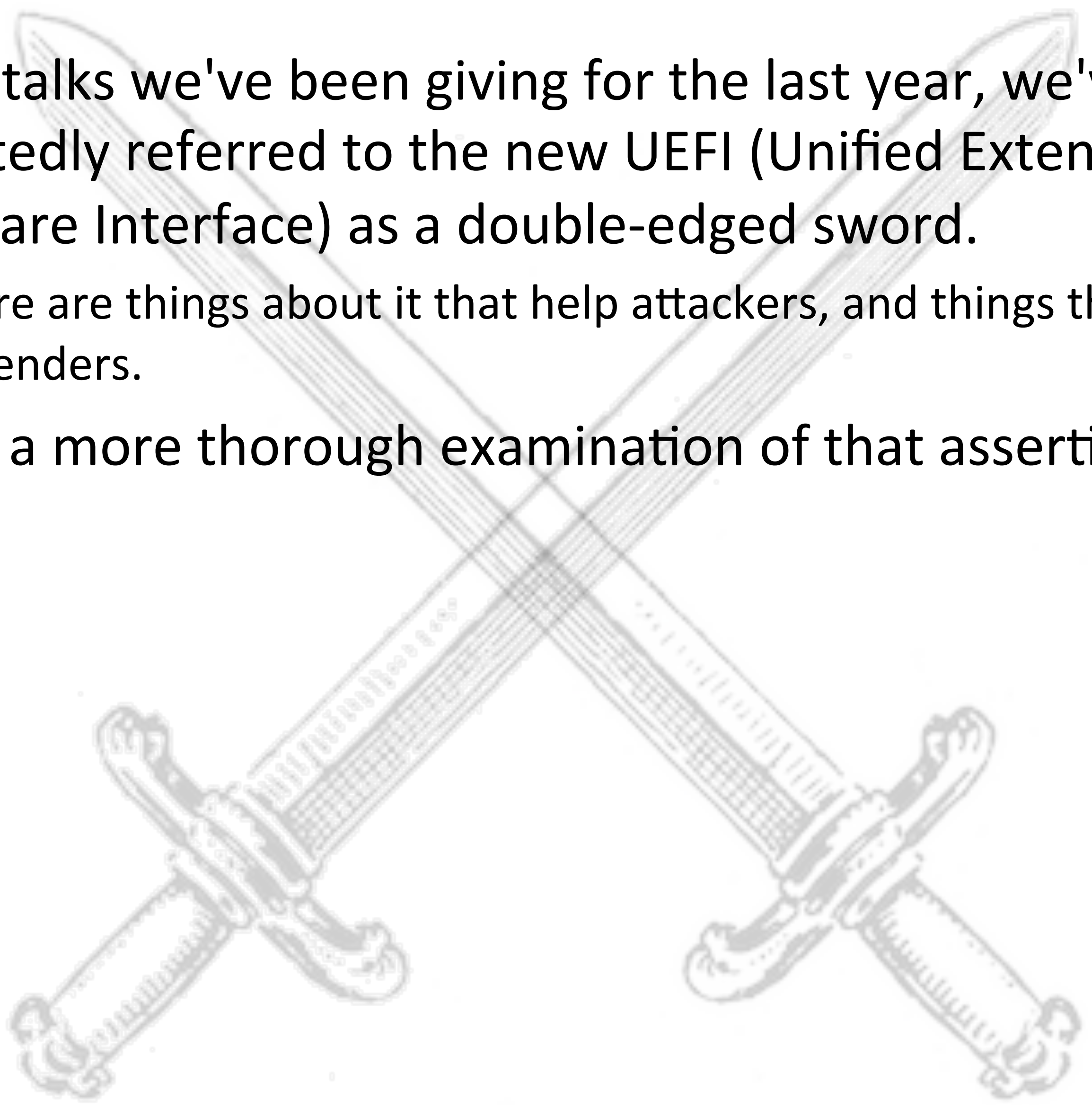
Advanced x86: BIOS and System Management Mode Internals *Unified Extensible Firmware Interface (UEFI)*

Xeno Kovah && Corey Kallenberg

LegbaCore, LLC

LEGBACORE
WE DO DIGITAL VOODOO

Introduction



- 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.
  - there are things about it that help attackers, and things that help defenders.
- This is a more thorough examination of that assertion

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 “release tag” of the “cutting edge” EDK2 (EFI Development Kit)

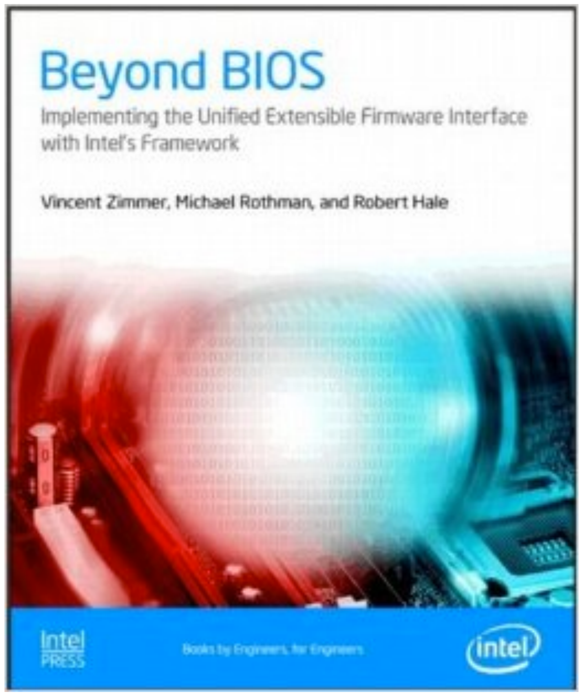
About UEFI

- 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!)

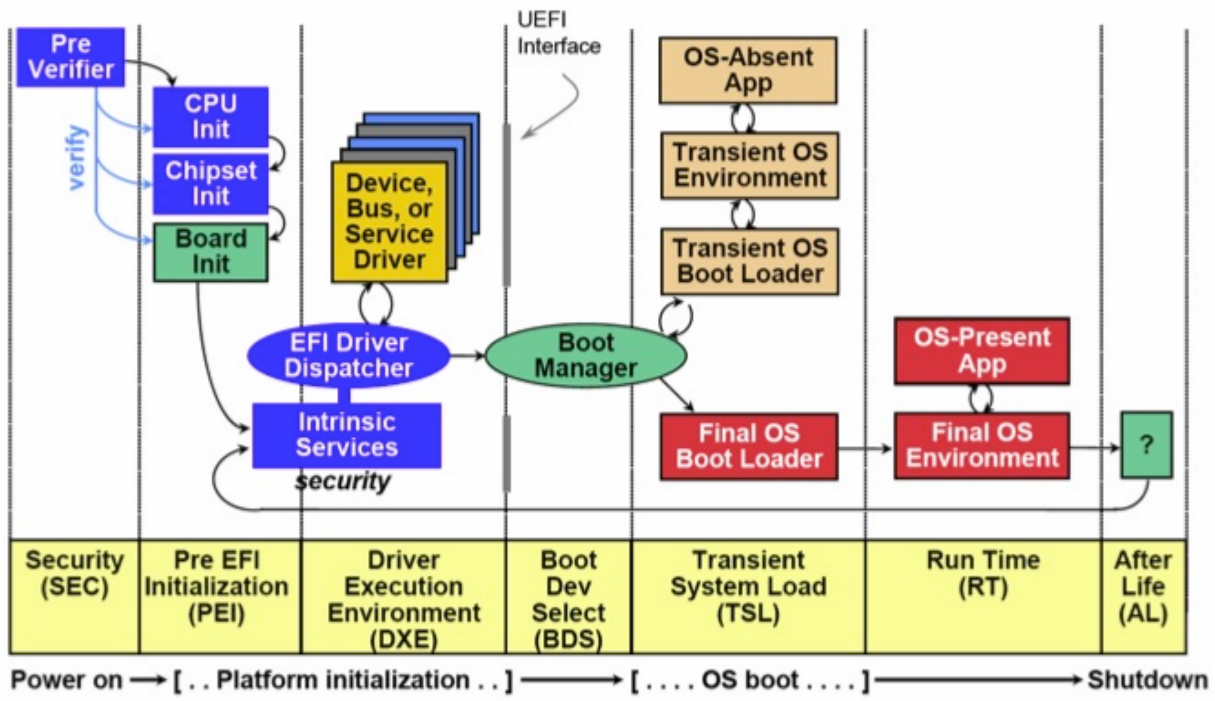
Something you may want to read

- 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

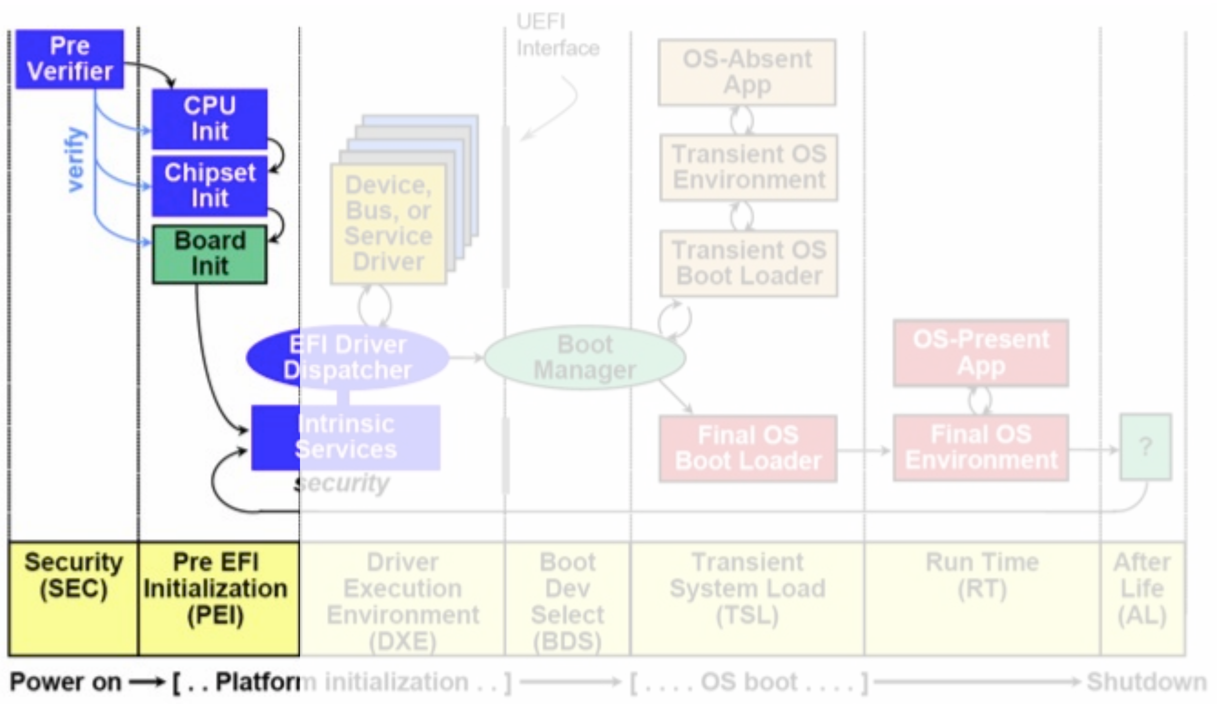

Beyond BIOS
Implementing the Unified Extensible Firmware Interface with Intel's Framework
Vincent Zimmer, Michael Rothman, and Robert Hale
Intel PRESS
Books by Engineers, for Engineers
intel

UEFI Differences: Boot Phases



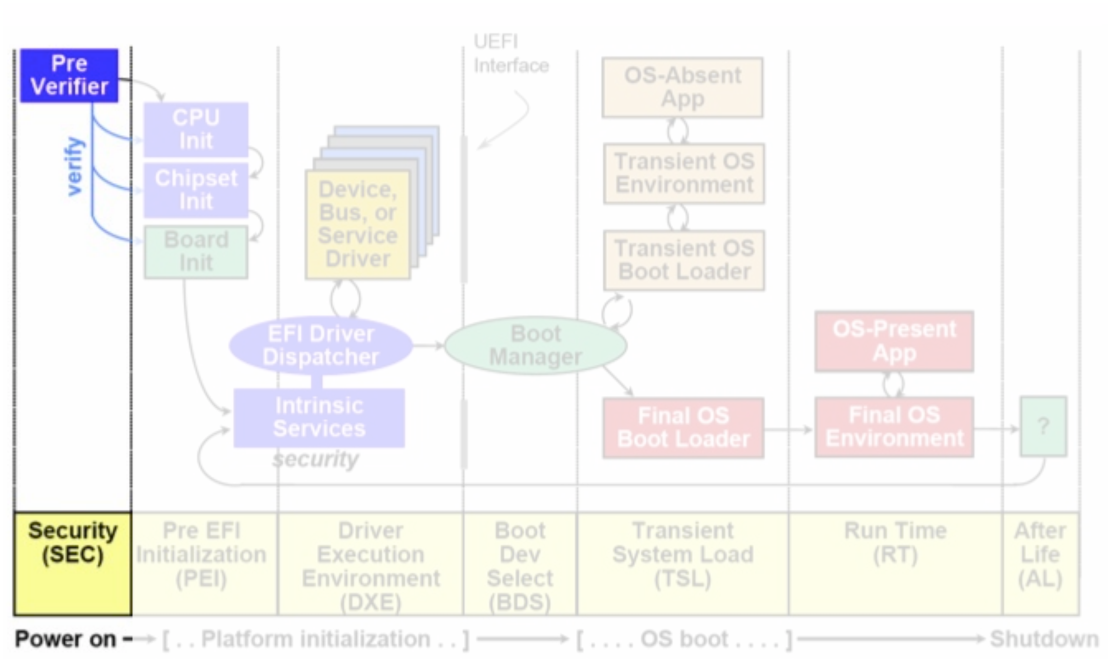
- 7 Phases total
- Each phase is defined via specification
- We'll briefly talk about the first 2 phases and then talk about Secure Boot and UEFI debugging

Legacy BIOS Equivalent



- However, everything we have covered up to this point is mostly performed within these first 2 phases
- Chipset configuration, etc.
  - Some SMM stuff happens in the 2nd phase (PEI), some in the 3rd (DXE)
- For this reason, we'll only briefly cover these first 2 phases

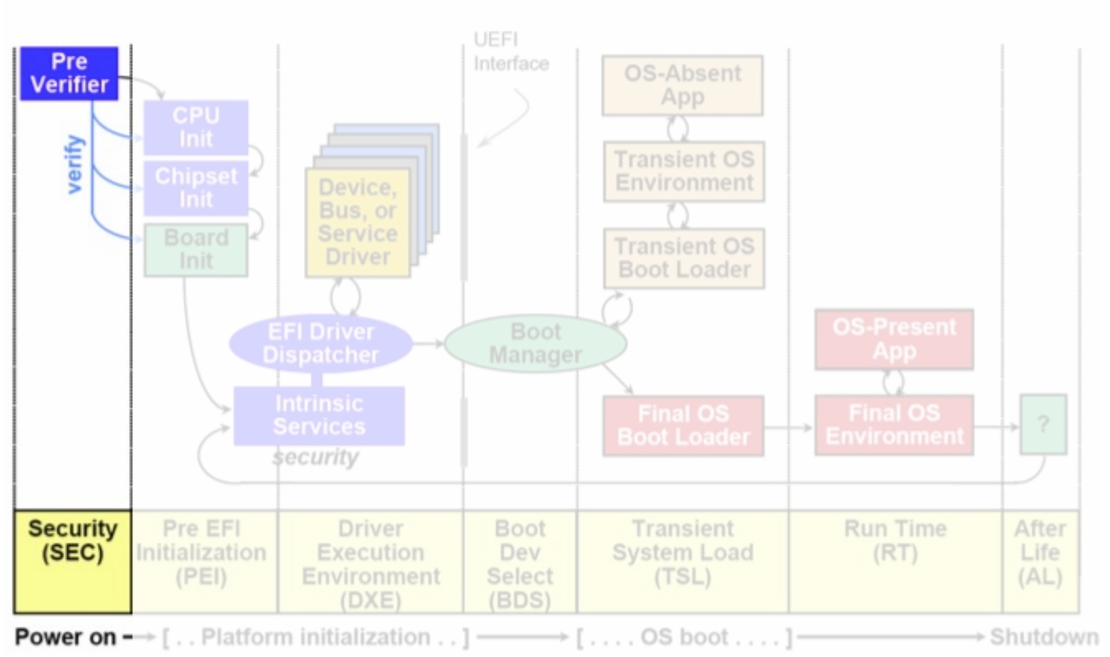
SEC (Security) Phase



- 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 (Security) Phase: Architecture vs. Implementation



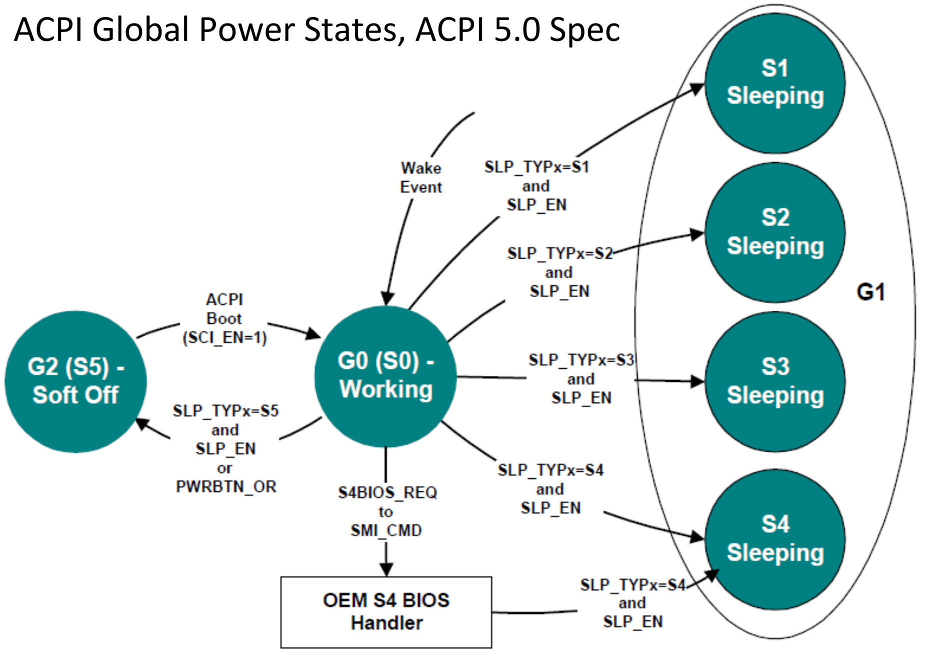
- This picture is *architecturally* correct. The SEC phase *should* involve some amount of measurement and verification of the first code to run
- Realistically if you look at the SEC core module, you will find that it actually is almost entirely CPU/Chipset/Board initialization handcoded asm
- “In theory, theory and practice are the same. In practice, they are not.”

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

SEC Responsibilities 1 of 2

- 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!

- This includes a wake-event from sleep mode, etc.
- We'll not be discussing ACPI at this time, but you can find more information in the ACPI spec:
  - http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf

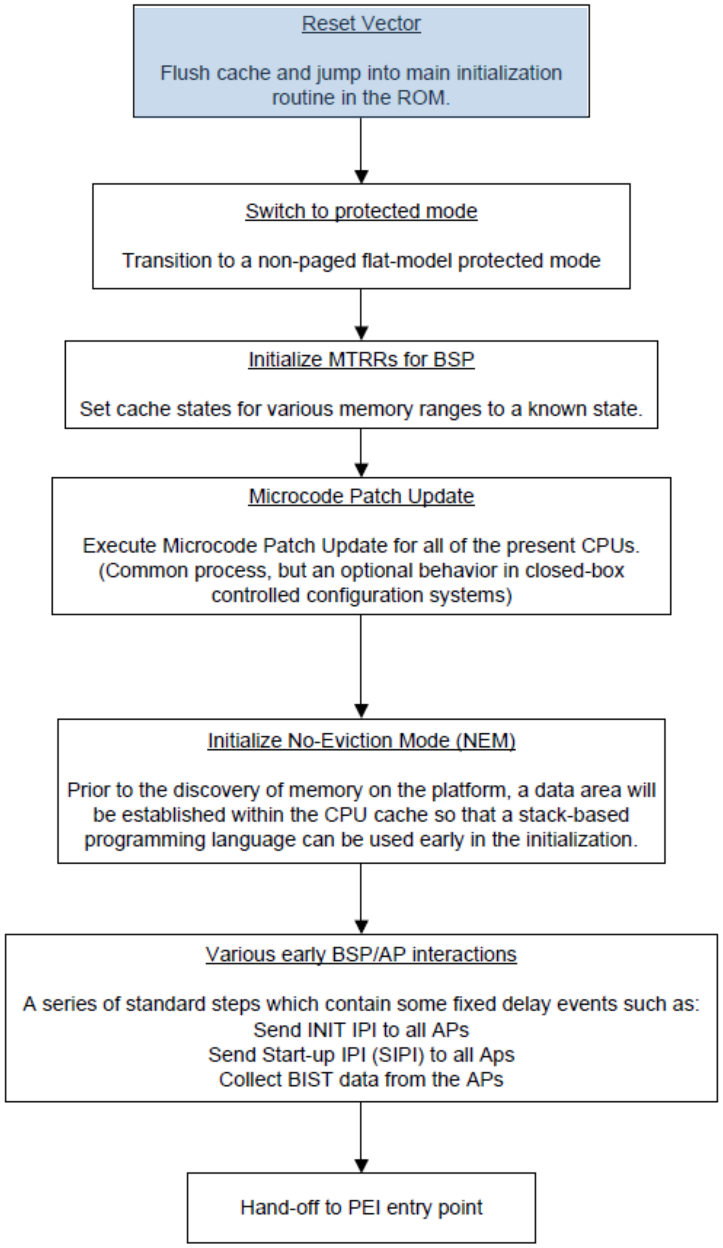
Platform Initialization Spec Vol. 1, Version 1.3, Sec. 13

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 (as in ANSI C)
- 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

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

SEC Phase

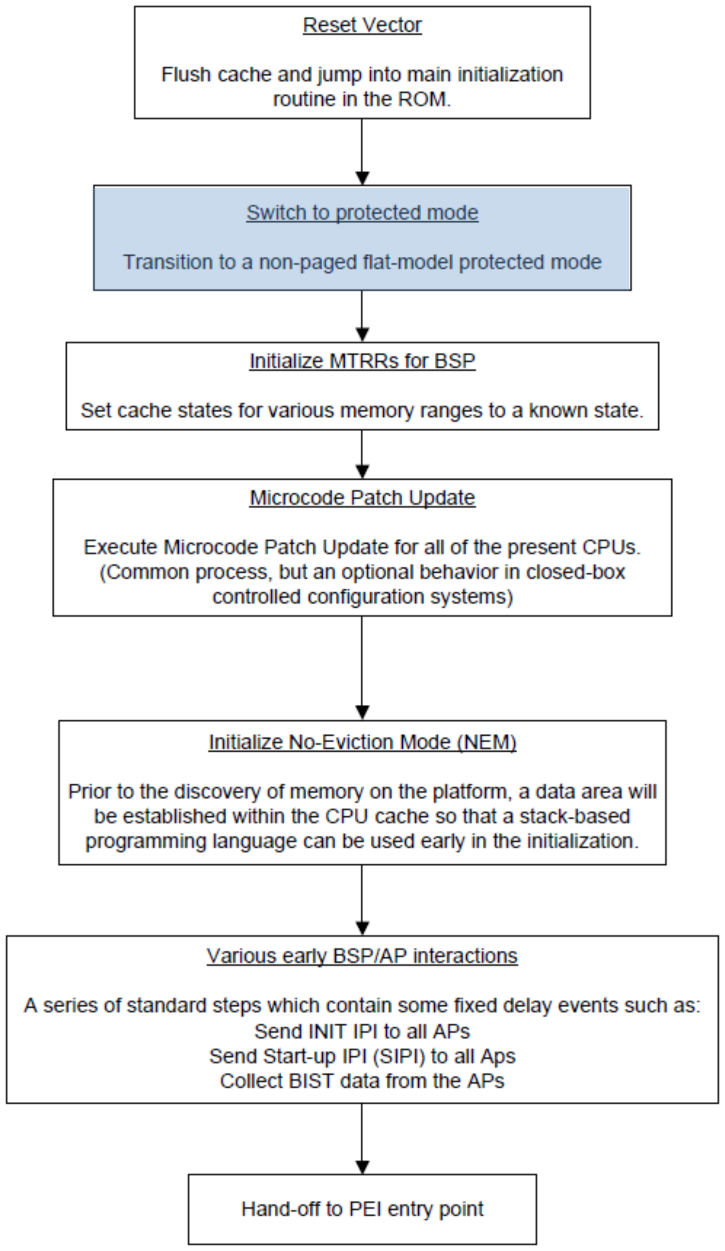


Reset Vector
Flush cache and jump into main initialization routine in the ROM.
Switch to protected mode
Transition to a non-paged flat-model protected mode
Initialize MTRRs for BSP
Set cache states for various memory ranges to a known state.
Microcode Patch Update
Execute Microcode Patch Update for all 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

- 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
  - In the UDK2010 I have seen the WBINVD instruction performed prior to the initial JMP, but in later revisions of the UDK2012 it has been replaced with NOPs
  - Which made me wonder if it was possible to cache-poison the reset vector?
  - No, because caching is disabled automatically on reset!

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

SEC Phase

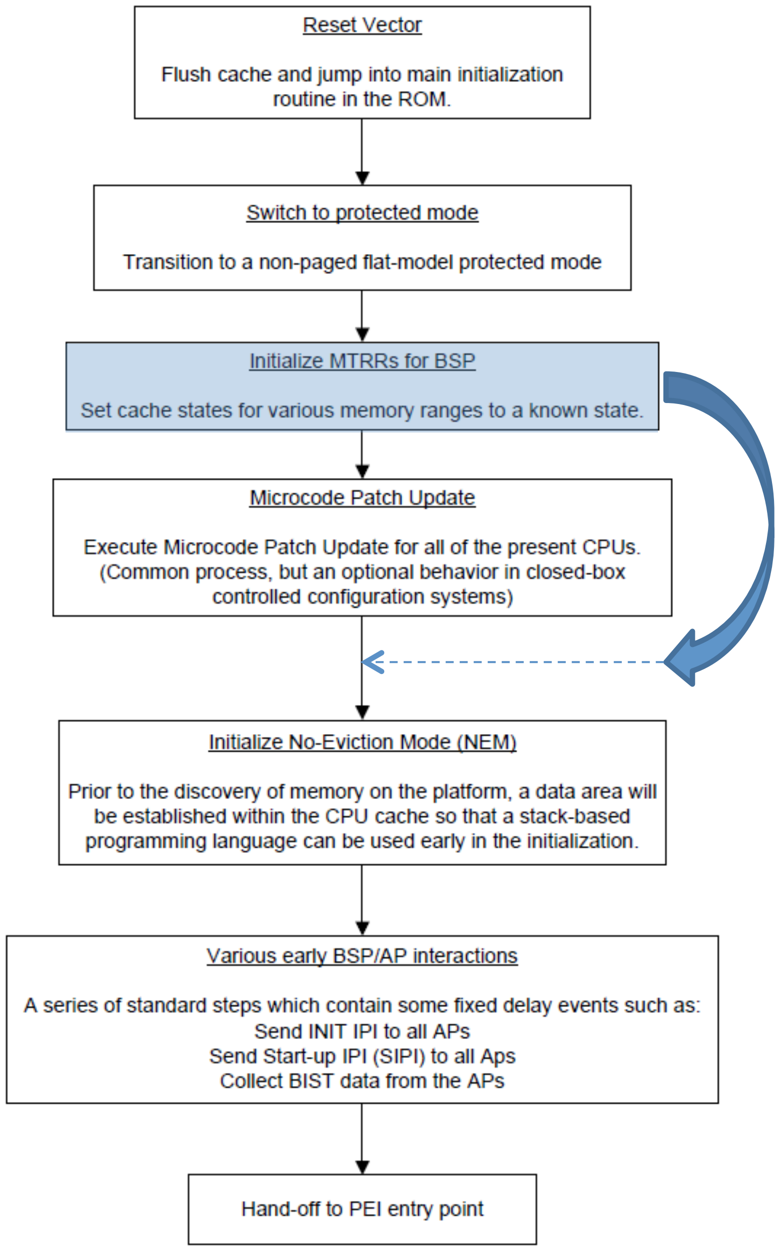


Reset Vector
Flush cache and jump into main initialization routine in the ROM.
Switch to protected mode
Transition to a non-paged flat-model protected mode
Initialize MTRRs for BSP
Set cache states for various memory ranges to a known state.
Microcode Patch Update
Execute Microcode Patch Update for all 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

- Immediate entry into protected mode after initial jump
- UDK2012 vB12: Loads a GDT from FFFF_FF98h
- UDK2012 vB12 does not load an IDT (not even a nominal one of all zeroes)
  - Just an observation, nothing wrong with this
- Immediately following the entry into protected mode, the Boot Firmware Volume (BFV) is located
  - GUID: 8C8CE578-8A3D-4F1C-3599-35896 185C32DD
- Also locates the SecCore FFS

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

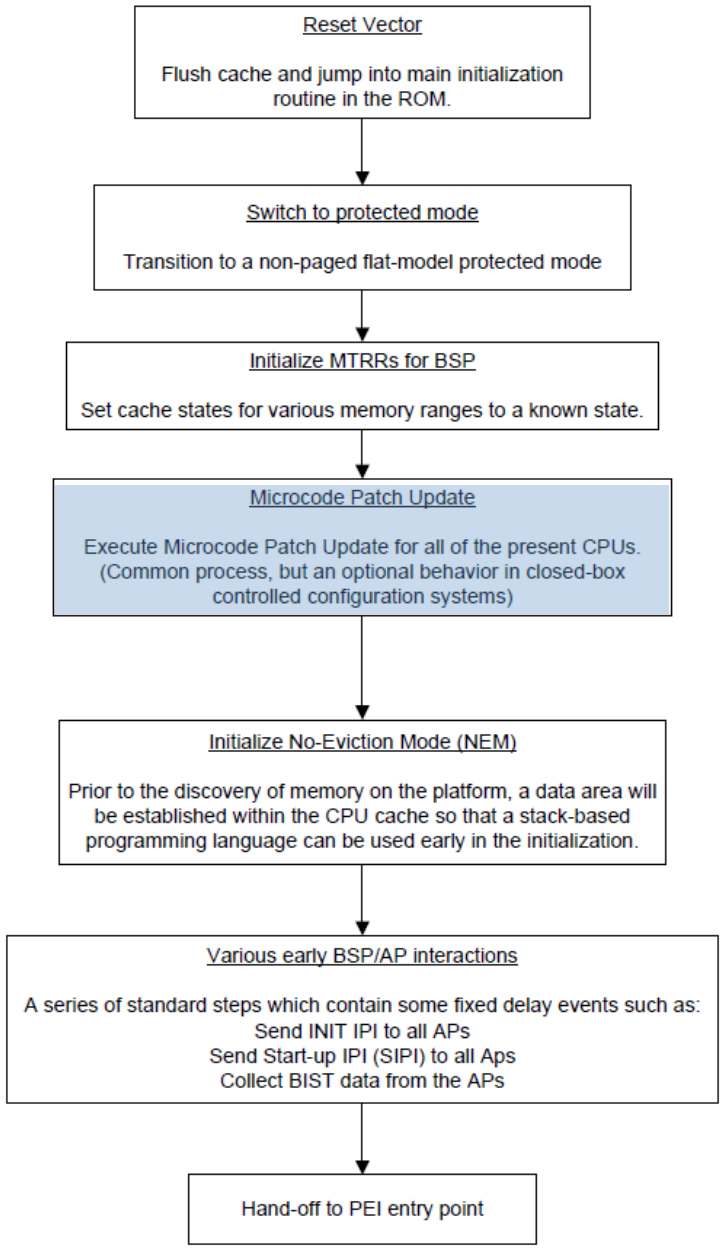
SEC Phase


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

- In the UDK2012 vB12 MTRRs are initialized just before CAR is initialized
- MTRR Default Memory Type is configured as Uncacheable
- UDK2010 also write-protects the Boot Firmware Volume (BFV)
  - An MTRR PhysBase to FFF0_0005h
  - An MTRR PhysMask to FFF0_0800h

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

SEC Phase

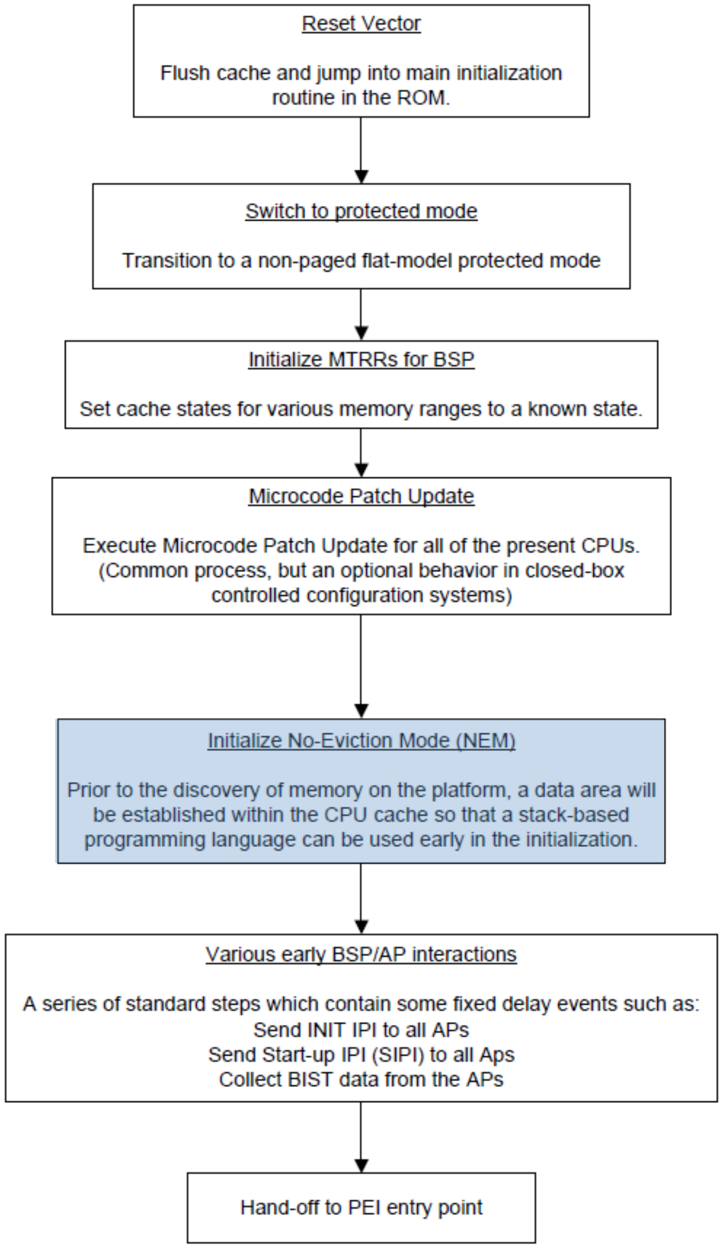


Reset Vector
Flush cache and jump into main initialization routine in the ROM.
Switch to protected mode
Transition to a non-paged flat-model protected mode
Initialize MTRRs for BSP
Set cache states for various memory ranges to a known state.
Microcode Patch Update
Execute Microcode Patch Update for all 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

- At this point in the UDK2012/EDK2 we are in SecCoreEntry()
- Cool paper on Microcode updates (by Ben Hawkes):
- http://inertiawar.com/microcode/
- Uses data and timing analysis to reveal some of the cryptographic design of the microcode update architecture
- To an attacker, being able to modify the Microcode updates could mean they could “update” the CPU to an older revision which could contain exploitable traits/flaws, or simply insert changes to remove security checks and mechanisms
- Note: CPU microcode updates are not permanently written to the CPU, microcode updates are (re)applied each time the system boots

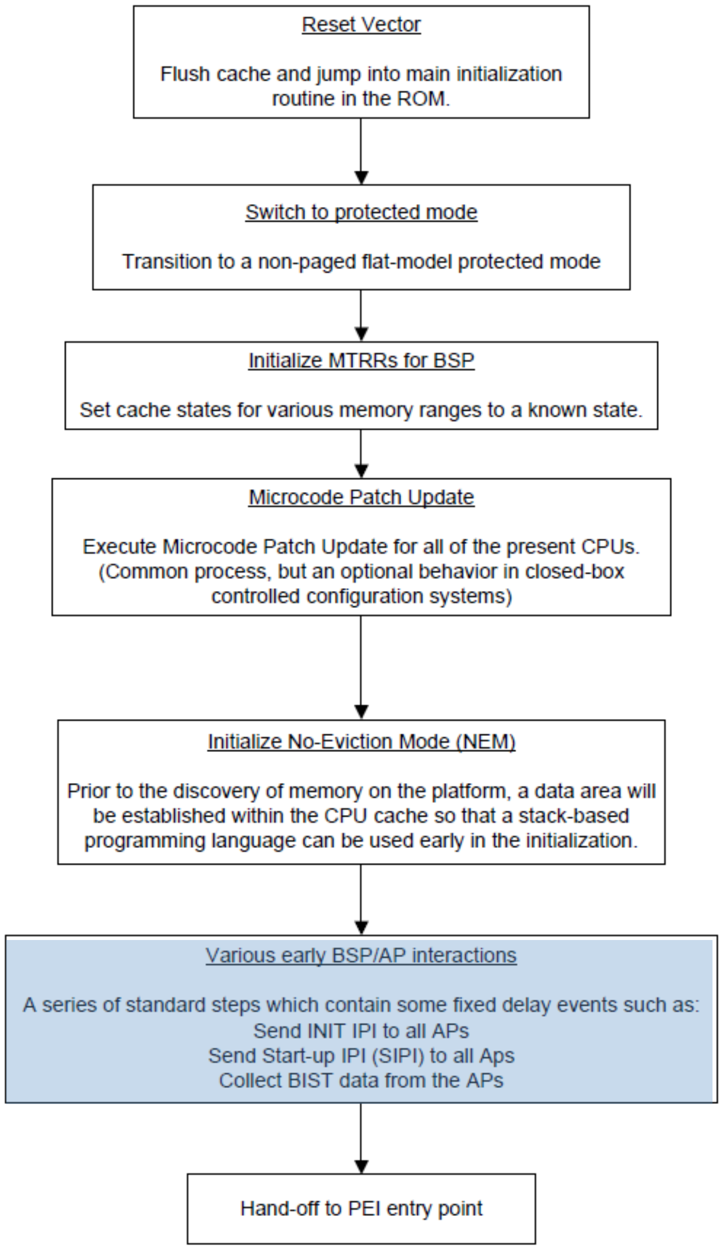
SEC Phase


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

- This is where Cache-As-RAM is initialized. No eviction mode means the CPU cache will not flush/sync to memory
- IMO I think CAR is very interesting. CoreBoot source has some well-commented code that explains the procedure very well (also used in Chromium):
- https://chromium.googlesource.com/chromiumos/third_party/coreboot/+/master/src/soc/intel/baytrail/romstage/cache_as_ram.inc
- UDK2012 vB12 allocates 8000h bytes of CPU cache to be used as RAM
- Stack region is placed in cache
  - MOV ESP, CAR_BASE_ADDR

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

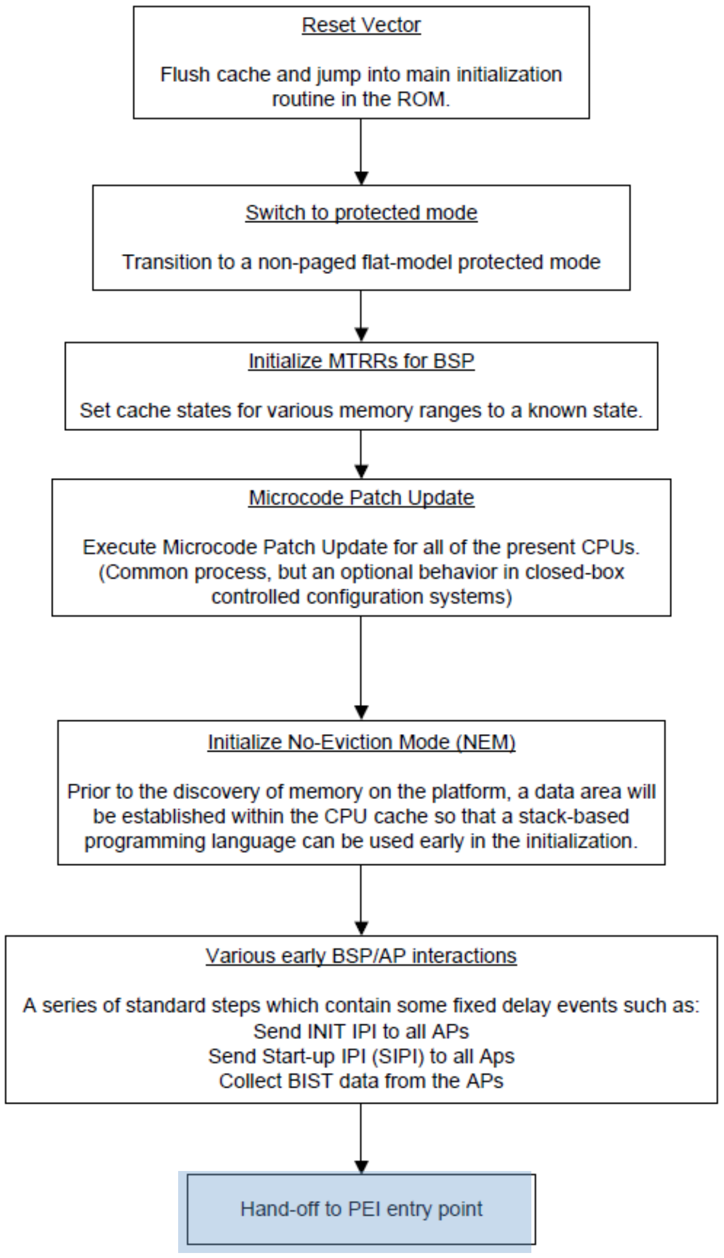
SEC Phase


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

- BIST here is Built-In Self Test that each processor performs before being "ready for duty"
- I have not observed this but there is a good chunk of configuration code that I skimmed over.
- I have not observed the other cores on the processor "wake-up" until the DXE phase

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

SEC Phase

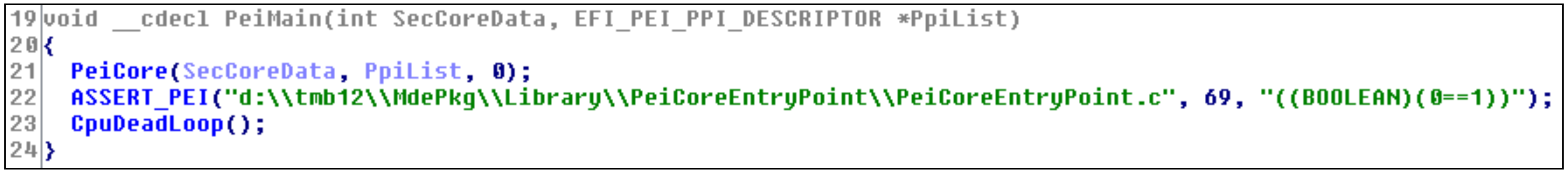


Reset Vector
Flush cache and jump into main initialization routine in the ROM.
Switch to protected mode
Transition to a non-paged flat-model protected mode
Initialize MTRRs for BSP
Set cache states for various memory ranges to a known state.
Microcode Patch Update
Execute Microcode Patch Update for all 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

- Locates the PEI Core module
- At this point it also configures some of the BARs (MCHBAR, PCIEXBAR, and some others)
  - But memory still has not yet been "discovered"
- At handoff, BIOS_CNTL is set to 0x28 Prefetching and caching are initially allowed for SMM
  - BIOS Region SMM protection is enabled
  - SMM/SMRAM are not yet instantiated

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

SEC Hand-off to PEI Entry Point

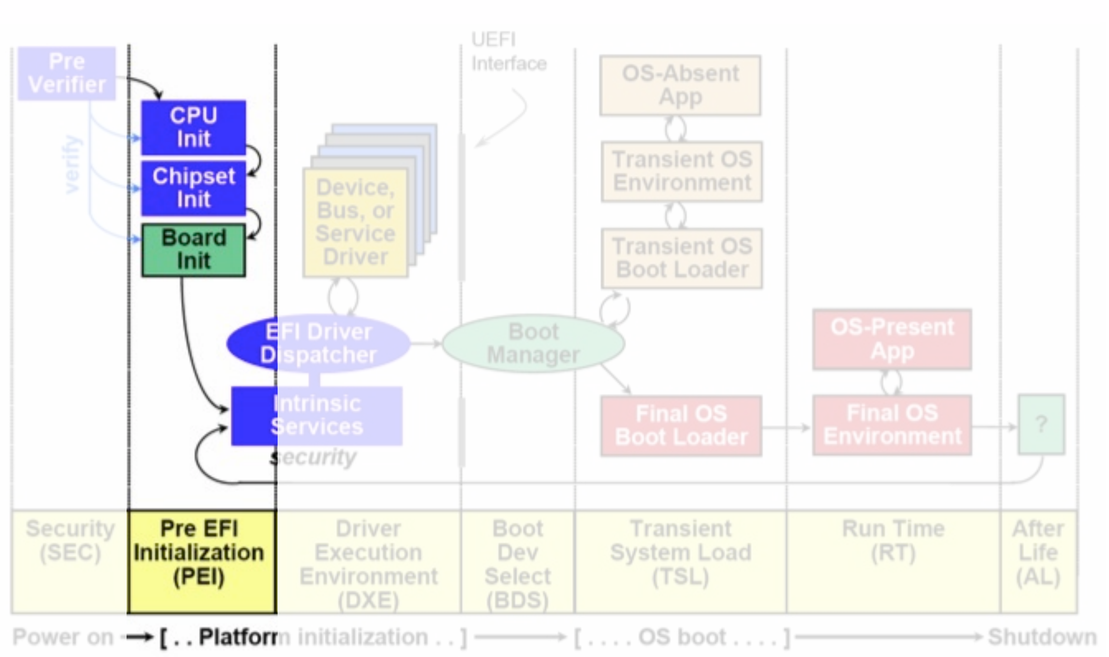


```
void __cdecl PeiMain(int SecCoreData, EFI_PEI_PPI_DESCRIPTOR *PpiList)
{
  PeiCore(SecCoreData, PpiList, 0);
  ASSERT_PEI("d:\\tmb12\\MdePkg\\Library\\PeiCoreEntryPoint\\PeiCoreEntryPoint.c", 69, "((BOOLEAN)(0==1))");
  CpuDeadLoop();
}
```

- 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 Initialization) 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 unit of code and/or data stored in a 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
  - 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.

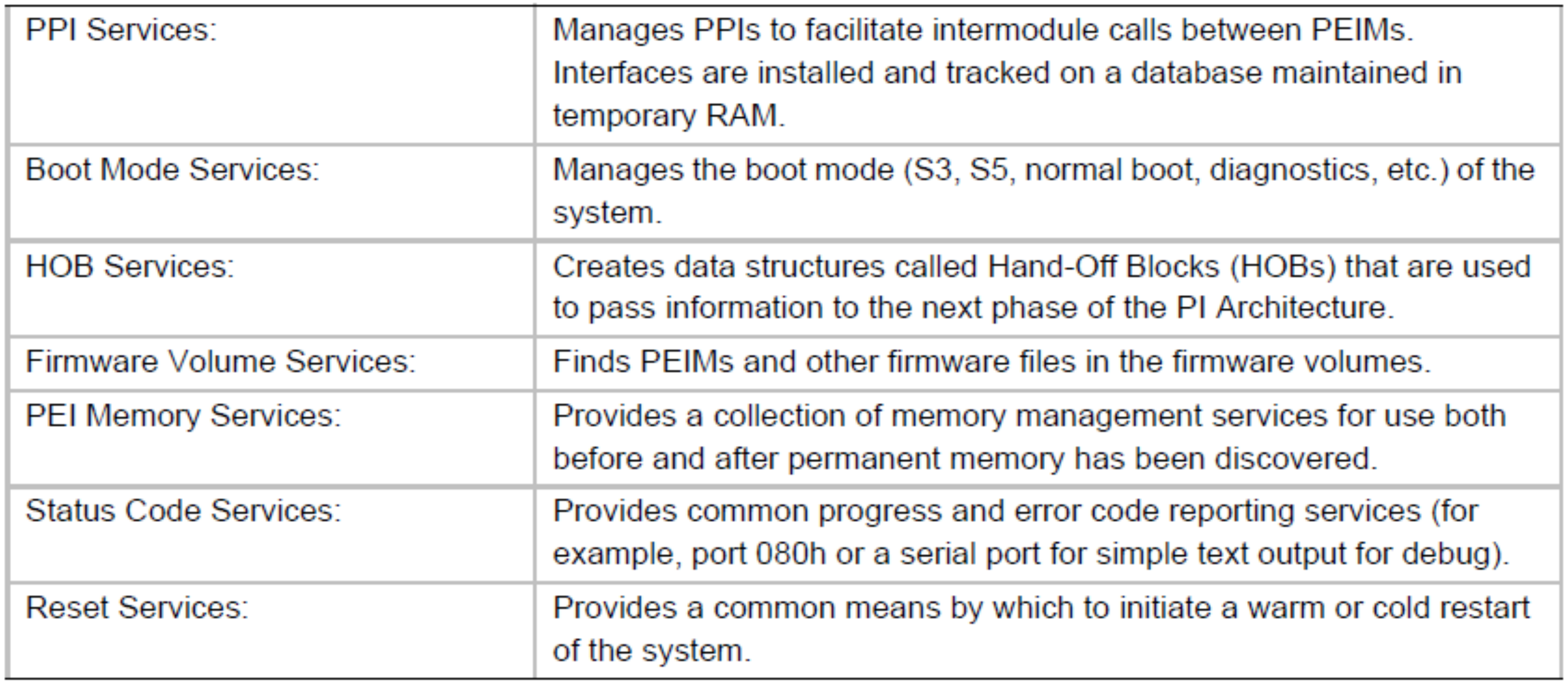
Components of PEI

- PEI Dispatcher
  - Evaluates the dependency expressions in PEIMs and, if they are met, installs them (and executes them)
  - PEIMs are dispatched a priori
- Dependency Expression(DEPEX)
  - Basically GUIDs of PPIs that must have already been dispatched before a PEIM is permitted to load/execute
- Firmware Volumes

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.

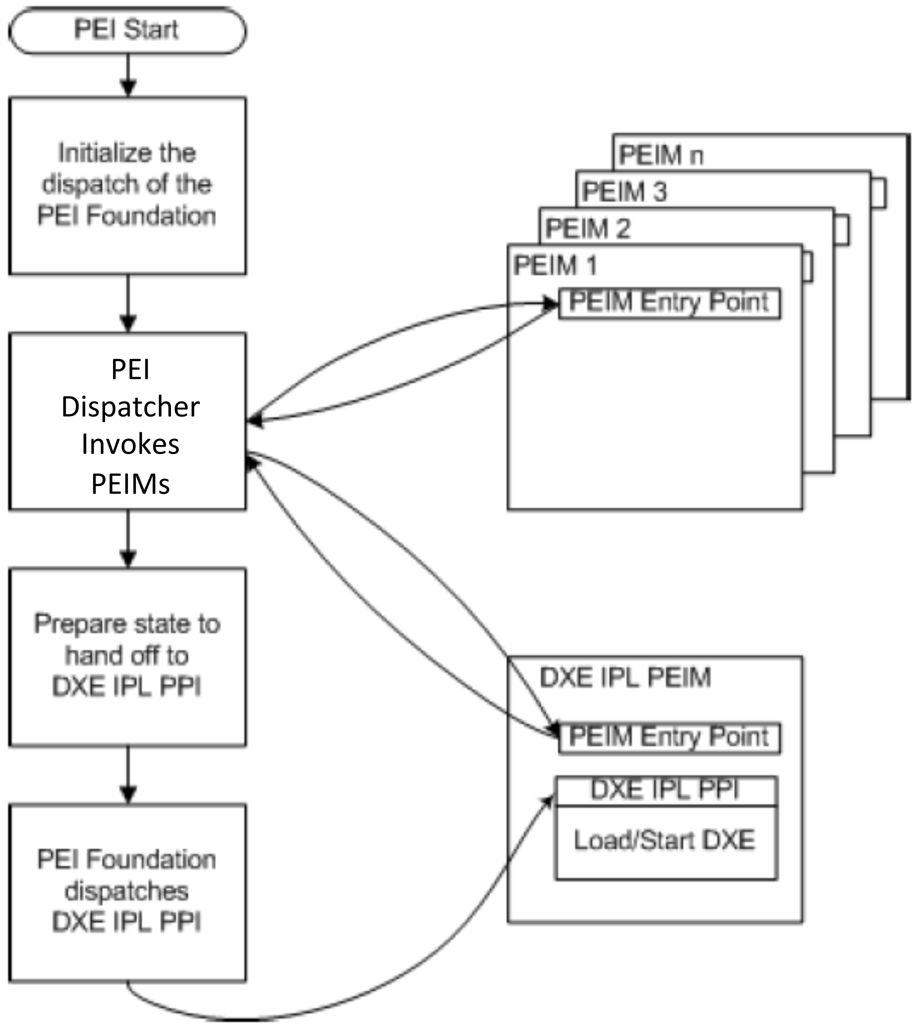
Extensive list of all PPIs can be found in Platform Initialization Spec Vol. 1, Version 1.3, Section 3.1

As the tables turn… 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

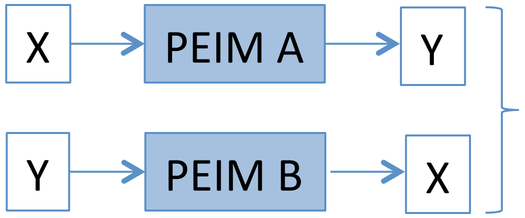

PEI Start
Initialize the dispatch of the PEI Foundation
PEI Dispatcher Invokes PEIMs
Prepare state to hand off to DXE IPL PPI
PEI Foundation dispatches DXE IPL PPI
PEIM n
PEIM 3
PEIM 2
PEIM 1
PEIM Entry Point
DXE IPL PEIM
PEIM Entry Point
DXE IPL PPI
Load/Start DXE

- 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.
- One of these PEIMs will be the DXE IPL (Initial Program Load) PEIM which will perform the transition to the DXE phase when all PEIMs that can be invoked have been invoked

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

PEI Dispatcher

- The PEI Dispatcher is a state machine and central to the PEI phase
- Evaluates each dependency expressions (DEPEXes) of PEIMs which are evaluated
- DEPEX is a list of list of GUIDs for PPIs & some logic associated with the condition that is desired (e.g. PPI must be loaded before this module's DEPEX is satisfied)
- If the DEPEX evaluates to True, the PEIM is invoked, otherwise the Dispatcher moves on to evaluate the next PEIM

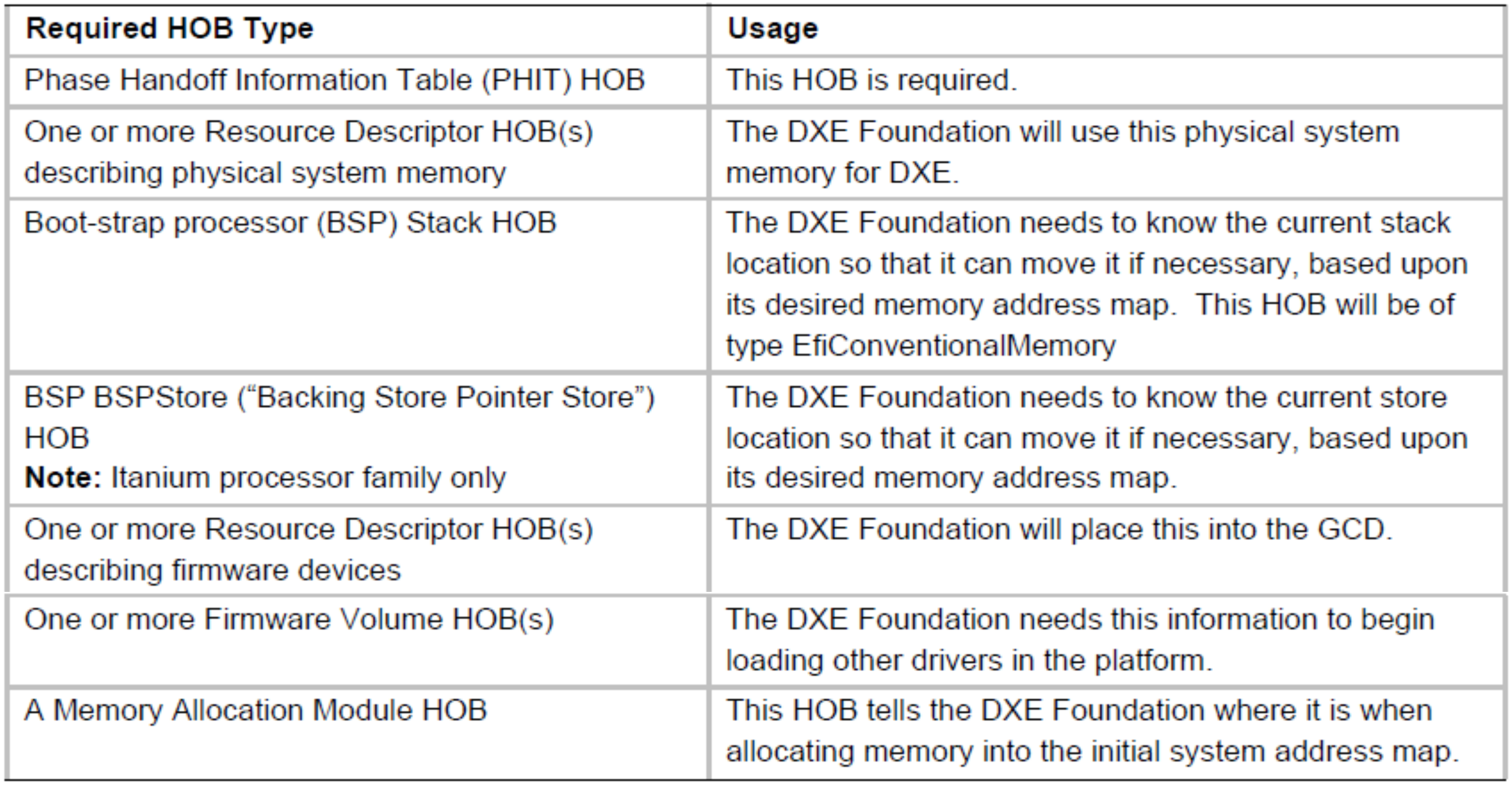


UEFI will prevent both PEIMs A and B in this endless cycle from executing. X and Y are PPIs

- 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

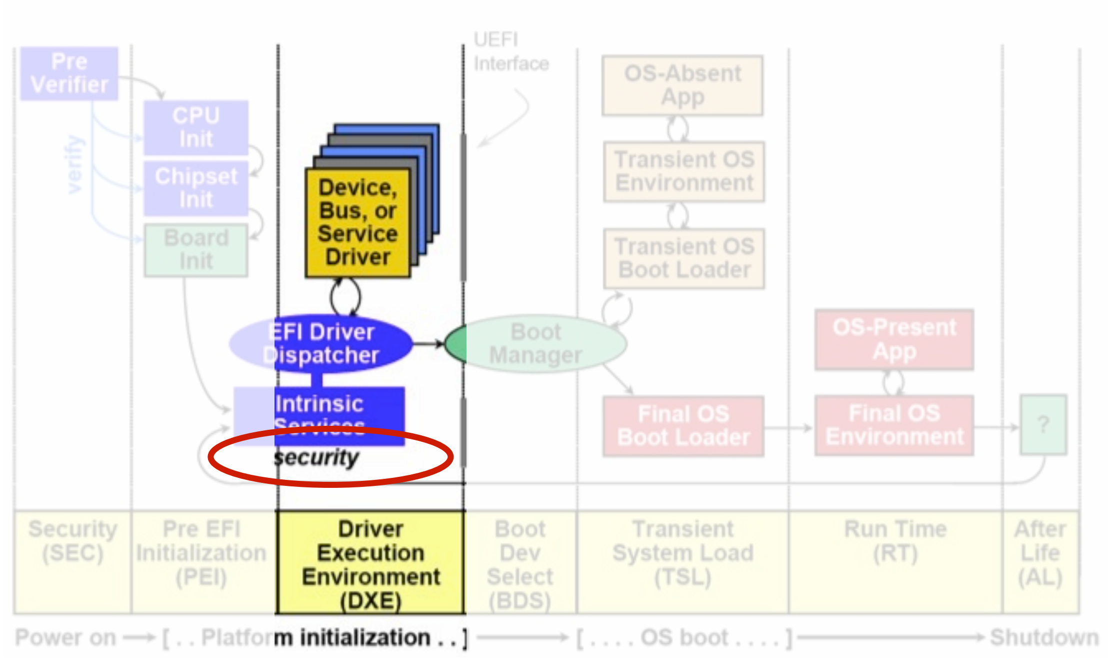
Exit conditions for handoff to DXE

- 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)

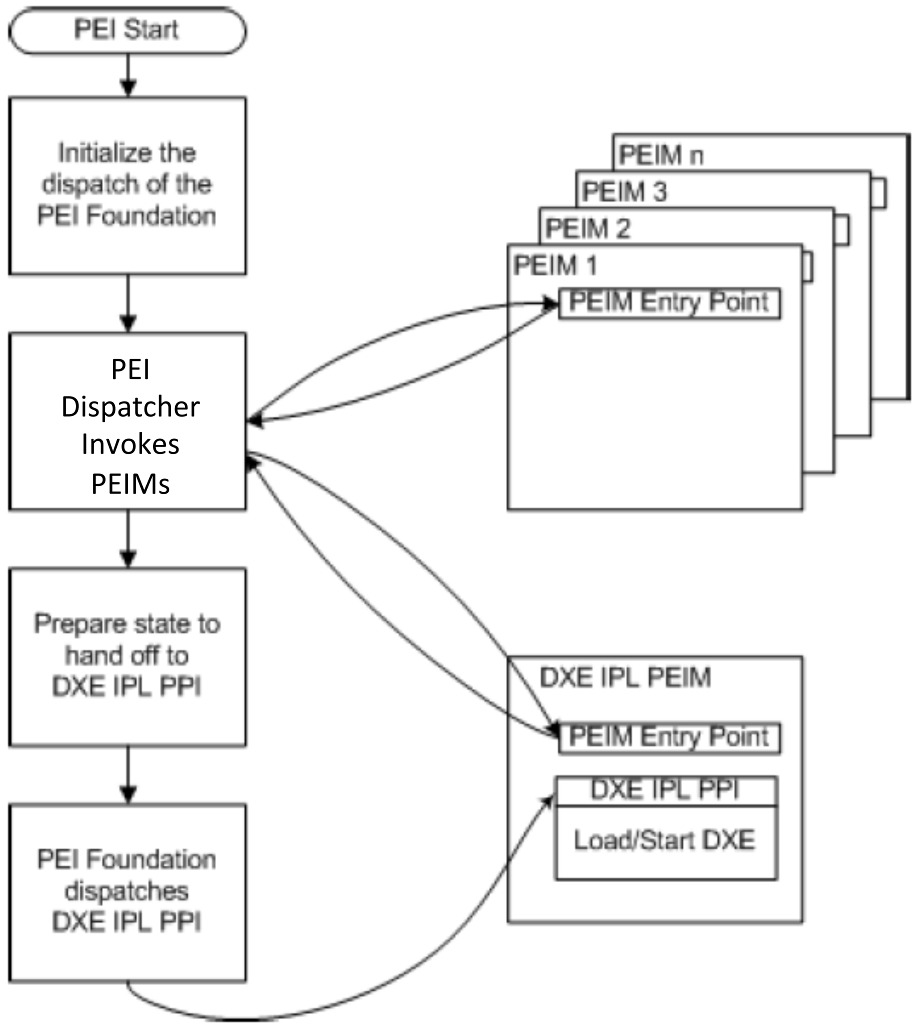


- The DXE phase is designed to be executed at a high-enough level where it is independent from architectural requirements
- Similar to PEI from a high-level; it creates services used by DXE, has a dispatcher that finds and loads DXE drivers, etc.
- System Management Mode set up, Secure Boot enforcement 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


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 make the following substitutions
- 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

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

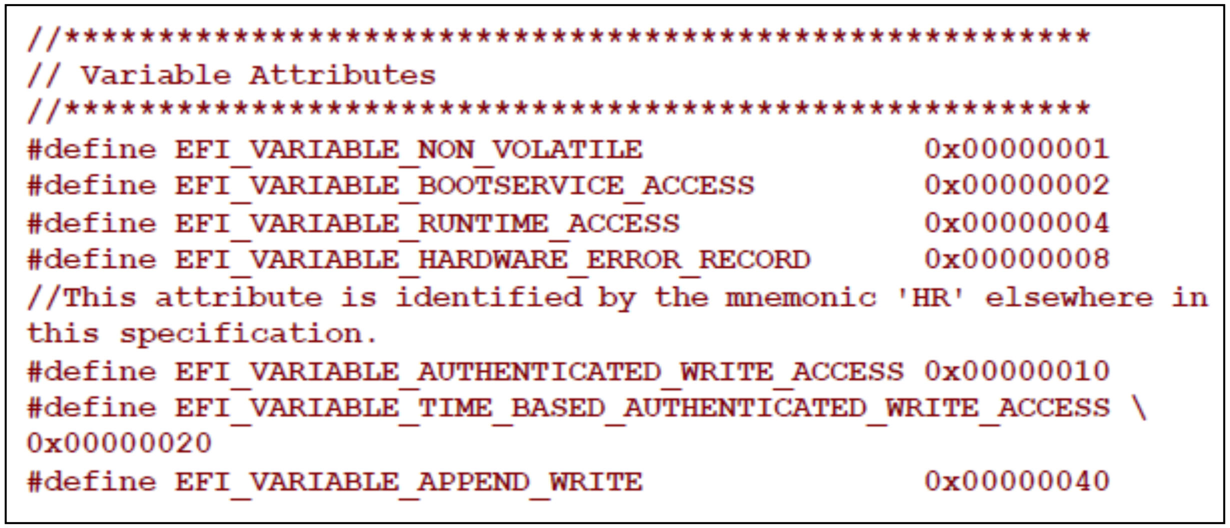
- (3/2011) Lenovo X220: 65 PEIMs, 278 DXE drivers
- (1/2014) Lenovo X240: 80 PEIMs, 352 DXE drivers
- (3/2010) HP Elitebook 2540p: 42 PEIMs, 164 DXE drivers
- (1/2014) HP Elitebook 850 G1: 117 PEIMs, 392 DXE drivers
- (11/2010) Dell Latitude E6410: 32 PEIMs, 315 DXE drivers
- (2/2014) Dell Latitude E6440: 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…

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
- Can 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

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
```

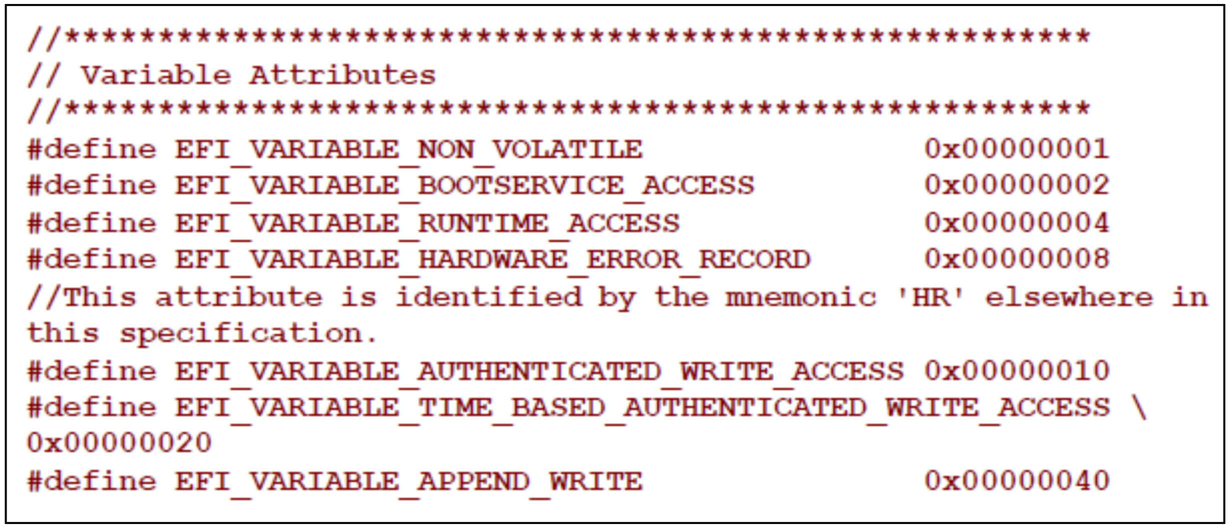
- Each UEFI variable has attributes that determine how the firmware stores and maintains the data:
- 'Non_Volatile'
  - The variable is stored on flash
- 'Bootservice_Access'
  - Can be accessed/modified during boot. Must be set in order for Runtime_Access to also be set

* UEFI 2.3.1 Errata C Final

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 not as Authenticated, the variable can be modified by any application
  - The Setup variable (of VU#758382 fame) is marked like this
  - Goto "SetupForFailure" slides

Looking at NVARs w/ ChipSec

- Copy Chipsec_Install to C:\Chipsec_Install
- (If you don't already have python 2.7 installed) run the Python 2.7.9 installer
  - NOTE: on the "Customize Python 2.7.9" page, make sure you select the "Add python.exe to Path" which is just barely visible at the bottom (you need to scroll down)
- Run the pywin32 installer
- From admin cmd prompt:

```
Bcdedit /set TESTSIGNING ON
shutdown /r /t 0
```

Looking at NVARs w/ ChipSec

– From admin cmd prompt

cd C:\ChipSec_Install\chipsec-master1-30-2015\chipsec-master \source\tool

python chipsec_main.py

    (will do the basic vulnerability checks)

python chipsec_util.py uefi nvram nvar C:\Copernicus_BIOS.bin

Or possibly

python chipsec_util.py uefi nvram vss C:\Copernicus_BIOS.bin

    Will parse out UEFI nvars and place them in the folder

    C:\Copernicus_BIOS.bin.nvram.dir

"Authenticate how?" Keys and Key Stores

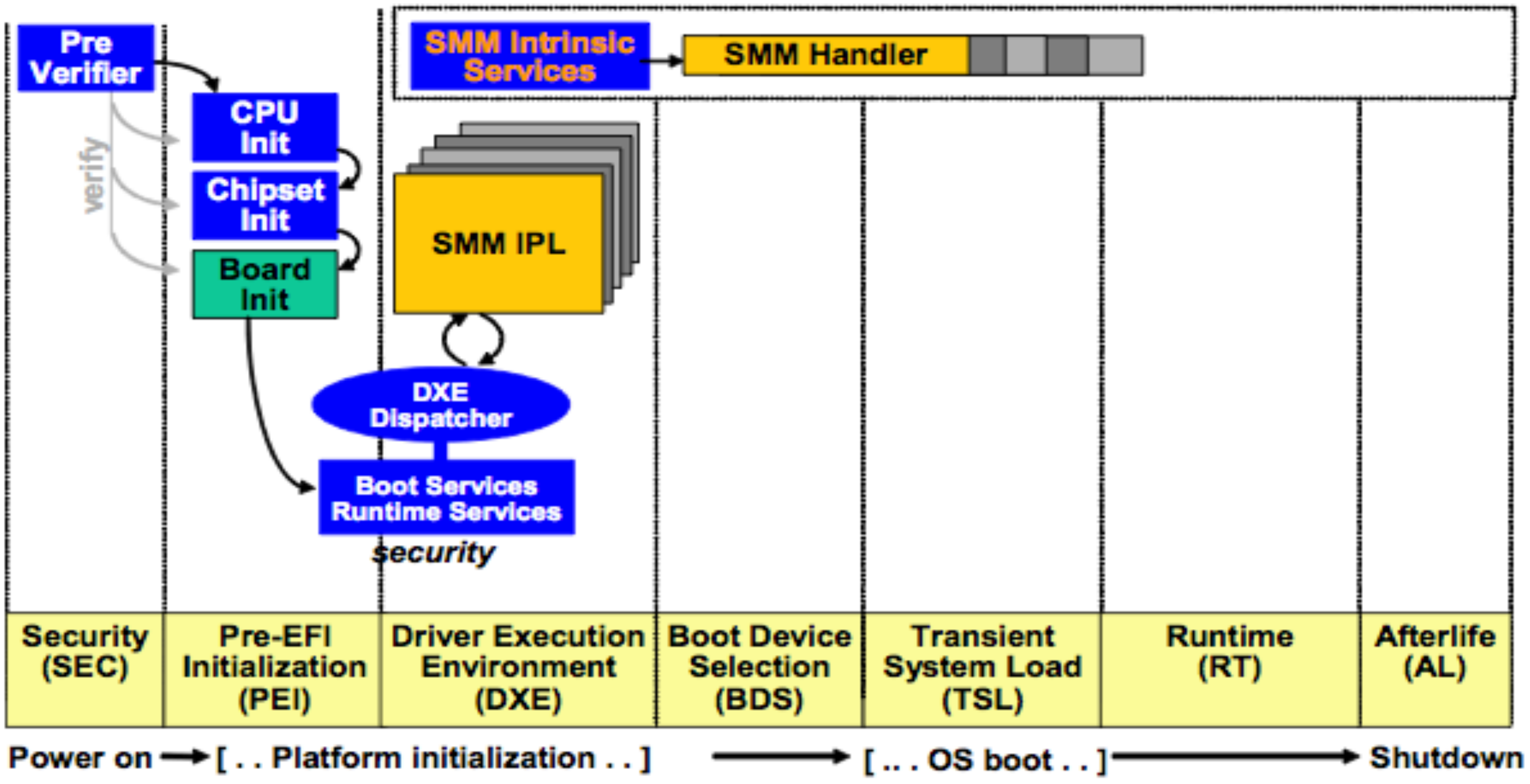
- UEFI implements 4 variables which store keys, signatures, and/or hashes:
- Platform Key (PK)
  - "The platform key establishes a trust relationship between the platform owner and the platform firmware." - spec
  - Controls access to itself and the KEK variables
  - Only a physically present user or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

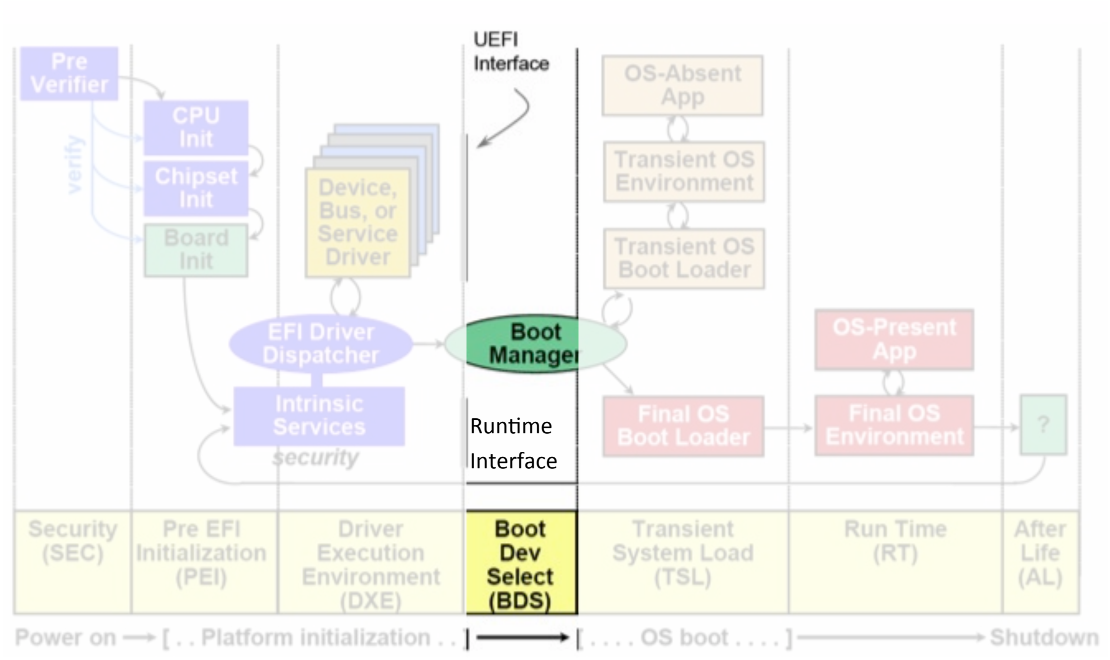
- As stated earlier, these variables are stored on the Flash file system
- Thus, if the SPI flash isn't locked down properly, these keys/ hashes can be overwritten by an attacker
- The problem is, the UEFI variables must rely solely on SMM to protect them!
- The secondary line of defense, the Protected 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!


Pre Verifier
verify
CPU Init
Chipset Init
Board Init
SMM Intrinsic Services
SMM Handler
SMM IPL
DXE Dispatcher
Boot Services Runtime Services
security
Security (SEC)
Pre-EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TSL)
Runtime (RT)
Afterlife (AL)
Power on → [. . Platform initialization . .] → [. . . OS boot . .] → Shutdown

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

Boot Device Selection (BDS)

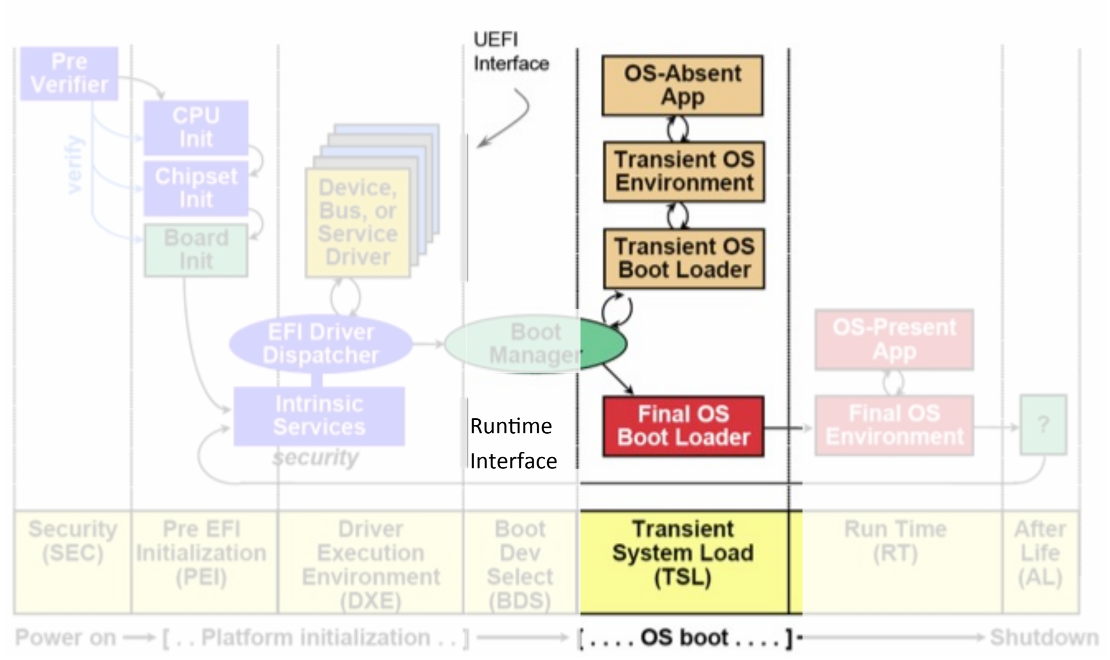


- The BDS will typically be encapsulated into a single file loaded by the DXE phase.
- It consults the configuration information to decide whether you're going to boot an OS or "something else"
- It has access to the full UEFI Boot Services Table of services that DXE set up. E.g. HD filesystem access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

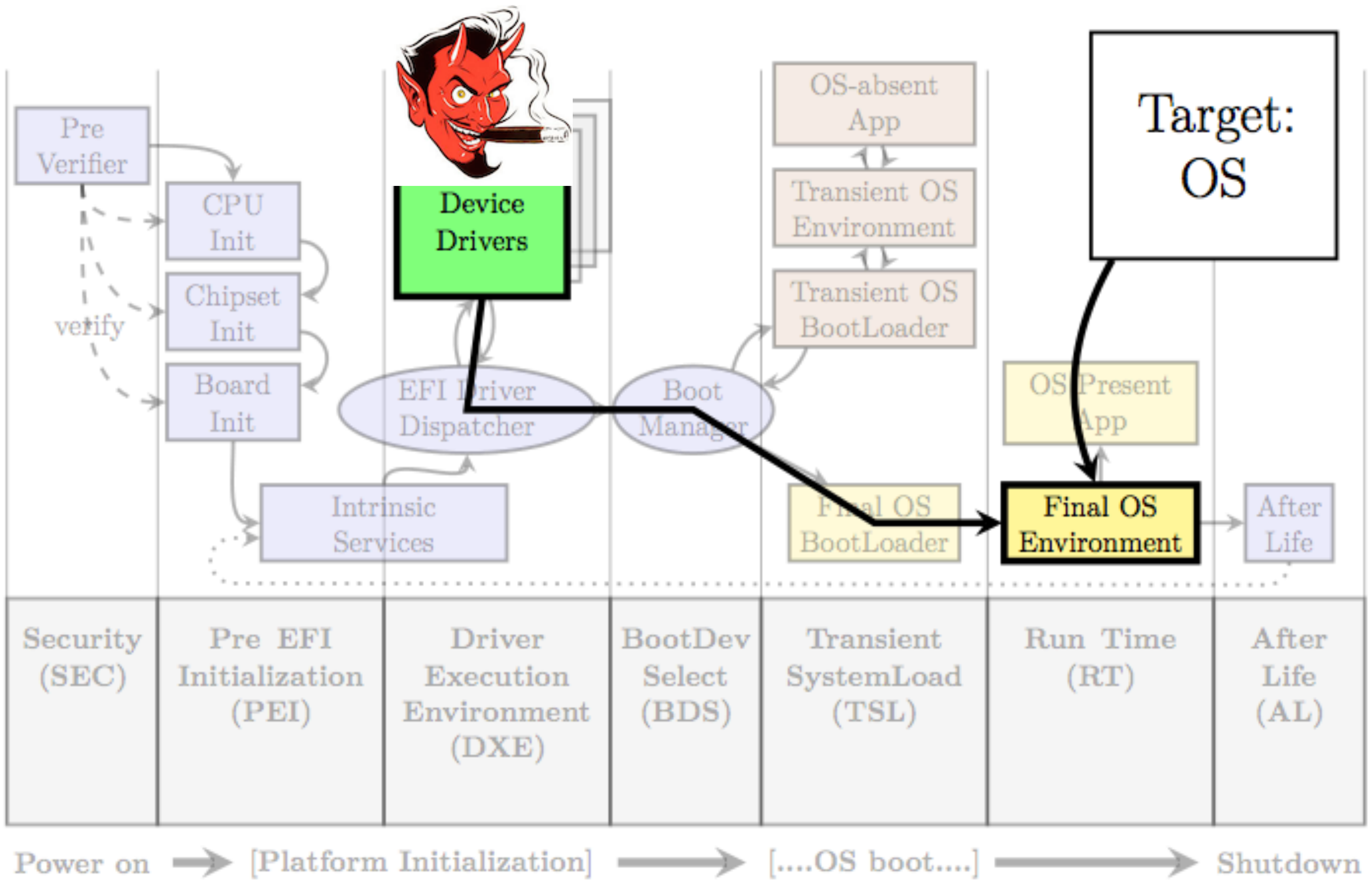
Transient System Load (TSL)



- 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]

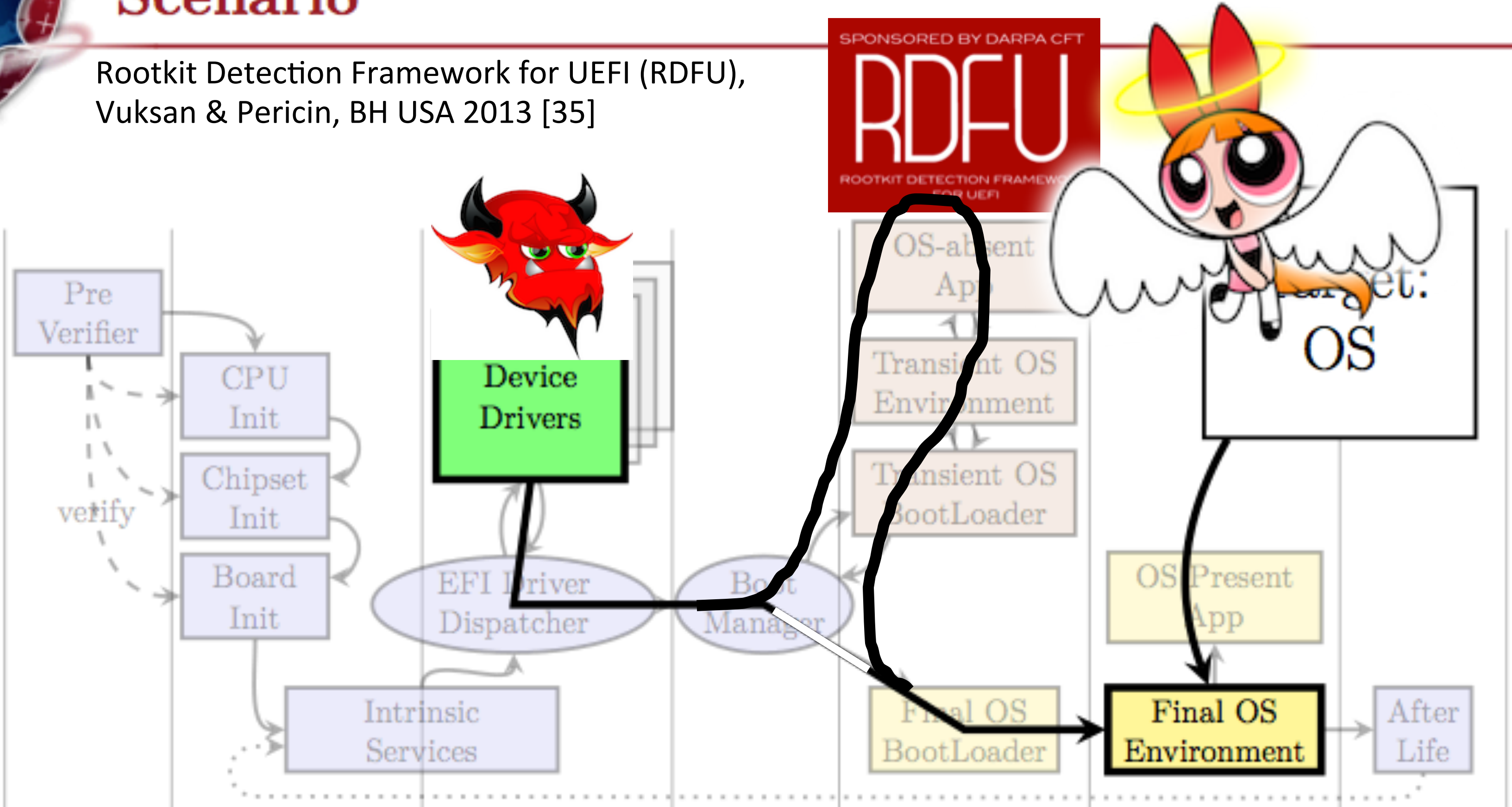
Scenario



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

Scenario

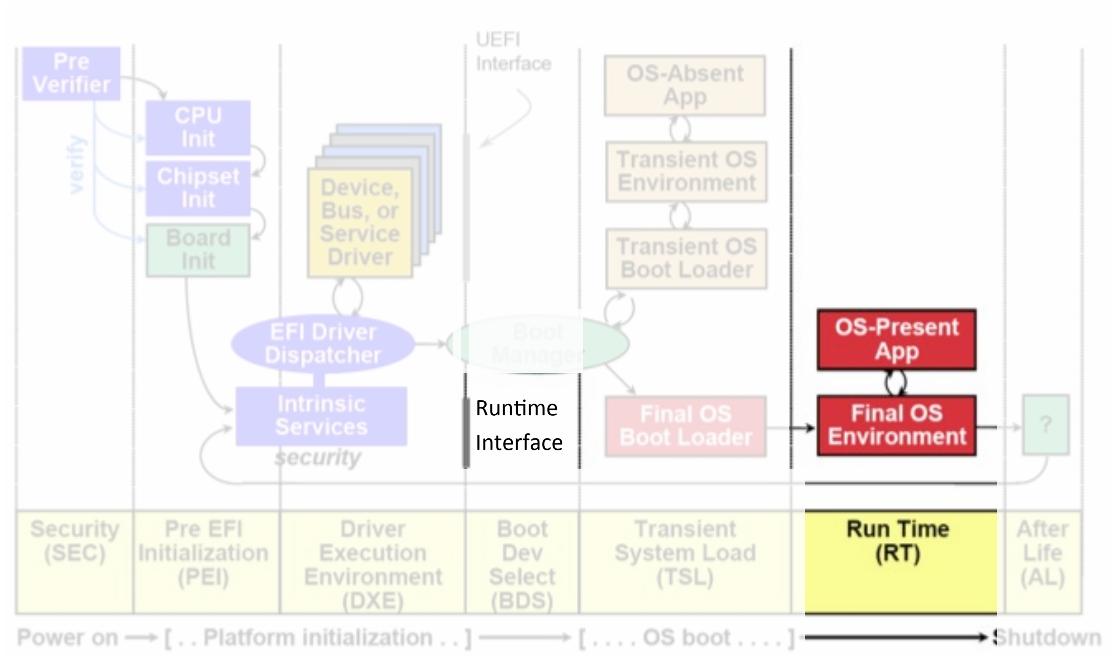
Rootkit Detection Framework for UEFI (RDFU), Vuksan & Pericin, BH USA 2013 [35]



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

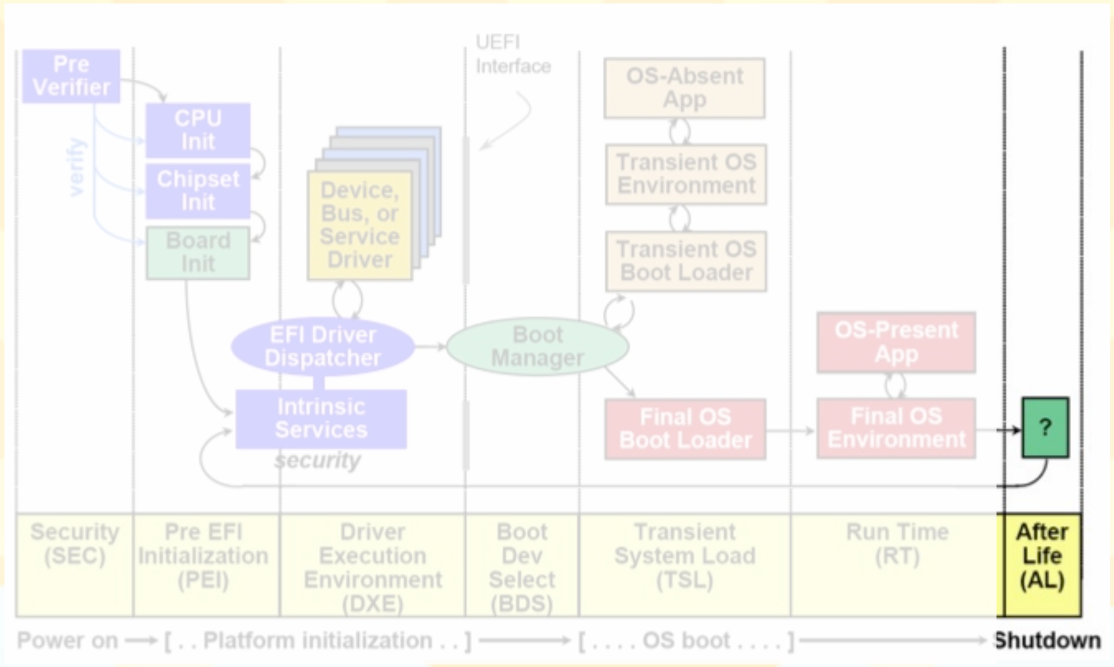
Power on → [Platform Initialization] → [....OS boot....] → Shutdown

Run Time (RT)



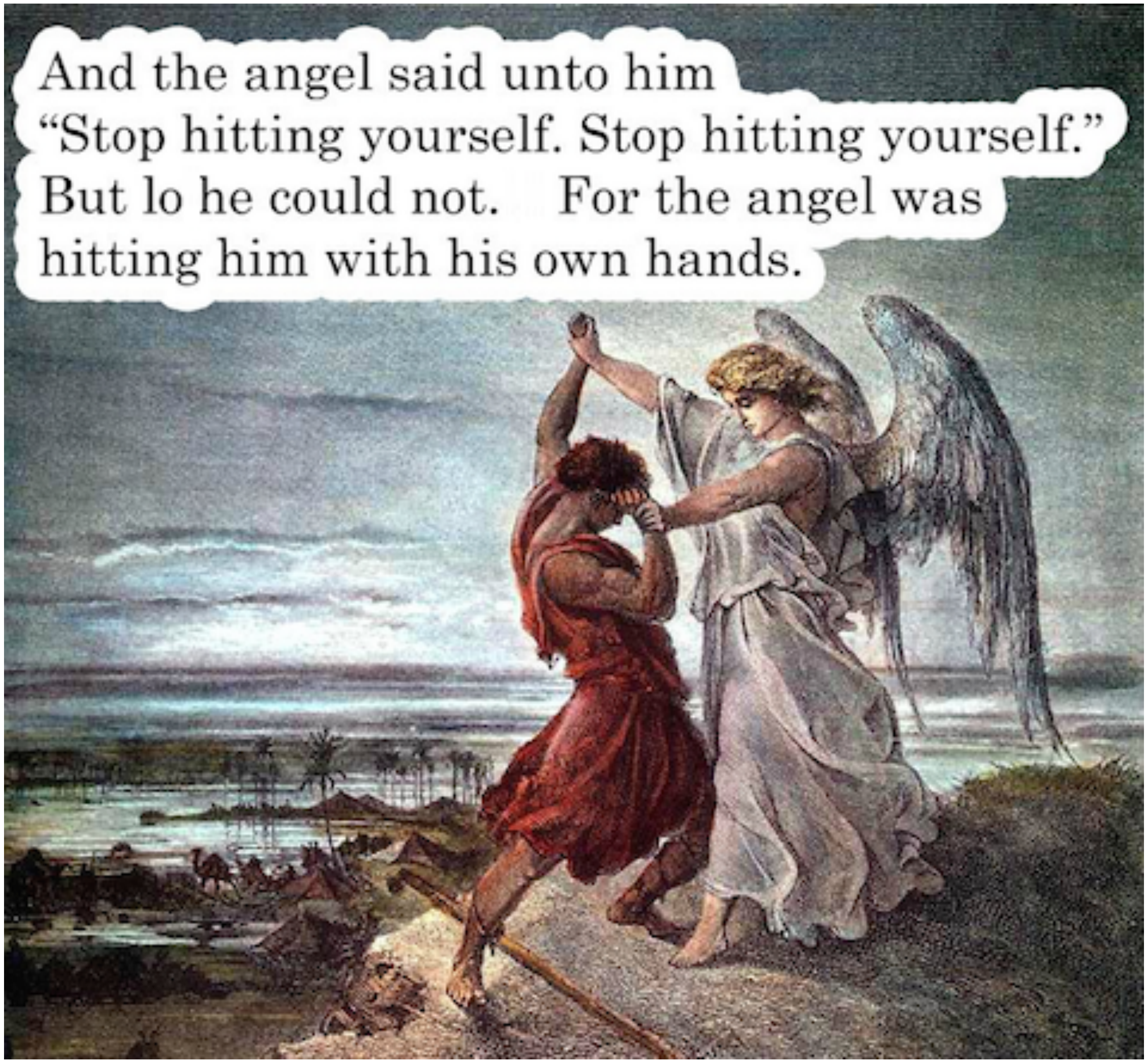
- Typically when the OS boot loader is done, it will call ExitBootServices() in the UEFI Boot Services table. This will reclaim the majority of UEFI memory so the OS can use it
- However some memory is retained, to be used for the Runtime Services Table talked about a while ago

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?)

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.