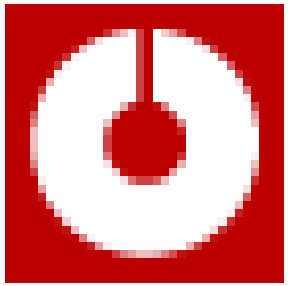


CanSecWest 2015 Vancouver, Canada

A New Class of Vulnerabilities in SMI Handlers

Advanced Threat Research (www.intelsecurity.com/atr)

Oleksandr Bazhaniuk, Yuriy Bulygin, **Andrew Furtak**, Mikhail Gorobets, **John Loucaides**, Alexander Matrosov, Mickey Shkatov

Agenda

- Background on SMM and SMI handlers
- Known SMM vulnerabilities
- New SMI handler vulnerabilities
- Demo
- Similar Issues & Mitigations

Background on SMM and SMI Handlers

System Management Mode (SMM)

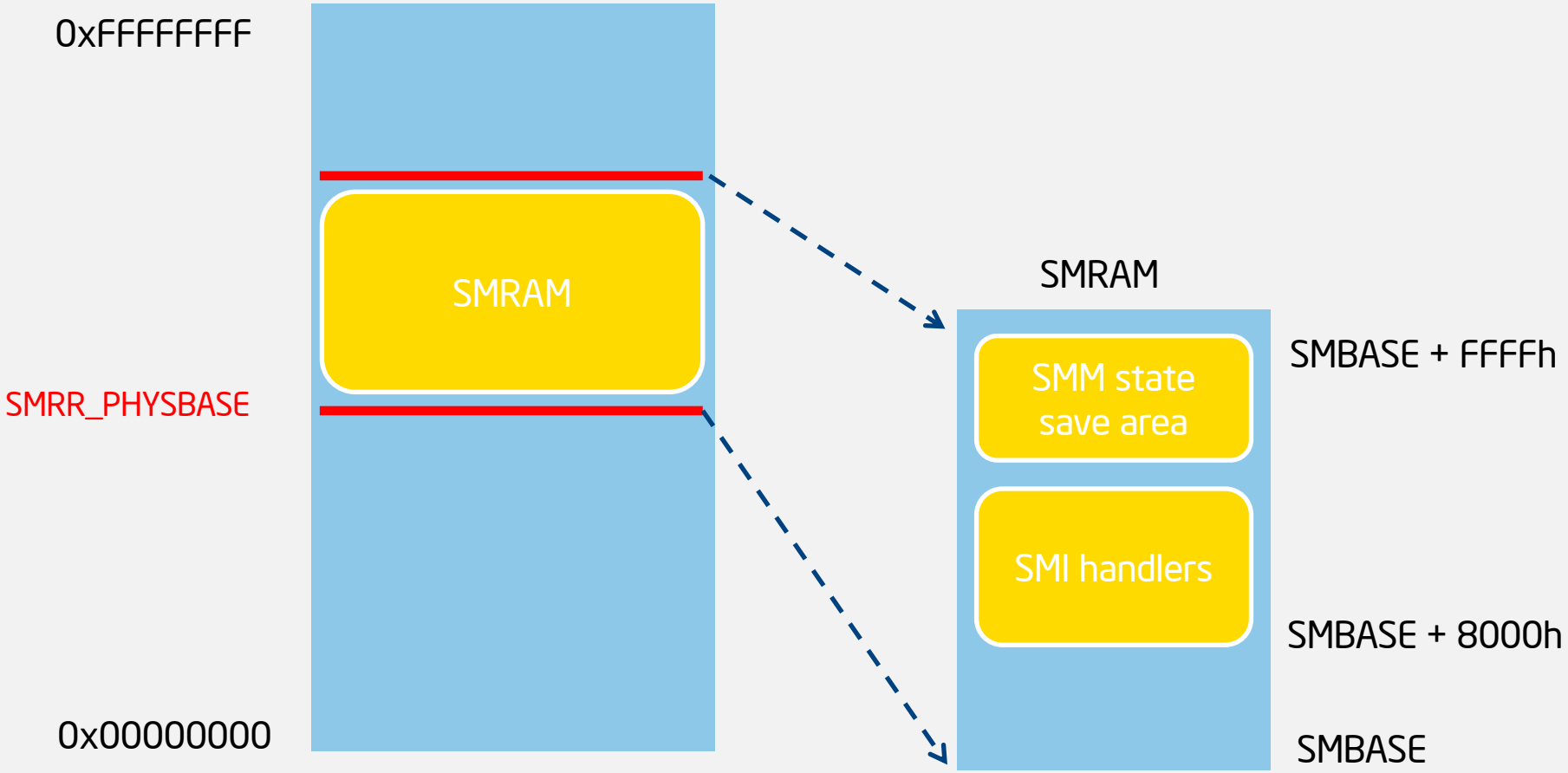
System Management Interrupt (SMI)

- CPU (OS) state is saved in SMRAM upon entry to SMM and restored upon exit from SMM
- SMI handlers are invoked by CPU upon System Management Interrupt (SMI) from chipset or other logical CPUs and execute in System Management Mode (SMM) of x86 CPU
- SMI handlers return to the OS using RSM instruction

SMRAM is a range of DRAM reserved by BIOS SMI handlers

- Protected from software and device acces

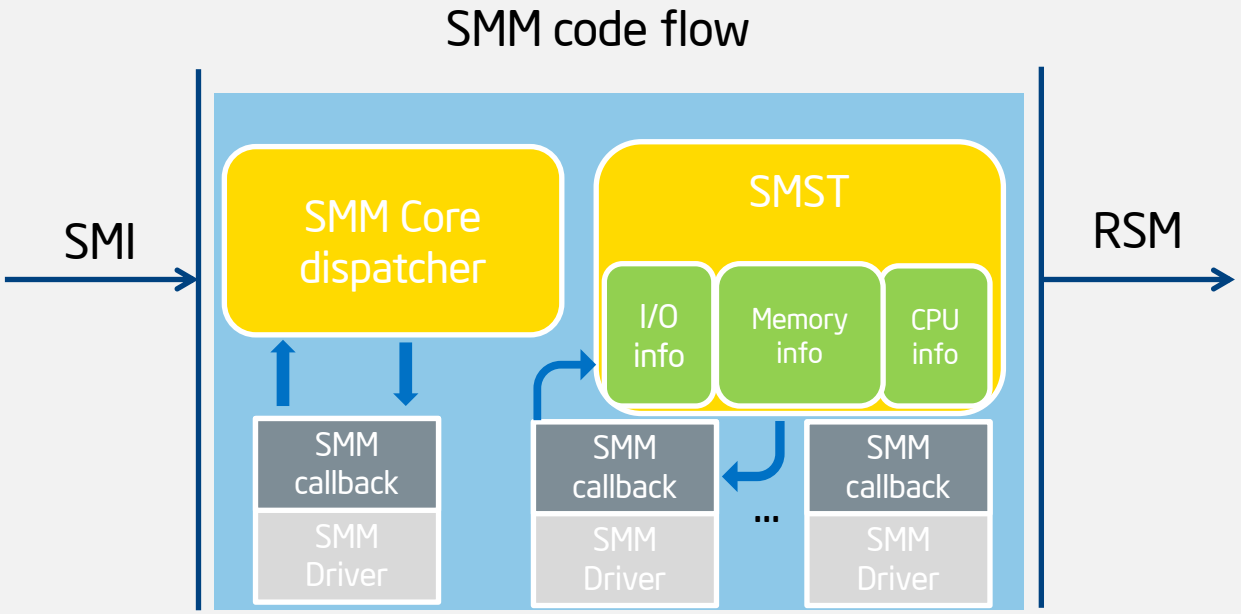
System Management Interrupt (SMI) Handlers



System Management Interrupt (SMI) Handlers

SMM handler execution environment:

- At entry, CS = SMBASE (0x30000 at reset), EIP=0x8000
- Addressable physical space from 0 to 0xFFFFFFFFh (4G)
- No paging
- All hardware interrupts are disabled



Trigger 'SW' SMI via APMC I/O Port

```
_swsmi PROC
    ..

    ; setting up GPR (arguments) to SMI handler call
    ; AX gets overwritten by _smi_code_data (which is passed in RCX)
    ; DX gets overwritten by the value of APMC port (= 0x00B2)
    mov rax, rdx ; rax_value
    mov ax, cx   ; smi_code_data
    mov rdx, r10 ; rdx_value
    mov dx, 0B2h ; APMC SMI control port 0xB2

    mov rbx, r8  ; rbx_value
    mov rcx, r9  ; rcx_value
    mov rsi, r11 ; rsi_value
    mov rdi, r12 ; rdi_value

    ; write smi_code_data value to SW SMI control/data ports (0xB2/0xB3)
    out dx, ax

    ..
    ret

_swsmi ENDP
```

Trigger 'SW' SMI using CHIPSEC

From command line:

```
# chipsec_util.py smi smic smid [RAX RBX RCX RDX RSI RDI]
```

From any module in CHIPSEC:

```
self.intr = chipsec.hal.Interrupts( self.cs )
self.intr.send_SW_SMI(smic,smid,rax,rbx,rcx,rdx,rsi,rdi)
```

Known SMI Vulnerabilities

Known Vulnerabilities Related to SMM

Unlocked Compatible/Legacy SMRAM

- **Issue**
  - When D_LCK is not set by BIOS, SMM space decode can be changed to open access to CSEG when CPU is not in SMM: Using CPU SMM to Circumvent OS Security Functions
  - Also Using SMM For Other Purposes
- **Mitigation**
  - D_LCK bit locks down Compatible SMM space (a.k.a. CSEG) configuration (SMRAMC)
  - SMRAMC[D_OPEN]=0 forces access to legacy SMM space decode to system bus rather than to DRAM where SMI handlers are when CPU is not in System Management Mode (SMM)
- **Check**
  - `chipsec_main --module common.smm`

Known Vulnerabilities Related to SMM

SMRAM "Cache Poisoning" Attacks

- **Issue**
  - CPU executes from cache if memory type is cacheable
  - Ring0 exploit can make SMRAM cacheable (variable MTRR)
  - Ring0 exploit can then populate cache-lines at SMBASE with SMI exploit code (ex. modify SMBASE) and trigger SMI
  - CPU upon entering SMM will execute SMI exploit from cache
  - Attacking SMM Memory via Intel Cache Poisoning
  - Getting Into the SMRAM: SMM Reloaded
- **Mitigation**
  - CPU System Management Range Registers (SMRR) forcing UC and blocking access to SMRAM when CPU is not in SMM
- **Check**
  - `chipsec_main --module common.smrr`

Legacy SMI Handlers Calling Out of SMRAM

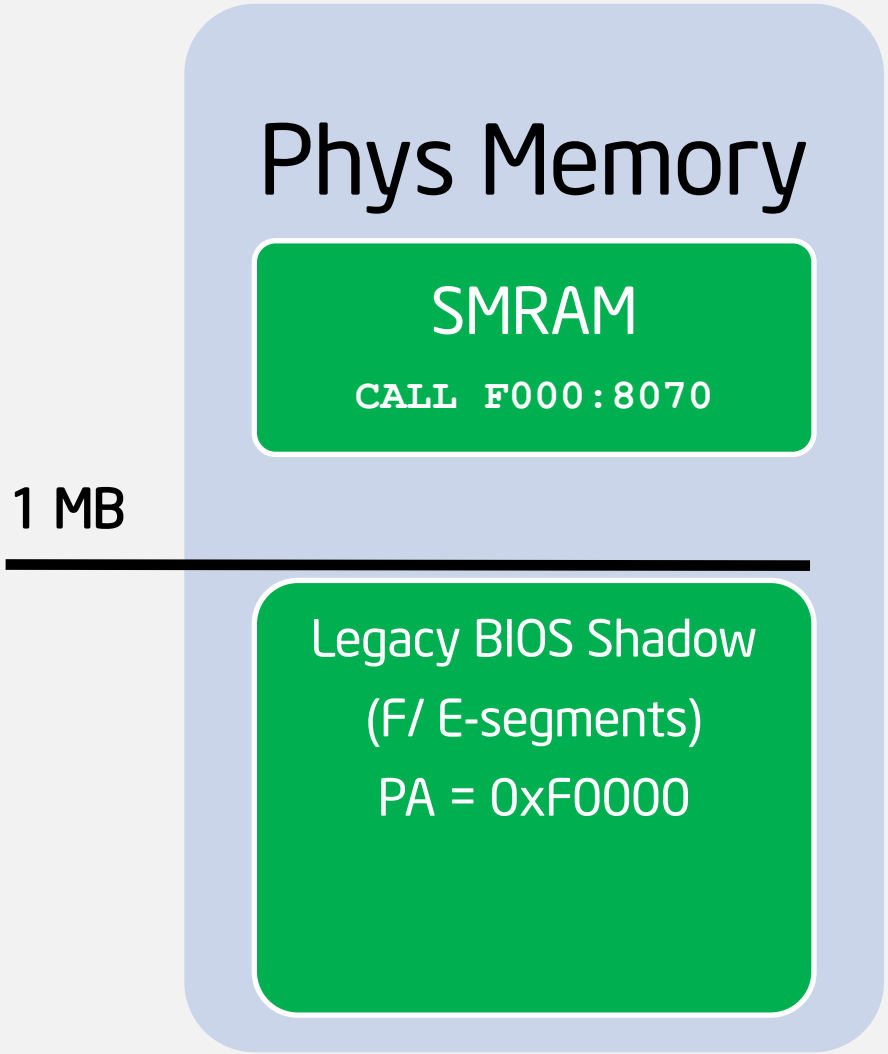
Branch Outside of SMRAM

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

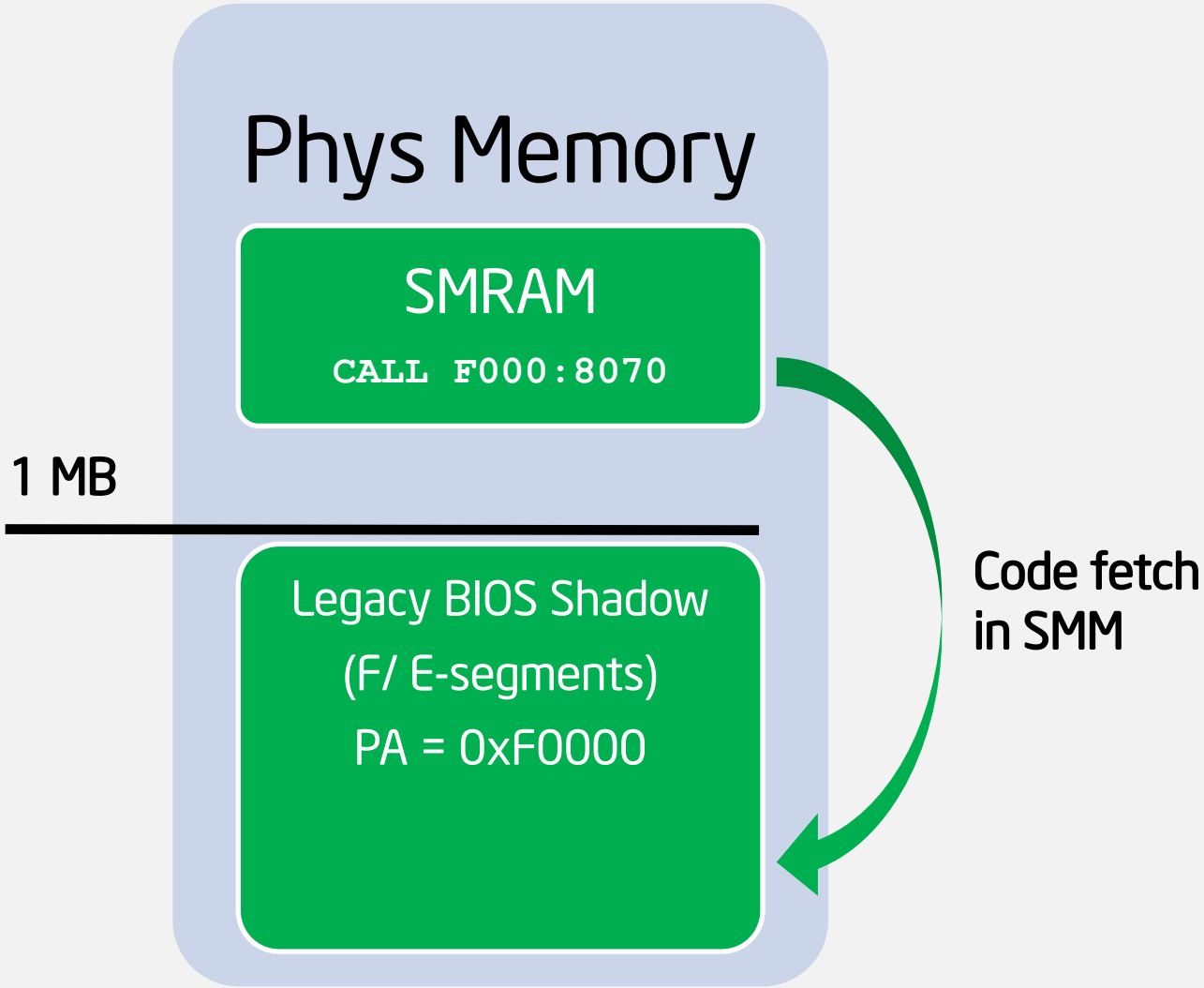
```
Disassembly of the code of $SMISS handler, one of SMI handlers in
the BIOS firmware in ASUS Eee PC 1000HE system.

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
```

- BIOS SMM Privilege Escalation Vulnerabilities (14 issues in just one SMI Handler)
- System Management Mode Design and Security Issues

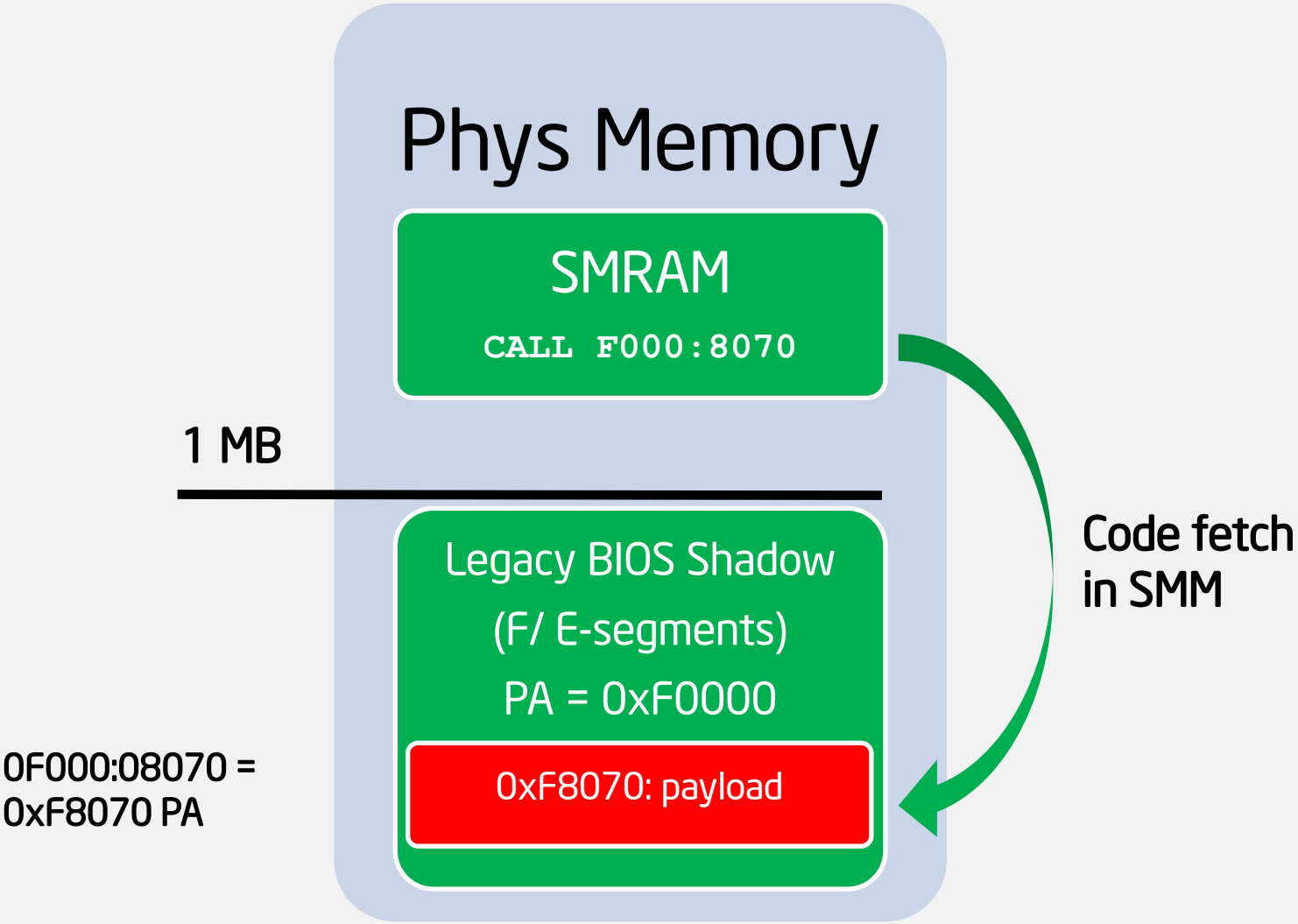
Legacy SMI Handlers Calling Out of SMRAM



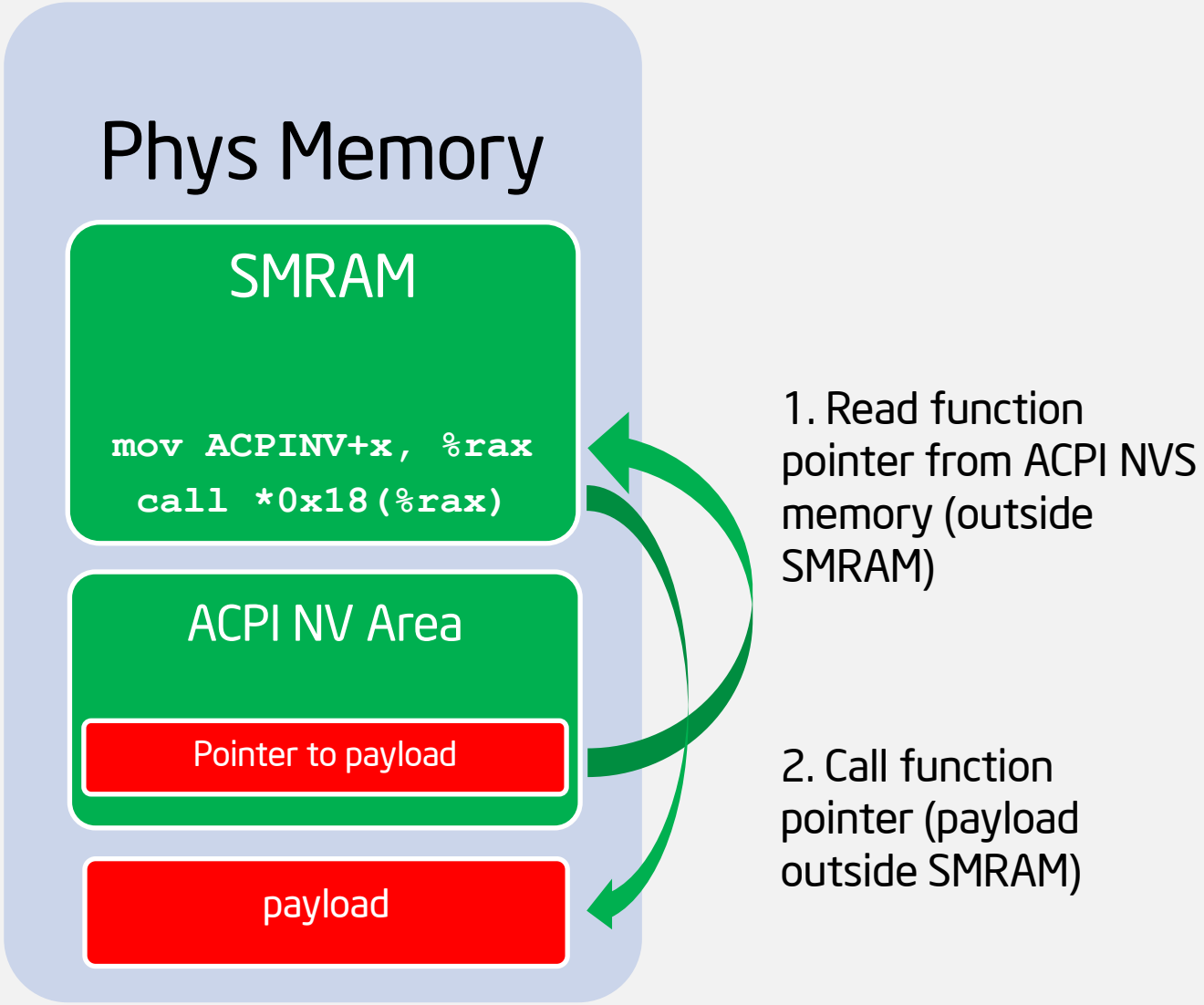
Legacy SMI Handlers Calling Out of SMRAM



Legacy SMI Handlers Calling Out of SMRAM



Function Pointers Outside of SMRAM (DXE SMI)



Mitigating SMM "Call-Outs"

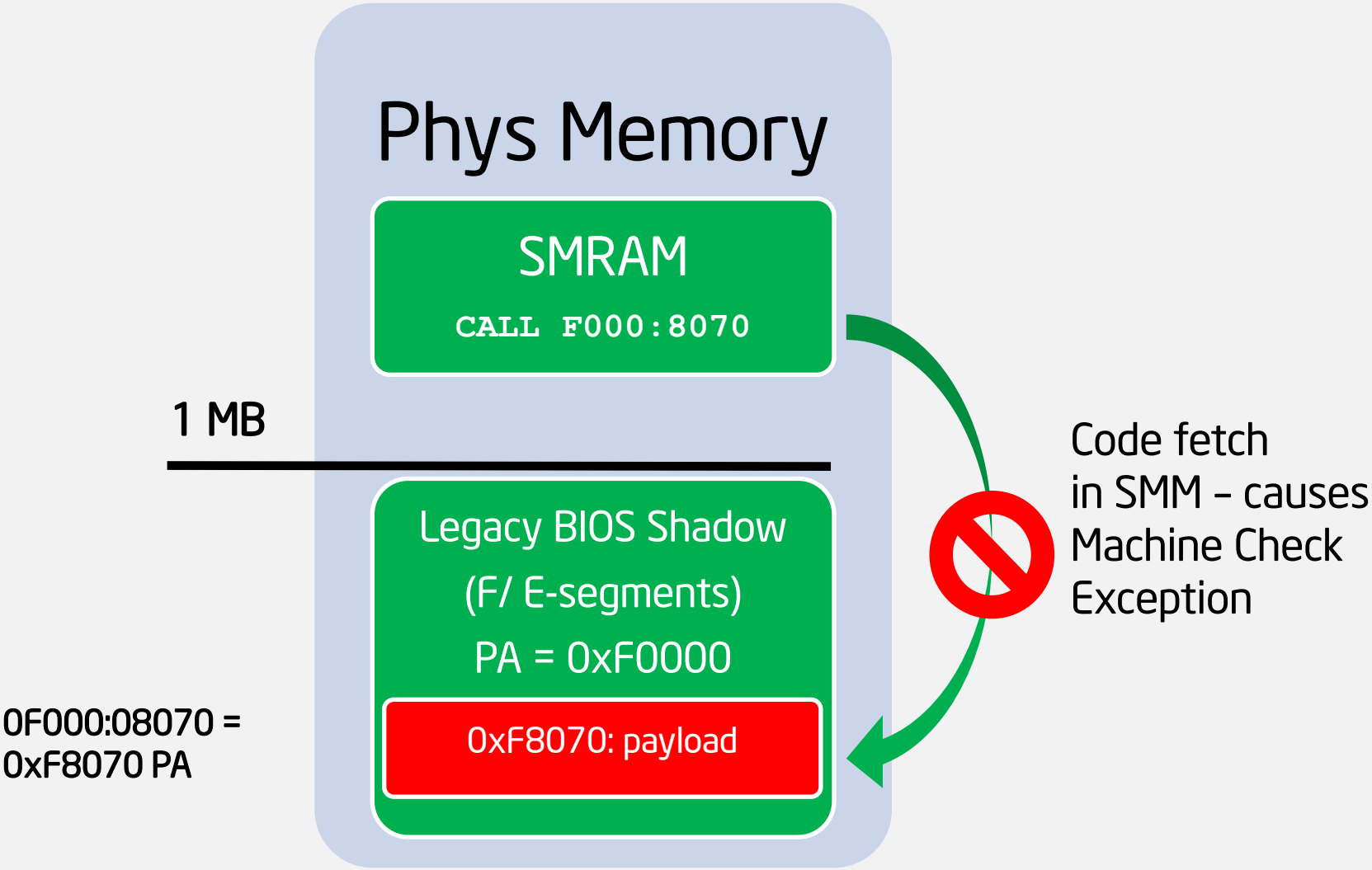
These issues are specific to particular SMI Handler implementation...

- static analysis of binary?
- run-time debugging?

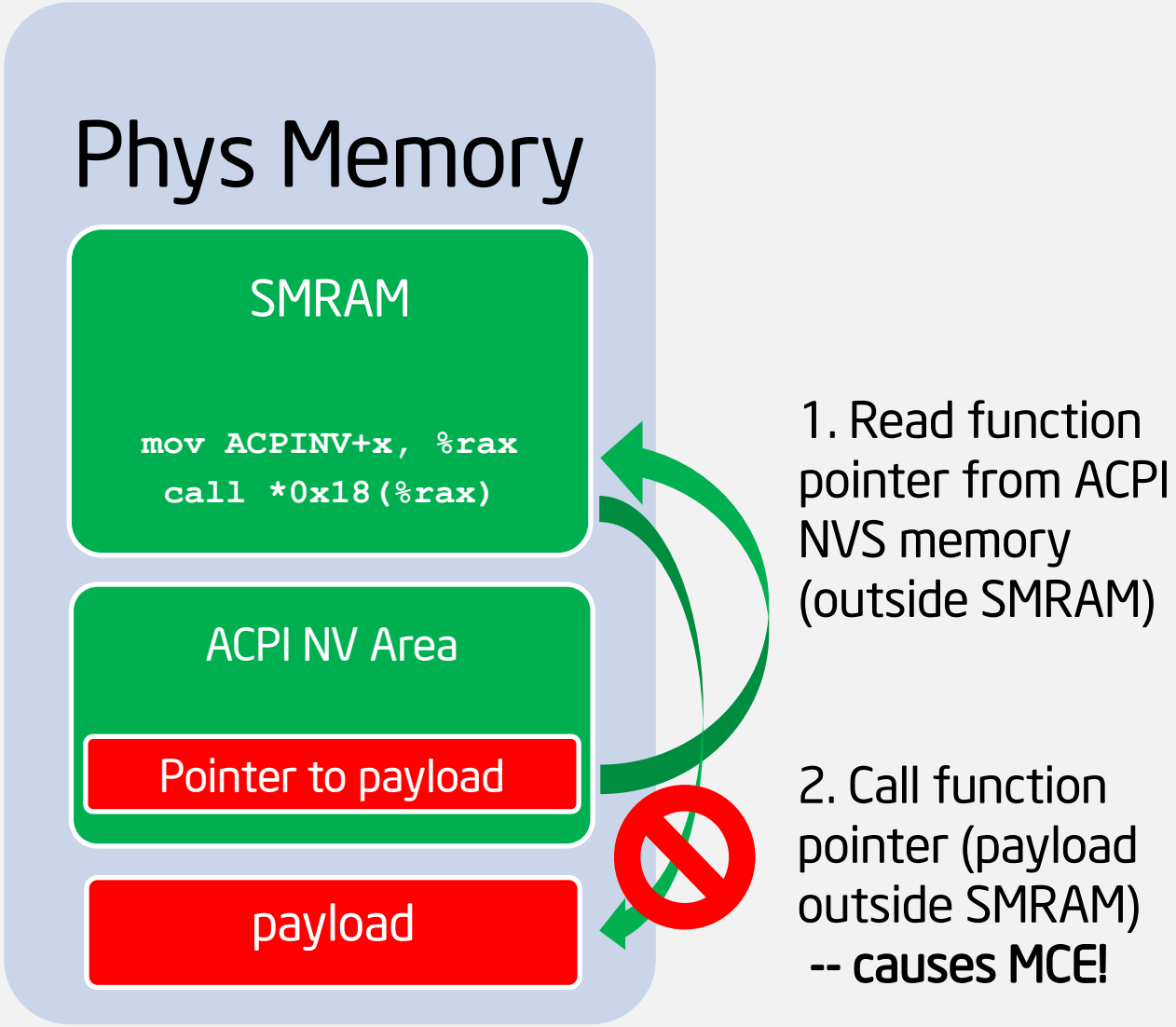
CPU Hardware Support (starting in Haswell)

- SMM Code Access Check
  - SMM_Code_Chk_En (SMM-RW)
    - » This control bit is available onlyif MSR_SMM_MCA_CAP[58] == 1.
    - » When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR.
    - » When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
- Attempts to execute code outside SMRAM while inside SMM result in a Machine Check Exception!

Legacy SMI Handlers Calling Out of SMRAM

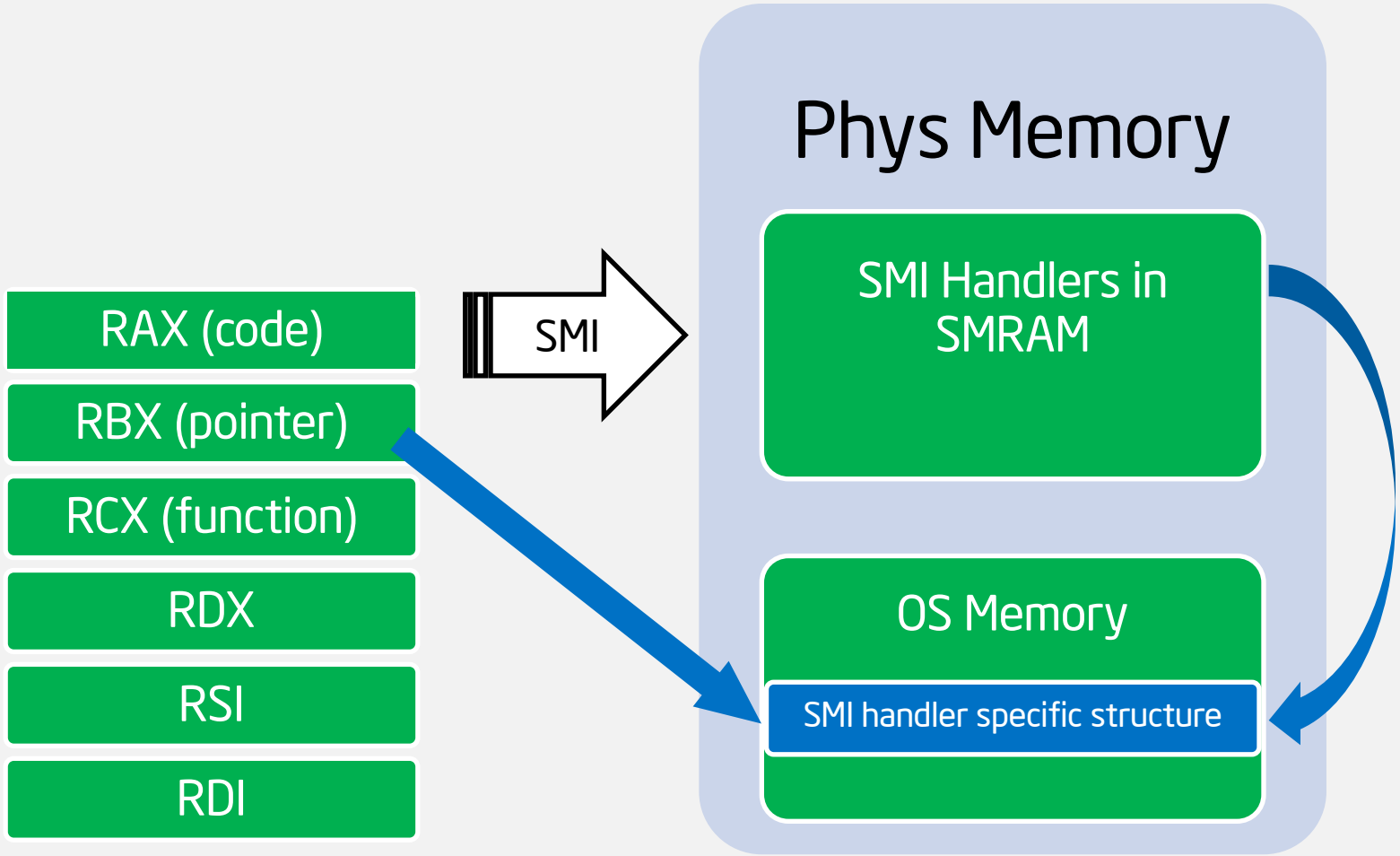


Function Pointers Outside of SMRAM (DXE SMI)



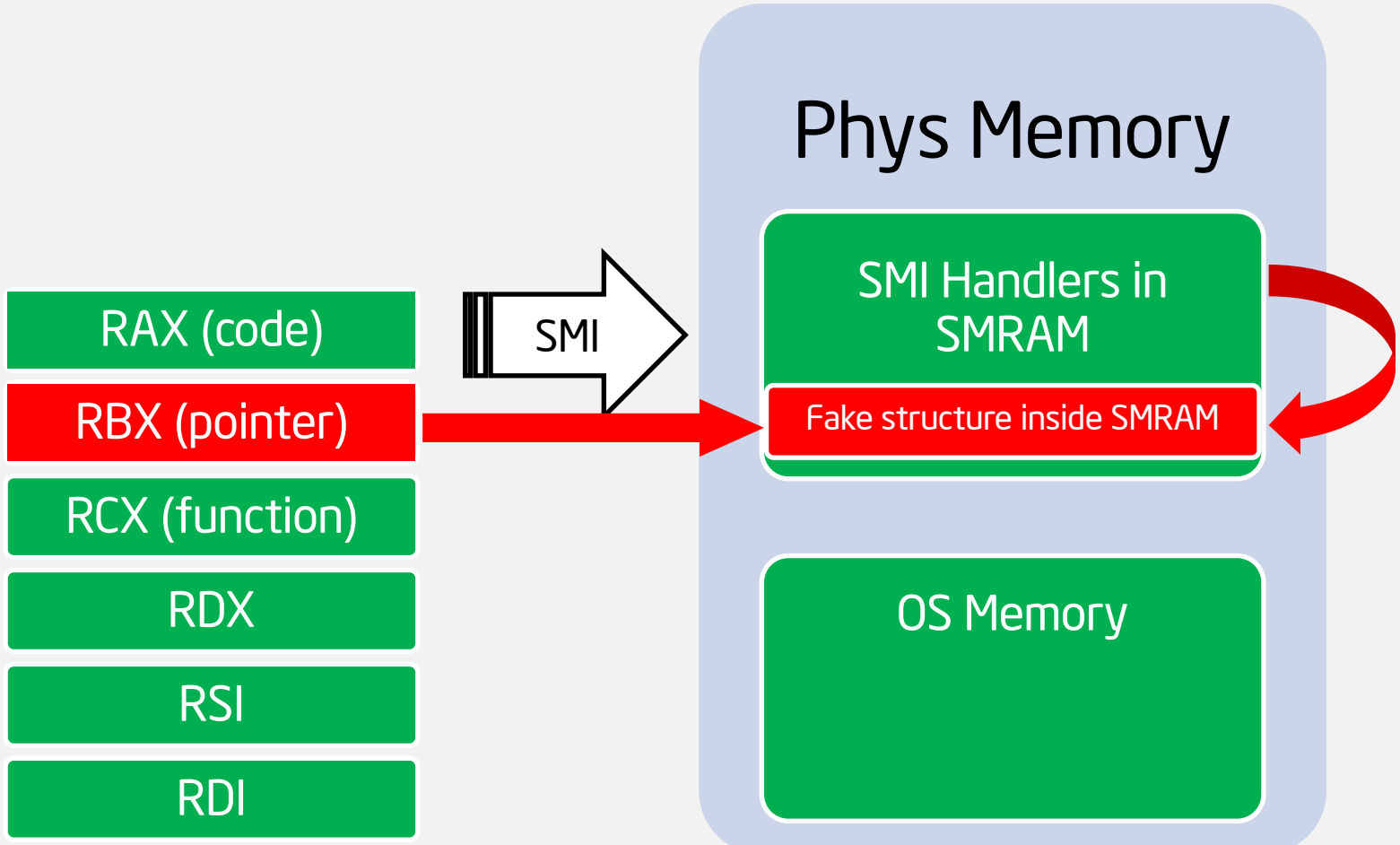
A new class of vulnerabilities in SMI handlers

Pointer Arguments to SMI Handlers



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

Pointer Vulnerabilities



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

What to overwrite inside SMRAM?

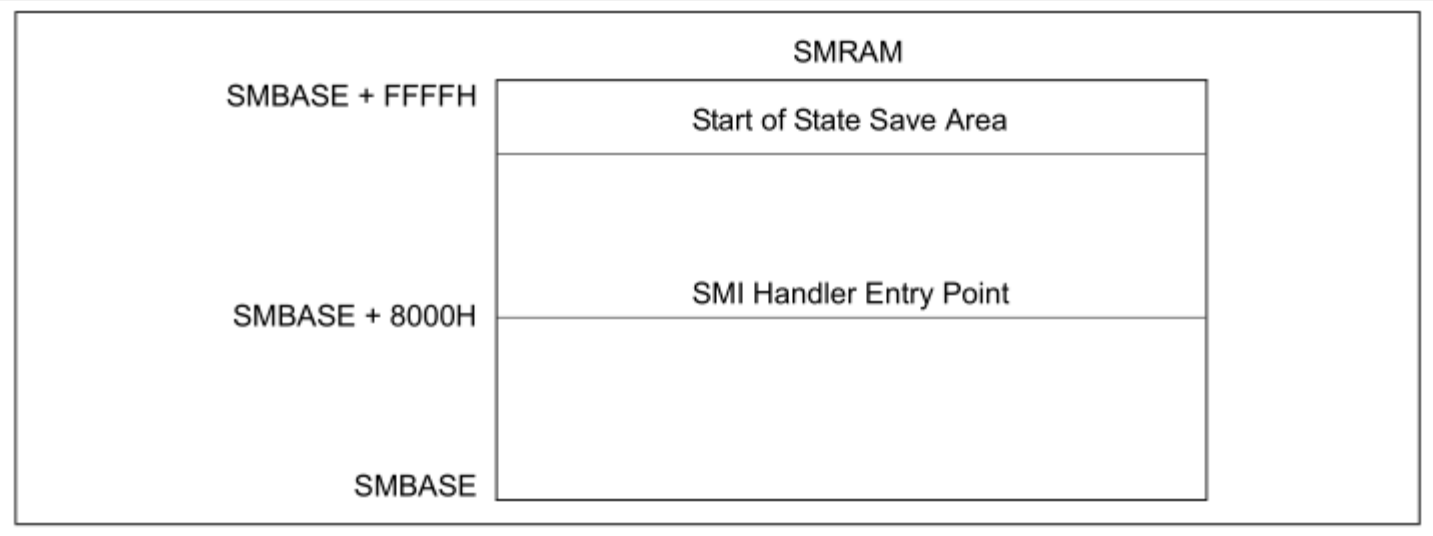
- Depending on the vulnerability, caller may control address to write, the value written, or both.
- Often the caller controls the address but doesn't have control over the values written to the address by the SMI handler
- In our example the attacker controls the address and does not control the value:

  The SMI handler writes 0 value to the specified address

What to overwrite inside SMRAM?

What can an exploit overwrite in SMRAM without crashing?

- 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



What to overwrite inside SMRAM?

- Current value of SMBASE MSR is also saved in SMM state save area by CPU at offset SMBASE + FEF8h upon SMI
- Stored value of SMBASE is restored upon executing RSM
- **SMBASE relocation:** SMI handler may change the saved value of SMBASE in order to change the location of SMRAM upon next SMI

- The idea:

  Move SMBASE to a new, unprotected location by changing the SMBASE value stored in the SMM state save area.

How do we know where to write?

SMM state save is at SMBASE + FC00h

But SMBASE is not known

- SMBASE MSR can be read only in SMM
- SMBASE is different per CPU thread
- SMBASE is different per BIOS implementation

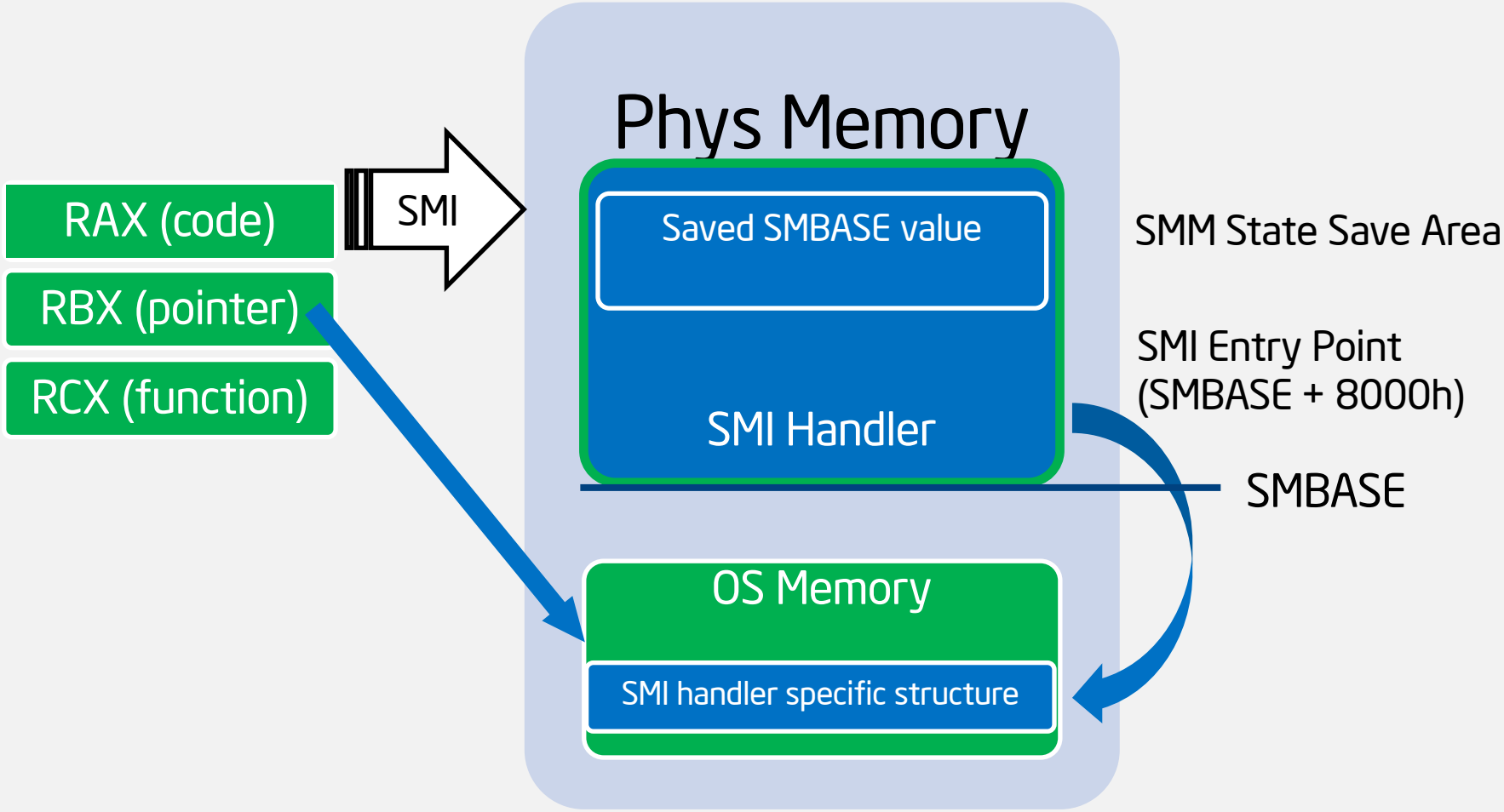
SMBASE should be programmed at some offset from TSEG/SMRR_PHYSBASE

How do we know where to write?

1. Dump contents of SMRAM
   - Use hardware debugger
   - Use similar "pointer read" vulnerability
   - Use another vulnerability (e.g. S3 boot script) to disable SMRAM DMA protection and use DMA via graphics aperture to read SMRAM

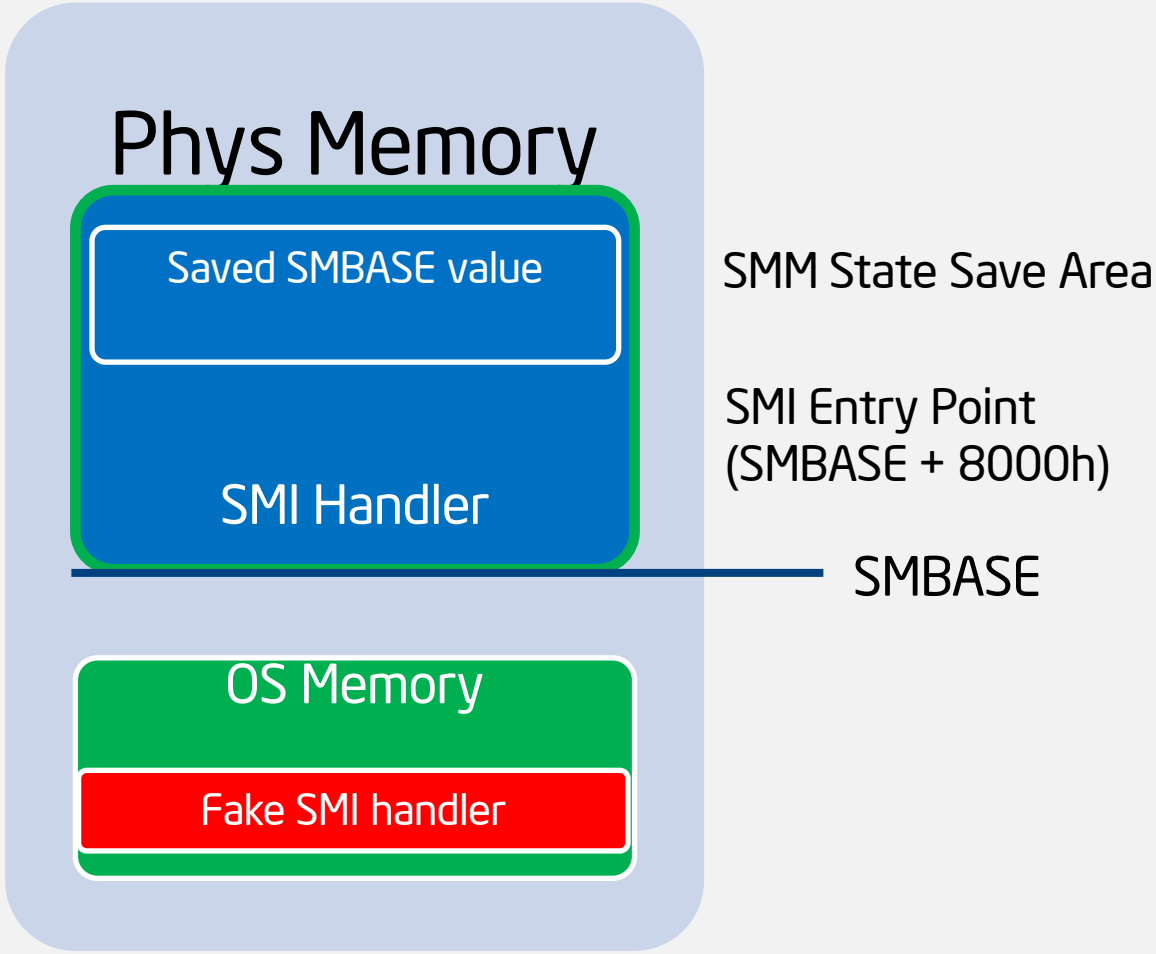
   And in the dump look for
   - SMI handler entry point code
   - Known values in GP registers (invoke SMI with these values before dump)
2. Read SPI flash memory & RE initialization code to find SMBASE there.
3. Guess location of SMBASE:
   - SMBASE = SMRR_PHYSBASE
   - SMBASE = SMRR_PHYSBASE - 8000h (SMRR_PHYSBASE at SMI handler location)
   - Blind iteration through all offsets within SMRAM as potential saved SMBASE value

How does the attack work?



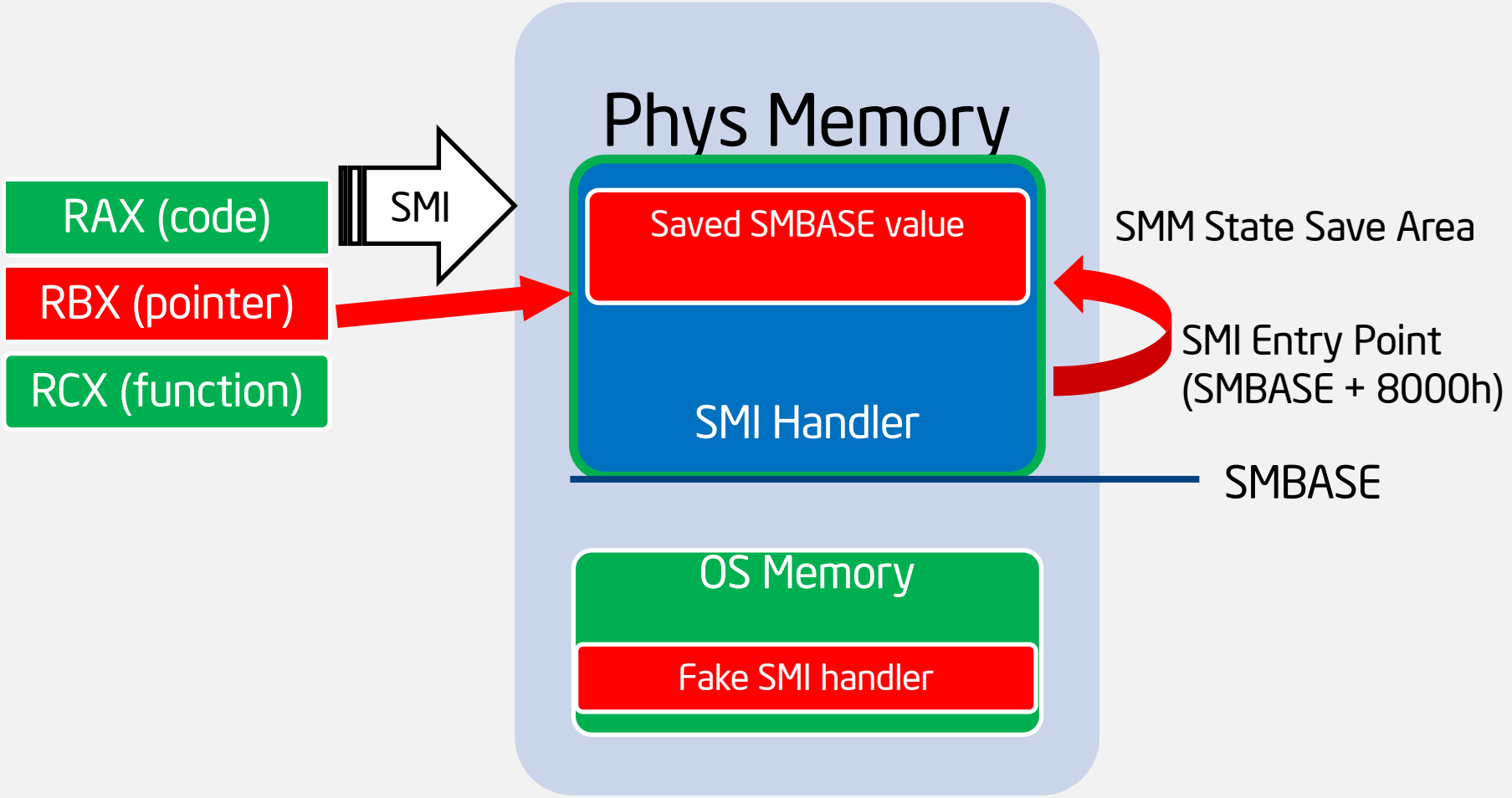
- CPU stores current value of SMBASE in SMM save state area on SMI and restores it on RSM

How does the attack work?



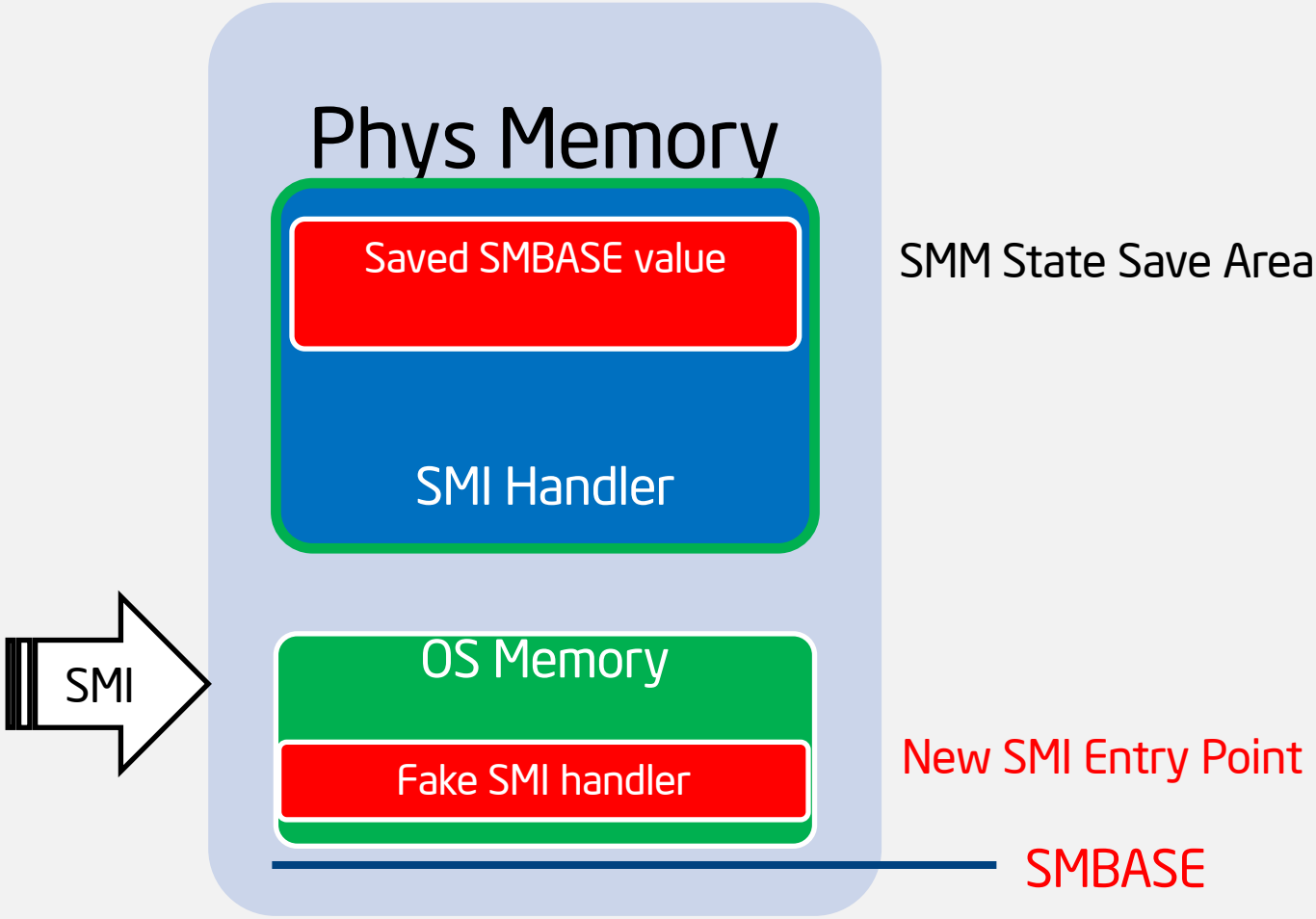
- Attacker prepares fake SMRAM with new SMI handler outside of SMRAM at some address (PA 0) that will be a new SMBASE

How does the attack work?



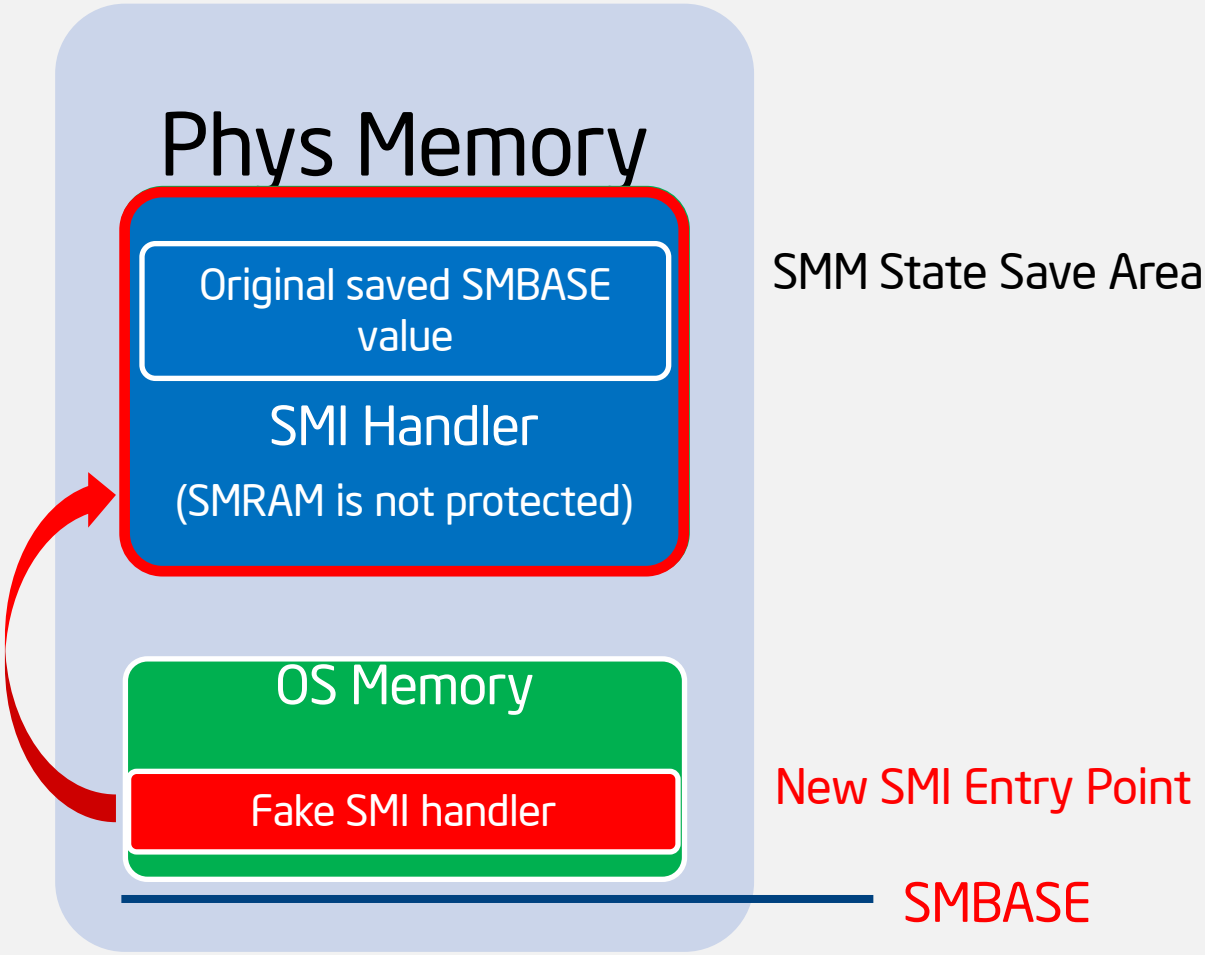
- Attacker triggers SMI w/ RBX pointing to saved SMBASE address in SMRAM
- SMI handler overwrites saved SMBASE on exploit's behalf with address of fake SMBASE (e.g. 0 PA)

How does the attack work?



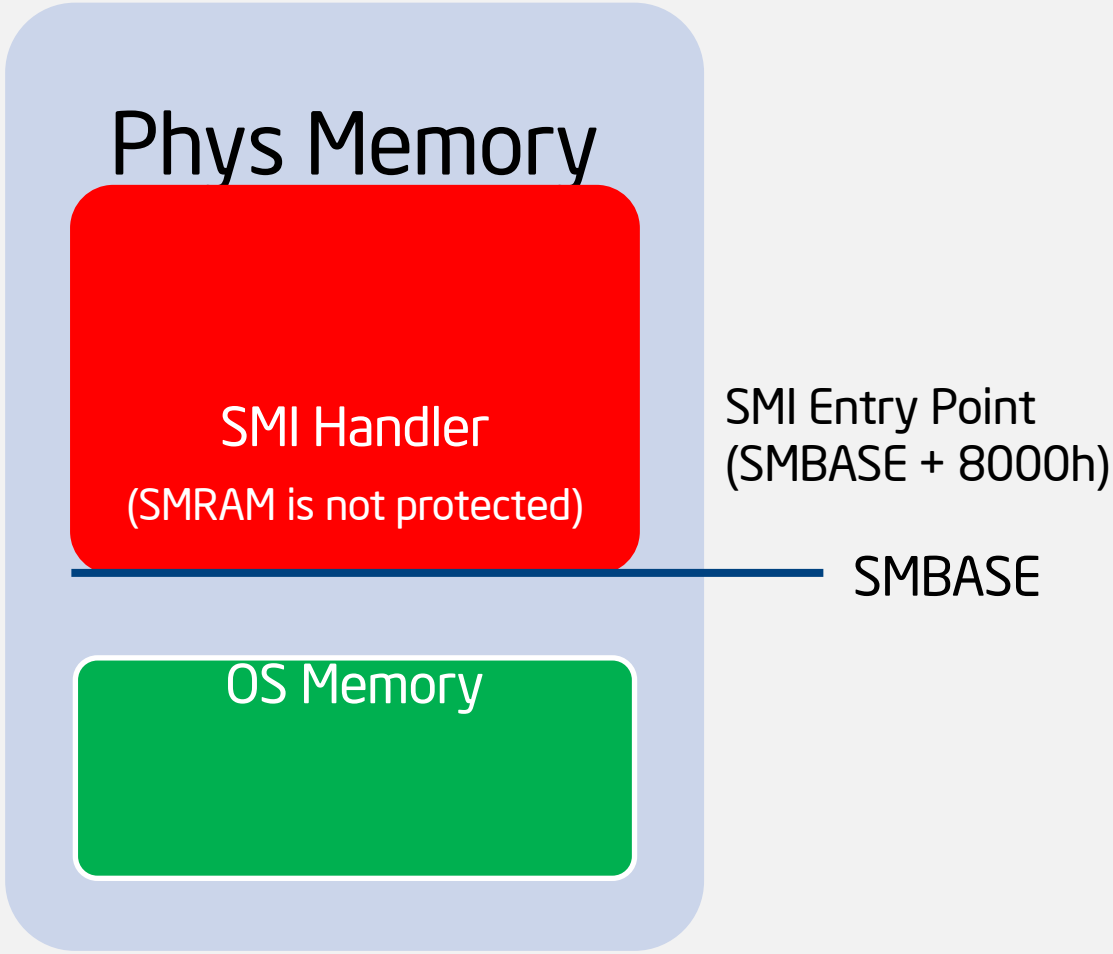
- Attacker triggers another SMI
- CPU executes fake SMI handler at new entry point outside of original protected SMRAM because SMBASE location changed

How does the attack work?



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

How does the attack work?



- The SMRAM is restored but not protected by HW anymore
- The SMI handler may be modified by attacker as well as other SMRAM data

Demo

Similar Vulnerabilities & Mitigations

EDKII "CommBuffer" Pointer

- CommBuffer is a memory buffer
  - Facilitates communication between OS runtime and SMI handlers
  - Pointer to CommBuffer is stored in "UEFI" ACPI table in ACPI NVS memory accessible to OS kernel code
- Contents of CommBuffer are specific to SMI handler.
  - For example, when calling the variable SMI handler, CommBuffer contains
    - UEFI variable Name, GUID, and Data
- Example 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 Issues in "CommBuffer"

- Addressed via Tianocore Advisory Logs
- 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

Validating Input Pointers

- ***Read pointer*** issues are also exploitable to expose the contents of SMRAM
- 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
  - For instance, 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
+  )
```

So, how do you find issues like that?

1. Allocate memory and fill with pattern
2. Set registers to address of allocated memory
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> ]
```

CHIPSEC Output

[*] 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

References

1. CHIPSEC: https://github.com/chipsec/chipsec
2. Trianocore security advisories
3. UEFI Forum USRT (security@uefi.org , PGP key)
4. Intel PSIRT (secure@intel.com, PGP key)

Thank You!