The Whole is Greater…

Sugumar Govindarajan
Security Architect, Intel Server BIOS

John Loucaides
Security Researcher, Intel Product Security Incident Response Team

Contributions from **Aaron Frinzell**, Intel Server BIOS

Legal information

Intel technologies, features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

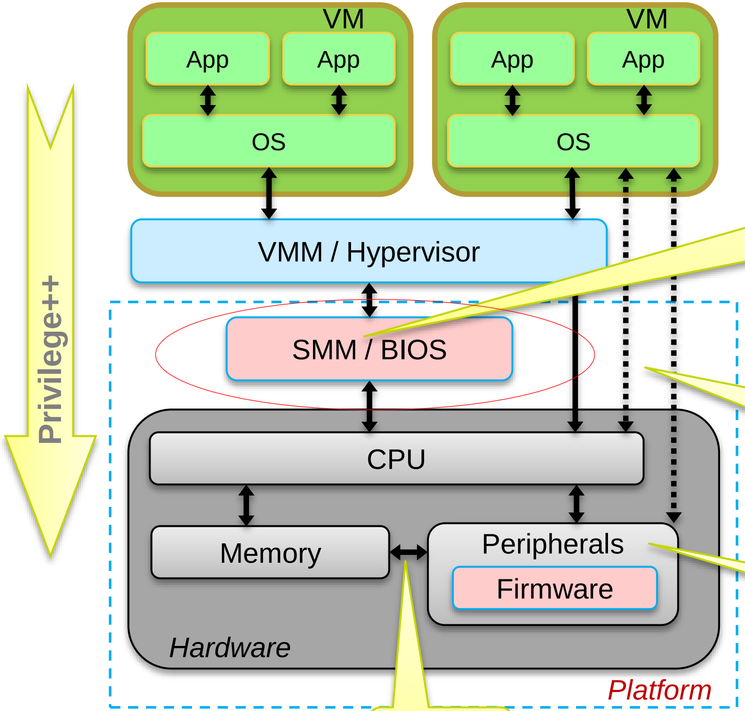
Intel, the Intel logo, Intel vPro, Look Inside., the Look Inside. logo, Intel Xeon Phi, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Firmware

Let’s set the stage

Where are we (system firmware)?



BIOS & OS/VMM share access, but not trust

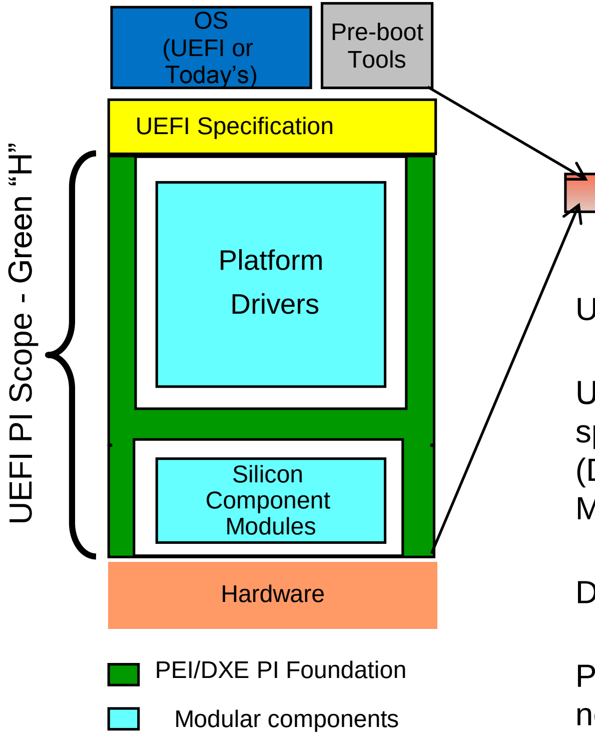
UEFI + PI SMM

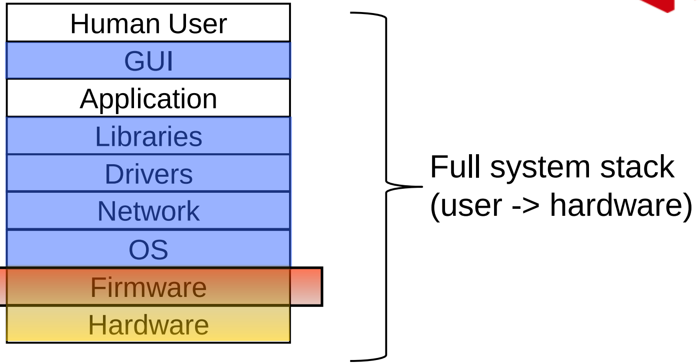
Hypervisor can grant VM direct hardware access

A specific Peripheral may have its own processor, and its own firmware, which is undetectable by host CPU/OS.

DMA

What is UEFI?





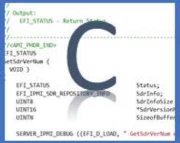
UEFI 2.4 specifies how firmware boots OS loader

UEFI's Platform Initialization (PI) 1.3 Architecture specifies how Driver Execution Environment (DXE) Drivers and Pre-EFI Initialization (PEI) Modules (PEIMs) initialize SI and the platform

DXE is preferred UEFI Implementation

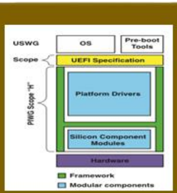
PEIMs, UEFI and DXE drivers implement networking, Update, other security features

What’s in UEFI



```
/ Output:
/   EFI_STATUS - Return Status
/
/--------------------------------------------
/<AMI_PHDR_END>
EFI_STATUS
GetSdrVerNum (
  VOID )

   EFI_STATUS                              Status;
   EFI_IPMI_SDR_REPOSITORY_INFO            SdrInfo;
   UINT8                                   SdrInfoSize
   UINT16                                  *SdrVersion
   UINTN                                   SizeofBuffer

   SERVER_IPMI_DEBUG ((EFI_D_LOAD, " GetSdrVerNum
```

Mostly written in C. High code re-use.

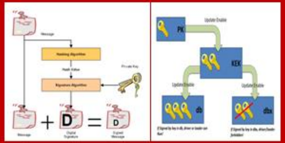


Emphasis on Specifications. Standards compliance.

Better platform scaling. For e.g. removes shadow ROM limits.

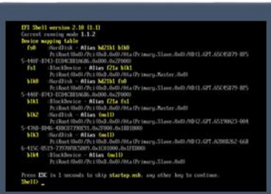
Storage. GPT removes 2.2 TB MBR restriction.

CPU Architecture independent. Platform design flexibility.



Secure boot solves “trust” related system integration challenges.

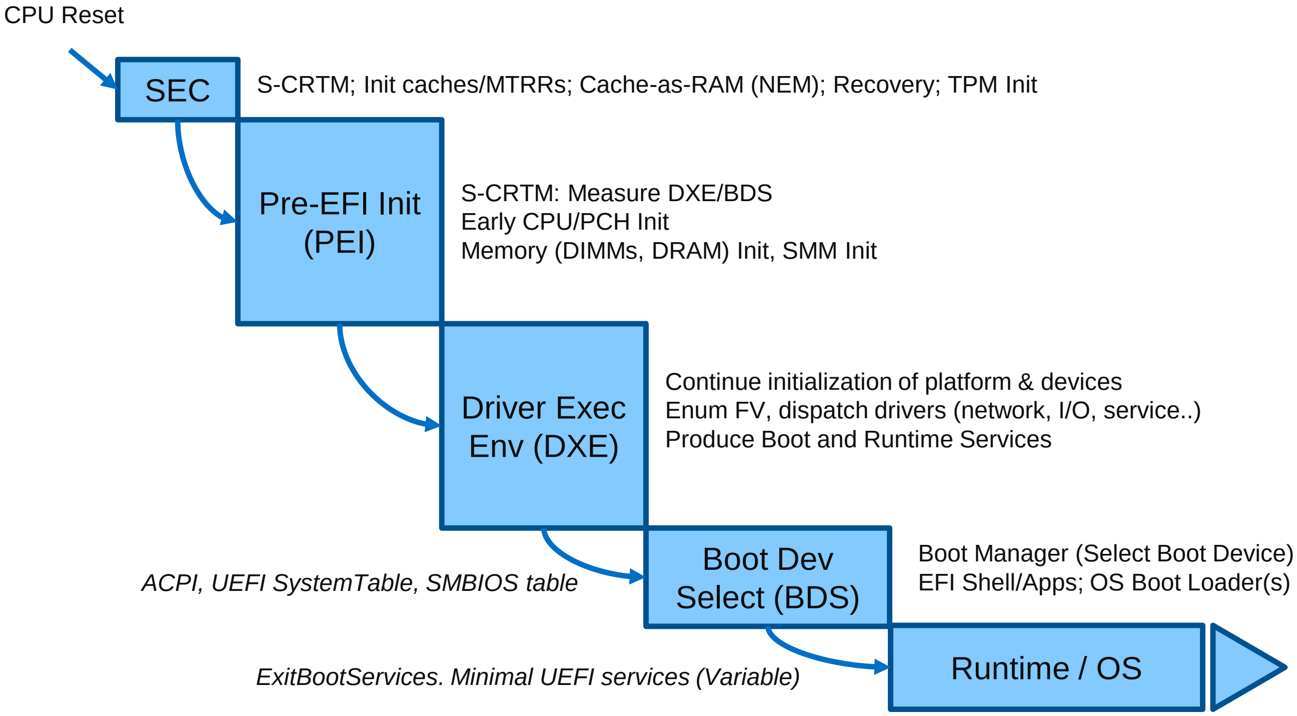
Pre-boot Networking. Ipv4, Ipv6, PXE, VLAN, iSCSI etc.



```
EFI Shell version 2.10 [1.1]
Current running mode 1.1.2
Device mapping table
  fs0    :HardDisk - Alias hd21b1 blk0
          PciRoot(0x0)/Pci(0x0,0x0)/Ata(Primary,Slave,0x0)/HD(1,GPT,[illegible]
[illegible]
  fs1    :BlockDevice - Alias f21a blk1
          PciRoot(0x0)/Pci(0x0,0x0)/Ata(Primary,Master,0x0)
  blk0   :HardDisk - Alias hd21b1 fs0
          PciRoot(0x0)/Pci(0x0,0x0)/Ata(Primary,Slave,0x0)/HD(1,GPT,[illegible]
[illegible]
  blk1   :BlockDevice - Alias f21a fs1
          PciRoot(0x0)/Pci(0x0,0x0)/Ata(Primary,Master,0x0)
  blk2   :HardDisk - Alias (null)
          PciRoot(0x0)/Pci(0x0,0x0)/Ata(Primary,Slave,0x0)/HD(2,GPT,[illegible]
[illegible]
  blk3   :HardDisk - Alias (null)
          PciRoot(0x0)/Pci(0x0,0x0)/Ata(Primary,Slave,0x0)/HD(3,GPT,[illegible]
[illegible]
  blk4   :BlockDevice - Alias (null)
          PciRoot(0x0)/Pci(0x0,0x0)/Ata(Primary,Slave,0x0)

Press ESC in 1 seconds to skip startup.nsh, any other key to continue.
Shell> _
```

UEFI shell improves pre-boot testing & diagnostics experience.

UEFI [Compliant] Firmware



CPU Reset

SEC

S-CRTM; Init caches/MTRRs; Cache-as-RAM (NEM); Recovery; TPM Init

Pre-EFI Init (PEI)

S-CRTM: Measure DXE/BDS
Early CPU/PCH Init
Memory (DIMMs, DRAM) Init, SMM Init

Driver Exec Env (DXE)

Continue initialization of platform & devices
Enum FV, dispatch drivers (network, I/O, service..)
Produce Boot and Runtime Services

ACPI, UEFI SystemTable, SMBIOS table

Boot Dev Select (BDS)

Boot Manager (Select Boot Device)
EFI Shell/Apps; OS Boot Loader(s)

ExitBootServices. Minimal UEFI services (Variable)

Runtime / OS

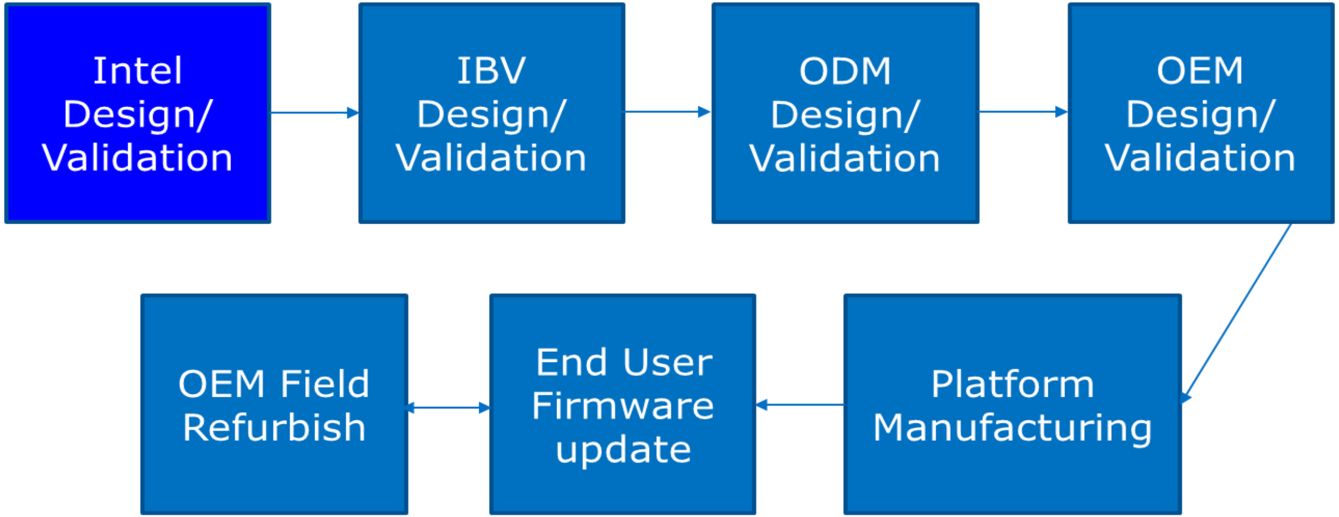
The Challenge

No one expects the **BIOS Inquisition**!!

The *hole* is also greater… than expected…

- **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”, insecure use of UEFI variables, etc.)
- Many issues **affect multiple vendors** at once (S3 boot script, BIOS write protections, UEFI variables, SMI call-outs, SMI input pointers)
- Issues may need to be checked separately in every product
- And updating firmware is not easy

Start with a complicated enabling process…



Intel Design/ Validation

IBV Design/ Validation

ODM Design/ Validation

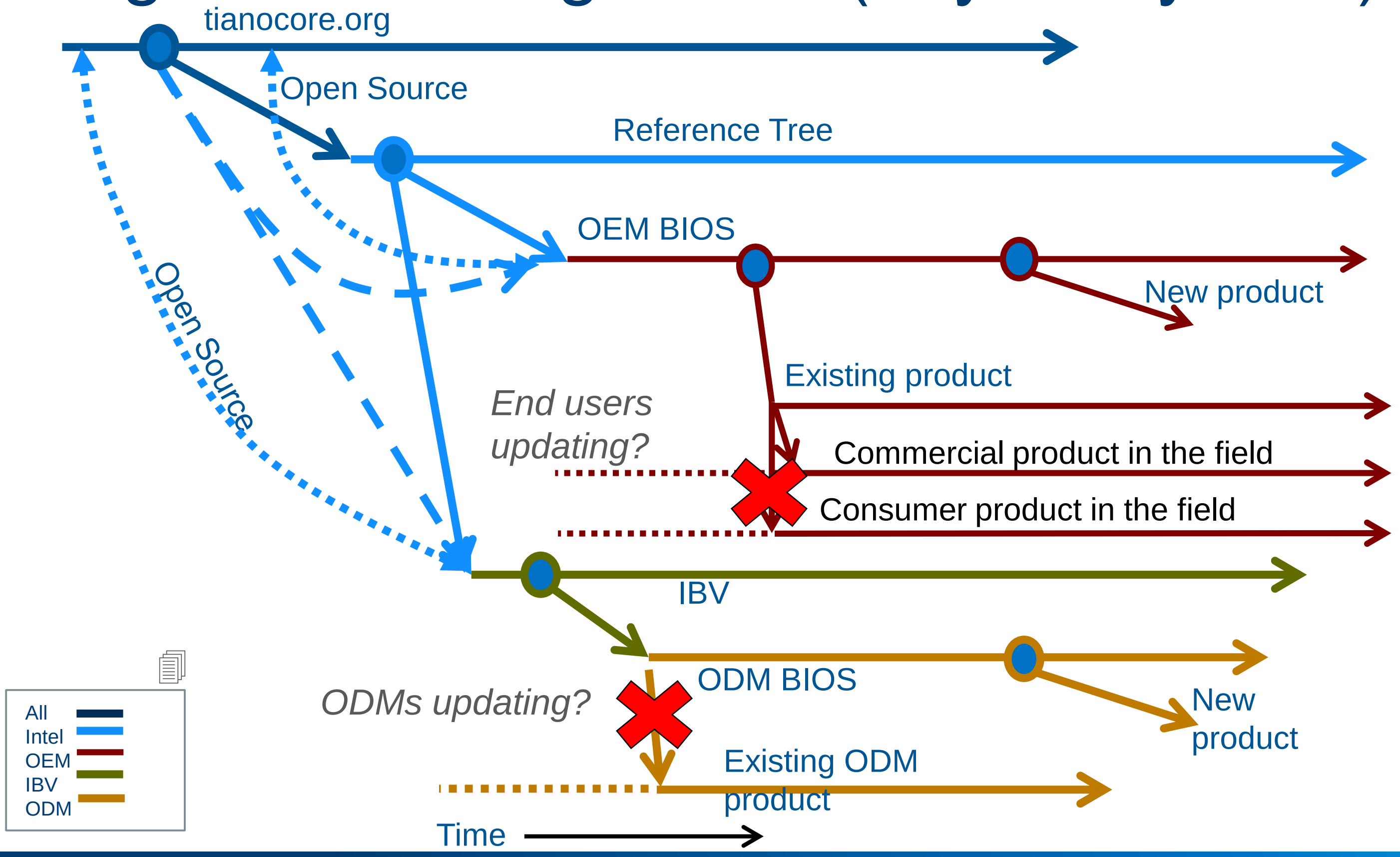
OEM Design/ Validation

Platform Manufacturing

End User Firmware update

OEM Field Refurbish

“The Long and Winding Road” (to your system)



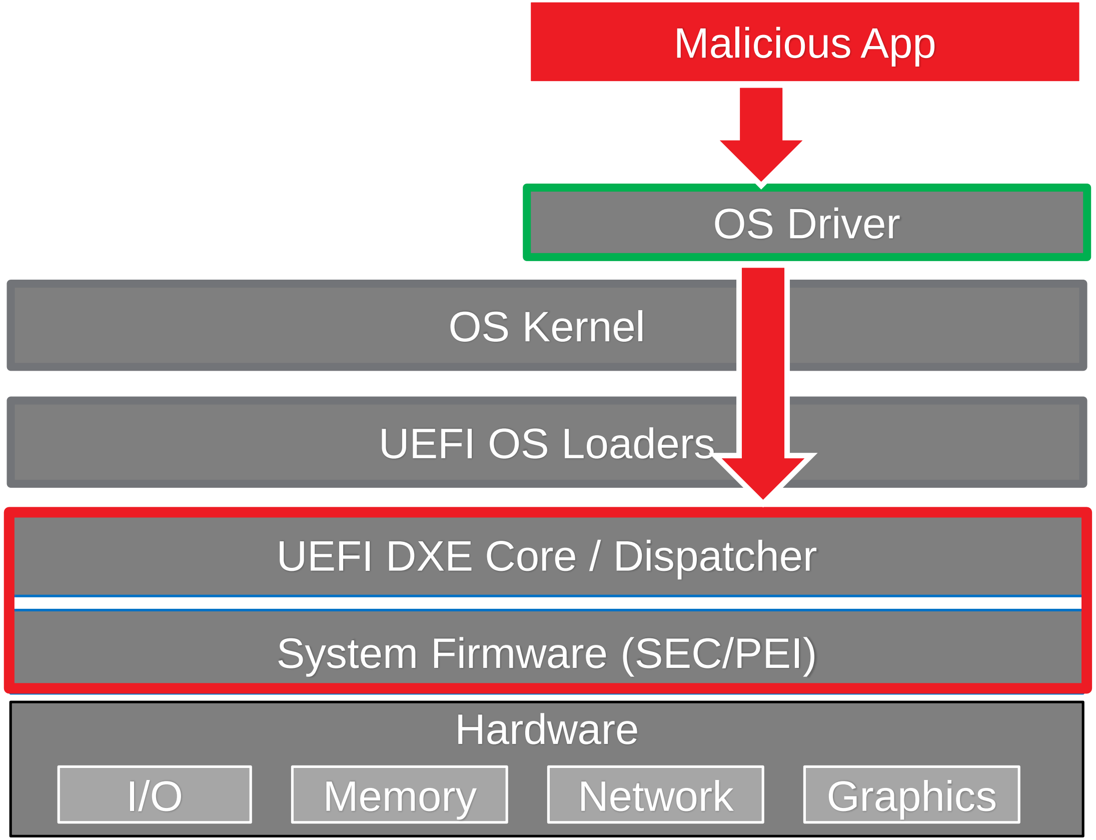
Some (quick) Examples

1. S3 Boot Script (Sleep/Resume) Issues

2. SMI Handler Call-Outs

3. Bad Pointer Input SMI Vulnerabilities

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 64bit Windows (co-discovered with other researchers)

The Sum of the Parts…

A quick summary of many ongoing efforts to improve system firmware security…

Parts to Consider…

- Finding Issues
  - Vulnerability Research
  - Security Development Lifecycle activities
  - Testing (CHIPSEC and more…)
- Hardening
  - Development processes and practices
  - Threat Model and Mitigations
- Community & Ecosystem Engagement
  - UEFI Security Response Team
  - Intel and Tianocore Advisories

Finding Issues

Intel is looking for (and finding) issues…

2013

- Secure Boot Bypass Research (BH 2013)

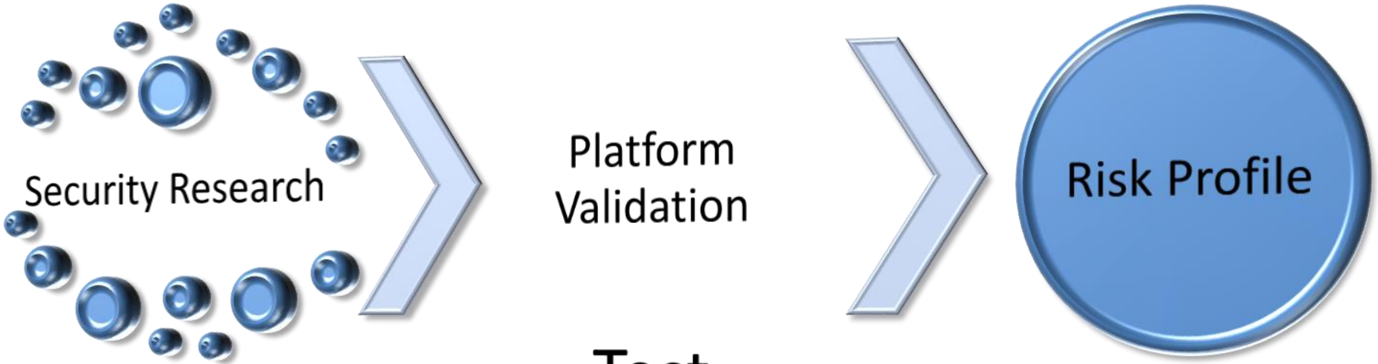
2014

- Tianocore issues (see http://tianocore.org/security)
- Multiple Secure Boot Bypasses (independent discovery by other researchers, co-presented at CanSecWest 2014)
- Summary of BIOS Attacks (DEFCON 22, Ekoparty 10, Ruxcon 2014)
- S3 Boot Script Vulnerabilities (independent discovery by external researchers, presented by Intel at Recon 2015)

2015

- SMM Bad Pointer Input Vulnerabilities (CanSecWest 2015)
- LTE Modem Security (DEFCON 23)
- Attacking VMMs through Firmware (BH 2015)

CHIPSEC



Security Research

Platform Validation

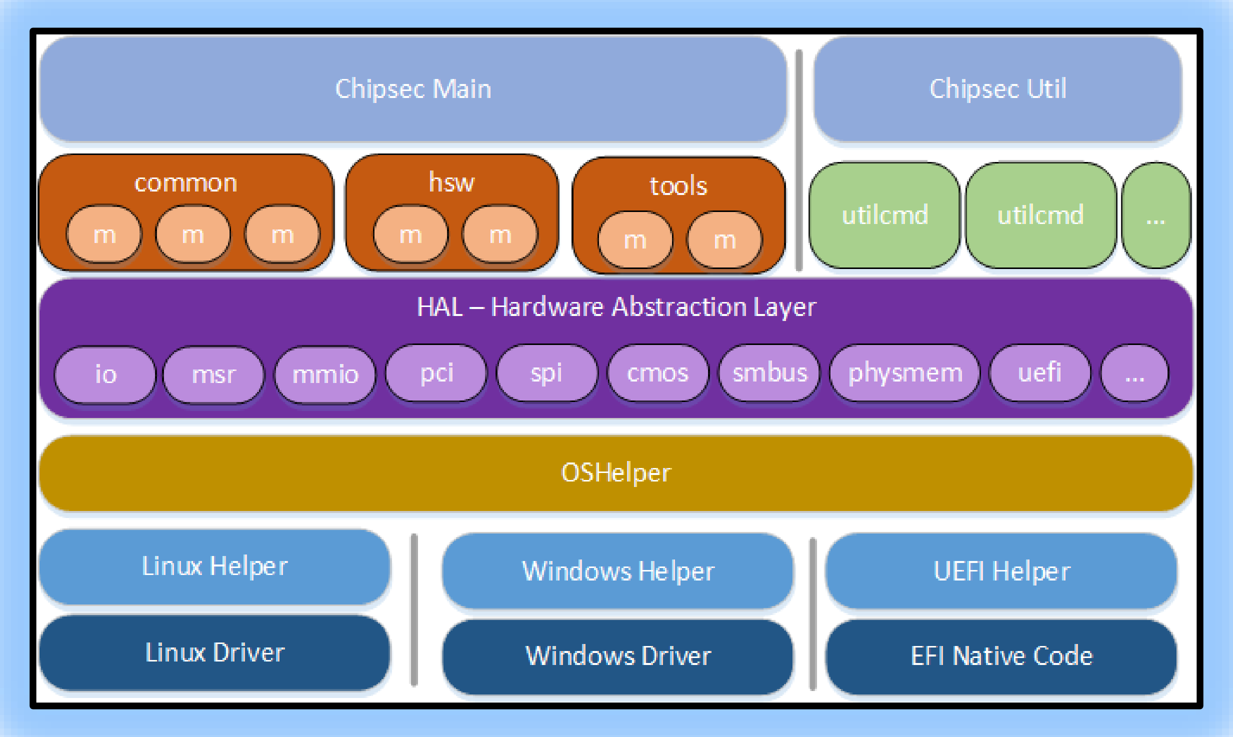
Risk Profile

New Attacks

Test Modules for OEMs/IBVs

End-user Risk

Raising the Bar for Platform Security



*Other names and brands may be claimed as the property of others.

Use CHIPSEC for access HW resources

```
chipsec_util msr 0x200
chipsec_util mem 0x0 0x41E 0x20
chipsec_util pci enumerate
chipsec_util pci 0x0 0x1F 0x0 0xDC byte
chipsec_util io 0x61 byte
chipsec_util mmcfg 0 0x1F 0 0xDC 1 0x1
chipsec_util mmio list
chipsec_util cmos dump
chipsec_util ucode id
chipsec_util smi 0x01 0xFF
chipsec_util idt 0
chipsec_util cpuid 1
chipsec_util spi info
chipsec_util spi read 0x700000 0x100000 bios.bin
chipsec_util decode spi.bin
chipsec_util uefi var-list
chipsec_util spd dump
chipsec_util acpi list
```

Known Threats and CHIPSEC modules

Chipsec Module	Publication
Modules/common/ bios_ts.py	"BIOS Boot Hijacking and VMware Vulnerabilities Digging" - Sun Bing
Modules/common/ bios_kbrd_buffer.py	DEFCON 16: "Bypassing Pre-boot Authentication Passwords by Instrumenting the BIOS Keyboard Buffer" – Jonathan Brossard
Modules/common/bios_smi.py	"Setup for Failure: Defeating SecureBoot" – Corey Kallenberg, Xeno Kovah, John Butterworth, Sam Cornwell
Modules/remap	Preventing & Detecting Xen Hypervisor Subversions – Joanna Rutkowska & Rafal Wojtczuk
Modules/smm_dma	Programmed I/O accesses: a threat to Virtual Machine Monitors? – Lioc Duflot & Laurent Absil
Modules/common/ bios_wp.py	Black Hat USA 2013 "BIOS Security" – MITRE (Kovah, Butterworth, Kallenberg) NoSuchCon 2013 "BIOS Chronomancy: Fixing the Static Root of Trust for Measurement" – MITRE (Kovah, Butterworth, Kallenberg)
Modules/common/ smm.py	CanSecWest 2006 "Security Issues Related to Pentium System Management Mode" – Duflot
Modules/common/spi_desc.py	Parsing of SPI descriptor access permissions is implemented in "ich_descriptors_tool" which is part of open source flashrom.
Modules/common/ smrr.py	"Attacking SMM Memory via Intel CPU Cache Poisoning" – ITL (Rutkowska, Wojtczuk) "Getting into the SMRAM: SMM Reloaded" – Duflot, Levillain, Morin, Grumelard
Modules/common/ spi_lock.py	FLOCKDN is in flashrom and MITRE's Copernicus
Modules/common/secureboot/keys.py	UEFI 2.4 spec Section 28
Modules/common/secureboot/variables.py	UEFI 2.4 spec Section 28
Modules/common/uefi/access_uefispec.py	UEFI spec – Table 11 "Global Variables"
Modules/tools/smm/smm_ptr.py	"A New Class of Vulnerability in SMI Handlers of BIOS/UEFI Firmware" Intel ATR

Some (quick) Examples

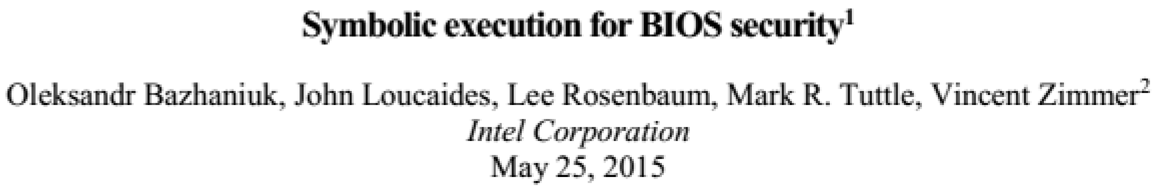
Check BIOS Write Protection

S3 Boot Script (Sleep/Resume) Issues

Bad Pointer Input SMI Vulnerabilities

Symbolic Execution for BIOS Security

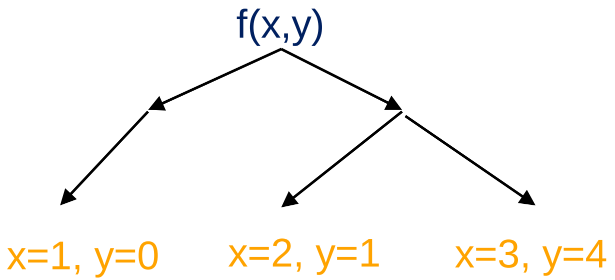
Presented at WOOT'15



Symbolic execution for BIOS security[1]

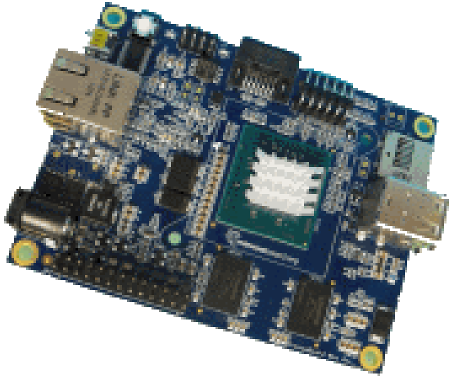
Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R. Tuttle, Vincent Zimmer[2]
Intel Corporation
May 25, 2015

Symbolic execution for code coverage and bug hunting



- Coverage: minimal test cases inducing maximal code coverage
- Bugs: test cases inducing common program vulnerabilities

Use Open Source HW, SW and Tools



HW: Minnow Board MAX Open hardware platform

64-bit Intel® Atom™ SoC E38xx Series

http://firmware.intel.com/projects

SW: Minnow Board MAX UEFI Open Source (EDKII project)

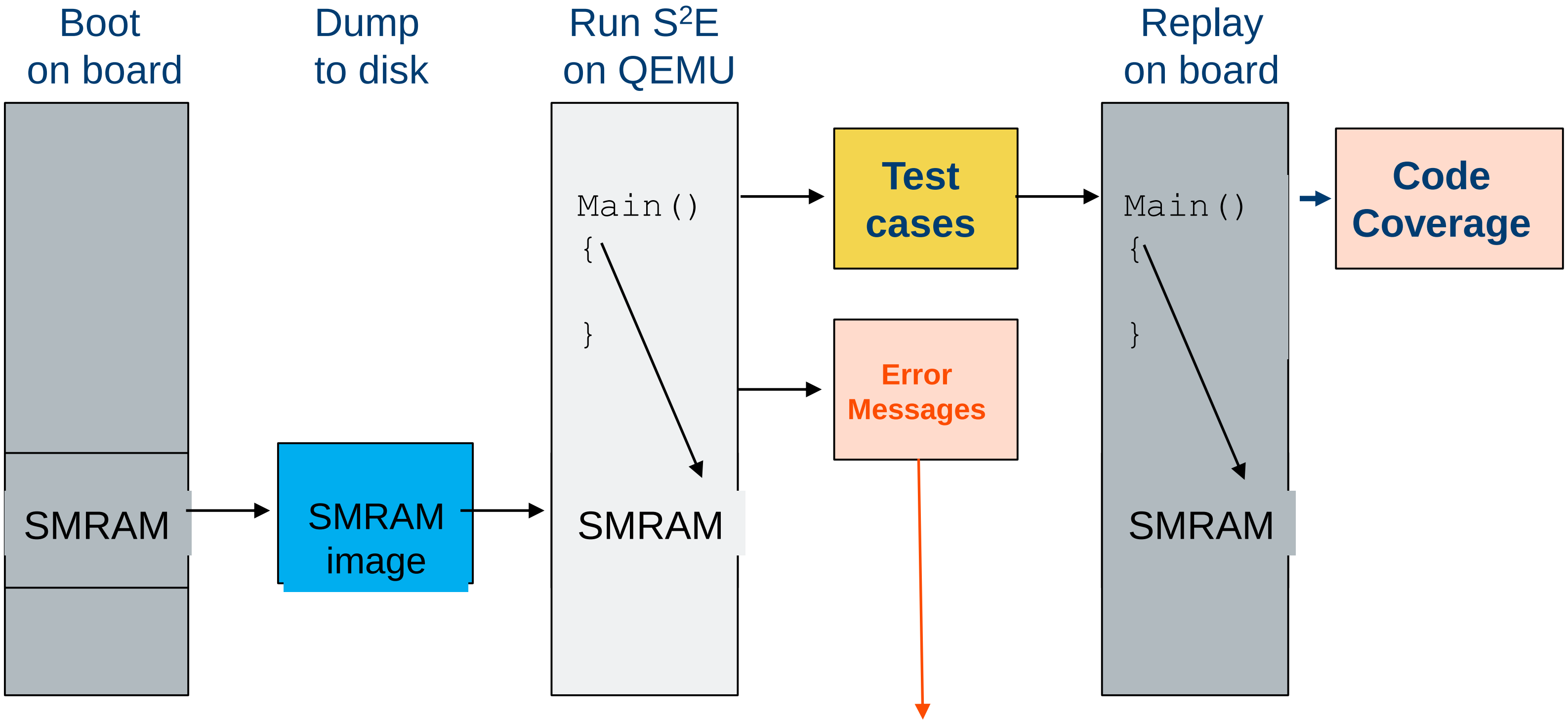
http://tianocore.sourceforge.net/wiki/EDK2

Builds using Microsoft Visual Studios or GNU C Compiler

Tools: S^{2}E

http://s2e.epfl.ch/

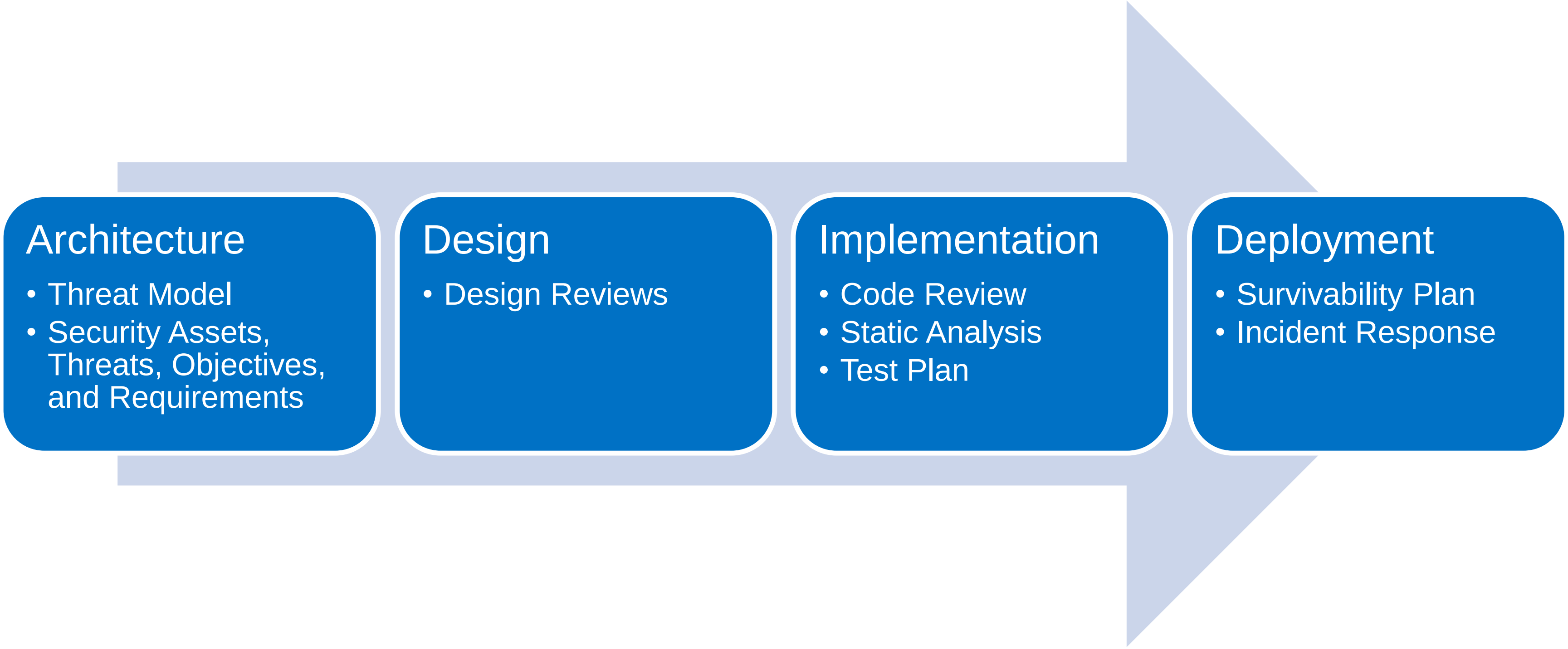
S2E on SMRAM Image



SmmMemoryChecker: address 0xffffffff8172eef4 out of range at pc 0x7b3ec435

Hardening

Development Process



Architecture

- Threat Model
- Security Assets, Threats, Objectives, and Requirements

Design

- Design Reviews

Implementation

- Code Review
- Static Analysis
- Test Plan

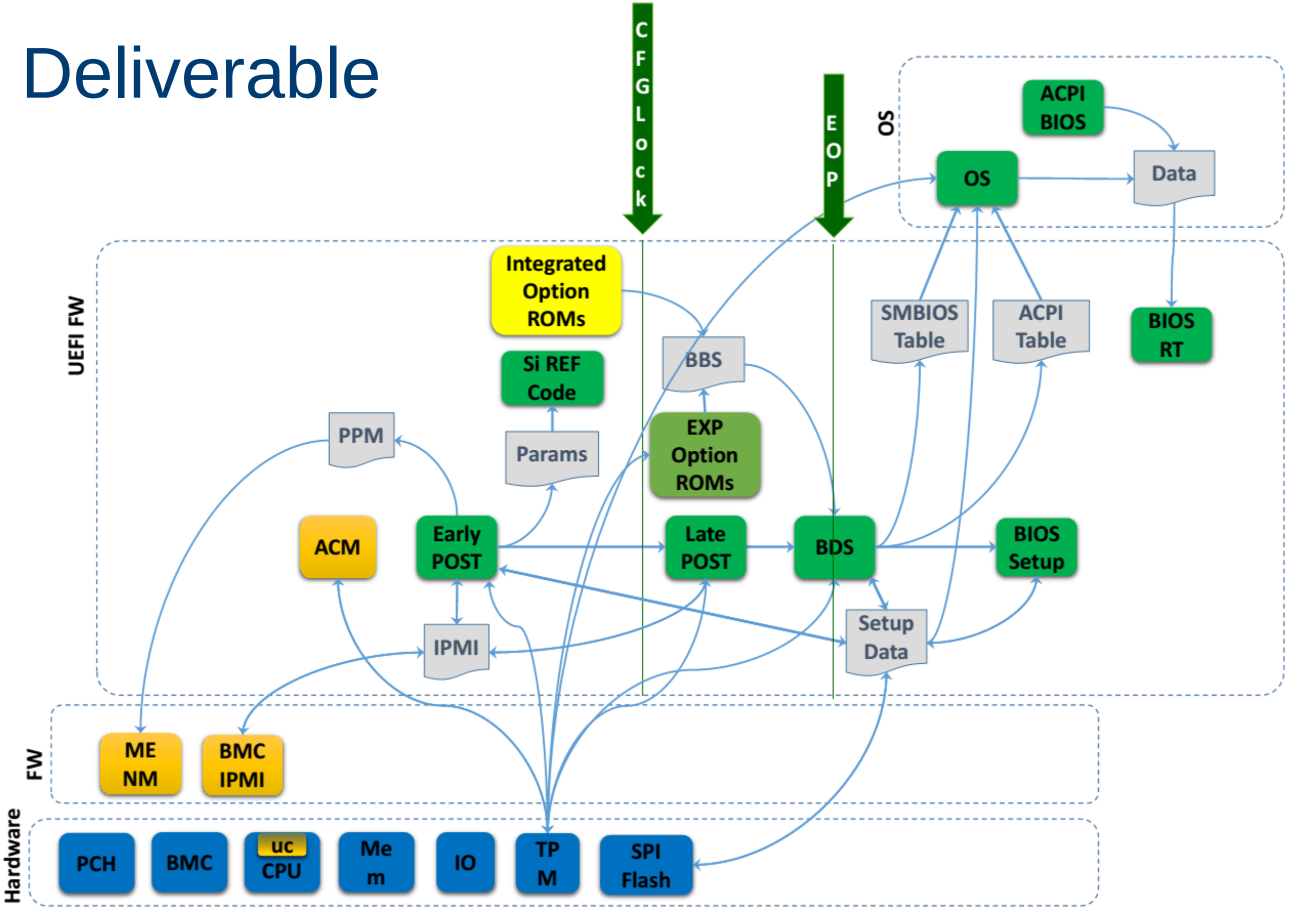
Deployment

- Survivability Plan
- Incident Response

Security VT 5-point Agenda

1. Incident Response
2. Security Requirements
   - Features, technologies, defining secure configuration
3. Secure Design
   - Automate discovery of code-of-concern using Static Analysis Tools
4. Secure Code
   - Custom checkers, go beyond standard plug-ins
5. Security Validation
   - Create test content
   - Test coverage metrics

Know Your Deliverable



🟩 Intel Components
🟩 3rd Party or Non-scope
🟧 Intel Signed & Trusted Components
🟦 HW entrusted Intel components
🟨 3rd Party Option ROM integrated in to BIOS (Legal Contract in place with Vendor)

Twist the Tools, Sync with Process

- Static Analysis into Continuous Integration
- Add Custom Checkers
- Make static analysis do secure design analysis
- Doxygen tags

~~~

- Stay away from project management
  - just feed the issues
  - impact projects metrics
- Define phases of quality goals
- Define security issues closure as milestone exit criteria
~~~

Hardware Security Testability Interface

Initially…

- Mechanism to help avoid misconfiguration of Security Features
- Provide greater assurance of the Security configuration on all deployed platforms - Even after configuration changes by IT and/or end-users
- Programmatic method for an Operating System to query security configuration

Eventually…

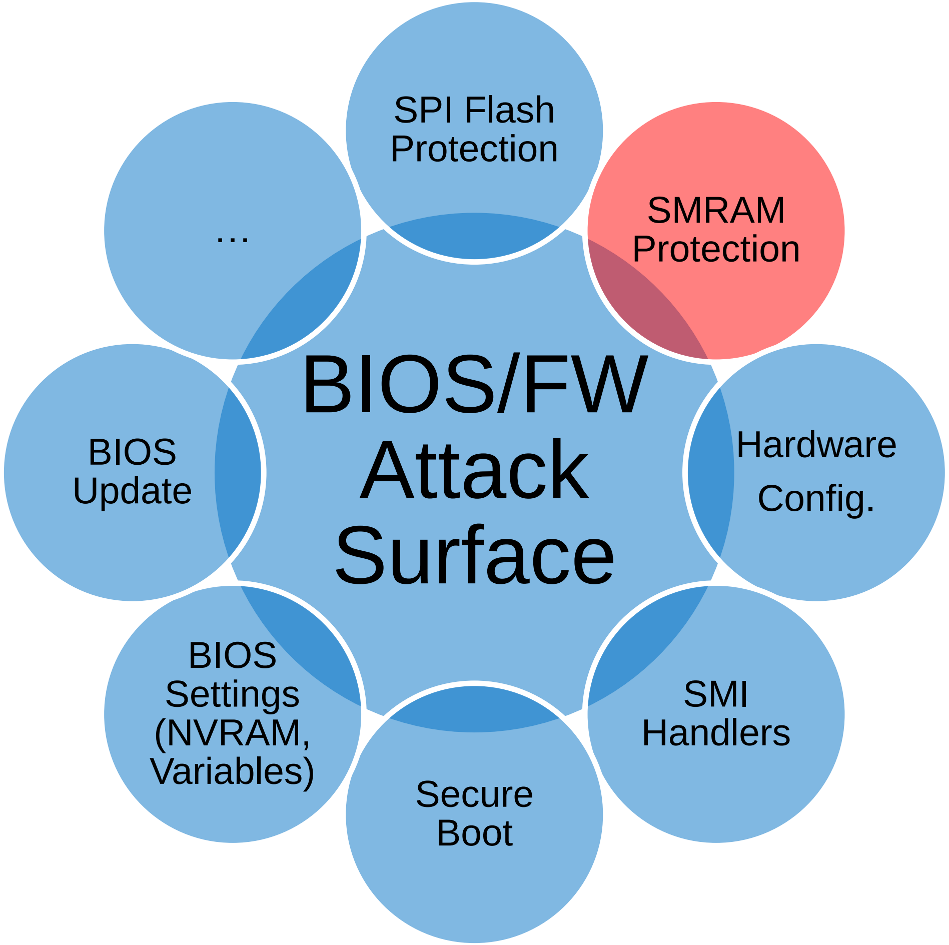
- Establish trust between platform, the Intel-provided test module, and the result published for the Operating System

HSTI Solution Overview

Collaboratively defined API released externally by Microsoft

- Built upon the UEFI 2.4 Adapter Information Protocol
- https://msdn.microsoft.com/en-us/library/windows/hardware/dn879006.aspx
- Defined a variety of Test Categories to be implemented by either the Silicon Vendor, IBV, or OEM

IHV	
1	Hardware Rooted Boot Integrity
2	Boot Firmware Media Protection
3	Signed Firmware Update
4	Measured Boot Enforcement
5	Integrated Device DMA Protection
6	Debug Mode Disabled Verification
7	Secure CPU Configuration
8	Secure System Agent Configuration
9	Secure Memory Configuration
10	Secure Integrated Graphics Configuration
11	Secure I/O Configuration
IBV/OEM	
1	Firmware Trust Continuation Crypto Strength
2	Test Key Verification
3	Firmware Version Rollback Protection
4	SecureBoot Bypass Checking
5	External Device DMA Protection
6	MOR Support



SPI Flash Protection
SMRAM Protection
Hardware Config.
SMI Handlers
Secure Boot
BIOS Settings (NVRAM, Variables)
BIOS Update
…
BIOS/FW Attack Surface

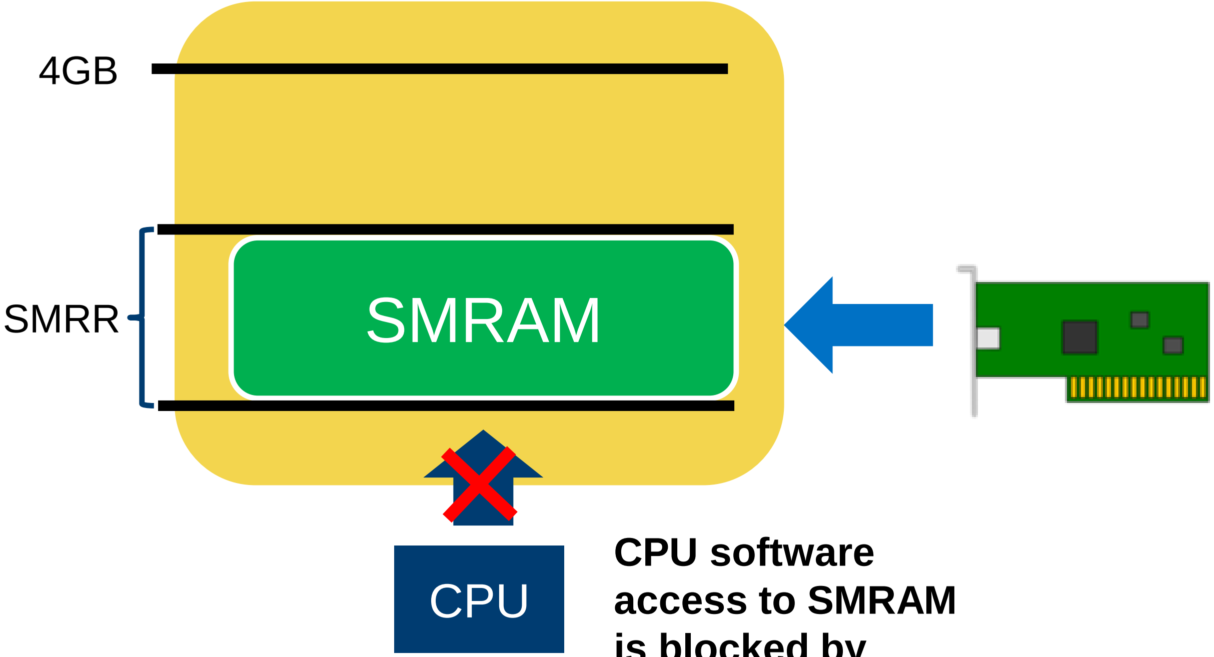
DMA Protection

DMA/GFx Aperture Attacks Against SMRAM

- SMRAM has to be protected from DMA Attack
- Protection from inbound DMA access is guaranteed by programming TSEG range
- When BIOS doesn't lock down TSEG range configuration, malware can move TSEG outside of where actual SMRAM is
- Then program one of DMA capable devices (e.g. GPU device) or Graphics Aperture to access SMRAM
- Programmed I/O accesses: a threat to Virtual Machine Monitors?
- System Management Mode Design and Security Issues

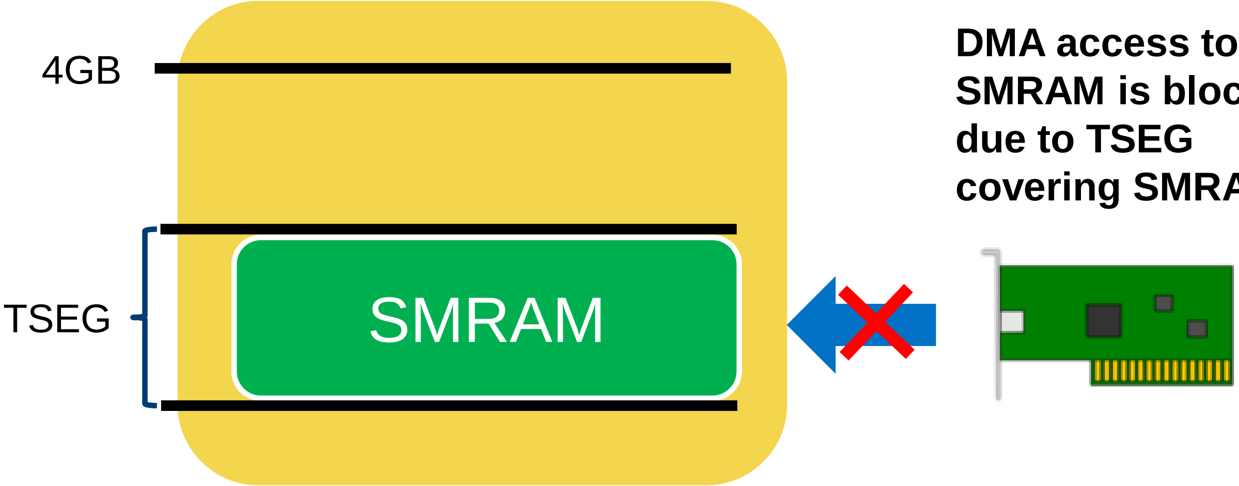
- BIOS has to lock down configuration required to define range protecting SMRAM from inbound DMA access (e.g. TSEG range)
- `chipsec_main --module smm_dma`

Software Access to SMRAM:



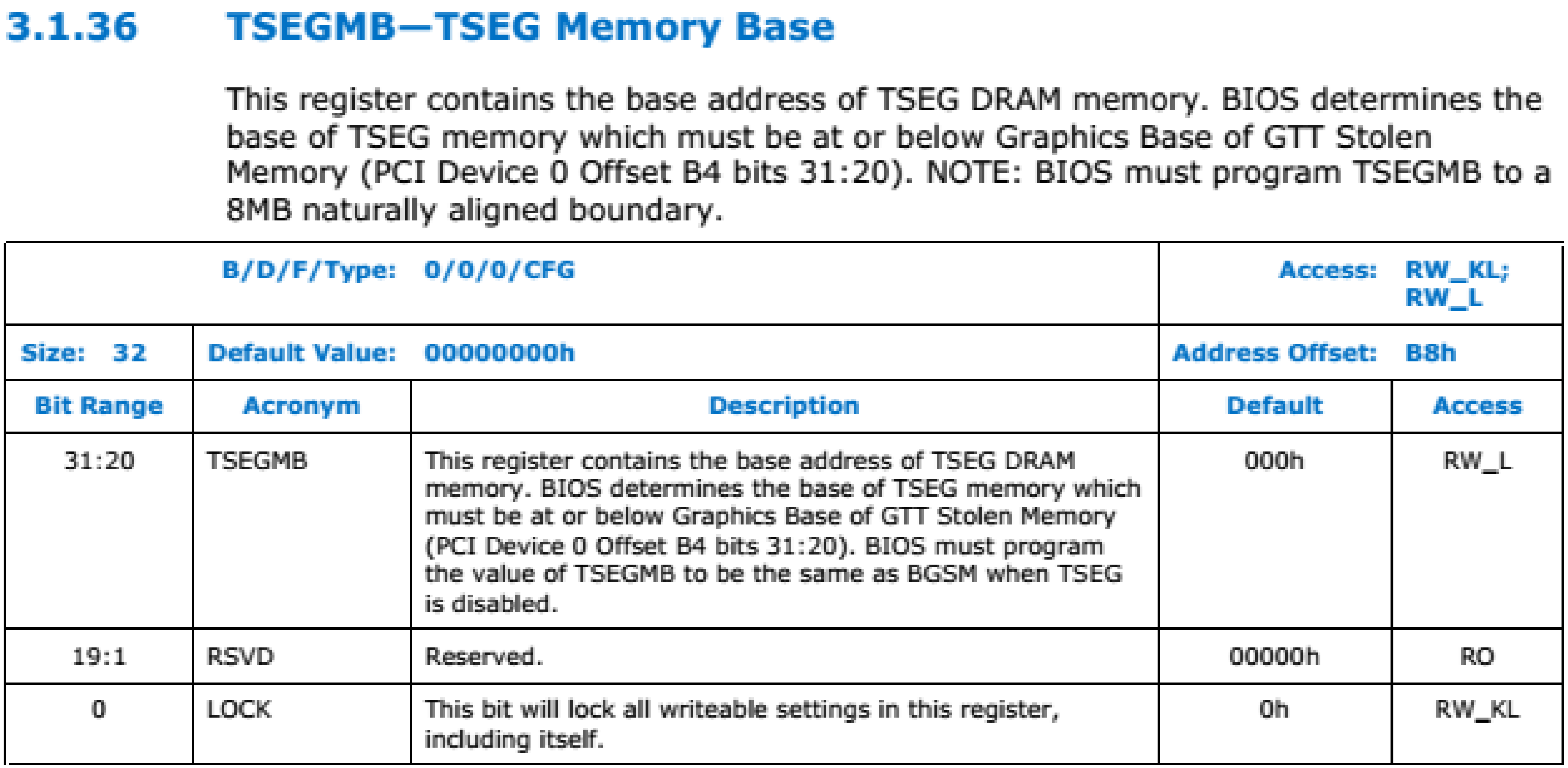
CPU software access to SMRAM is blocked by SMRR

DMA Access to SMRAM:



DMA access to SMRAM is blocked due to TSEG covering SMRAM

DMA Protection of SMRAM – Client

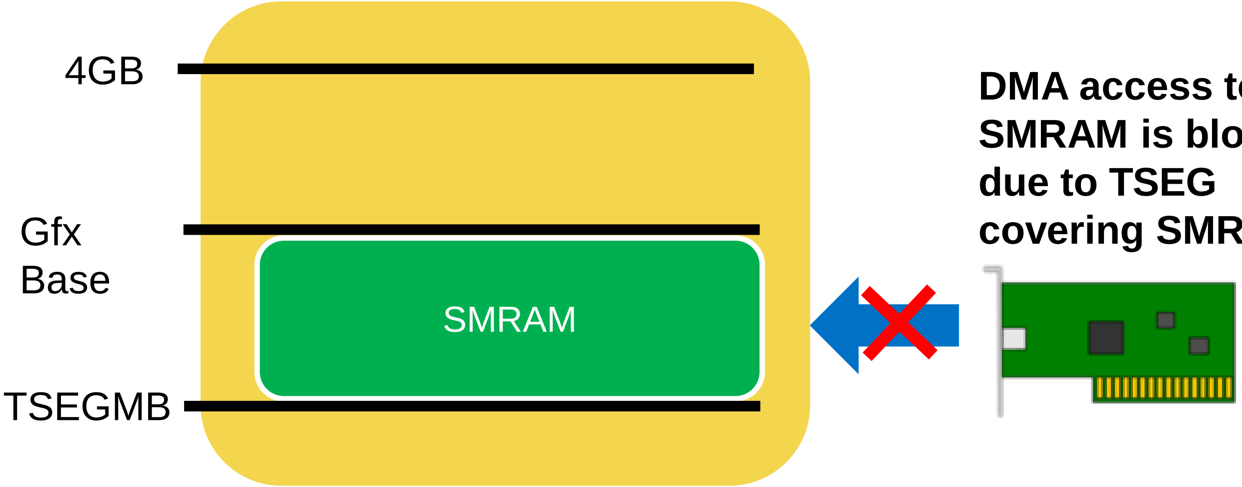


3.1.36 TSEGMB—TSEG Memory Base

This register contains the base address of TSEG DRAM memory. BIOS determines the base of TSEG memory which must be at or below Graphics Base of GTT Stolen Memory (PCI Device 0 Offset B4 bits 31:20). NOTE: BIOS must program TSEGMB to a 8MB naturally aligned boundary.

B/D/F/Type: 0/0/0/CFG			Access: RW_KL; RW_L	
Size: 32	Default Value: 00000000h		Address Offset: B8h	
Bit Range	**Acronym**	**Description**	**Default**	**Access**
31:20	TSEGMB	This register contains the base address of TSEG DRAM memory. BIOS determines the base of TSEG memory which must be at or below Graphics Base of GTT Stolen Memory (PCI Device 0 Offset B4 bits 31:20). BIOS must program the value of TSEGMB to be the same as BGSM when TSEG is disabled.	000h	RW_L
19:1	RSVD	Reserved.	00000h	RO
0	LOCK	This bit will lock all writeable settings in this register, including itself.	0h	RW_KL

DMA Access to SMRAM:



DMA access to SMRAM is blocked due to TSEG covering SMRAM

Servers are People (computers) too ☺

Architectural differences

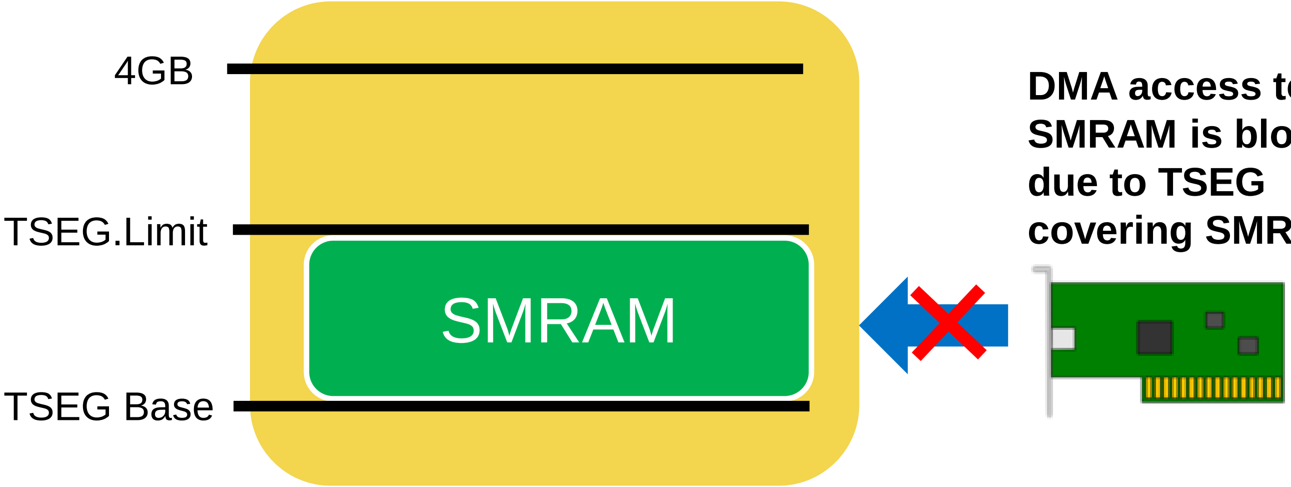
- Memory
- Multi-processor

Register implementation differences

Feature support differences

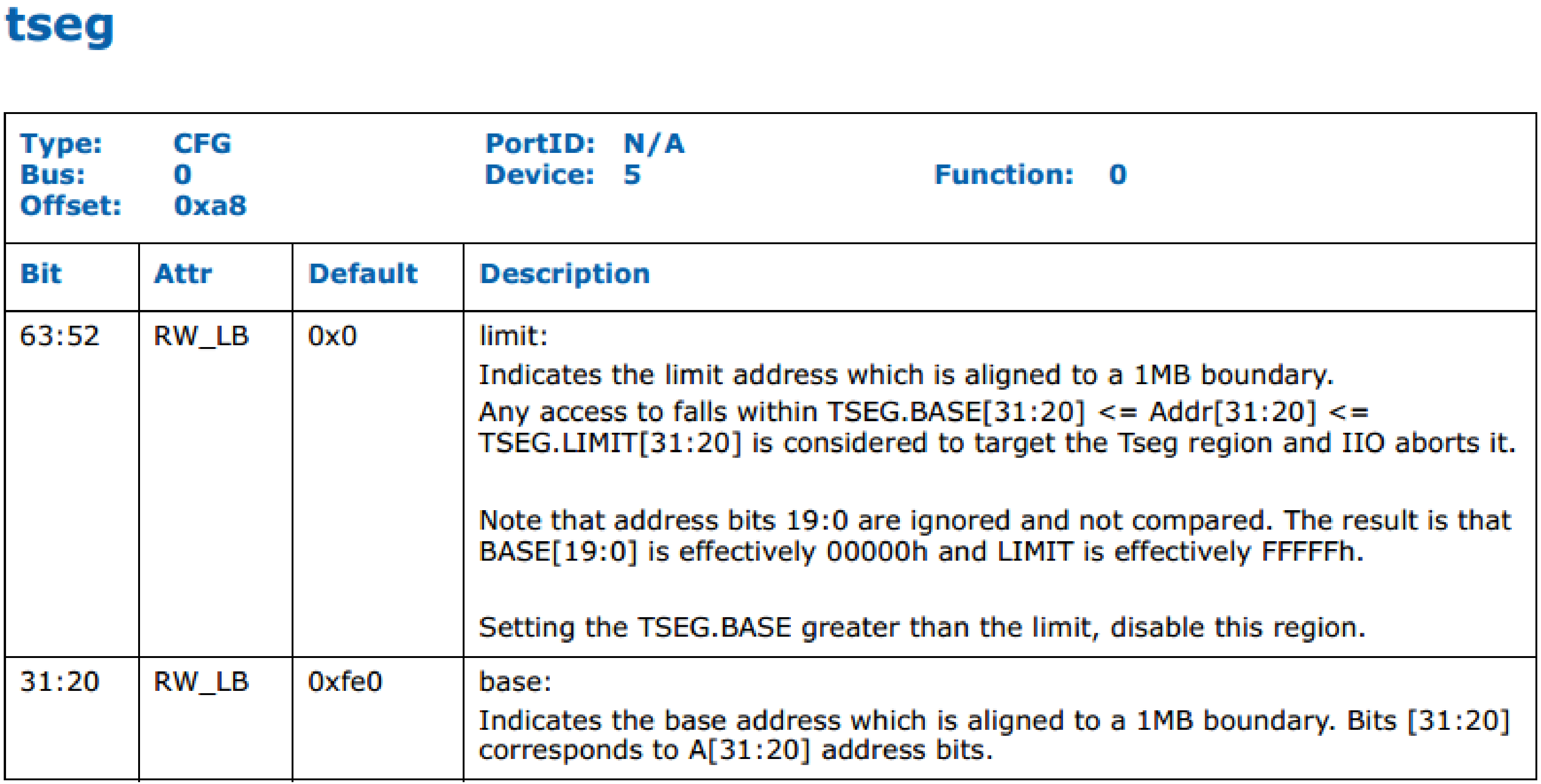
Mitigation differences

DMA Access to SMRAM:



DMA access to SMRAM is blocked due to TSEG covering SMRAM

DMA Protection of SMRAM – Server



tseg

Type:	**CFG**	**PortID:**	**N/A**		
Bus:	**0**	**Device:**	**5**	**Function:**	**0**
Offset:	**0xa8**				

Bit	**Attr**	**Default**	**Description**
63:52	RW_LB	0x0	limit: Indicates the limit address which is aligned to a 1MB boundary. Any access to falls within TSEG.BASE[31:20] <= Addr[31:20] <= TSEG.LIMIT[31:20] is considered to target the Tseg region and IIO aborts it. Note that address bits 19:0 are ignored and not compared. The result is that BASE[19:0] is effectively 00000h and LIMIT is effectively FFFFFh. Setting the TSEG.BASE greater than the limit, disable this region.
31:20	RW_LB	0xfe0	base: Indicates the base address which is aligned to a 1MB boundary. Bits [31:20] corresponds to A[31:20] address bits.

GENPROTRANGE

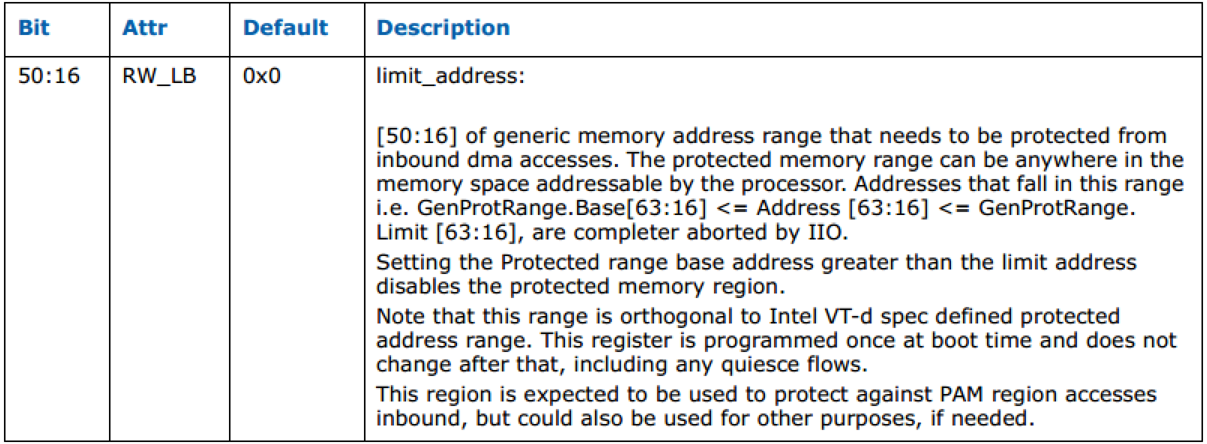
TSEG Protects SMRAM (along with SMRR)…

Generic Protected Memory Range

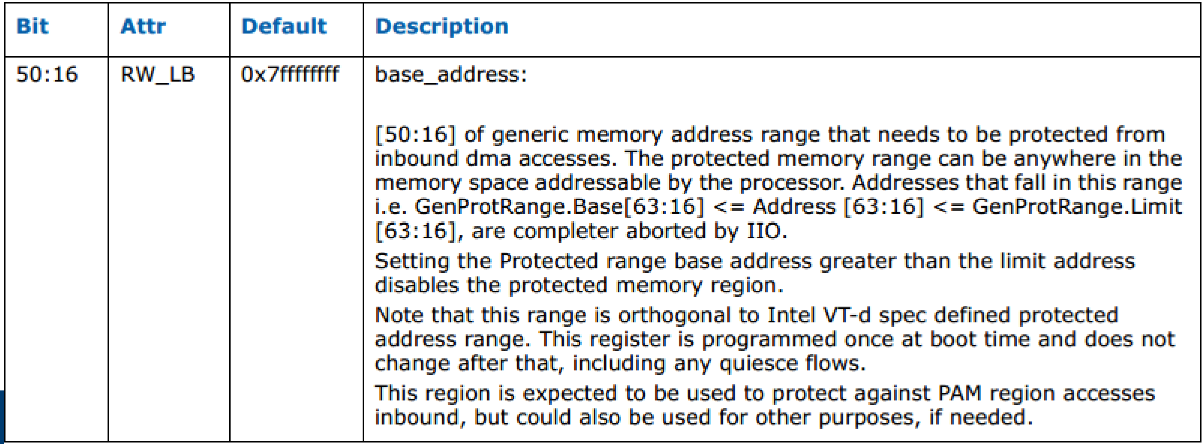
Using GENPROTRANGE, BIOS can protect other address ranges.

- Base
- Limit
- Base > Limit = disabled

GENPROTRANGE Base and Limit

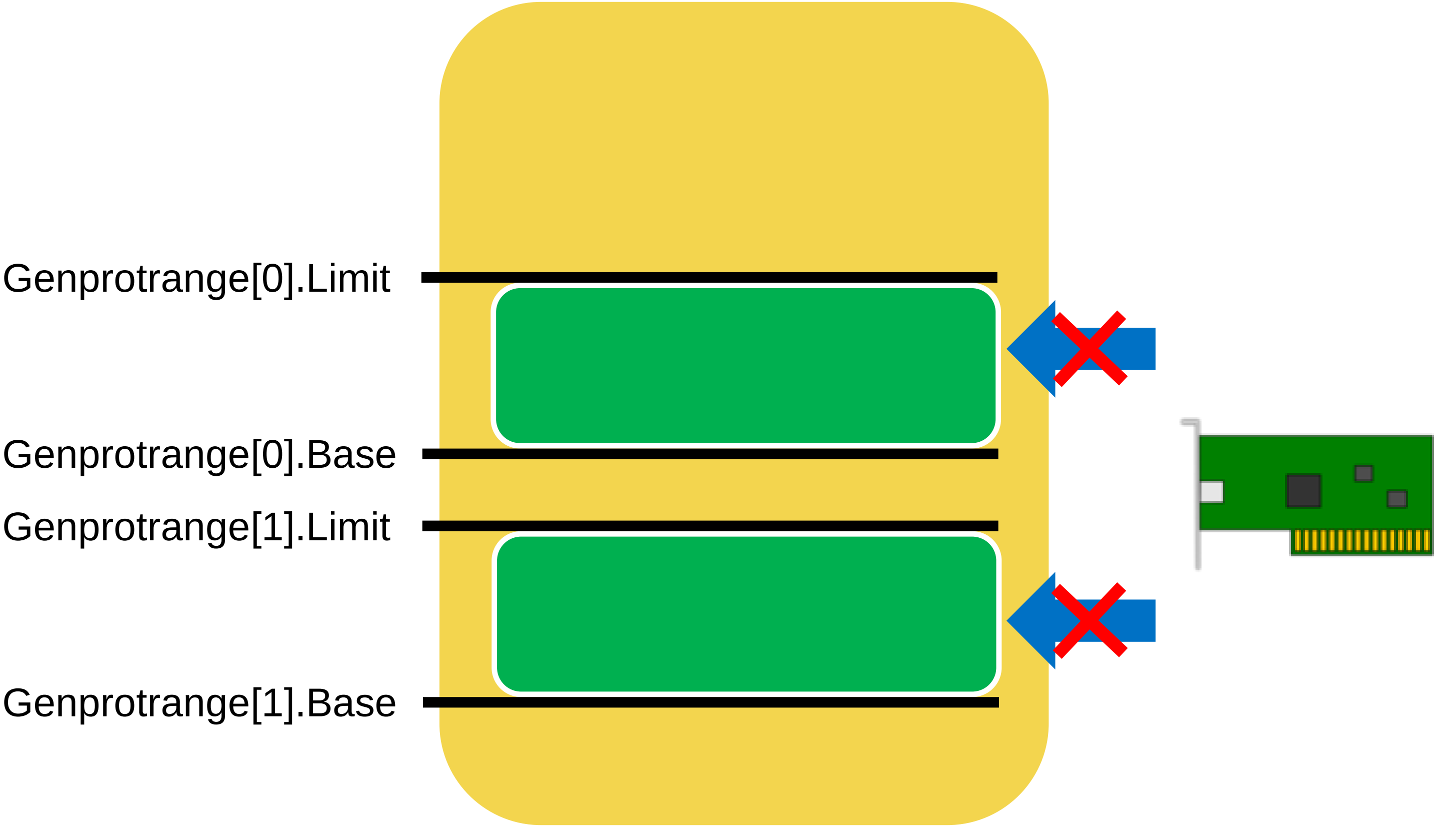


Bit	Attr	Default	Description
50:16	RW_LB	0x0	limit_address: [50:16] of generic memory address range that needs to be protected from inbound dma accesses. The protected memory range can be anywhere in the memory space addressable by the processor. Addresses that fall in this range i.e. GenProtRange.Base[63:16] <= Address [63:16] <= GenProtRange. Limit [63:16], are completer aborted by IIO. Setting the Protected range base address greater than the limit address disables the protected memory region. Note that this range is orthogonal to Intel VT-d spec defined protected address range. This register is programmed once at boot time and does not change after that, including any quiesce flows. This region is expected to be used to protect against PAM region accesses inbound, but could also be used for other purposes, if needed.



Bit	Attr	Default	Description
50:16	RW_LB	0x7ffffffff	base_address: [50:16] of generic memory address range that needs to be protected from inbound dma accesses. The protected memory range can be anywhere in the memory space addressable by the processor. Addresses that fall in this range i.e. GenProtRange.Base[63:16] <= Address [63:16] <= GenProtRange.Limit [63:16], are completer aborted by IIO. Setting the Protected range base address greater than the limit address disables the protected memory region. Note that this range is orthogonal to Intel VT-d spec defined protected address range. This register is programmed once at boot time and does not change after that, including any quiesce flows. This region is expected to be used to protect against PAM region accesses inbound, but could also be used for other purposes, if needed.

DMA Access to SMRAM:



Genprotrange[0].Limit

Genprotrange[0].Base

Genprotrange[1].Limit

Genprotrange[1].Base

Community & Ecosystem Engagement

After all, we’re in this together…

Response Process

1. Issue Discovery
   - ➢ secure@intel.com security@uefi.org (other security teams)
2. Investigation and Disposition of Impacts
   - ➢ PSIRT BIOS Core, representing BIOS leads in every segment
3. Mitigation Development and Validation
4. Non-public Communication for Mitigation Development/Deployment
   - ➢ Technical Advisories
   - ➢ USRT Coordination (http://uefi.org/security)
5. Public Communication
   - ➢ https://security-center.intel.com
   - ➢ http://tianocore.org/security

CHIPSEC Information

On the web: https://github.com/chipsec/chipsec

Email List: https://lists.01.org/mailman/listinfo/chipsec

- Gaining activity/support from community
- External interest in
  - Deployable drivers
  - Supporting other platforms/architectures
  - Greater coverage of issues
  - Increased user-friendliness

Contact: chipsec@intel.com

Putting it together…

Finding Issues

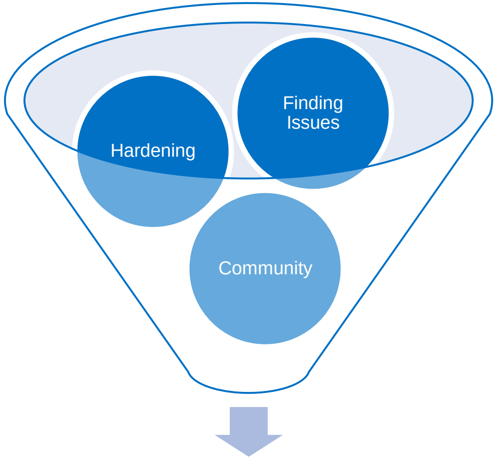
- Research
- Testing

Hardening

- People
- Process
- Tools

Community

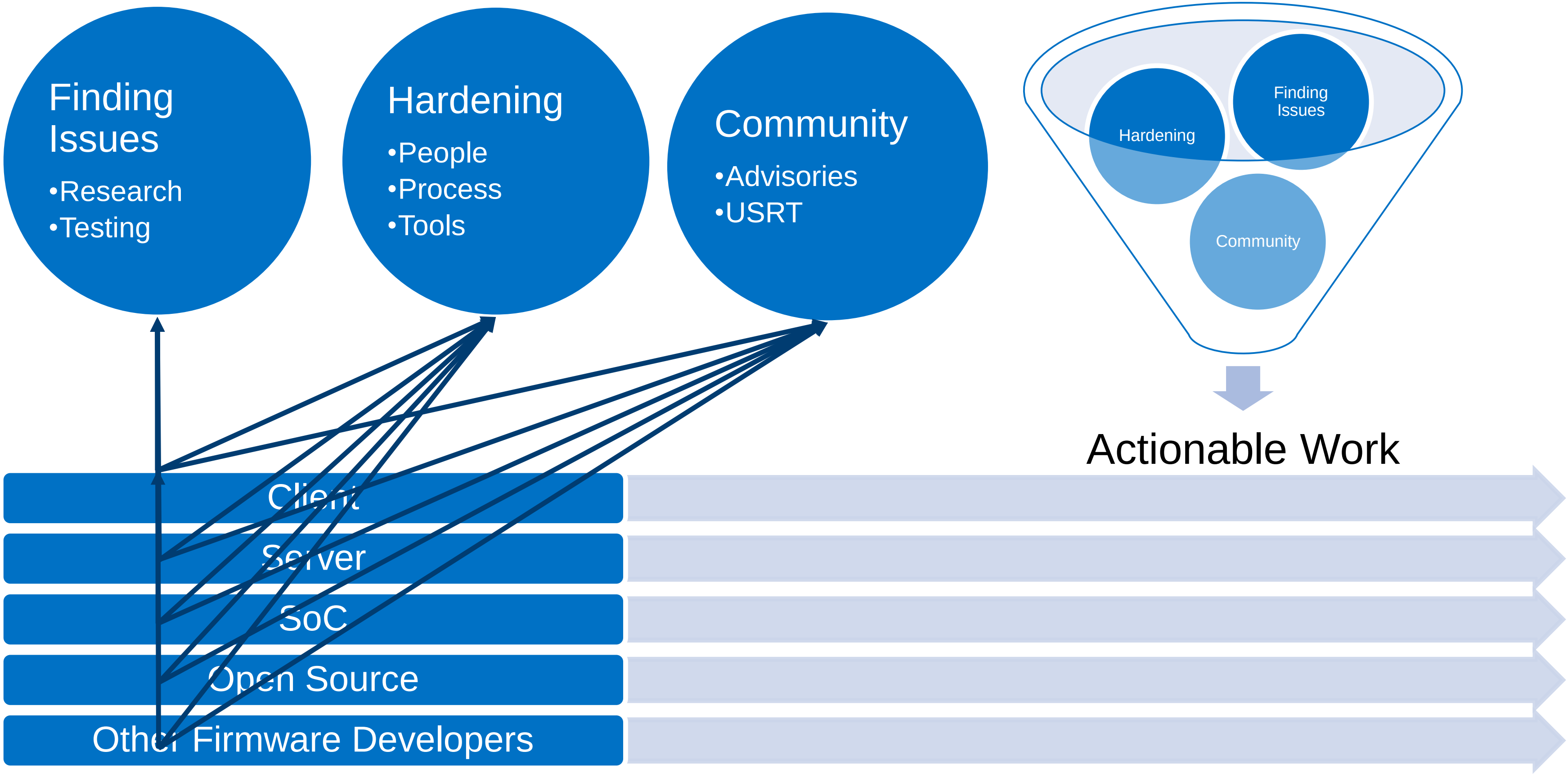
- Advisories
- USRT



Actionable Work

- Client
- Server
- SoC
- Open Source
- Other Firmware Developers

Putting it together…



How this applies to YOU…

- Look for Issues and Mitigations
  - Vulnerability Research
  - Platform-specific assessments
    - Next level of detail for concepts like “non-bypassible” in 800-147
    - Could include testing such as with the CHIPSEC Framework
- Harden the Implementation
  - Consider both development practices and response/update
  - Consider technologies like secure boot,
- Engage the Community
  - UEFI Forum (USRT & USST)

intel®

experience
what's inside™

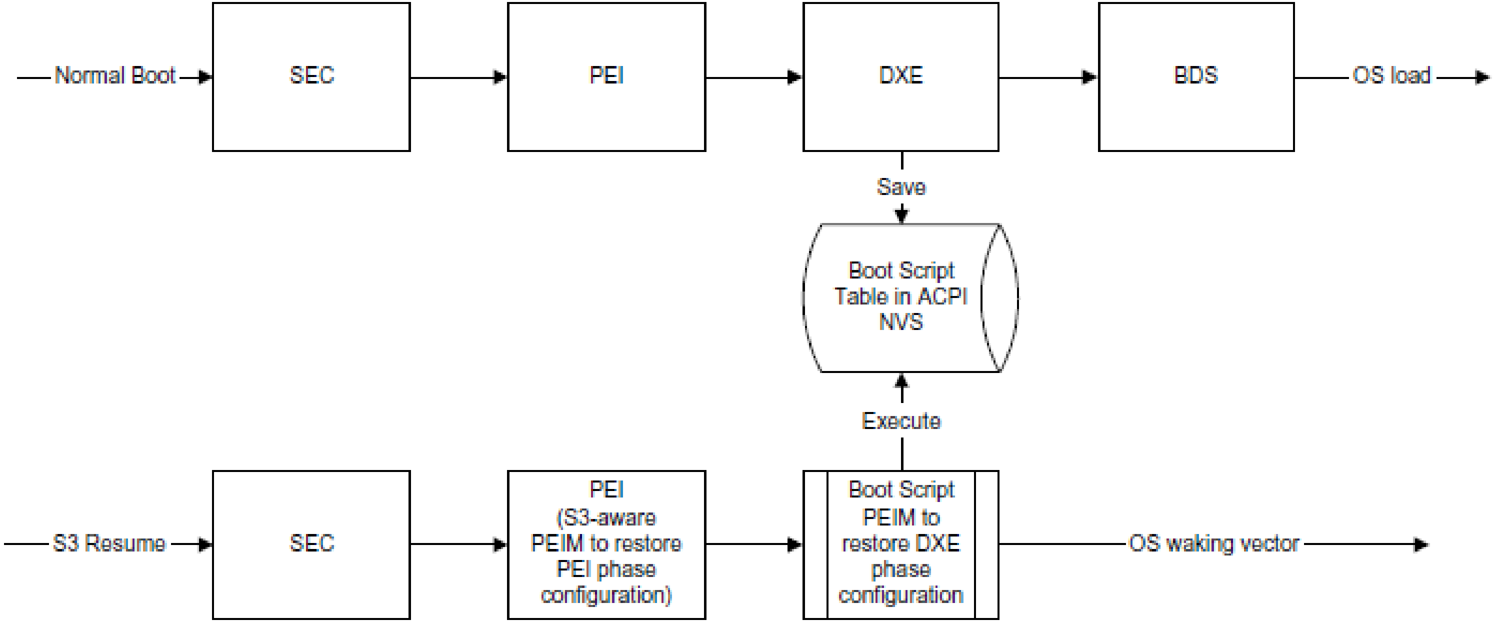
Please visit Intel in the Show Floor Area in Booth 100

S3 Boot Script

Example 1

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
```

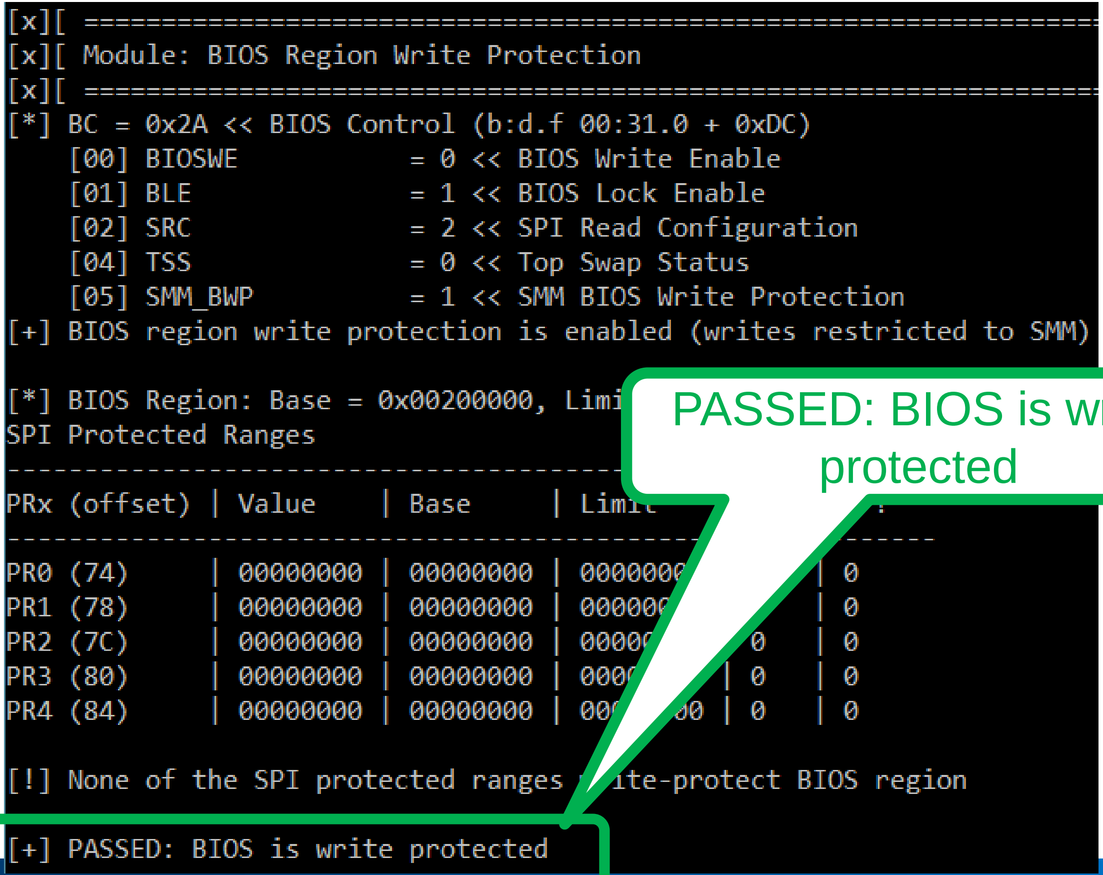
S3 Boot Script Opcodes

- 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 (**`0x09`**)
- Information (**`0x0A`**)
- …

So what can go wrong with the script?

1. Address (pointer) to S3 boot script is not protected (eg. **`NV+RT+BS AcpiGlobalTable`**)
2. The S3 boot script itself is stored in memory (eg. ACPI NVS) accessible to the OS or DMA capable devices
3. The executable parsing and interpreting the S3 boot script is running out of unprotected memory
4. S3 boot script contains **`Dispatch`** (or **`Dispatch2`**) opcodes with entry-points in unprotected memory
5. EFI firmware does not add S3 boot script opcodes which restore all required hardware locks and protections

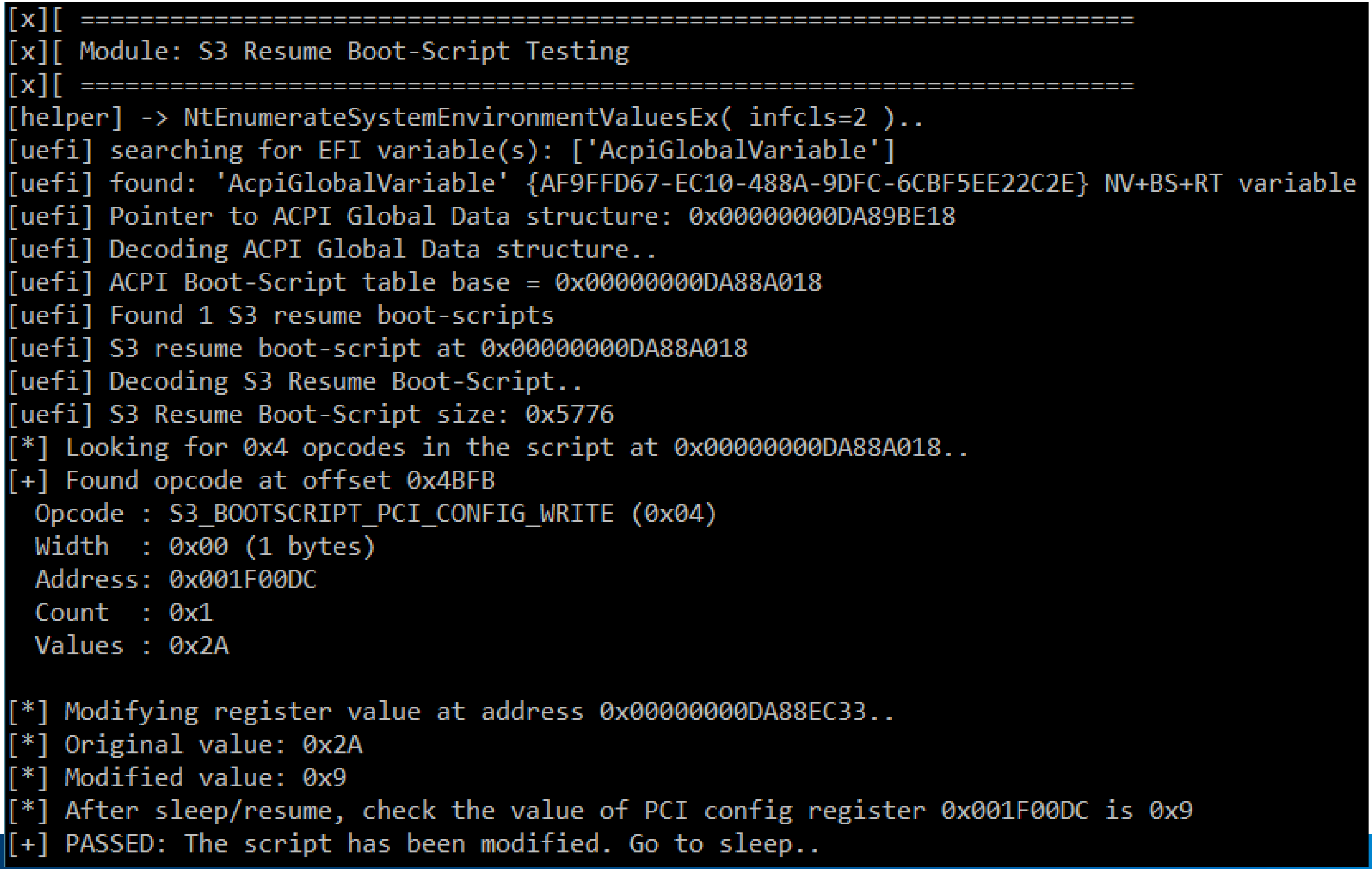
BIOS protection is ON



```
[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 | 0000           0      | 0
PR3 (80)     | 00000000 | 00000000 | 000         | 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
```

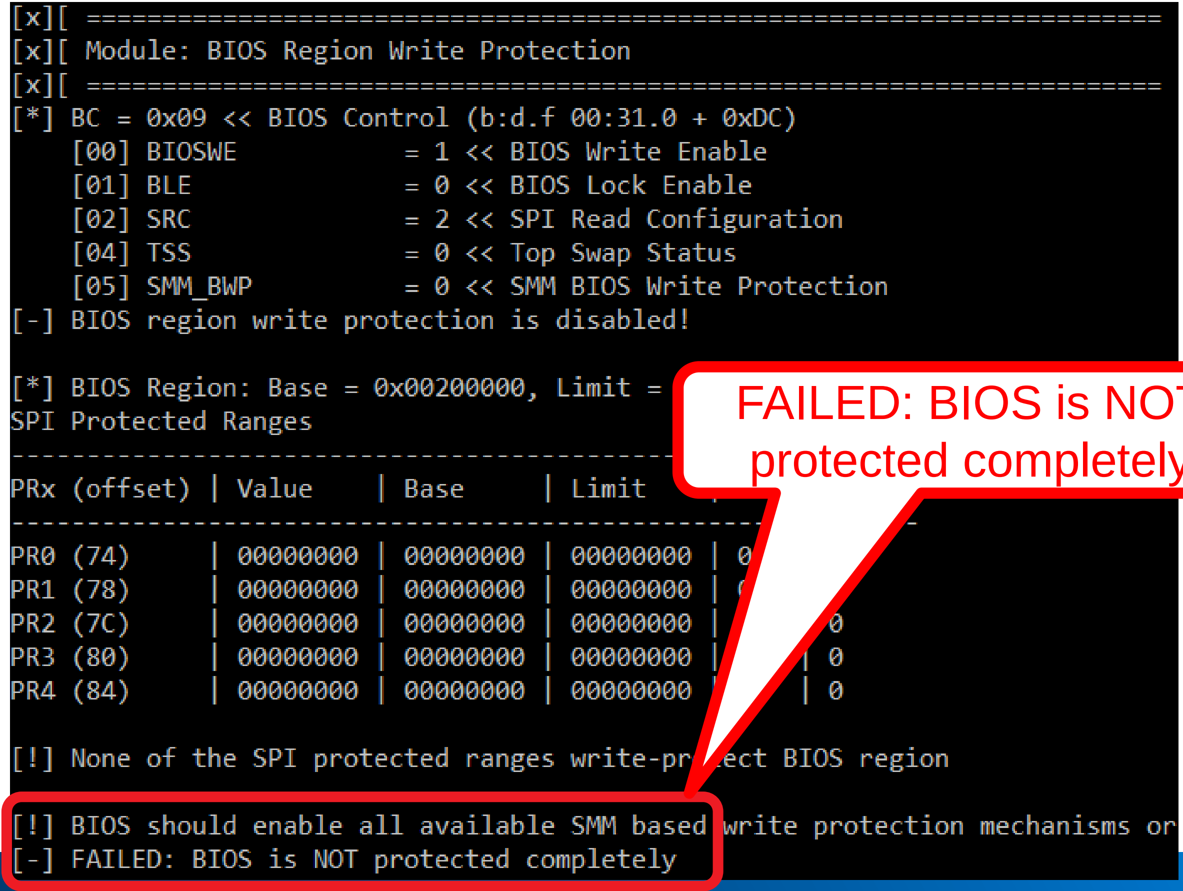
PASSED: BIOS is write protected

Sleep well!



```
[x][ =======================================================================
[x][ Module: S3 Resume Boot-Script Testing
[x][ =======================================================================
[helper] -> NtEnumerateSystemEnvironmentValuesEx( infcls=2 )..
[uefi] searching for EFI variable(s): ['AcpiGlobalVariable']
[uefi] found: 'AcpiGlobalVariable' {AF9FFD67-EC10-488A-9DFC-6CBF5EE22C2E} NV+BS+RT variable
[uefi] Pointer to ACPI Global Data structure: 0x00000000DA89BE18
[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 0x00000000DA88A018..
[+] 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..
```

Oh wait…

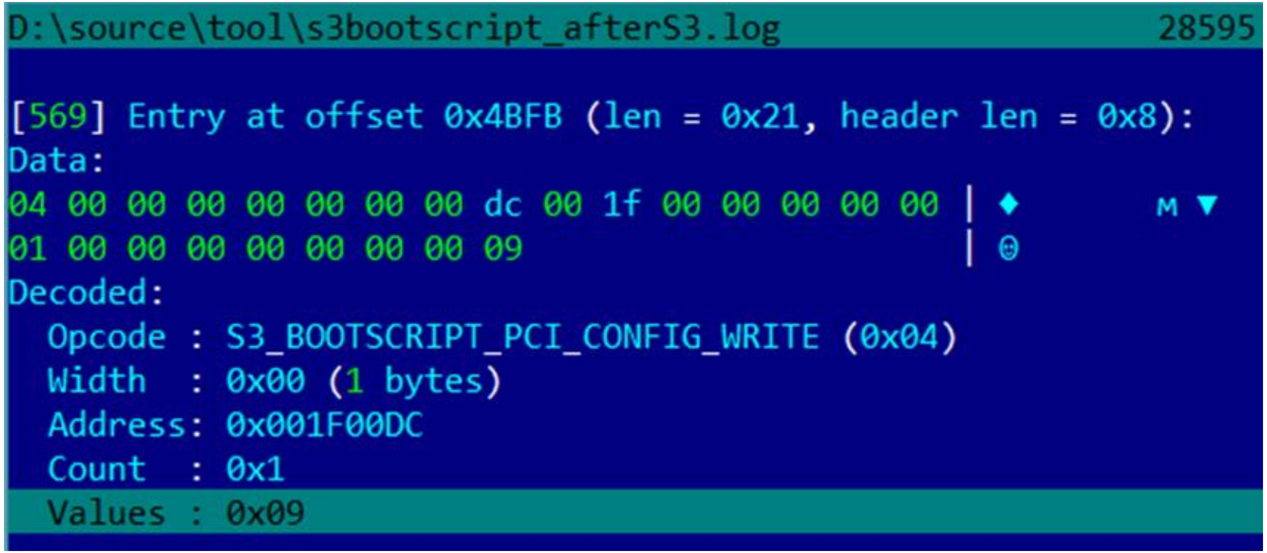


```
[x][ =======================================================================
[x][ Module: BIOS Region Write Protection
[x][ =======================================================================
[*] BC = 0x09 << BIOS Control (b:d.f 00:31.0 + 0xDC)
    [00] BIOSWE              = 1 << BIOS Write Enable
    [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
```

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
```

Protecting the S3 Boot Script

1. Do not keep address (pointer) to S3 Boot Script table in unprotected non-volatile UEFI variable (ex. **`NV+RT+BS AcpiGlobalVariable`**)

2. Do not keep address (pointer) to S3 Boot Script table or to a structure containing address of the script in unprotected non-volatile UEFI variable (ex. **`NV+RT+BS AcpiGlobalVariable`**)

3. Do not save the S3 Boot Script table to memory accessible by the OS or DMA capable devices

4. 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

5. Saving a pointer to the S3 boot script in a UEFI variable that is accessible by the OS or DMA capable devices

6. Review the S3 Boot Script for Dispatch opcodes and establish whether the target code is protected.

SMI Handler Call-Outs Vulnerabilities

Example 2

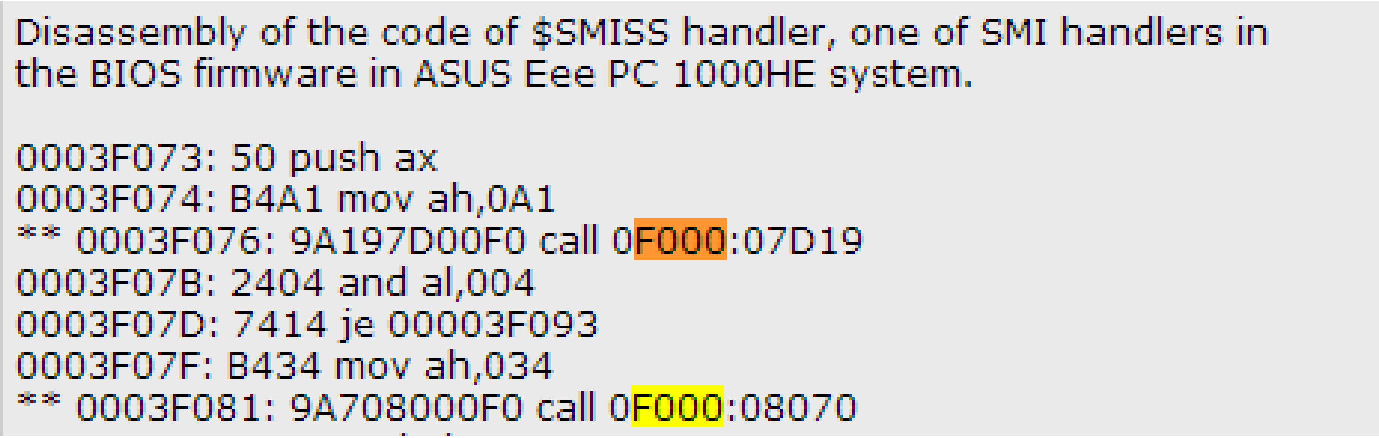
Once upon a time…

- **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



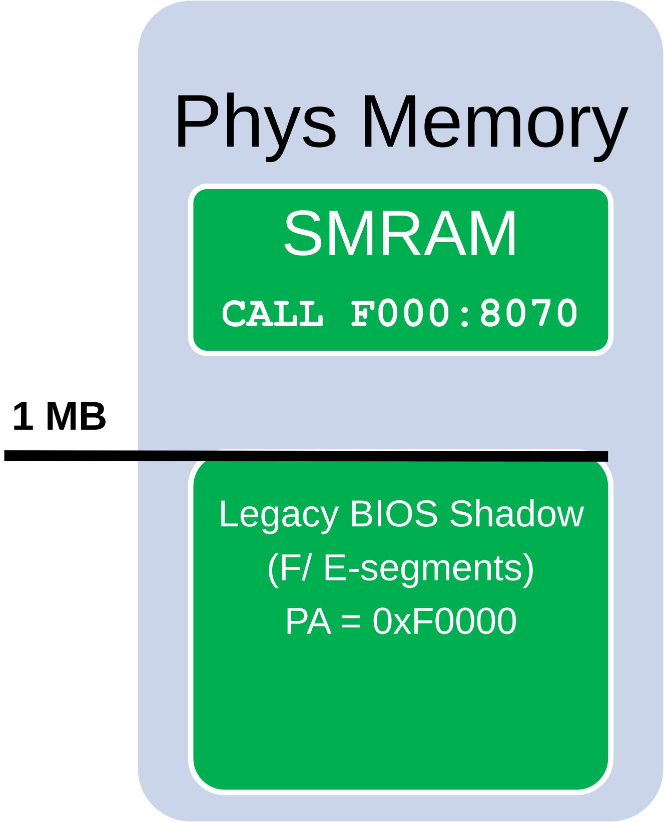
Disassembly of the code of $SMISS handler, one of SMI handlers in the BIOS firmware in ASUS Eee PC 1000HE system.

```
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

Legacy SMI Handlers Calling Out of SMRAM



Phys Memory

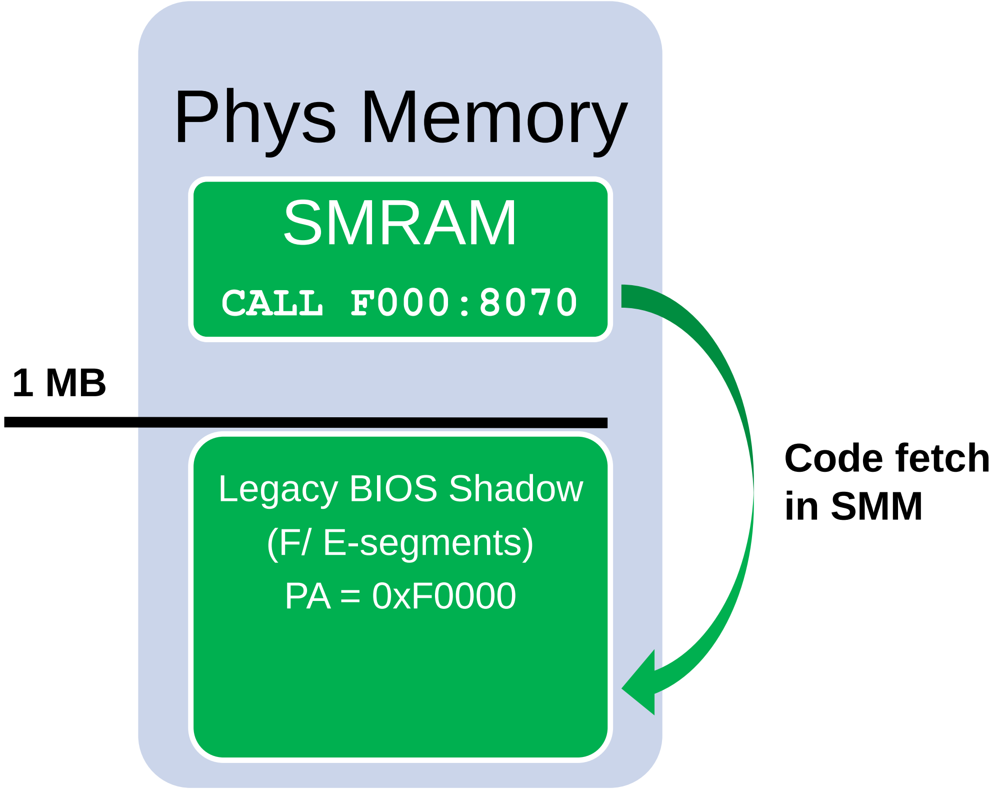
SMRAM

`CALL F000:8070`

1 MB

Legacy BIOS Shadow
(F/ E-segments)
PA = 0xF0000

SMI Handlers Calling Out of SMRAM



Phys Memory

SMRAM

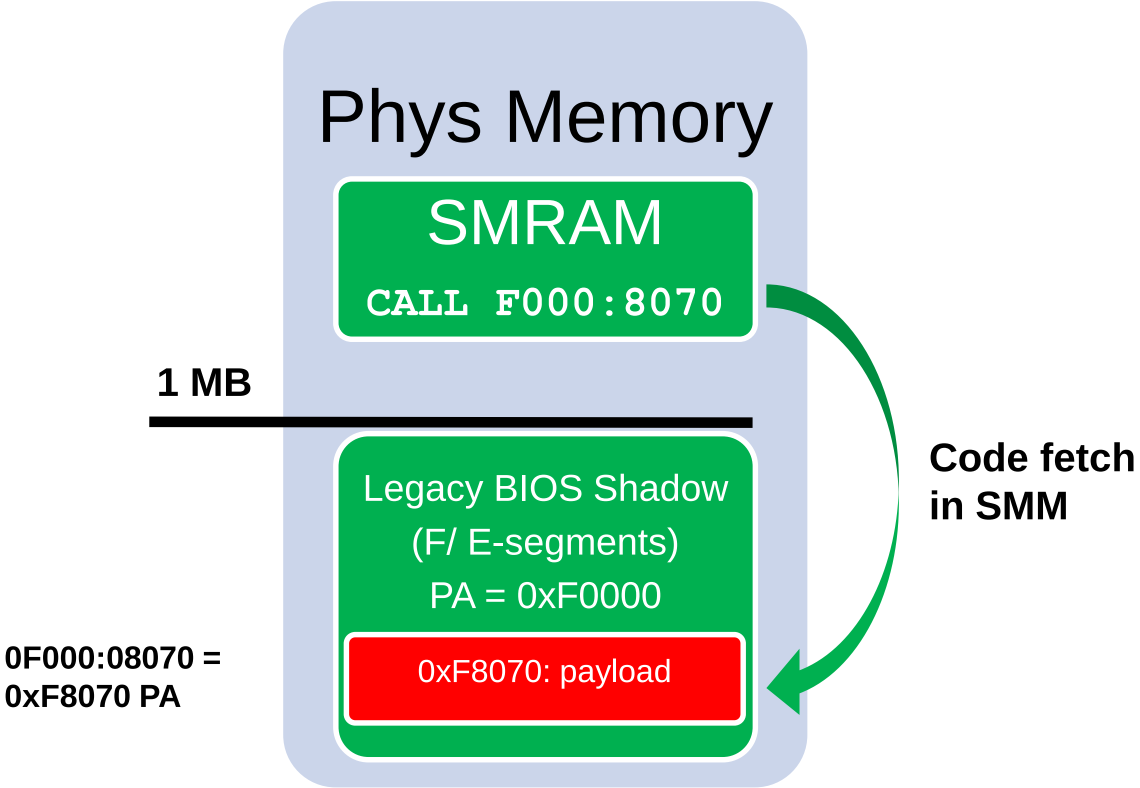
`CALL F000:8070`

1 MB

Legacy BIOS Shadow
(F/ E-segments)
PA = 0xF0000

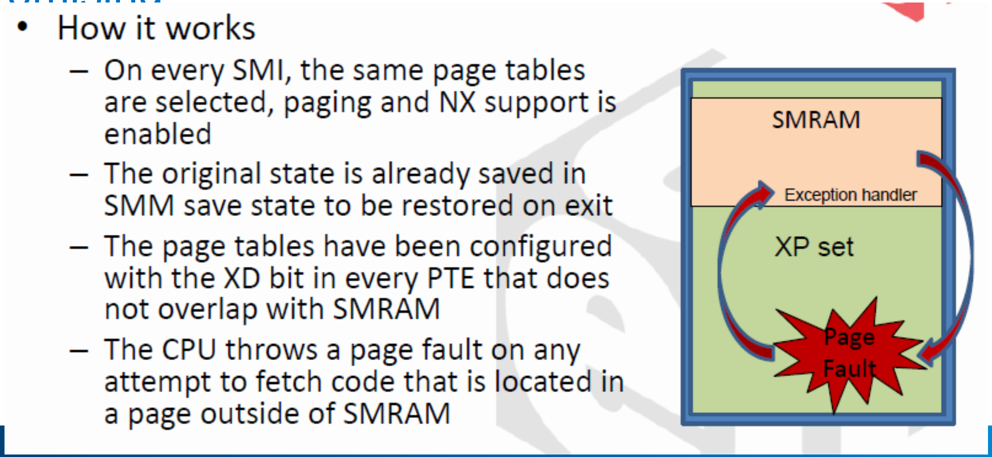
Code fetch
in SMM

SMI Handlers Calling Out of SMRAM



BIOS developers can easily detect call-outs

1. A “simple” HW 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

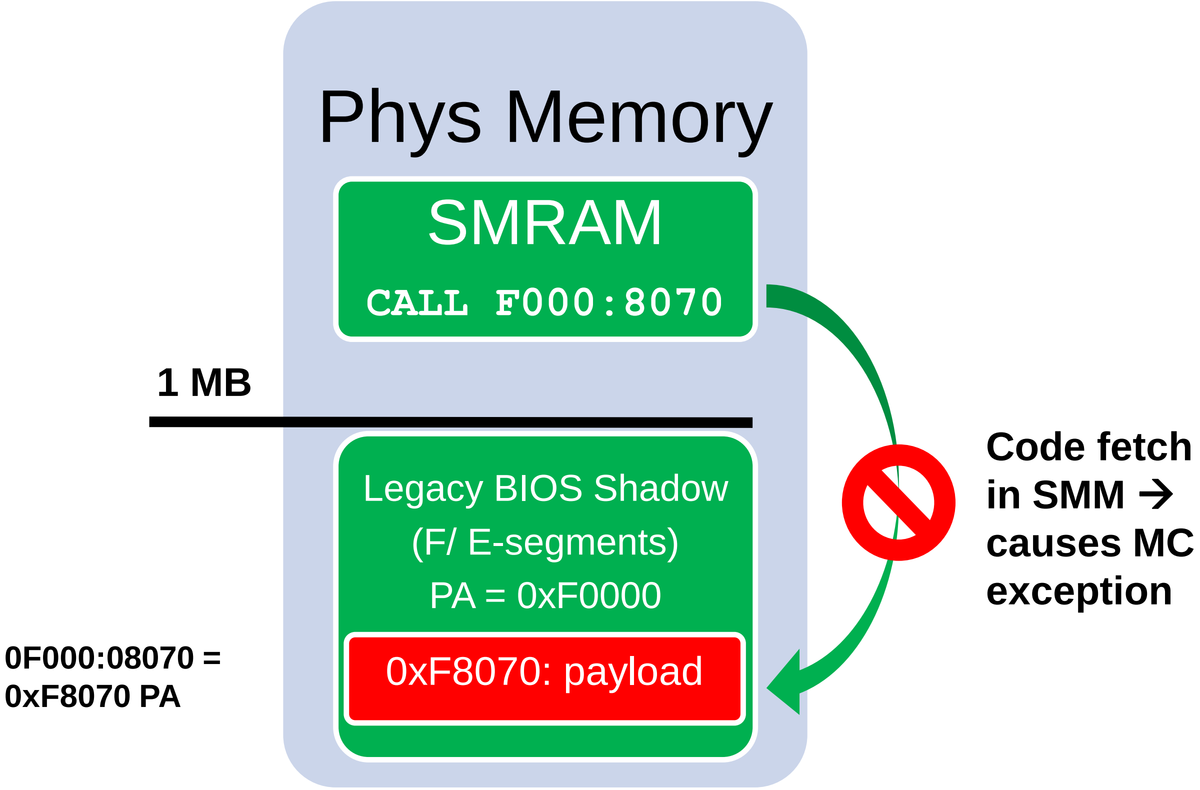
SMRAM
Exception handler
XP set
Page Fault

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



Phys Memory

SMRAM

`CALL F000:8070`

1 MB

Legacy BIOS Shadow (F/ E-segments)

PA = 0xF0000

0xF8070: payload

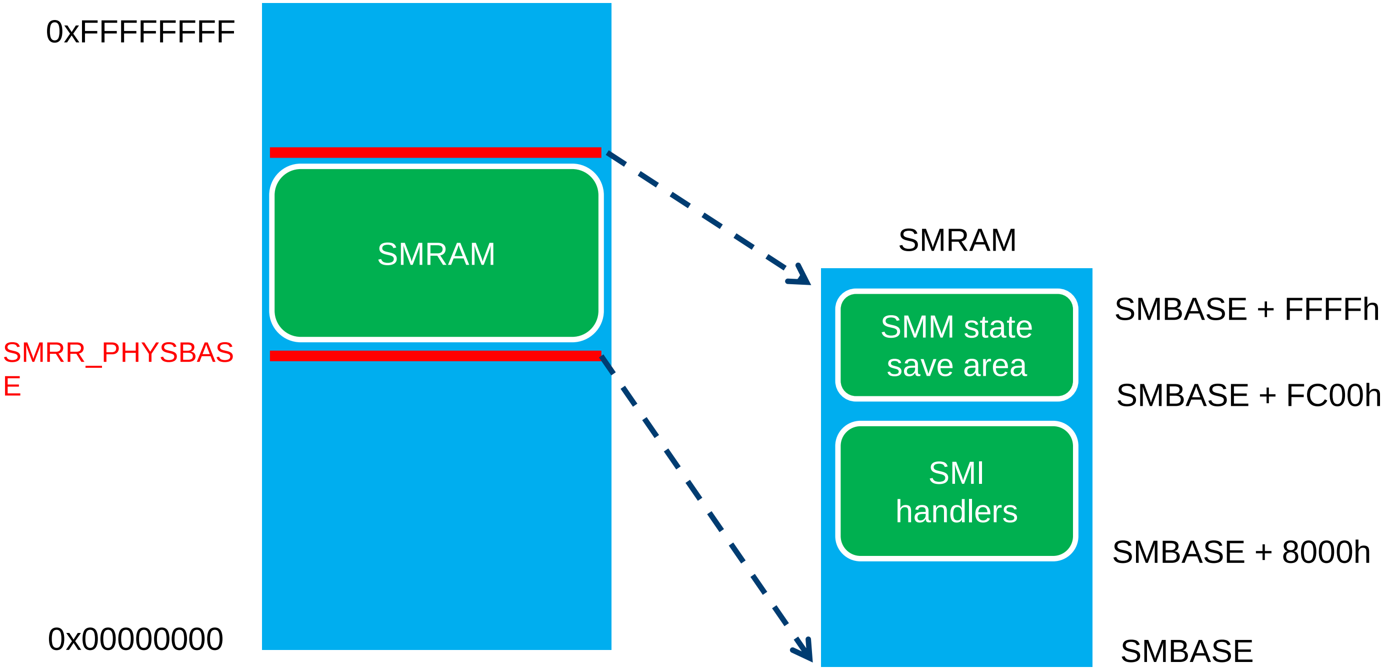
0F000:08070 = 0xF8070 PA

Code fetch in SMM → causes MC exception

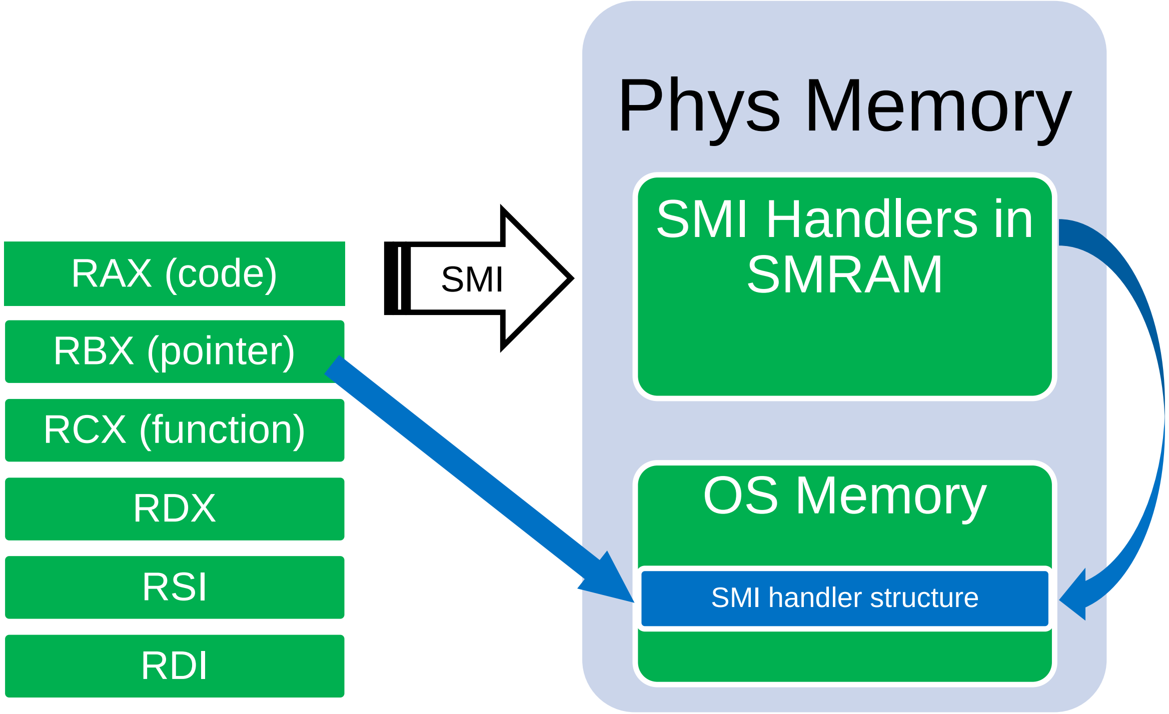
Bad Input Pointer Vulnerabilities

Example 3

System Management Interrupt (SMI) Handlers

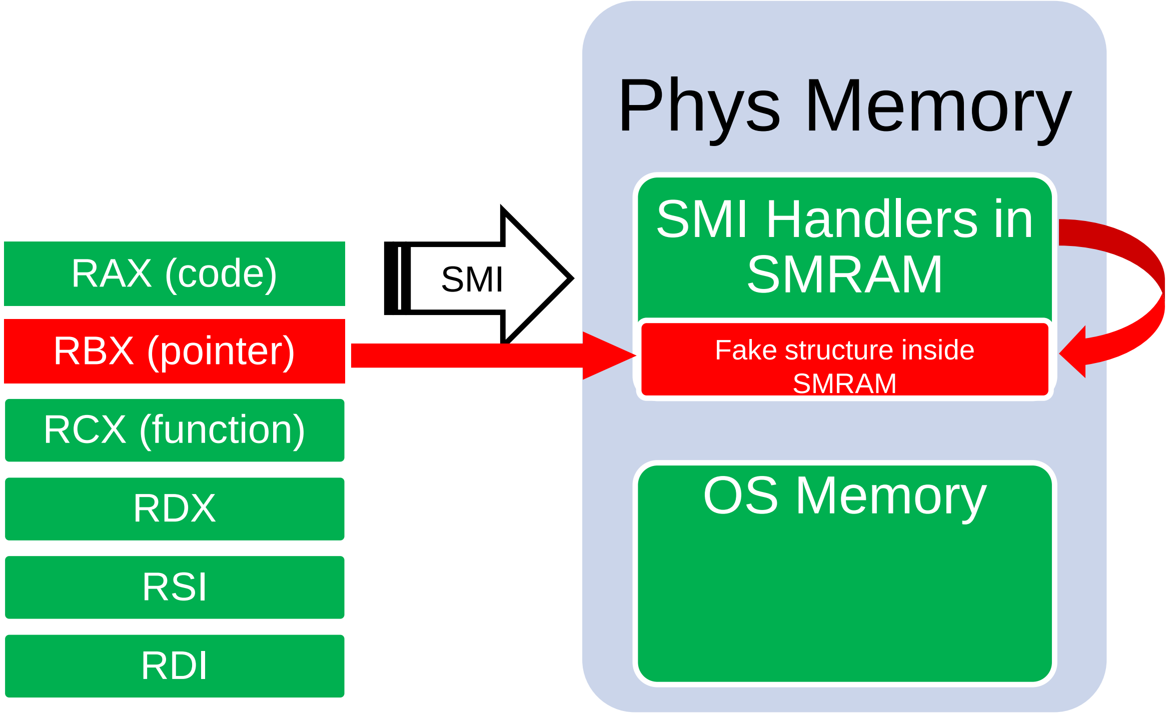


Pointer Arguments to SMI Handlers



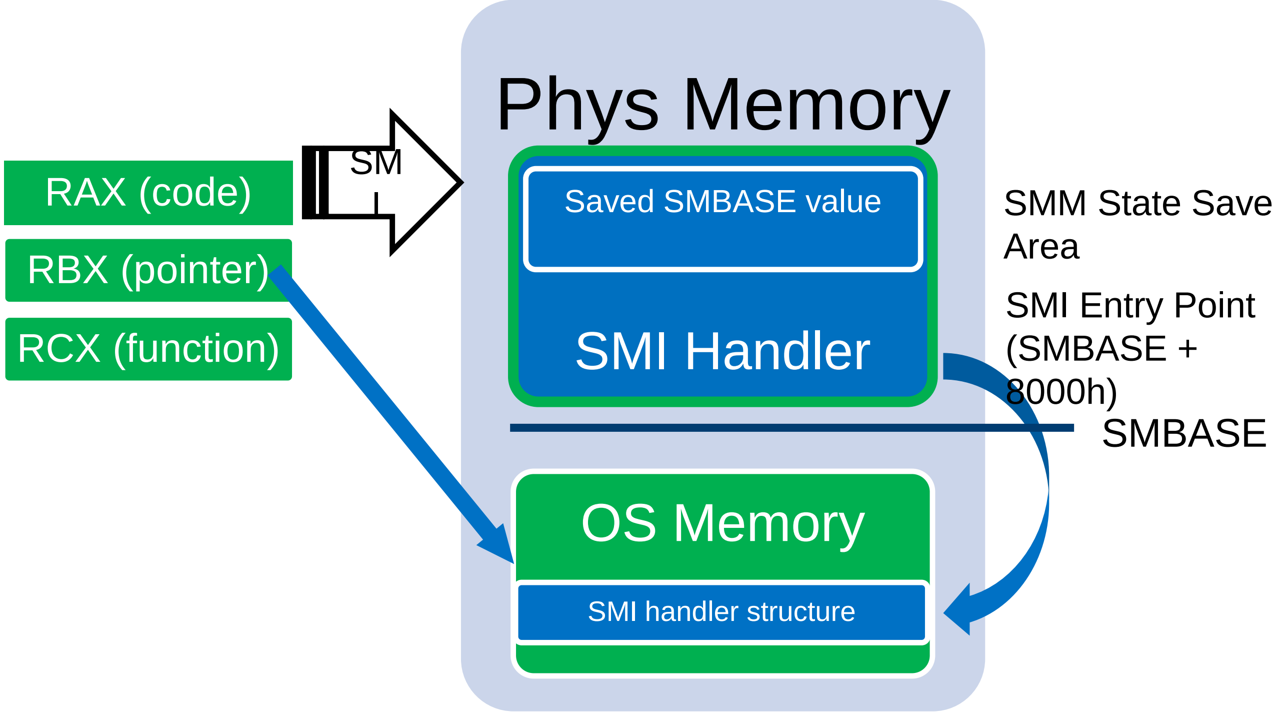
SMI Handler writes result to a buffer at address passed in RBX…

Pointer Vulnerabilities



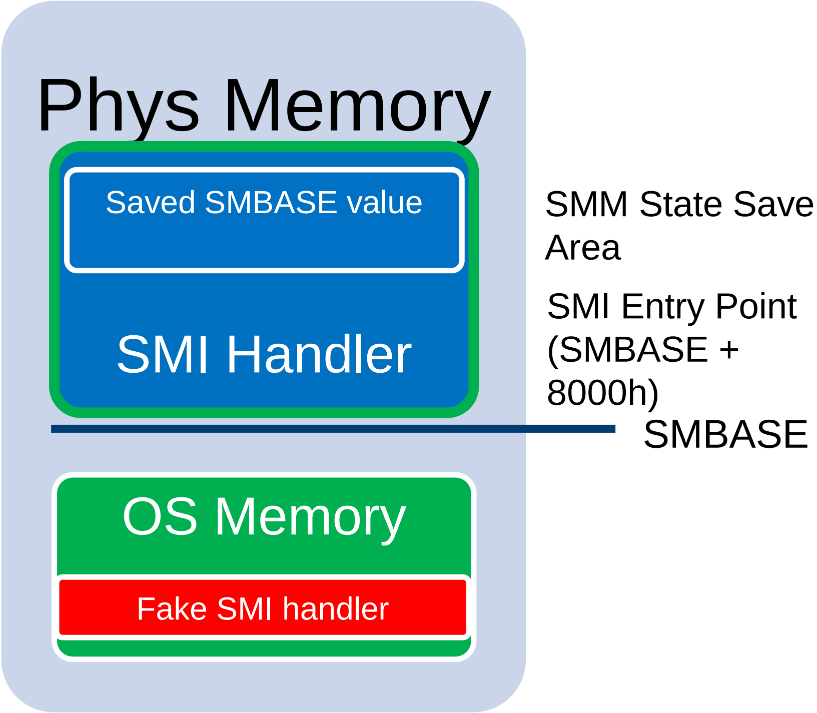
Exploit tricks SMI handler to write to an address **inside SMRAM**

How does the attack work?



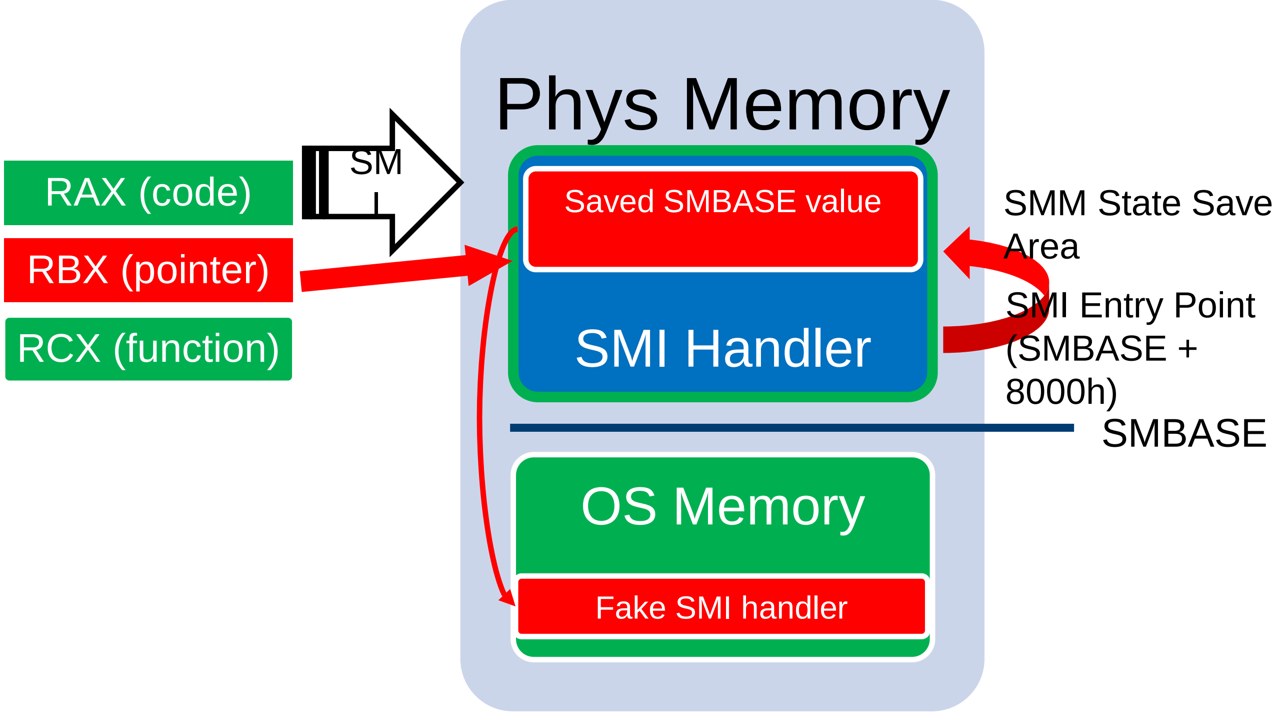
- CPU stores current value of SMBASE in SMM save state area on SMI and restores it on RSM

How does the attack work?



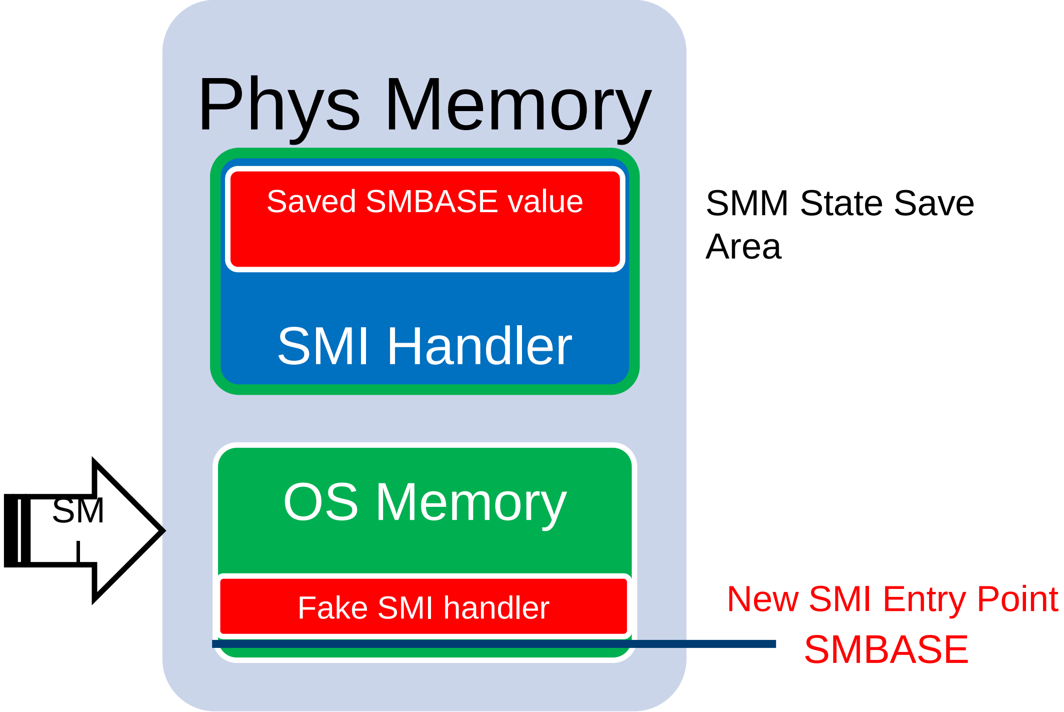
- Exploit prepares fake SMRAM with fake SMI handler outside of SMRAM

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?



Phys Memory

Saved SMBASE value

SMI Handler

SMM State Save Area

SM
I

OS Memory

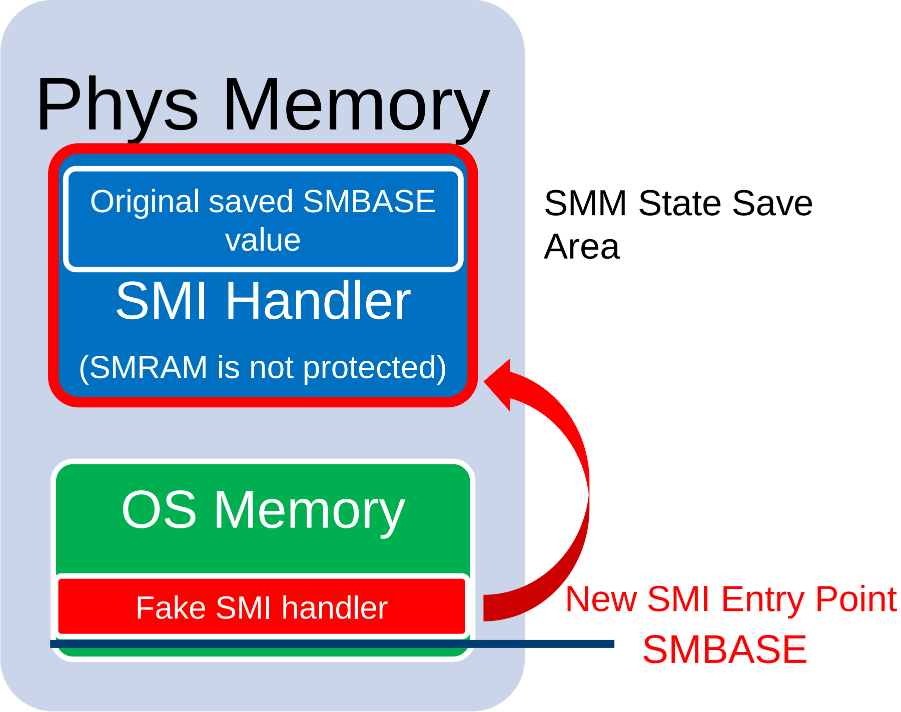
Fake SMI handler

New SMI Entry Point

SMBASE

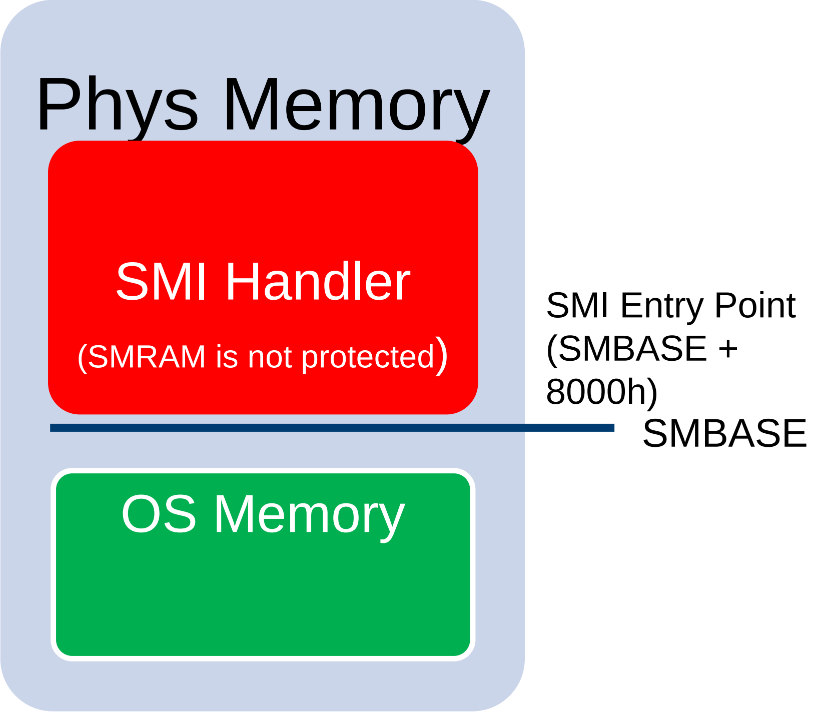
- Exploit triggers another SMI
- CPU executes fake SMI handler at new entry point outside of original protected SMRAM because SMBASE location changed

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

Input Pointers in EDKII: *CommBuffer*

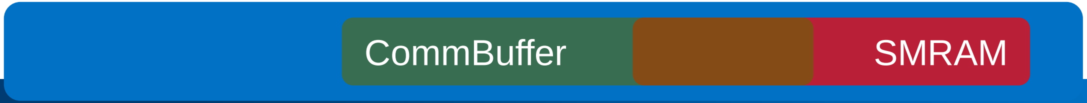
- **`CommBuffer`** is a memory buffer used as a communication protocol between OS runtime and DXE SMI handlers
- Pointer to **`CommBuffer`** is stored in “UEFI” ACPI table in ACPI NVS memory accessible to OS
- Contents of **`CommBuffer`** are specific to SMI handler. Variable SMI handler read UEFI variable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

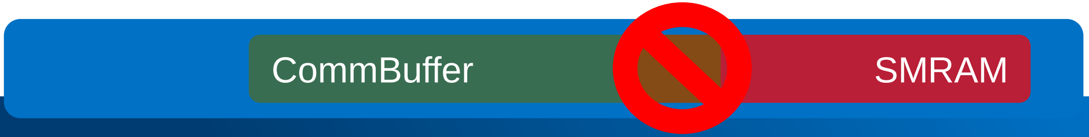
```
SmmVariableHandler (
...
  SmmVariableFunctionHeader = (SMM_VARIABLE_COMMUNICATE_HEADER *)CommBuffer;
  switch (SmmVariableFunctionHeader->Function) {
    case SMM_VARIABLE_FUNCTION_GET_VARIABLE:
      SmmVariableHeader = (SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE *)
                          SmmVariableFunctionHeader->Data;
      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

Detecting Bad SMI Input Pointers

1. Allocate a buffer in physical memory
2. Set GP registers to address of allocated memory buffer
3. Invoke SW SMI
4. Check fill pattern

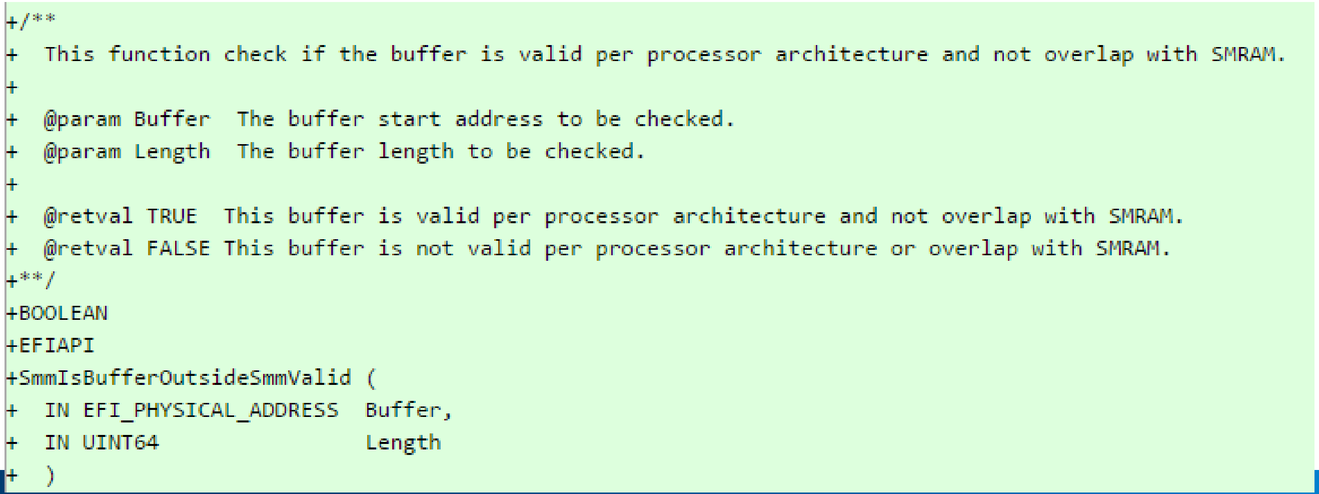
```
[x][ ======================================================================
[x][ Module: A tool to test SMI handlers for pointer validation vulnerabilies
[x][ ======================================================================
Usage: chipsec_main -m tools.smm.smm_ptr [ -a <fill_byte>,<size>,<config_file>,<address> ]
```

Detecting Bad SMI Input Pointers

```
[*] Configuration:
    Byte to fill with             : 0x11
    No of bytes to fill            : 0x500
    SMI config file                : chipsec/modules/tools/smm/smm_config.ini
    Default value of GP registers : 0x5A5A5A5A5A5A5A5A
    Allocated physmem buffer      : 0x0000000071A20800 (passed in GP reg to SMI)
    Second order buffer mode      : OFF

[*] Fuzzing SMI handlers defined in 'chipsec/modules/tools/smm/smm_config.ini'..
[*] Filling in 1280 bytes at PA 0x0000000071A20800 with ''..
[*] Sending SMI# 0x5A (data = 0x5A) SW_SMI_Name (swsmi_desc)..
    RAX: 0x0000000071A20800 (AX will be overwritten with values of SW SMI ports B2/B3)
    RBX: 0x0000000071A20800
    RCX: 0x0000000071A20800
    RDX: 0x0000000071A20800 (DX will be overwritten with 0x00B2)
    RSI: 0x0000000071A20800
    RDI: 0x0000000071A20800
Checking contents at PA 0x0000000071A20800..
[+] Contents at PA 0x0000000071A20800 have not changed
```

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
+  )
```