Attacking and Defending BIOS in 2015

Oleksandr Bazhaniuk, Yuriy Bulygin (presenting), Andrew Furtak, Mikhail Gorobets, John Loucaides, Alex Matrosov, Mickey Shkatov

Intel Security Advanced Threat Research

Agenda

- State of BIOS/EFI Firmware Security
- Recent Classes of Vulnerabilities
  - S3 Resume Boot Script
  - Firmware Configuration (UEFI Variables)
  - Input Pointers in SMI Handlers
  - Call-Outs in SMI Handlers
- Detecting and Mitigating These Vulnerabilities
- Conclusions

Plain Ordinary Art of Breaking BIOS...

* Quotes are from or based on novels by Strugatsky brothers

We seem to have a bit of a problem

- **37 unique** publicly disclosed **issues** in the last ~2 years (by only a handful of researchers)
- **Multiple** of these are really **classes of issues** with many instances in affected firmware products (SMI input pointers, SMI call-outs, indiscriminate use of UEFI vars..)
- Many same issues **affect multiple vendors** at once (S3 boot script, UEFI variables, SMI call-outs, SMI input pointers, missing basic BIOS write protections…)
- Issues in open source EDK reference implementation may find their way in multiple UEFI firmware products
- And updating system firmware is not an easy thing

Jolly Ghosts (2013-2014)

Vulnerability	Ref	Affected	Discoverer
EFI firmware is not write protected (attack on Full-Disk Encryption with TPM aka "Angry Evil Maid", subverting TPM measured boot). In 2009, Sacco & Ortega discovered legacy BIOS were not write protected	CSW2013, NoSuchCon 2013	Multiple	Intel ATR, MITRE, LegbaCore
Secure Boot bypass due to SPI flash protections are not used	BH2013		Intel ATR
Secure Boot bypass due to PE/TE Header confusion	CSW2014		
Secure Boot bypass due to CSM default enabled or CSM enable/disable stored in Setup (2 issues)	CSW2014		
Secure Boot bypass due to "Clear keys" and "Restore default keys" stored in Setup	CSW2014		
Secure Boot bypass due to ignoring SecureConfig integrity mismatch	CSW2014		
Secure Boot bypass via on/off switch stored in Setup variable	CSW2014	Multiple	Intel ATR, LegbaCore
Unauthorized modification of UEFI variables in UEFI systems (Secure Boot policies stored in Setup, corrupting Setup contents) – 2 issues	VU#758382 Tianocore	Multiple	LegbaCore, Intel ATR
SMM Cache attack protections (SMRR) not enabled ("The Sicilian")	VU#255726	Multiple	LegbaCore
Dell BIOS in some Latitude laptops and Precision Mobile Workstations vulnerable to buffer overflow ("Ruy Lopez")	VU#912156	Dell	
SMI Suppression if SMM BIOS protection is not used ("Charizard")	VU#291102	Multiple	
Intel BIOS locking mechanism contains race condition that enables write protection bypass ("Speed Racer")	VU#766164	AMI, Phoenix	

Exploding Rainbows (2014)

Vulnerability	Ref	Affected	Discoverer
UEFI EDK2 Capsule Update vulnerabilities a.k.a. “King and Queen’s Gambit” (2 issues)	VU#552286 Tianocore	Multiple, EDK2	LegbaCore
UEFI Variable “Reinstallation” (bypassing Boot-Service only variables)	Tianocore	Multiple, EDK2	Intel ATR
Insecure Default Secure Boot Policy for Option ROMs	Tianocore	EDK2	Intel ATR
Incorrect PKCS#1v1.5 Padding Verification for RSA Signature Check			
Overwrite from PerformanceData Variable			
CommBuffer SMM Overwrite/Exposure (3 issues)			
TOCTOU (race condition) Issue with CommBuffer (2 issues)			
SMRAM Overwrite in Fault Tolerant Write SMI Handler (2 issues)			
SMRAM Overwrite in SmmVariableHandler (2 issues)			
Integer/Heap Overflow in SetVariable			
Heap Overflow in UpdateVariable			
Overwrite from FirmwarePerformance Variable			
Integer/Buffer Overflow in TpmDxe Driver			
Protection of PhysicalPresence Variable			

Spitting Devil's Cabbage (2014-2015)

Vulnerability	Ref	Affected	Discoverer
Boot Failure Related to UEFI Variable Usage (36 issues)	Tianocore	EDK2	Intel ATR, TianoCore dev, LegbaCore
Boot Failure Related to TPM Measurements	Tianocore	EDK2	TianoCore dev
Tianocore UEFI implementation reclaim function vulnerable to buffer overflow (2 issues)	VU#533140 Tianocore	EDK2, Insyde	Rafal Wojtczuk, LegbaCore
Overflow in Processing of AuthVarKeyDatabase	Tianocore	EDK2	Rafal Wojtczuk, LegbaCore
Counter Based Authenticated Variable Issue	Tianocore	EDK2	TianoCore dev
Some UEFI systems do not properly secure the EFI S3 Resume Boot Path boot script (“Venamis”)	VU#976132	Multiple	Rafal Wojtszuk, Intel ATR, LegbaCore
Some BIOS protections are unlocked on resume (“Snorlax”)	VU#577140		LegbaCore
Loading unsigned Option ROMs (“Thunderstrike”) based on earlier work by @snare	trmm.net	Apple	Trammell Hudson
SMI input pointer validation vulnerabilities (multiple issues)	CSW2015	Multiple	Intel ATR
SMI handler call-out vulnerabilities (multiple issues) Earlier by Filip Wecherowski & ITL (bugtraq, ITL)	LegbaCore	Multiple	LegbaCore
SPI flash configuration lock (FLOCKDN) is lost after resume from S3 sleep (**Update:** Apple advisory)	reverse.put.as	Apple	Pedro Vilaça **Update:** Trammell Hudson, LegbaCore

The list may be incomplete

Your BIOS is definitely maybe vulnerable

This is one way to handle the problem

http://sovietart.me/

Calm silence ends the history of mankind...

So let's talk what needs to be done

But, first, why we need any changes

- Attacks via S3 Resume Boot Script

  `#S3SleepResumeBootScript`

- Attacks via UEFI Variables

  `#BadBIOSVariableContents`

- Attacks via Bad SMI Handlers Input Pointers

  `#SMIHandlerBadInputPointers`

- Attacks via SMI Handlers Call-Outs

  `#ThisVulnSeriouslyHadToBeGoneLongAgo`

Attacking Firmware via S3 Resume Boot Script

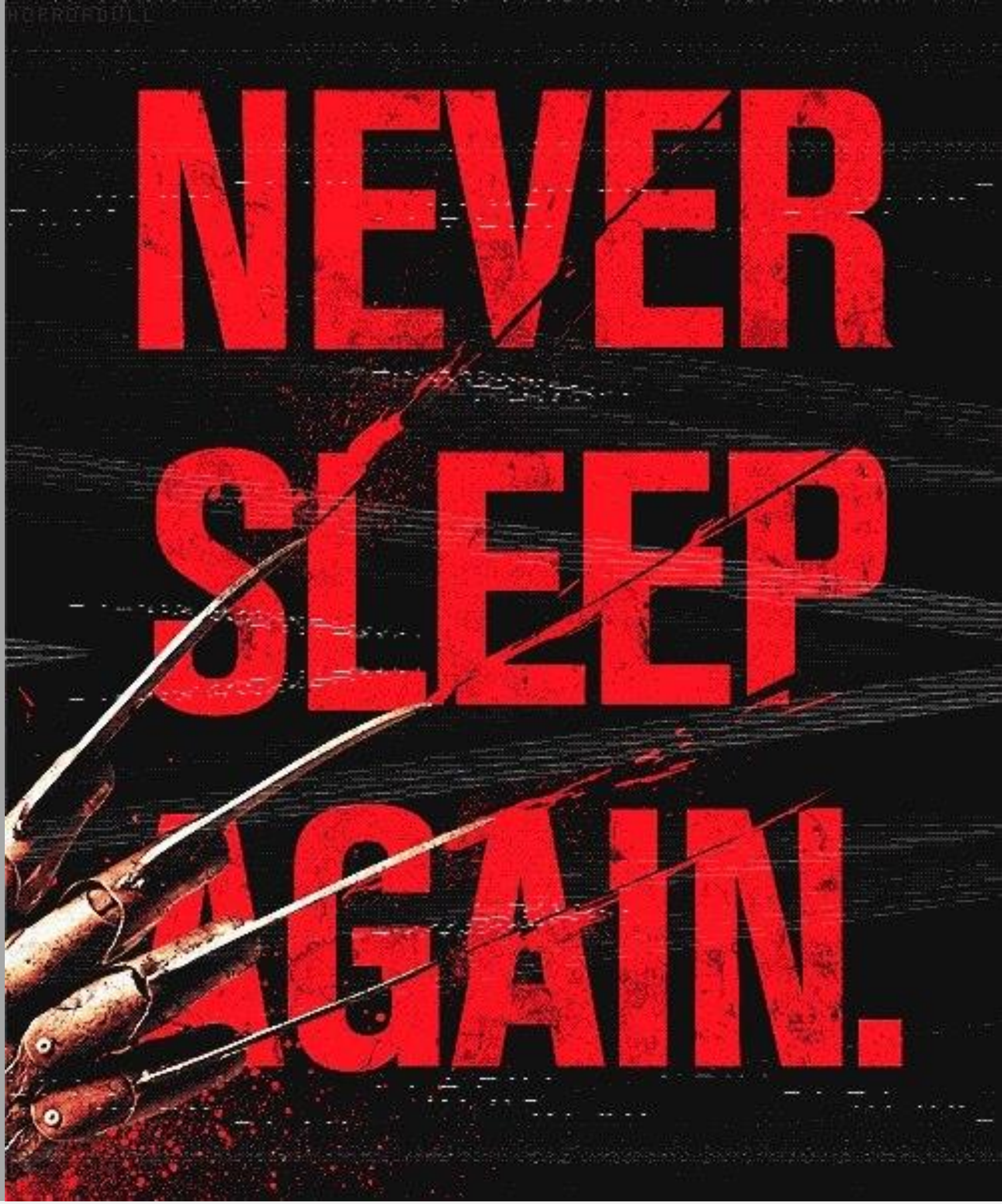


Image source

VU# 976132 (CVE-2014-8274)

- Freddy Krueger vulnerabilities (S3 Resume Boot Script) were independently discovered by us and other security researchers
- Rafal Wojtczuk of Bromium and Corey Kallenberg (@coreykal) of LegbaCore first published *Attacks on UEFI Security* (paper)
- Details of PoC exploit were described by Dmytro Oleksiuk (@d_olex) in Exploiting UEFI boot script table vulnerability
- Pedro Vilaça (@osxreverser) disclosed a related vulnerability in Mac EFI firmware (SPI Flash Configuration HW lock bit FLOCKDN is gone after waking from sleep)

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

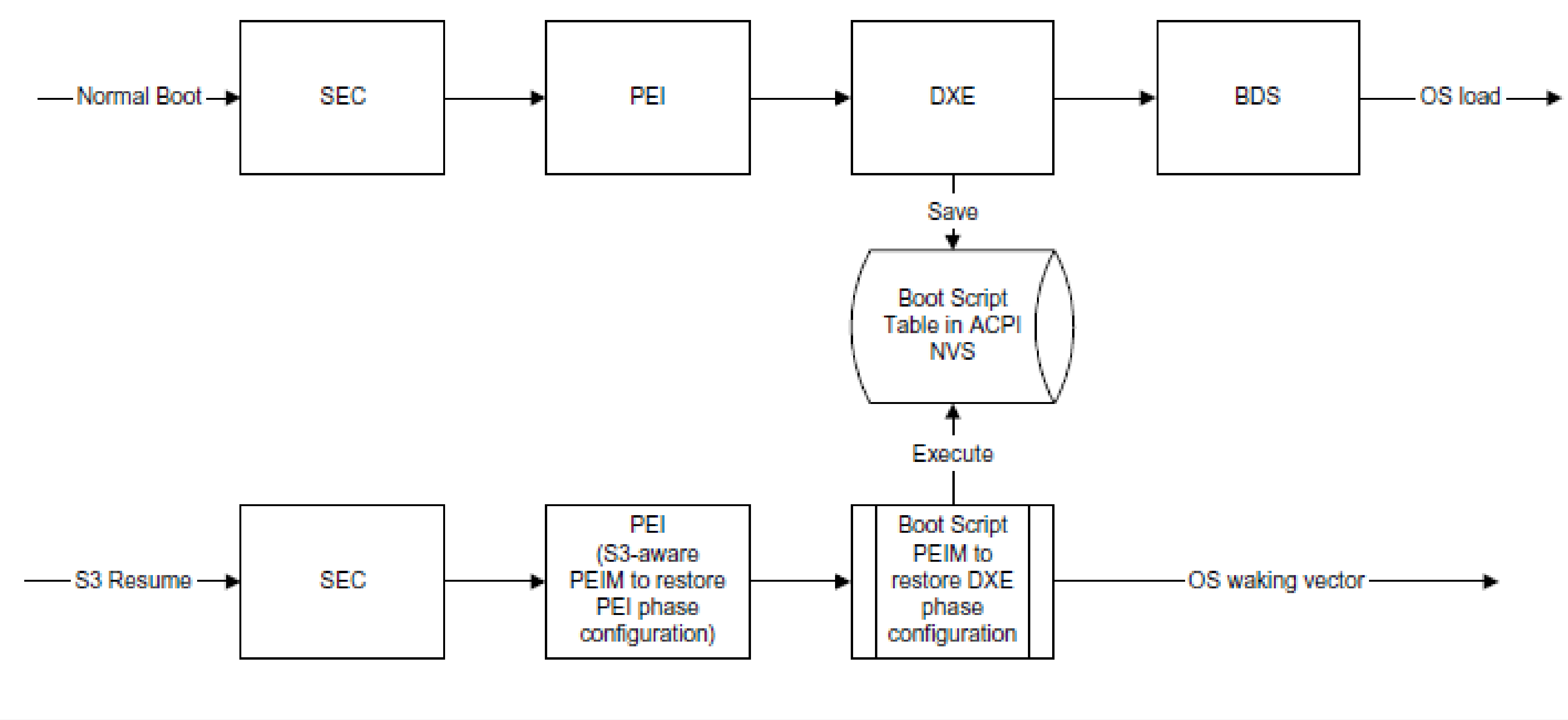
```
typedef struct {
//
// Acpi Related variables
//
EFI_PHYSICAL_ADDRESS AcpiReservedMemoryBase;
UINT32 AcpiReservedMemorySize;
EFI_PHYSICAL_ADDRESS S3ReservedLowMemoryBase;
EFI_PHYSICAL_ADDRESS AcpiBootScriptTable;
..
} ACPI_VARIABLE_SET_COMPATIBILITY;
```

“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



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-write (`0x01`)
- Memory write (`0x02`)
- Memory read-write (`0x03`)
- PCIe configuration write (`0x04`)
- PCIe configuration read-write (`0x05`)
- SMBus execute (`0x06`)
- Stall (`0x07`)
- Dispatch (`0x08`) / Dispatch2 (`0x09`)
- Information (`0x0A`)
- …

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
```

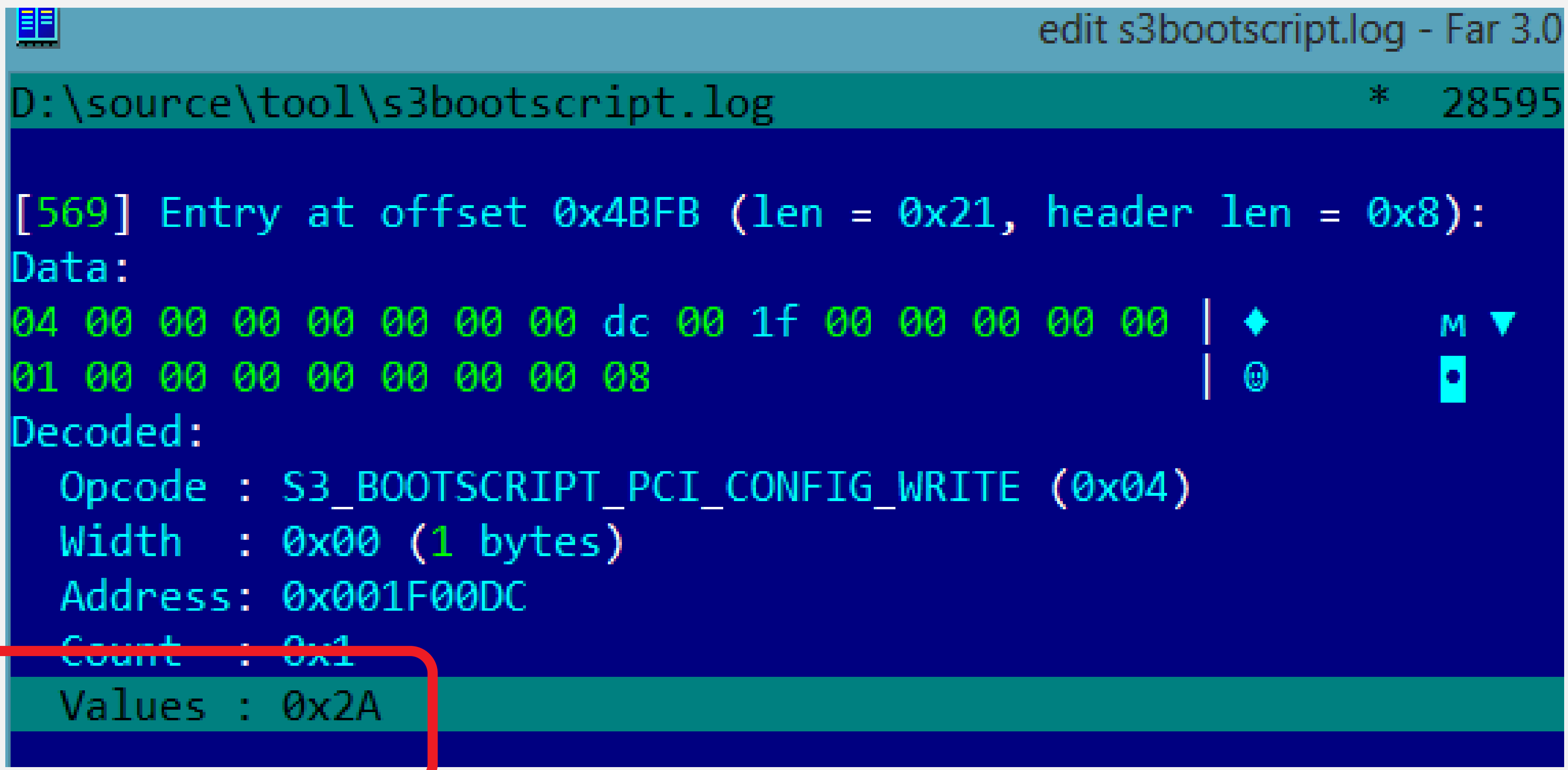
“Dispatch” Opcodes

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`** indicates BIOS hardware write protection is ON)



So what can go wrong with the script?

- 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

So what’s the impact?

Malware in the OS may be able to change the actions that are performed by firmware on S3 resume before the OS resumes at the waking vector

Ok… And?

- Execute arbitrary firmware code during early resume
- Disable hardware protections such as BIOS write protection which are going to be restored by the script
- Install persistent BIOS rootkit in the SPI flash memory
- Read/write any memory or execute arbitrary code in the context of system firmware during early boot (PEI)
- Bypass secure boot of the OS and install UEFI Bootkit

Yes, It Can Steal

Your PGP keys!

'Voodoo' Hackers: Stealing Secrets From Snowden's Favorite OS Is Easier Than You'd Think

[Forbes]

Image source: http://www.imdb.com/title/tt0439581/

83% of all days in a year start the same:
alarm clock rings…

then vulnerable BIOS awakes…

Attacking S3 Boot Script (Demo)

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
```

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
```

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
```

Detecting & Mitigating
S3 Resume Boot Script Issues

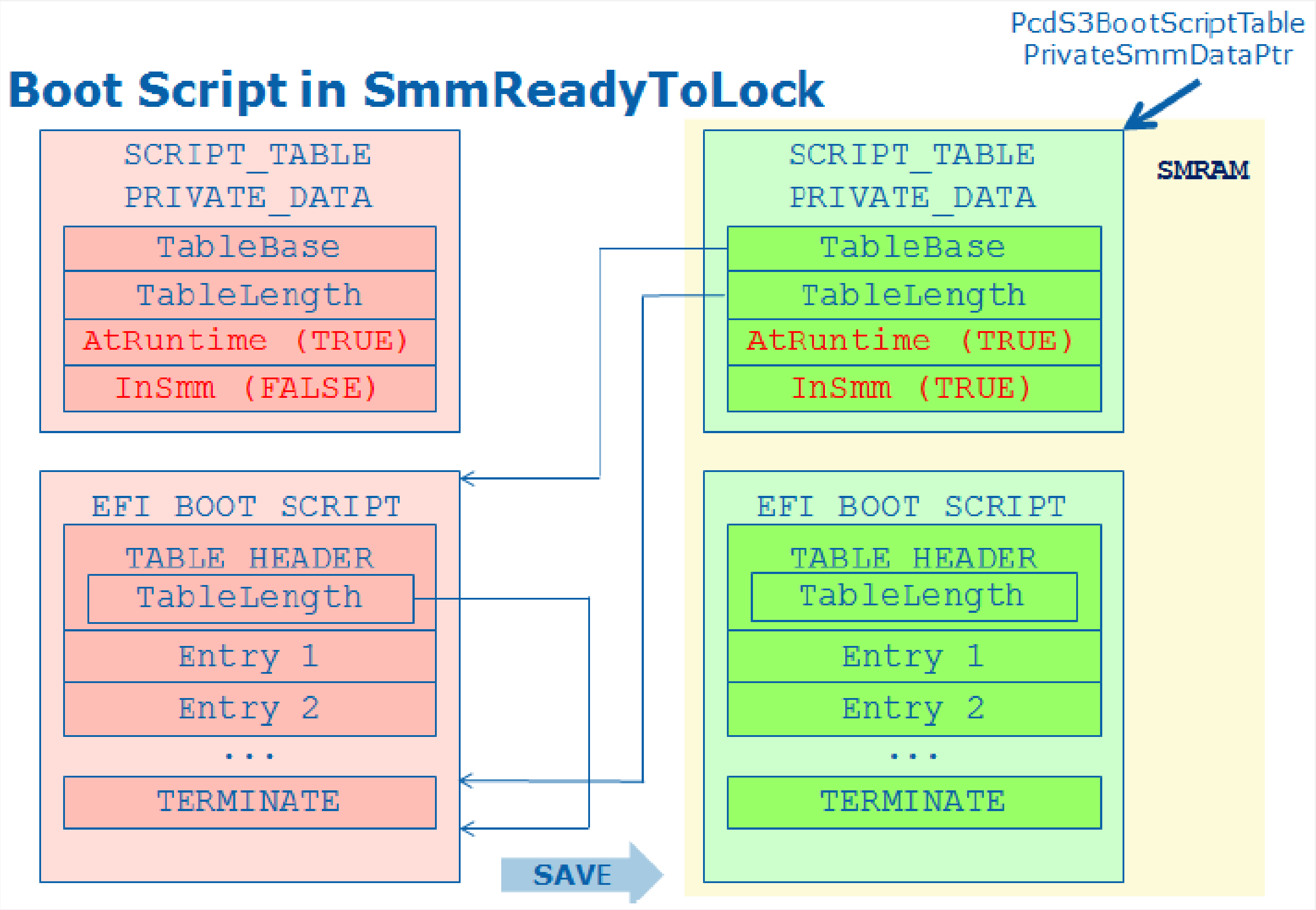
There's a script to detect these issues

```
# 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
```

Fixing S3 Boot Script Protections

1. Do not keep address of S3 Boot Script table (or a structure with a pointer to the table) in unprotected NV UEFI variable (ex. **`NV+RT+BS AcpiGlobalVariable`**)
2. Do not save the S3 Boot Script table to memory accessible by the OS or DMA capable devices (e.g. use EDKII *LockBox*)
3. Do not save the PEI executable that parses and executes the S3 Boot Script table and any other PEI executable(s) needed for S3 resume to memory accessible by the OS or DMA capable devices
4. Review the S3 Boot Script for *Dispatch* opcodes and establish whether the target code is protected.

Protecting S3 Boot Script with *LockBox*



[A Tour Beyond BIOS Implementing S3 Resume with EDKII]

LockBox: [https://github.com/tianocore/edk2-MdeModulePkg/blob/master/Include/Protocol/LockBox.h]

Saving S3 Boot Script to LockBox

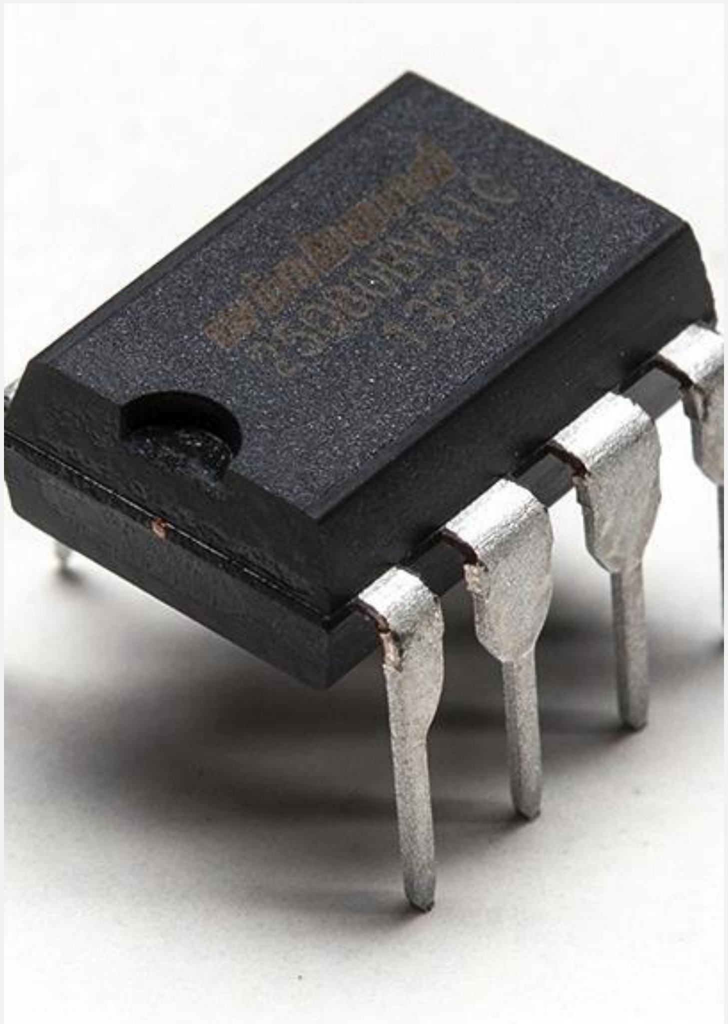
SaveBootScriptDataToLockBox():

```
…
//
// mS3BootScriptTablePtr->TableLength does not include
EFI_BOOT_SCRIPT_TERMINATE, because we need add entry at runtime.
// Save all info here, just in case that no one will add boot
script entry in SMM.
//
Status = SaveLockBox (
           &mBootScriptDataGuid,
           (VOID *)mS3BootScriptTablePtr->TableBase,
          mS3BootScriptTablePtr->TableLength +
          sizeof(EFI_BOOT_SCRIPT_TERMINATE)
           );
ASSERT_EFI_ERROR (Status);
Status = SetLockBoxAttributes (&mBootScriptDataGuid,
LOCK_BOX_ATTRIBUTE_RESTORE_IN_PLACE);
```

https://svn.code.sf.net/p/edk2/code/trunk/edk2/MdeModulePkg/Library/PiDxeS3BootScriptLib/BootScriptSave.c

Attacking EFI Firmware via 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`**

Image Source: Adafruit

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

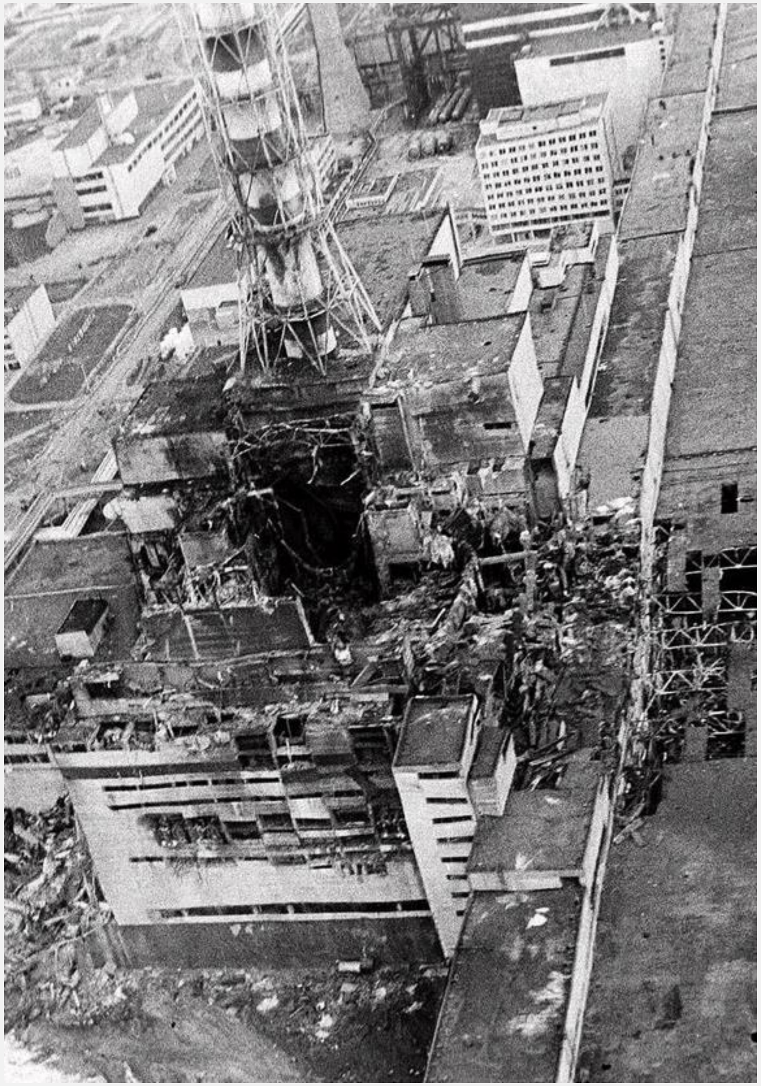
Secure Boot certificates (PK, KEK, db, dbx)

Setup

Things we found in unprotected runtime (read “user-mode”) accessible variables

- **Secure Boot configuration** (All You 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
- Data which firmware really **really** needs to just boot
- **Secrets**: BIOS passwords in clear

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

Image Source: [The Atlantic]

But that was a theory. In practice…

- Multiple unique vulnerabilities (~50 instances), related to UEFI variables, were discovered only recently
- Both in EFI firmware and in open source Tiano reference implementation
- Resulted in
  - OS Secure Boot bypass due to settings stored in EFI variables
  - Unbootable platform due to corruption of EFI variable contents
  - Buffer overflows when consuming EFI variable contents
  - Arbitrary overwrites due to pointers in EFI variables
  - Bypassing Boot-Services protection by re-installing as Runtime
  - Bypassing physical presence protection of EFI 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

Image Source: Anchorman

Why bother? Just bring it to IT and ask to "re-install" firmware…

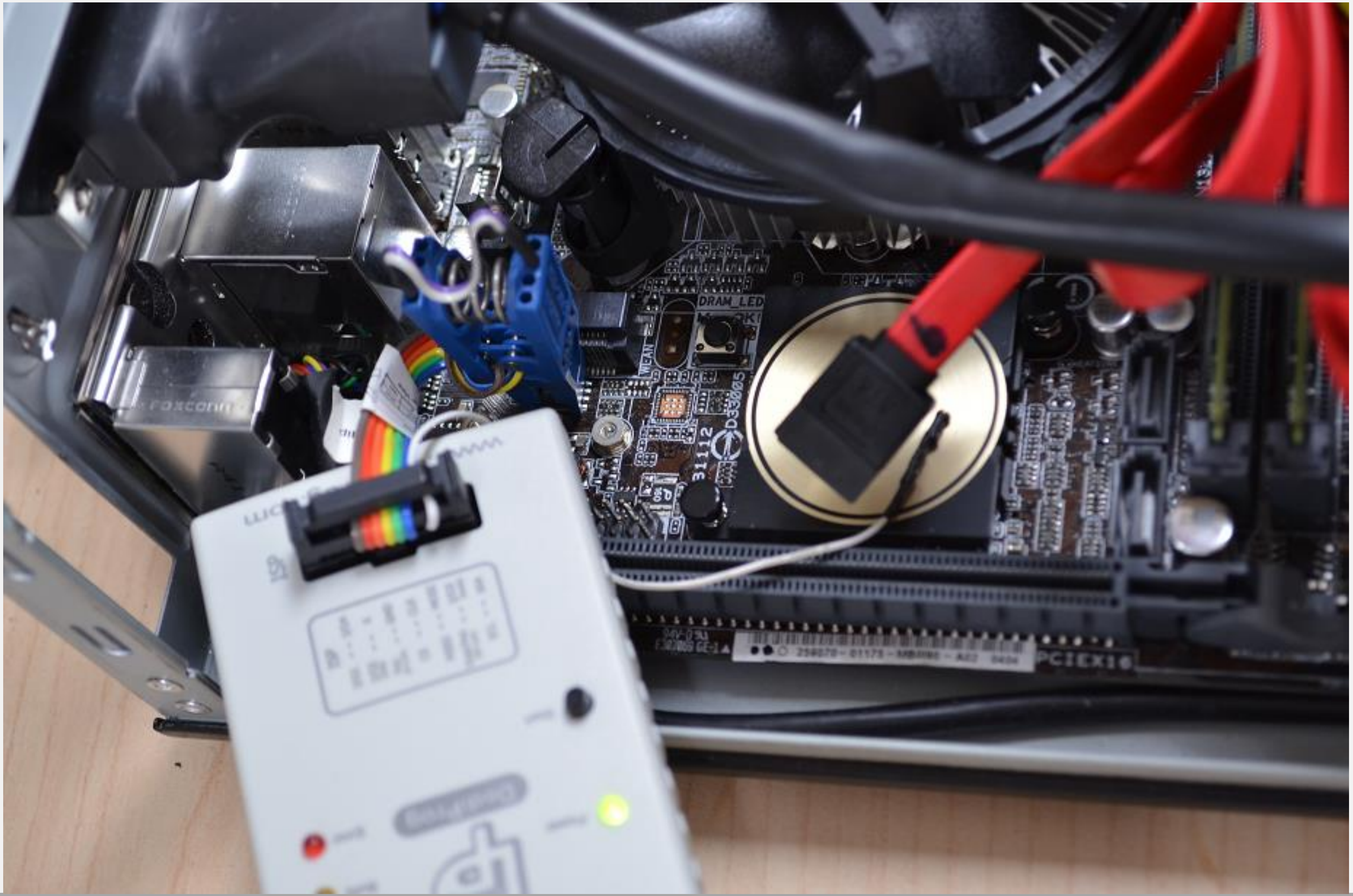


Image Source: Intel ATR ;)

You may as well bring this

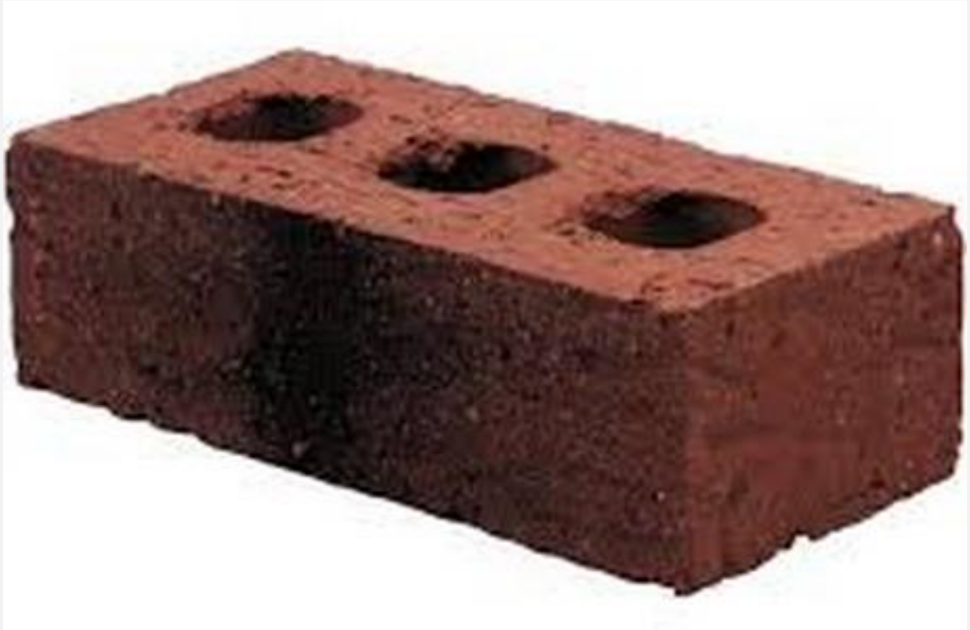


Image Source: [Anchorman]

Avoiding Problems with UEFI Variables

Image Source: [KEEP CALM-O-MATIC]

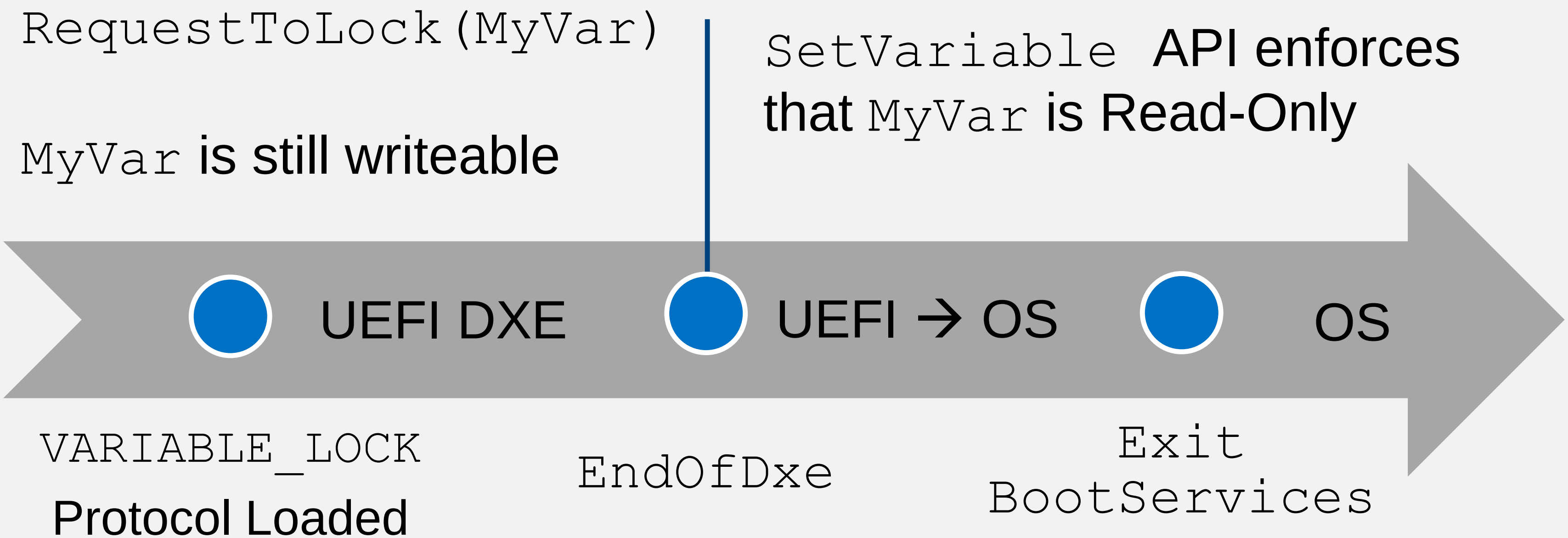
Limit Access to UEFI Variables

- Separate critical settings from other setting. Store them in different variables with different protections
- Remove `RUNTIME_ACCESS` attribute
- Make them Read-Only via `VARIABLE_LOCK_PROTOCOL`
- Use UEFI Authenticated Variables
- Remove debug/test content (e.g. HW lock policies)
- Use PCD instead of variables
- Some variables require user present (e.g. `SetupMode`)
- May implement integrity checks for critical variables
- Storing BIOS passwords or other sensitive content in variables in clear is not a good protection

Validate Contents of UEFI Variables

- Assume contents of the variables are malicious. Validate them before consuming
- Is there an address in the variable? Is it pointing to your own code/data?
- Validate data written to variables is within allowed range
- Can you boot if variable is corrupted? If no, apply defaults and enter recovery
- Recover to defaults if critical settings are invalid or missing. Implement a catastrophic recovery

Read-Only Variables (Variable Lock)



EDKII reference code implements Variable Lock Protocol:
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Protocol/VariableLock.h

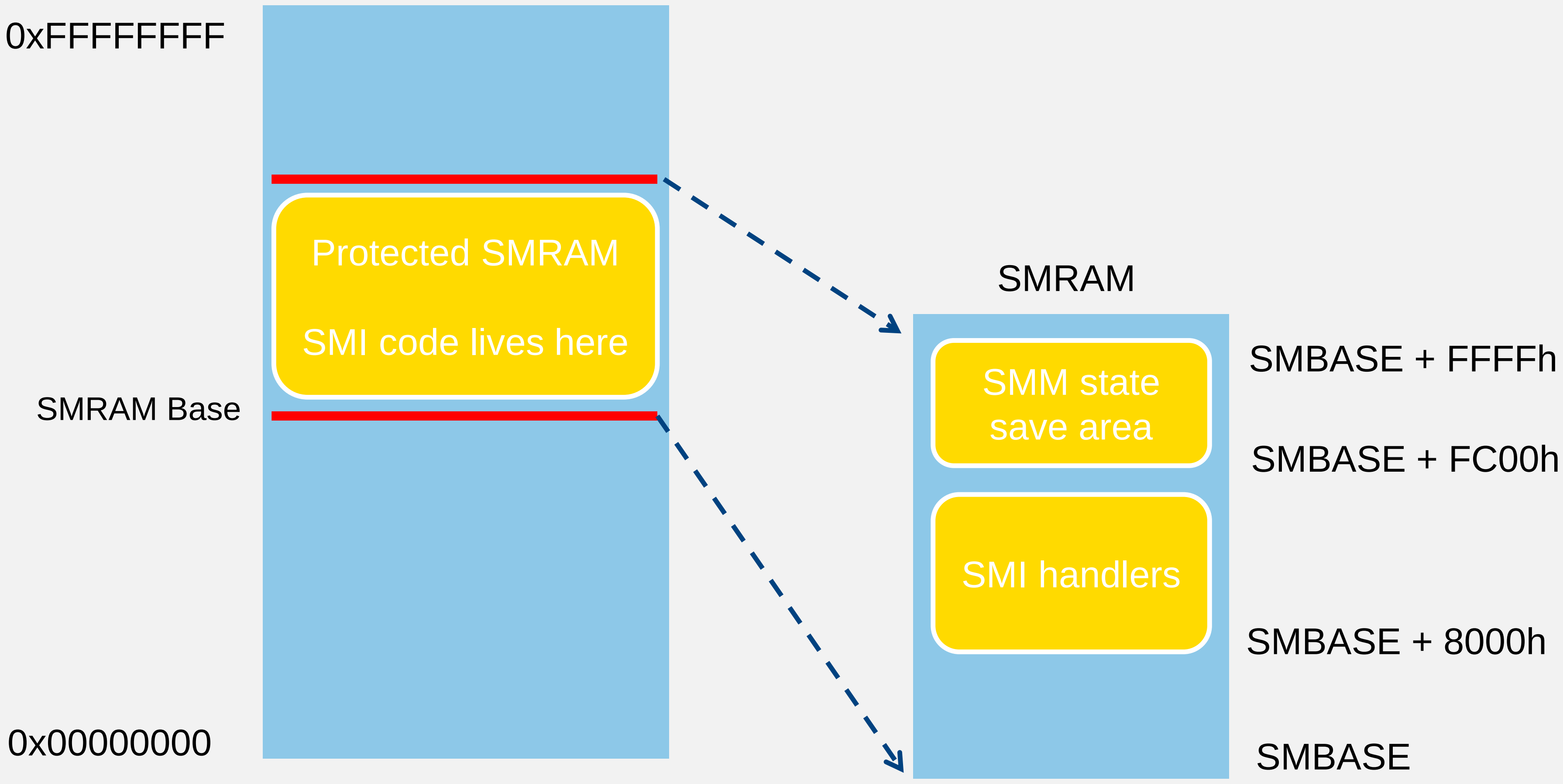
Poisonous Pointers

Attacking SMI Handlers via Unvalidated Input Pointers

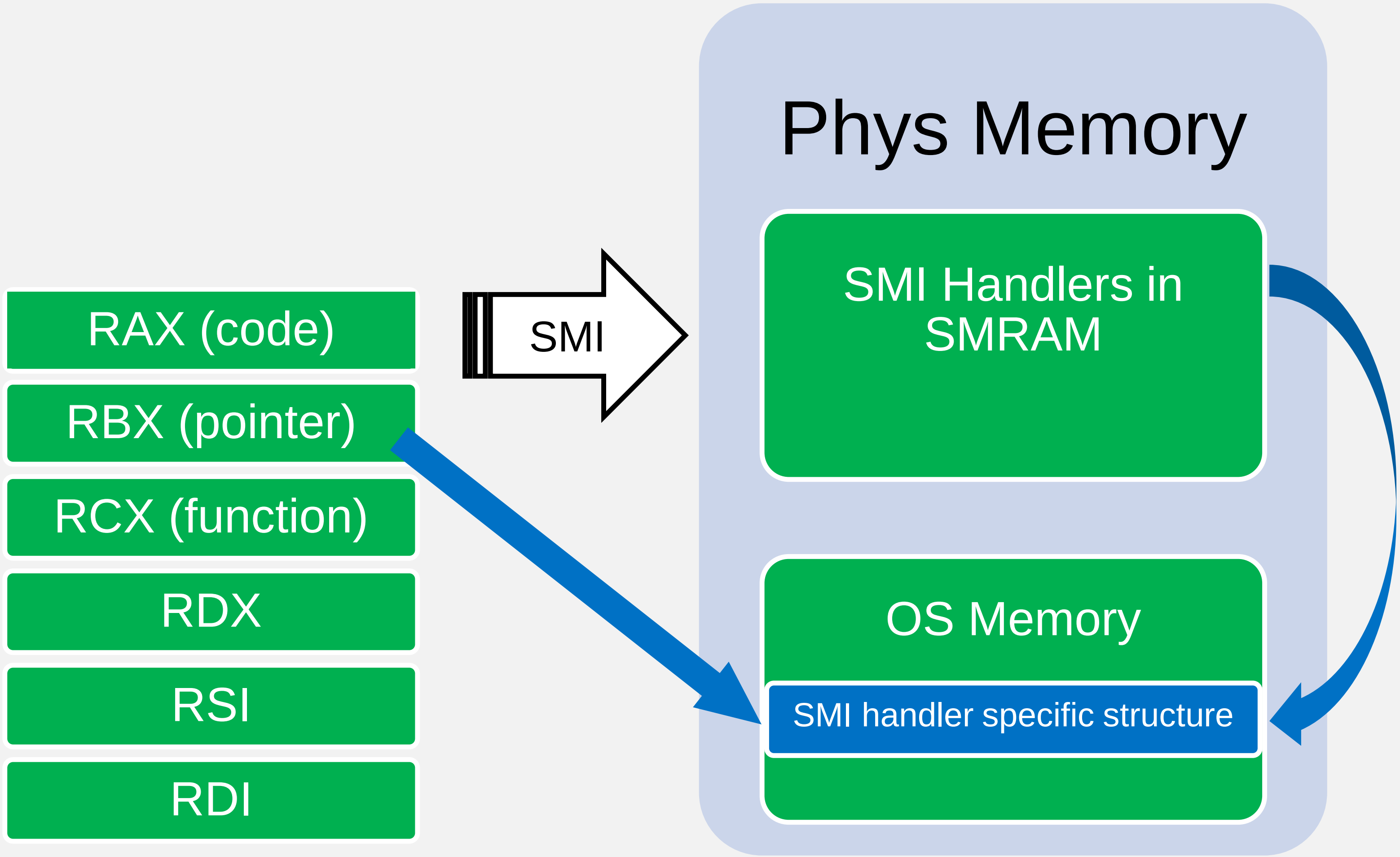
Image source

Where there is no BIOS, there is boredom. BIOS makes life interesting.

System Management Interrupt (SMI) Handlers

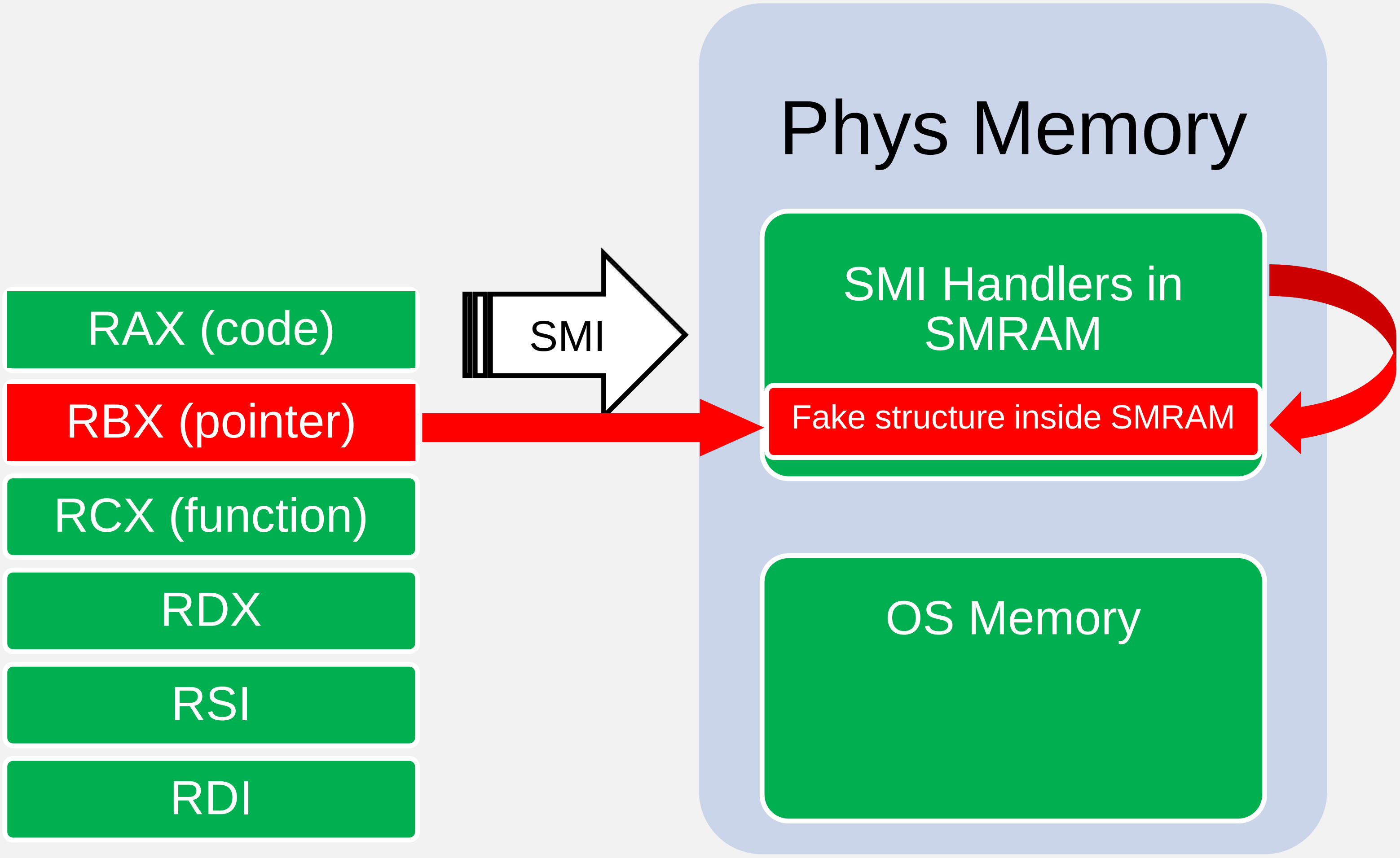


Pointer Arguments to SMI Handlers



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

If SMI Handler Doesn’t Check Pointers



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

What to overwrite inside SMRAM?

- Exploit often doesn't control values written to target address

- What can an exploit overwrite in SMRAM?
  - SMI handler code starting at `SMBASE + 8000h`
  - Internal SMI handler's state/flags inside SMRAM
  - Contents of SMM state save area at `SMBASE + FC00h`, where the CPU state is stored on SMM entry

- Current value of SMBASE value is also saved in state save area at offset `FEF8h` and restored on SMM exit (RSM)

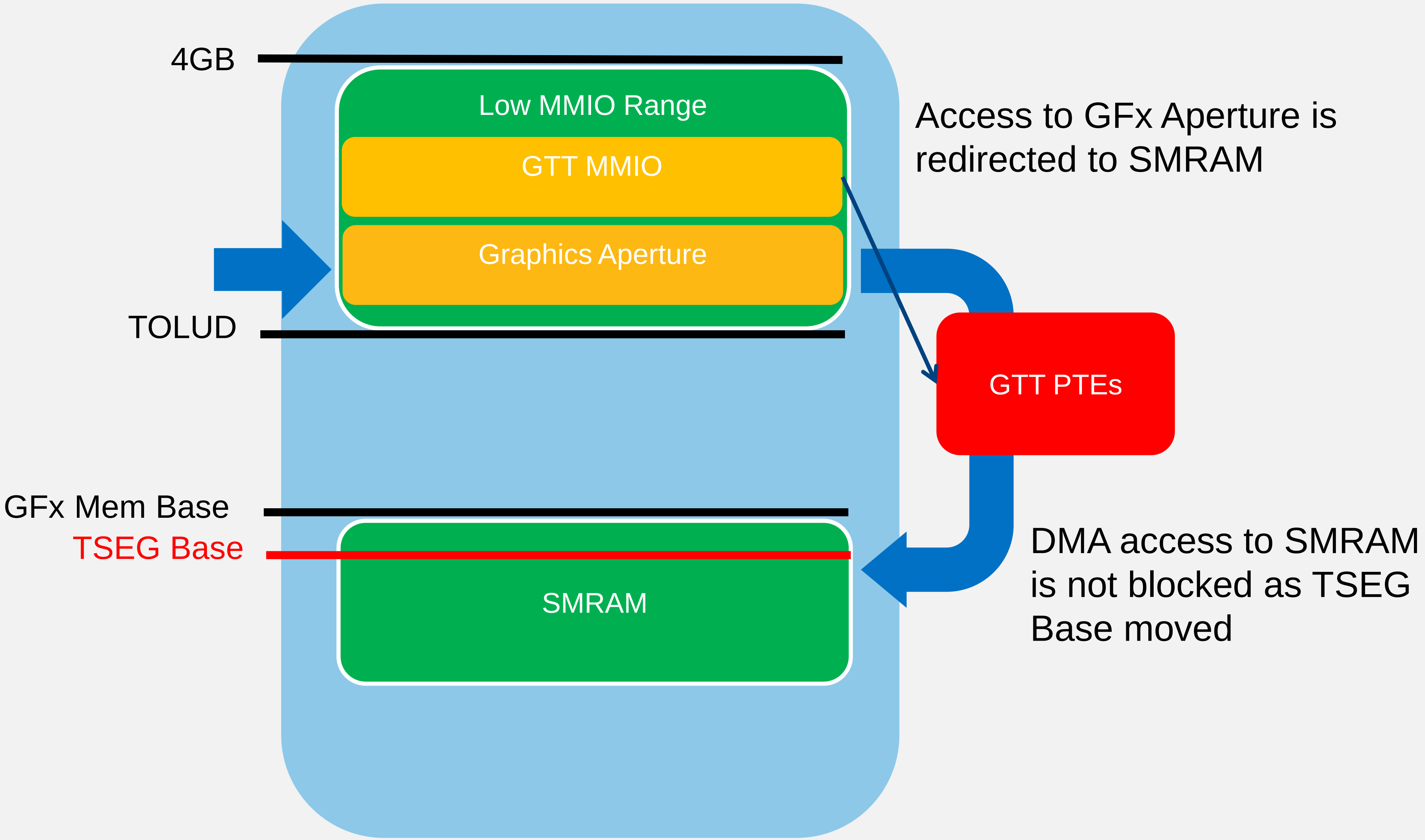
- An exploit can move SMRAM to a new, unprotected location by changing the SMBASE value stored in the SMM state save area

How does exploit know where to write?

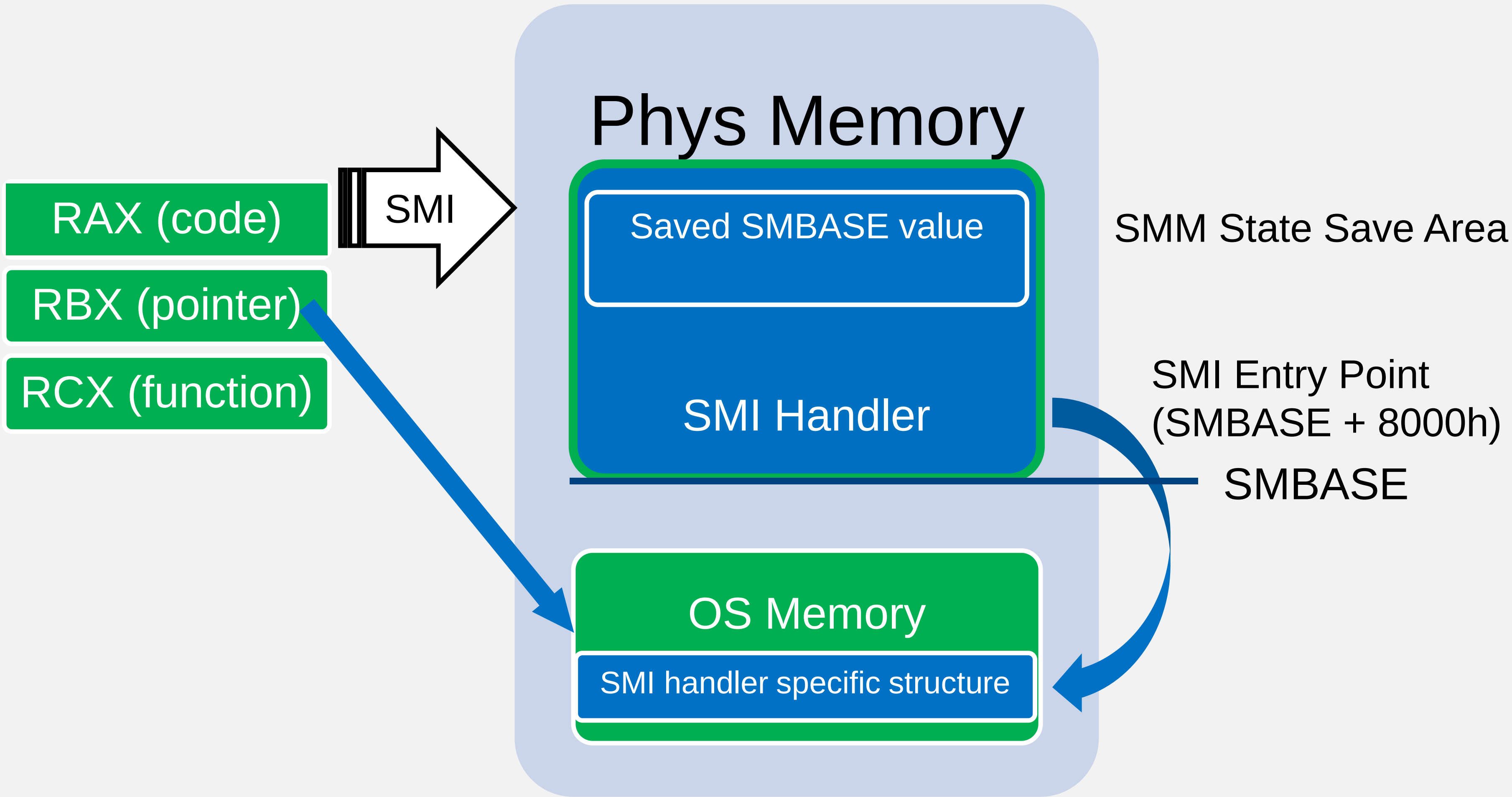
1. Dump contents of SMRAM to find SMBASE
   - Use another vulnerability (e.g. S3 boot script) to disable SMRAM protections and use DMA or graphics to read SMRAM
   - Read SPI flash, extract SMM EFI binaries and RE SMM init code
   - Use similar SMI pointer *read* vulnerability to expose SMRAM
   - Use hardware JTAG debugger offline

2. Exploit can guess location of SMBASE
   - Try SMBASE locations equal to SMRR base or SMRR base – `8000h` (SMRR base at SMI entry point)
   - Blind iteration through all offsets within SMRAM as potential saved SMBASE value

One way to acquire contents of SMRAM

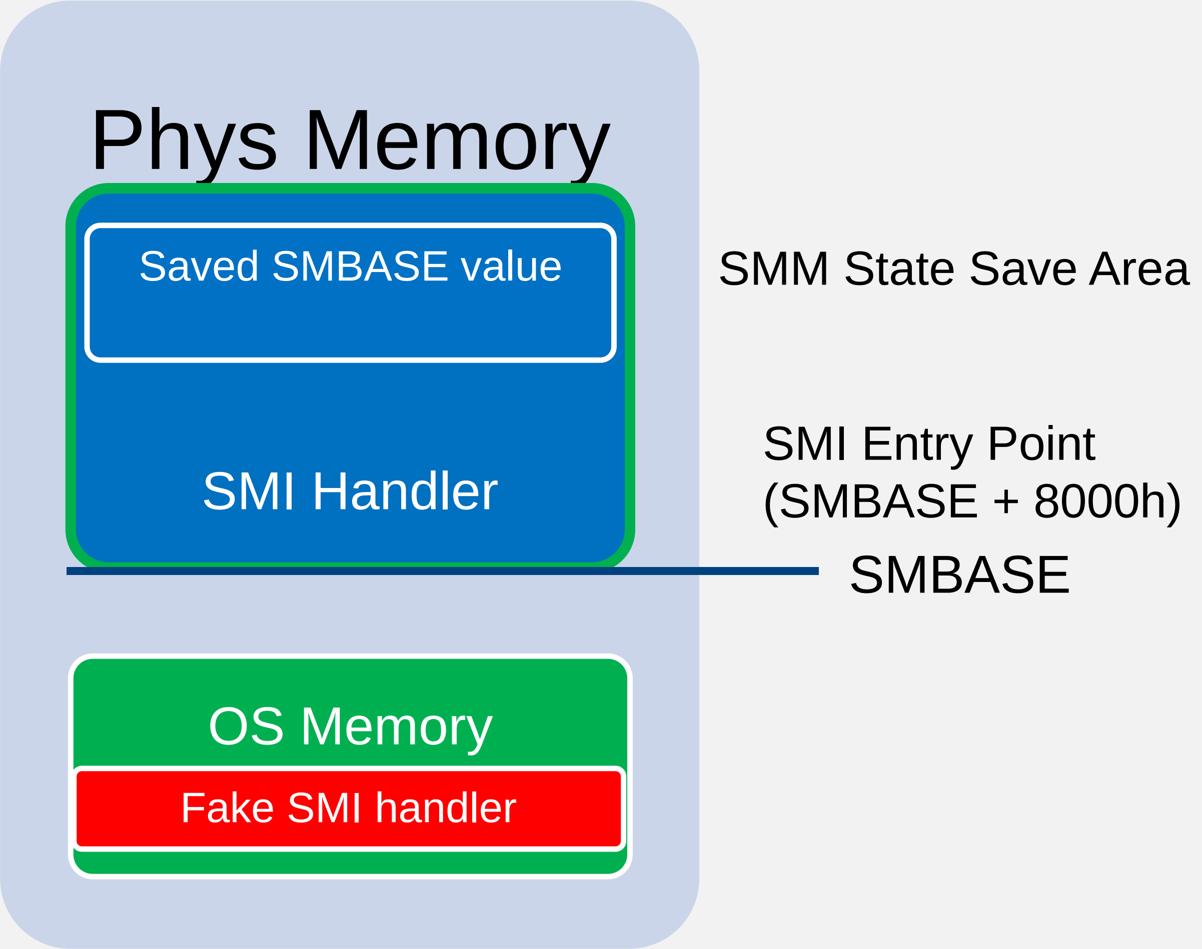


How does the attack work?



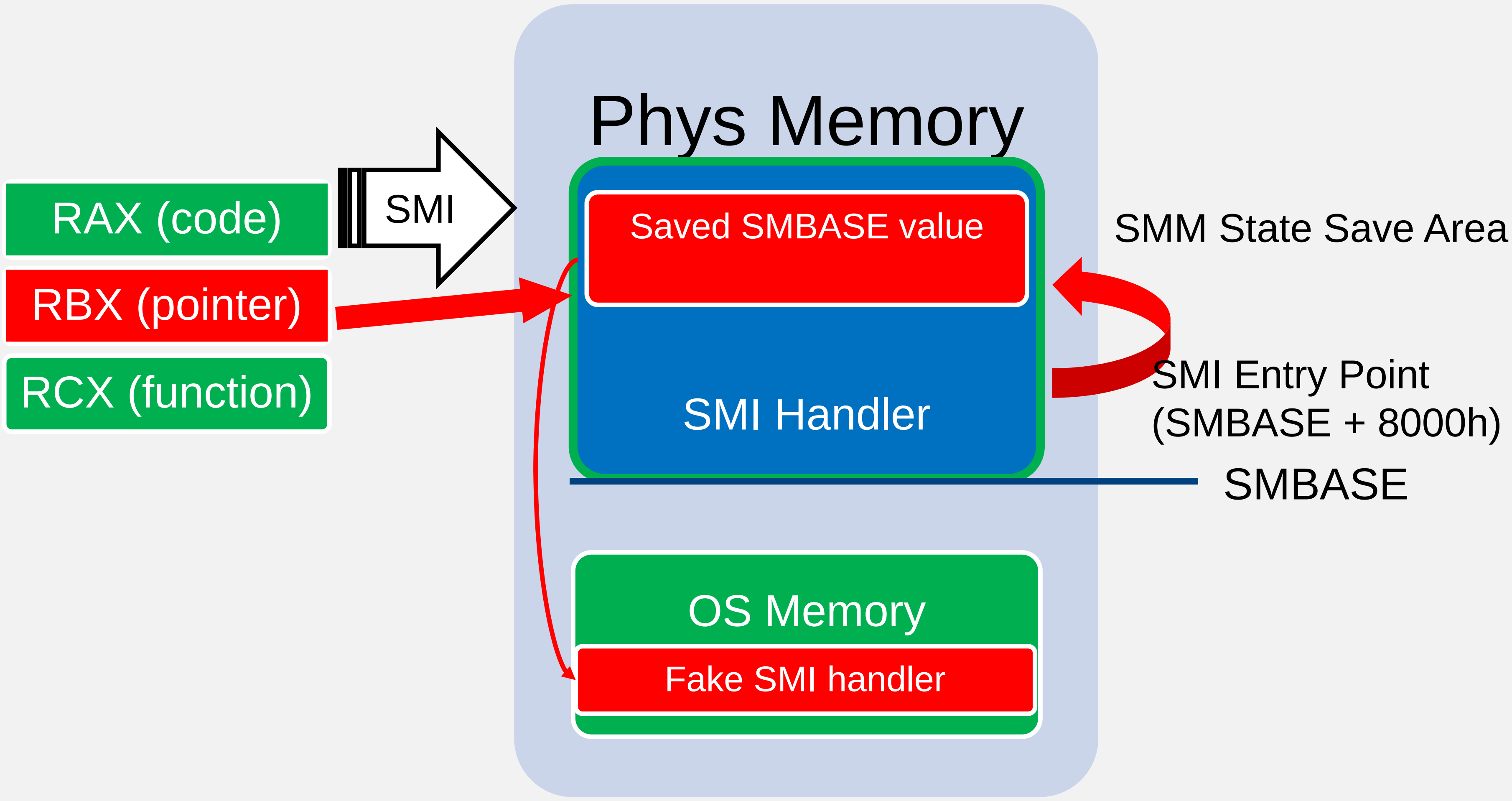
- CPU stores current value of SMBASE in SMM save state area on SMI and restores it on RSM

How does the attack work?



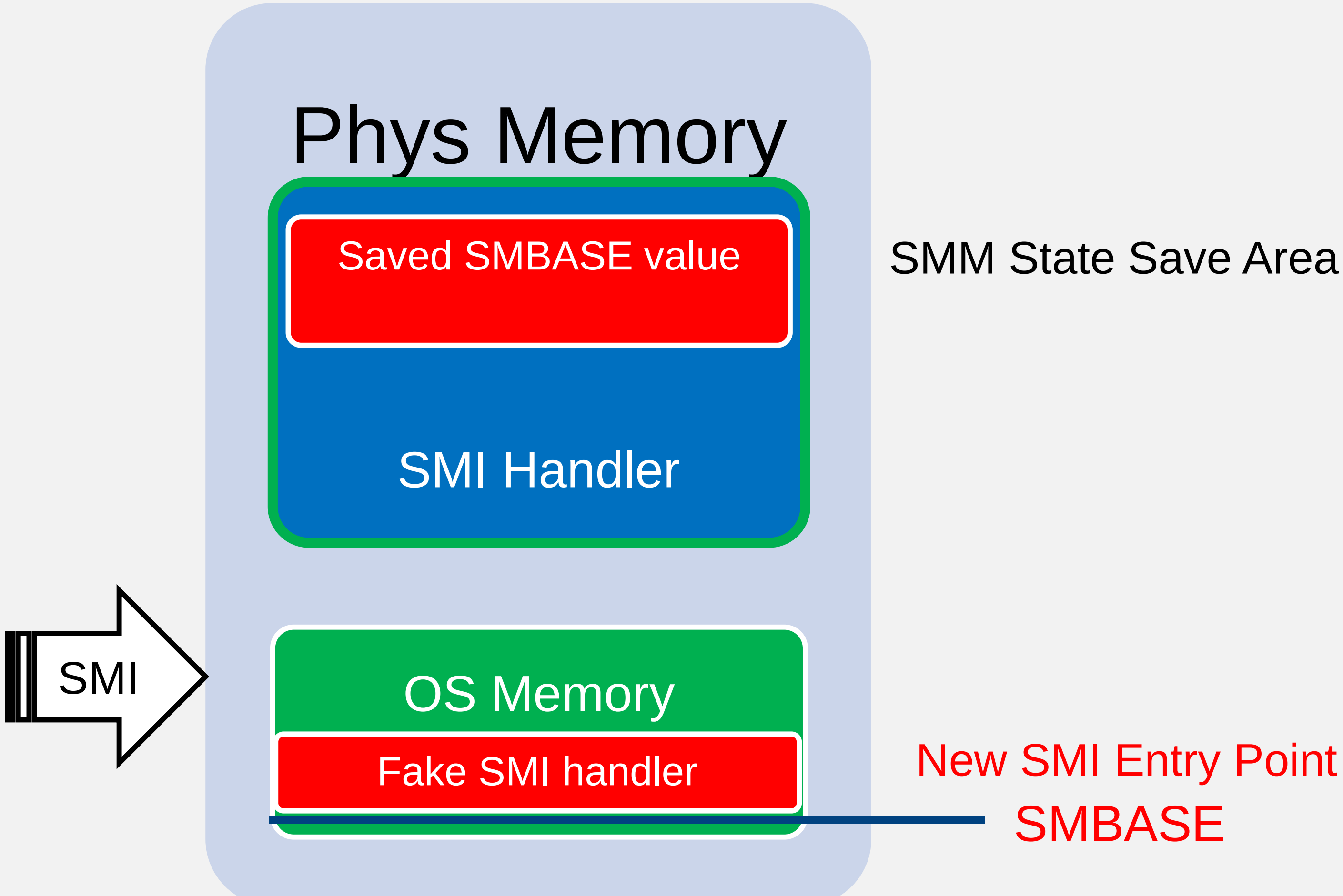
- Exploit prepares fake SMRAM with fake SMI handler outside of SMRAM

How does the attack work?



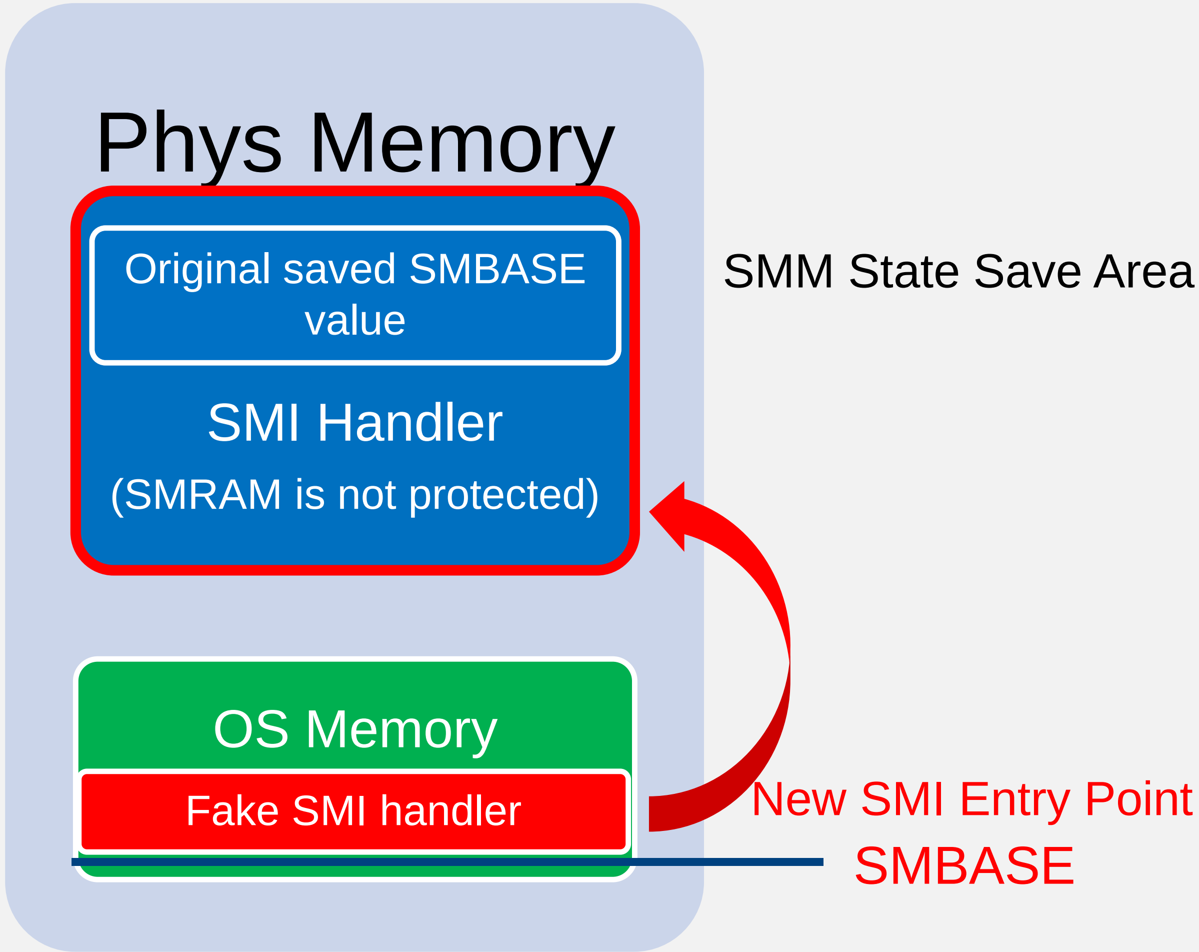
- Exploit triggers SMI w/ RBX pointing to saved SMBASE address in SMRAM
- SMI handler overwrites saved SMBASE on exploit's behalf with address of fake SMI handler outside of SMRAM (e.g. 0 PA)

How does the attack work?



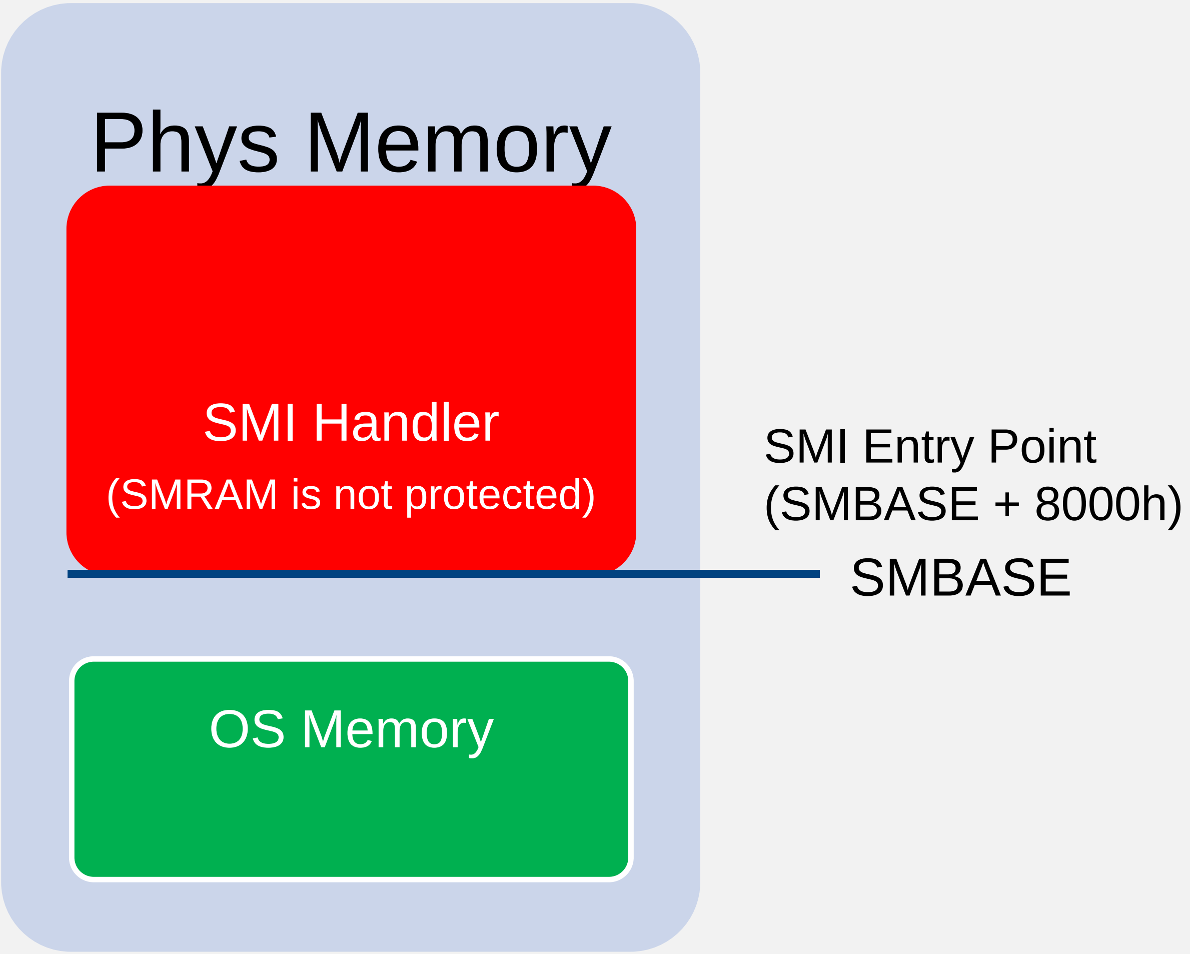
- Exploit triggers another SMI
- CPU executes fake SMI handler at new entry point outside of original protected SMRAM because SMBASE location changed

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

Exploiting SMI Input Pointers (Demo)

```
[+] 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
```

EDKII *CommBuffer*

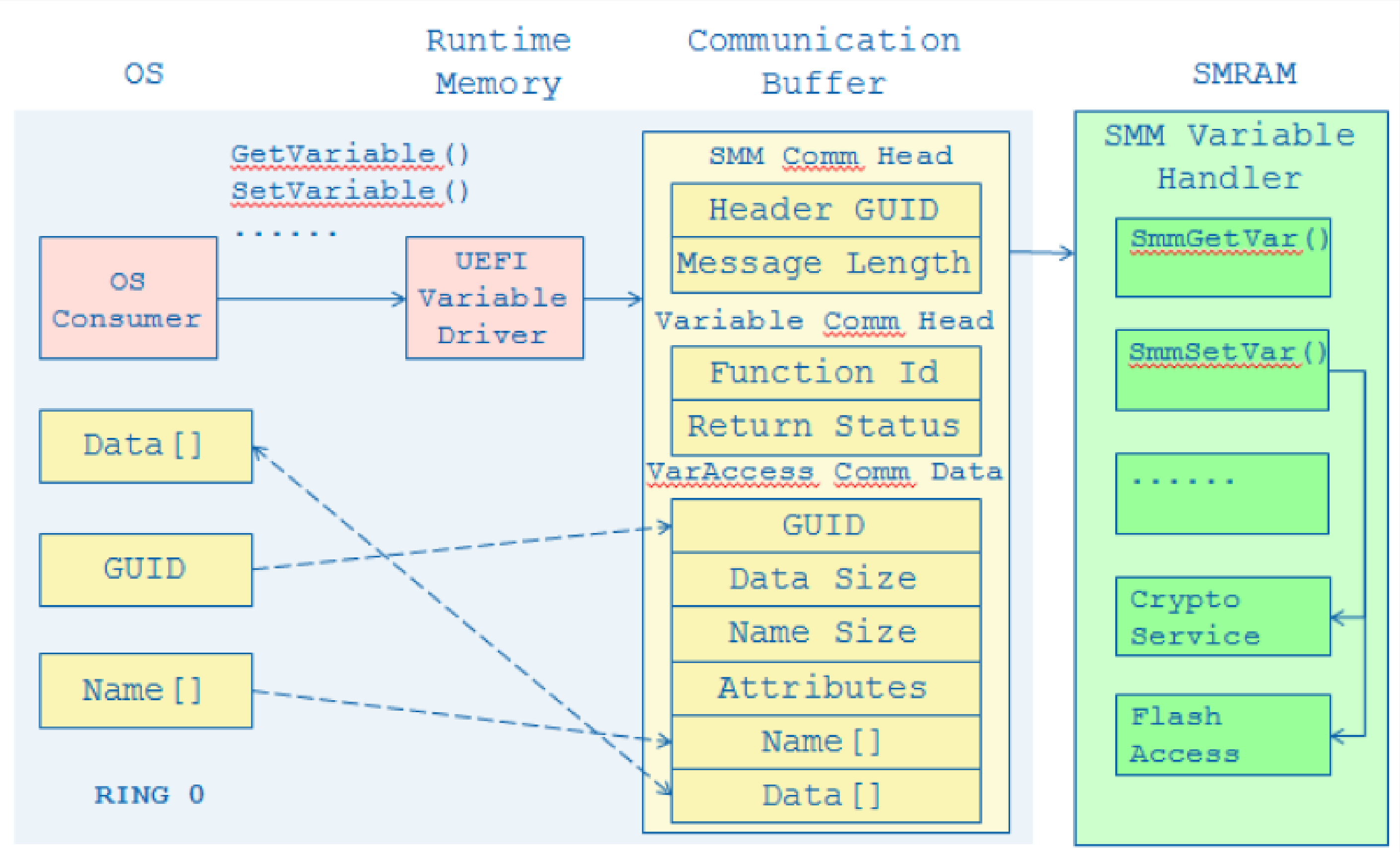


Figure 1 SMM Variable

- **`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 memory
- Contents of **`CommBuffer`** are specific to SMI handler. Variable SMI handler read UEFI variable GUID, Name and Data from **`CommBuffer`**
- Example: [VariableAuthenticated]() SMI Handler reads/writes UEFI variables from/to **`CommBuffer`**

Attacking *CommBuffer* Pointer

```
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);
```

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

Detecting & Mitigating Unvalidated SMI Input Pointers

Tools For Everybody, Free, And No One Will Go Away Unsatisfied!

Discovering SMI Pointer Vulns with CHIPSEC

```
# chipsec_main.py -m tools.smm.smm_ptr -a config,smm_config.ini

[x][ =======================================================================
[x][ Module: Testing SMI handlers for pointer validation vulnerabilities
[x][ =======================================================================
...
[*] Allocated memory buffer (to pass to SMI handlers) : 0x00000000DAAC3000
[*] >>> Testing SMI handlers defined in 'smm_config.ini'..
...

[*] testing SMI# 0x1F (data: 0x00) SW SMI 0x1F
[*] writing 0x500 bytes at 0x00000000DAAC3000
    > SMI 1F (data: 00)
      RAX: 0x5A5A5A5A5A5A5A5A
      RBX: 0x00000000DAAC3000
      RCX: 0x0000000000000000
      RDX: 0x5A5A5A5A5A5A5A5A
      RSI: 0x5A5A5A5A5A5A5A5A
      RDI: 0x5A5A5A5A5A5A5A5A
    < checking buffers contents changed at 0x00000000DAAC3000 +[29,32,33,34,35]
[!] DETECTED: SMI# 1F data 0 (rax=5A5A5A5A5A5A5A5A rbx=DAAC3000 rcx=0 rdx=...)

[-] <<< Done: found 2 potential occurrences of unchecked input pointers
```

```
[*] testing SMI# 0x1E (data: 0x00) SW SMI 0x1E ()
[*] writing 0x500 bytes at 0x00000000DAA69000
    > SMI 1E (data: 00)
      RAX: 0x5A5A5A5A5A5A5A5A
      RBX: 0x00000000DAA69000
      RCX: 0x0000000000000000
      RDX: 0x5A5A5A5A5A5A5A5A
      RSI: 0x5A5A5A5A5A5A5A5A
      RDI: 0x5A5A5A5A5A5A5A5A
    < checking buffers
    contents changed at 0x00000000DAA69000 +[0, 1, 258]
[!] DETECTED: SMI# 1E data 0 (rax=5A5A5A5A5A5A5A5A rbx=DAA69000 rcx=0 rdx=5A5A5A5A5A5A5A5A rsi

[*] testing SMI# 0x1F (data: 0x00) SW SMI 0x1F ()
[*] writing 0x500 bytes at 0x00000000DAA69000
    > SMI 1F (data: 00)
      RAX: 0x5A5A5A5A5A5A5A5A
      RBX: 0x00000000DAA69000
      RCX: 0x0000000000000000
      RDX: 0x5A5A5A5A5A5A5A5A
      RSI: 0x5A5A5A5A5A5A5A5A
      RDI: 0x5A5A5A5A5A5A5A5A
    < checking buffers
    contents changed at 0x00000000DAA69000 +[29, 32, 33, 34, 35]
[!] DETECTED: SMI# 1F data 0 (rax=5A5A5A5A5A5A5A5A rbx=DAA69000 rcx=0 rdx=5A5A5A5A5A5A5A5A rsi
[-] <<< Done: found 2 potential occurrences of unchecked input pointers
```

Wash pointers before consuming! They may be poisonous

- SMI code has to validate address/pointer (+ offsets) they receive from OS prior writing to it including returning status/error code
- Check input pointer + size for overlap with SMRAM range. E.g. use **`SmmIsBufferOutsideSmmValid`** function in EDKII
- Also validate pointers before reading. They can expose SMRAM

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

One Missed CALL

Attacking SMI Handlers Via SMM Call-Outs

#ThisVulnHadToBeGoneLongAgo

- **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)

These issues seem to come in packs

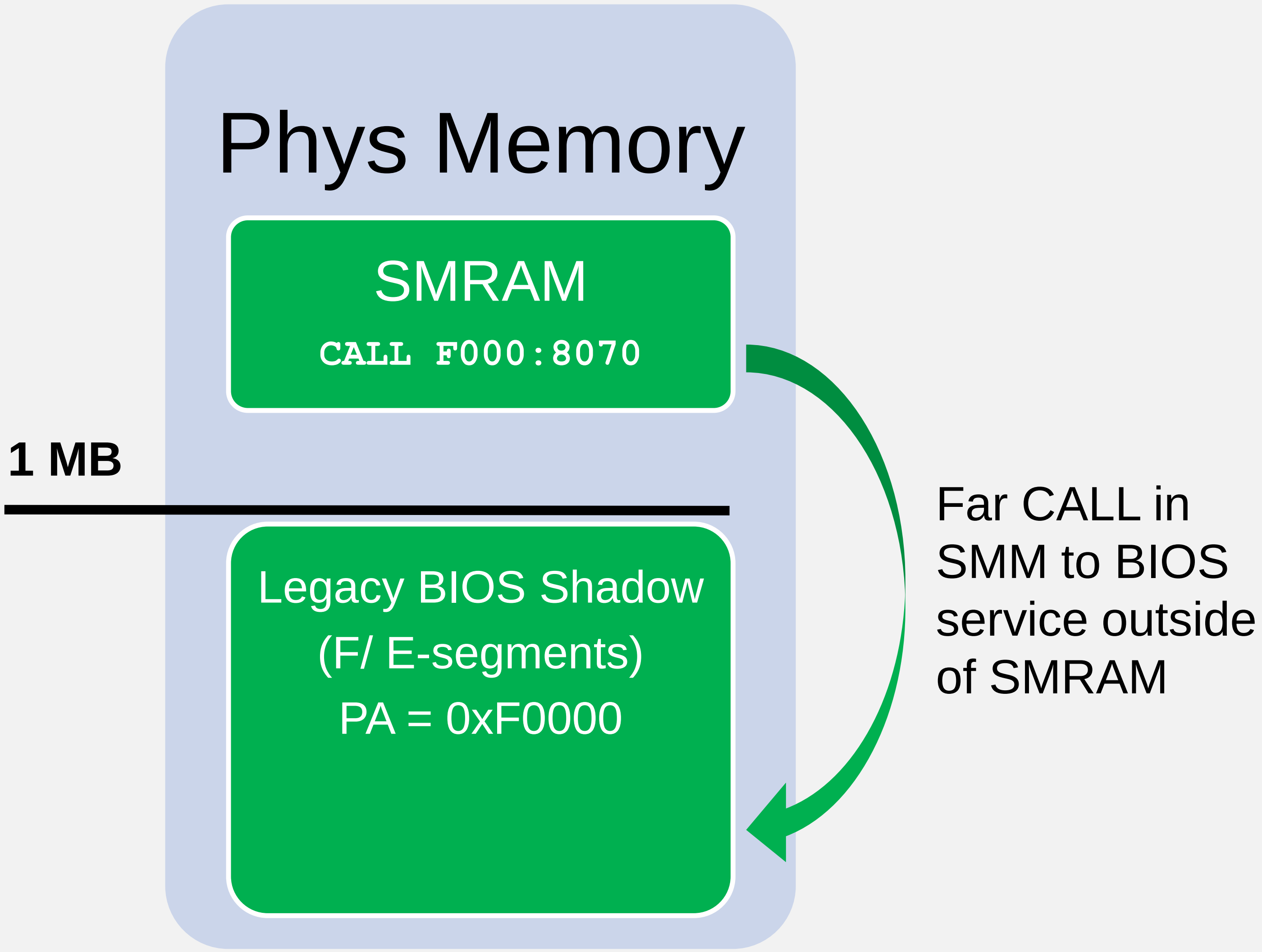
Disassembly of the code of $SMISS handler, one of SMI handlers in the BIOS firmware in ASUS Eee PC 1000HE system.

```
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
```

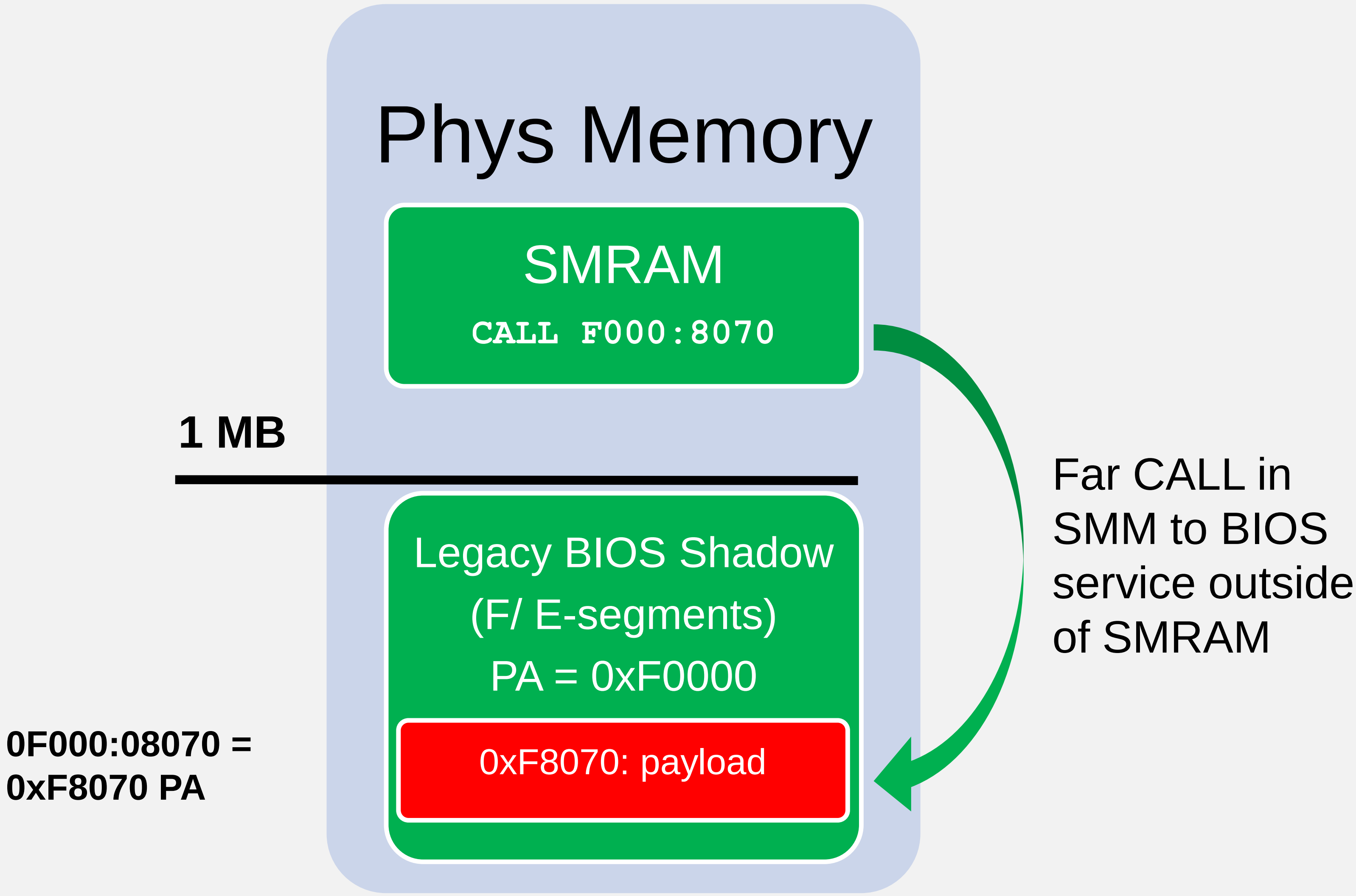
14 call-out vulnerabilities in one SMI handler!

[BIOS SMM Privilege Escalation Vulnerabilities]()

SMI Handlers Calling Out of SMRAM



SMI Handlers Calling Out of SMRAM

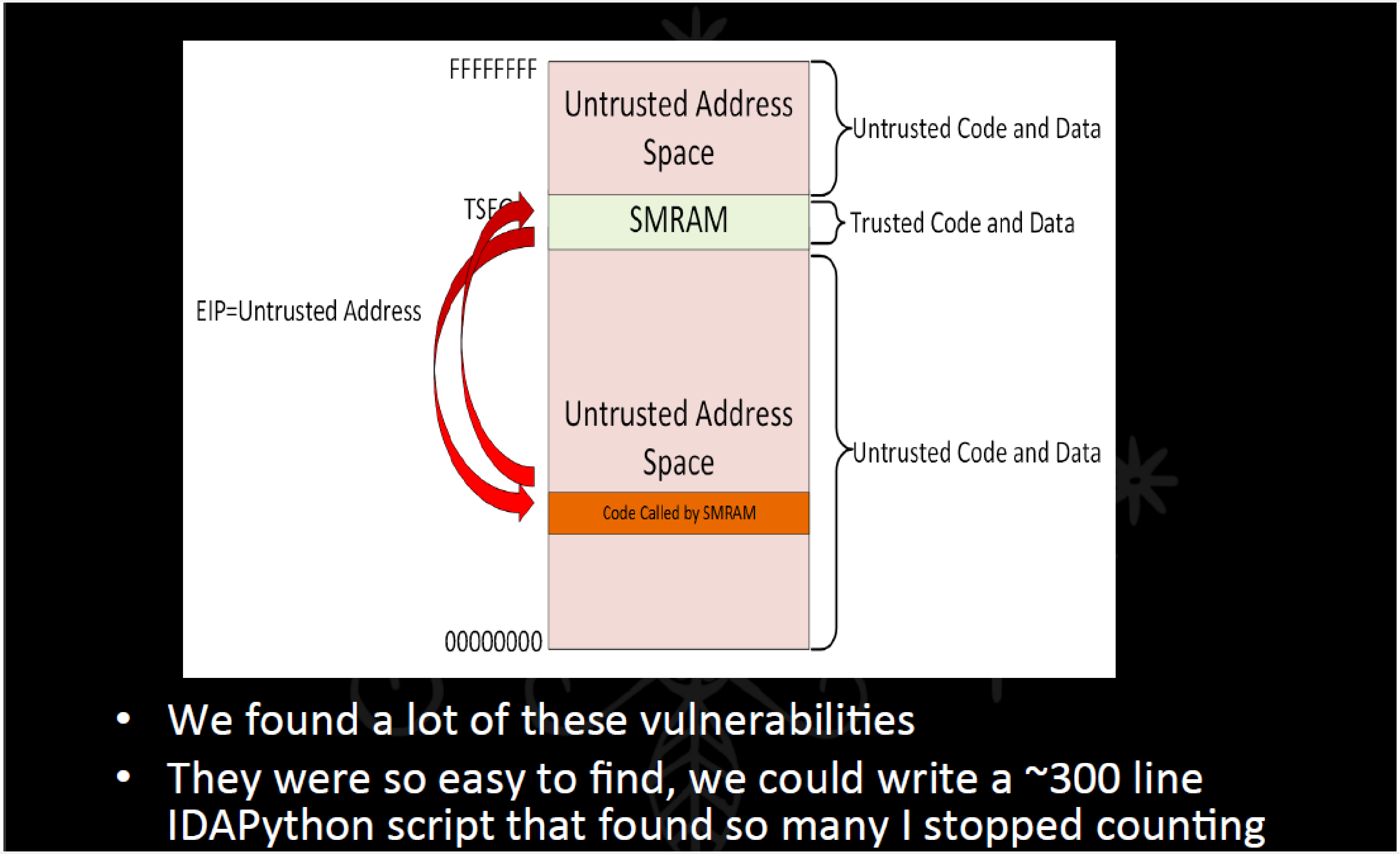


UEFI SMI Call-Outs

```
[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
```

DXE SMM drivers may call Runtime Services functions

Are SMI call-outs fixed yet?



[How Many Million BIOS Would You Like To Infect]() by LegbaCore

Detecting & Mitigating SMI Call-Outs

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 non in RT/BS/DXE service tables

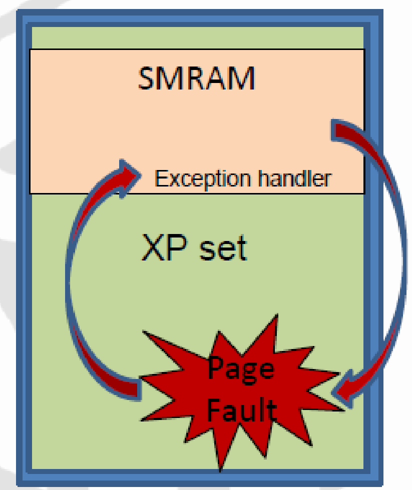
Hooking runtime UEFI services…

```
[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

- How it works
  - On every SMI, the same page tables are selected, paging and NX support is enabled
  - The original state is already saved in SMM save state to be restored on exit
  - The page tables have been configured with the XD bit in every PTE that does not overlap with SMRAM
  - The CPU throws a page fault on any attempt to fetch code that is located in a page outside of SMRAM

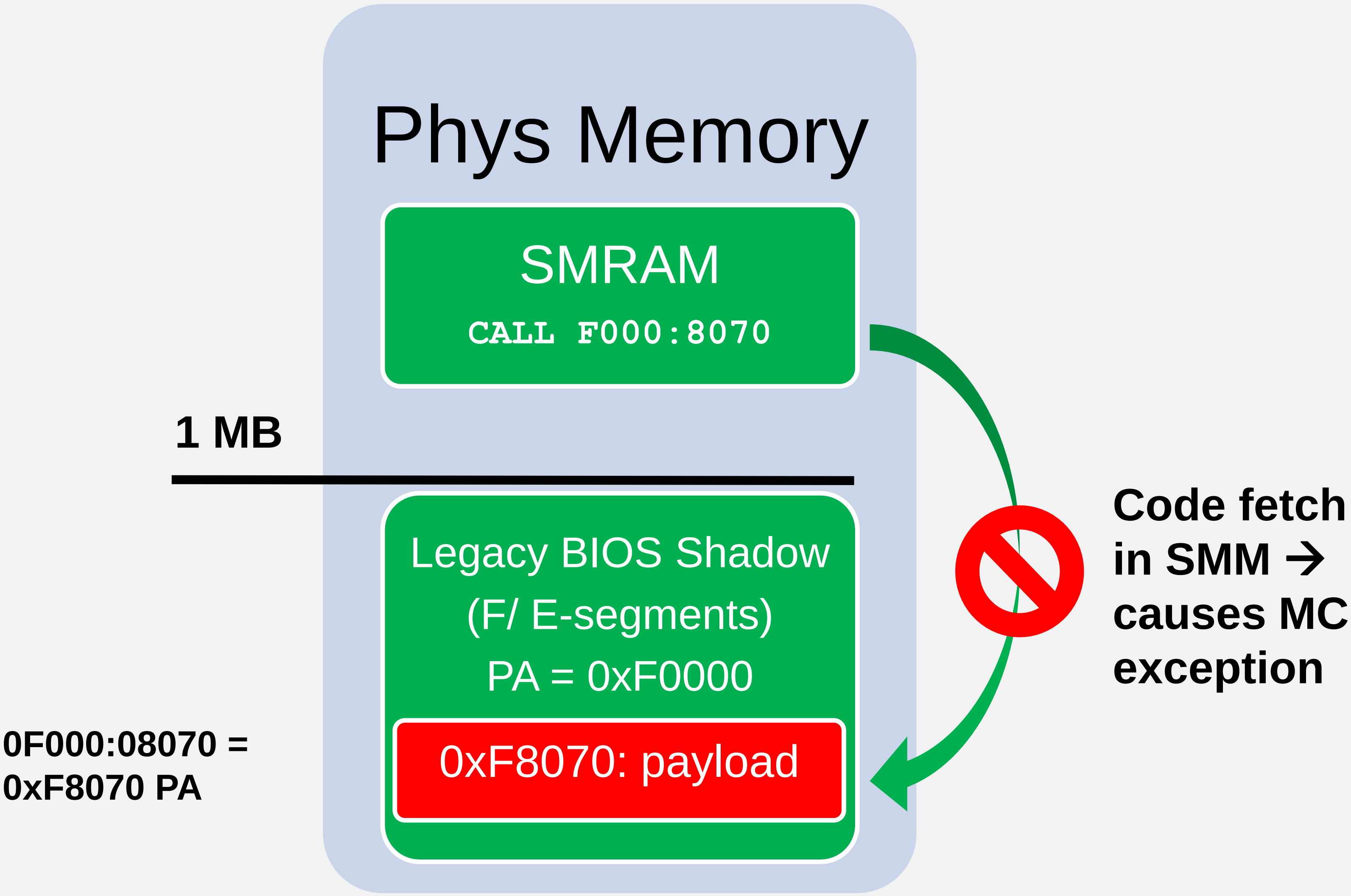


Mitigating SMM Call-Outs

1. Don’t call any function outside of protected SMRAM
   - Violates “No read down” rule of classical Biba integrity model

2. Enable SMM Code Access Check CPU protection
   - Available starting in Haswell based CPUs
   - Available if `MSR_SMM_MCA_CAP[58] == 1`
   - When enabled, attempts to execute code not within the ranges defined by the SMRR while inside SMM result in a Machine Check Exception

Blocking Code Fetch Outside of SMRAM



It's like trying to fit an octopus into a pair of tuxedo pants…

Image source: speckyboy.com

Why are we investing in CHIPSEC?

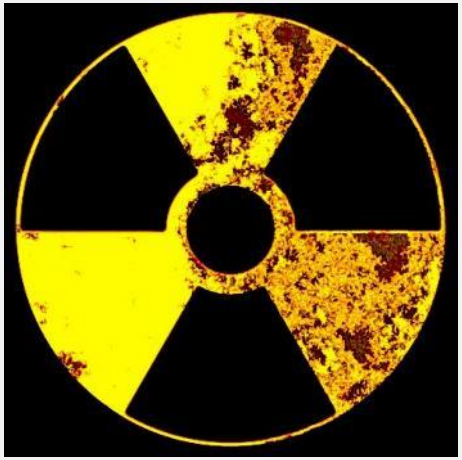
- Security researchers need a way to develop PoCs to test exploitability and impact of firmware issues
- OEM/BIOS vendors need a way to consistently run regression tests when building their firmware products
- We need security researchers to be able to capture their research in a way easily consumable by OEM/BIOS vendors
- Corporate IT needs a way to know how secure the systems they are about to deploy to 1000’s of employees
- It’s got to be open source so everyone could see what it’s testing and trust its results

Conclusions

BIOS/UEFI firmware security is an industry wide concern. Everyone is affected. There are often multiple issues of the same type. Some take years to mitigate

Researchers keep finding dragons and drive awareness. Classes of issues start to disappear. Now we have tools – use them to test your systems!

Many OEM/BIOS vendors are responsive to security issues, stepping up to improve security of their products (and using CHIPSEC now). HW protections are slowly being adopted



I was told that this road would take me to the ocean of death, and turned back halfway. Since then crooked, round-about, godforsaken paths stretch out before me.

Acknowledgements

We’d like to thank the following teams or individuals for making the BIOS and EFI firmware a bit more secure

- Nick Adams, Aaron Frinzell, Sugumar Govindarajan, Jiewen Yao, Vincent Zimmer, Bruce Monroe from Intel
- Corey Kallenberg, Xeno Kovah, Rafal Wojtczuk, @snare, Trammell Hudson, Dmytro Oleksiuk, Pedro Velaça
- UEFI Forum (USRT, USST), OEMs and IBVs who suggest solutions

References

1. Intel’s Advanced Threat Research Security of System Firmware
2. CHIPSEC: https://github.com/chipsec/chipsec
3. http://www.legbacore.com/Research.html
4. Low level PC attack papers by Xeno Kovah
5. MITRE Copernicus
6. Trianocore security advisories
7. UEFI Forum USRT

A little knowledge can be a dangerous thing...

Thank You!