetTimer;
  EFI_WAIT_FOR_EVENT                     WaitForEvent;
  EFI_SIGNAL_EVENT                       SignalEvent;
  EFI_CLOSE_EVENT                        CloseEvent;
  EFI_CHECK_EVENT                        CheckEvent;
  EFI_INSTALL_PROTOCOL_INTERFACE         InstallProtocolInterface;
  EFI_REINSTALL_PROTOCOL_INTERFACE       ReinstallProtocolInterface;
  EFI_UNINSTALL_PROTOCOL_INTERFACE       UninstallProtocolInterface;
  EFI_HANDLE_PROTOCOL                    HandleProtocol;
  VOID*                                  Reserved;
```

Phoenix Wiki has good descriptions of what they all do:
http://wiki.phoenix.com/wiki/index.php/EFI_BOOT_SERVICES

As the tables turn… Boot Services Table 2

```
EFI_REGISTER_PROTOCOL_NOTIFY        RegisterProtocolNotify;
EFI_LOCATE_HANDLE                   LocateHandle;
EFI_LOCATE_DEVICE_PATH              LocateDevicePath;
EFI_INSTALL_CONFIGURATION_TABLE     InstallConfigurationTable;
EFI_IMAGE_LOAD                      LoadImage;
EFI_IMAGE_START                     StartImage;
EFI_EXIT                            Exit;
EFI_IMAGE_UNLOAD                    UnloadImage;
EFI_EXIT_BOOT_SERVICES              ExitBootServices;
EFI_GET_NEXT_MONOTONIC_COUNT        GetNextMonotonicCount;
EFI_STALL                           Stall;
EFI_SET_WATCHDOG_TIMER              SetWatchdogTimer;
EFI_CONNECT_CONTROLLER              ConnectController;
EFI_DISCONNECT_CONTROLLER           DisconnectController;
EFI_OPEN_PROTOCOL                   OpenProtocol;
EFI_CLOSE_PROTOCOL                  CloseProtocol;
```

Phoenix Wiki has good descriptions of what they all do:
http://wiki.phoenix.com/wiki/index.php/EFI_BOOT_SERVICES

MITRE

As the tables turn… Boot Services Table 3

```
  EFI_OPEN_PROTOCOL_INFORMATION             OpenProtocolInformation;
  EFI_PROTOCOLS_PER_HANDLE                  ProtocolsPerHandle;
  EFI_LOCATE_HANDLE_BUFFER                  LocateHandleBuffer;
  EFI_LOCATE_PROTOCOL                       LocateProtocol;
  EFI_INSTALL_MULTIPLE_PROTOCOL_INTERFACES  InstallMultipleProtocolInterfaces;
  EFI_UNINSTALL_MULTIPLE_PROTOCOL_INTERFACES UninstallMultipleProtocolInterfaces;
  EFI_CALCULATE_CRC32                       CalculateCrc32;
  EFI_COPY_MEM                              CopyMem;
  EFI_SET_MEM                               SetMem;
  EFI_CREATE_EVENT_EX                       CreateEventEx;
} EFI_BOOT_SERVICES;
```

Phoenix Wiki has good descriptions of what they all do:
http://wiki.phoenix.com/wiki/index.php/EFI_BOOT_SERVICES

As the tables turn… Runtime Services Table

```
typedef struct {
  EFI_TABLE_HEADER                 Hdr;
  EFI_GET_TIME                     GetTime;
  EFI_SET_TIME                     SetTime;
  EFI_GET_WAKEUP_TIME              GetWakeupTime;
  EFI_SET_WAKEUP_TIME              SetWakeupTime;
  EFI_SET_VIRTUAL_ADDRESS_MAP      SetVirtualAddressMap;
  EFI_CONVERT_POINTER              ConvertPointer;
  EFI_GET_VARIABLE                 GetVariable;
  EFI_GET_NEXT_VARIABLE_NAME       GetNextVariableName;
  EFI_SET_VARIABLE                 SetVariable;
  EFI_GET_NEXT_HIGH_MONO_COUNT     GetNextHighMonotonicCount;
  EFI_RESET_SYSTEM                 ResetSystem;
  EFI_UPDATE_CAPSULE               UpdateCapsule;
  EFI_QUERY_CAPSULE_CAPABILITIES   QueryCapsuleCapabilities;
  EFI_QUERY_VARIABLE_INFO          QueryVariableInfo;
} EFI_RUNTIME_SERVICES;
```

Used for our ring 3 BIOS exploit - BH USA 2014, by Kallenberg et al. [31] CERT VU # 552286

SetVariable also used for CERT VU #758382. Co-discovered with Intel, and first described at CSW 2014

Phoenix Wiki has good descriptions of what they all do:
http://wiki.phoenix.com/wiki/index.php/EFI_RUNTIME_SERVICES

Relative magnitude of PEIMs vs. DXE drivers

Machine release dates are not definitive, just based on first page of Google previews

- **(3/2011) Lenovo X220: 65 PEIMs, 278 DXE drivers**
- **(1/2014) Lenovo X240: 80 PEIMs, 352 DXE drivers**
- **(3/2010) HP Elitebook 2540p: 42 PEIMs, 164 DXE drivers**
- **(1/2014) HP Elitebook 850 G1: 117 PEIMs, 392 DXE drivers**
- **(11/2010) Dell Latitude E6410: 32 PEIMs, 315 DXE drivers**
- **(2/2014) Dell Latitude E6440: 63 PEIMs, 456 DXE drivers**
- **DXE has got it going on!**
- **Increase in code & complexity over time? Sounds like we're on the highway to hell, not a stairway to heaven…**

MITRE

UEFI Non-Volatile Variables

- The (much more extensible and (eventually) secure) replacement for "CMOS" / "NVRAM" as a BIOS configuration mechanism
- Stored on the SPI flash chip along with the rest of the BIOS code
- Growing pains: there've been at least two examples (Samsung & Lenovo) of systems that were implemented incorrectly and once the variable space was filled up (e.g. accidentally by an OS logging mechanism), the system was bricked
- Are be accessed in PEI (the CapsuleUpdate variable of VU#552286 fame certainly was), but overall, variables are more likely to be accessed in DXE and later phases (up to and including runtime)

Samsung - http://mjg59.dreamwidth.org/22028.html
Lenovo - https://bugzilla.redhat.com/show_bug.cgi?id=919485

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
```

- **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

- **‘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
- **‘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

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 <u>not</u> as Authenticated, the variable can be modified by any application**
  – The Setup variable (of VU#758382 fame) is marked like this

"Authenticate how?" Keys and Key Stores

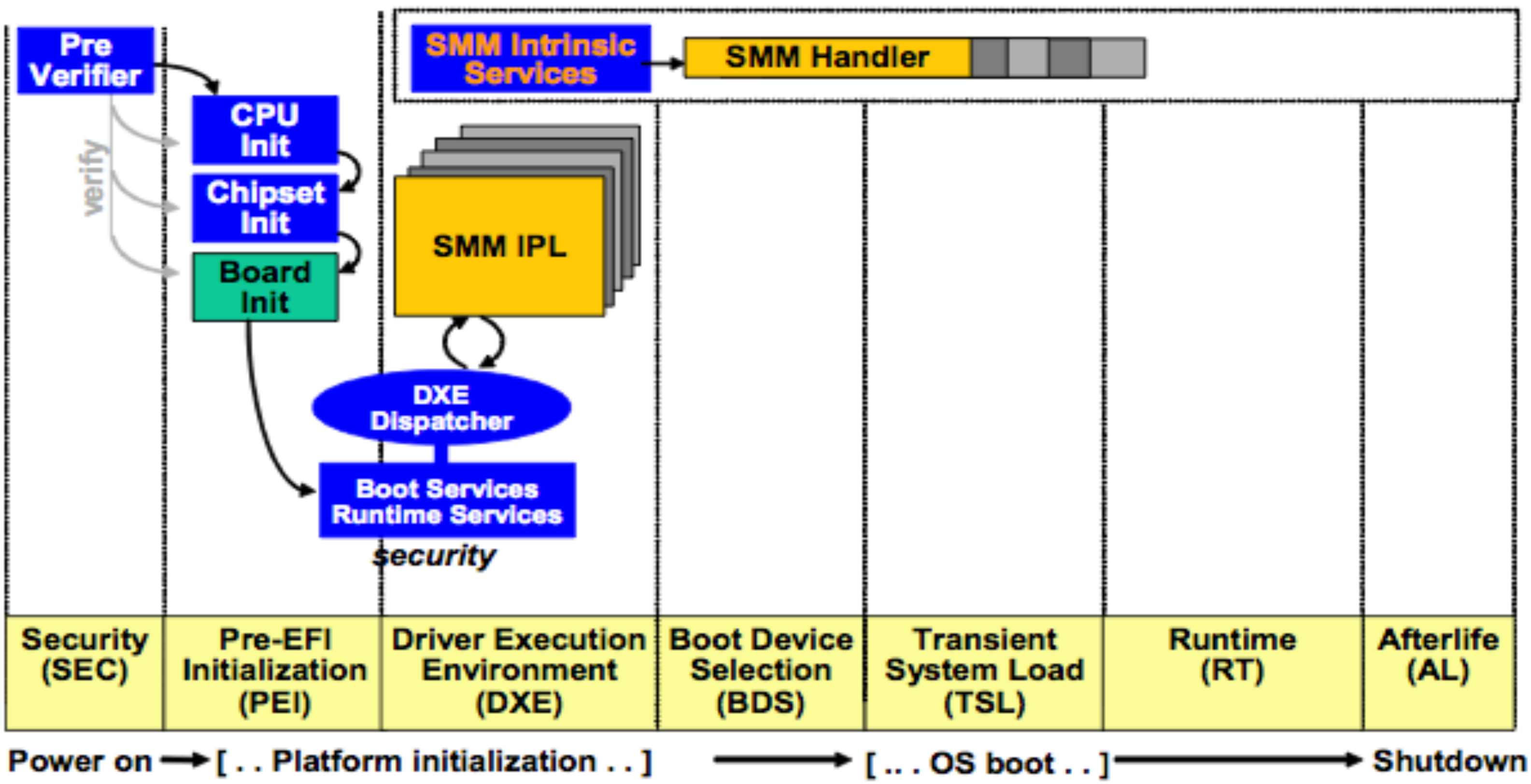
- **UEFI implements 4 variables which store keys, signatures, and/or hashes:**
- **Platform Key (PK)**
  - "The platform key establishes a trust relationship between the platform owner and the platform firmware." - spec
  - Controls access to itself and the KEK variables
  - Only a physically present 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." - spec
  - 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) 2

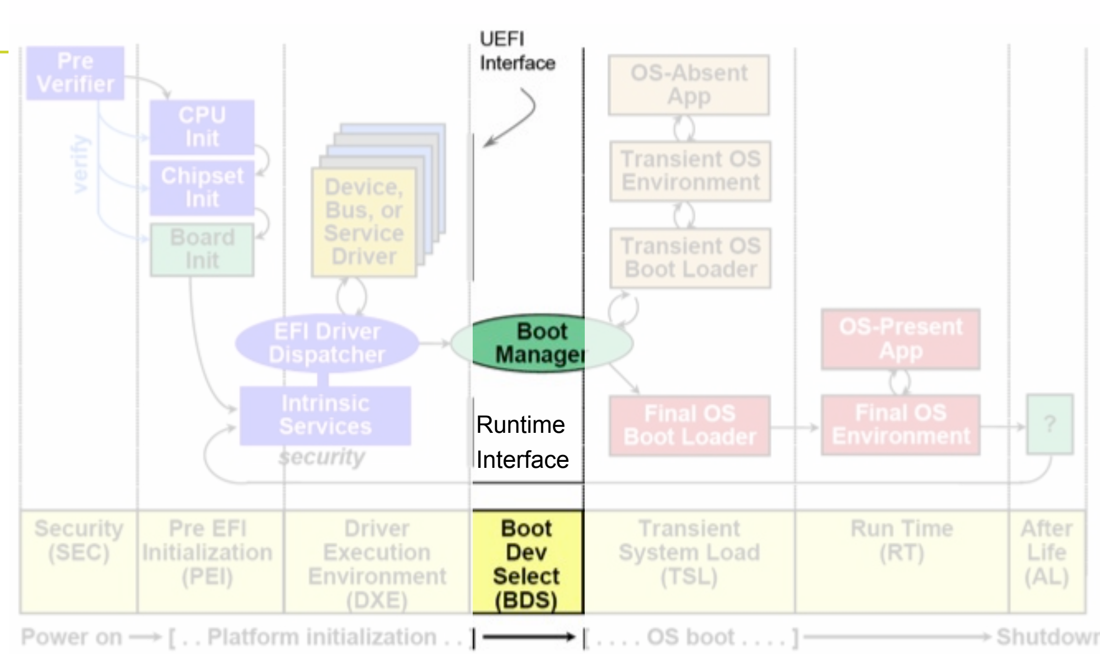
- As stated earlier, these variables are stored on the Flash file system
- Thus, if the SPI flash isn't locked down properly, these keys/hashes can be overwritten by an attacker
- The problem is, the UEFI variables must rely solely on SMM to protect them!
- The secondary line of defense, the Protected 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
- See our "Setup for Failure" [29] talk to see an example of SMI suppression to write to the DB to whitelist the "Charizard" PoC bootkit (also check out the video ;) [33])

DXE & SMM, BFF 4EVA!



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

Boot Device Selection (BDS)

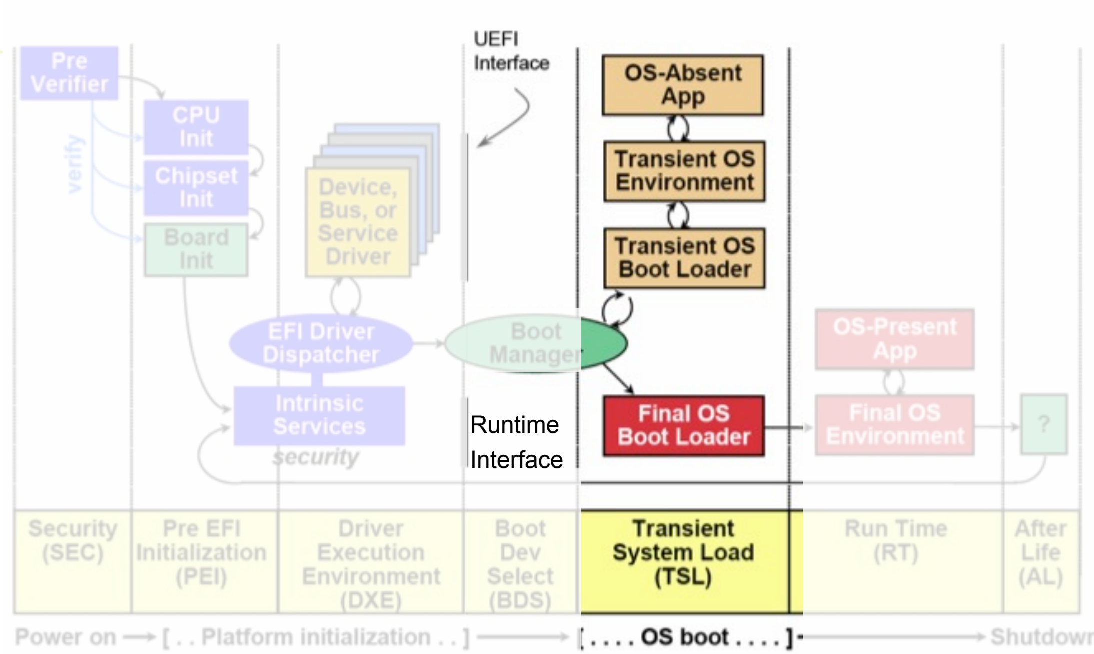


- **The BDS will typically be encapsulated into a single file loaded by the DXE phase.**
- **It consults the configuration information to decide whether you're going to boot an OS or "something else"**
- **It has access to the full UEFI Boot Services Table of services that DXE set up. E.g. HD filesystem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

I give unto thee: an interface!

- Unlike the transition from SEC -> PEI or PEI -> DXE, there's no collecting of information to give to BDS
- Instead what's given is a pointer to the system table, which in turn points to the boot services and DXE services tables, for the BDS (and next) phase(s) to use as need be.

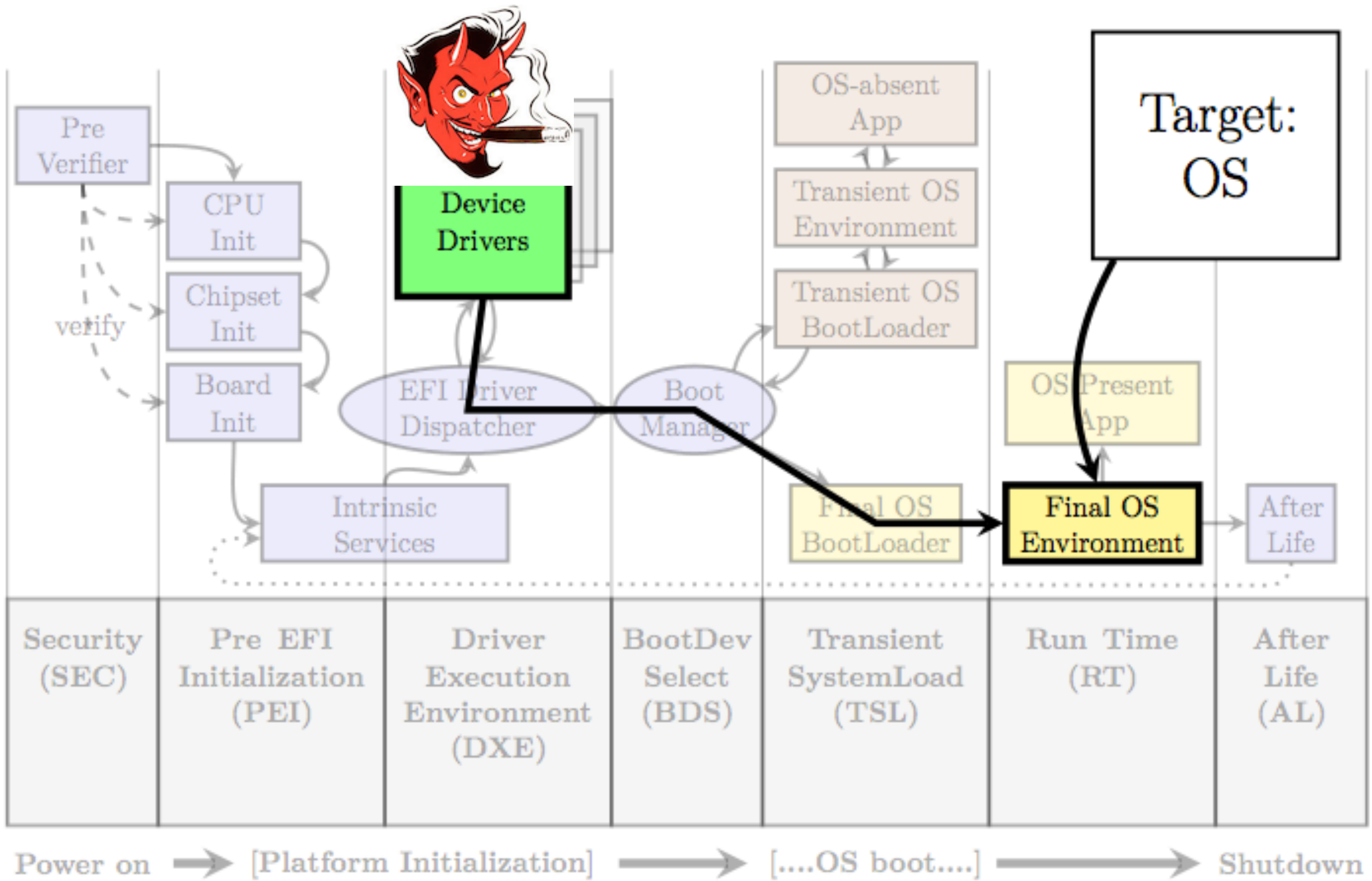
Transient System Load (TSL)



- **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 *denies anything un-signed* (e.g. super 1337 "Oooh look at me, I made the first UEFI bootkit!!1" bootkits ;))**

Pierre Chifflier, UEFI and PCI Bootkits, PacSec 2013 [34]

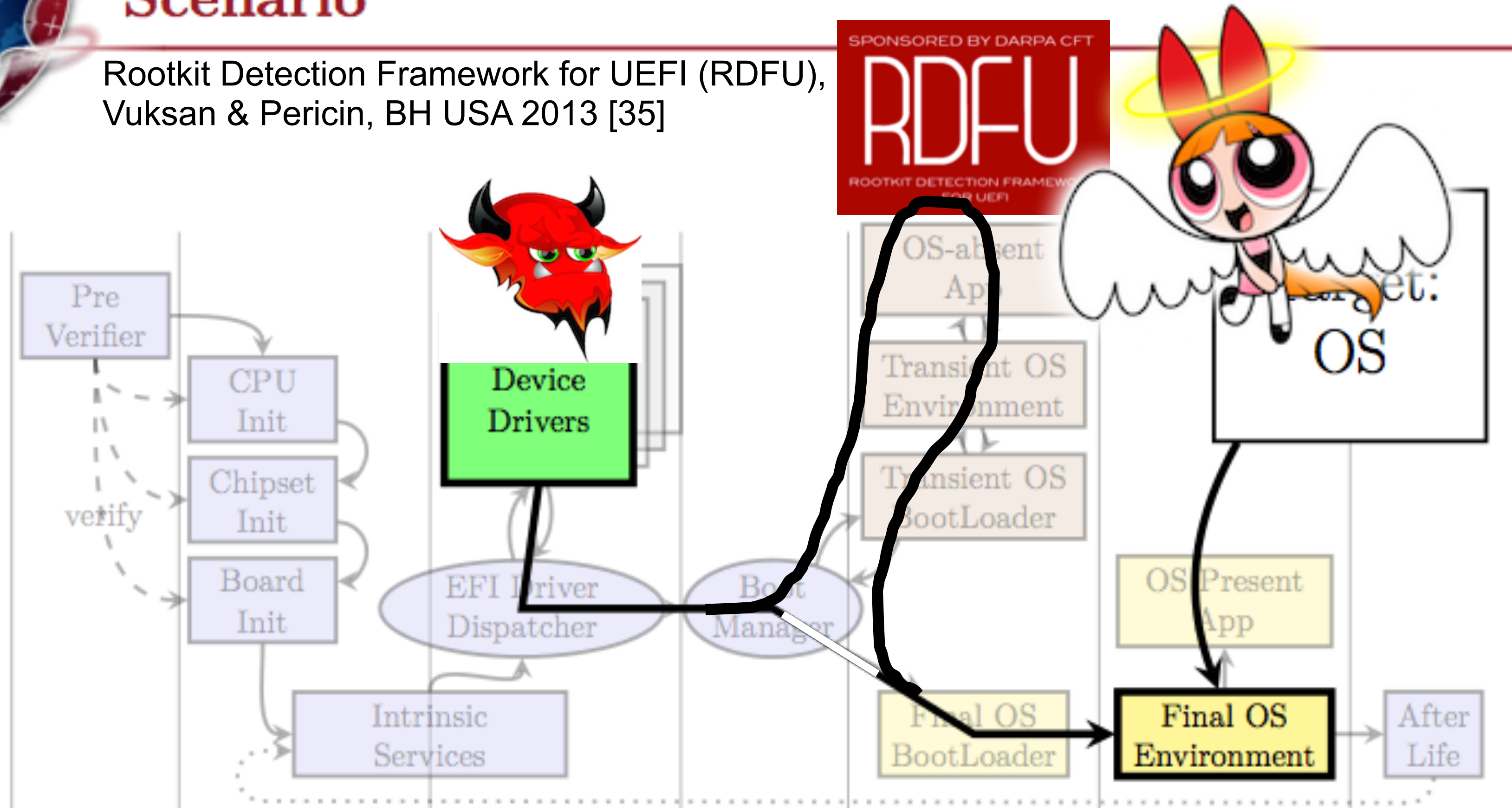
Scenario



Pierre Chifflier, UEFI and PCI Bootkits, PacSec 2013 [34]

Scenario

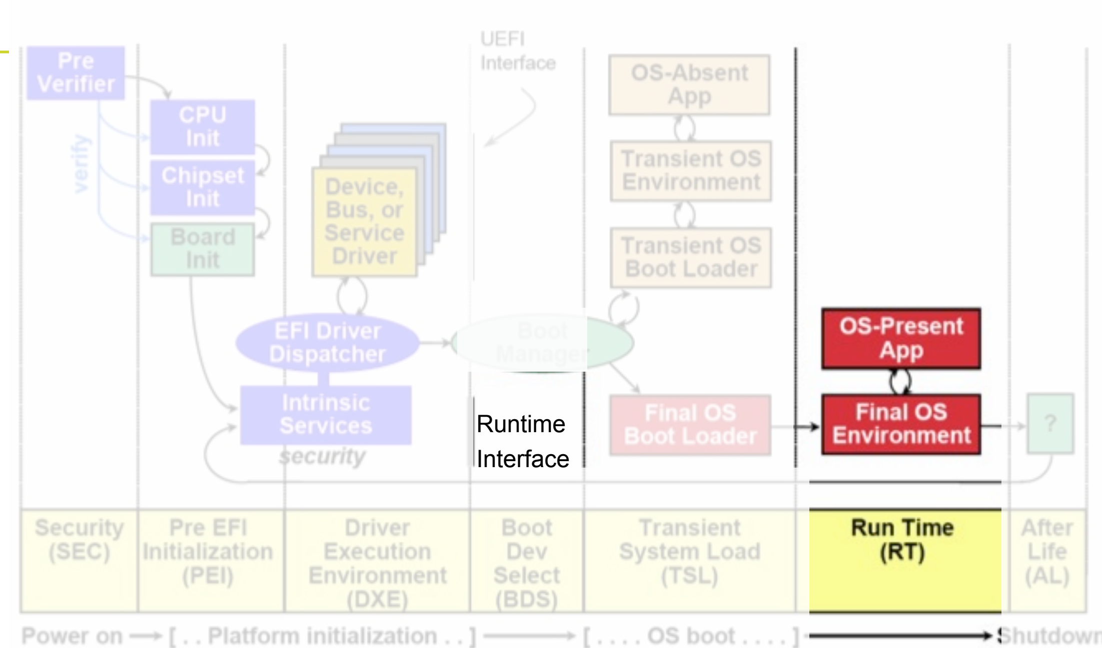
Rootkit Detection Framework for UEFI (RDFU), Vuksan & Pericin, BH USA 2013 [35]



What if we *forced* boot to go through a randomized OS absent security application (that ideally uses the TPM/TXT to ensure its trustworthiness?)

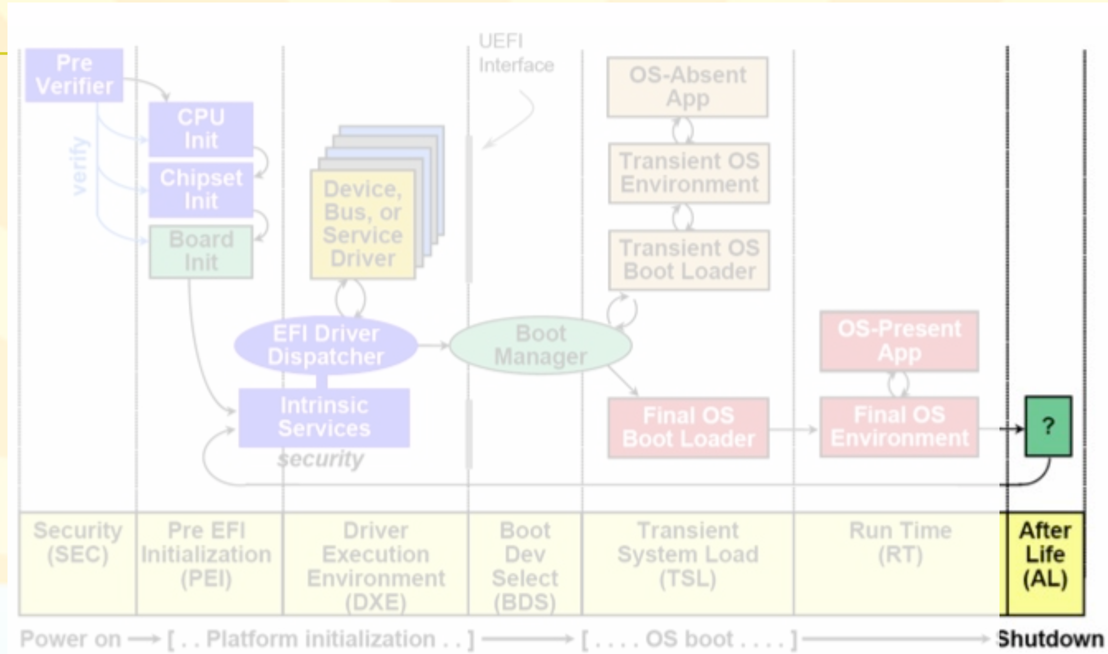
Power on → [Platform Initialization] → [....OS boot....] → Shutdown

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 talked about a while ago

After Life (AL)

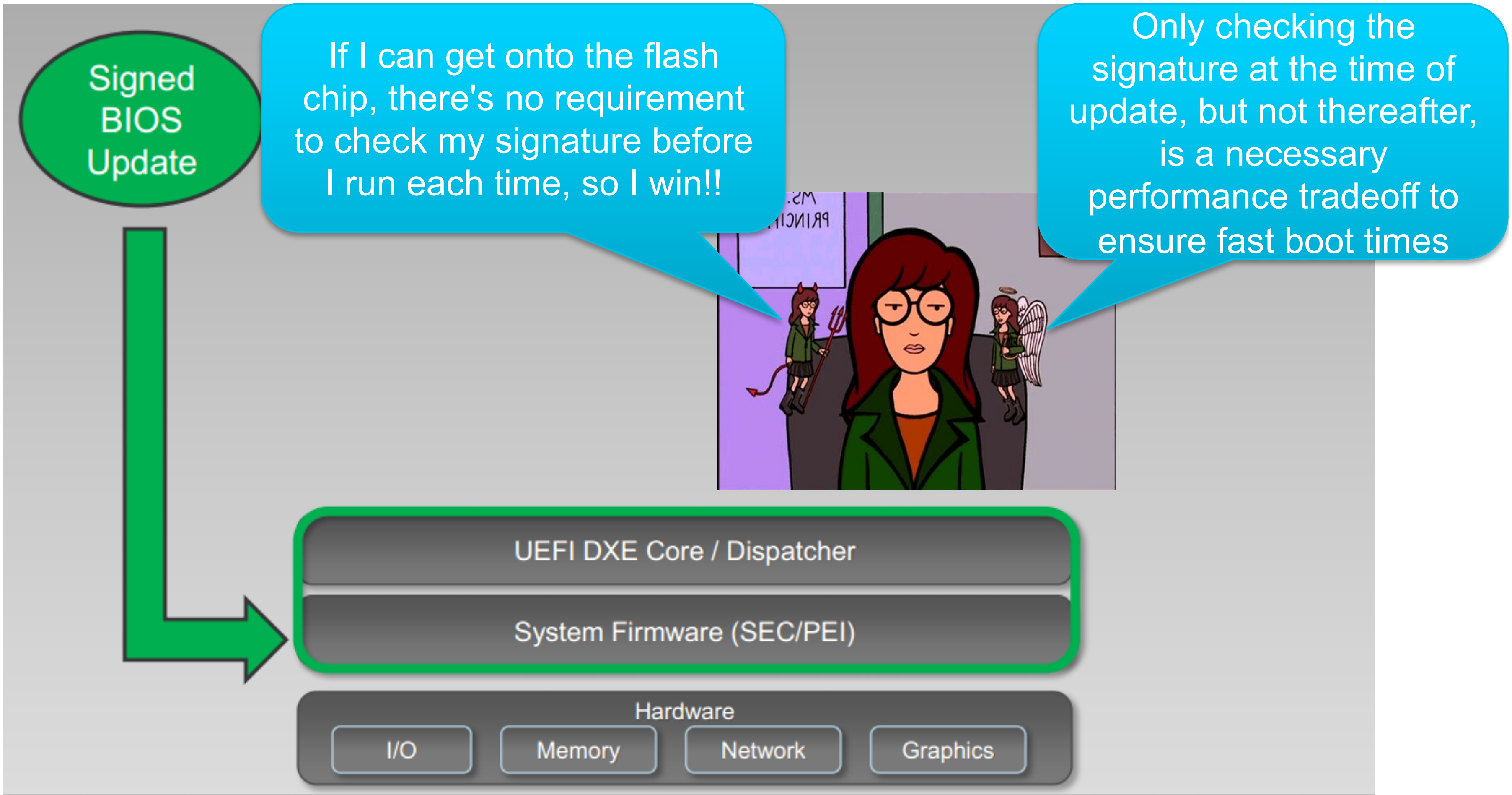


- We haven't checked extensively, but we don't think anyone is doing anything with this right now
- We think it's just something put there so that architecturally they would have the option to do "stuff" upon graceful shutdown (e.g. clearing secrets?)

Where does UEFI SecureBoot fit into all this?

- **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)

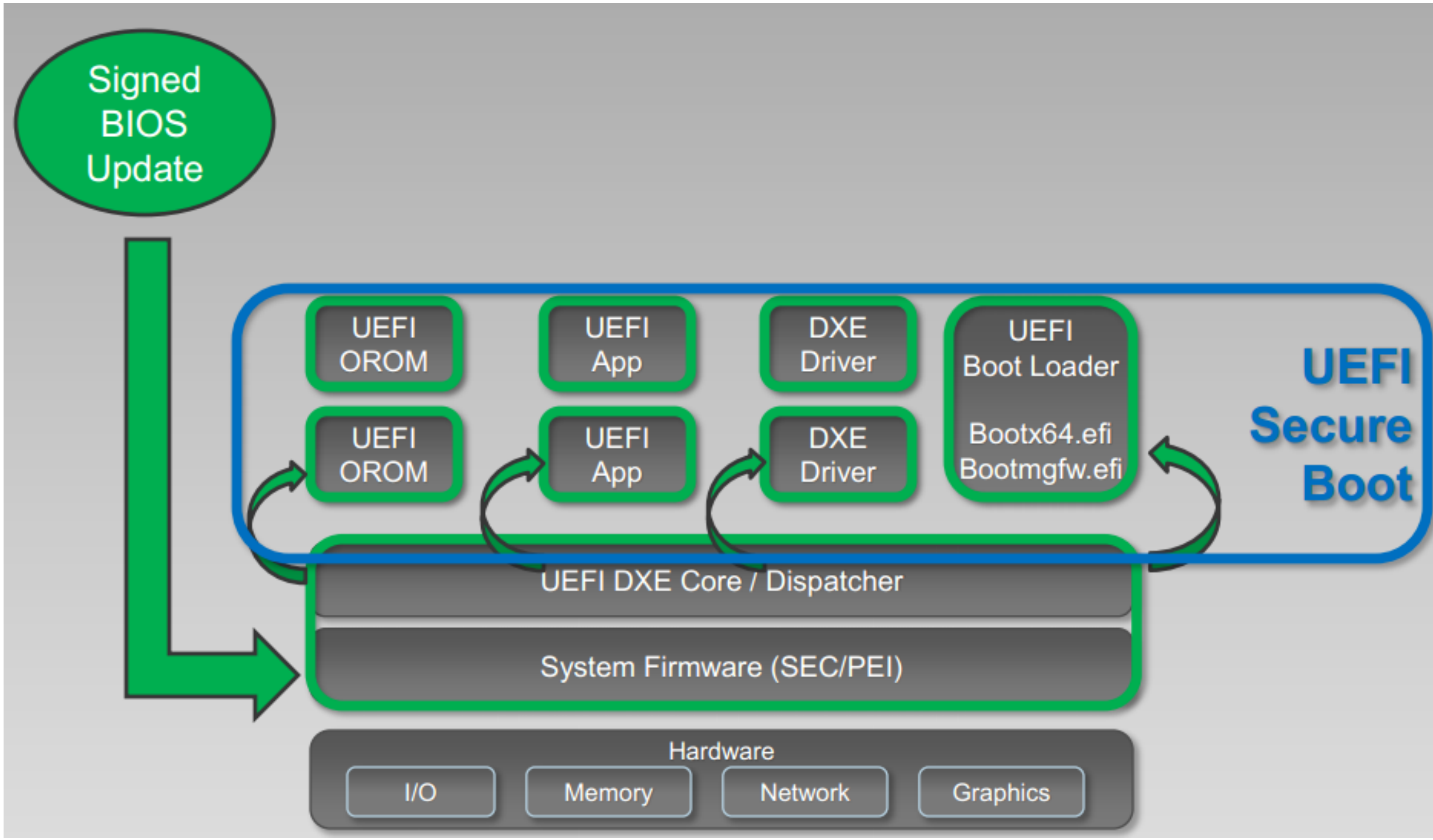
Firmware Signing



- **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

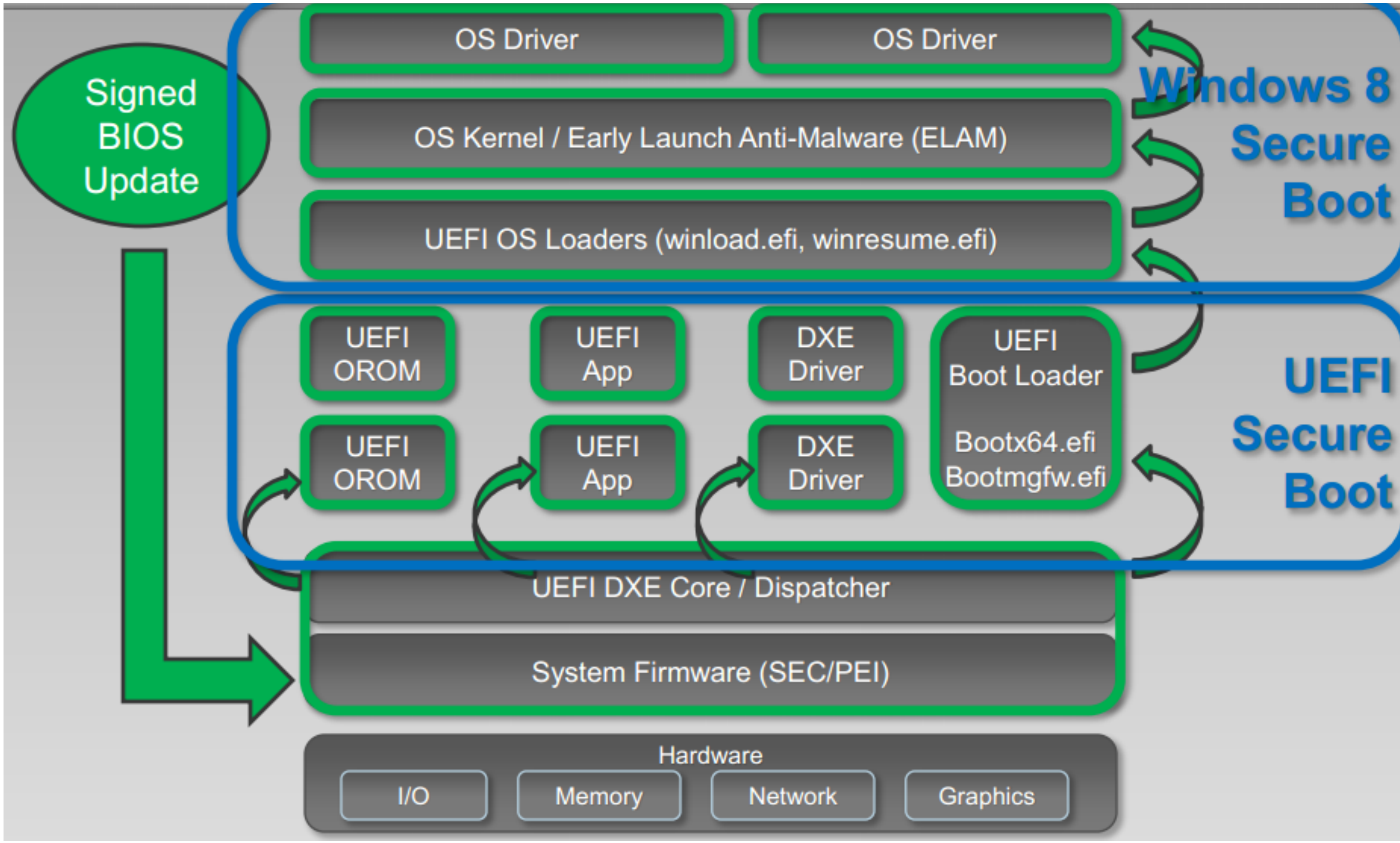
UEFI Secure Boot



- **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

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

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

Demo: Back to the FFS!

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

Structure

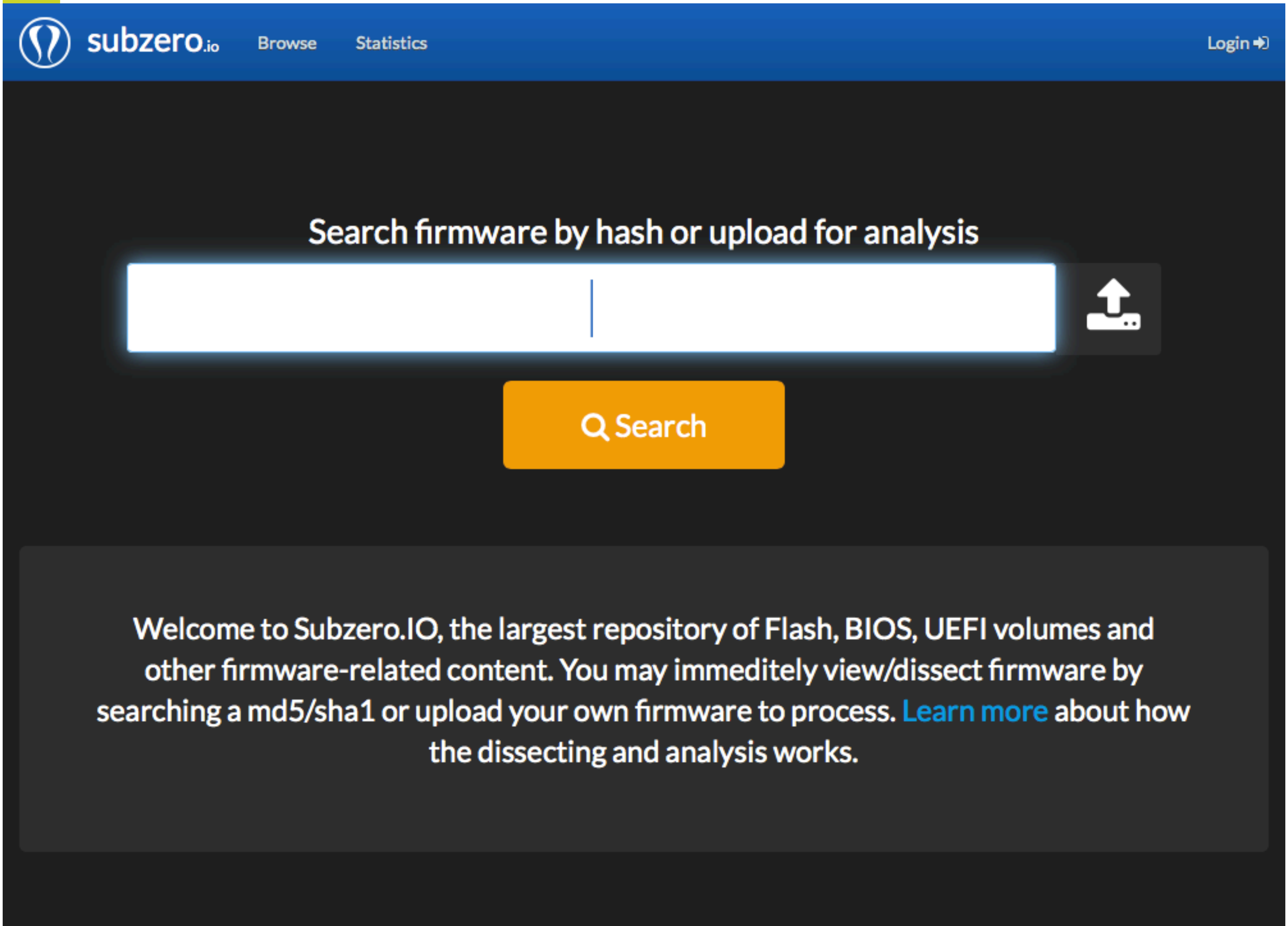
Name	Action	Type	Subtype	Text
▼Intel image		Image	Intel	
Descriptor region		Region	Descriptor	
GbE region		Region	GbE	
ME region		Region	ME	
▼BIOS region		Region	BIOS	
Padding		Padding		
▼7A9354D9-0468-444A-81CE-0BF617D890DF		Volume		
▼4A538818-5AE0-4EB2-B2EB-488B236570…		File	Volume image	
▼Compressed section		Section	Compressed	
Raw section		Section	Raw	
▼Volume image section		Section	Volume image	
▼7A9354D9-0468-444A-81CE-0BF61…		Volume		
▼35B898CA-B6A9-49CE-8C72-904…		File	DXE core	DxeMain.efi
PE32+ image section		Section	PE32+ image	
User interface section		Section	User interface	
▶4D37DA42-3A0C-4EDA-B9EB-BC0…		File	PEI module	SystemPpisNeededByDxeCore.efi
▶9EA5DF0F-A35C-48C1-BAC9-F63…		File	DXE driver	SystemCapsuleRt.efi
▶1C6B2FAF-D8BD-44D1-A91E-732…		File	DXE driver	SystemBootScriptSaveDxe.efi
▶B601F8C4-43B7-4784-95B1-F42…		File	DXE driver	SystemRuntimeDxe.efi
▶F1EFB523-3D59-4888-BB71-EAA…		File	DXE driver	SystemSecurityStubDxe.efi
▶07A9330A-F347-11D4-9A49-009…		File	DXE driver	SystemMetronomeDxe.efi
▶246F9F0A-11E3-459A-AE06-372…		File	DXE driver	SystemStatusCodeGenericRt.efi
▶29A1A717-36E9-49E0-B381-EA3…		File	DXE driver	SystemStatusCodePort80Rt.efi
▶A196BA47-8ED3-4188-A765-FA9…		File	DXE driver	SystemErrorLogDxe.efi
▶4D62B5E9-71C8-412A-8604-878…		File	DXE driver	SystemErrorLogSmm.efi
▶DA5D9983-033C-4823-9349-8B1…		File	DXE driver	SystemStatusCodeGenericSmm.efi
▶C0CFEB8B-6EE1-443B-BCC9-854…		File	DXE driver	SystemStatusCodePort80Smm.efi
▶FCE47C4E-5ECC-4A41-B90E-0BA…		File	DXE driver	SystemSecureFlashSleepTrapSmm.efi
▶15DD5676-2679-4E24-9CAA-85B…		File	DXE driver	SecureFlashVerifySmm.efi
▶793CBEA0-DA56-47F2-8264-243…		File	DXE driver	SystemVariableDxe.efi
▶65246A3B-33EF-4F7E-B657-A4A…		File	DXE driver	SystemVariableSmm.efi

Information

Type: 10
Size: 00fe60

Subzero.io

- **Ted Reed has created a website that allows you to upload BIOS files, and they will be processed with his UEFI firmware parser**
  - Similar to firmware.re, but PC BIOS specific
  - Does one thing and does it well
- **Just in time for BH EUR, he also started parsing Copernicus CSV output with protections.py in order to report whether your BIOS is vulnerable or not**
- **In a business day or two, a new version of Copernicus should be bundled and posted to the MITRE website which has a script which will automatically submit your BIOS for analysis at the site.**
- **The site will serve to crowd source what good BIOSes look like, so that we can report when we see something that doesn't look like everyone elses**


subzero.io
Browse
Statistics
Login
Search firmware by hash or upload for analysis
Search
Welcome to Subzero.IO, the largest repository of Flash, BIOS, UEFI volumes and other firmware-related content. You may immeditely view/dissect firmware by searching a md5/sha1 or upload your own firmware to process. Learn more about how the dissecting and analysis works.

subzero.io Browse Statistics Login Search

- Firmware Details
- Copernicus Report
- Structure 3
- Data Parts
- UEFI Files 177
- Variables
- Strings 9314
- Similarity
- Analysis

Copernicus Report

Report ID: ac00401fe39121bb2a0ace91addce61818976b3c

Protections

Protection	Setting	Description
BIOSWE_LOCK	False	BIOS write enable is unlocked
D_LCK	False	SMRAM writable
PR_COVERS_BIOS	False	BIOS writeable
SMI_LOCK	False	SMI is unlocked
SMM_BWP	False	SMM Write Protect disabled
SMRR_ENABLED	True	System Management Range Register

System Attributes

Attribute	Setting	Description
BIOSWE_LOCK	False	No description available
BIOS_CNTL	00	No description available

MITRE

Identifying Changes in BIOS (bios_diff.py)

```
C:\Tools\CoP>python bios_diff.py -dpan -e C:\EFIPWN-sam\EFIPWN "F:\UEFI Binaries\e6430A03.bin" "F:\U
FI Binaries\e6430A03_haxed.bin" -o .
```

- **Copernicus provides us the full dump of the BIOS flash**
  - Any BIOS dump should work as long as it's a UEFI BIOS (structured for better parsing)
- **Comparing BIOS dumps over time can provide change detection**
- **bios_diff.py now out of beta and included with Copernicus**
  - "python bios_diff.py –dpan –e <path to EFIPWN> <path to file 1> <path to file 2> -o <output directory>"
- **This script uses EFIPWN to parse and diff the modules between two BIOS dumps**
- **EFIPWN decomposes the BIOS into its firmware volumes (FVs) and then decomposes those into the individual files/modules**
- **In this example we're analyzing an earlier "known-good" BIOS with one which we suspect has changed**
  - We took a known good and purposefully made a small change in the "haxed" one

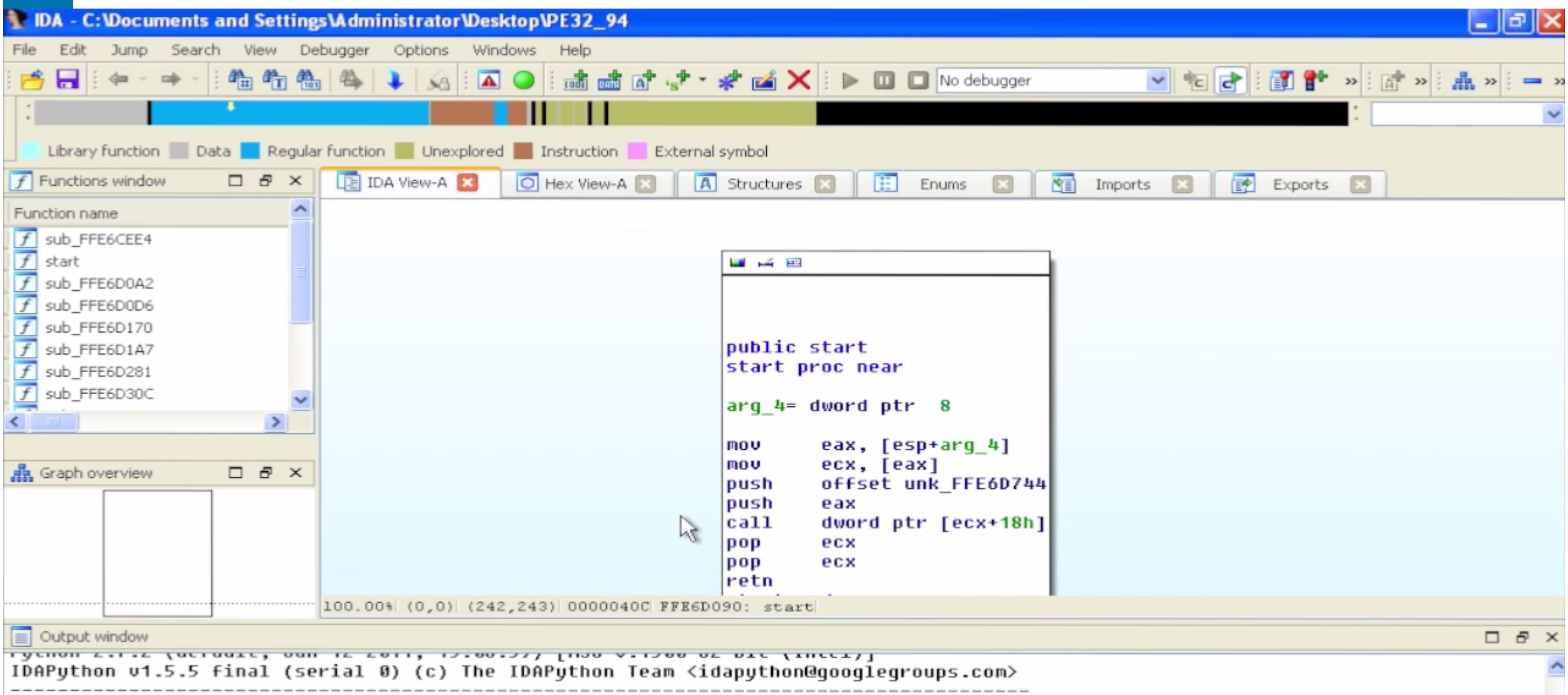
Identifying Changes in BIOS (bios_diff.py)

```
C:\Tools\CoP>python bios_diff.py -dpan -e C:\EFIPWN-sam\EFIPWN "F:\UEFI Binaries\e6430A03.bin" "F:\U
FI Binaries\e6430A03_haxed.bin" -o .
Differing file found:
.\e6430A03.bin\fv3\e9312938-e56b-4614-a252-cf7d2f377e26\PE32_73 (AmiTcgPlatformPeiBeforeMem)
7 unique bytes out of 2976
1036,1042
PE Information:
Section .text
RVA 0x40c
VA 0xffe6d090
.\e6430A03_haxed.bin\fv3\e9312938-e56b-4614-a252-cf7d2f377e26\PE32_73 (AmiTcgPlatformPeiBeforeMem)
7 unique bytes out of 2976
1036,1042
PE Information:
Section .text
RVA 0x40c
VA 0xffe6d090
```

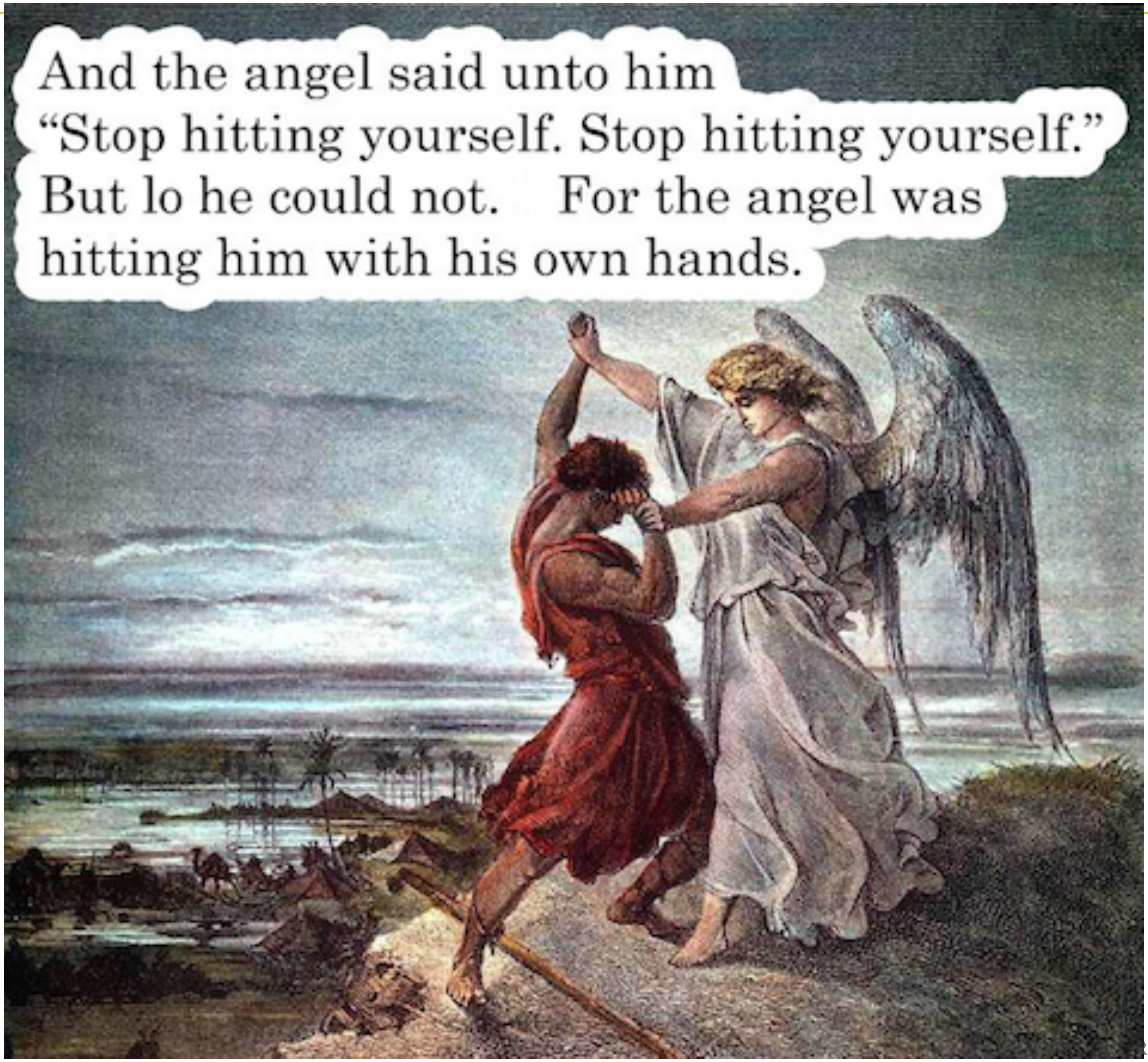
- **The script has found a difference located in firmware volume 3**
- **Some files/modules have user-friendly names and if this is the case the script outputs this name: "AmiTcgPlatformPeiBeforeMem"**
- **"Ami" could mean it's derived from an AMI (American Megatrends Inc.) codebase**
- **"Tcg" could be Trusted Computing Group and "BeforeMem" likely means this PEIM executes before memory is established**

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**



Conclusion


And the angel said unto him
“Stop hitting yourself. Stop hitting yourself.”
But lo he could not. For the angel was
hitting him with his own hands.

Conclusion

- **UEFI brings with it important improvements to boot-time security, and**
  - That's why we already do it with Copernicus, and are looking for commercial organizations to incorporate equivalent capabilities
  - Google "MITRE Copernicus" to download the binary-only version
  - Contact us with a proposal of what data you will share to get the src
- **Standardization, being programmed in a high level language, availability of developer platforms, and an open source code base has made it easier for attackers to get started creating firmware attacks**
  - Thanks to the FUD surrounding UEFI SecureBoot, it's highly unlikely that typical open source advocates are doing extensive code review/contribution to the UEFI EDK2 code base
- **We can no longer ignore firmware security as a credible threat**

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 :)

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

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

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

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] All your boot are belong to us – Joint Intel & MITRE talk – Kallenberg et al. & Bulygin et al., Mar 2014**
  - MITRE slides: https://cansecwest.com/slides/2014/AllYourBoot_csw14-mitre-final.pdf
  - Intel slides: https://cansecwest.com/slides/2014/AllYourBoot_csw14-intel-final.pdf
- **[30] Setup for Failure: Defeating UEFI Secure Boot – Kallenberg et al., Apr 2014 http://syscan.org/index.php/download/get/6e597f6067493dd581eed737146f3afb/SyScan2014_CoreyKallenberg_SetupforFailureDefeatingSecureBoot.zip**
- **[31] "Charizard" attack (CERT VU#758382) demo video - http://youtu.be/XfJ4S42MVLw**

References 6

- [32] SENTER Sandman: Using Intel TXT to Attack BIOSes – Kovah et al., June 2014 - slides not posted anywhere yet
- [33] 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
- [34] UEFI and PCI Bootkits, Pierre Chifflier, Nov 2013 - http://pacsec.jp/psj13/psj2013-day2_Pierre_pacsec-uefi-pci.pdf
- [35] Rootkit Detection Framework for UEFI (RDFU) - Vuksan & Pericin, July 2013 - http://www.reversinglabs.com/technology/open-source.html