All Your Boot Are Belong To Us

CanSecWest 2014

Intel Security

Yuriy Bulygin, Andrew Furtak, Oleksandr Bazhaniuk, John Loucaides

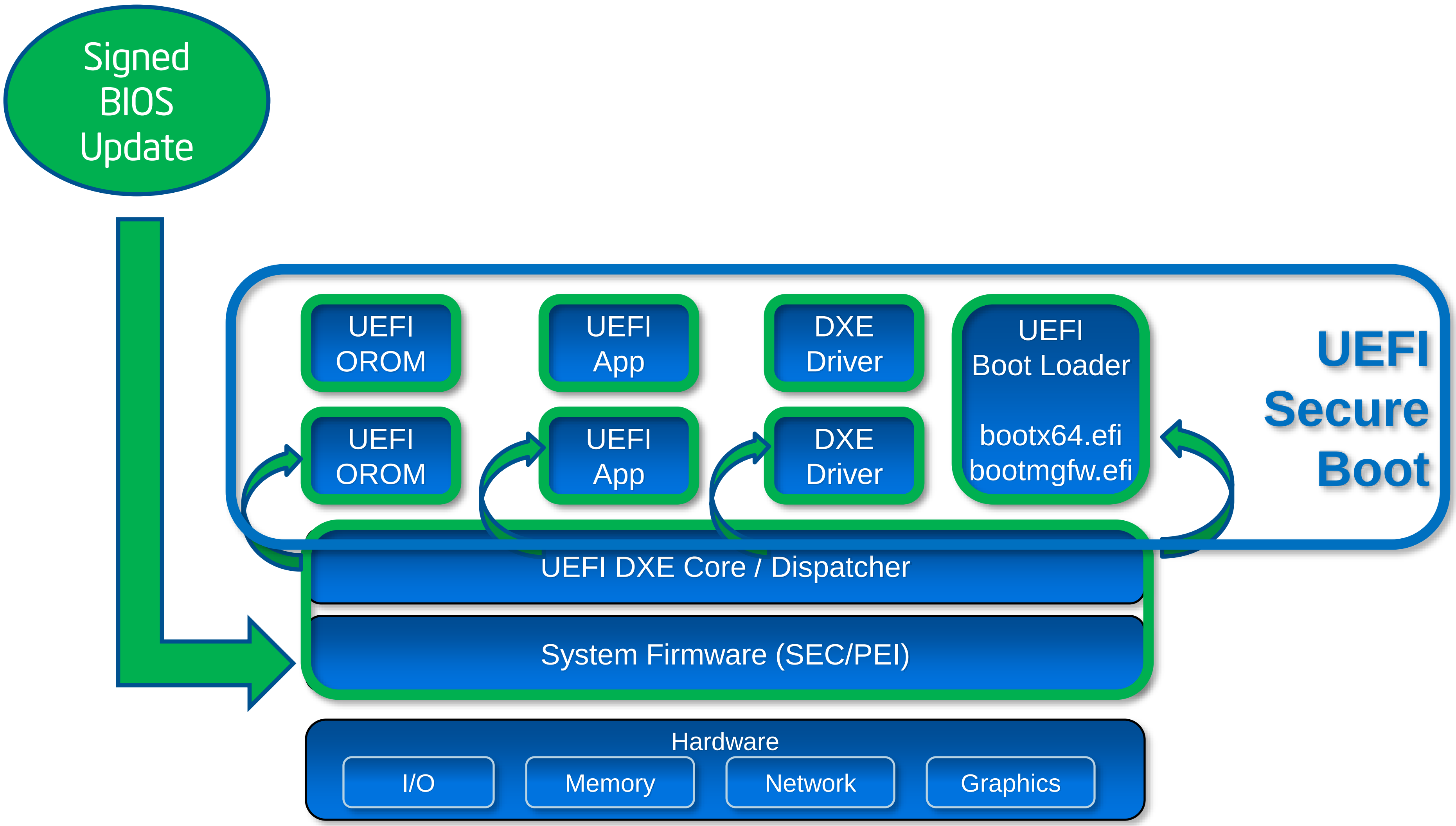
UEFI Secure Boot

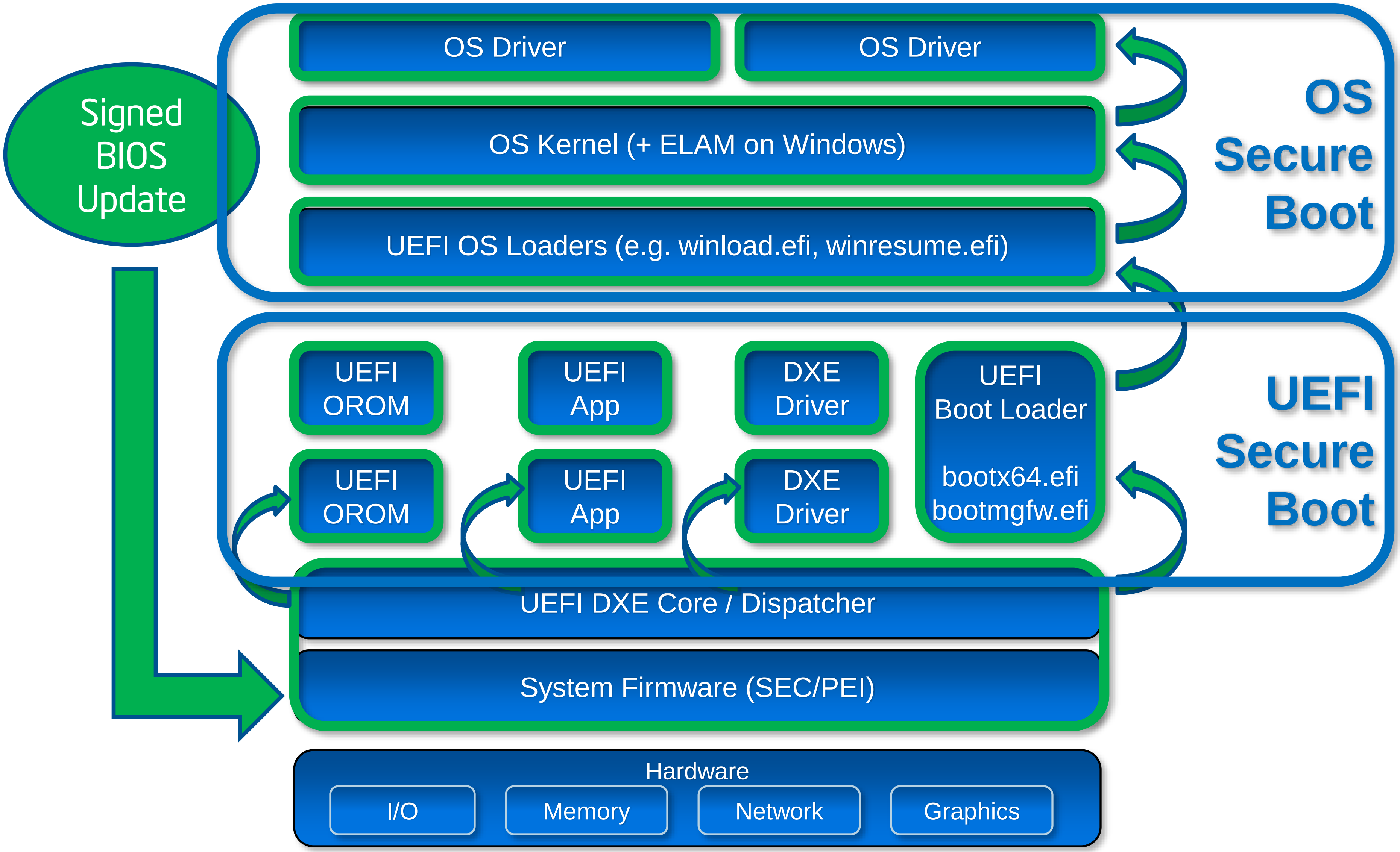
UEFI Secure Boot

UEFI has largely replaced conventional BIOS for PC platform firmware on new systems.

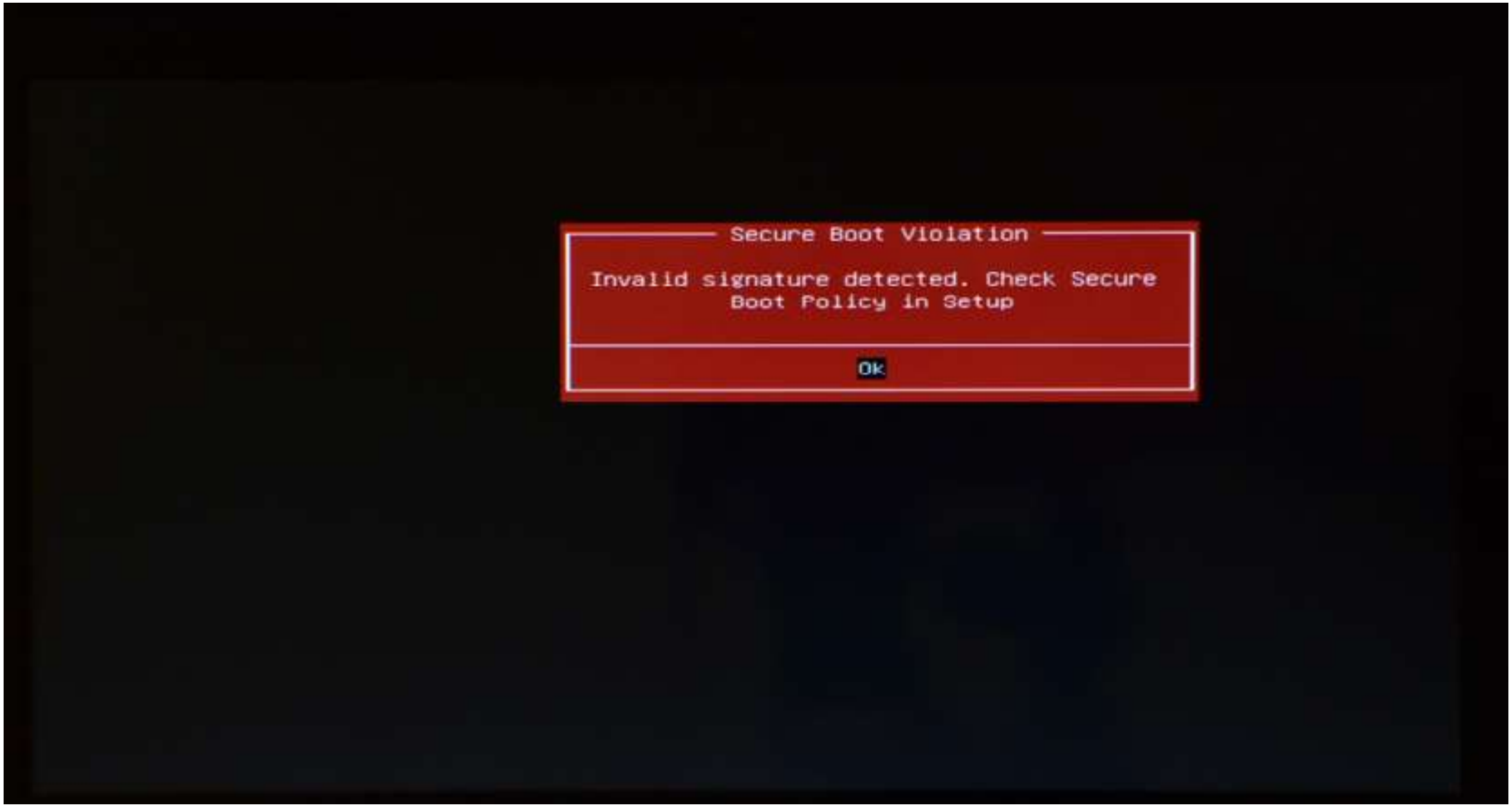
UEFI 2.3.1 specified a new security feature "Secure Boot" intended to protect UEFI based systems from bootkits which were affecting systems with legacy BIOS/OS boot.

When enabled, Secure Boot validates the integrity of the operating system boot loader before transferring control to it.


Signed BIOS Update
UEFI OROM
UEFI App
DXE Driver
UEFI Boot Loader
bootx64.efi
bootmgfw.efi
UEFI OROM
UEFI App
DXE Driver
UEFI Secure Boot
UEFI DXE Core / Dispatcher
System Firmware (SEC/PEI)
Hardware
I/O
Memory
Network
Graphics


OS Driver
OS Driver
OS Kernel (+ ELAM on Windows)
UEFI OS Loaders (e.g. winload.efi, winresume.efi)
OS Secure Boot
Signed BIOS Update
UEFI OROM
UEFI App
DXE Driver
UEFI Boot Loader
bootx64.efi
bootmgfw.efi
UEFI OROM
UEFI App
DXE Driver
UEFI Secure Boot
UEFI DXE Core / Dispatcher
System Firmware (SEC/PEI)
Hardware
I/O
Memory
Network
Graphics

UEFI Secure Boot in Action



MITRE

First Issue

MITRE's research fit nicely with research and guidance we were already coordinating.

All of this guidance has been shared previously with BIOS vendors and platform manufacturers.

This guidance resulted from analysis of the BIOS implementations on specific systems. We did not perform analysis on all systems.

Unprotected Secure Boot Enable/Disable Control a.k.a. “Quest For Disabling Secure Boot”

Recommendations

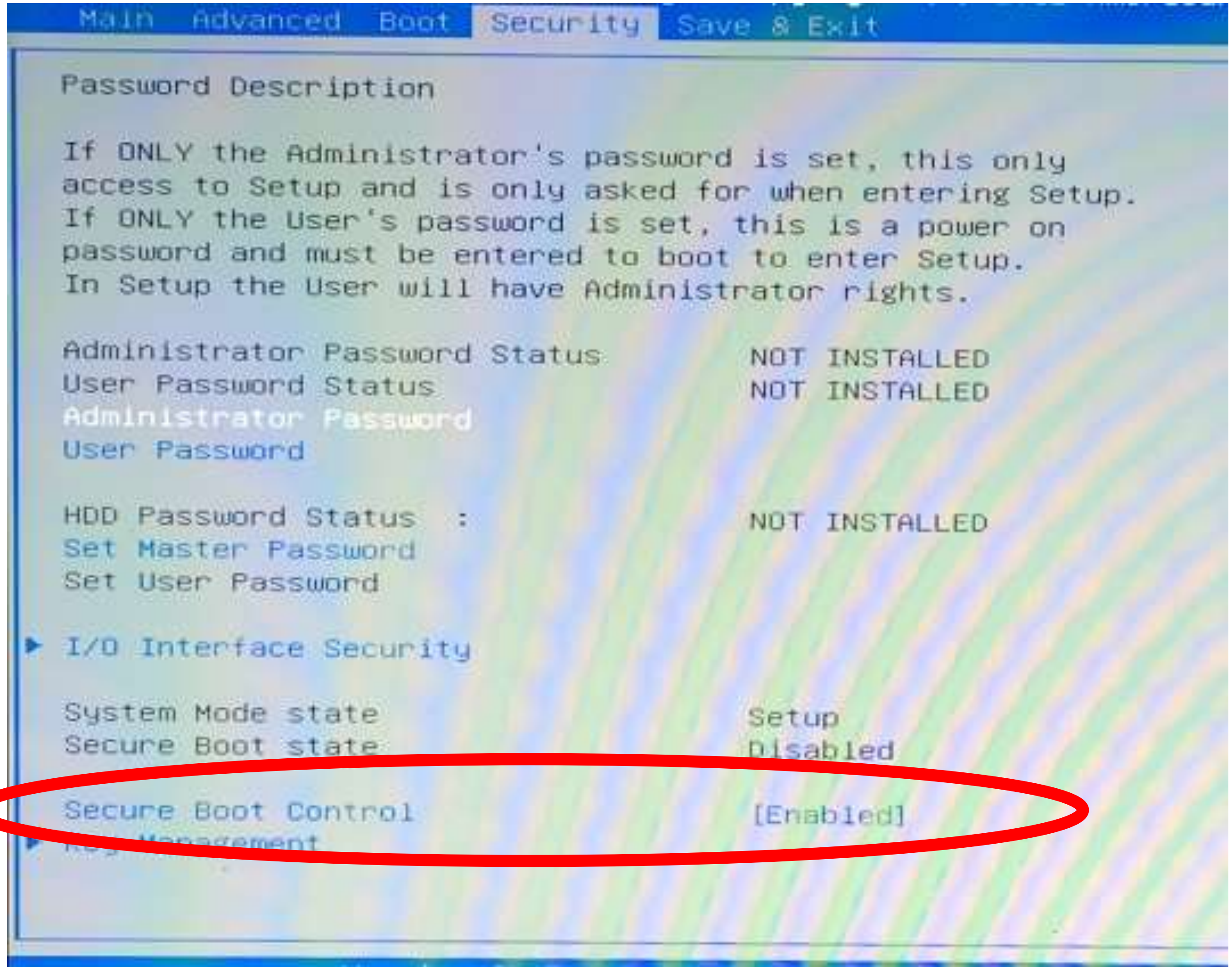
Protect Secure Boot Enable/Disable Control

- Don’t store it in places writeable by malware (like RUNTIME_ACCESS UEFI Variables)
- Use “SecureBootEnable” UEFI Variable defined in edk2
- Require user physical presence to change

CHIPSEC test for this

```
chipsec_main.py -m tools.secureboot.te
```

Turn On/Off Secure Boot in BIOS Setup



Secure Boot Enable/Disable Control

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

```
chipsec_util.py spi dump spi.bin
```

- Then compare two SPI flash images? Good luck with that ;)

Secure Boot Enable/Disable in Setup

Found! It really is in BIOS Setup → in 'Setup' UEFI Variable

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 uefi nvram
chipsec_util.py decode
```

Demo

PE/TE Header Confusion
a.k.a. PETE

Recommendations

Don't skip checks on EFI executables with TE header

- Don't skip Secure Boot checks on EFI executables with TE Header
- Beware of customizations to open source **`DxeImageVerificationLib`**

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 then 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:

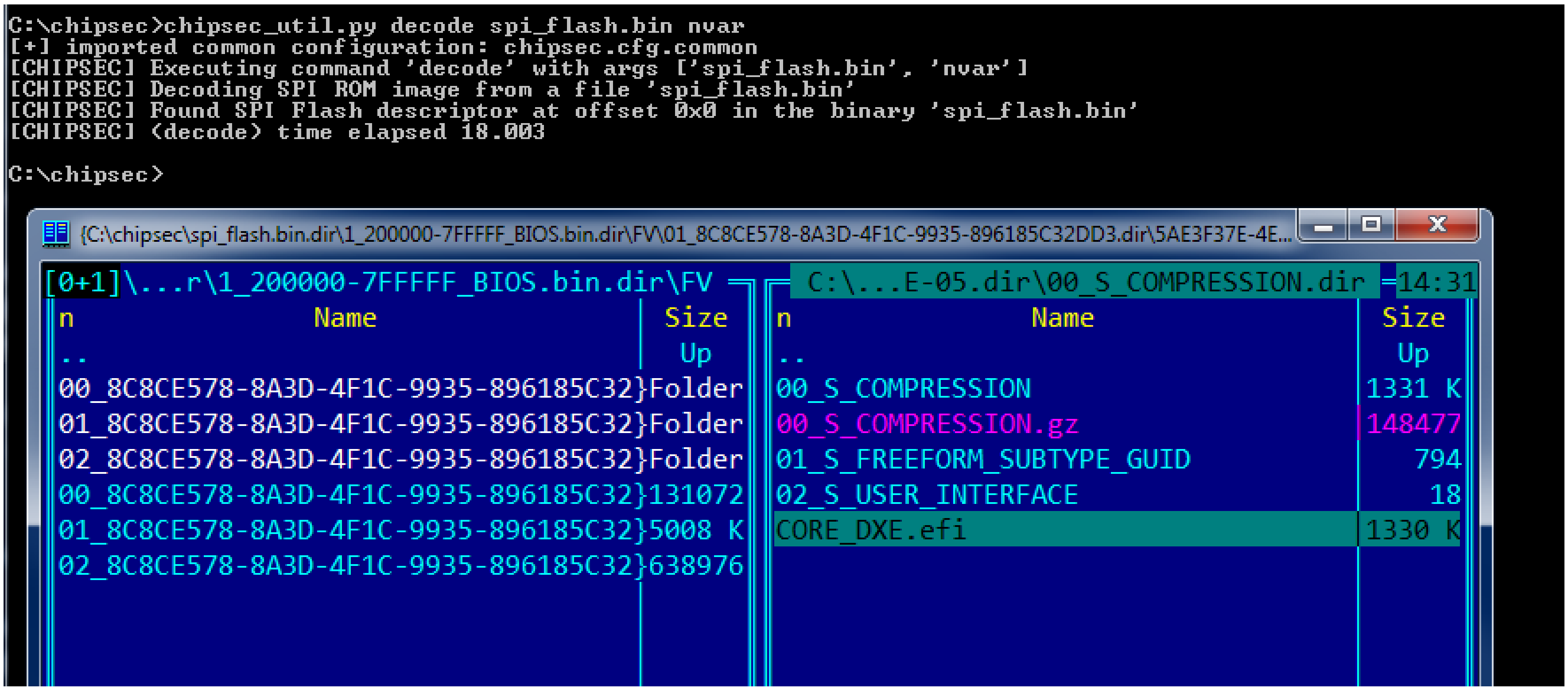
```
///
/// Header format for TE images
///
typedef struct {
  UINT16                     Signature;            // signature for TE format = "VZ"
  UINT16                     Machine;              // from the original file header
  UINT8                      NumberOfSections;     // from the original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

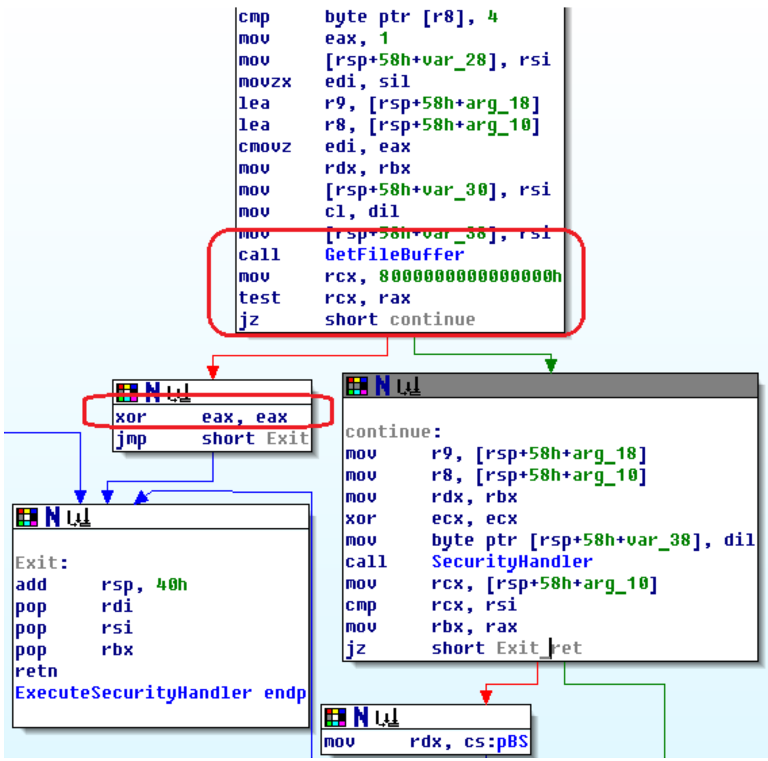
PE/TE Header Handling by the BIOS

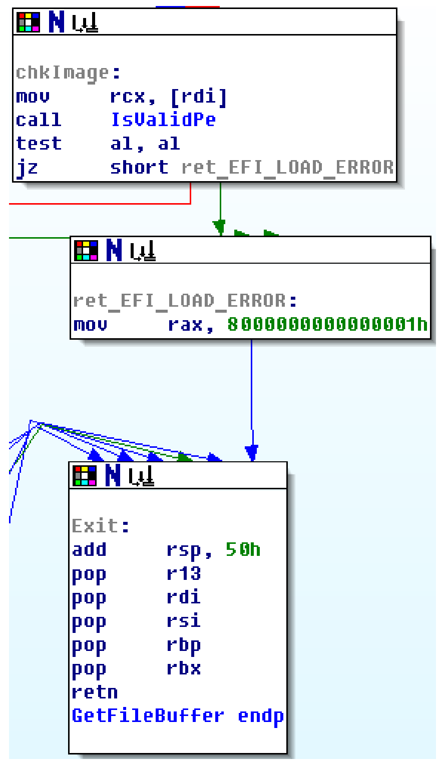
- Decoded UEFI BIOS image from SPI Flash



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

- **`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 without 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

Demo

Abusing Compatibility Support Module (CSM) a.k.a. "Legacy Strikes Back"

Recommendations

CSM ^ Secure Boot

- Force CSM to Disabled if Secure Boot is Enabled
- But don't do that only in Setup HII
- Implement isCSMEnabled() function always returning FALSE when Secure Boot is enabled

Don't Fall Back to Legacy Boot

- Don't fall back to legacy boot through MBR if Secure Boot verification of UEFI executable fails

CSM vs. 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 firmware 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

Clearing of Secure Boot Keys & Restoring Defaults

a.k.a. “No keys, no problem”

Recommendations

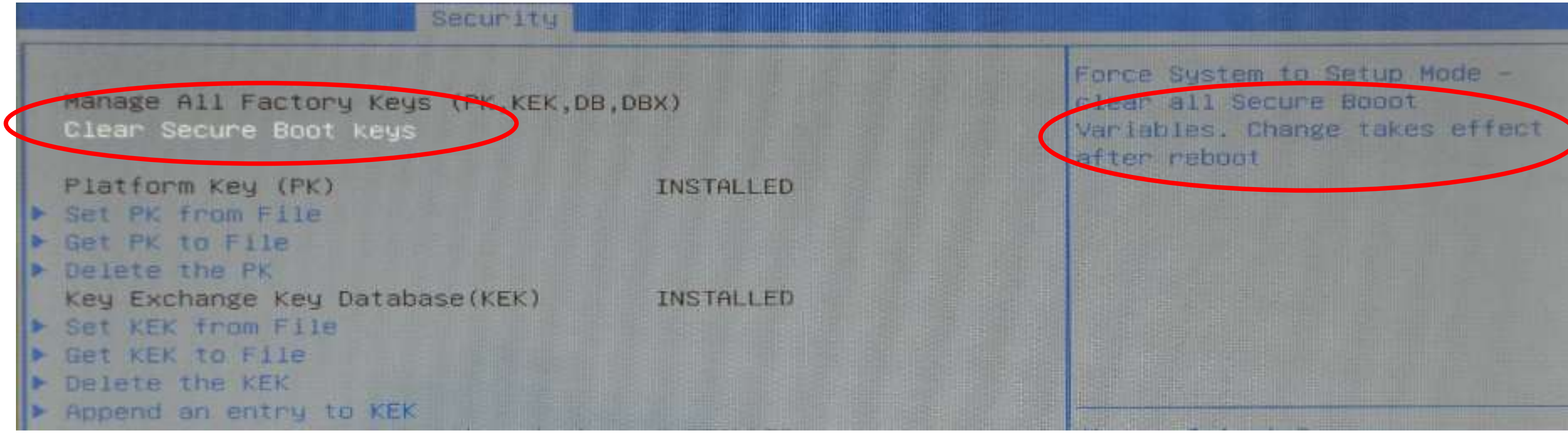
Protect Controls Clearing/Restoring Secure Boot Keys

- Don’t store them in places writeable by malware (like RUNTIME_ACCESS UEFI Variables)
- Require physically present user to clear Secure Boot keys or restore default values

Protect Default Values of Secure Boot Keys (PK, KEK..)

- Store default values in protected areas (e.g. embedded into Firmware Volumes)

Clearing Platform Key… from Software



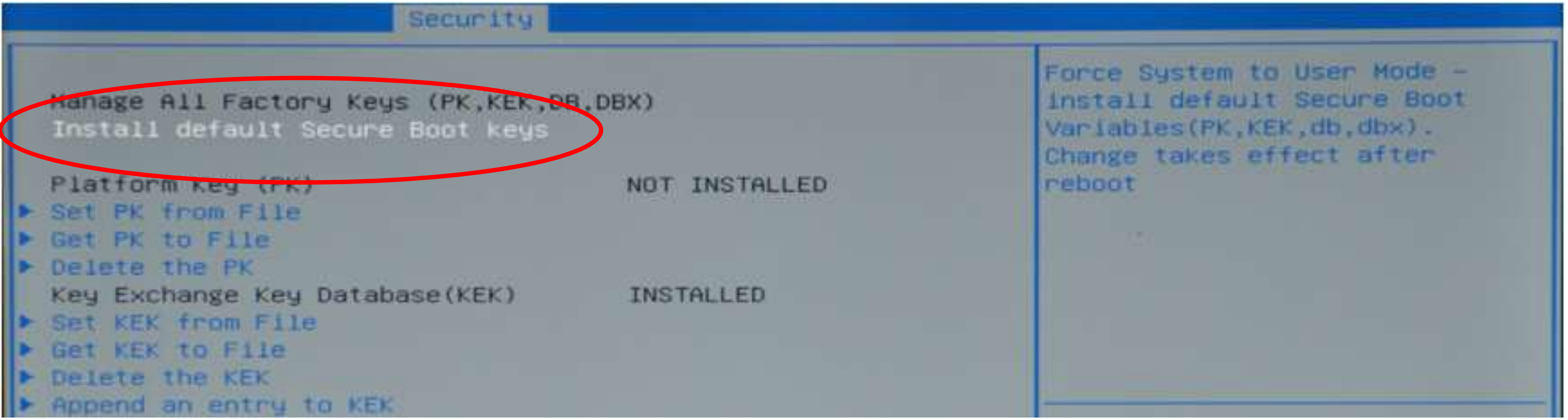
"Clear Secure Boot keys" takes effect after reboot

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

But recall our earlier presentation!

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 signatures to the DBX blacklist ;(

Waiting for Physical Presence
a.k.a. “Beep Beep Beep Boot”

Recommendations

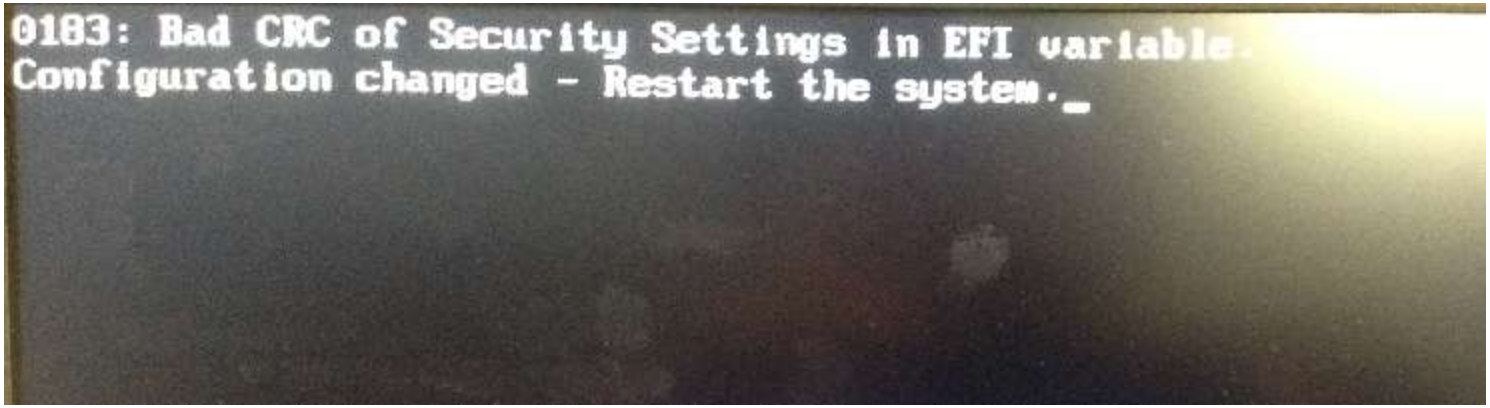
Physical Presence is Important Protection

- Make sure there's a platform specific mechanism to assert physically present user (for example, a button on a device)
- Require physical presence to modify certain Secure Boot configuration (On/Off switch, Custom Mode..)

Honest mistake, but entertaining…

The system protects Secure Boot configuration from unauthorized modification

- Stores and verifies "CRC" of Secure Boot settings
- Upon "CRC" mismatch, beeps 3 times, waits timeout (a few seconds) then…



Keeps booting with modified Secure Boot settings

Summary

Secure Boot is a complex protection relying on correct implementation and configuration of multiple components.

Both efforts enhance the transparency of firmware security research and result in security assessment tools.

We continue to actively research and coordinate issues with platform manufacturers and BIOS vendors.