



UEFI & EDK II TRAINING

UEFI PLATFORM FIRMWARE SECURITY

tianocore.org



LESSON OBJECTIVE

- ★ Why is platform firmware Security important
- ★ UEFI boot flow with the threat model
- ★ Security technologies overview
- ★ Tools and resources on how to test firmware for security

WHY?

Why is Platform Firmware Security Important ?

Why is Platform Firmware Security Important ?



Why is Platform Firmware Security Important ?

Firmware is the Infrastructure



Initialize
hardware

Establish
root-of-trust

Hand-off
to the OS

Why is Platform Firmware Security Important ?

Firmware is the Infrastructure



Initialize
hardware

Establish
root-of-trust

Hand-off
to the OS

Why is Platform Firmware Security Important ?

Firmware is the Infrastructure



Initialize
hardware

Establish
root-of-trust

Hand-off
to the OS

Why??? Security

Without

With

Without**With**

Possible corrupted or destroyed data

- BootKit virus – MBR Rootkits
- Network boot attacks e.g. PXESPOILT
- Code Injection Attacks

Without**With**

Possible corrupted or destroyed data

- BootKit virus – MBR Rootkits
- Network boot attacks e.g. PXESPOILT
- Code Injection Attacks



Without

Possible corrupted or destroyed data

- BootKit virus – MBR Rootkits
- Network boot attacks e.g. PXESPOILT
- Code Injection Attacks

With

- Data integrity
- Trusted boot to OS
- Trusted drivers
- Trusted Applications



Without

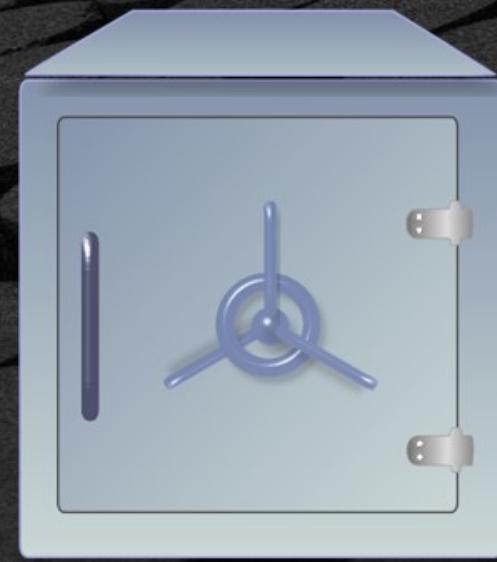
Possible corrupted or destroyed data

- BootKit virus – MBR Rootkits
- Network boot attacks e.g. PXESPOILT
- Code Injection Attacks

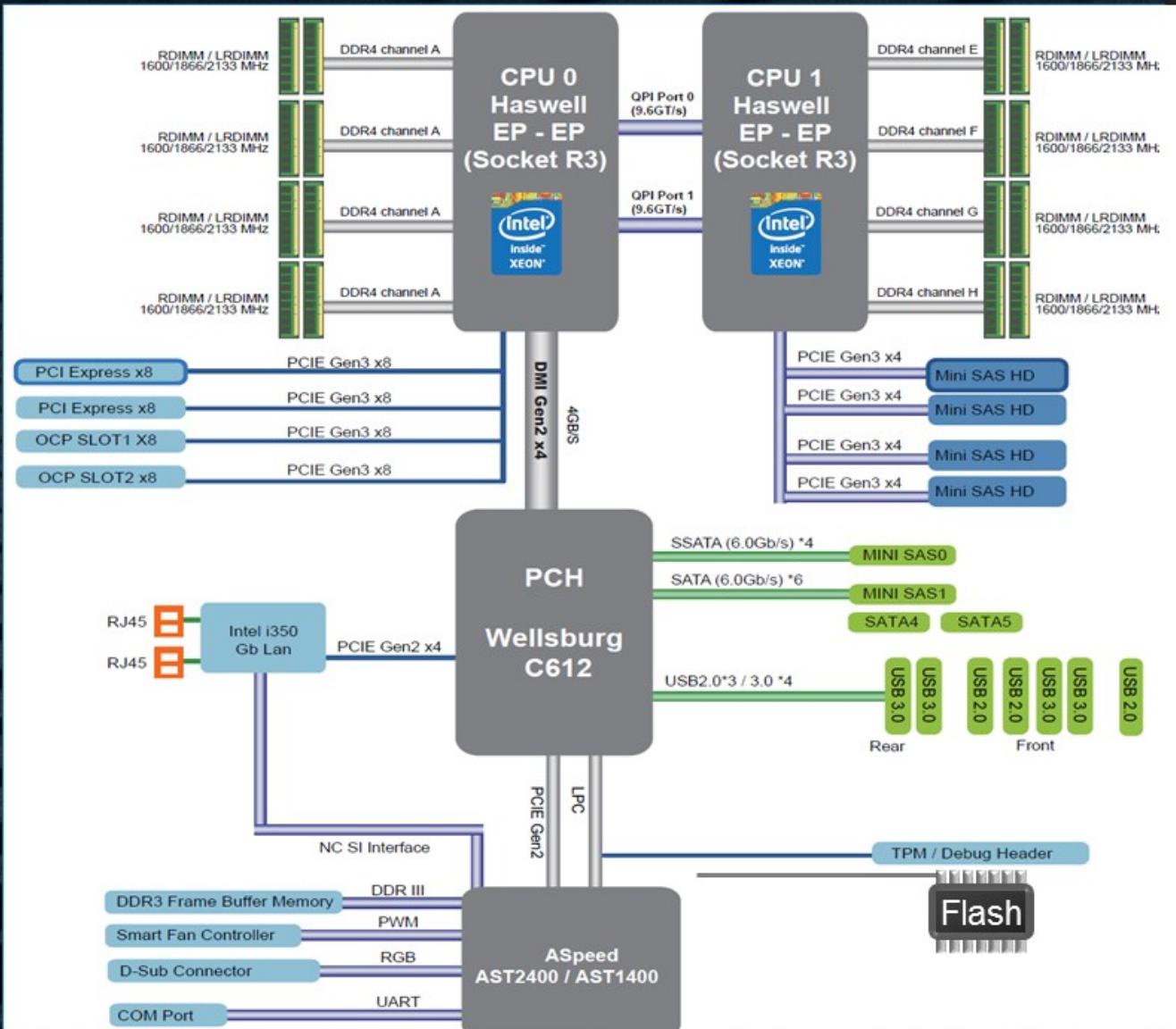


With

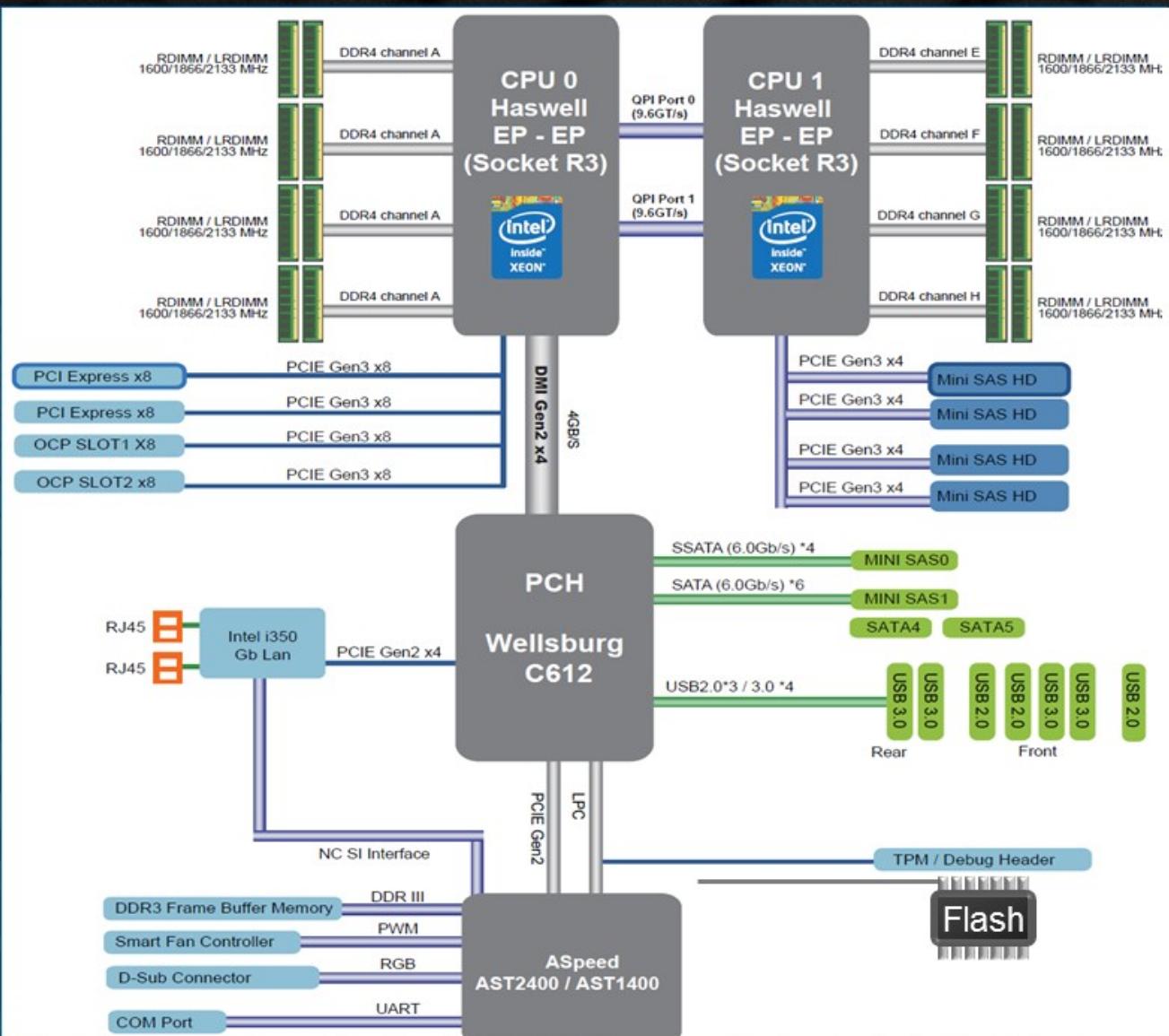
- Data integrity
- Trusted boot to OS
- Trusted drivers
- Trusted Applications



Confidentiality
Integrity
Availability

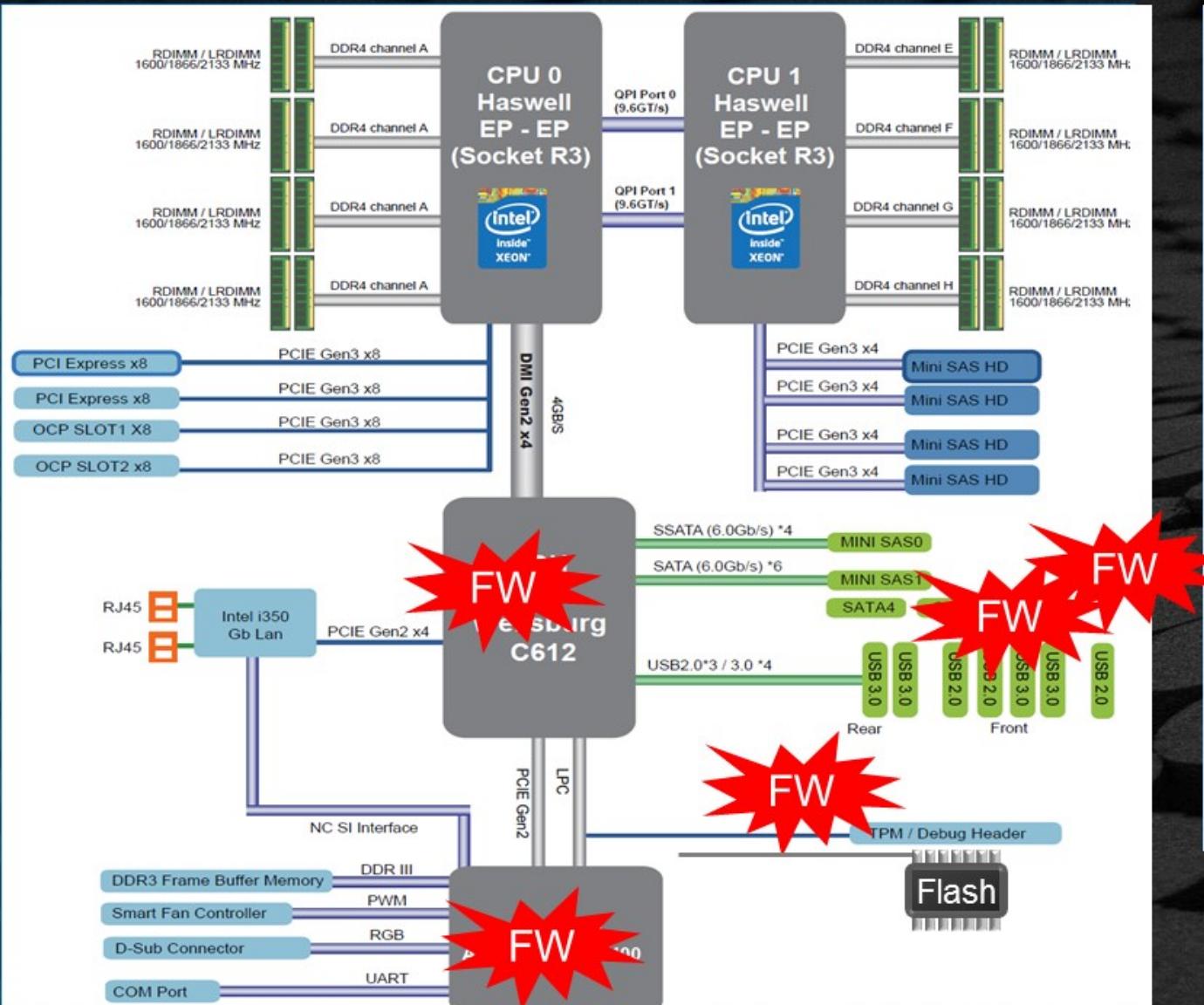


Firmware Everywhere



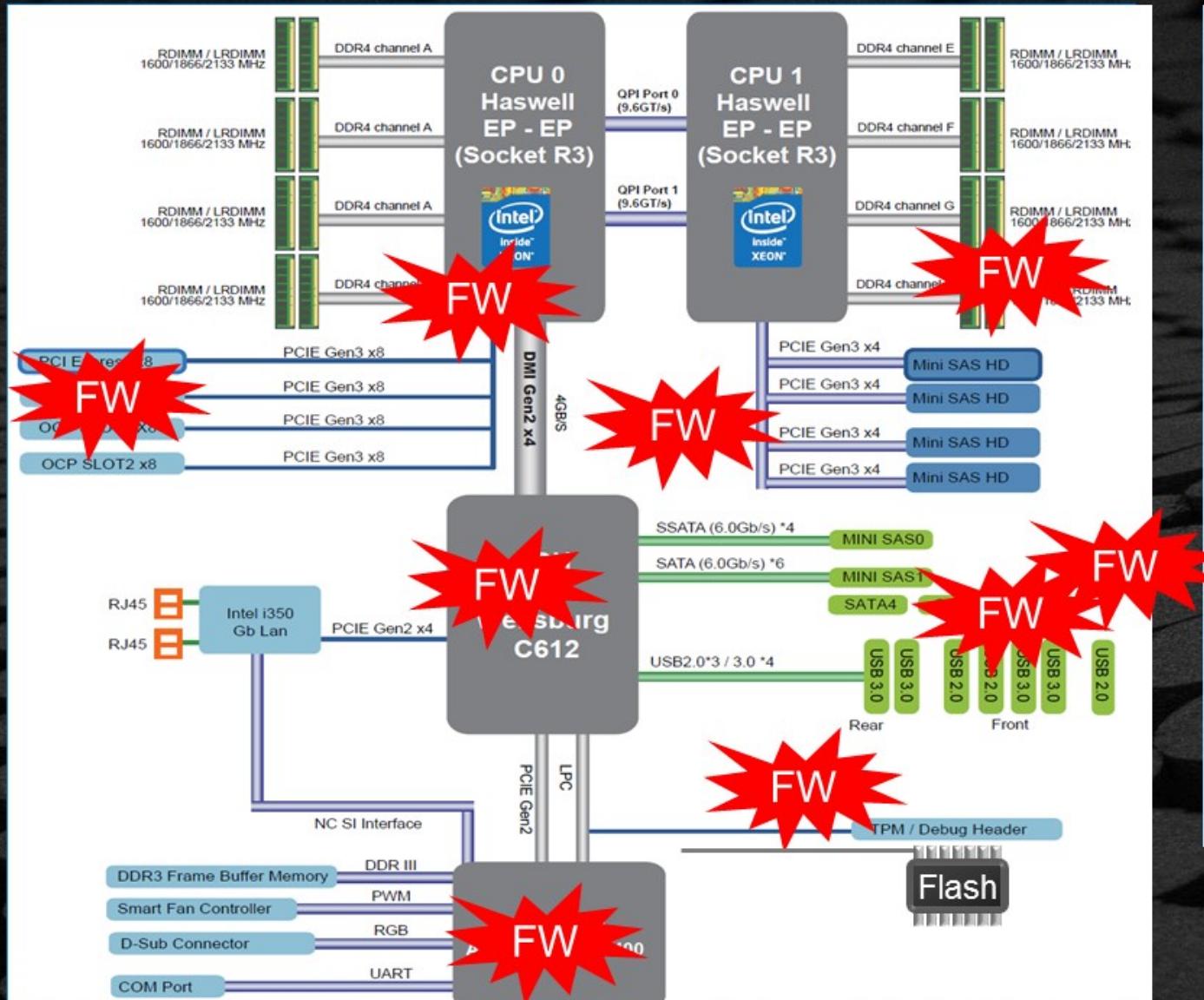
- GBe NIC, WiFi, Bluetooth, WiGig
- Baseband (3G, LTE) Modems
- Sensor Hubs
- NFC, GPS Controllers
- HDD/SSD
- Keyboard and Embedded Controllers
- Battery Gauge
- Baseboard Management Controllers (BMC)
- Graphics/Video
- USB Thumb Drives, keyboards/mice
- Chargers, adapters
- TPM, security coprocessors
- Routers, network appliances

Firmware Everywhere



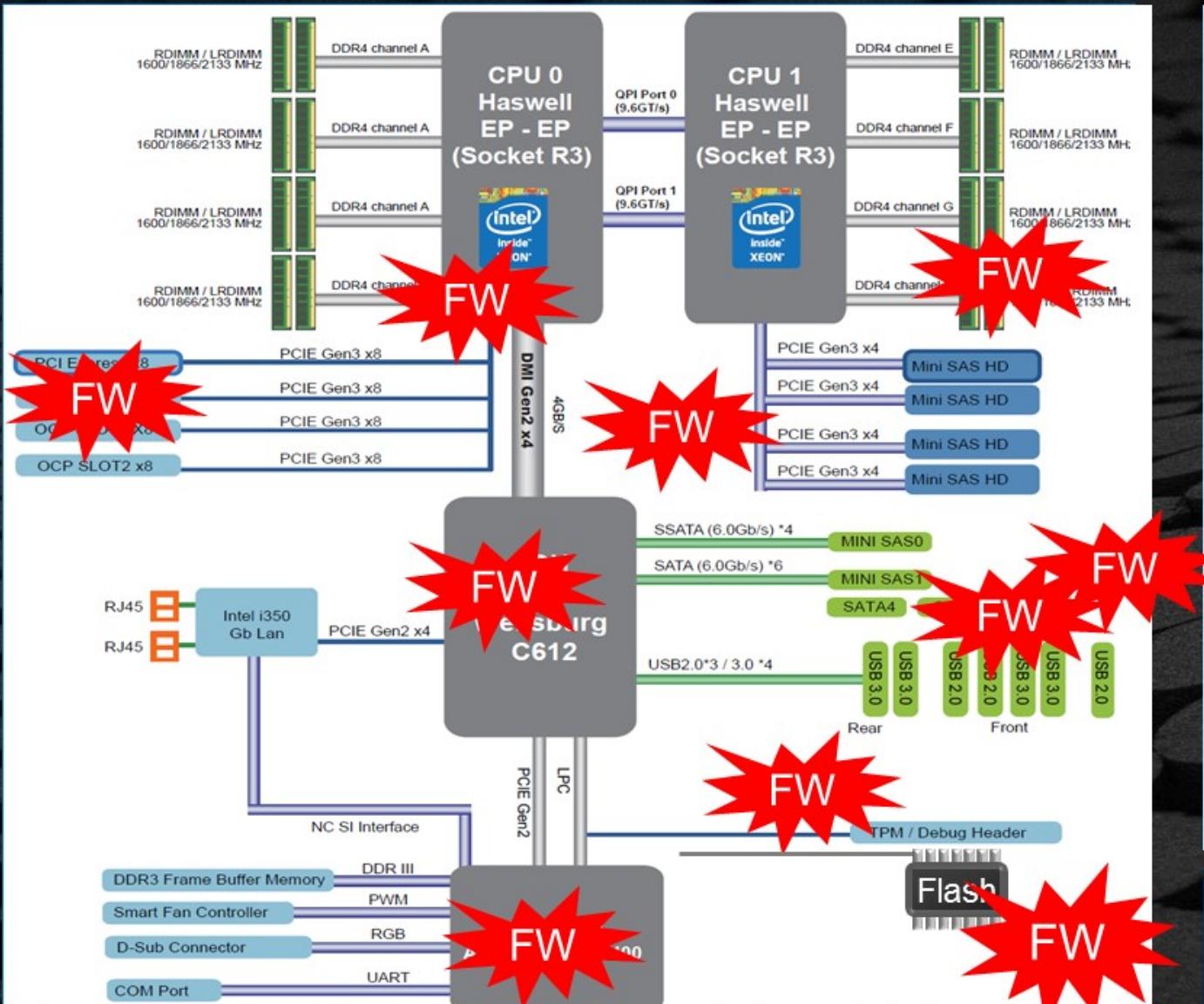
- GBe NIC, WiFi, Bluetooth, WiGig
- Baseband (3G, LTE) Modems
- Sensor Hubs
- NFC, GPS Controllers
- HDD/SSD
- Keyboard and Embedded Controllers
- Battery Gauge
- Baseboard Management Controllers (BMC)
- Graphics/Video
- USB Thumb Drives, keyboards/mice
- Chargers, adapters
- TPM, security coprocessors
- Routers, network appliances

Firmware Everywhere



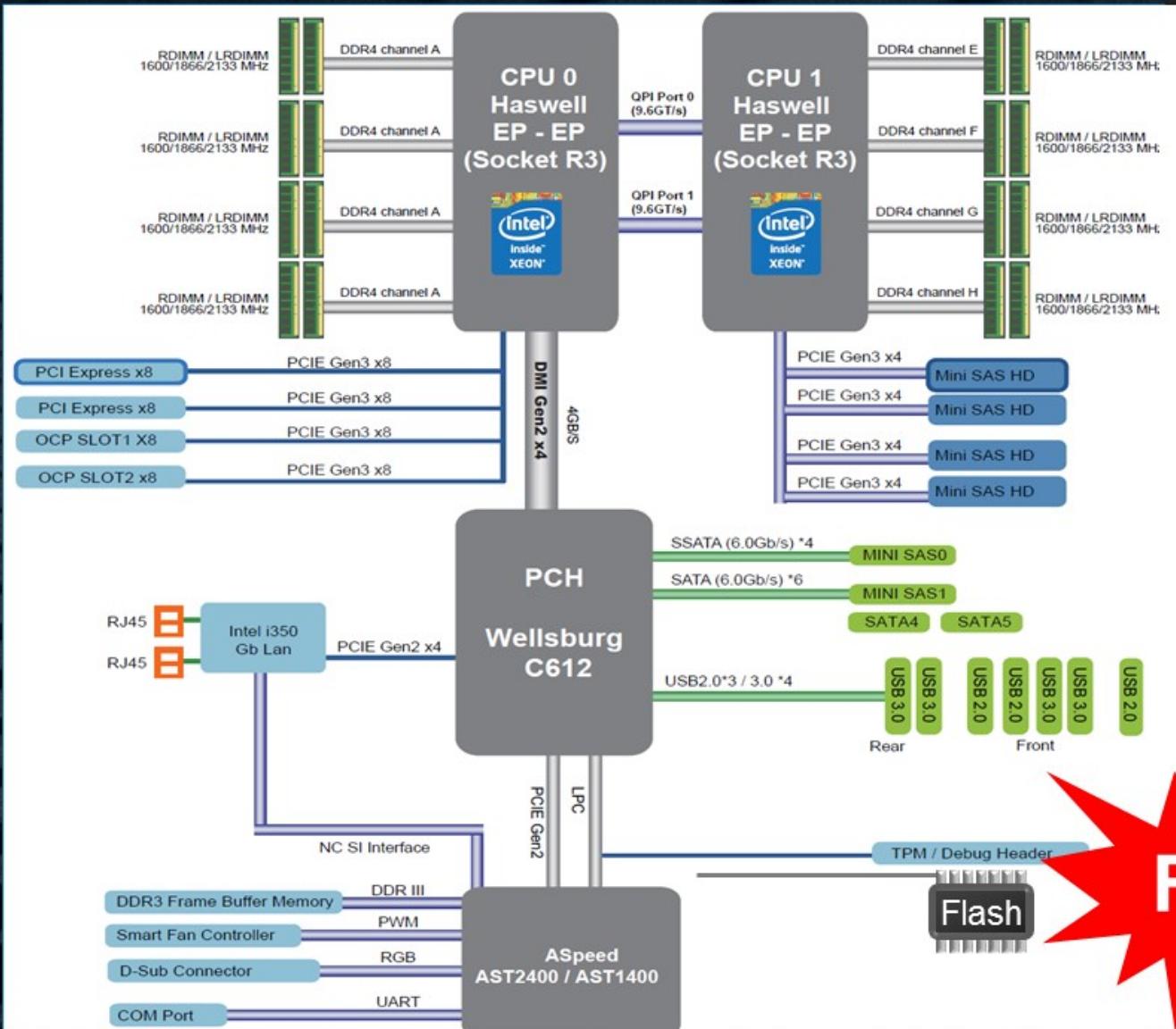
- GBe NIC, WiFi, Bluetooth, WiGig
- Baseband (3G, LTE) Modems
- Sensor Hubs
- NFC, GPS Controllers
- HDD/SSD
- Keyboard and Embedded Controllers
- Battery Gauge
- Baseboard Management Controllers (BMC)
- Graphics/Video
- USB Thumb Drives, keyboards/mice
- Chargers, adapters
- TPM, security coprocessors
- Routers, network appliances

Firmware Everywhere



- GBe NIC, WiFi, Bluetooth, WiGig
- Baseband (3G, LTE) Modems
- Sensor Hubs
- NFC, GPS Controllers
- HDD/SSD
- Keyboard and Embedded Controllers
- Battery Gauge
- Baseboard Management Controllers (BMC)
- Graphics/Video
- USB Thumb Drives, keyboards/mice
- Chargers, adapters
- TPM, security coprocessors
- Routers, network appliances

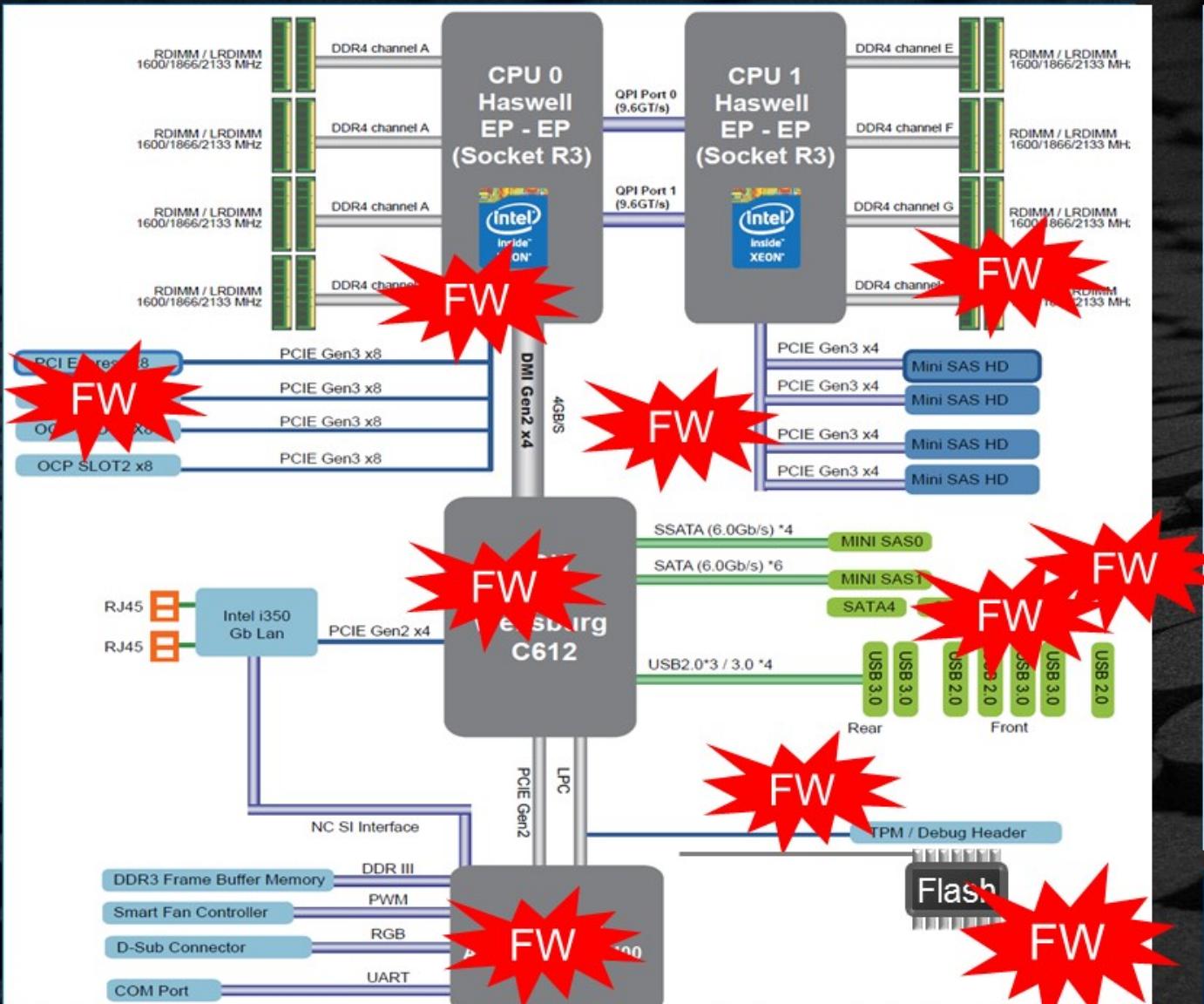
Main system firmware (BIOS, UEFI firmware, coreboot)



Main system firmware (BIOS,
UEFI firmware, coreboot)

FW

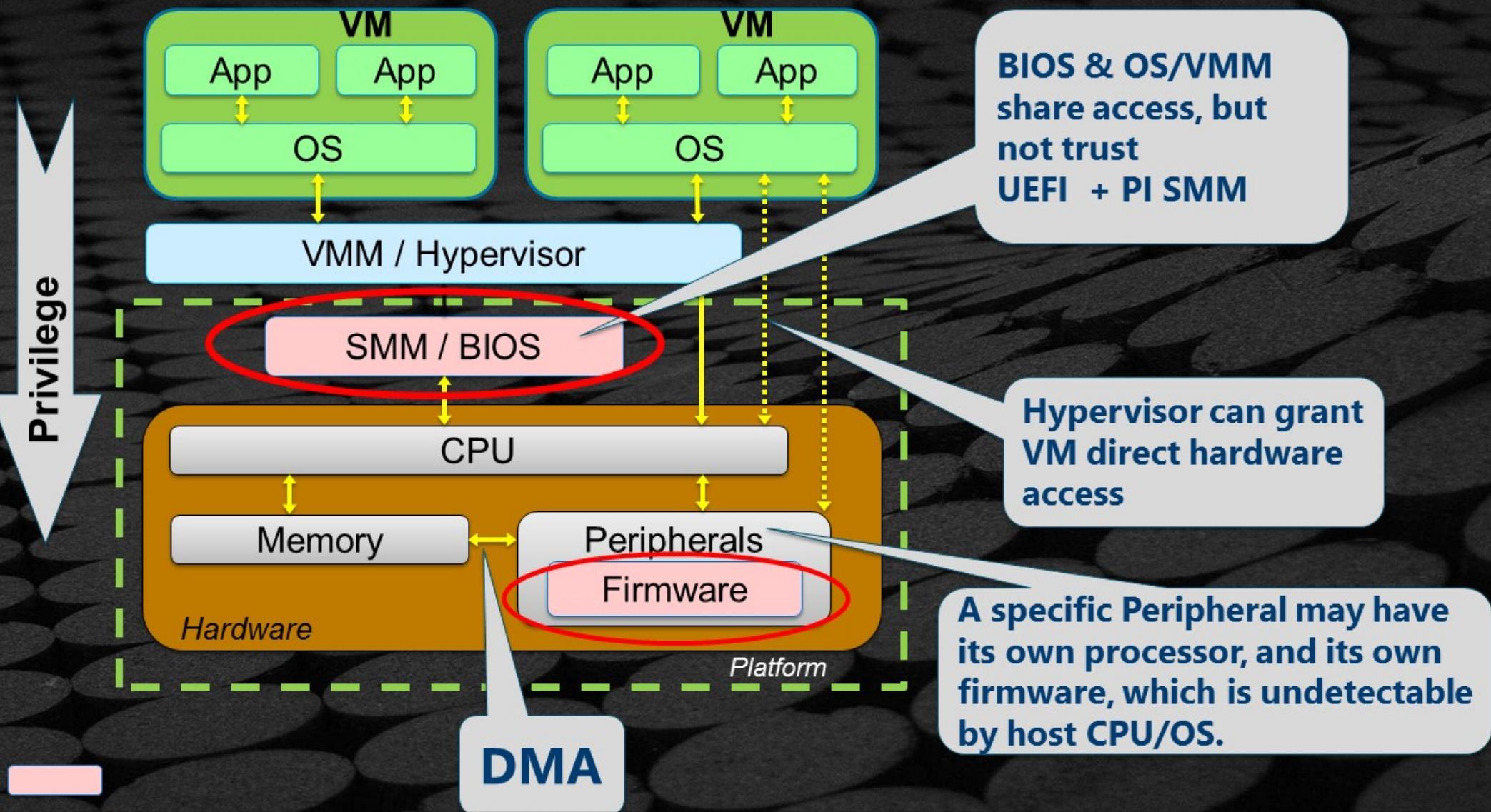
Firmware Everywhere



- GBe NIC, WiFi, Bluetooth, WiGig
- Baseband (3G, LTE) Modems
- Sensor Hubs
- NFC, GPS Controllers
- HDD/SSD
- Keyboard and Embedded Controllers
- Battery Gauge
- Baseboard Management Controllers (BMC)
- Graphics/Video
- USB Thumb Drives, keyboards/mice
- Chargers, adapters
- TPM, security coprocessors
- Routers, network appliances

Main system firmware (BIOS, UEFI firmware, coreboot)

Where is UEFI firmware?

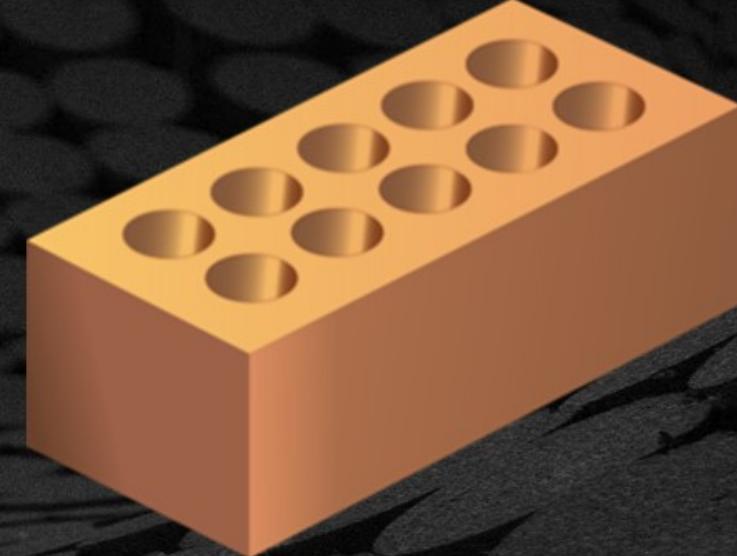


In-the-wild Firmware Attacks

- Legacy Bootkits ([TDL4](#) , [Gapz](#)...)
- [Mebromi BIOS rootkit](#)
- [Stuxnet](#)
- [EQUATION Group](#) HDD firmware malware
- [Hacking Team \[UEFI rootkit\]](#)
- [Petya](#) MBR Ransomware
- Legitimate BIOS “backdoors”: SuperFish, Computrace



Why Attack Firmware?



Extreme persistence

Unfettered access to hardware

Stealth

OS independence

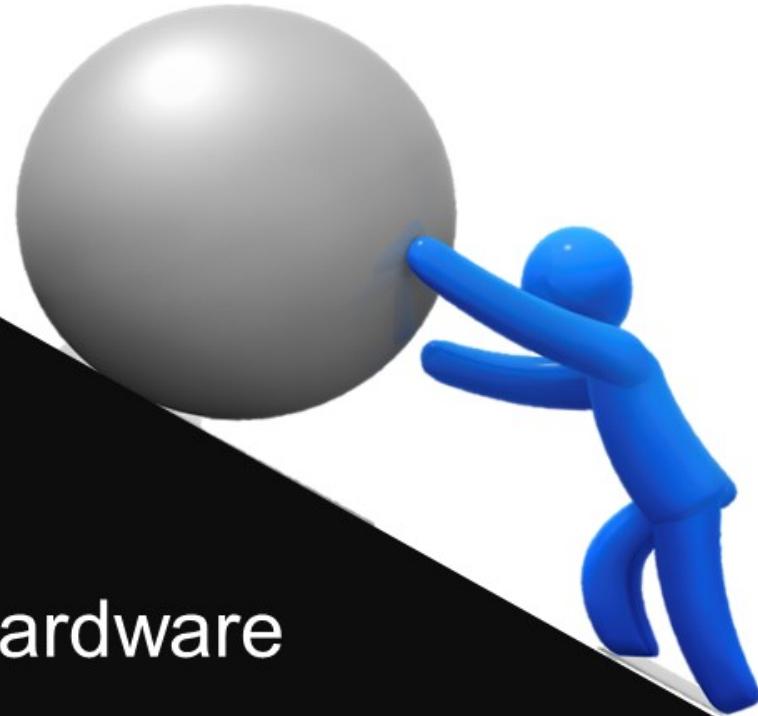
Bypass software
(OS or VMM)
based security

Making the system
unbootable
(bricking)

Firmware Attack
Methods

EXTREME PERSISTENCE

- System firmware rootkit
(in SMM or BIOS/UEFI)
- Replaces OS boot
loader every boot
- Which patches OS kernel
- Firmware rootkit is protected by the hardware
write protections
- Only way to fully remove the infection is to
physically re-flash the flash “ROM” chip



STEALTH

Is Security Software Real Protection?

- Monitor of ~~All~~ firmware
- Reliably tell if infected False | Positive
- Devices use *obscure* hardware interfaces to their firmware
- Which Tool for which firmware infection?
 - Rootkit in firmware of SSD, NIC, EC, BMC, modem, USB thumb-drive, battery gauge, charger

≡



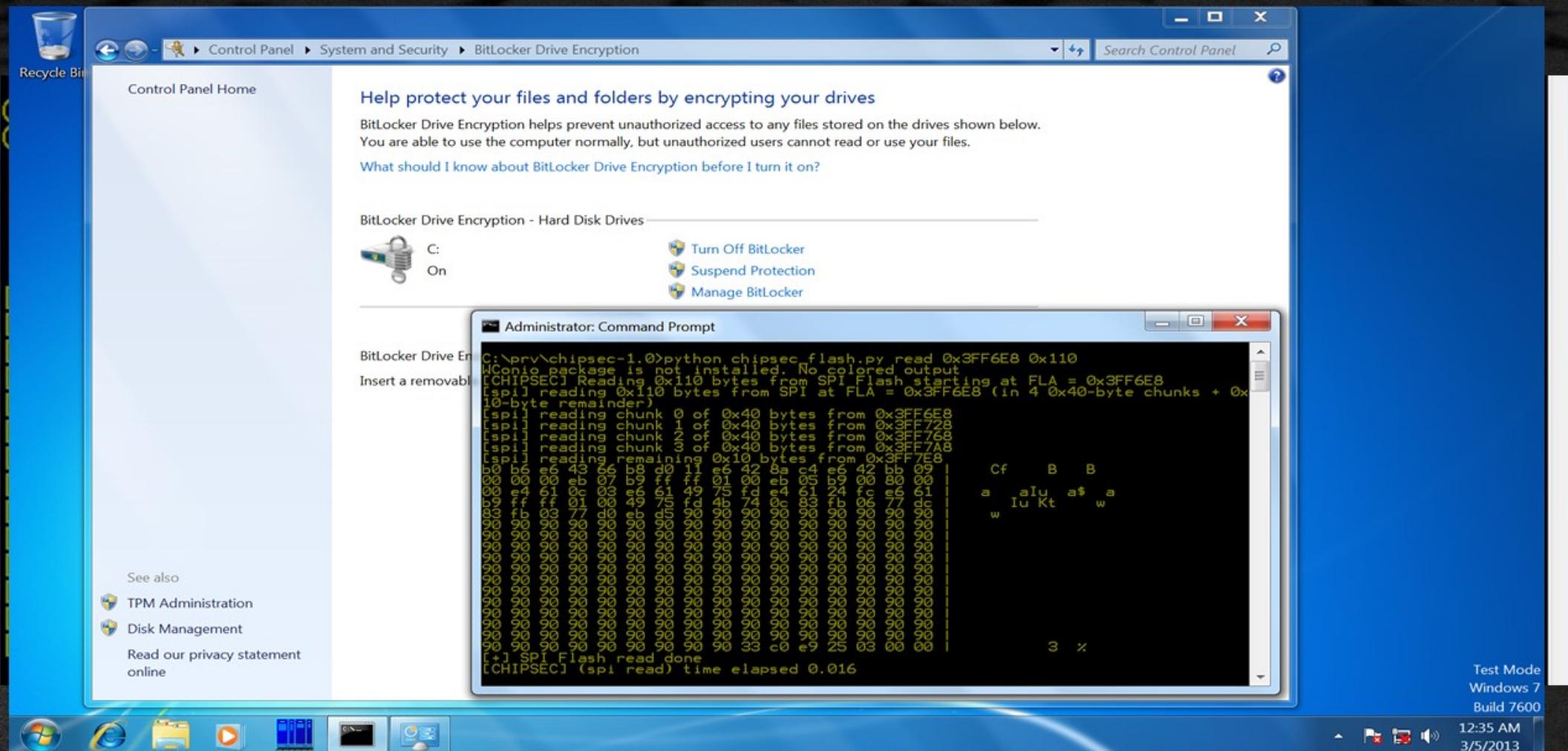
Bypass Software Security – Full Disk

```
OS      : Windows 7 6.1.7600 AMD64
Chipset:
  VID:      8086
  DID:      0100
  Name:     Sandy Bridge (SNB)
  Long Name: Sandy Bridge CPU / Cougar Point PCH

[+] loaded exploits:bios.bitlocker
[+] imported chipsec.modules.exploits.bios.bitlocker
[*] Reading 0x1000-byte block from SPI at FLA = 0x3FF000..
[+] Done reading block from SPI flash
[*] Filling with 0x109 bytes of NOP slide..
[*] Injecting 0x47 bytes of payload at offset 0x6E8 in the block..
[+] Checking protection of UEFI BIOS region in SPI flash..
[spi] UEFI BIOS write protection enabled but not locked. Disabling..
[!] UEFI BIOS write protection is disabled
[*] Erasing original 0x1000-byte SPI block at FLA = 0x3FF000..
[+] Done erasing block of SPI flash
[*] Writing SPI block with payload to FLA = 0x3FF000 (payload at 0x3FF6E8)..
[+] Done writing modified UEFI BIOS boot block in SPI flash
[!] Reboot to verify that BitLocker decrypts Windows volume
```

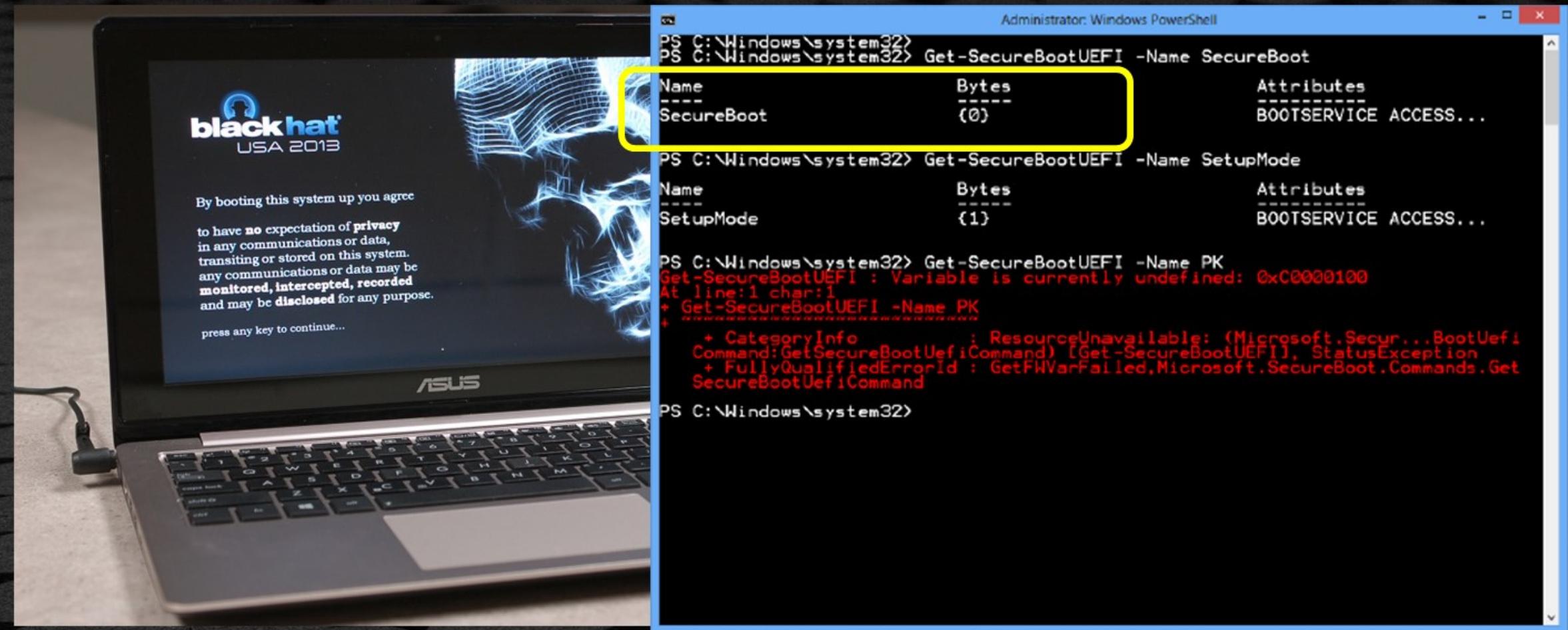
Source: Evil Maid Just Got Angrier: Why Full-Disk Encryption with TPM is not Secure on Many Systems

Bypass Software Security – Full Disk



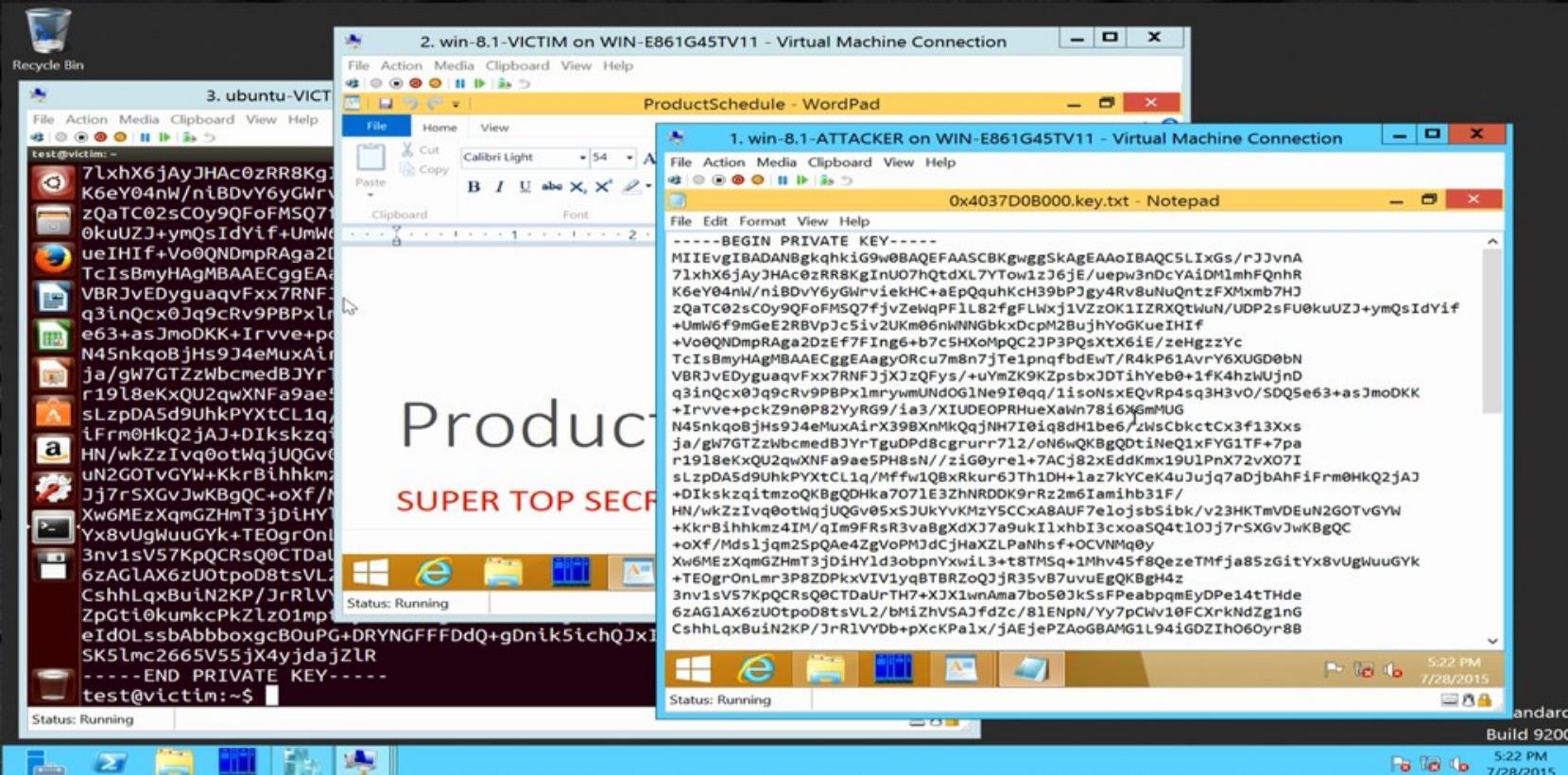
Source: [Evil Maid Just Got Angrier: Why Full-Disk Encryption with TPM is not Secure on Many Systems](#)

Bypass Software Security – UEFI Secure Boot



[Source](#): A Tale of One Software Bypass of Windows Secure Boot

Bypass Software Security – Virtual Machine Manager



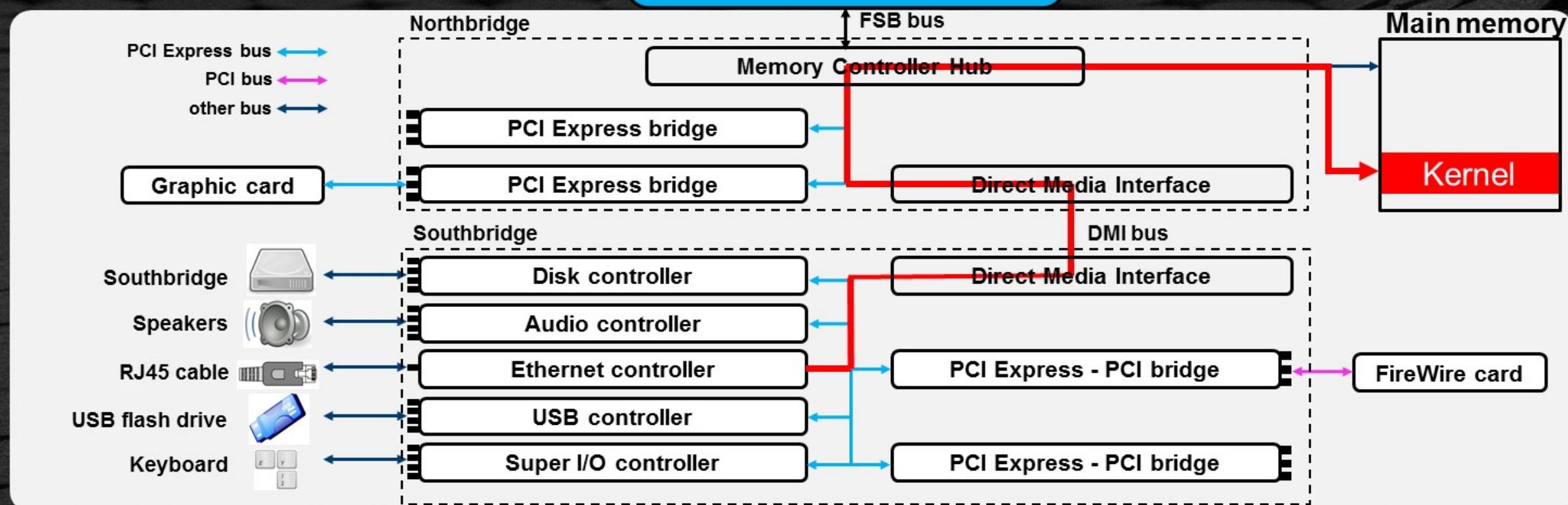
Attacker VM exposes secrets of other VMs through a backdoor opened by the rootkit

Source: Attacking Hypervisors via Firmware and Hardware

Unfettered Access to Hardware - DRAM

DMA – Based Attacks

[Source :I/O Attacks in Intel-PC Architecture ...](#)



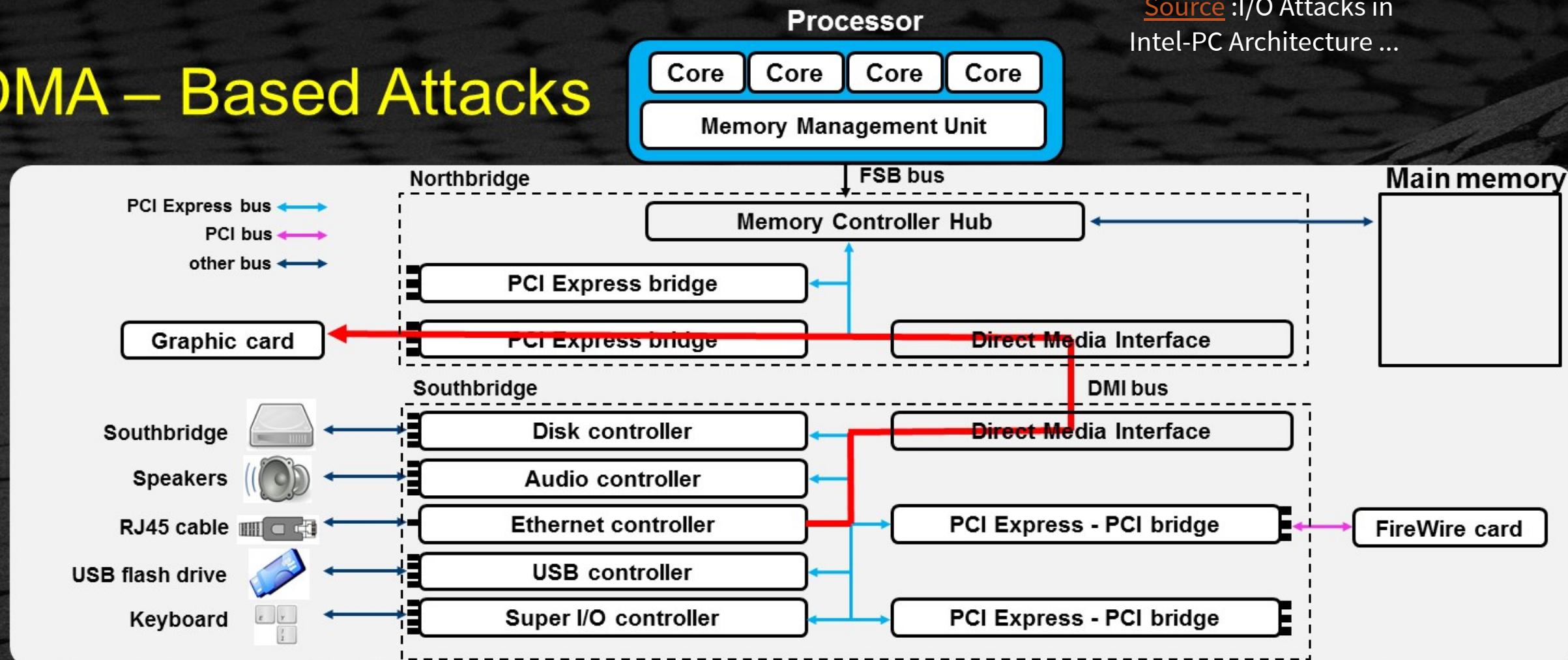
≡

DMA attacks aiming at the main memory

Unfettered Access to Hardware - DRAM

Source : I/O Attacks in Intel-PC Architecture ...

DMA – Based Attacks



≡

DMA attacks aiming at I/O controllers' internal memory

Unfettered Access to Hardware – HDD/SSD

Access to all data stored on HDD/SSD

Even when data is stored on self-encrypting drives (SED)



Cyber espionage groups utilize advanced form of malware that is **very hard to remove** once it's infected your PC.

Malware reprograms the hard drive's firmware, creating hidden sectors

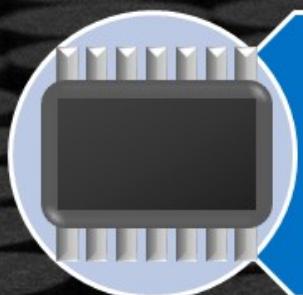
Destroying your hard drive is the only way to stop this super advanced malware

Source : PC World – Destroying your hard drive is the only way to stop this super advanced malware

Unfettered Access to Hardware – Etc.



NIC, WiFi, baseband modem firmware rootkits have direct access to network communications



EC or BMC firmware rootkit has access to platform management functions (power, thermal, NIC, keystrokes)

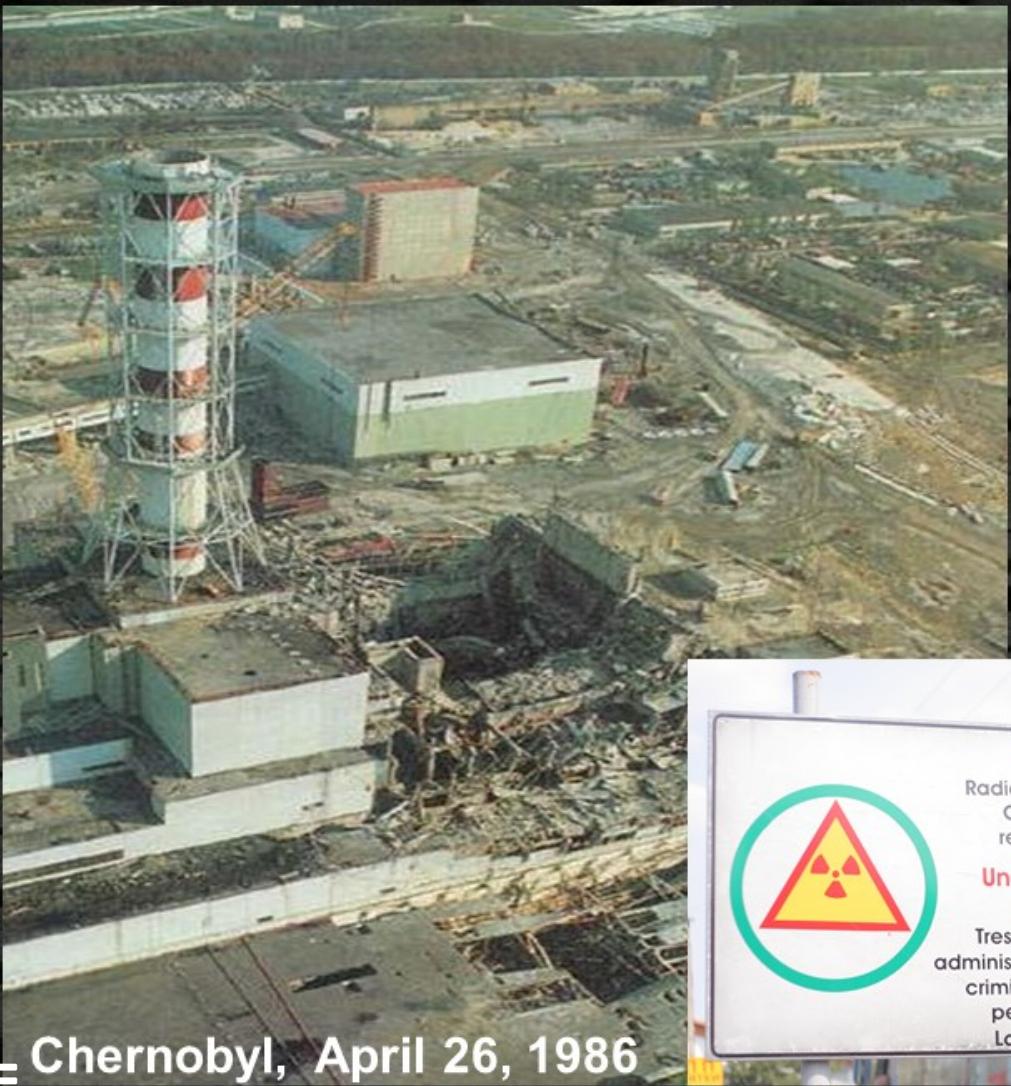


Etcetera

EC – embedded controllers

BMC – Baseboard Management Component

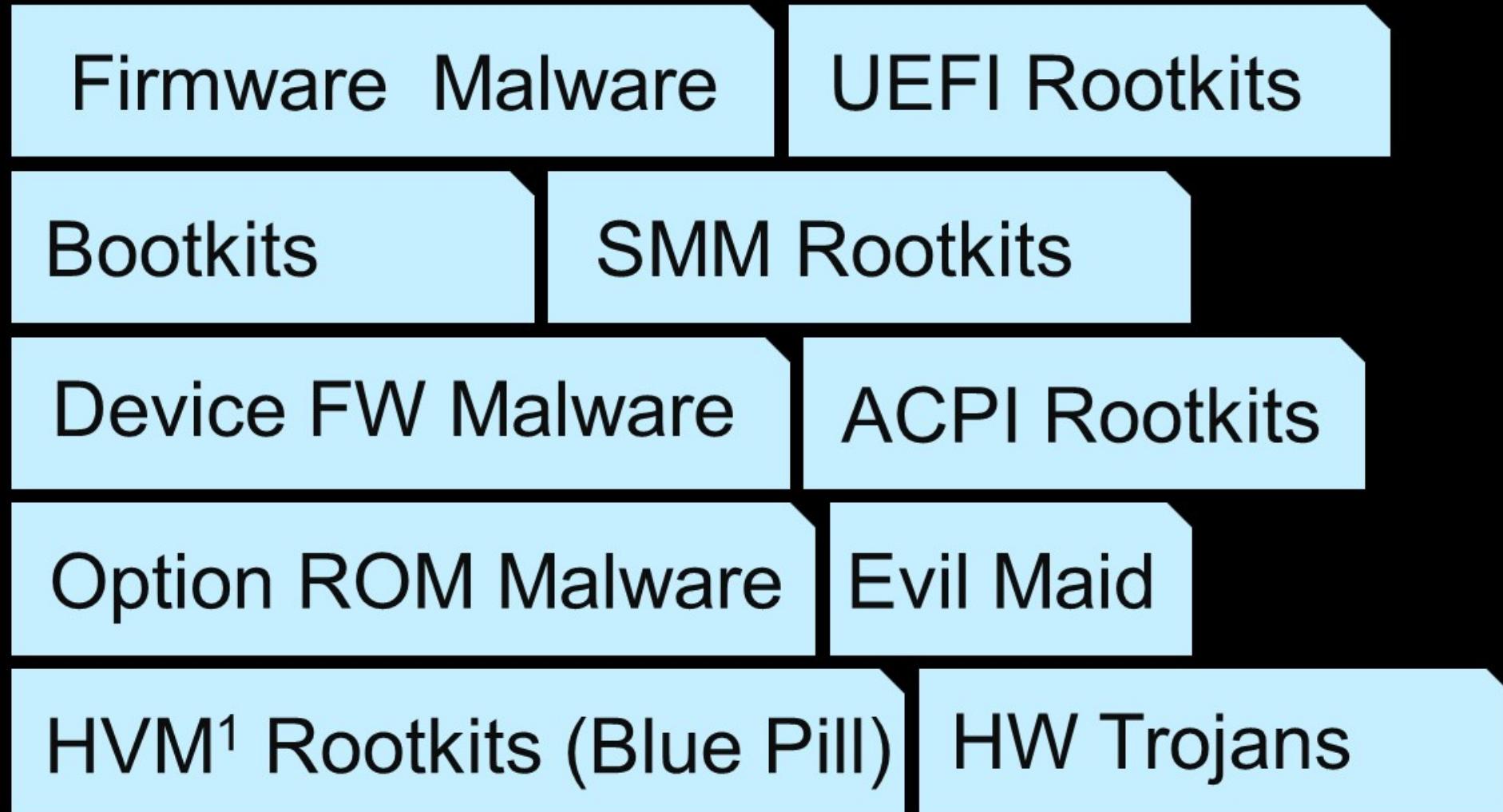
Making System Unbootable (Bricking)



- Corrupt firmware or
- Corrupt critical configuration
- Stored in flash “ROM” memory
- Of a device which is critical for system to boot or operate

Source : CIH virus, was first discovered in 1998

Pre-Boot Threats



≡ Summary Why Attack Firmware

¹ Hypervisor Virtual Machine

Platform Firmware Security – Why is it important?

Why is platform firmware Security important

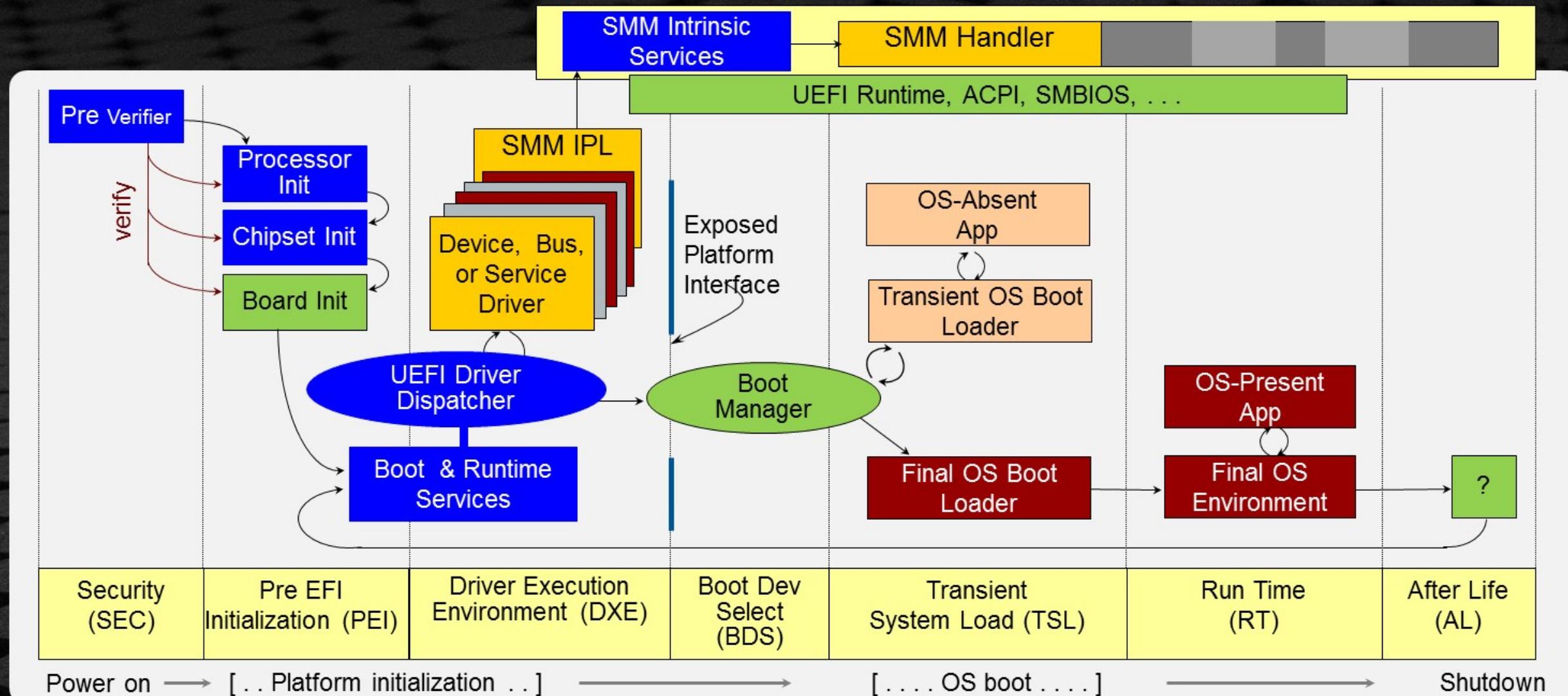
→ Prevent low level attacks that could “brick” the system

UEFI BOOT FLOW

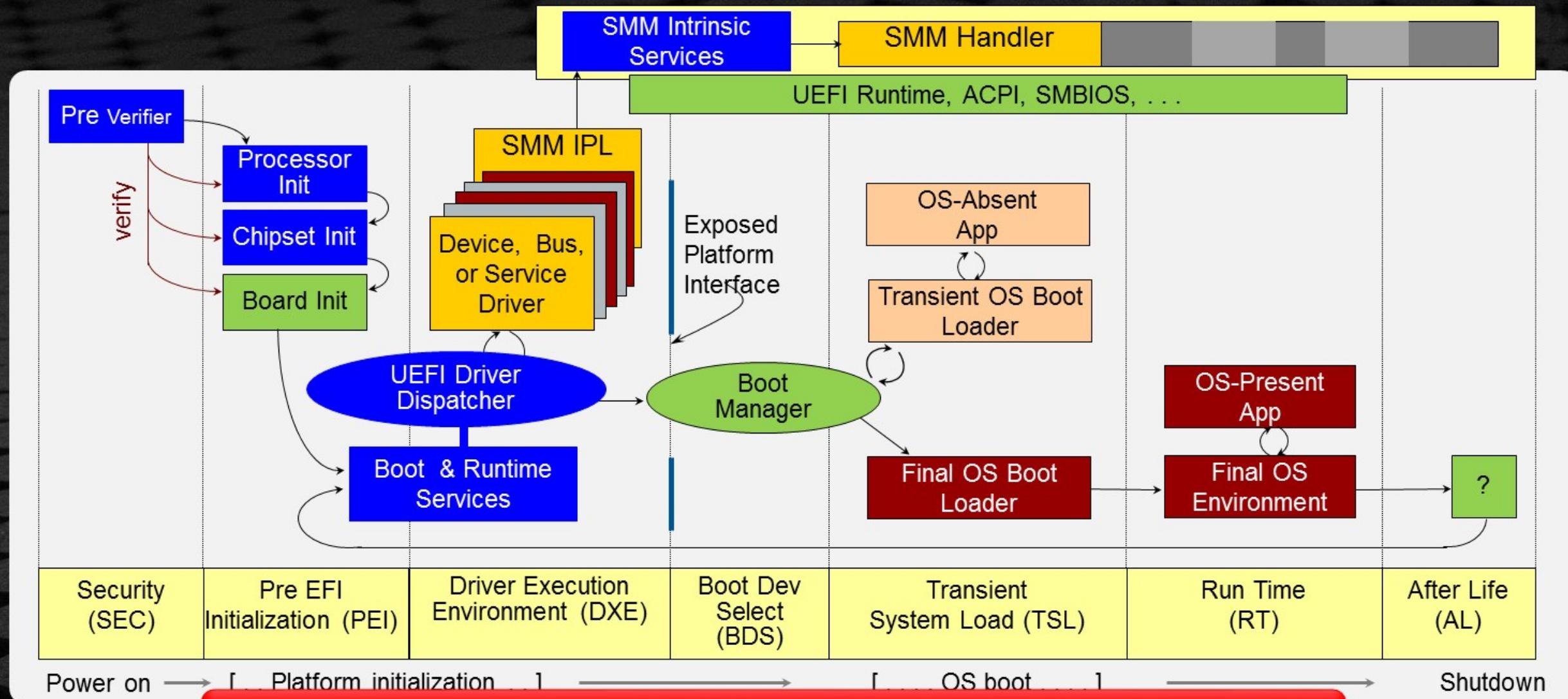
UEFI boot flow with the threat model



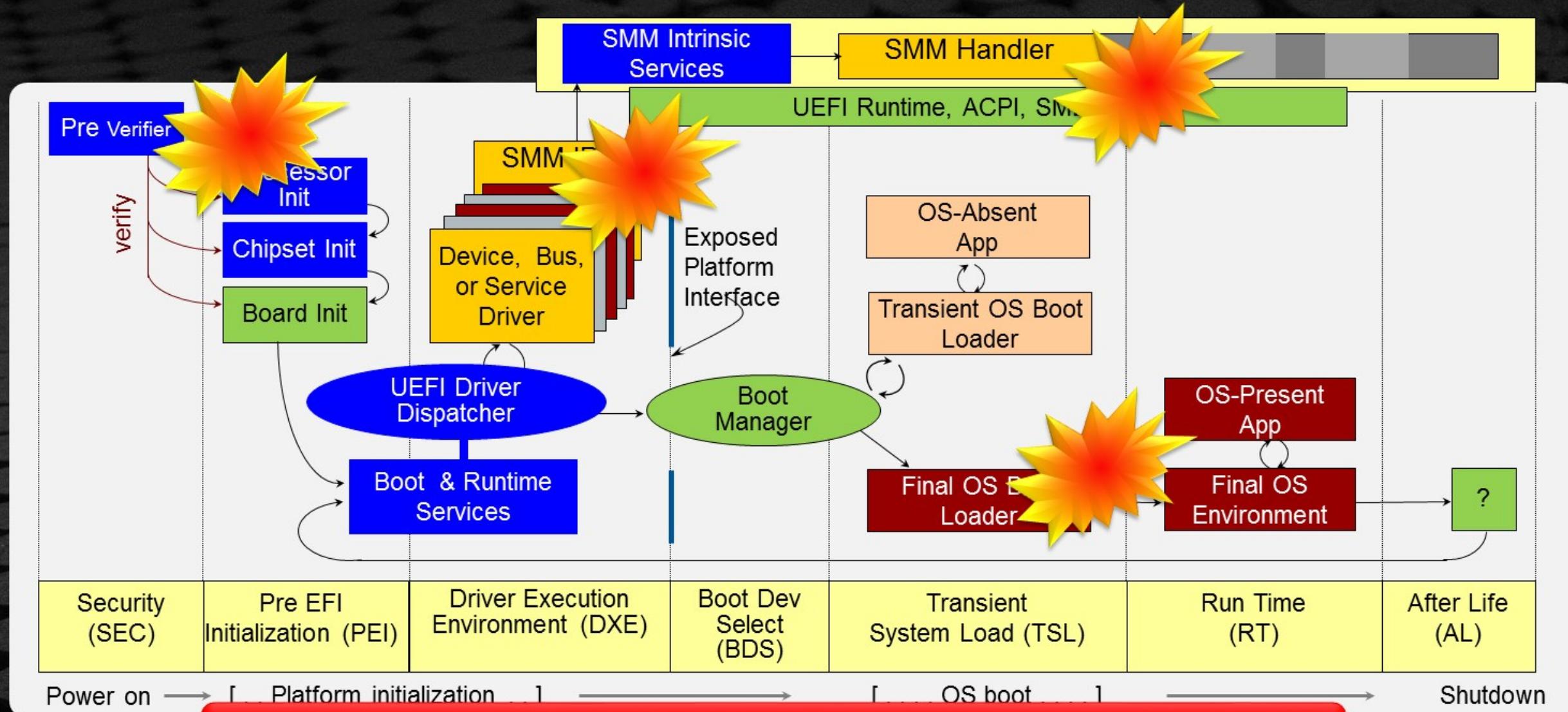
UEFI Boot Execution Flow



UEFI Boot Execution Flow

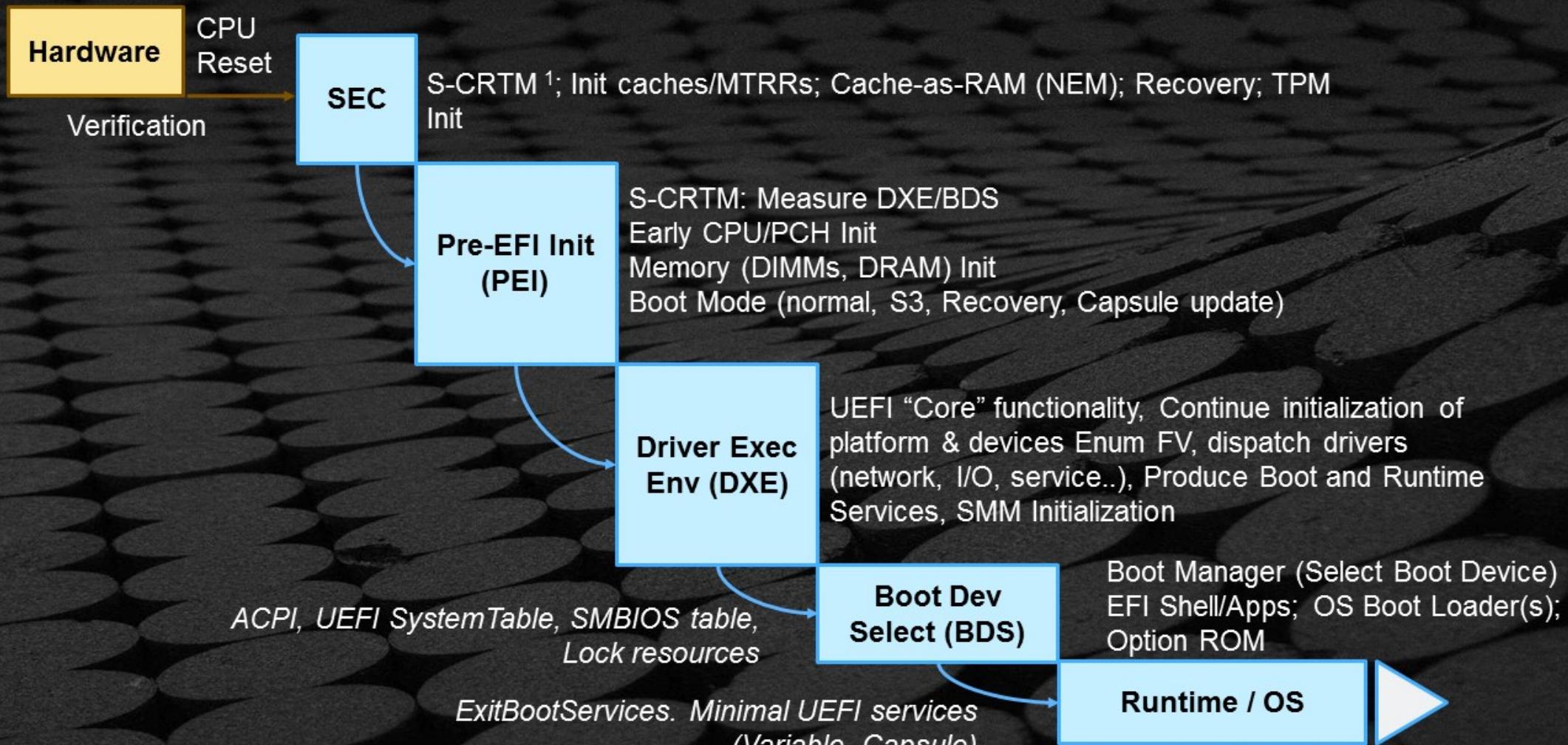


UEFI Boot Execution Flow



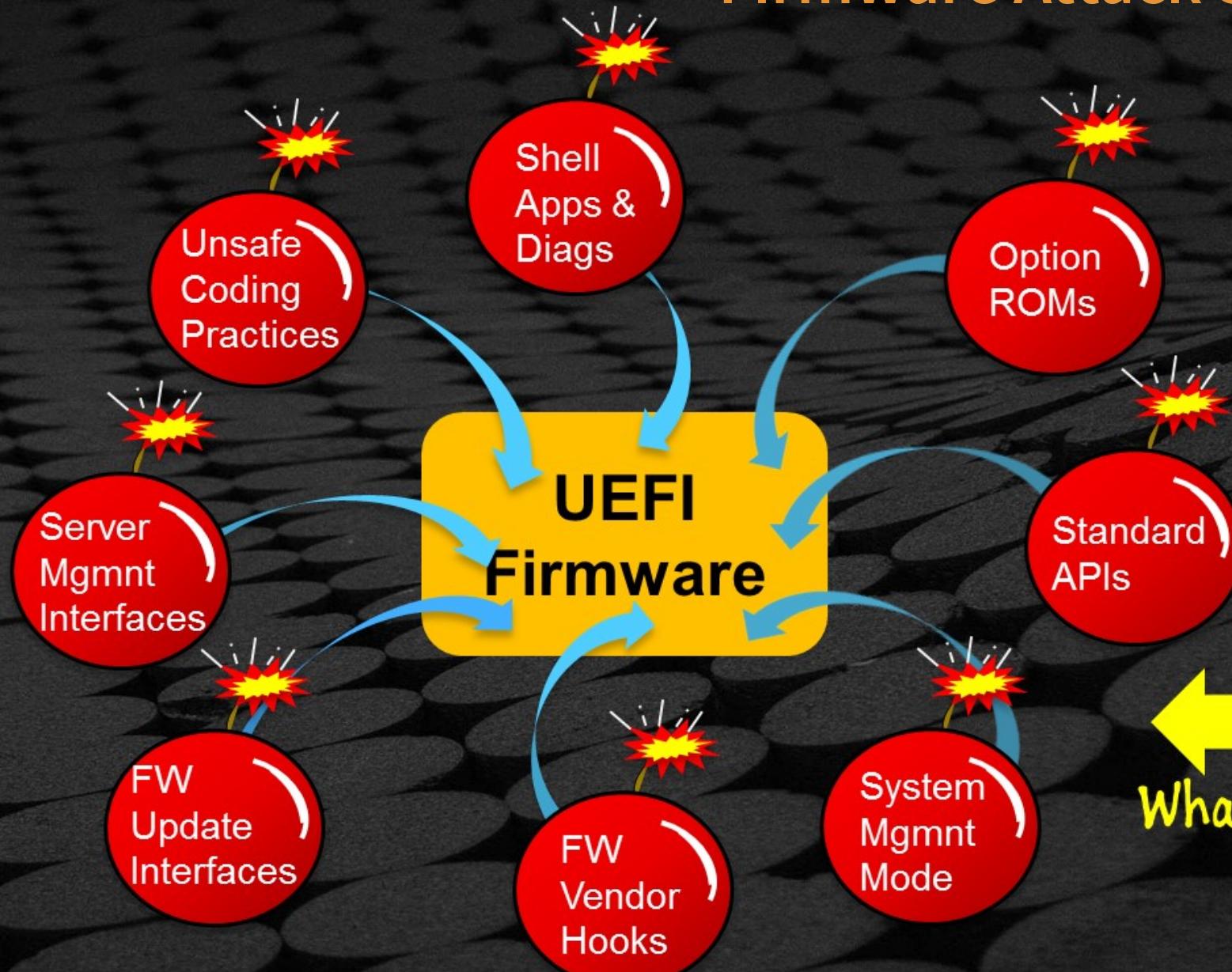
What Could Possibly Go Wrong ???

UEFI & Platform Initialization Task Flow



¹ S-CRTM - static core root of trust for measurement

Firmware Attack Surfaces



Goals of security architecture and assets that are protected

[NIST SP 800-33](#) – IT Security Objectives

Availability, Integrity, Confidentiality, Accountability, Assurance

The goal of information technology security:

Enable an organization to meet all of its mission/business objectives by implementing systems with due care consideration of IT-related risks to the organization, its partners and customers.

Goals of security architecture and assets that are protected

[NIST SP 800-33](#) – IT Security Objectives

Availability, Integrity, Confidentiality, Accountability, Assurance

Confidentiality

- Protect against unauthorized access

Integrity

- Protection of Content & Quality

Availability

- Ensure access

Accountability

- Also Authenticity - traced uniquely to the source entity

Assurance

- Guarantee on the correctness, Non-repudiation

Goals of security architecture and assets that are protected

[NIST SP 800-33](#) – IT Security Objectives

Availability, Integrity, Confidentiality, Accountability, Assurance

Confidentiality

- Protect against unauthorized access

Integrity

- Protection of Content & Quality

Availability

- Ensure access

Accountability

- Also Authenticity - traced uniquely to the source entity

Assurance

- Guarantee on the correctness, Non-repudiation

All of these objectives are interdependent with Assurance

What to build & defend – Rationale for a threat model

“My house is secure” is almost meaningless

- Against a burglar? Against a meteor strike? A thermonuclear device?

“My system is secure” is almost meaningless

- Against what? To what extent?

Threat modeling is a process to define the goals and constraints of a (software) security solution

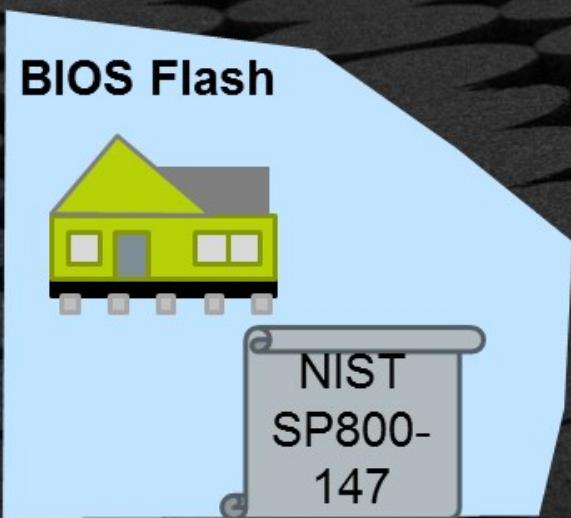
- Translate user requirements to security requirements

We use threat modeling for our UEFI / PI codebase

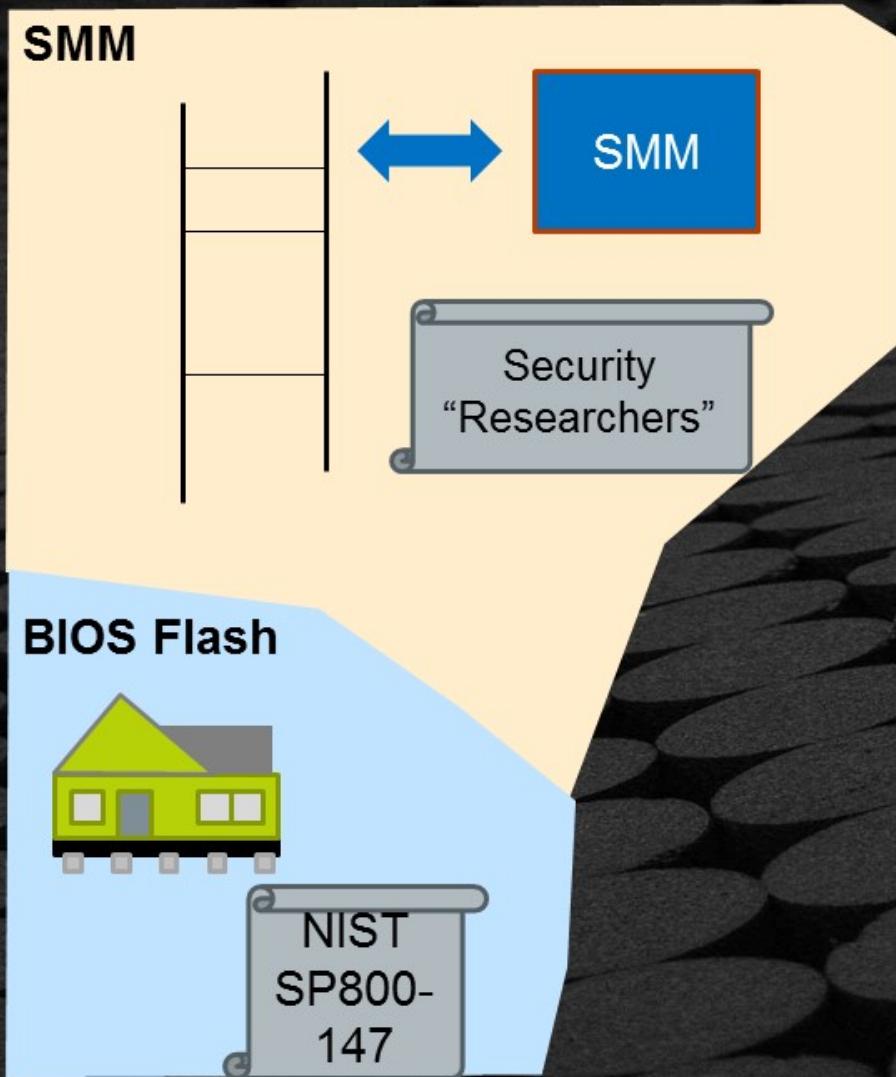
- We believe the process and findings are applicable to driver implementations as well as UEFI implementations in general

≡ We Need to protect our Assets from Threats

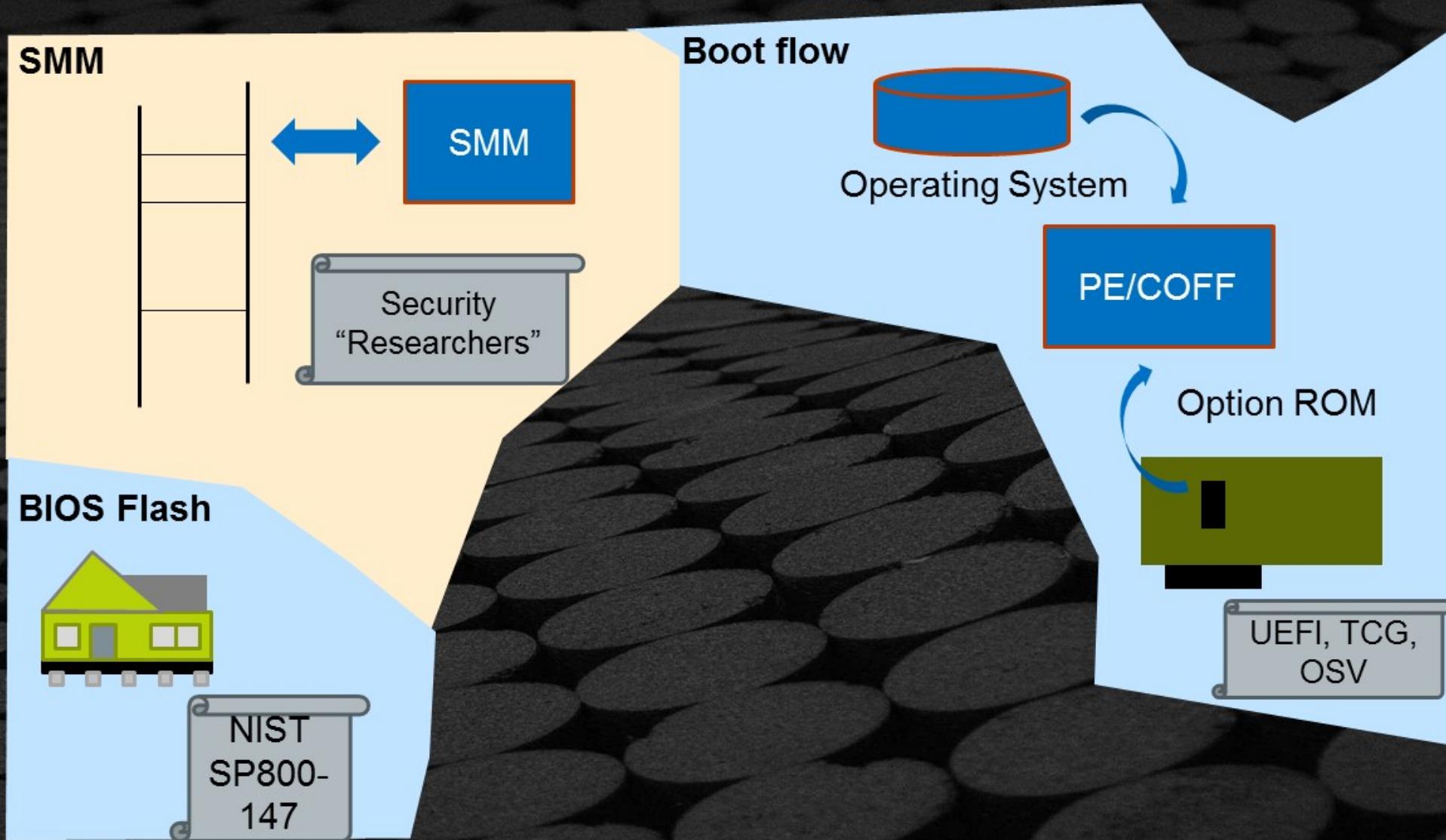
We don't always get to choose our Assets



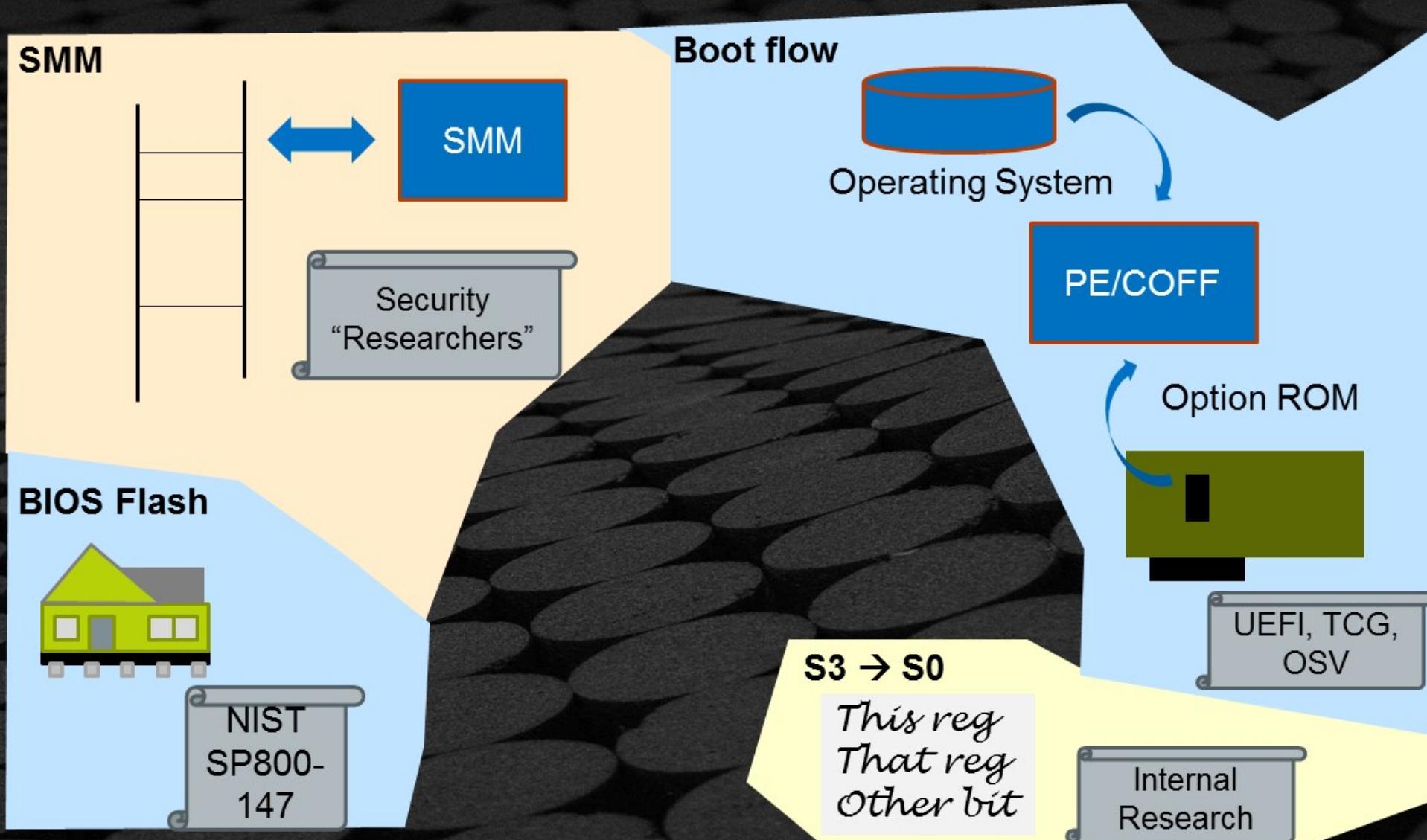
We don't always get to choose our Assets



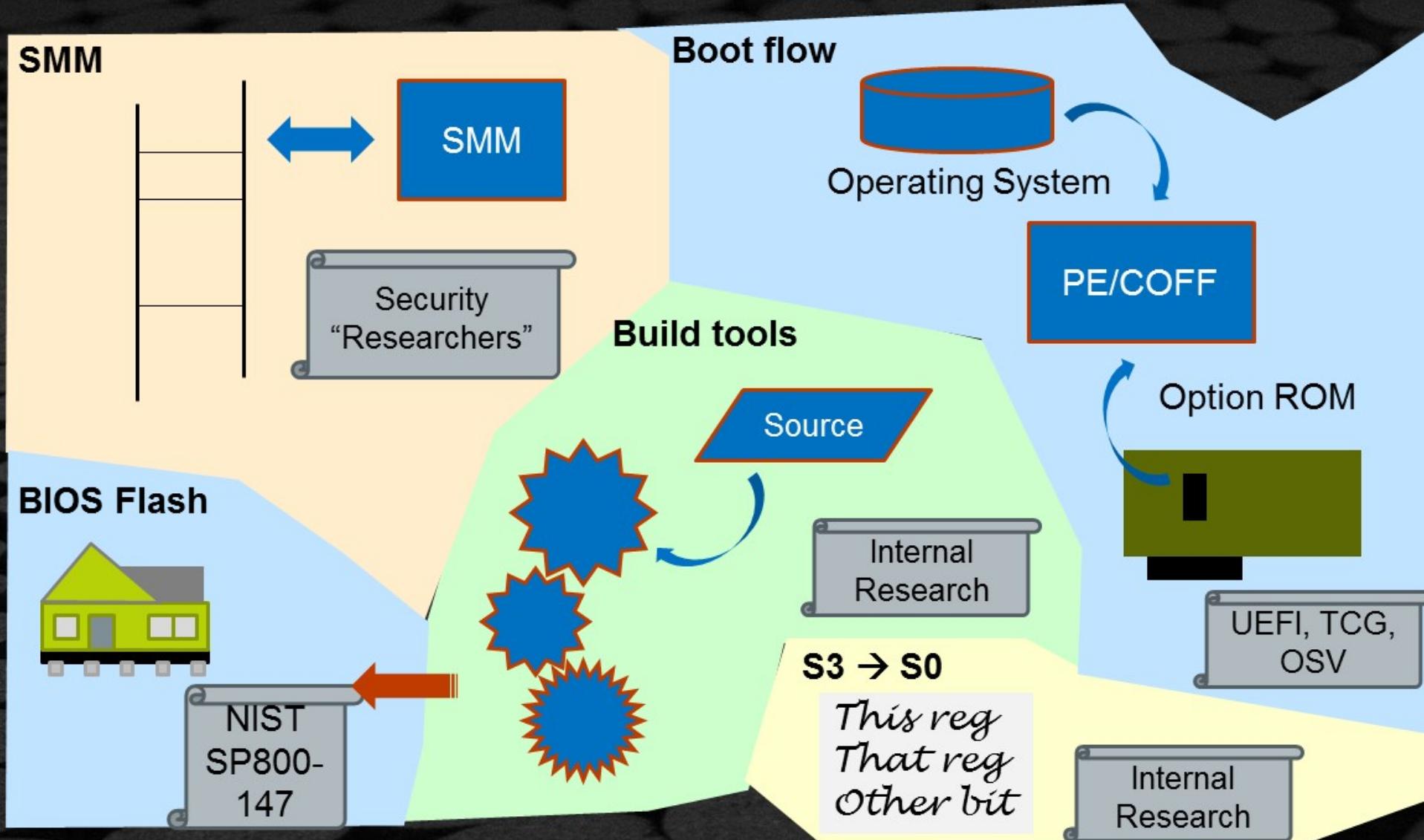
We don't always get to choose our Assets



We don't always get to choose our Assets



We don't always get to choose our Assets

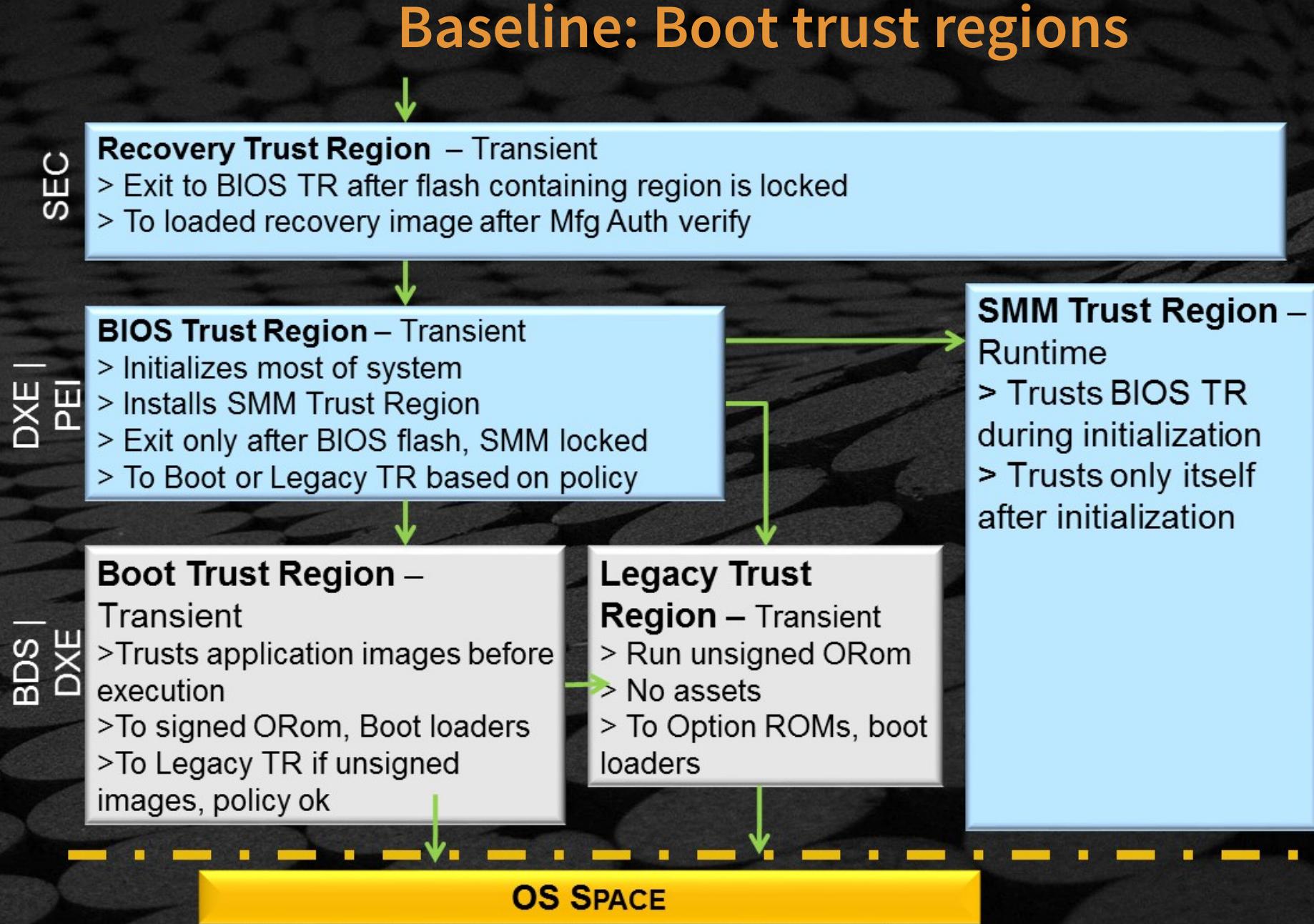


A Threat Model (TM) defines the security assertions and constraints for a product

- Assets:
- Threats:
- Mitigations:

Mfg Auth only

MA + Others



Threat Model with Examples

Boot Trust Regions and Assets

Asset	Example Threats	Mitigations	Checks
Firmware/BIOS Flash Contents	CIH attack: erase boot block	SPI locks, descriptor	CHIPSEC
SMM	Callouts; Access to SMM	TSEG, SMRR, SMM_CODE_CHK	CHIPSEC
Execution During Boot Flow	Run malware in Op ROM	Secure Boot, DMA protection	Manual testing (eg CHIPSEC)
S3 Boot Script & S3 Resume Boot Flow	Resume reconfiguration losing locks	SMM Lock Box	Manual testing (eg CHIPSEC)
UEFI Variables (includes Authenticated & non-Authenticated)	Variable store full; Content change	Attributes, Lock Protocol	Manual testing (eg CHIPSEC)
Etc . . .			

Platform Firmware Security – Why is it important?

Why is platform firmware Security important

→ Prevent low level attacks that could “brick” the system

UEFI boot flow with the threat model

→ Identify where UEFI firmware is vulnerable and define a Threat Model

SECURITY TECHNOLOGIES

Security Technologies Overview



BOOT SECURITY



Boot Security Technologies

Hardware Root of Trust

Measured Boot

Verified Boot

- Boot Guard, Intel® TXT
- Using TPM to store hash values
- Boot Guard + UEFI Secure Boot

Hardware root of Trust

Boot Guard

CPU verifies signature

Verification occurs before BIOS starts

Hash of signature is fused in CPU

Verification

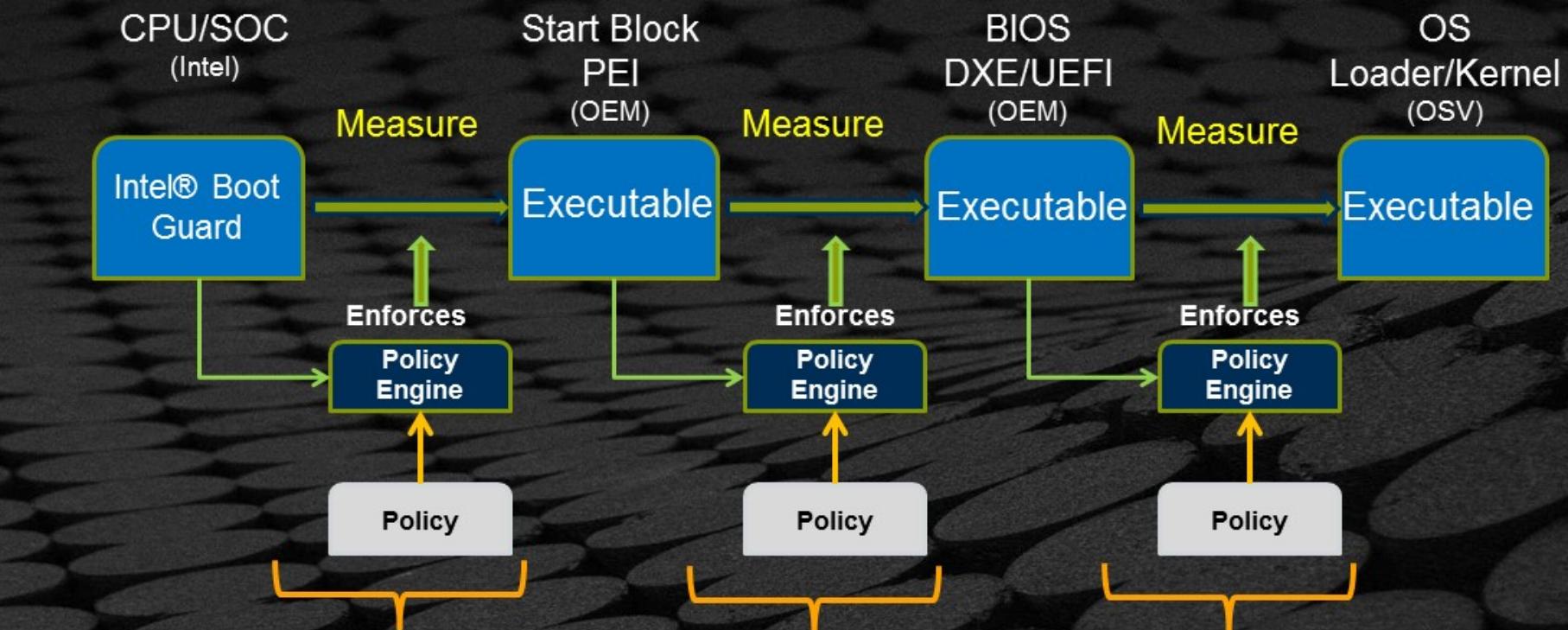
Intel® TXT

Uses a Trusted Platform Module (TPM)
& cryptographic

Provides Measurements

Measurements

Full Verified Boot Sequence



Intel® Device Protection Technology with Boot Guard

References : [Platform Security Review PDF](#),
[4th-gen-core-family PDF](#)

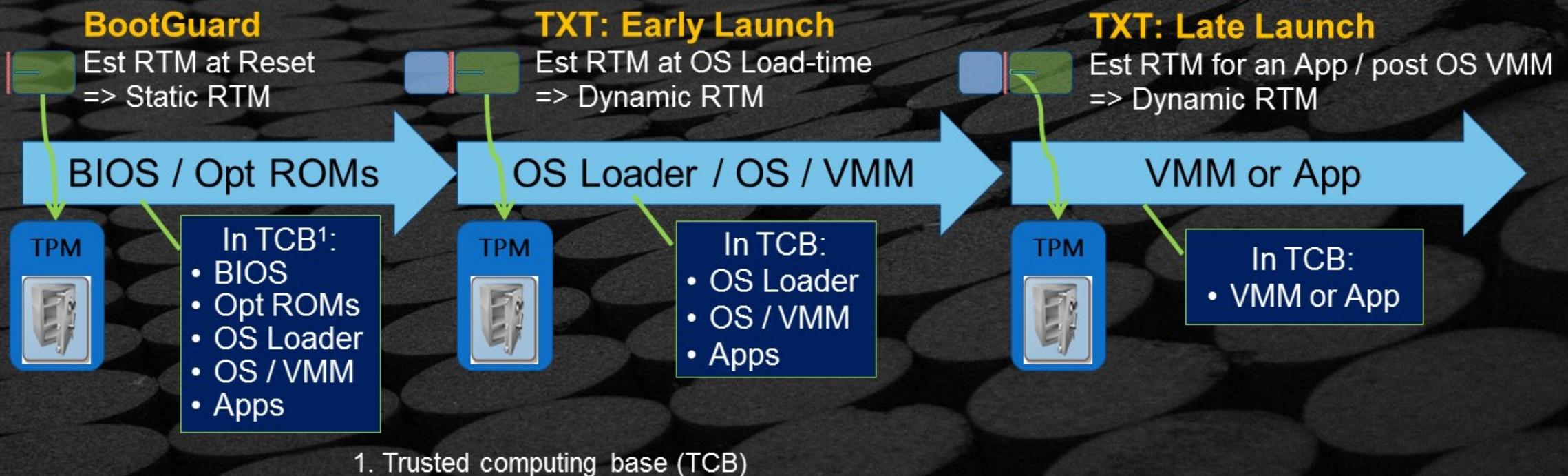
OEM PI Verification Using PI Signed Firmware Volumes
Vol 3, section 3.2.1.1 of PI 1.3 Specification

OEM UEFI 2.4 Secure Boot

Chapter 27.2 of The UEFI 2.4 Specification

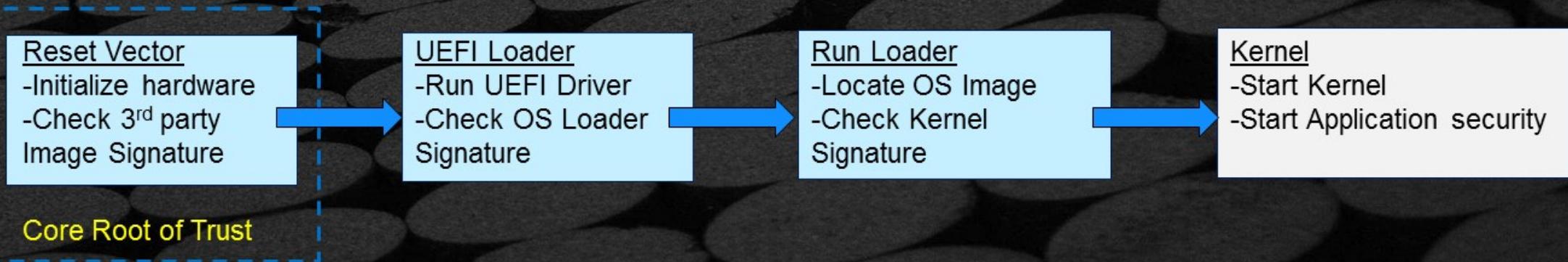
Both features rooted in Trusted Computing.

Establishing a Trusted Environment (Intel provides 3 modes)
Starts with Root of Trust for Measurement (RTM)



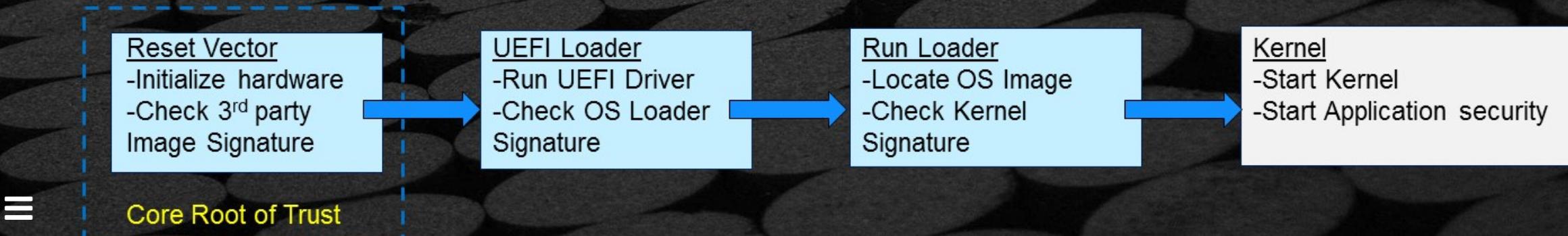
Software ID checking during every step of the boot flow:

1. UEFI System BIOS (updated via secure process)



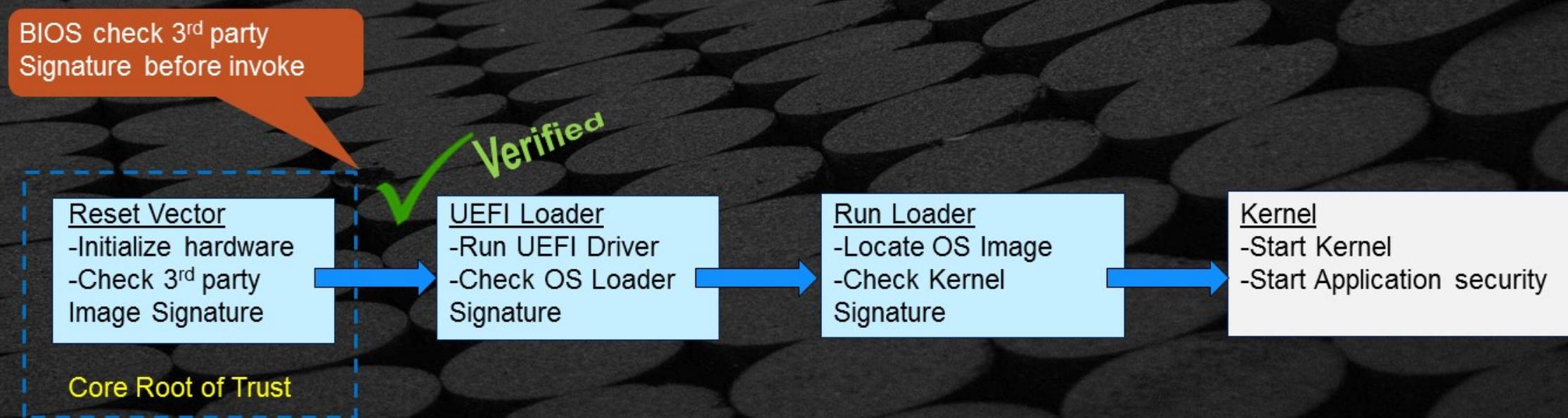
Software ID checking during every step of the boot flow:

1. UEFI System BIOS (updated via secure process)
2. Add-In Cards (signed UEFI Option ROMs)



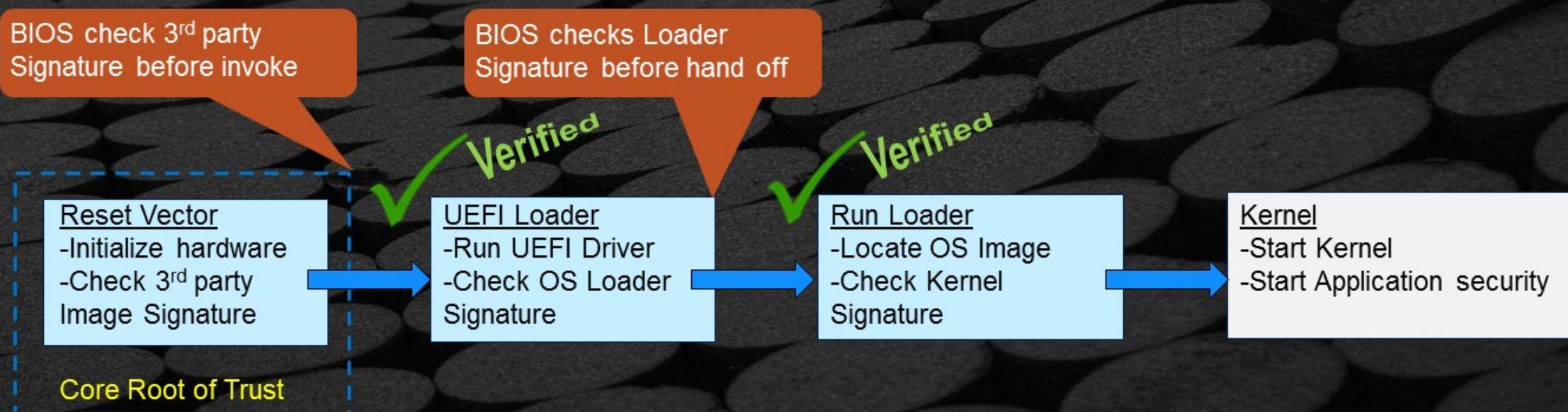
Software ID checking during every step of the boot flow:

1. UEFI System BIOS (updated via secure process)
2. Add-In Cards (signed UEFI Option ROMs)



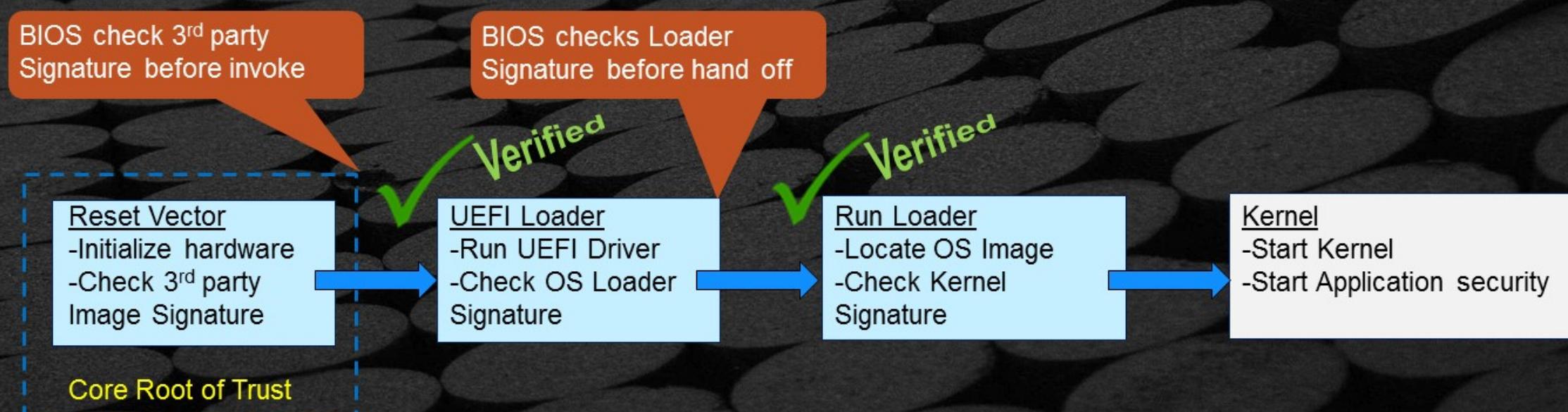
Software ID checking during every step of the boot flow:

1. UEFI System BIOS (updated via secure process)
2. Add-In Cards (signed UEFI Option ROMs)



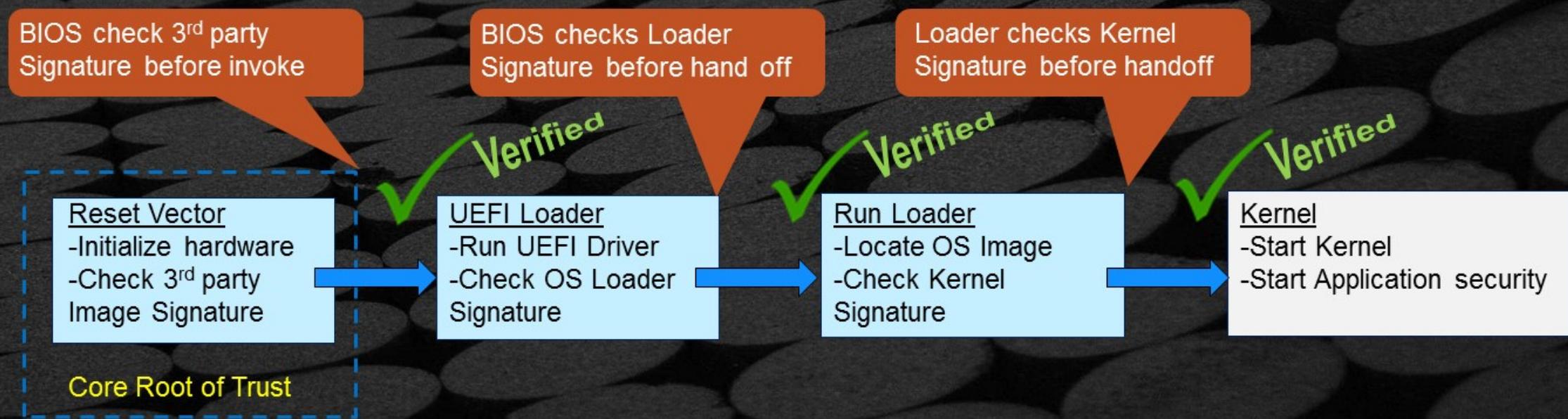
Software ID checking during every step of the boot flow:

1. UEFI System BIOS (updated via secure process)
2. Add-In Cards (signed UEFI Option ROMs)
3. OS Boot Loader (checks for “secure mode” at boot)

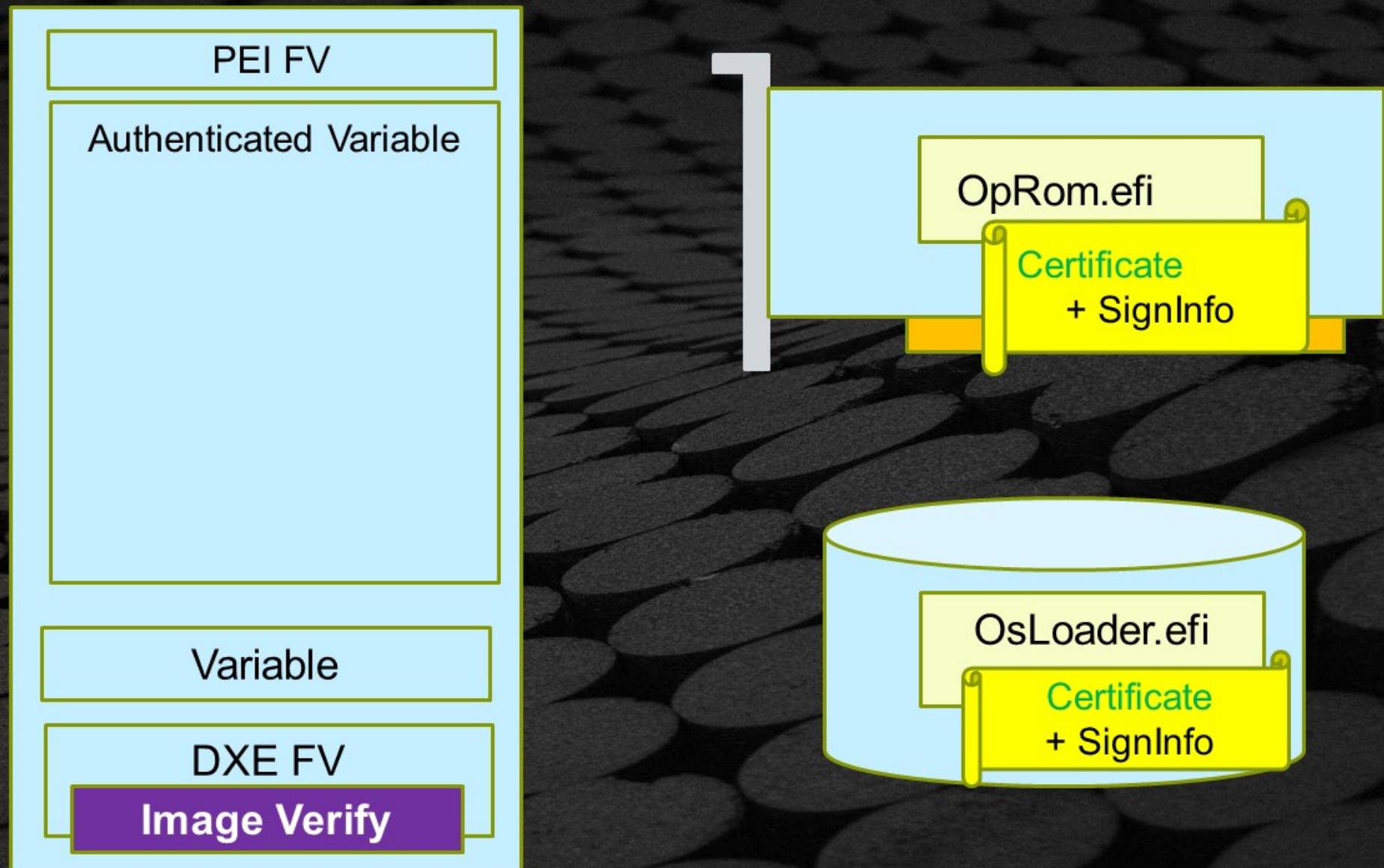


Software ID checking during every step of the boot flow:

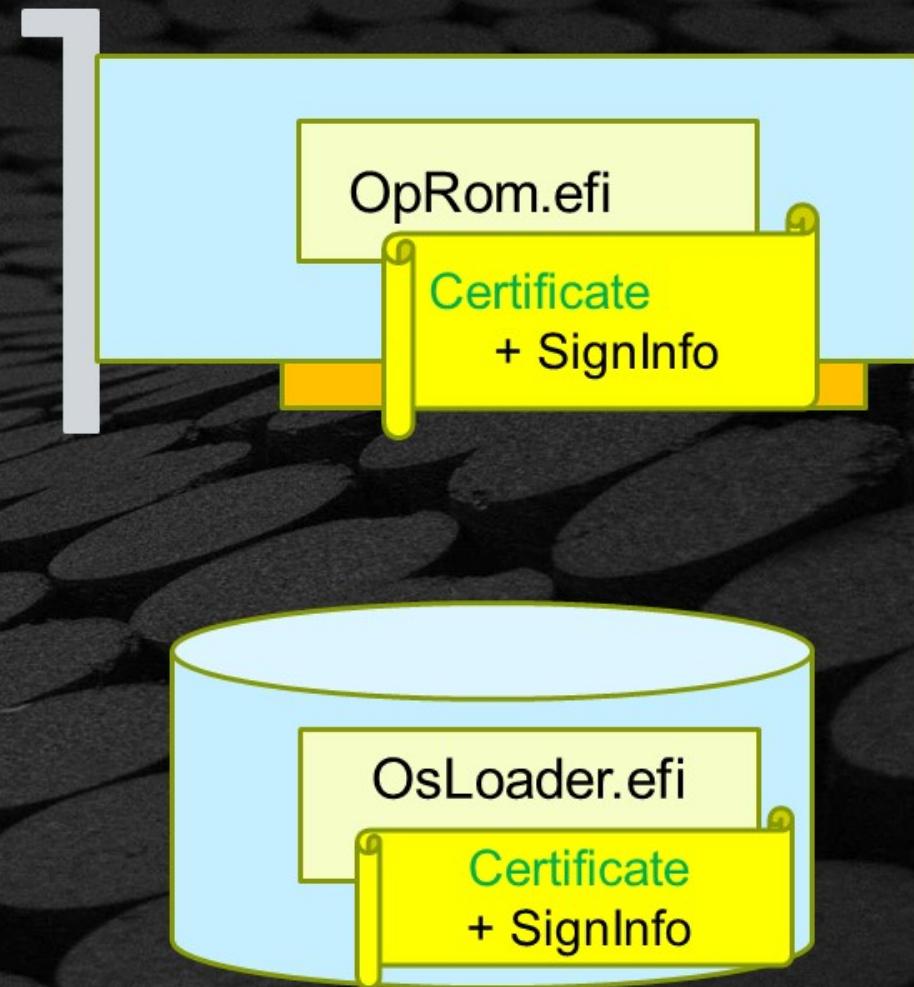
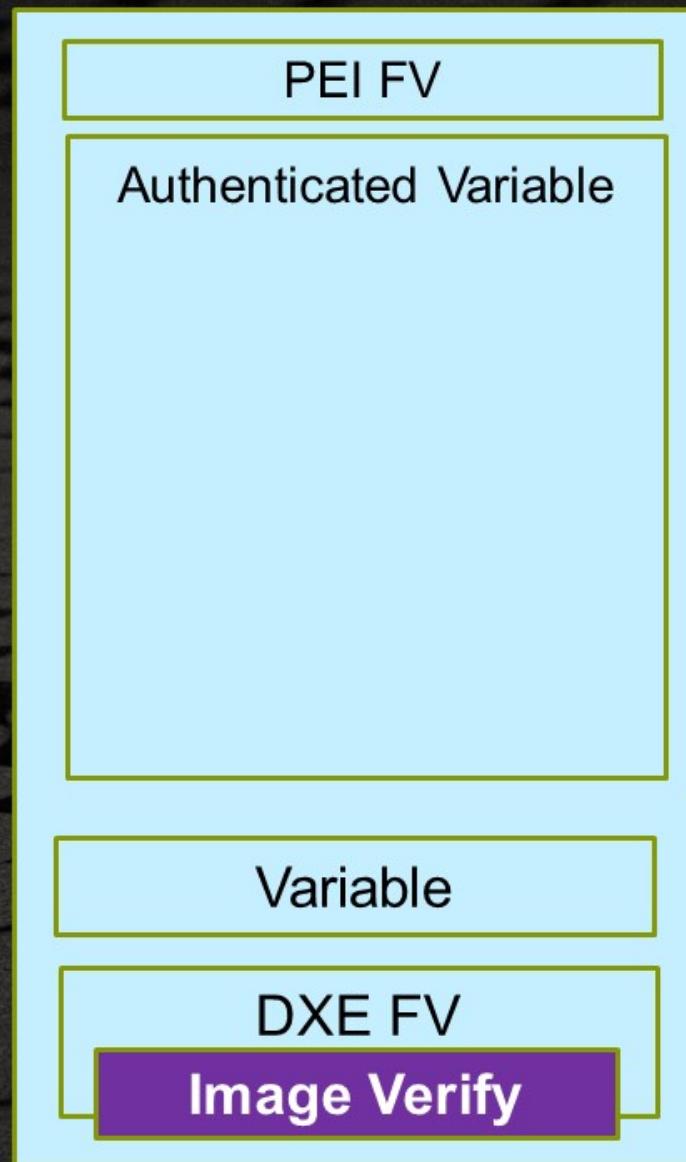
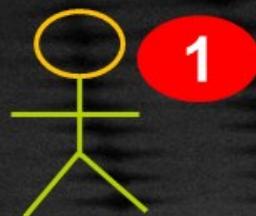
1. UEFI System BIOS (updated via secure process)
2. Add-In Cards (signed UEFI Option ROMs)
3. OS Boot Loader (checks for “secure mode” at boot)



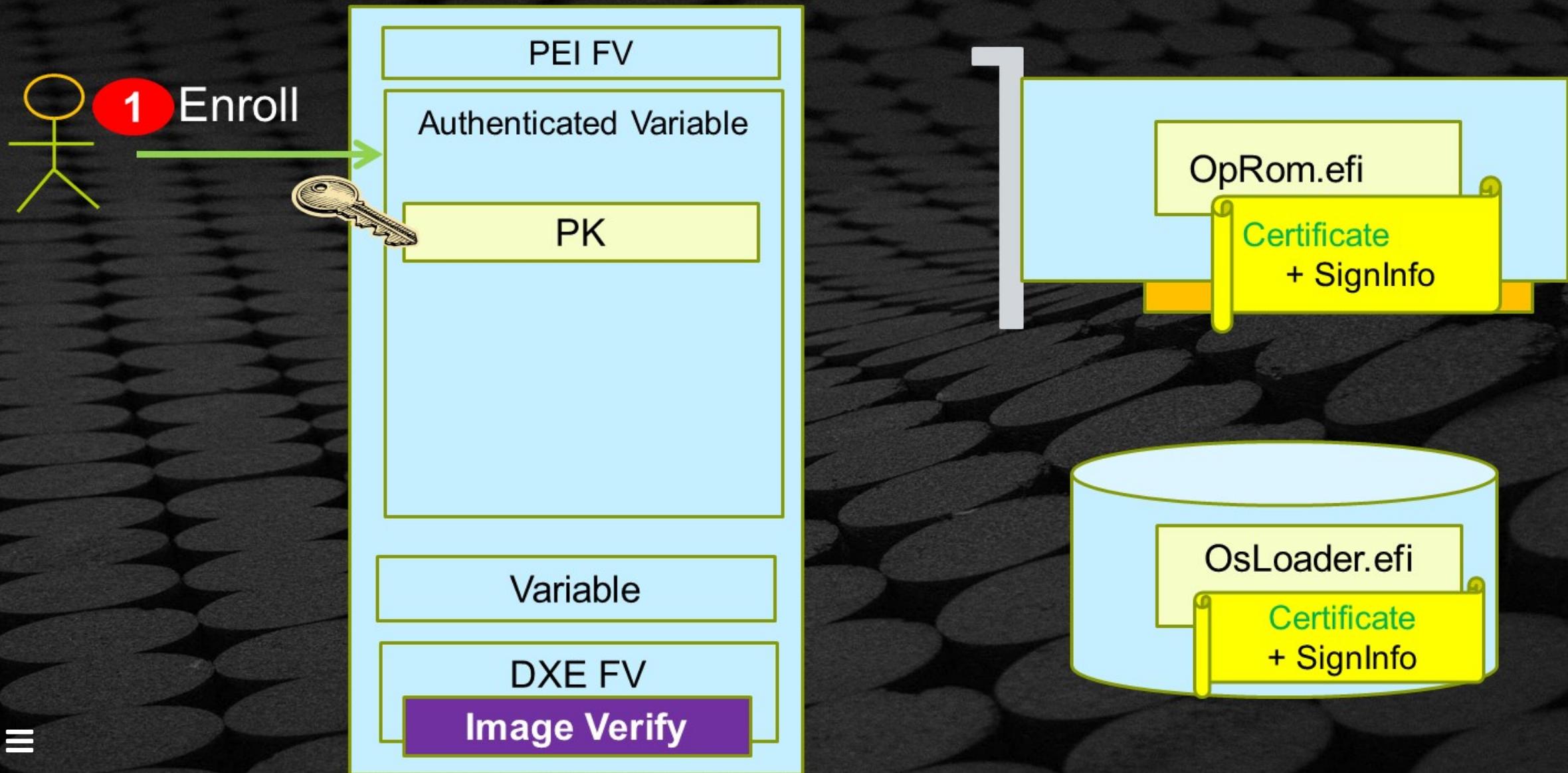
UEFI Secure Boot Flow



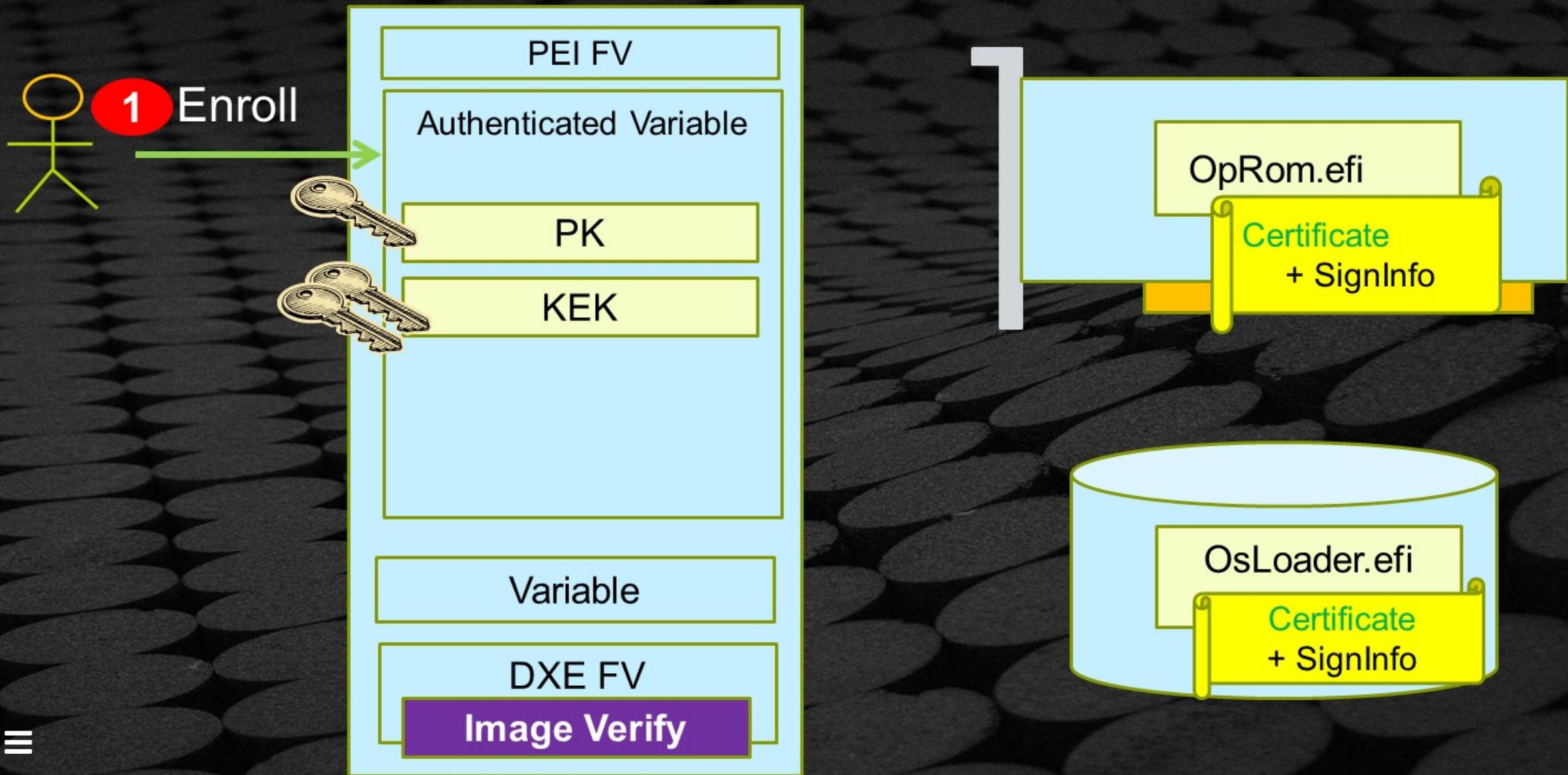
UEFI Secure Boot Flow



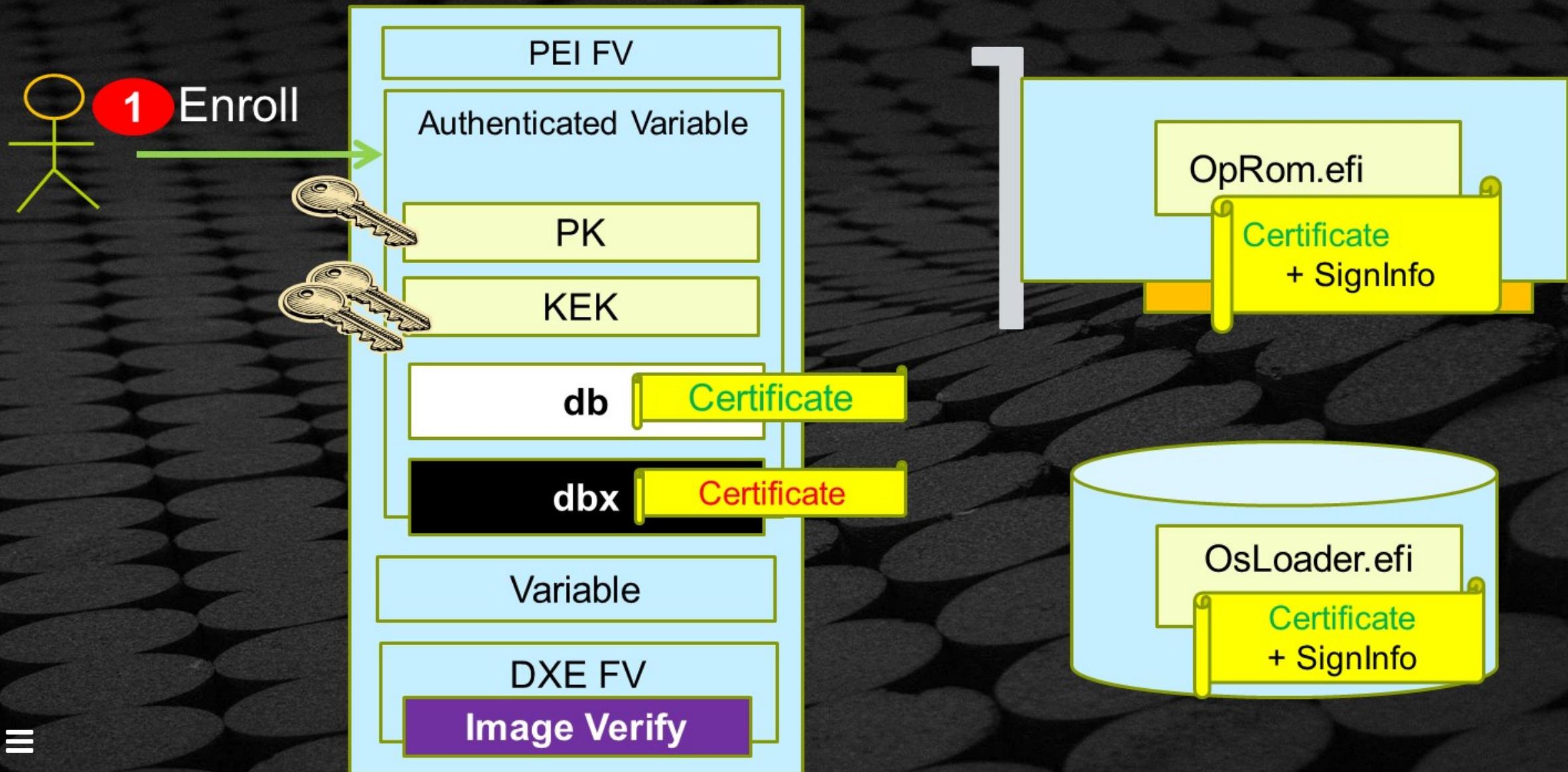
UEFI Secure Boot Flow



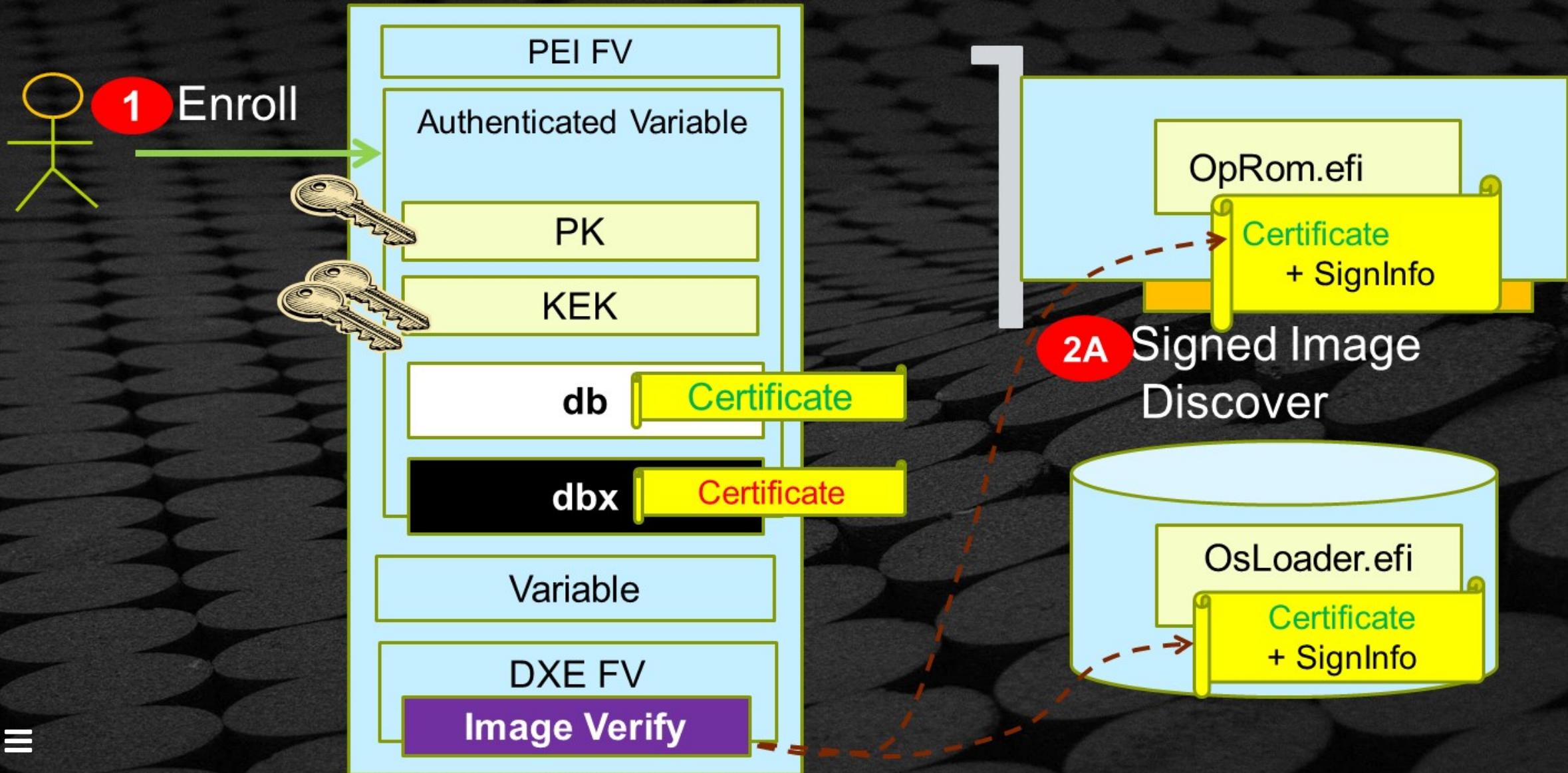
UEFI Secure Boot Flow



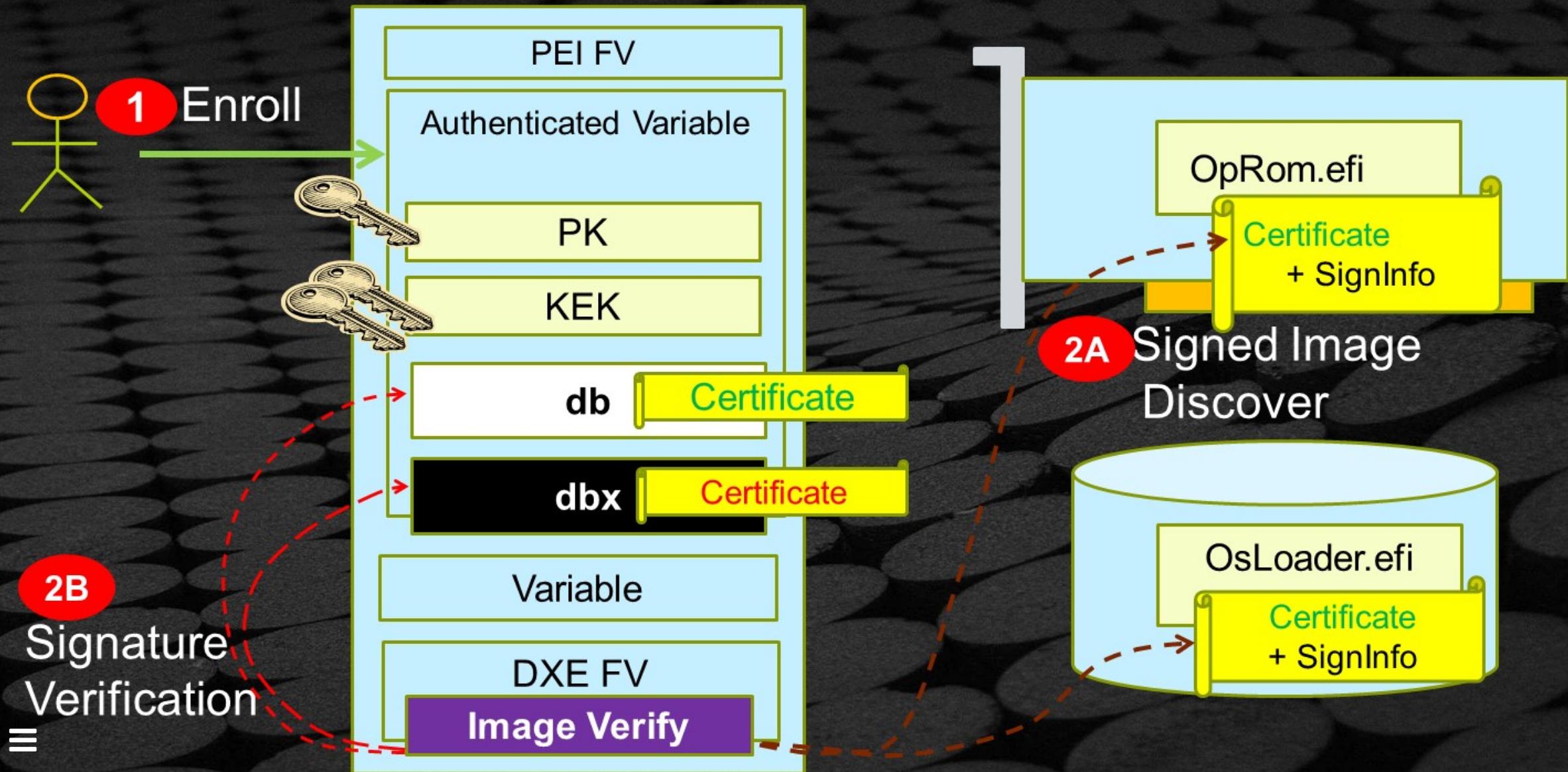
UEFI Secure Boot Flow



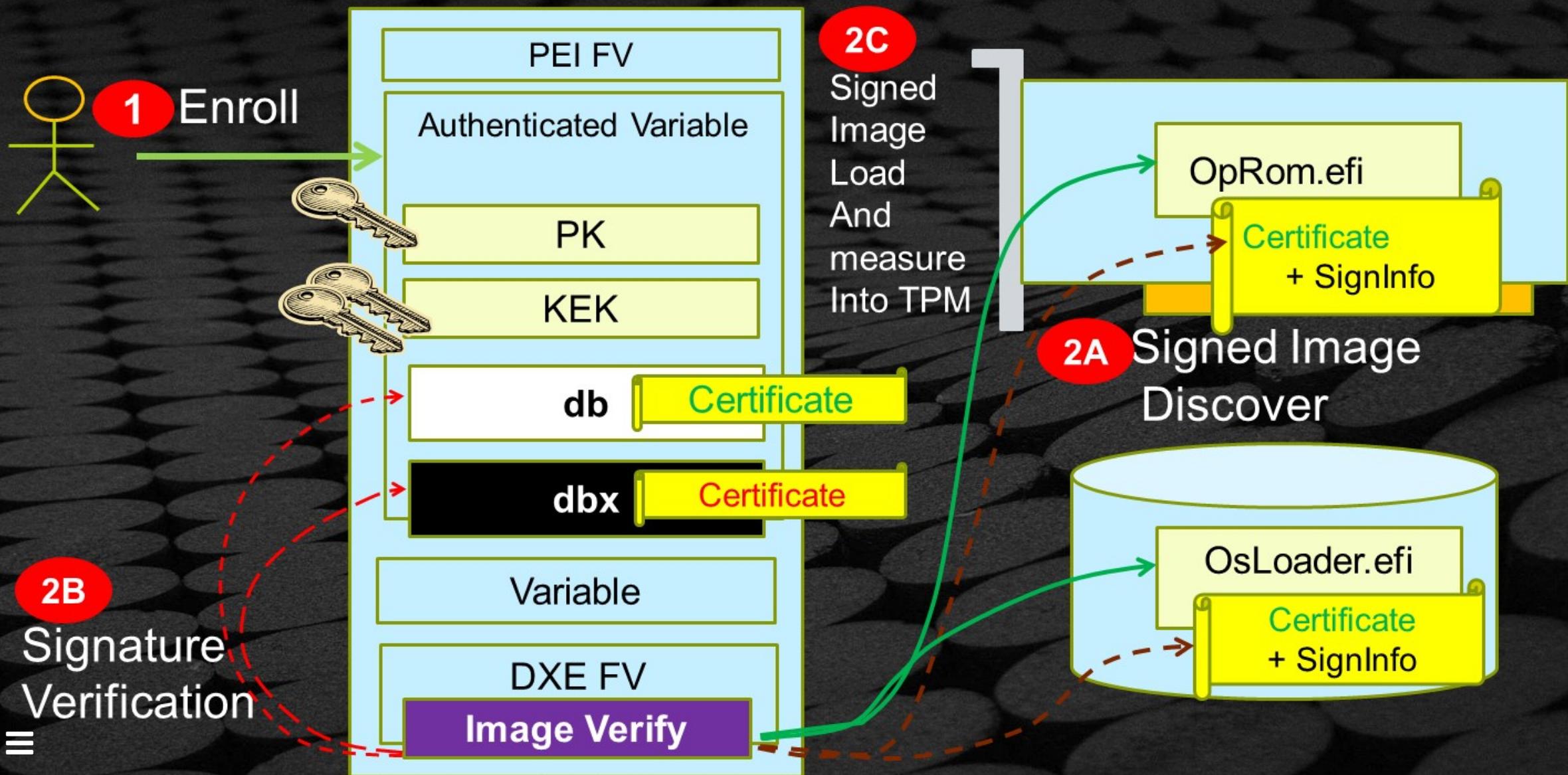
UEFI Secure Boot Flow



UEFI Secure Boot Flow

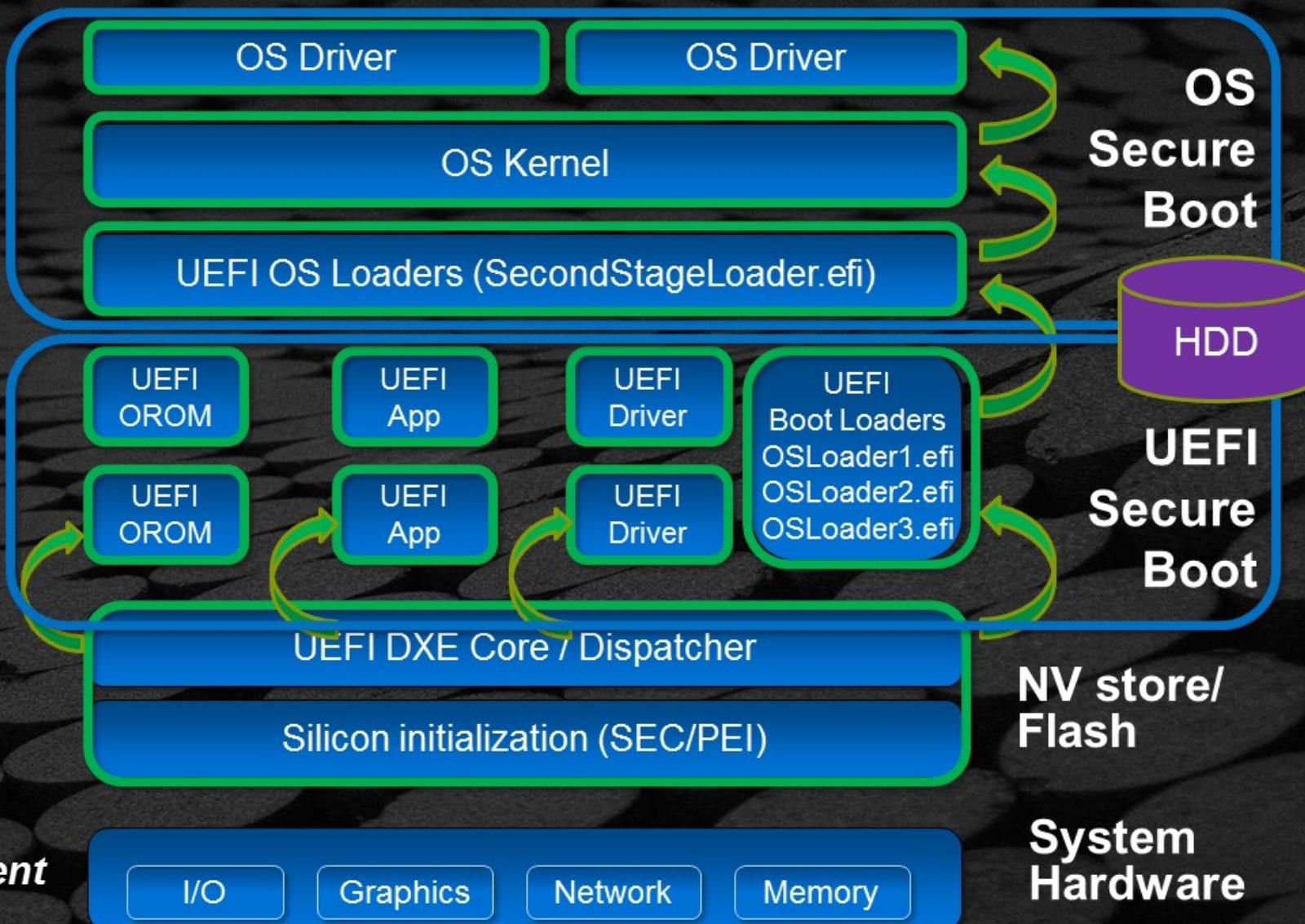


UEFI Secure Boot Flow



End to End Platform Integrity

Reference:
[Where do I 'sign' up?](#)

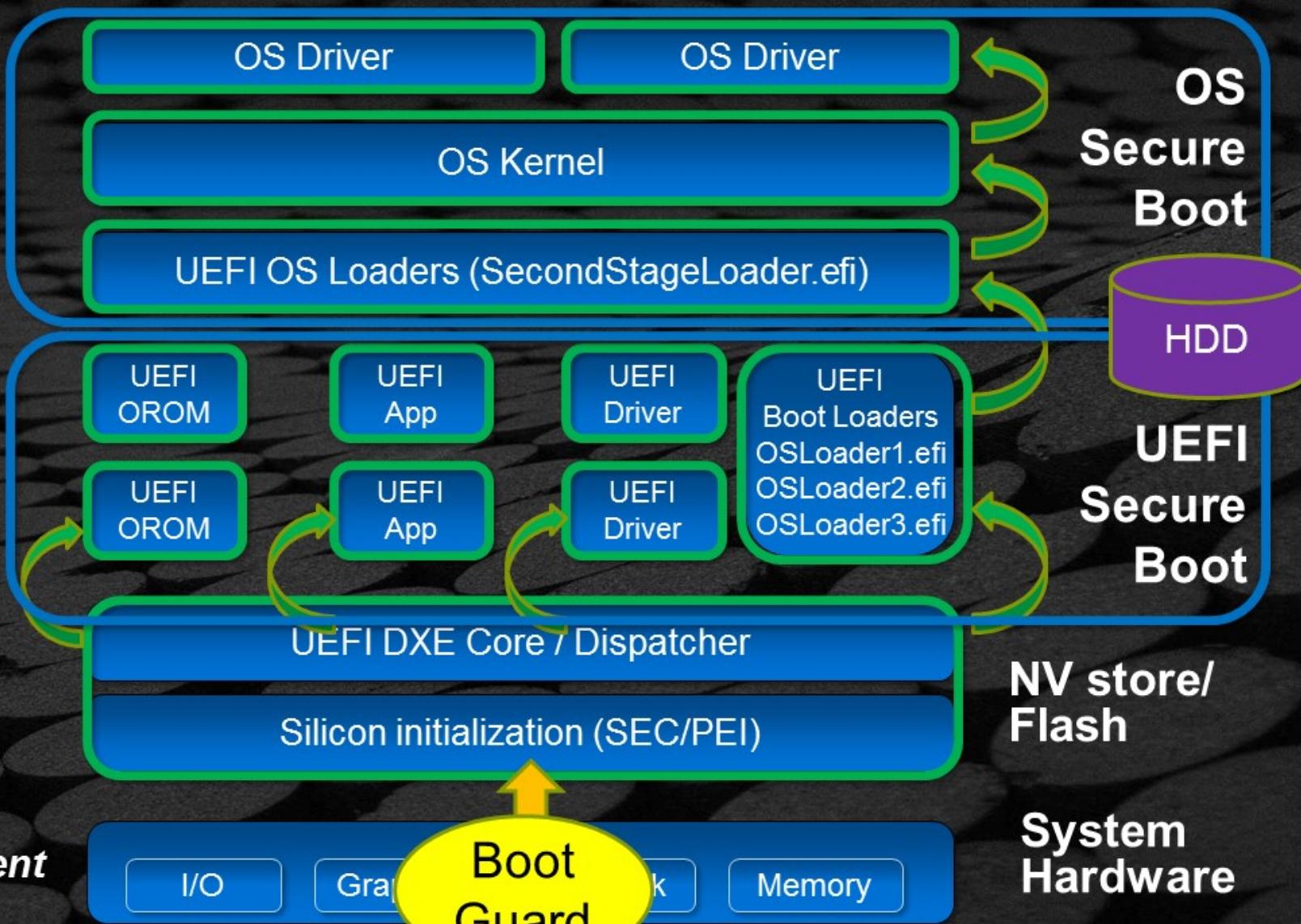


Intel® Device Protection
Technology with
Boot Guard – Secure
Boot Policy Enforcement

End to End Platform Integrity

Reference:
[Where do I 'sign' up?](#)

Intel® Device Protection
Technology with
Boot Guard – Secure
Boot Policy Enforcement



FLASH DEVICE



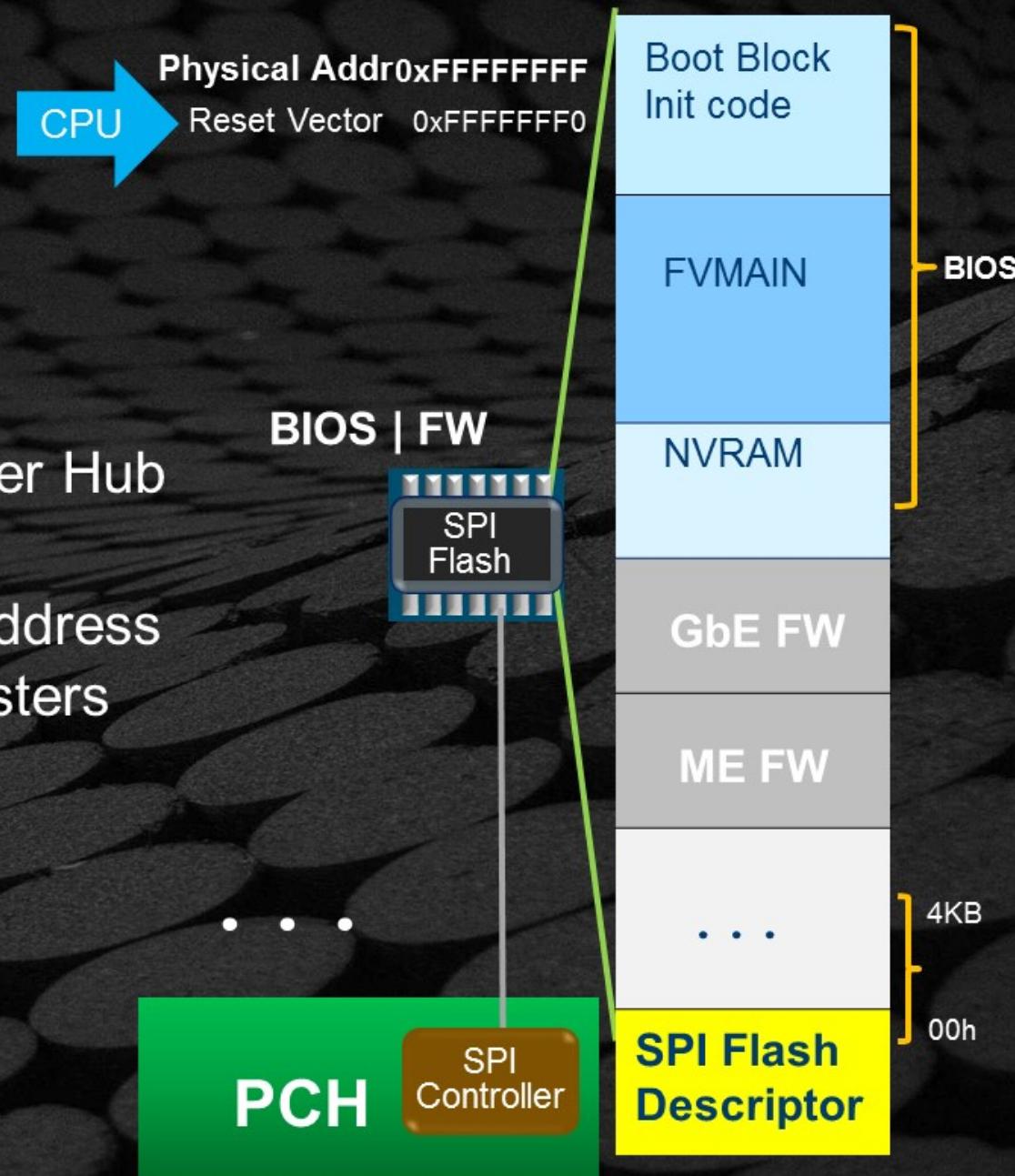
BIOS Firmware uses SPI Flash

Serial Peripheral Interface (SPI)

- Access controlled by the Peripheral Controller Hub (PCH)
- Flash Memory - Direct access to physical address space is programmed through SPI MMIO registers
- SPI Flash Descriptor - Access Control table defines which masters (CPU, ME, GbE) can access which regions

Flash Descriptor has to be write-protected

≡ ME – Manageability Engine
 ≡ GbE – Gigabit Ethernet



Chipset (SPI controller) based protections

- SMM based BIOS Write Protection: write-protects entire BIOS/Firmware region from software other than SMI handler firmware executing in SMM
- SPI Protected Range registers(PR0 - PR4) : read/write protection of SPI flash regions based on Flash Linear Address for program register access
- Flash Descriptor based access control: defines read/write access to each flash region by each master

Firmware may use SPI flash chips write protection(WP#)

PR0 - PR4 defined in SPI MMIO

Lock SPI - BIOS Range is not protected - Threats

- BIOS Write Protections often still not properly enabled on many systems
- SMM based write protection of entire BIOS region is often not used:
BIOS_CONTROL[SMM_BWP]
- If SPI Protected Ranges (mode agnostic) are used (defined by PR0-PR4 in SPI MMIO), they often don't cover entire BIOS & NVRAM
- Some platforms use SPI device specific write protection but only for boot block/startup code or SPI Flash descriptor region

Mitigations:

- Set BIOS_CONTROL[SMM_BWP] <- 1
- Program SPI flash protected ranges (**PRx**) to cover BIOS range

References: [Persistent BIOS Infection](#) (used [flashrom](#) on legacy BIOS), [Evil Maid Just Got Angrier](#), [BIOS Chronomancy](#), [A Tale Of One Software Bypass...](#)

FIRMWARE SECURE UPDATE



Solving Firmware Update

Reliable update story

- Fault tolerant
- Scalable & repeatable

Integrity Check



Reference [6] at : [UEFI](#), [Open Platforms](#) and [ppt](#)

Solving Firmware Update

Reliable update story

- Fault tolerant
- Scalable & repeatable

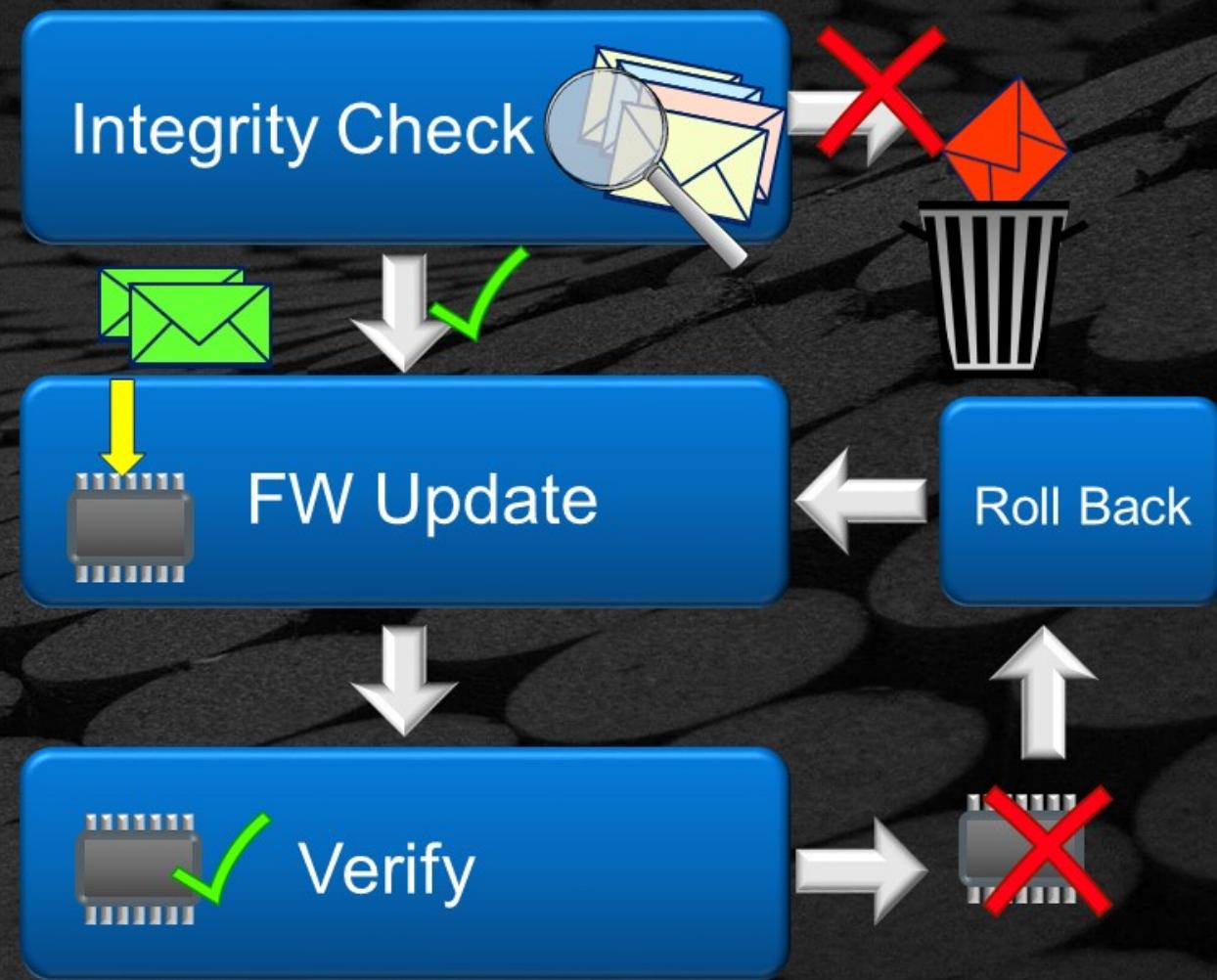


Reference [6] at: [UEFI](#), [Open Platforms](#) and [ppt](#)

Solving Firmware Update

Reliable update story

- Fault tolerant
- Scalable & repeatable



Reference [6] at: [UEFI](#), [Open Platforms](#) and [ppt](#)

Solving Firmware Update

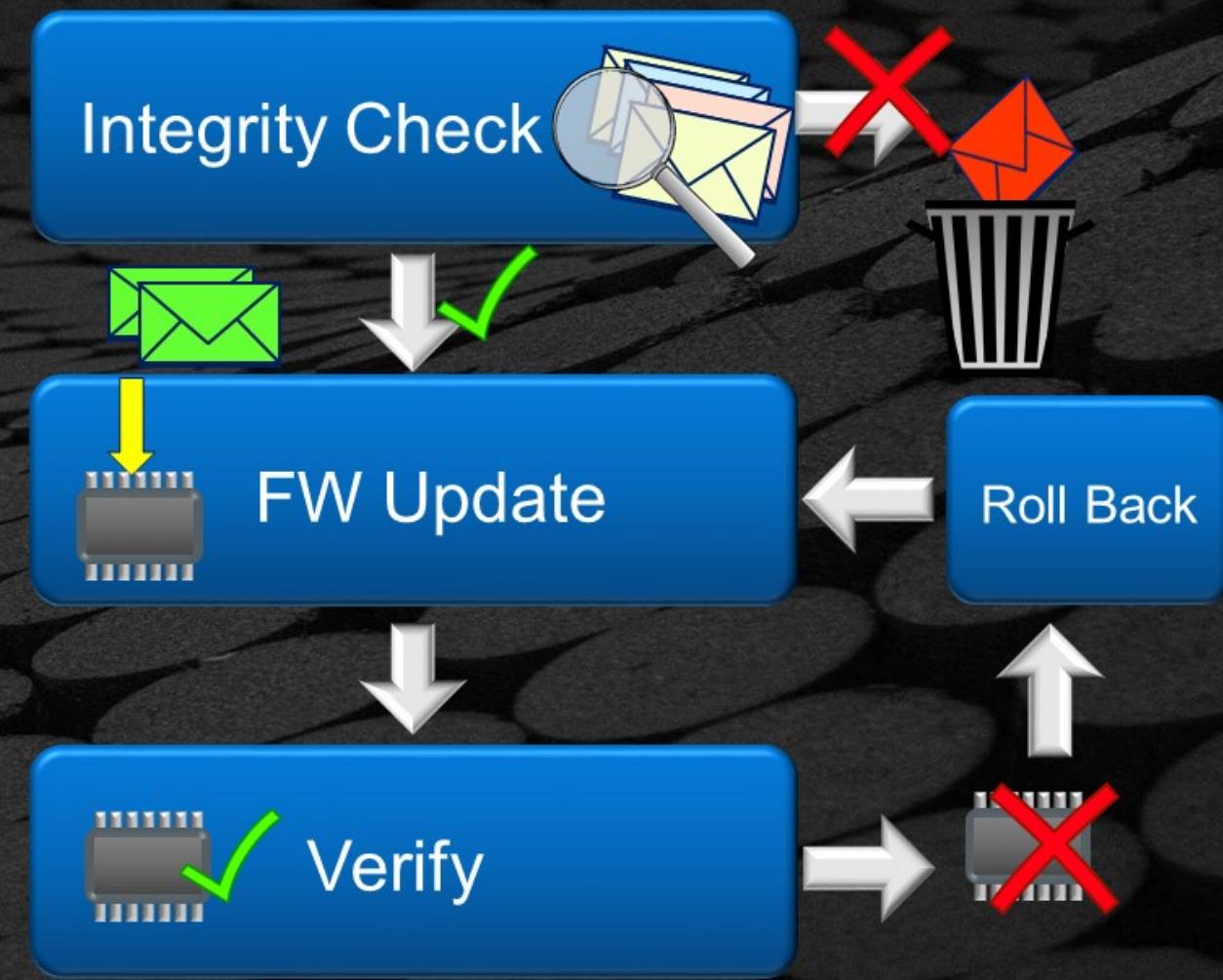
Reliable update story

- Fault tolerant
- Scalable & repeatable

How can UEFI Help?

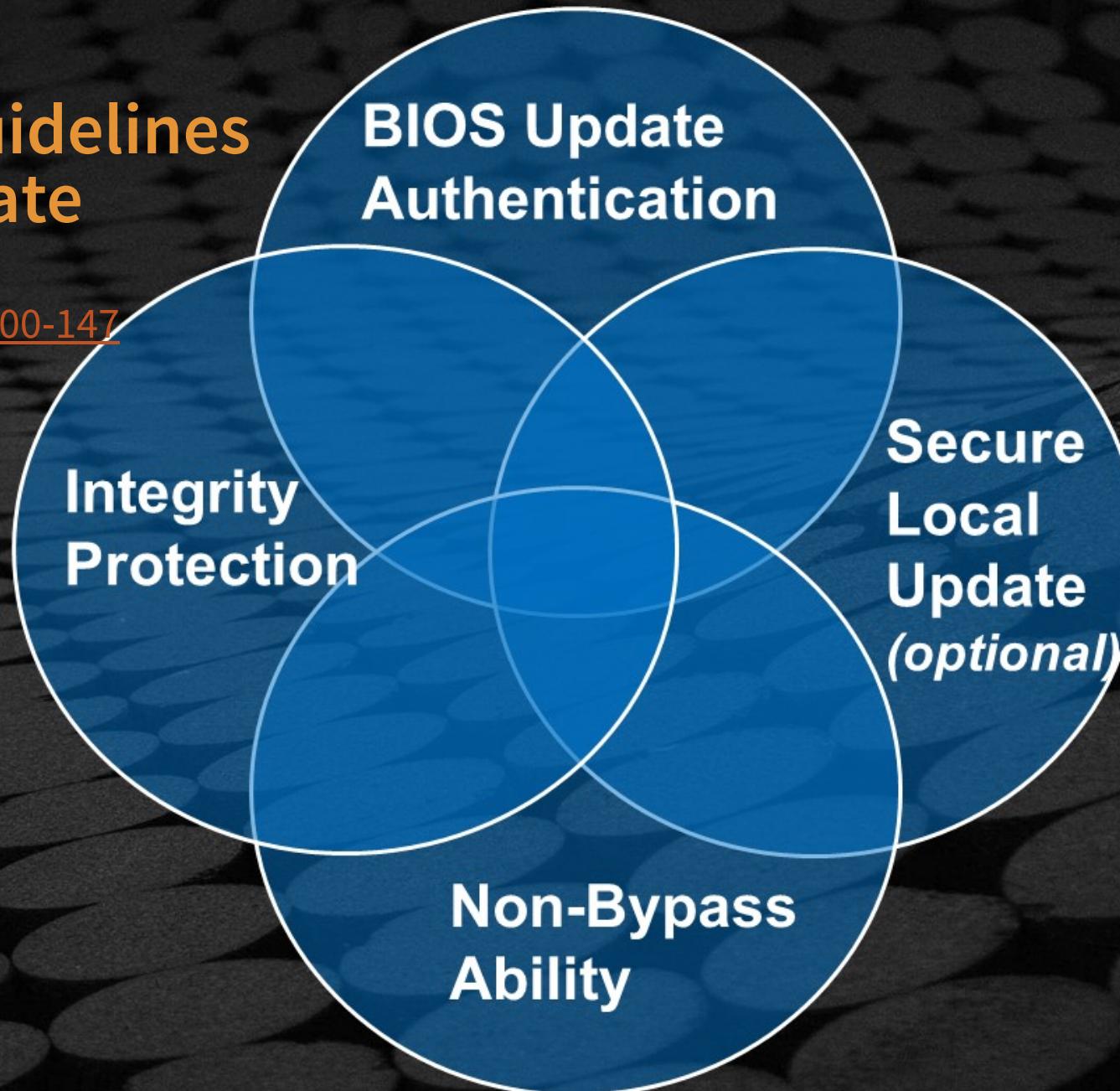
- Capsule model for binary delivery
- Bus / Device Enumeration
- Managing updates via
 - EFI System Resource Table
 - Firmware Management Protocol
 - Capsule Signing

Reference [6] at: [UEFI](#), [Open Platforms](#) and [ppt](#)



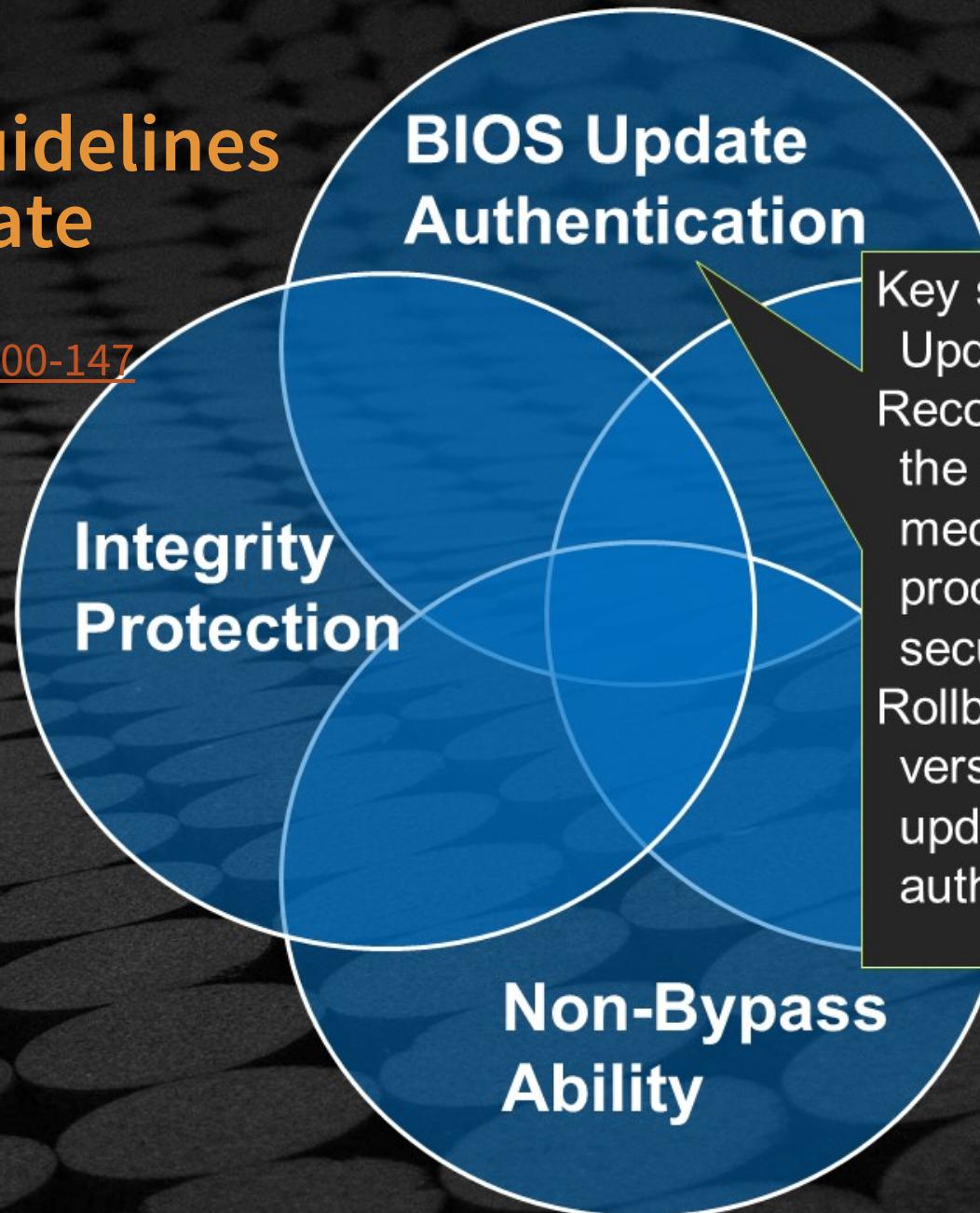
Security Guidelines - BIOS Update

NIST Standard SP 800-147



Security Guidelines - BIOS Update

NIST Standard SP 800-147

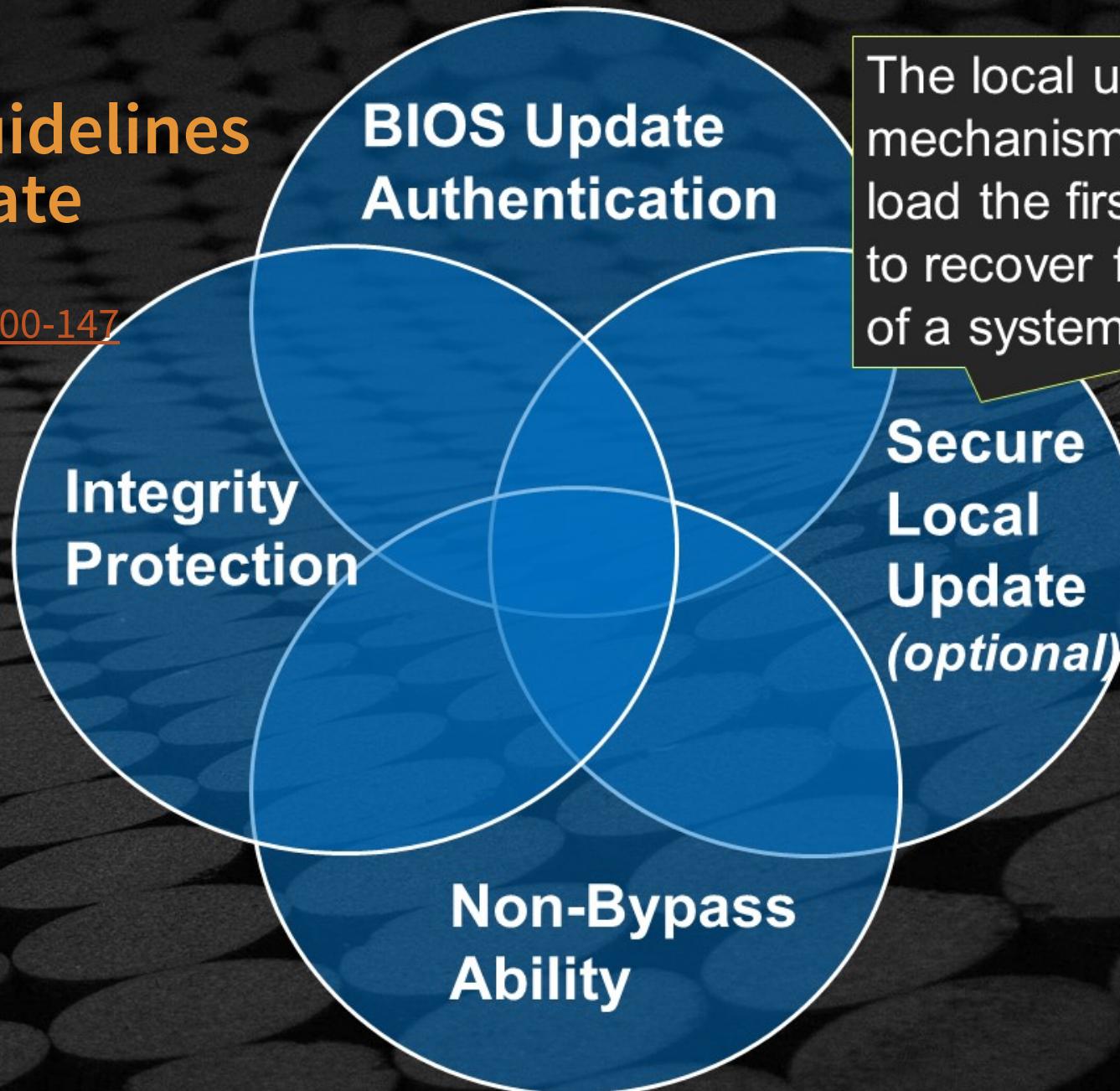


Key storage in Root of Trust for Update (RTU)
Recovery mechanisms shall also use the authenticated update mechanism unless the recovery process meets the guidelines for a secure local update.
Rollbacks of the BIOS to an earlier version are permitted only if the update or rollback has been authorized by the organization.

Recommendations of the National Institute of Standards and Technology

Security Guidelines - BIOS Update

NIST Standard SP 800-147



The local update mechanism be used only to load the first BIOS image or to recover from a corruption of a system BIOS.



Special Publication 800-147

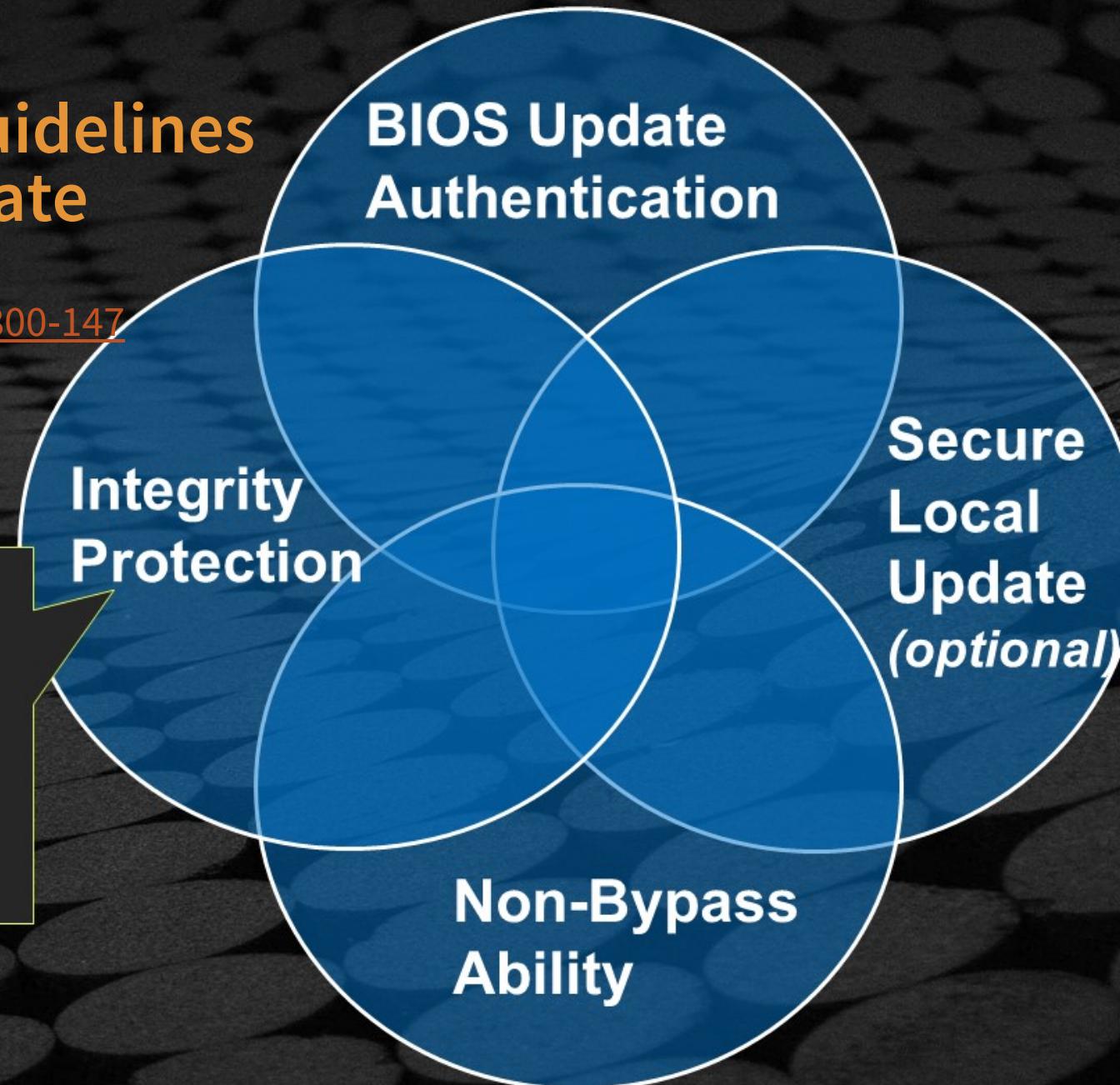
BIOS Protection Guidelines

Recommendations of the National Institute of Standards and Technology

Security Guidelines - BIOS Update

NIST Standard SP 800-147

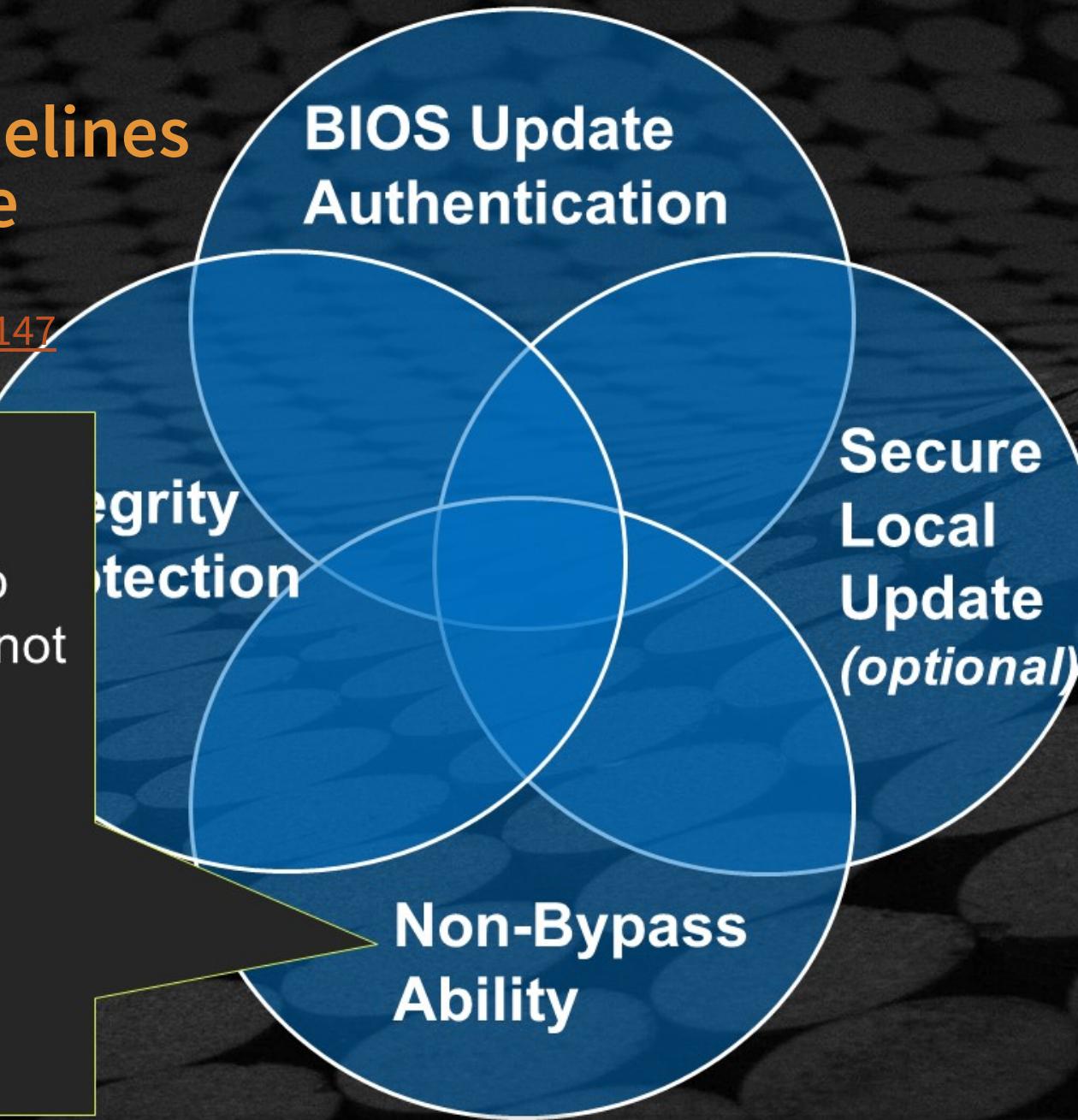
The RTU and the system BIOS shall be protected from unintended modification.



Security Guidelines - BIOS Update

NIST Standard SP 800-147

Bus mastering that bypasses the main processor (e.g., DMA to the system flash) shall not be capable of directly modifying the firmware. Microcontrollers on the system shall not be capable of directly modifying the firmware.



Security Guidelines - BIOS Update

NIST Standard SP 800-147

**Integrity
Protection**

**BIOS Update
Authentication**

**Non-Bypass
Ability**

The RTU and the system BIOS shall be protected from unintended modification.

Bus mastering that bypasses the main processor (e.g., DMA to the system flash) shall not be capable of directly modifying the firmware. Microcontrollers on the system shall not be capable of directly modifying the firmware.

Key storage in Root of Trust for Update (RTU) Recovery mechanisms shall also use the authenticated update mechanism unless the recovery process meets the guidelines for a secure local update. Rollbacks of the BIOS to an earlier version are permitted only if the update or rollback has been authorized by the organization.

The local update mechanism be used only to load the first BIOS image or to recover from a corruption of a system BIOS.

**Secure
Local
Update
(optional)**

NIST
National Institute of
Standards and Technology
U.S. Department of Commerce

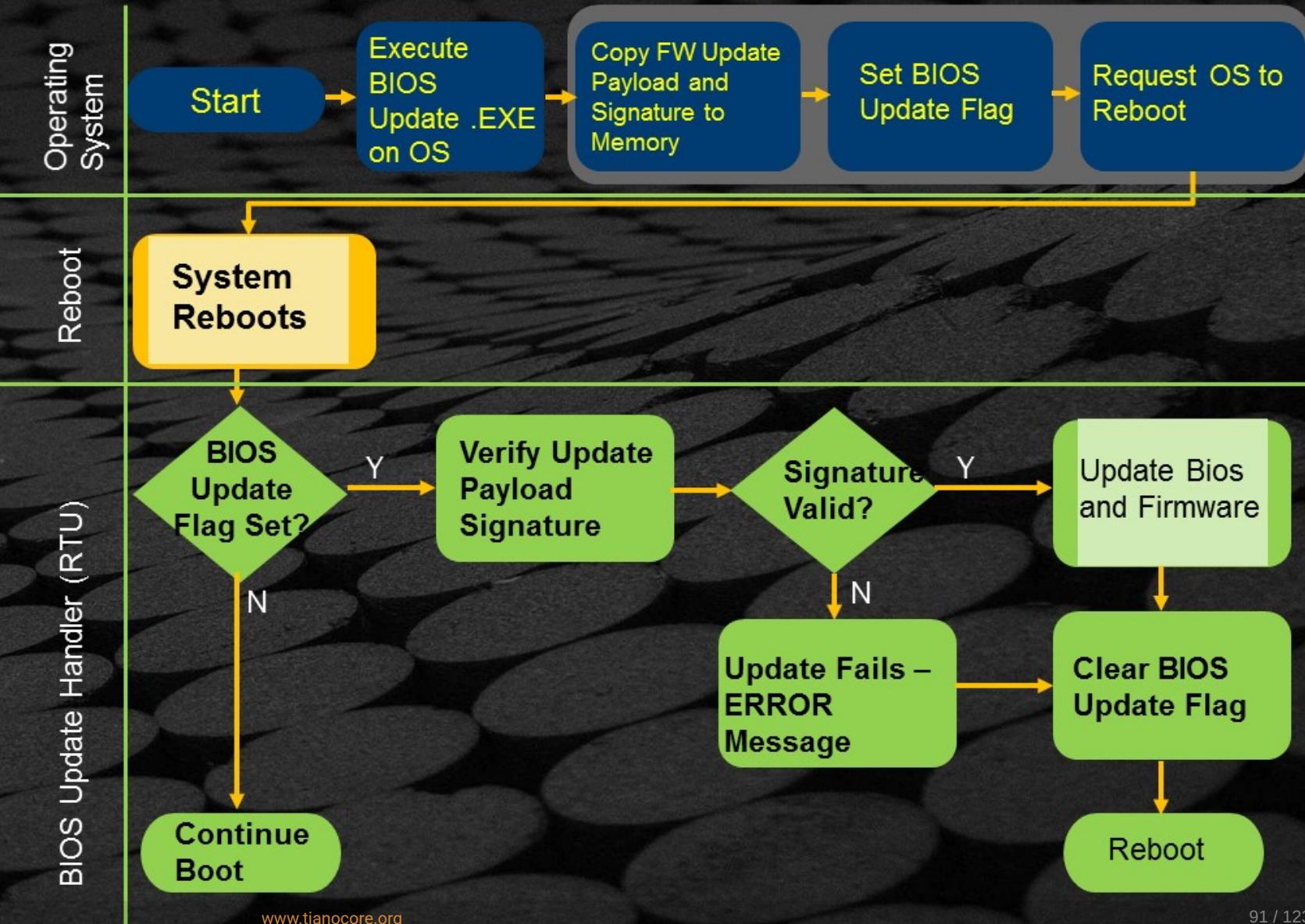
Special Publication 800-147

BIOS Protection Guidelines

Recommendations of the National Institute of Standards and Technology

Signed Firmware Update

- RTU protected by flash locking mechanisms at the hardware level
- BIOS key store includes the full public key used to verify the signature of all BIOS and firmware updates
- Capsule Update with UEFI FMP



RTU - BIOS Root of Trust for Update
 FMP - Firmware Management Protocol

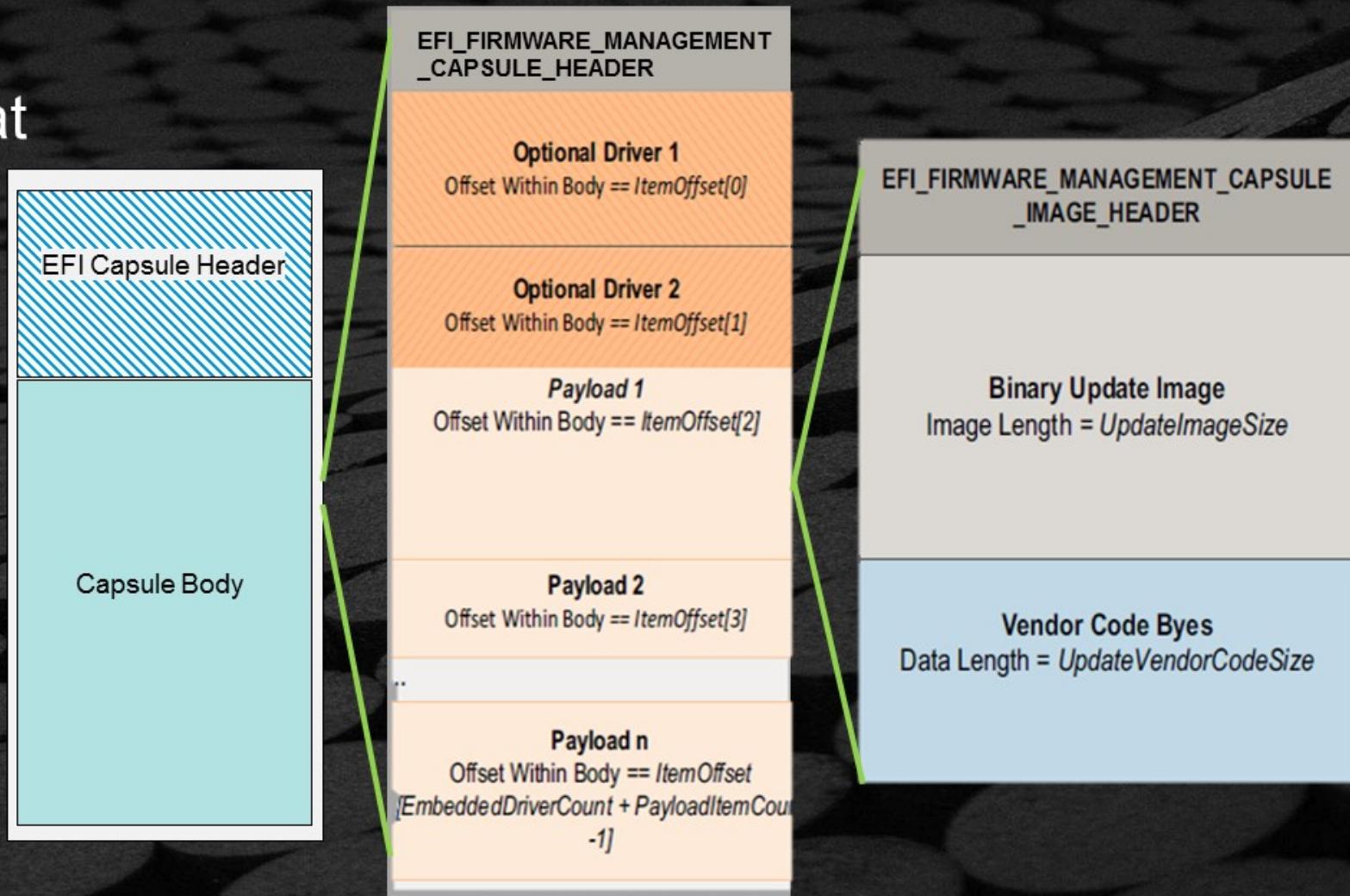
UEFI Capsule Update – Firmware Management Protocol (FMP)

FMP capsule image format

- Update FMP drivers
- FMP payloads

binary update image and optional vendor code

The platform may consume a FMP protocol to update the firmware image



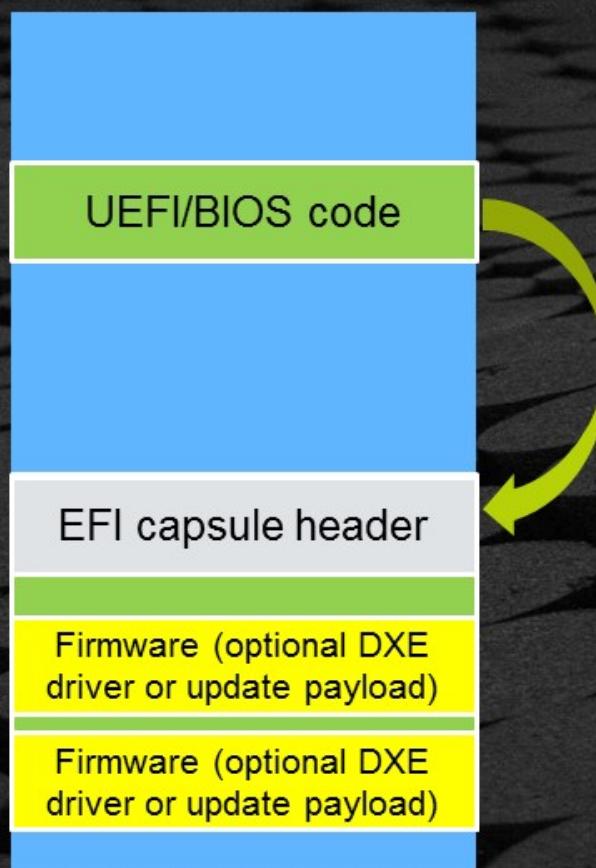
UEFI Firmware Secure “Capsule” Update

Capsule update is a runtime service used to update UEFI FW

0xFFFFFFFF

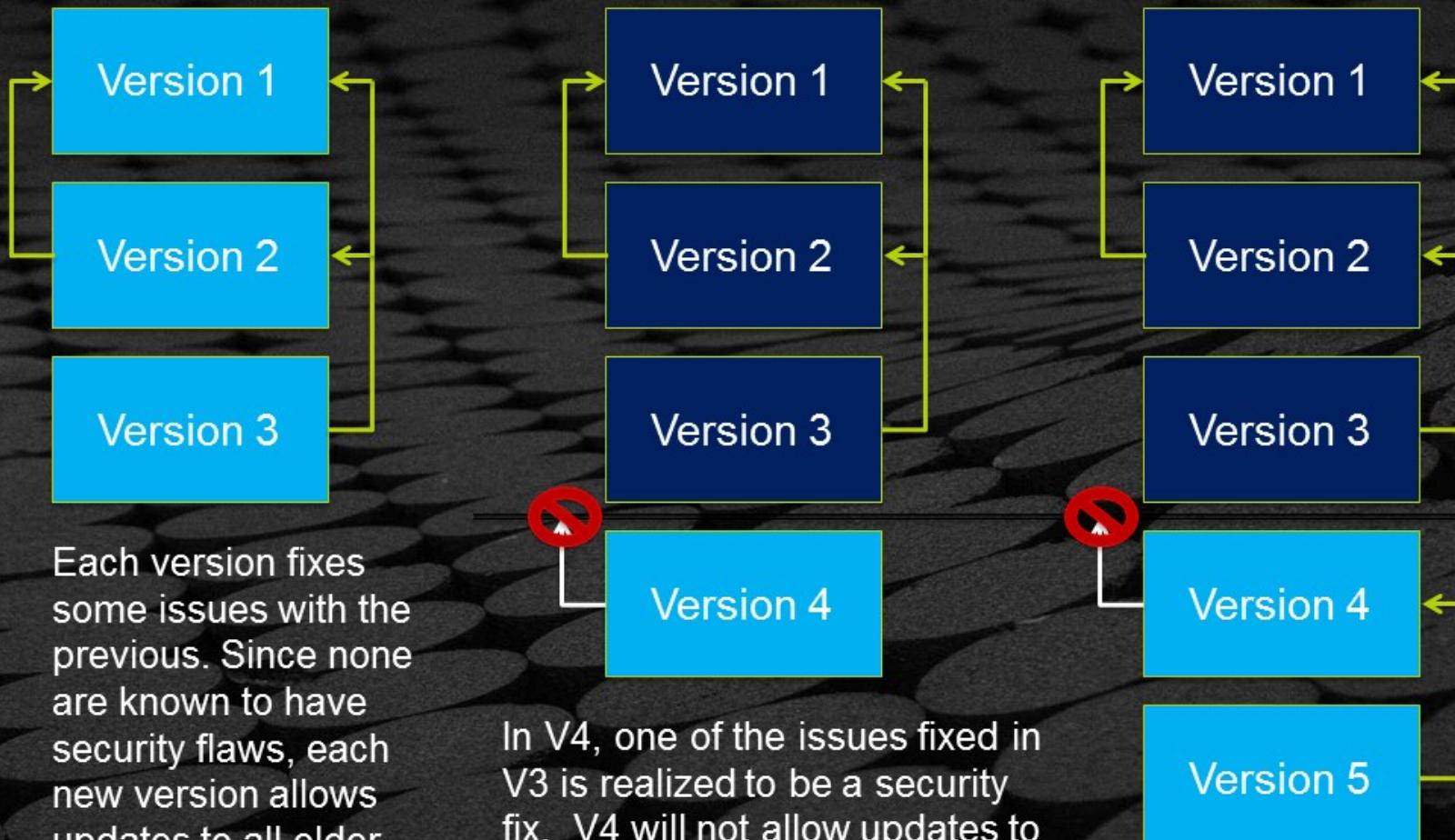
Update capsule

0x00000000



1. Update is initiated by update application/OS run-time
2. Update application stores update “capsule” in DRAM or HDD on ESP (e.g. **\EFI\CapsuleUpdate**)
3. Upon reboot or S3 resume, FW finds and parses update capsule
4. After FW verifies digital signature of the capsule, FW writes new BIOS FV(s) to SPI flash memory

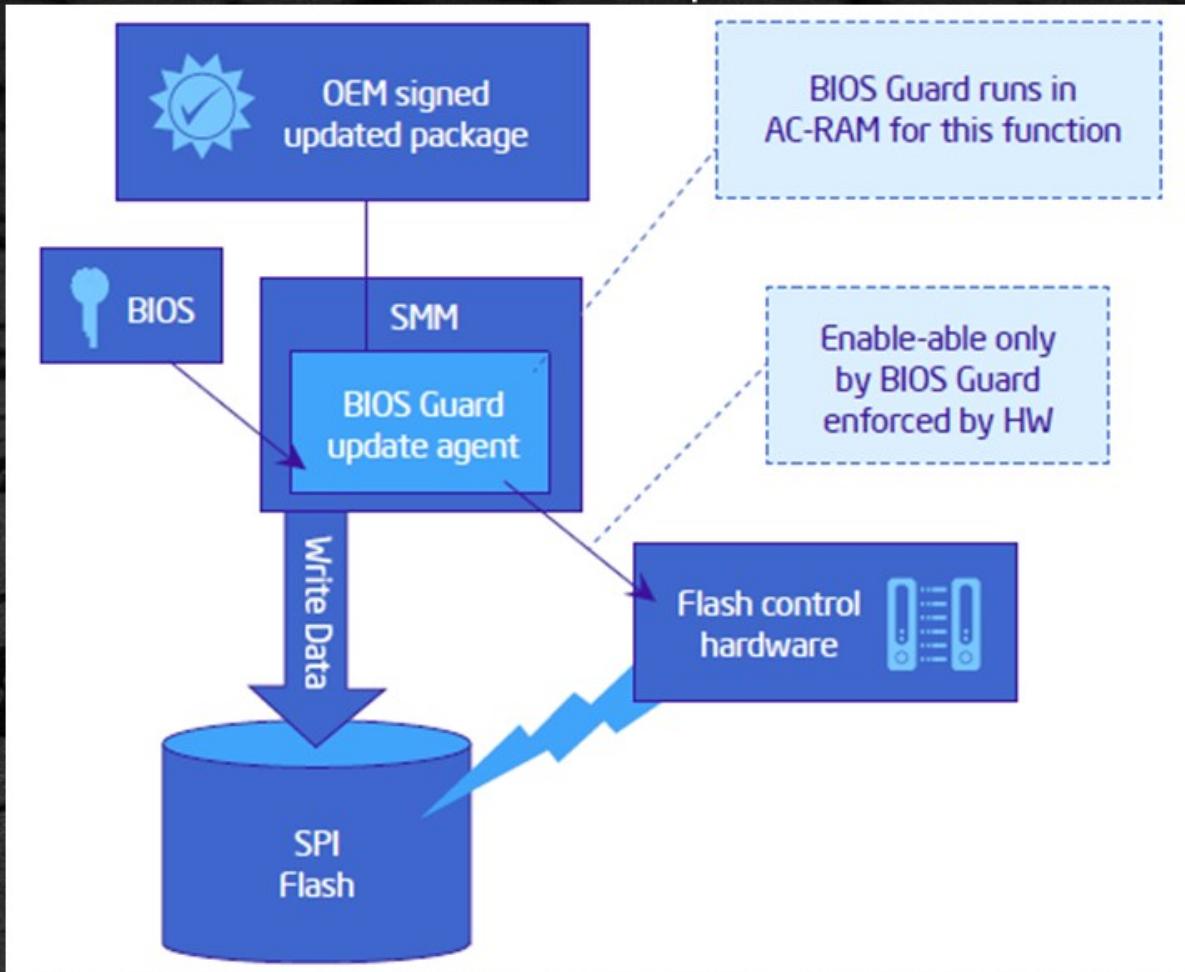
Firmware Update Rollback Protection



Hardware based System Firmware Update

BIOS Guard address SMM vulnerabilities by strengthening the update trust boundary

Example: Intel® BIOS Guard



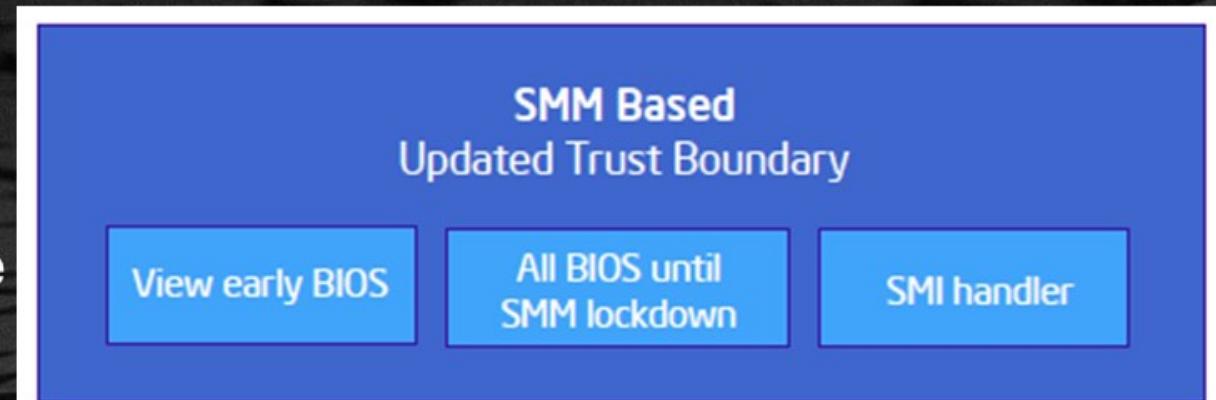
SMM BIOS Update Trust Boundary

- For runtime BIOS Update (e.g. on server platforms), all complex SMI handlers code is in the trust boundary of the firmware update
- Different systems have different SMI handlers which makes it difficult to ensure consistent security level of SMI code across all system and security level of firmware update
- BIOS Guard reduces SMI handler attack surface, using one signed BIOS Guard Authenticated Code Module (ACM)
- Platforms enabling BIOS Guard only need to use one module for a given processor generation

BIOS Guard Based Firmware Update

- BIOS Guard can update contents of the BIOS region in system SPI flash and EC firmware on EC flash memory
- BIOS Guard module is Authenticated Code Module (ACM) executing in internal processor AC RAM
- When BIOS Guard is enabled, only BIOS Guard module is able to write to system SPI flash memory
- BIOS Guard verifies the signature of a firmware update package signed by a platform manufacturer prior to writing to system SPI flash memory

Trust Boundary with BIOS Guard



EC – Embedded Controller

When Is Secure Boot Actually Secure?

When all platform manufacturers...

- protect the UEFI BIOS from programmable SPI writes by malware,
- allow only signed UEFI BIOS updates,
- protect authorized update software,
- correctly program and protect SPI Flash descriptor,
- protect Secure Boot persistent configuration variables in NVRAM,
- implement authenticated variable updates,
- protect variable update API,
- disable Compatibility Support Module (Legacy BIOS),
- don't allow unsigned legacy Option ROMs,
- configure secure image verification policies,
- ...

= and don't introduce a single bug in all of this, of course.

Platform Firmware Security – Why is it important?

Why is platform firmware Security important

→ Prevent low level attacks that could “brick” the system

UEFI boot flow with the threat model

→ Identify where UEFI firmware is vulnerable and define a Threat Model

Security technologies overview

→ Boot Guard, Secure Boot and NIST Secure Updates provide mitigations to some hacking methods

TOOLS AND RESOURCES

How to test firmware for security



AUTOMATION:

Automated Test Generation Tools



Build a Security Mindset

“Finding BIOS Vulnerabilities with Symbolic Execution and Virtual Platforms”: [Excite Link](#)



What is Excite?

Goal

- Automatically detect security issues in UEFI BIOS
 - Complement other approaches: security code reviews, static analysis, Chipsec ...

Approach

- Apply open source **fuzzing & symbolic execution** tools to Intel Architecture binaries

Versions

- **Excite** : Symbolic Execution using CRETE and Simics
- **Excite DBI** (Dynamic Binary Instrumentation): Fuzzing using AFL and Intel Pin Tool

AFL - American Fuzzy Lop (fuzzer)

Pin - Intel tool to determine trace

CRETE - [github crete-dev](https://github.com/tianocore/crete-dev)

What is Fuzzing?

Technique of providing variety of invalid, unexpected, or random data as inputs to target code to try to make it misbehave

- Some inputs are non-intuitive, harder for developers to discover

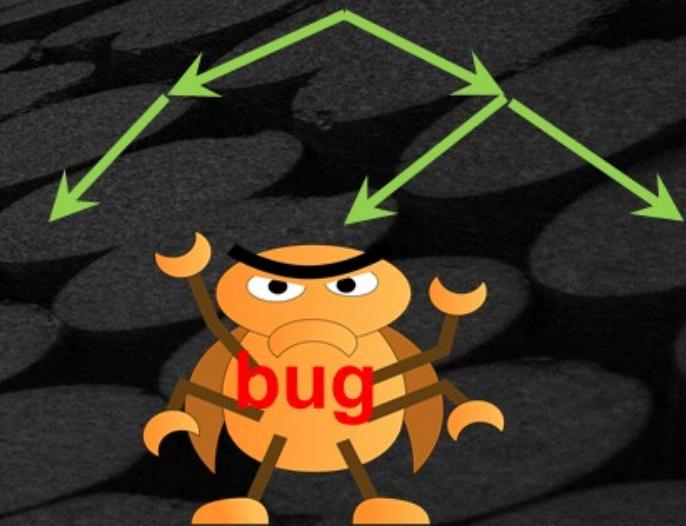
Why fuzz UEFI?

- Find security issues early
 - Complement manual code reviews
 - Re-run as software evolves
 - New vulnerability detection capability

Symbolic Execution (S2E)- SMM Example

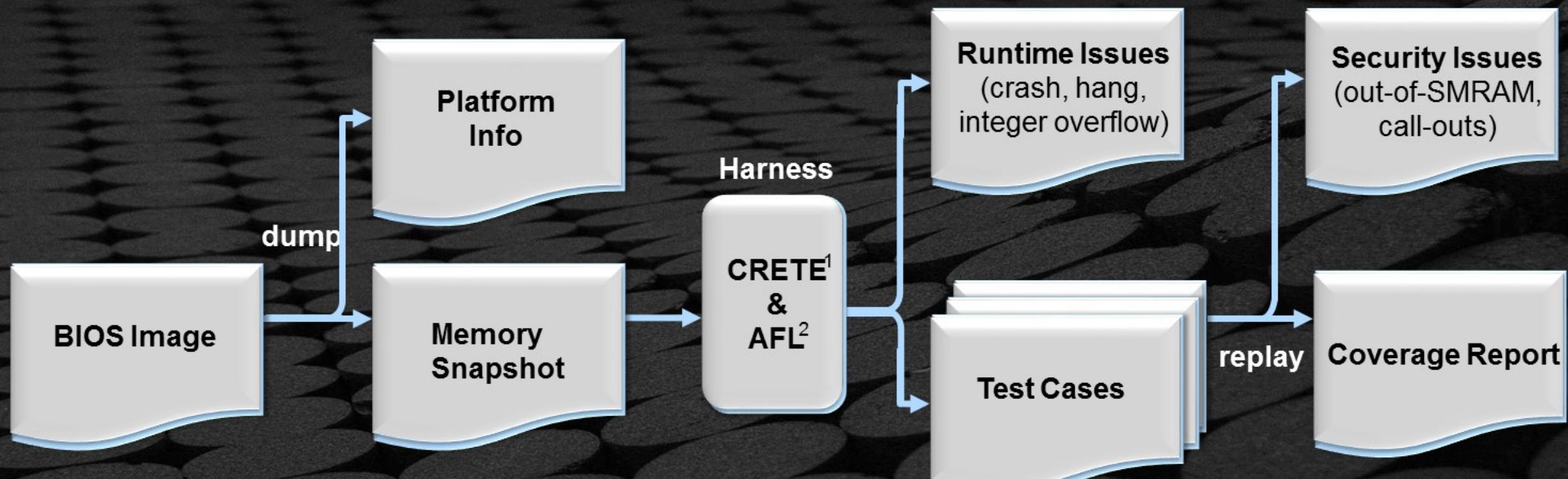
- Searching for SMM security vulnerabilities
- Symbolic execution generates test cases that cover the computation tree of the handler and induce vulnerabilities.
- Those test cases are replayed in Simics which detects illegal accesses and SMM callouts. (e.g. read memory outside SMRAM)

SmiHandler
(SymbolicData)



Symbolic execution for BIOS security
[woot15-paper-bazhaniuk PDF](#)

Excite - Common Flow



¹ Selective Symbolic Execution (S²E | CRETE)

² American Fuzzy Lop (fuzzer)

Extend fuzzing beyond interface level

- UEFI code (SMM, STM, UEFI PE modules) accessible to AFL/CRETE, Pin, gdb.
- Record/analyze code flow, call trace, memory access data
- Individual functions accessible, new inputs created, can test different flows

Quantify code coverage reporting



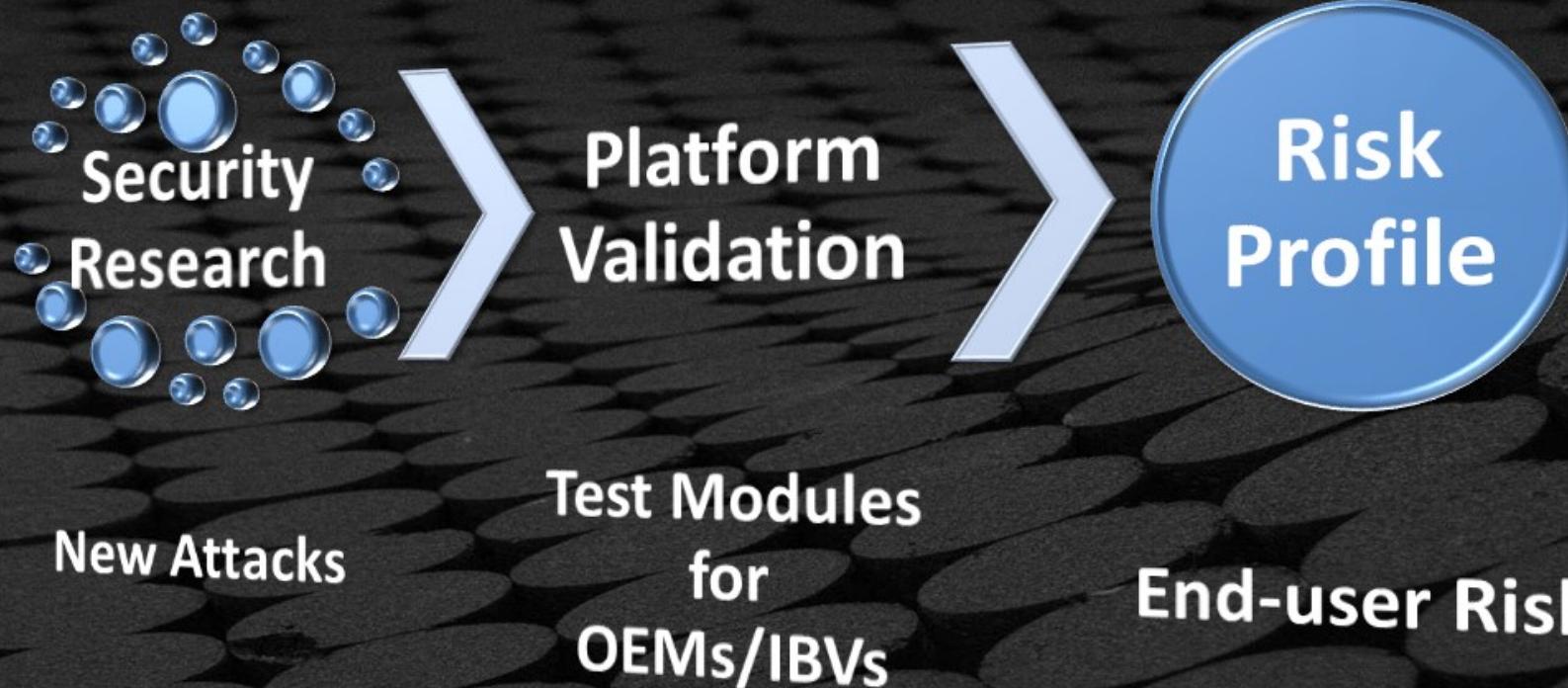
Resource: [STM-SMI Transfer Monitor](#)

KNOWN ISSUE:

Tools for Testing Known Issue



Raising the Bar for Platform Security



Empowering End-Users to Make a Risk Decision

What is CHIPSEC?



CHIPSEC
Platform Security
Assessment Framework

Framework for Platform Security Assessment

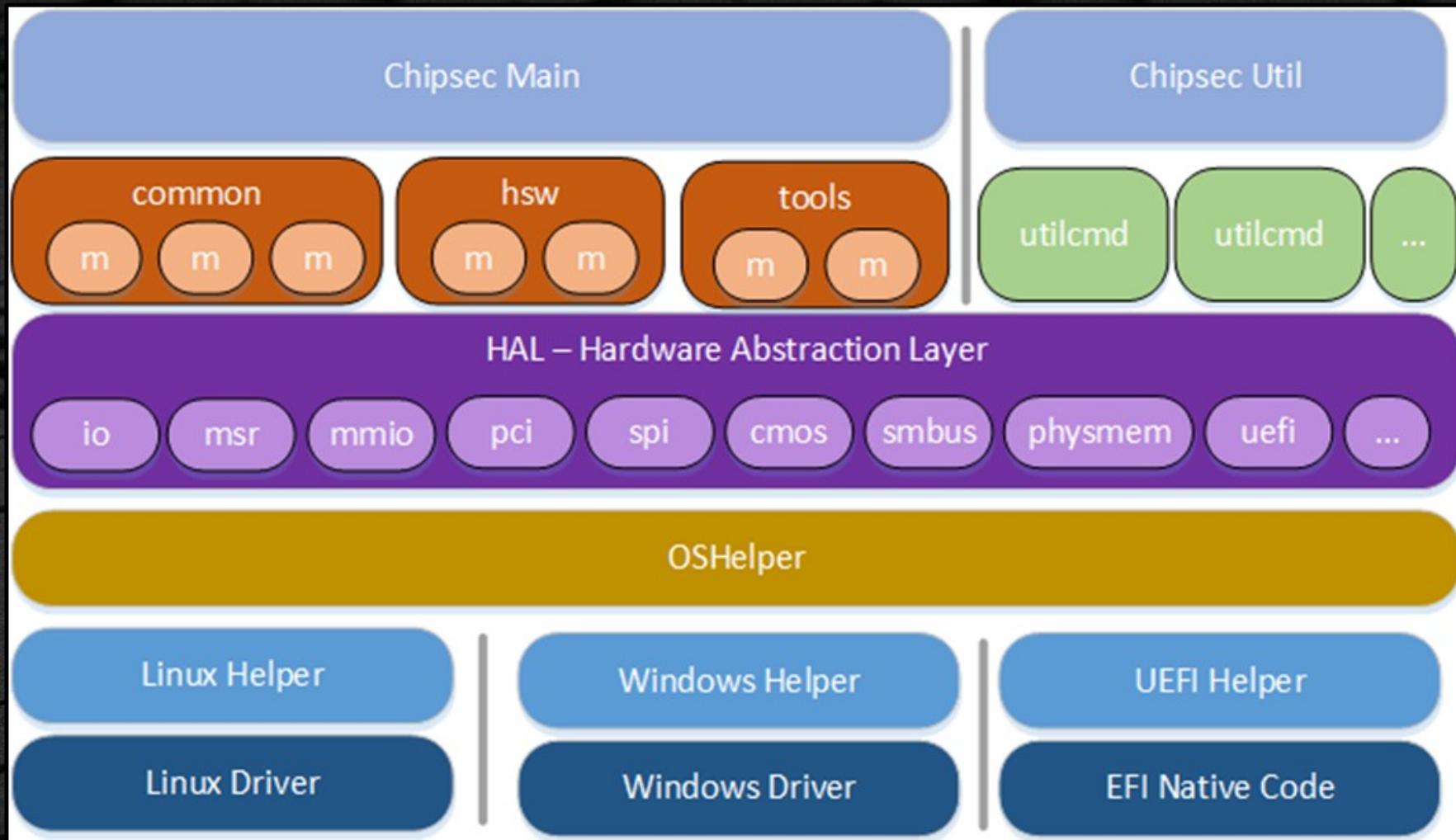
- Tests for known security issues
- Tools for investigation of platform properties

Open Source (GPLv2 License)

- Released in 2014 –  <https://github.com/chipsec/chipsec>
- Currently active community
- Top referring sites: Google, Github, pcworld.com, community.spiceworks.com, anonhq.com, Twitter, securingtomorrow.mcafee.com, intelsecurity.com, cylance.com

The screenshot shows the GitHub repository page for 'chipsec / chipsec'. The header includes the repository name, a fork button, and a star count of 190. Below the header are tabs for Code, Issues (6), Pull requests (10), Projects (0), Wiki, and Insights. The main content area has a heading 'Home' with a note that 'CHIPSEC edited this page on Mar 2, 2017 - 12 revisions'. It features a section titled 'Welcome to the chipsec wiki!' followed by a 'Platform Security Assessment' section which links to 'Platform Firmware Security Assessment with CHIPSEC - CanSecWest 2014' and '[Vulnerabilities and CHIPSEC Modules]'. There is also a 'Getting Started' section with a link to 'Creating a Bootable USB with CHIPSEC'.

Chipsec Framework overview



Example Chipsec Check - SPI Controller LOCK

- FLOCKDN – Reg that Locks critical SPI controller registers
 - Flash Region Access Permissions
 - Secondary Flash Region Access Permissions
 - Flash Protected Range Registers
 - Other Settings
- Without setting this bit some SPI protections may be modified or bypassed
- Chipsec provides the common .spi_lock test to verify this setting

≡ FLOCKDN – Flash Configuration Lock down Reg

Sample Issue Discovered

Is BIOS correctly protected?

```
[*] running module: chipsec.modules.common.spi_lock
[x] =====
[x] Module: SPI Flash Controller Configuration Lock
[x] =====
[*] HSFS = 0x6008 << Hardware Sequencing Flash Status Register (SPIBAR + 0x4)
[00] FDONE          = 0 << Flash Cycle Done
[01] FCERR          = 0 << Flash Cycle Error
[02] AEL             = 0 << Access Error Log
[03] BERASE          = 1 << Block/Sector Erase Size
[05] SCIP             = 0 << SPI cycle in progress
[13] FDOPSS          = 1 << Flash Descriptor Override Pin-Strap Status
[14] FDV              = 1 << Flash Descriptor Valid
[15] FLOCKDN         = 0 << Flash Configuration Lock-Down
[-] FAILED: SPI Flash Controller configuration is not locked
```

[-] FAILED: SPI Flash Controller configuration is not locked

Known Threats and Chipsec modules



Issue	CHIPSEC Module	References
SMRAM Locking	common.smm	CanSecWest 2006
BIOS Keyboard Buffer Sanitization	common.bios_kbrd_buffer	DEFCON 16
SMRR Configuration	common.smrr	ITL 2009 , CanSecWest 2009
BIOS Protection	common.bios_wp	BlackHat USA 2009 , CanSecWest 2013 , Black Hat 2013 , NoSuchCon 2013 ,
SPI Controller Locking	common.spi_lock	Flashrom , Copernicus
BIOS Interface	common.bios_ts	PoC 2007

Known Threats and Chipsec modules



Issue	CHIPSEC Module	References
BIOS Interface Locking	common.bios_ts	PoC 2007
Secure Boot variables with keys and configuration are protected	common.secureboot.variables	UEFI 2.4 Spec , All Your Boot Are Belong To Us (here & here)
Memory remapping attack	remap	Preventing and Detecting Xen Hypervisor Subversions
DMA attack against SMRAM	smm_dma	Programmed I/O accesses: a threat to VMM? , System Management Mode Design and Security Issues
SMI suppression	common.bios_smi	Setup for Failure: Defeating Secure

Known Threats and Chipsec modules



Issue	CHIPSEC Module	References
SMI suppression attack	common.bios_smi	<u>Setup for Failure: Defeating Secure Boot</u>
ETC ...		See <u>github.com/chipsec/</u>

CHIPSEC Modules

Modules encapsulate the main functionality of CHIPSEC:

1. Tests for known vulnerabilities in firmware
2. Tests for insufficient or incorrectly configured hardware protections
3. Hardware/firmware-level security tools
 - Fuzzing tools for firmware interfaces/formats
 - Manual security checkers (e.g. TE checker, DMA dumper)
 - Reside in modules/tools directory are not launched automatically (only through -m command-line option)
 - PoC exploit modules demonstrating vulnerabilities

CHIPSEC Modules

- All modules reside in `chipsec/modules` directory
- Modules can be specific to one or more platforms or common for all supported platforms
 - Modules in `modules/<platform_code>` directory will only be executed on `<platform_code>` platform
 - Modules in `modules/common` directory will always be executed
- Modules can implement `is_supported` function which can further check for supported platforms, OS environments (legacy vs. UEFI boot), etc.

Platform Firmware Security – Why is it important?

Why is platform firmware Security important

Prevent low level attacks that could “brick” the system

UEFI boot flow with the threat model

Identify where UEFI firmware is vulnerable and define a Threat Model

Security technologies overview

Boot Guard, Secure Boot and NIST Secure Updates provide mitigations to some hacking methods

Tools and resources on how to test firmware for security

Ongoing tools to validate security mechanism

SUMMARY

- Why is platform firmware Security important
- UEFI boot flow with the threat model
- Security technologies overview
- Tools and resources on how to test firmware for security





tianocore



ACKNOWLEDGEMENTS

/**

Redistribution and use in source (original document form) and 'compiled' forms (converted to PDF, epub, HTML and other formats) with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code (original document form) must retain the above copyright notice, this list of conditions and the following disclaimer as the first lines of this file unmodified.

Redistributions in compiled form (transformed to other DTDs, converted to PDF, epub, HTML and other formats) must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS DOCUMENTATION IS PROVIDED BY TIANOCORE PROJECT "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL TIANOCORE PROJECT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE OF DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.