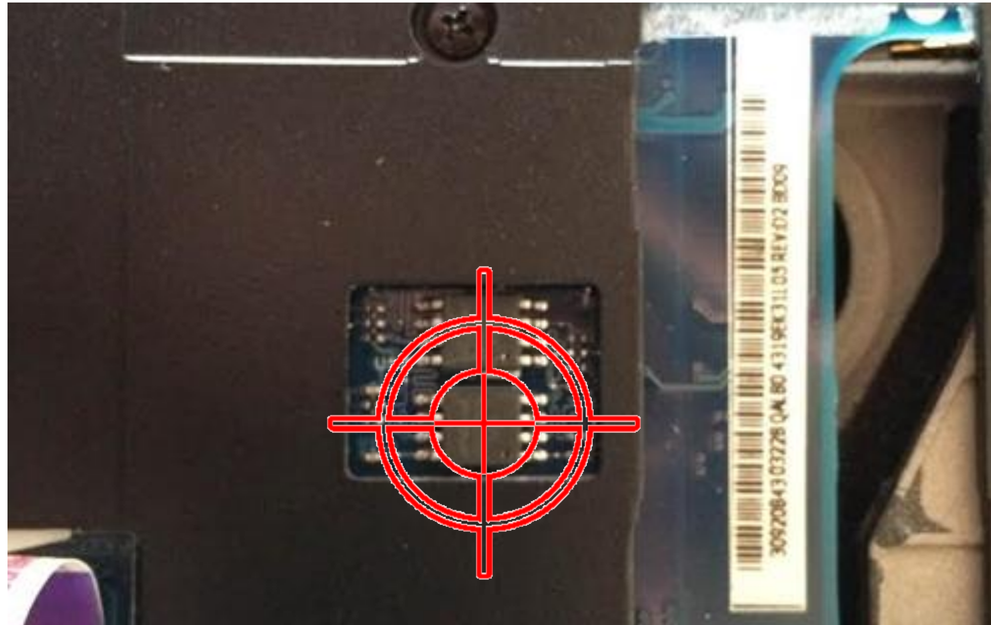
Attacks on UEFI Security

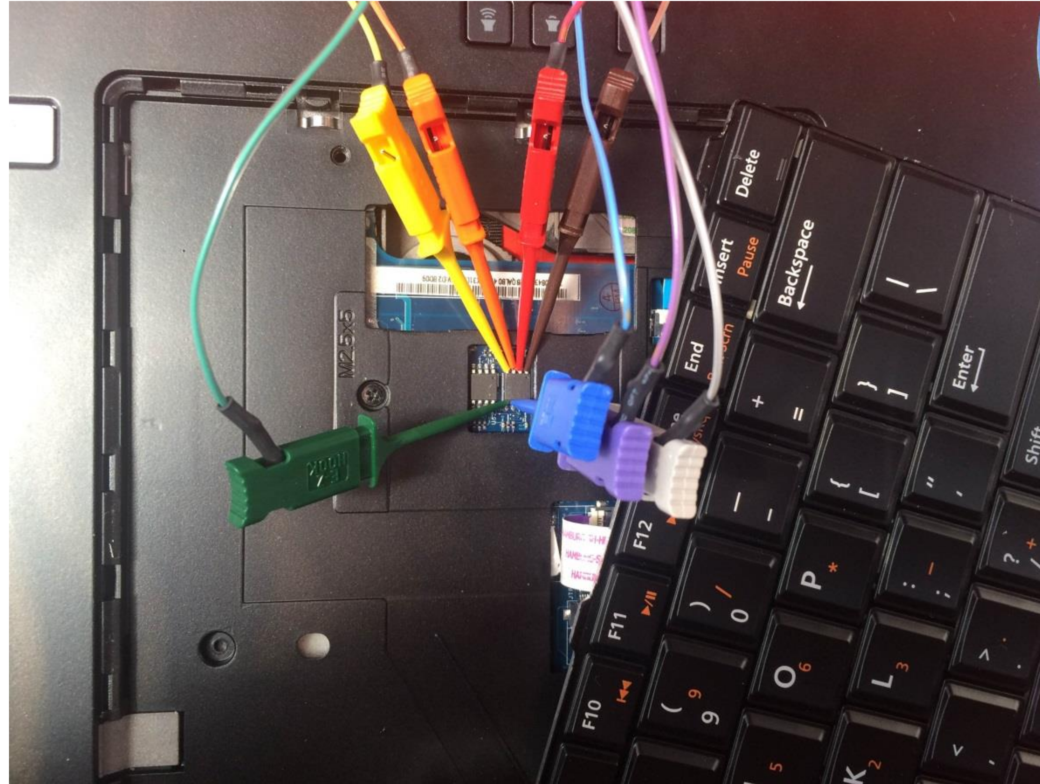
Rafal Wojtczuk <rafal@bromium.com>
Corey Kallenberg <coreykal@gmail.com>

Objective



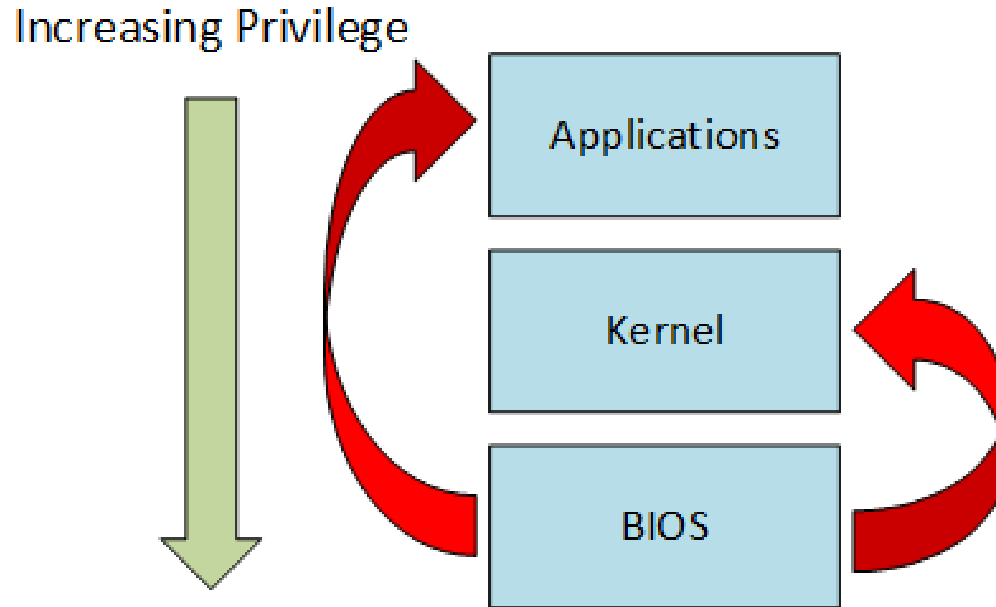
- Overwrite the contents of the firmware (UEFI), which is typically stored on a SPI flash chip that is soldered to the motherboard

Rules



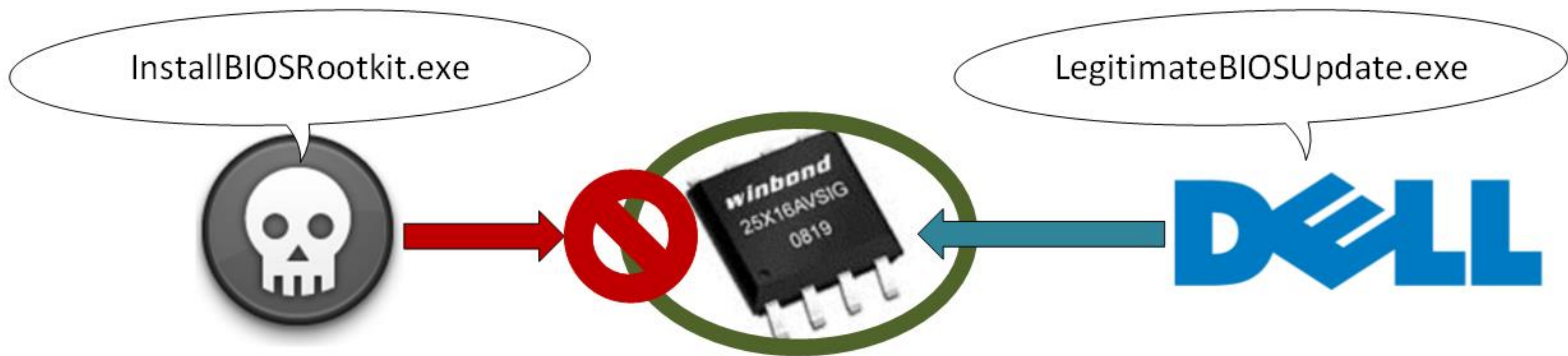
- Only software attacks against the firmware are considered
- With physical access, reprogramming the firmware is accomplished trivially with a flash programmer

Why Bother?

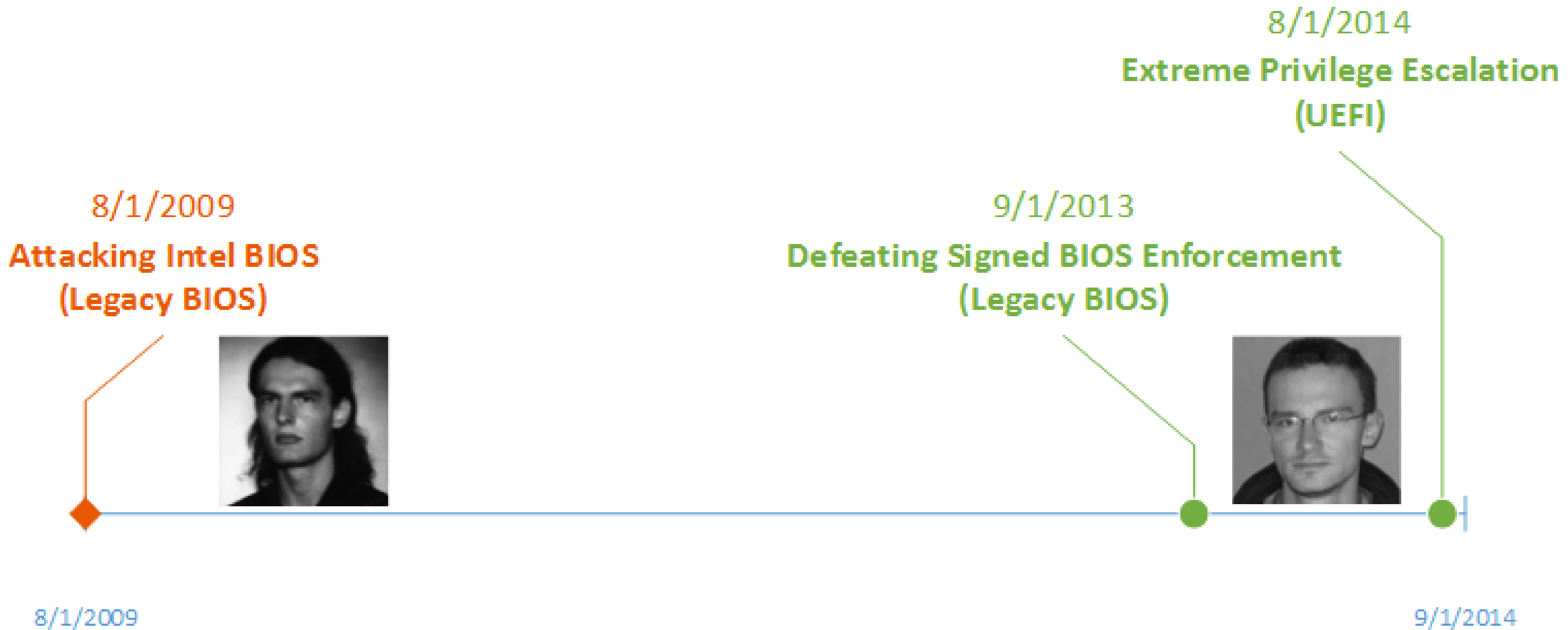


- The firmware can:
  - Compromise the rest of the software stack
  - Brick the platform
  - Survive OS reinstallations
- Ideal place for a rootkit!

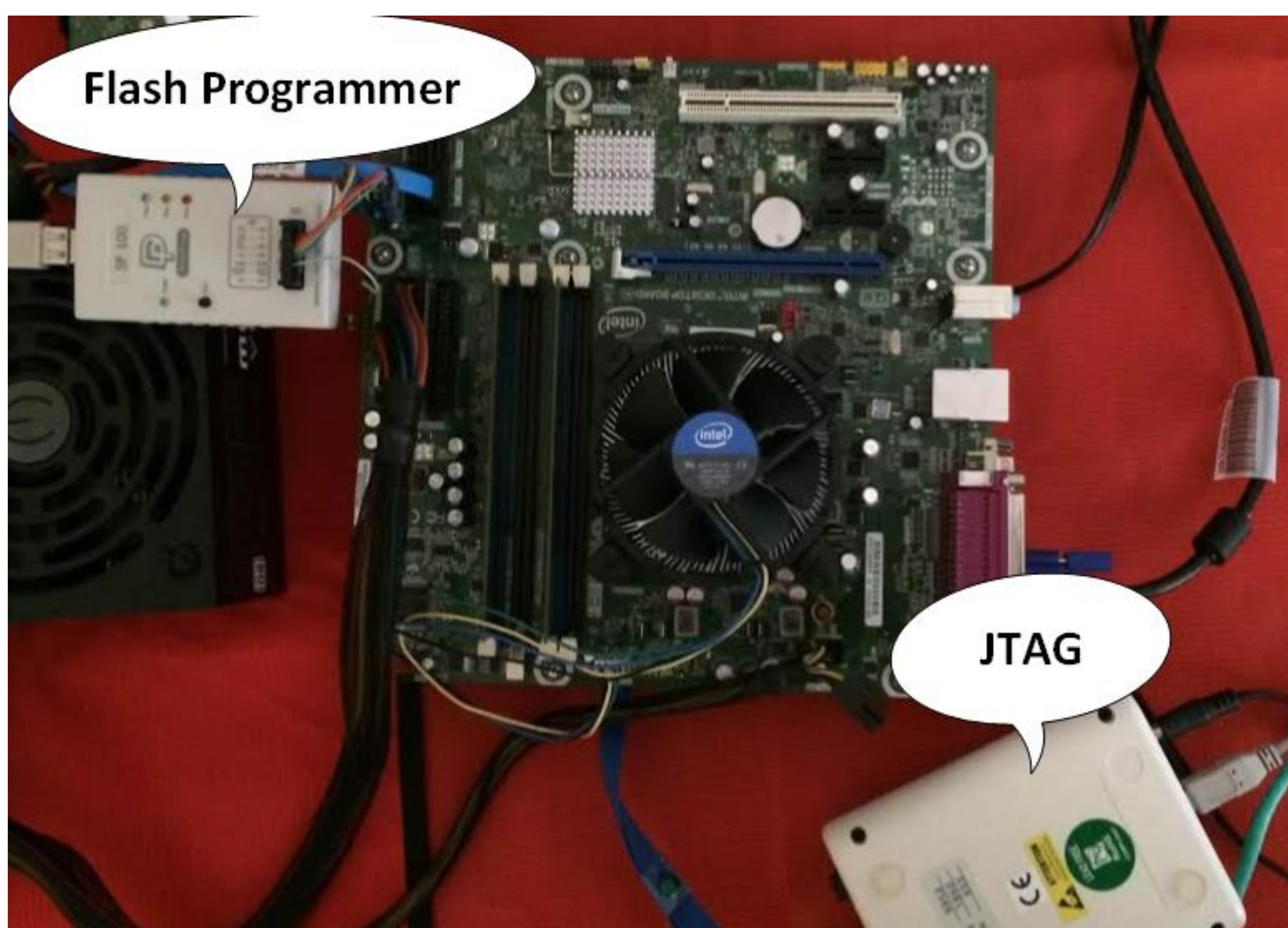
Features vs Security



- The chipset provides features to:
  - Reprogram the contents of the firmware
  - Protect the firmware against arbitrary programming attempts
- It's up to the OEM to utilize these features to:
  - Allow legitimate firmware updates
  - Deny malicious firmware programming attempts



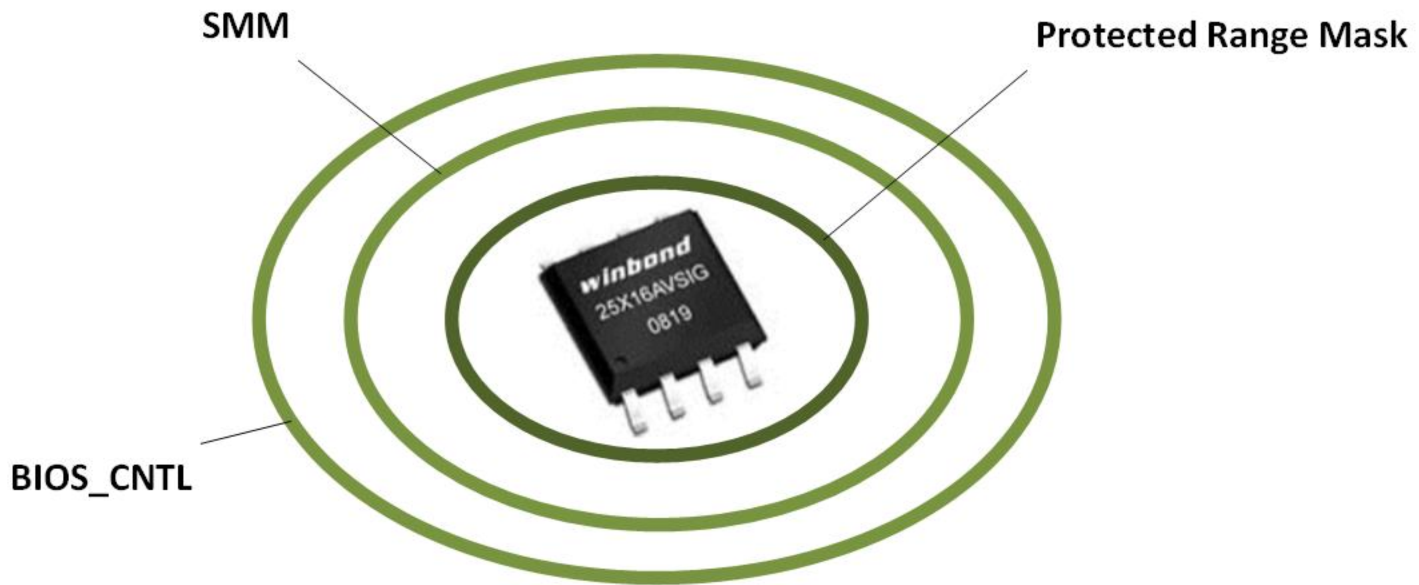
- 1st attack against flash protections presented by Wojtczuk and Tereshkin in 2009[1]
- 2nd and 3rd attack by Kallenberg et al[2][3]
- But these attacks were suboptimal…



- Previous attacks:
  - Complex memory corruption vulnerabilities
  - Required expensive and tedious testing to exploit
  - Difficult to port and reproduce
  - Extremely system dependent
- Unlikely to be exploited “in the wild” for these reasons

- Today we bestow onto you vulnerabilities which are:
  - Prevalent among UEFI and legacy BIOS systems
  - Result in reflash of firmware and/or SMM breakin
  - Straight forward enough to DIY, no exotic equipment needed

Multi-Layered Protection



- There are multiple layers of protection that prevent arbitrary flash programming attempts
- We will evaluate and then break through each layer in series

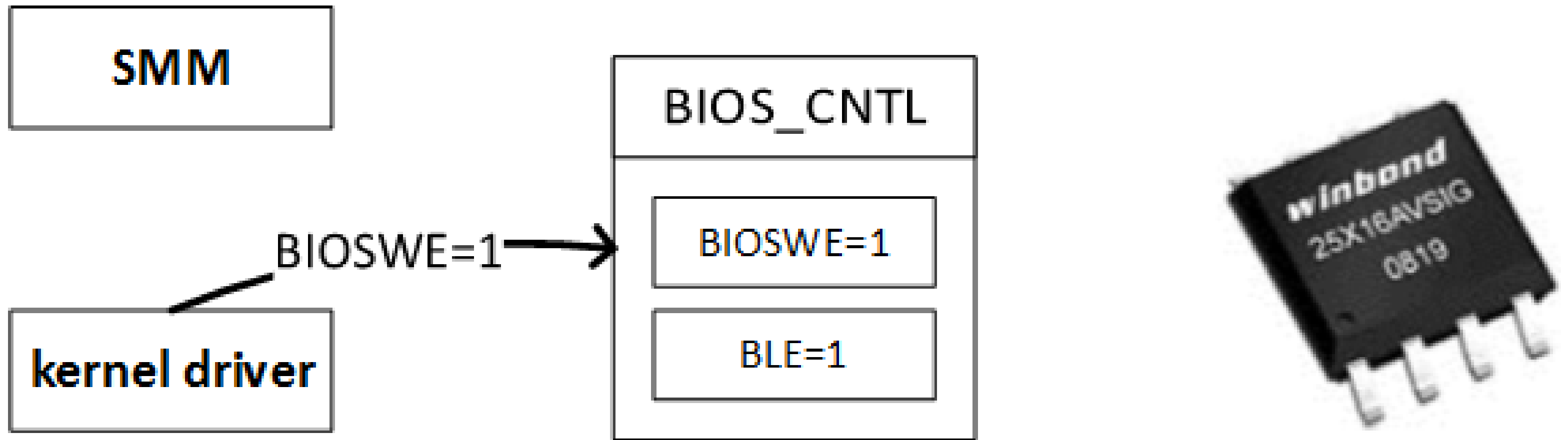
Layer 1: BIOS_CNTL

Bit	Description
1	**BIOS Lock Enable (BLE)** — R/WLO. 0 = Setting the BIOSWE will not cause SMIs. 1 = Enables setting the BIOSWE bit to cause SMIs. Once set, this bit can only be cleared by a PLTRST#
0	**BIOS Write Enable (BIOSWE)** — R/W. 0 = Only read cycles result in Firmware Hub I/F cycles. 1 = Access to the BIOS space is enabled for both read and write cycles. When this bit is written from a 0 to a 1 and BIOS Lock Enable (BLE) is also set, an SMI# is generated. This ensures that only SMI code can update BIOS.

- Write access to the flash is only possible if BIOSWE is set
- Setting BLE allows SMM to arbitrate write access to the flash

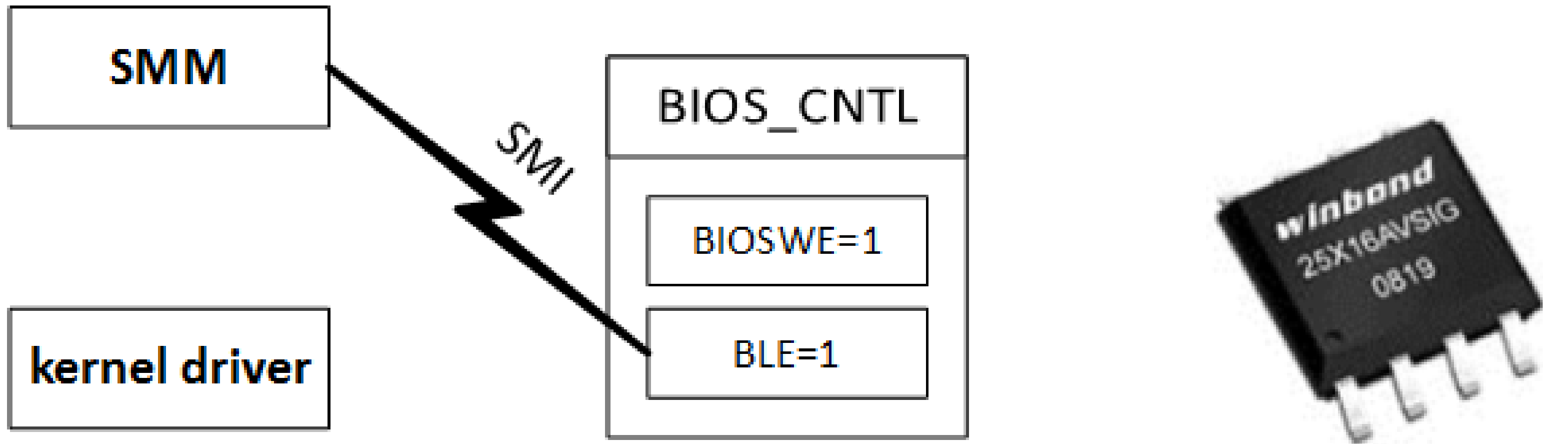
from: http://www.intel.com/content/www/us/en/chipsets/6-chipset-c200-chipset-datasheet.html

BIOS_CNTL Action 1/5



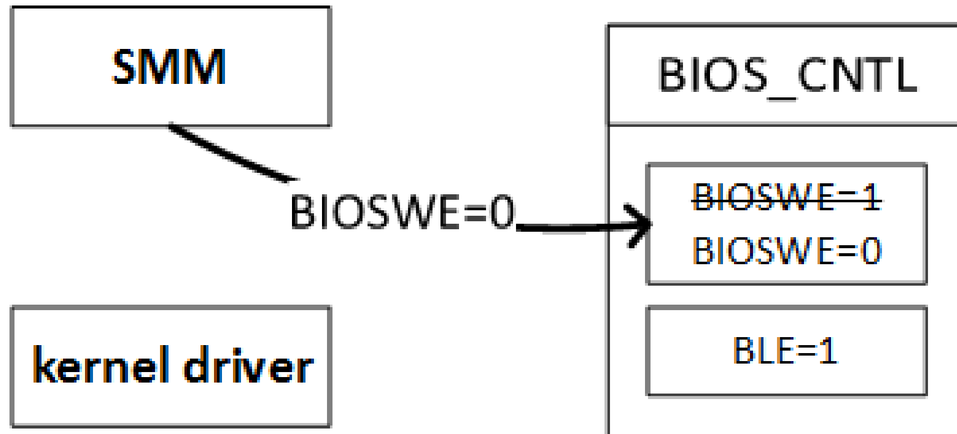
- Kernel driver attempts to set BIOSWE using a memory mapped write transaction to the chipset

BIOS_CNTL Action 2/5



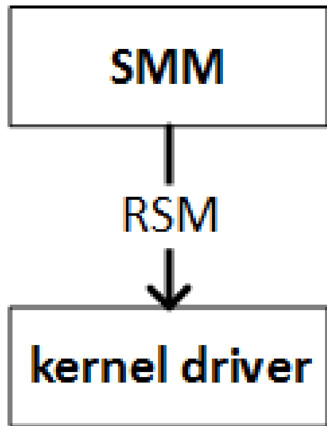
- Because BLE is set, an SMI occurs
- SMI handler begins executing

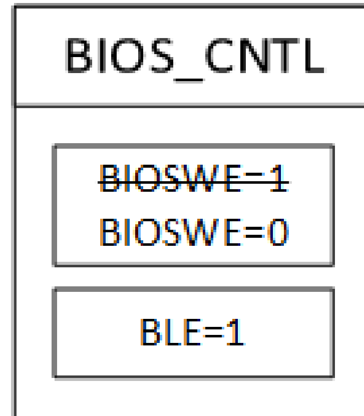
BIOS_CNTL Action 3/5



- SMM determines the write attempt is illegitimate and unsets BIOSWE

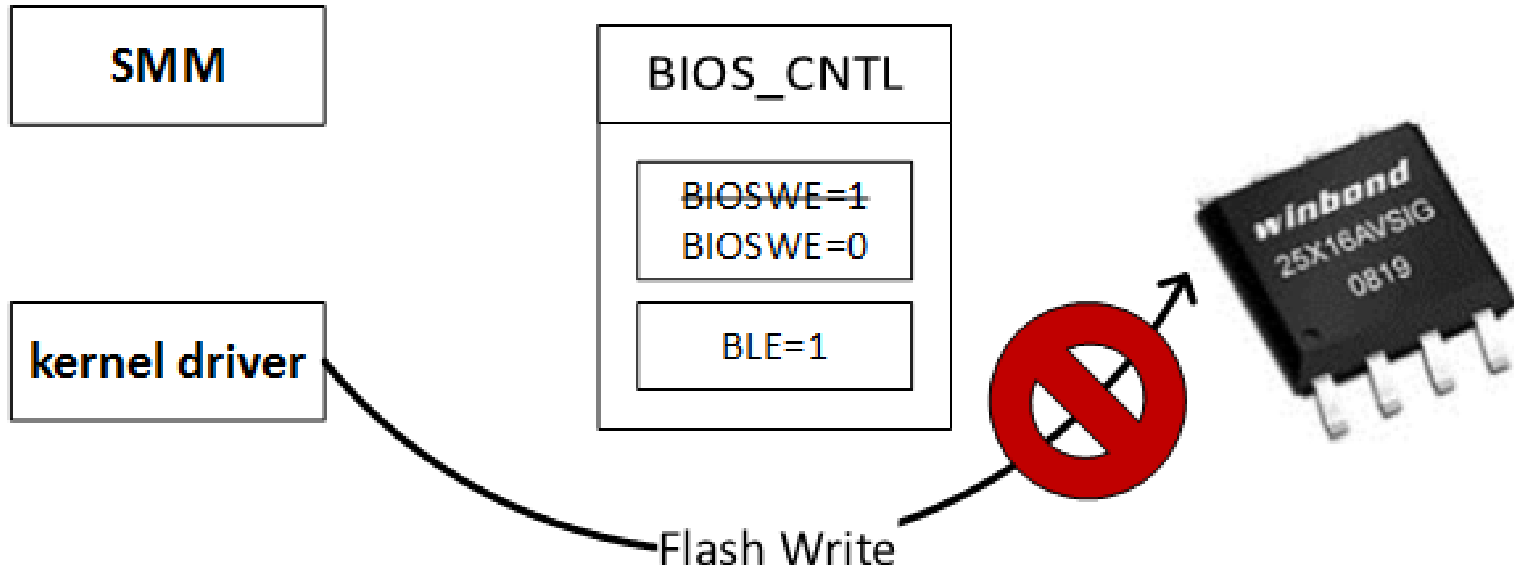
BIOS_CNTL Action 4/5





- Control is returned from SMM to the original thread

BIOS_CNTL Action 5/5



- Flash write cycle fails because BIOSWE is unset

BIOS_CNTL—BIOS Control Register (LPC I/F—D31:F0)

Offset Address: DCh | Attribute: R/WLO, R/W, RO
Default Value: 00h | Size: 8 bit
Lockable: No | Power Well: Core

Bit	Description
7:5	Reserved
4	**Top Swap Status (TSS)** — RO. This bit provides a read-only path to view the state of the Top Swap bit that is at offset 3414h, bit 0.

Old (ICH)

BIOS_CNTL—BIOS Control Register (LPC I/F—D31:F0)

Offset Address: DCh | Attribute: R/WLO, R/W, RO
Default Value: 20h | Size: 8 bit
Lockable: No | Power Well: Core

Bit	Description
7:6	Reserved
5	**SMM BIOS Write Protect Disable (SMM_BWP)**—R/WLO. This bit set defines when the BIOS region can be written by the host. 0 = BIOS region SMM protection is disabled. The BIOS Region is writable regardless if Processors are in SMM or not. (Set this field to 0 for legacy behavior) 1 = BIOS region SMM protection is enabled. The BIOS Region is not writable unless all Processors are in SMM.

New (PCH)

- Move to PCH chipset architecture introduced new feature to BIOS_CNTL with "interesting language"
- "BIOS Region is not writable unless all processors are in SMM" ???

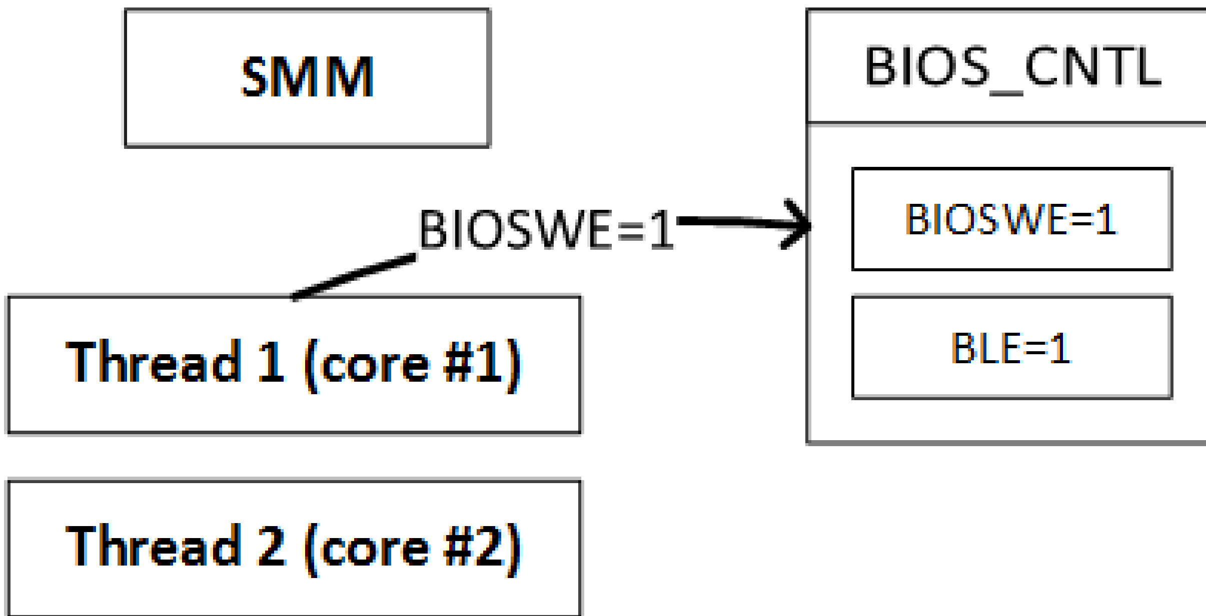
http://www.intel.com/content/dam/doc/datasheet/io-controller-hub-10-family-datasheet.pdf

http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/5-chipset-3400-chipset-datasheet.pdf

Speed Racer!

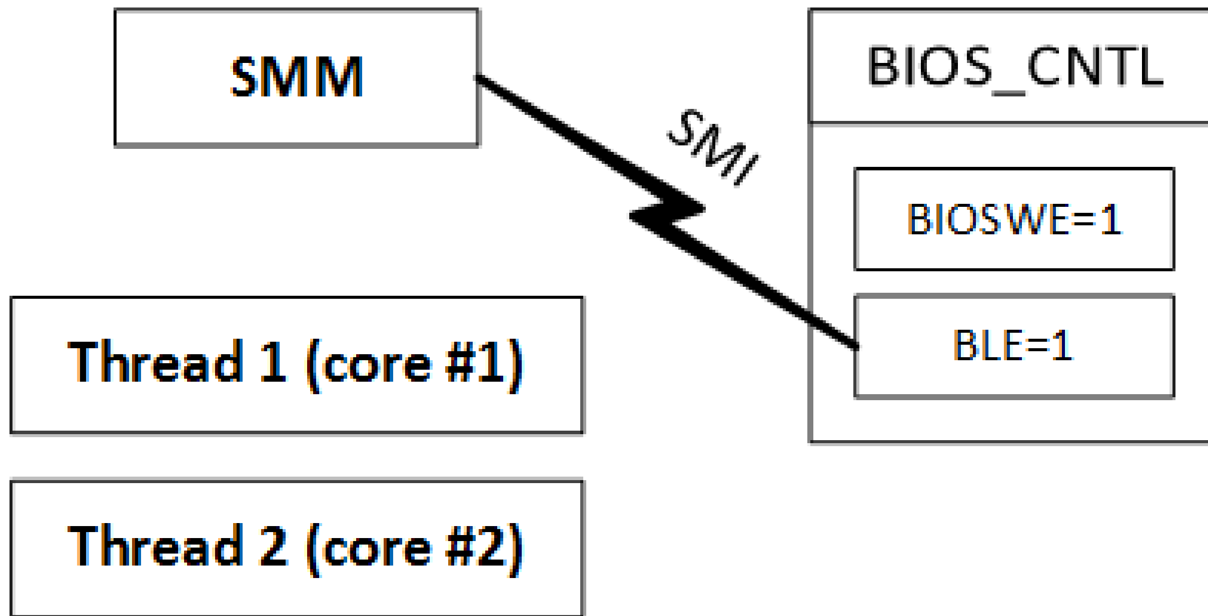
- It appears that Intel was patching a latent race condition in BIOS_CNTL protections!
- In private conversations, Sam Cornwell and John Butterworth of MITRE suggested this might be an issue!

BIOS_CNTL Race 1/4



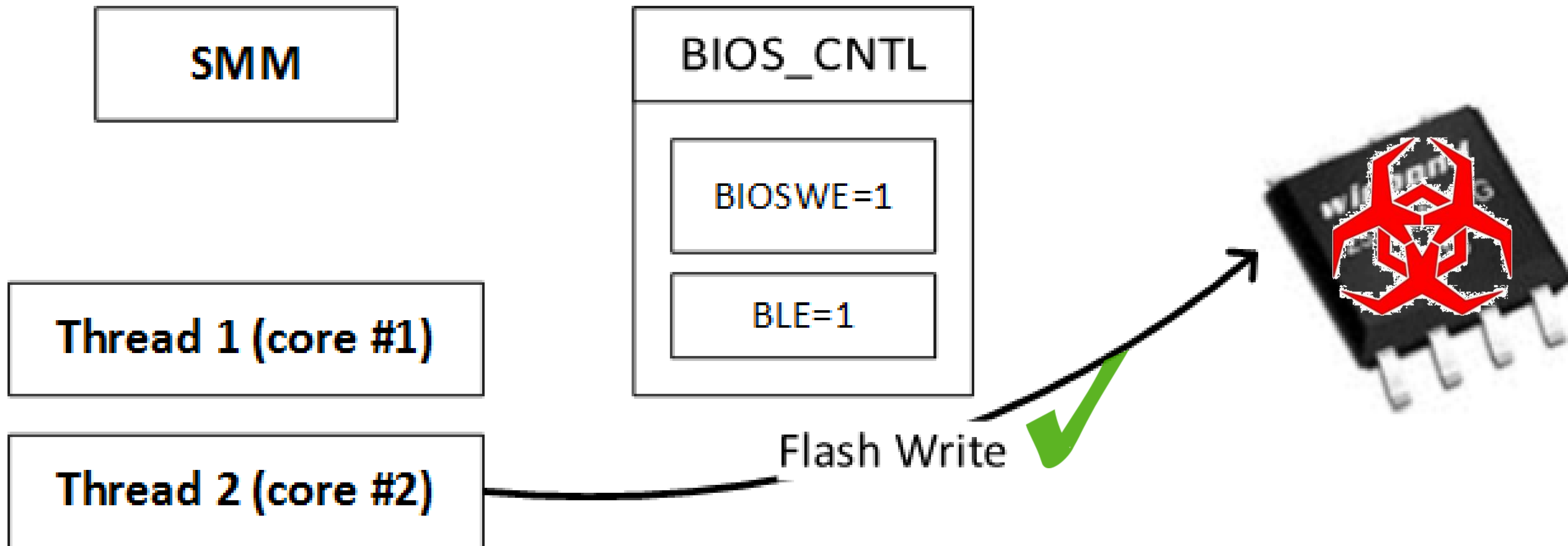
- This time we consider a multicore environment
- Core 1 begins the process by write enabling the flash

BIOS_CNTL Race 2/4



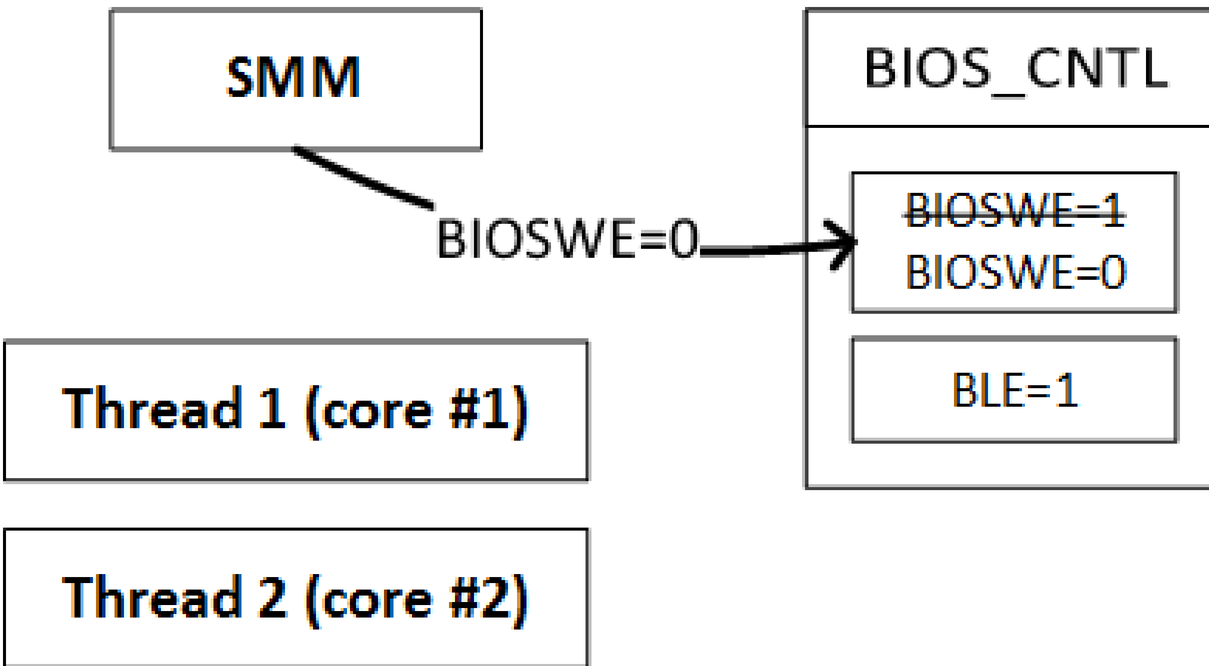
- Because BLE is set, an SMI is generated and core 1 immediately enters SMM

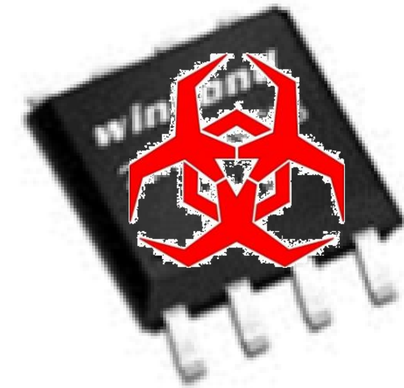
BIOS_CNTL Race 3/4



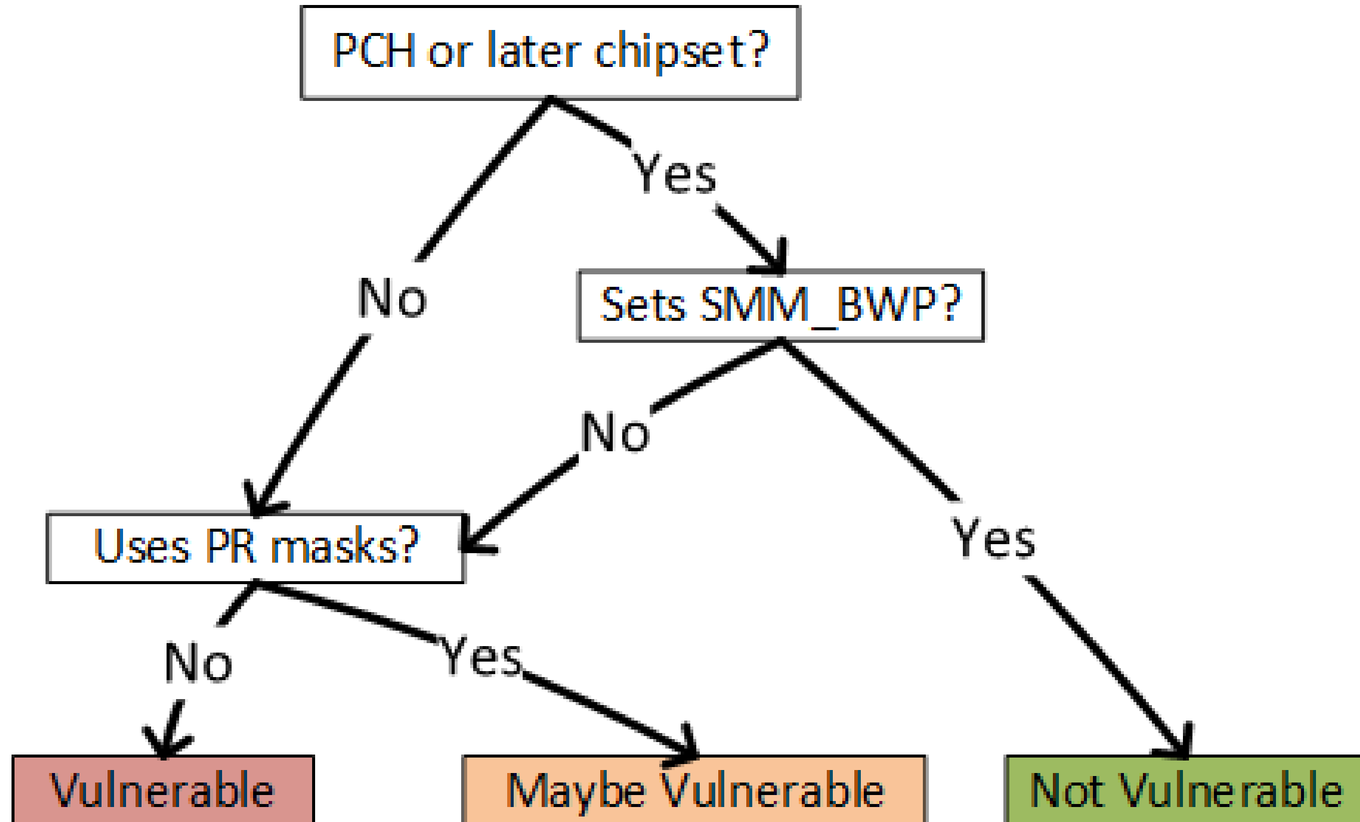
- Although core 2 will also enter SMM, it does not happen instantaneously.
- Core 2 has a small window in which to attempt flash write operations

BIOS_CNTL Race 4/4

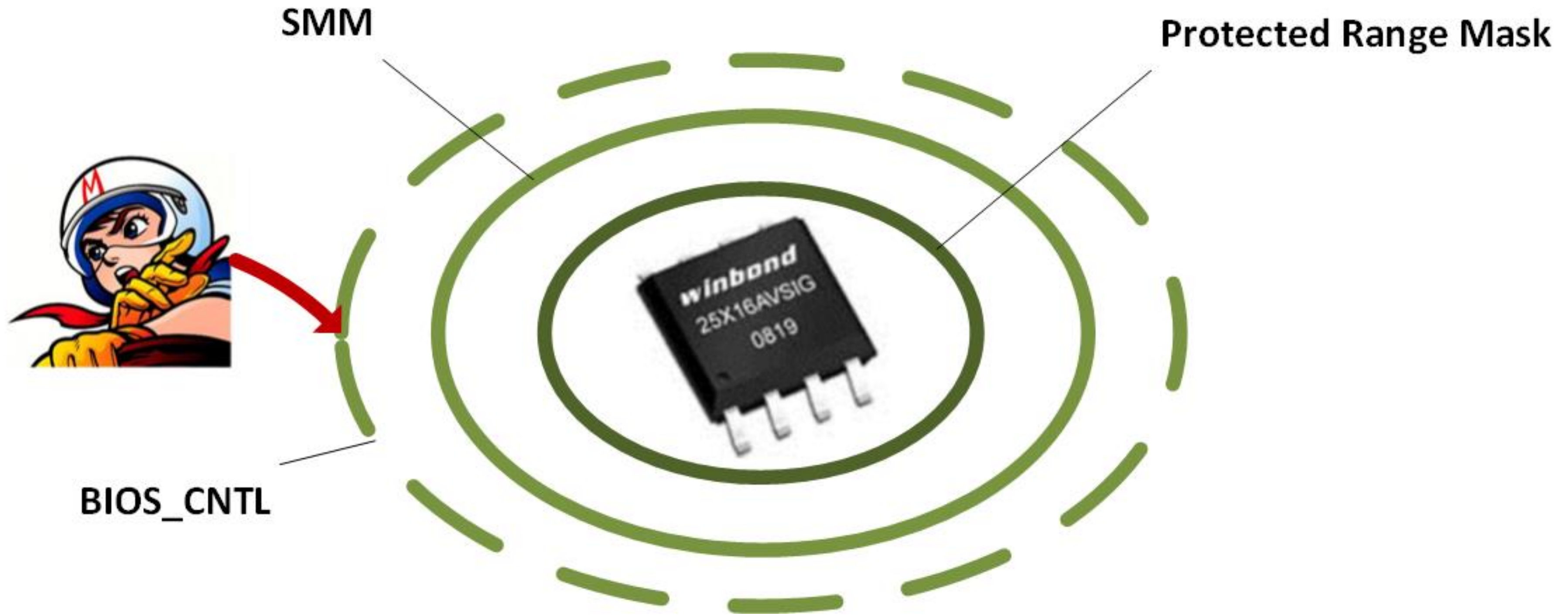




- The SMI handler unsets BIOSWE, but it's already too late.



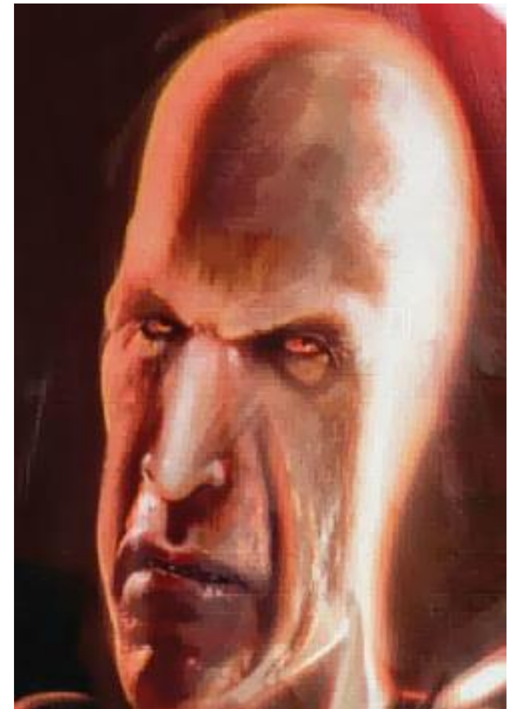
- “speed racer” assigned CERT VU#766164
- On systems with PCH chipsets, setting SMM_BWP resolves the issue, but adoption rate of SMM_BWP appears sporadic[4].
- Vulnerable systems are trivially exploited with a pair of kernel drivers
  - One for setting BIOSWE in a tight loop
  - Another for attempting the flash programming operation in a tight loop
  - No penalty for failing, so you can just brute force!



- BLE/BWE protection can be defeated by speed racer in the absence of SMM_BWP
- If SMM_BWP is supported and utilized, we are forced to break into SMM to continue our assault on the firmware

Attacking SMM: Inspired by the misery of Darth Venamis

- In the star wars universe, Darth Venamis is kept comatose by Darth Plagueis for the purpose of exploitation
- Perhaps we can put UEFI into a coma for exploitative purposes as well?



1	**BIOS Lock Enable (BLE)** — R/WLO. 0 = Setting the BIOSWE will not cause SMIs. 1 = Enables setting the BIOSWE bit to cause SMIs. Once set, this bit can only be cleared by a PLTRST#

15	**Flash Configuration Lock-Down (FLOCKDN)** — R/W/L. When set to 1, those Flash Program Registers that are locked down by this FLOCKDN bit cannot be written. Once set to 1, this bit can only be cleared by a hardware reset due to a global reset or host partition reset in an Intel® ME enabled system.

A reset in which the host platform is reset and PLTRST# is asserted is called a Host Reset or Host Partition Reset. Depending on the trigger, a host reset may also result in

- The bits that lock down SMM and the firmware are cleared during a reset
- “sleep”/”suspend” are typically implemented as an ACPI S3 sleep, which results in these lockdown bits being cleared
- S3 sleep = dark jedi coma

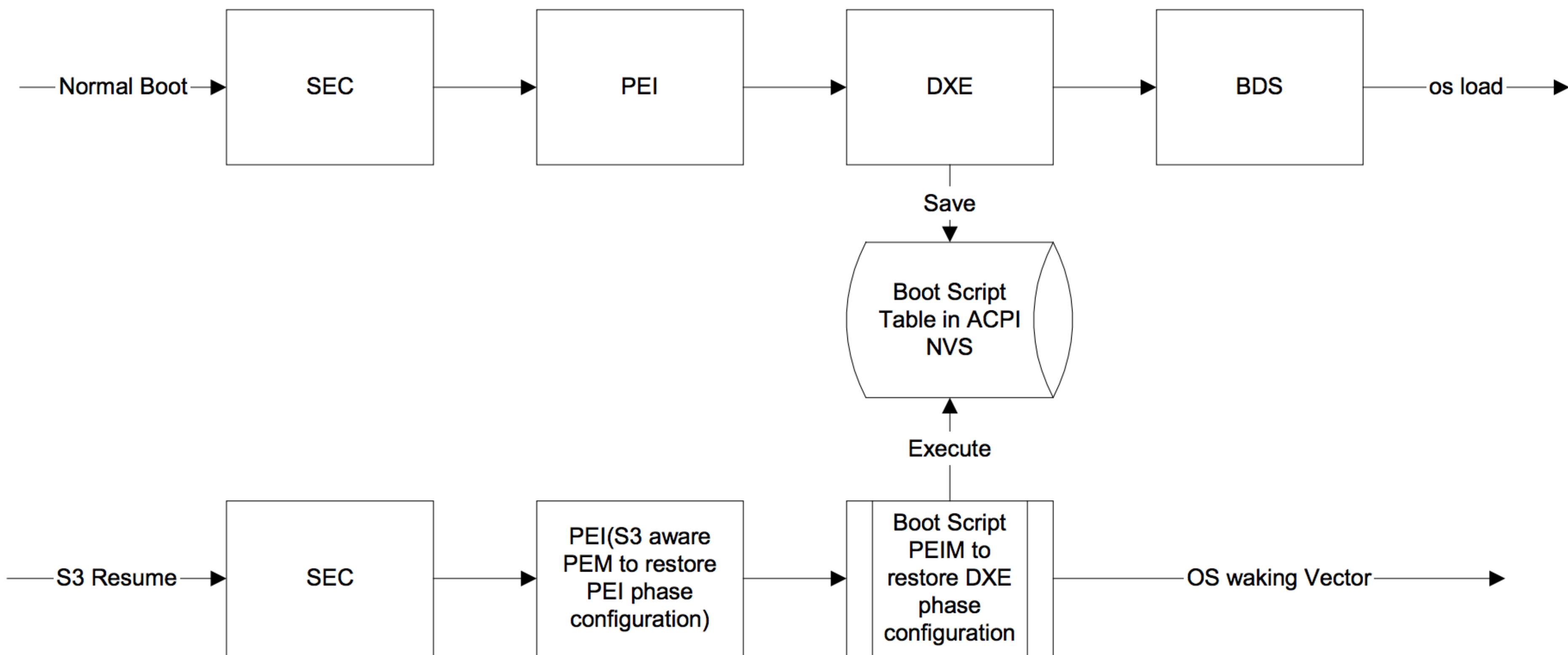


Figure 2-2. Role of Boot Script Usage in S3 Resume Boot Path

- During boot, the "platform configuration" is saved to a "boot script" so that s3 resume can happen more efficiently
- Included in the boot script are the contents of registers involved in locking down the platform
  - Such as TSEG and BIOS_CNTL

http://www.intel.com/content/dam/doc/guide/efi-boot-script-specification-v091.pdf

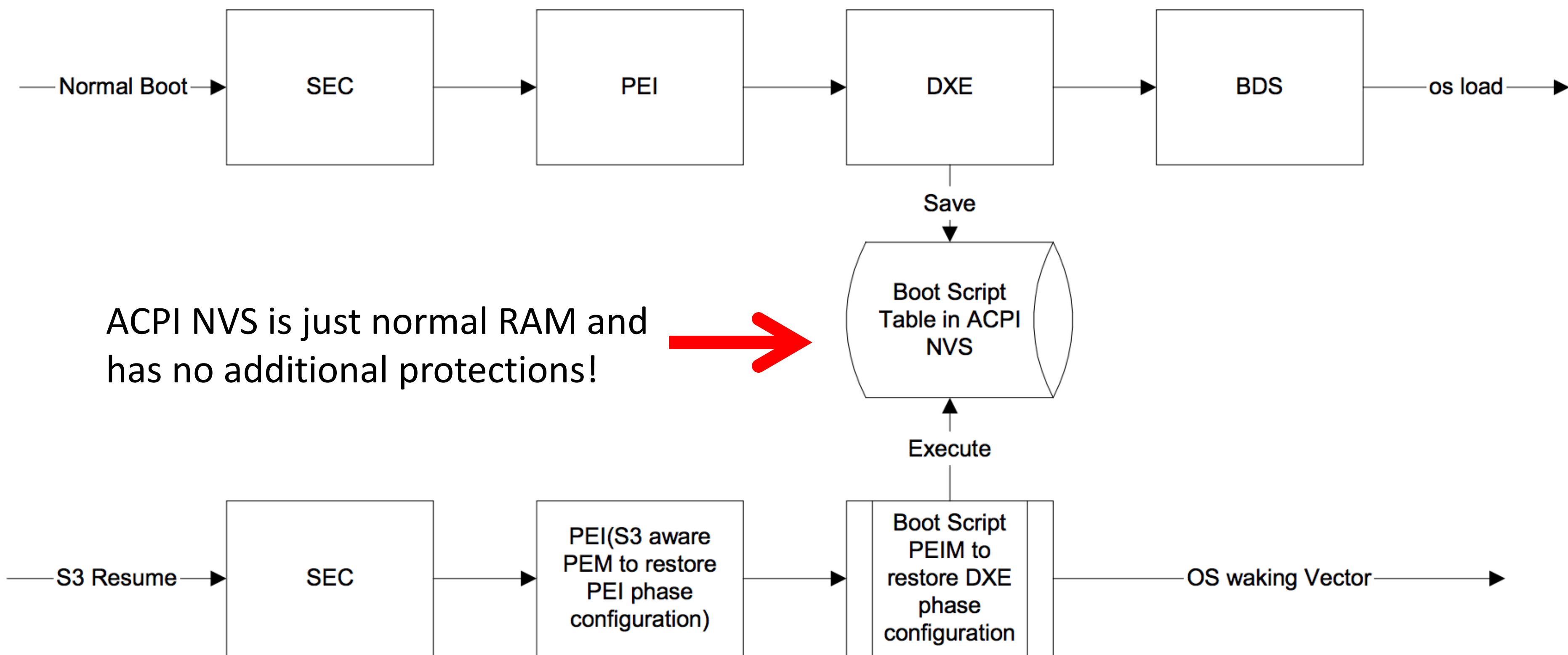


Figure 2-2. Role of Boot Script Usage in S3 Resume Boot Path

- Contents of the boot script were stored in ACPI NVS (unprotected) RAM on the consumer systems we looked at
- Attacker with access to physical memory could manipulate boot script contents

http://www.intel.com/content/dam/doc/guide/efi-boot-script-specification-v091.pdf

Boot Script

- From [12] “During a normal boot, DXE drivers record the platform’s configuration in the boot script, which is saved in NVS. During the S3 resume boot path, a boot script engine executes the script, thereby restoring the configuration.”
- “The chipset configuration can be viewed as a series of memory, I/O, and PCI configuration operations, which DXE drivers record in the Framework boot script. During an S3 resume, a boot script engine executes the boot script to restore the chipset settings.”

```
#define EFI_BOOT_SCRIPT_IO_WRITE_OPCODE                 0x00
#define EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE            0x01
#define EFI_BOOT_SCRIPT_MEM_WRITE_OPCODE                0x02
#define EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE           0x03
#define EFI_BOOT_SCRIPT_PCI_CONFIG_WRITE_OPCODE         0x04
#define EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE    0x05
#define EFI_BOOT_SCRIPT_SMBUS_EXECUTE_OPCODE            0x06
#define EFI_BOOT_SCRIPT_STALL_OPCODE                    0x07
#define EFI_BOOT_SCRIPT_DISPATCH_OPCODE                 0x08
```

EFI_BOOT_SCRIPT_DISPATCH_OPCODE

Summary

Adds a record for dispatching specified arbitrary code into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
 (EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN   struct _EFI_BOOT_SCRIPT_SAVE_PROTOCOL    *This,
  IN   UINT16                                   TableName,
  IN   UINT16                                   OpCode,
  IN   EFI_PHYSICAL_ADDRESS                     EntryPoint
  )
```

EntryPoint

Entry point of the code to be dispatched. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *EFI 1.10 Specification.*

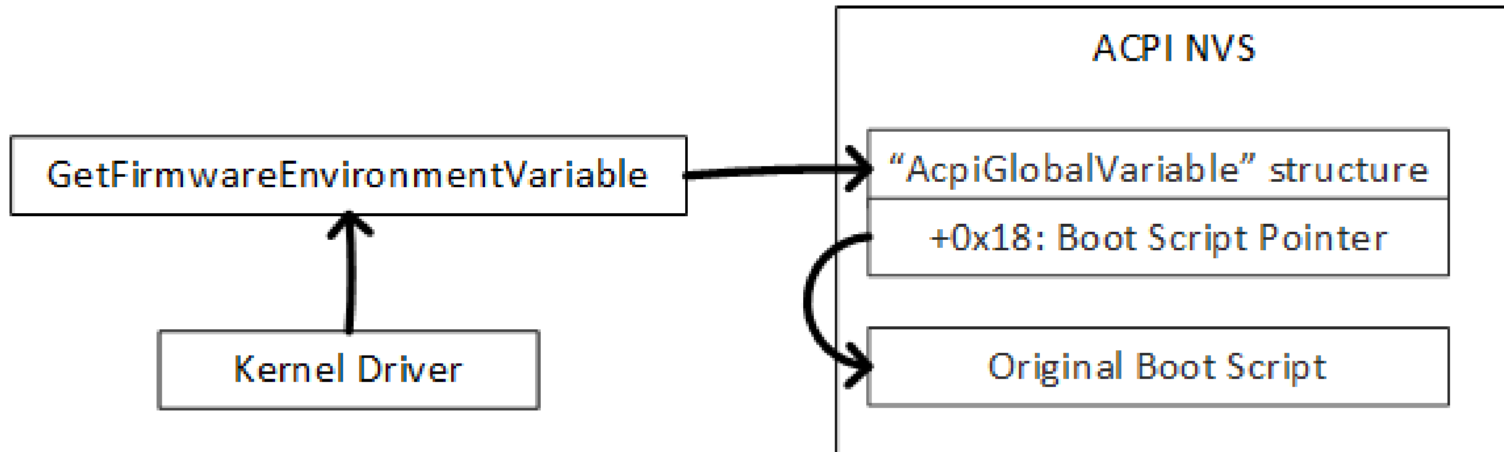
Description

This function adds a dispatch record into a specified boot script table, with which it can run the arbitrary code that is specified. This script can be used to initialize the processor. When the script is executed, the script incurs jumping to the entry point to execute the arbitrary code. After the execution is returned, it goes on executing the next opcode in the table. If the codes to be dispatched have dependencies on other PPIs or codes, the caller should guarantee that all dependencies are sufficient before dispatching the codes.

Okaaaayyy…arbitrary code eh?

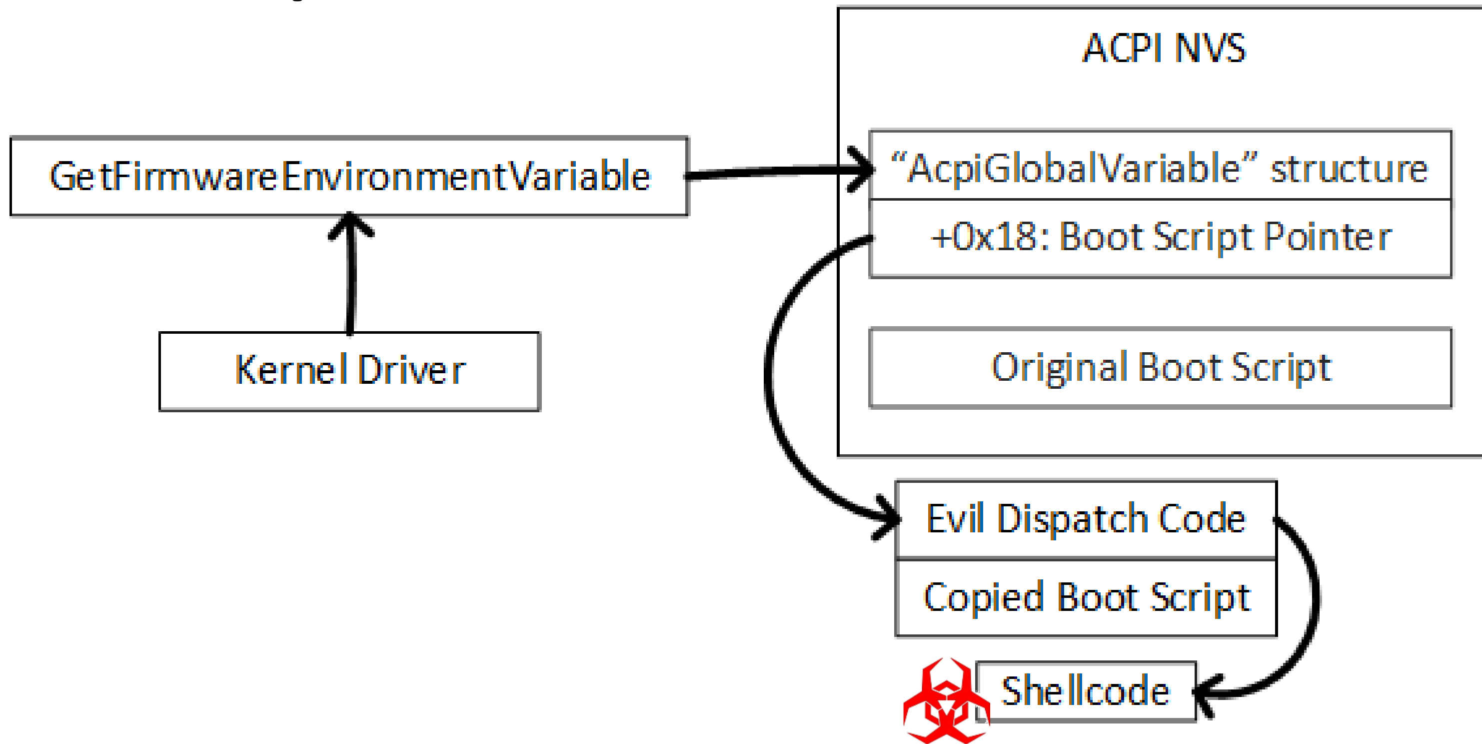
- It means that if we can achieve any of the below then we can force S3 suspend/resume cycle and run arbitrary code in the context of the Boot Script interpreter
  - Alter the content of the Boot Script (insert a custom dispatch opcode)
  - Alter the target of any of existing EFI BOOT SCRIPT DISPATCH OPCODE
  - Alter the data structures used by firmware to locate the Boot Script

Exploitation Visualized 1/2



- Pointer to the boot script can be discovered by reading the contents of the "AcpiGlobalVariable" UEFI non-volatile variable.
  - Contents of "AcpiGlobalVariable" point to a structure
  - At +0x18 is a pointer to the boot script

Exploitation Visualized 2/2



1. Attacker makes a copy of original boot script
2. Attacker inserts an evil dispatch at the top of the copied boot script
3. Attacker overwrites AcpiGlobalVariable boot script pointer with a pointer to his evil boot script

Hardened Boot Script

LockBox for BootScript

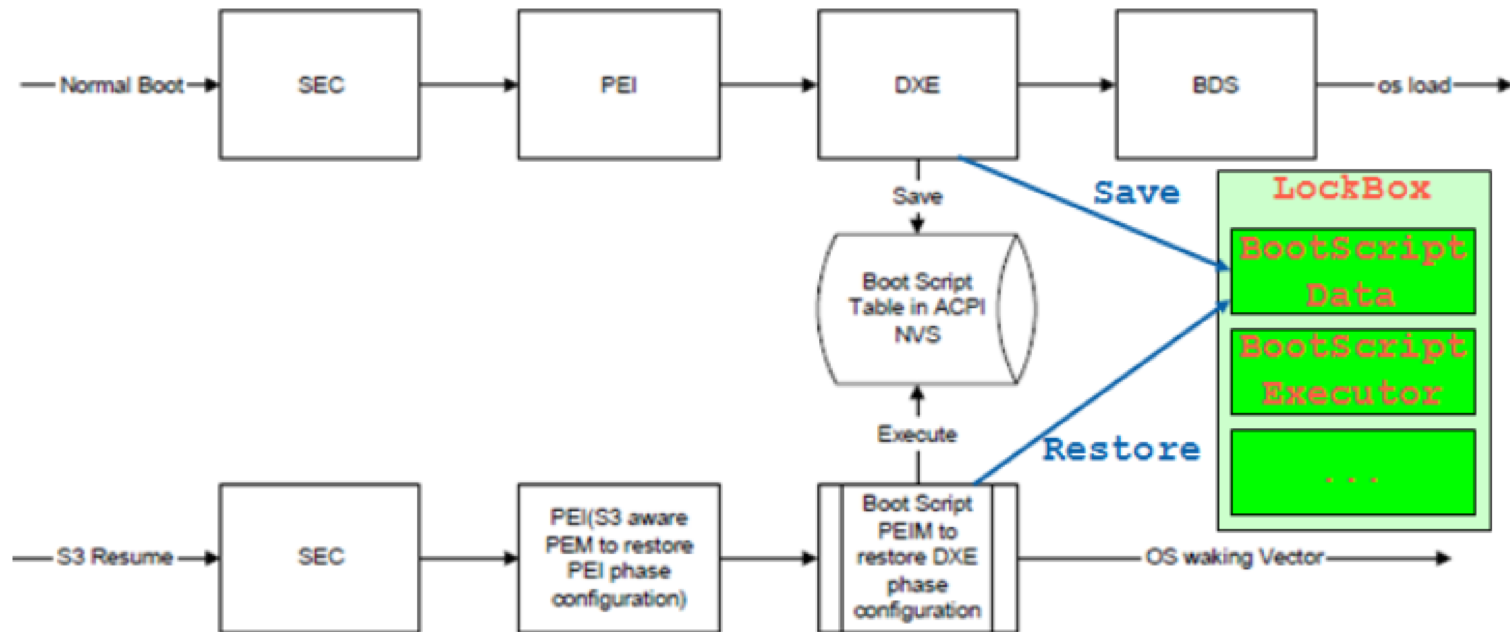
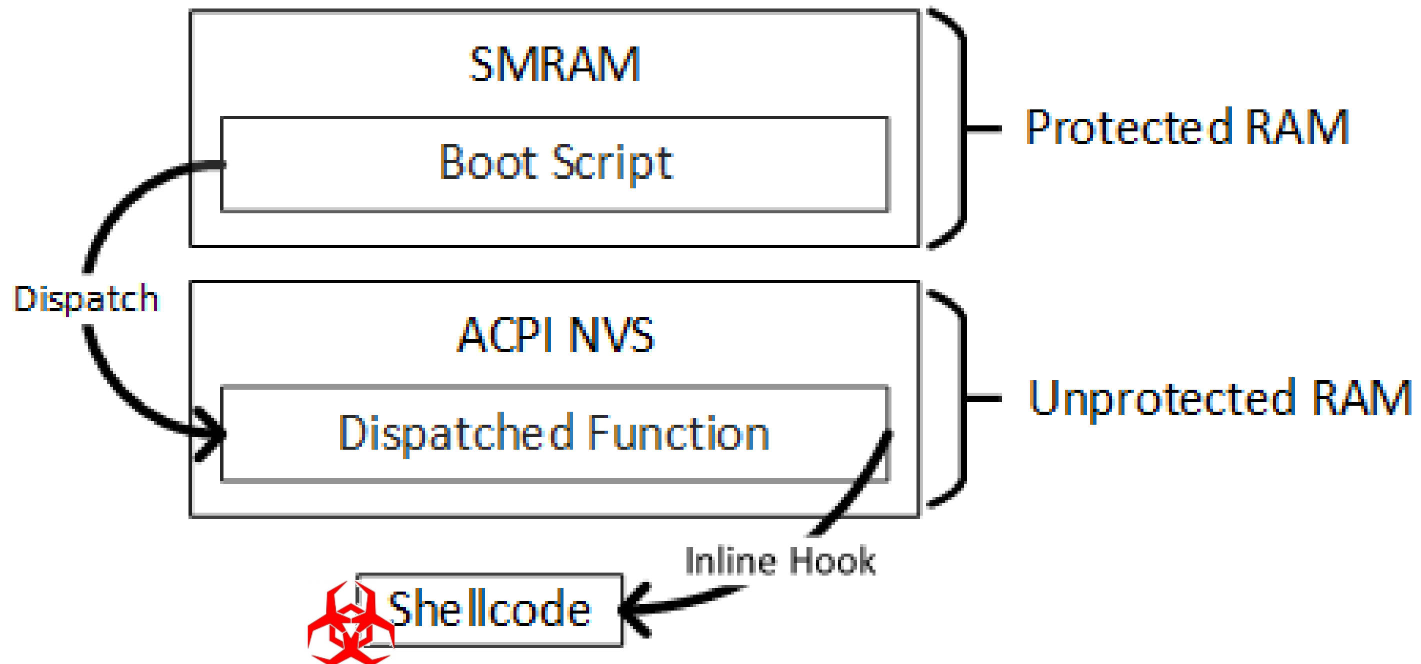


Figure 12 S3 Resume Boot Path with BootScript in LockBox

- All of the available systems we evaluated stored boot script in unprotected ACPI NVS
- However, EDK2[5] protects the contents of the boot script with a “lockbox” which is protected in SMRAM

Picture from [11]

Hardened Boot Script Exploitation

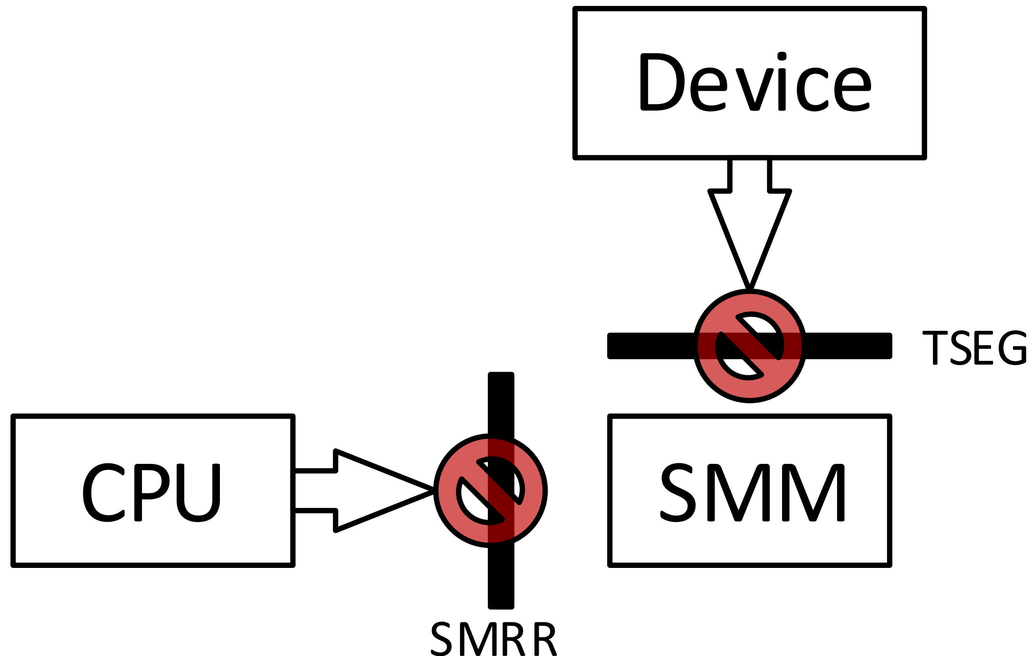


- The only system we identified that used the SMM lockbox to protect the boot script was a UEFI development motherboard[6]
- Its implementation was vulnerable because it dispatched functions in unprotected ACPI NVS
- An attacker could hook these functions to gain arbitrary code execution in the context of the boot script

Boot Script Execution Context

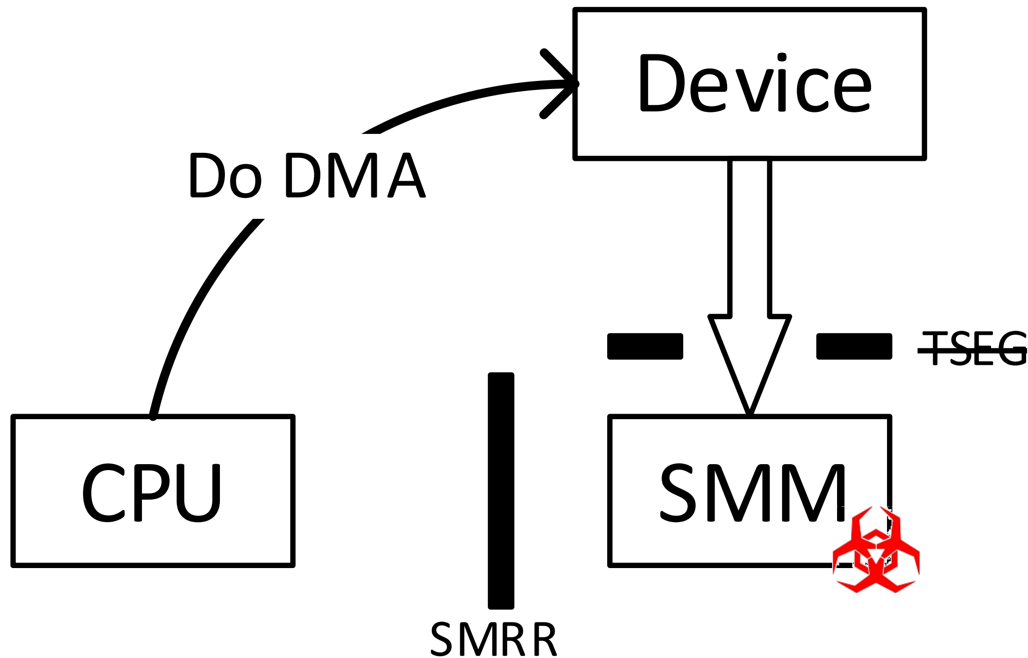
- Platform is largely "unlocked" at this point
- BIOS_CNTL unlocked so BWE can be freely enabled
  - This means we don't necessarily have to break into SMM to attack the firmware
  - But it would still be fun!
- Regarding SMM:
  - SMRRs are set, so no CPU read/write access to SMRAM
  - TSEG is unlocked however
    - So we can disable TSEG by locking it to a value that doesn't actually protect SMRAM

SMM Protection



- SMM is protected from non-SMM CPU access by SMM Range Registers (SMRR)
  - SMRRs are enabled in boot script context
- SMM is protected from DMA by TSEG
  - TSEG is unlocked in boot script context!

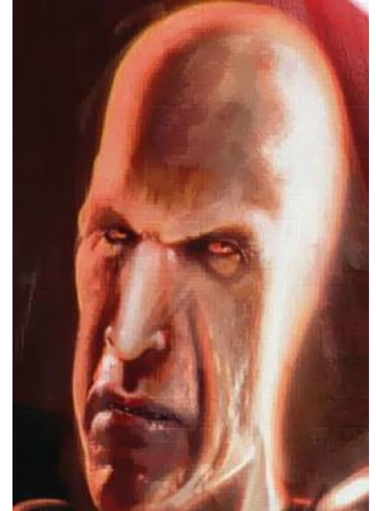
SMM Protection Disabled



- We were able to disable TSEG by locking it to a value above SMRAM (FF000000)
- DMA code is very device specific, so we wait until context has returned to the OS and then use a hard disk driver to initiate the DMA transaction on our behalf[7]

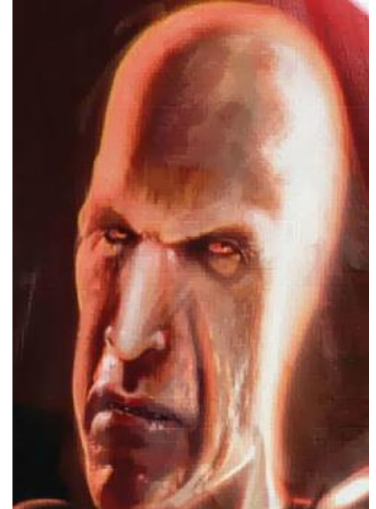
Venamis Summary

- Assigned CERT VU #976132
- All of the UEFI systems we surveyed were vulnerable
- Allows a kernel level attacker to:
  - Bypass BIOS_CNTL flash protections
  - Escalate to SMM
- Relatively easy to exploit, just requires some reversing of the boot script format

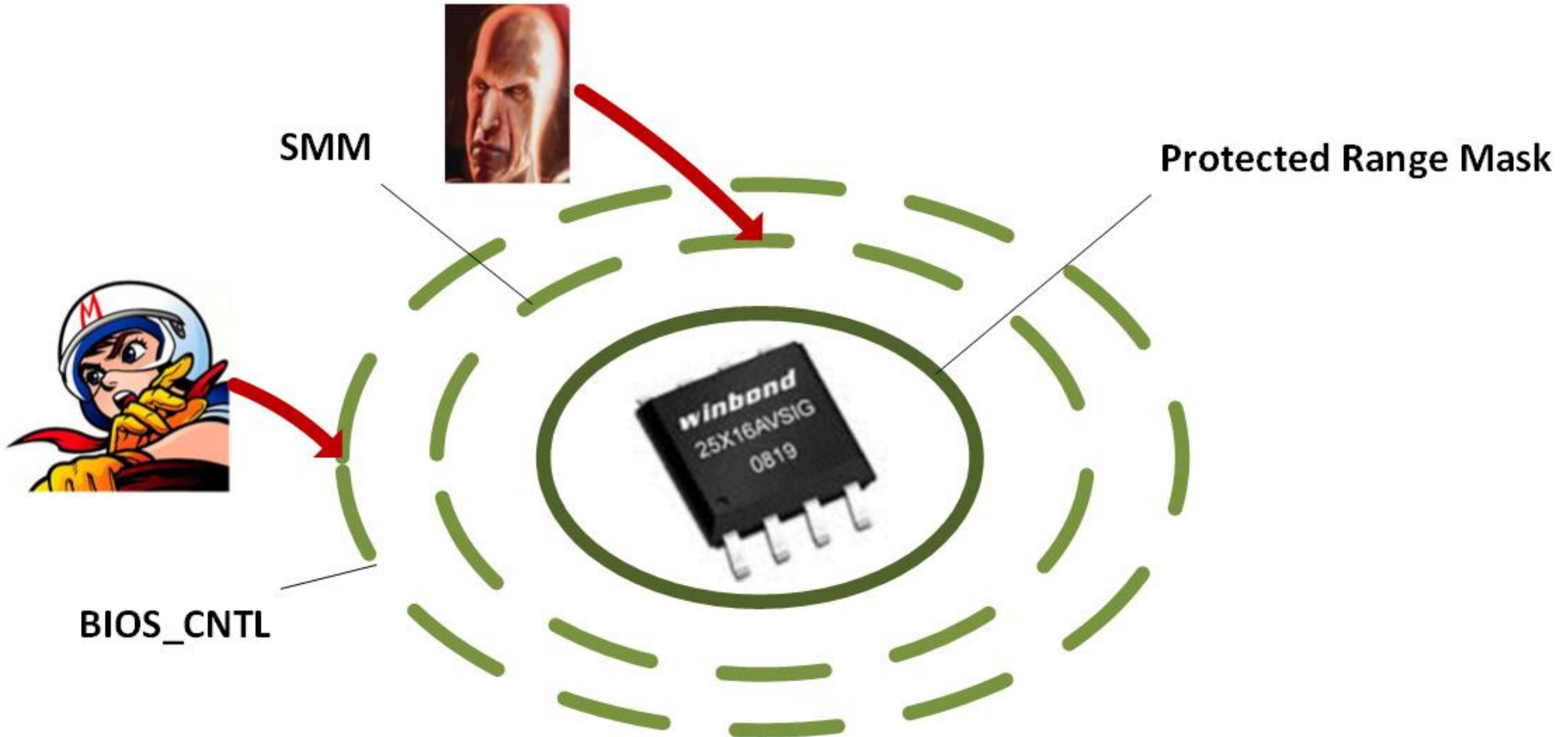


Co-discovery

- CERT VU #976132 was co-reported by the Intel Advanced Threat Research Team!
  - Yuriy Bulygin, Mikhail Gorobets, Andrew Furtak, Oleksander Bazhaniuk, Alexander Matrosov

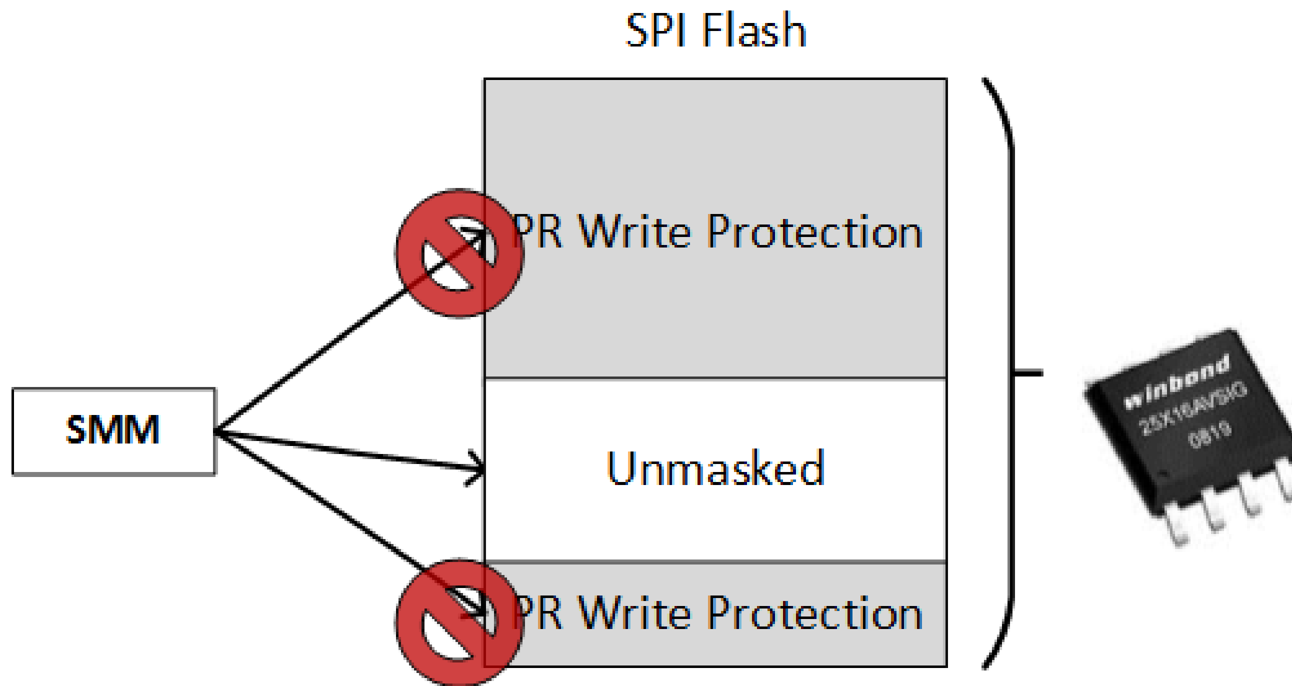


Where we're at



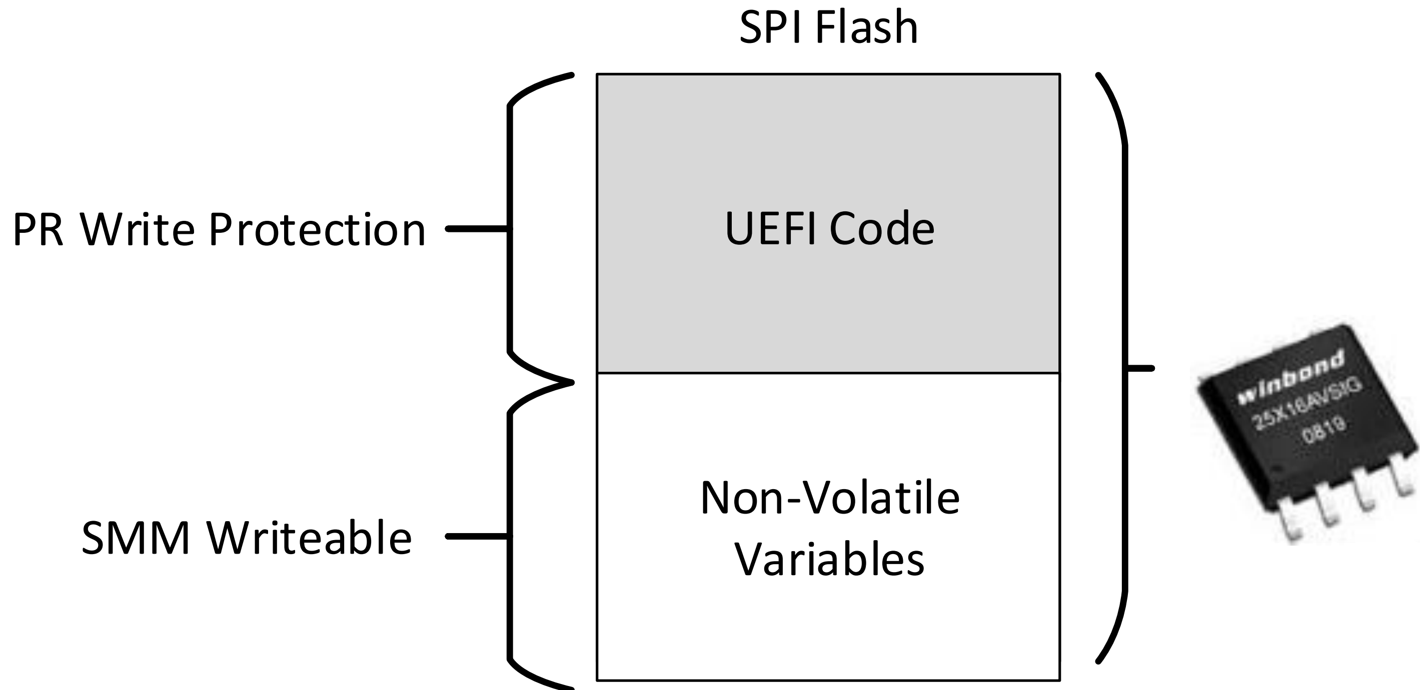
- One last hurdle remains:
  - Protected Range register masks

Protected Range Masks



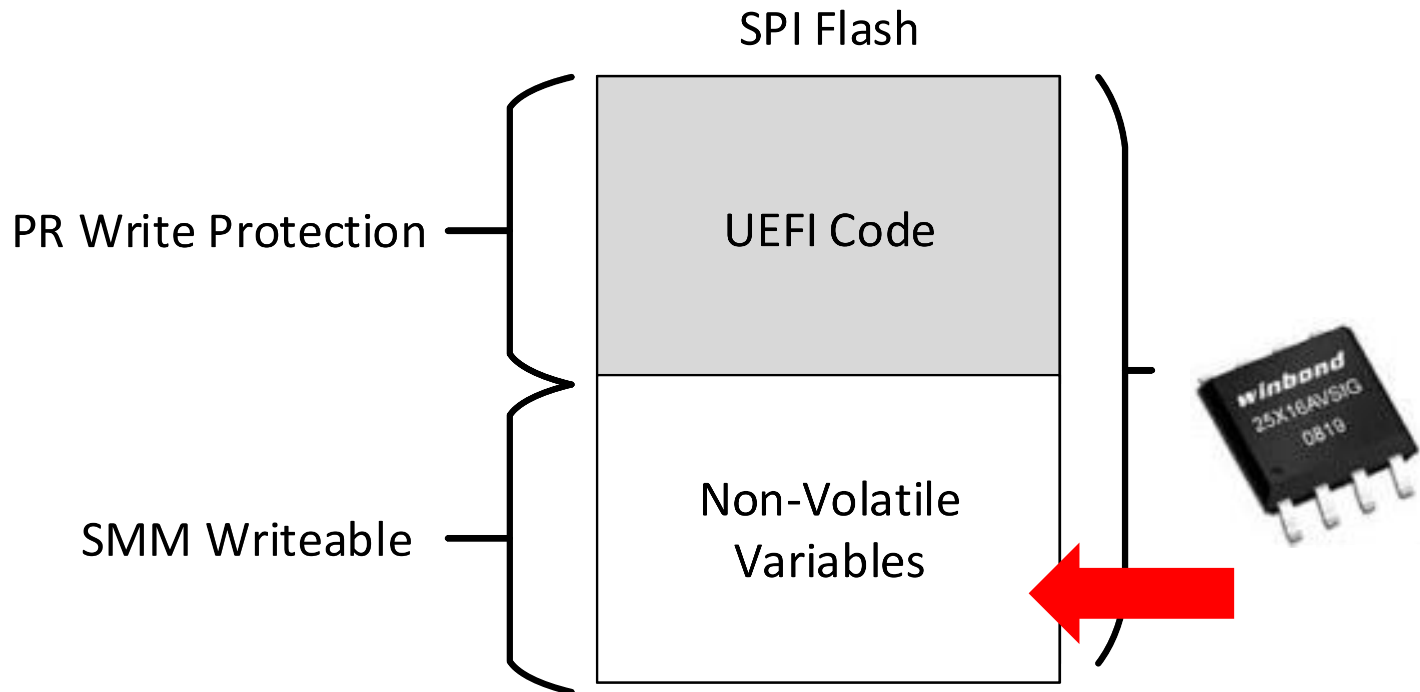
- Protected Range registers allow you to define regions on the flash that are non-writable
- Even SMM is unable to make writes to these regions

Flash Protections and UEFI



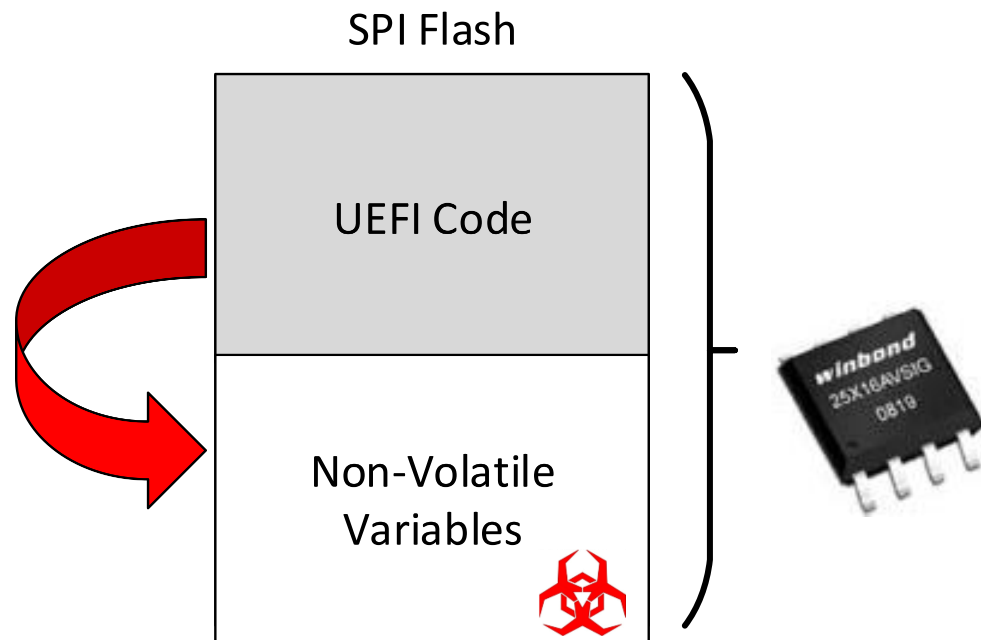
- The UEFI Code region may (or may not) be write protected by PR masks
- Region of the flash where UEFI Non-Volatile Variables are stored must be left writable at runtime, because the variables may be updated by the operating system

Flash Protections and UEFI



- Can we find something in the Non-Volatile Variable region that will allow us to corrupt the rest of the firmware?

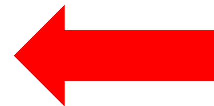
Idea #1



- Force a memory corruption vulnerability in the UEFI code by corrupting the contents of the Non-Volatile variable region
- If this vulnerability occurs before PR masks are set during boot, we win

```
Status = FindVariable (
           AUTHVAR_KEYDB_NAME,
           &gEfiAuthenticatedVariableGuid,
           &Variable,
           &mVariableModuleGlobal->VariableGlobal,
           FALSE
           );

if (Variable.CurrPtr == NULL) {
  ...
} else {
  //
  // Load database in global variable for cache.
  //
  DataSize  = DataSizeOfVariable (Variable.CurrPtr);
  Data      = GetVariableDataPtr (Variable.CurrPtr);
  ASSERT ((DataSize != 0) && (Data != NULL));
  CopyMem (mPubKeyStore, (UINT8 *) Data, DataSize);
```



- Normally we can’t control DataSize because it’s part of an authenticated variable
- SMM is able to arbitrarily modify this data however

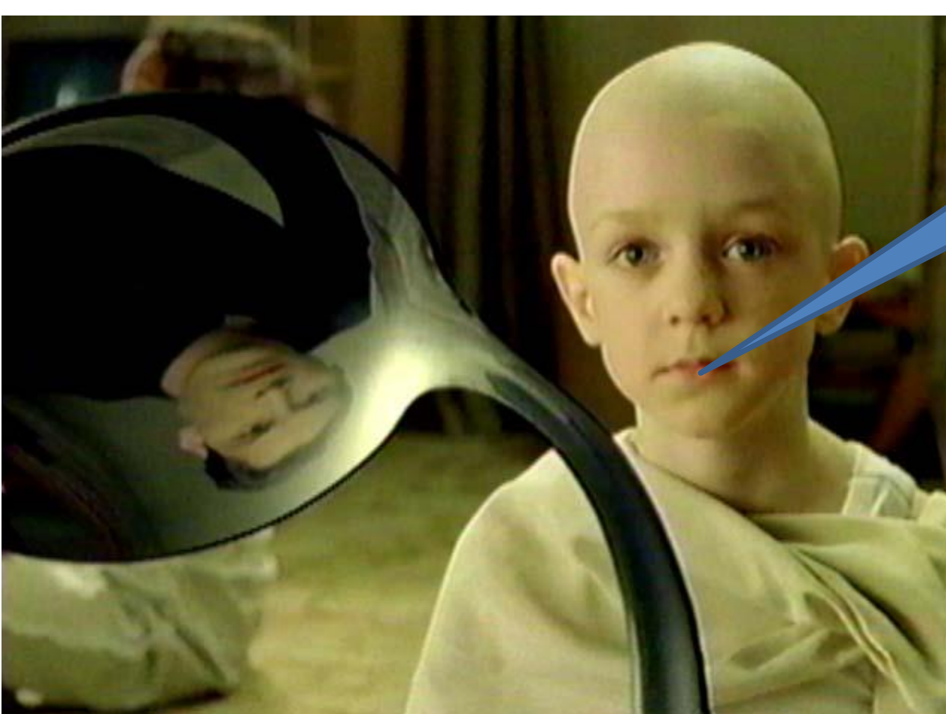
AutenticatedVariableServiceInitialize() in UDK2014/SecurityPkg/VariableAuthenticated/RuntimeDxe/AuthService.c [8]

```
Variable = (VARIABLE_HEADER *) (VariableStoreHeader + 1);
...
ValidBufferSize = sizeof (VARIABLE_STORE_HEADER);
while (IsValidVariableHeader (Variable)) {
  NextVariable = GetNextVariablePtr (Variable);
  ...
  if (Variable->State == VAR_ADDED) {
    VariableSize = (UINTN) NextVariable - (UINTN) Variable;
    EfiCopyMem (CurrPtr, (UINT8 *) Variable, VariableSize);
    ValidBufferSize += VariableSize;
    CurrPtr += VariableSize;
```

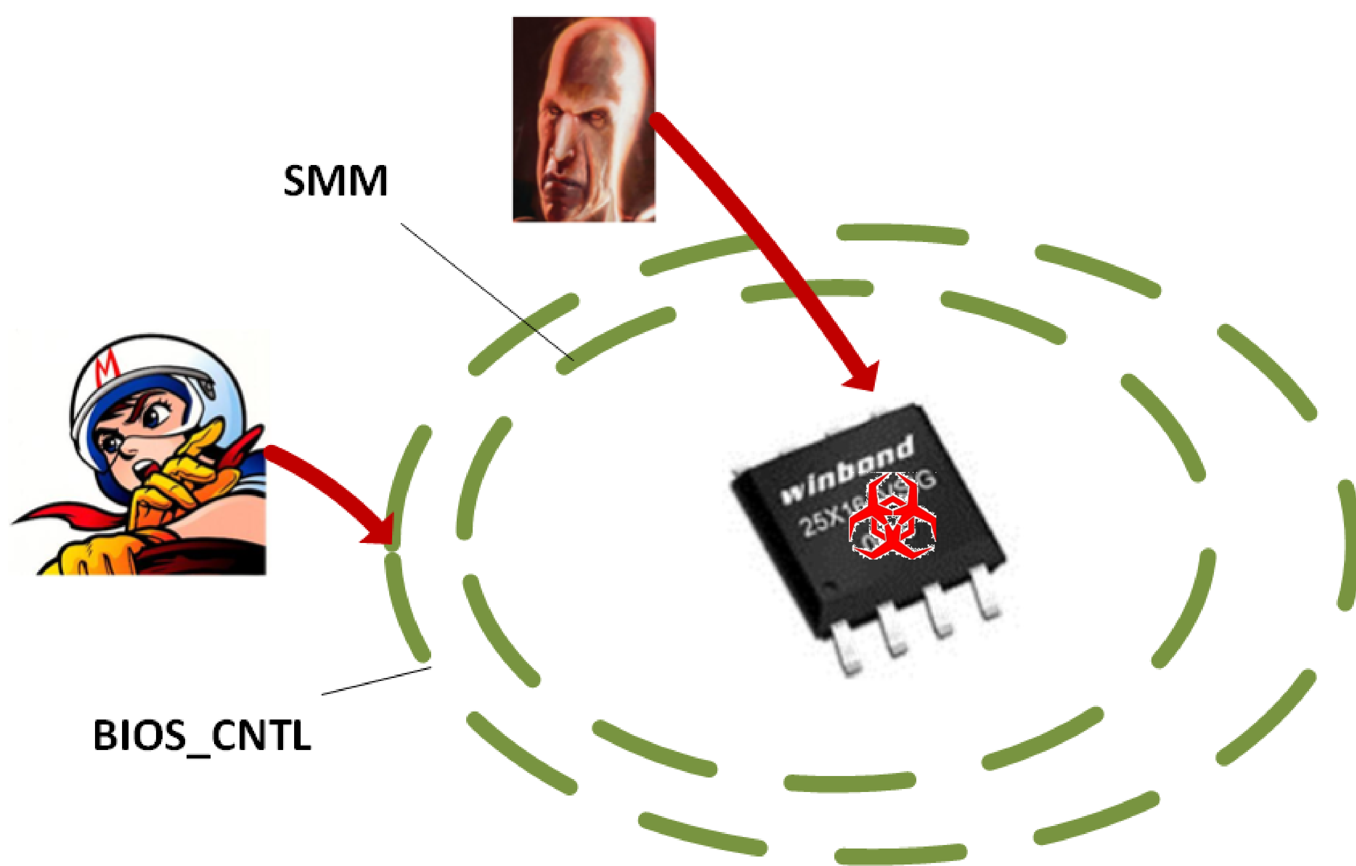
- An SMM attacker can control the metadata associated with the variables, and hence control what is returned by GetNextVariablePtr
- This can lead to buffer overflow during the EfiCopyMem

Reclaim() in EDK/Sample/Universal/Variable/RuntimeDxe/FS/FSVariable.c [9]

- Vulnerabilities assigned CERT VU#533140
- May allow bypassing of PR masks if they can be triggered early in the boot up process
  - Implementation dependent
- But wait! You said we wouldn’t have to do complicated, difficult to reproduce, memory corruption vulnerabilities to hack our BIOS’s!



- Upon further investigating the UEFI code[8], it was determined that authenticated variable contents are used to verify incoming firmware updates
  - So SMM can reflash the firmware by leveraging the normal firmware update path
- Private discussions with a UEFI developer confirmed that this is the case, and that SMM is in the trusted code base for UEFI in general
  - A new hardware feature[10] will address this in the future



- It turns out that many systems do not even make use of PR masks[2]
- Of all the system's we surveyed, only HP made use of PR masks
  - And they had incomplete coverage over the code region of the flash chip
- So on most UEFI systems, if you can get into SMM, UEFI will also fall

Summary

- An important component of firmware flash protections on Intel chipsets is subject to a race condition

  - Patched on newer systems, but only if the OEM makes use of a new feature, which many don't
  - Consequence is DoS or a malicious reflash of the firmware
  - Easily exploited with 2 kernel drivers
- The UEFI boot script can be maliciously modified to break into SMM
  - Every UEFI system we surveyed was vulnerable
  - Easy to exploit with a kernel driver and some reverse engineering of the boot script format

- An attacker who escalates to SMM can likely reflash your firmware
  - A new hardware feature will help address this in the future[10]

Acknowledgements

- Thanks to Intel PSIRT, UEFI Security Response Team, and CERT for helping coordinate these vulnerabilities!

Bibliography 1

[1] Attacking Intel BIOS – Alexander Tereshkin & Rafal Wojtczuk – Jul. 2009
http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf

[2] Defeating Signed BIOS Enforcement – Kallenberg et al., Sept. 2013 –
http://www.mitre.org/publications/technical-papers/defeating-signed-bios-enforcement

[3] Extreme Privilege Escalation on Windows 8/UEFI Systems – Kallenberg et al., August 2014

http://www.mitre.org/publications/technical-papers/extreme-privilege-escalation-on-windows-8uefi-systems

[4] Setup for Failure: Defeating UEFI – Kallenberg et al., Apr 2014
http://syscan.org/index.php/download/get/6e597f6067493dd581eed737146f3afb/SyScan2014_CoreyKallenberg_SetupforFailureDefeatingSecureBoot.zip

[5] EDK2 UEFI Reference implementation

http://tianocore.sourceforge.net/wiki/EDK2

[6] DQ57TML UEFI Development Kit Firmware

http://uefidk.com/develop/development-kit

Bibliography 2

[7] Poacher Turned Gamekeeper: Lessons Learned from 8 years of breaking hypervisors" – Wojtczuk, Aug 2014 https://www.blackhat.com/docs/us-14/materials/us-14-Wojtczuk-Poacher-Turned-Gamekeeper-Lessons_Learned-From-Eight-Years-Of-Breaking-Hypervisors-wp.pdf

[8] UDK2014

http://tianocore.sourceforge.net/wiki/UDK2014

[9] EFI Development Kit

http://tianocore.sourceforge.net/wiki/EDK

[10] Intel Platform Firmware Armoring Technology

http://www.google.com/patents/WO2012039971A2?cl=en

[11] A Tour Beyond BIOS: Implementing S3 Resume with EDK2 – Jiewen Yao and Vincent Zimmer, October 2014

[12] Intel Platform Innovation Framework for EFI S3 Resume Boot Path Specification

http://www.intel.com/content/dam/doc/reference-guide/efi-s3-resume-boot-path-specification.pdf