Automated vulnerability hunting in SMM using Brick

Assaf Carlsbad, Itai Liba

SentinelOne™

carlsbad@sentinelone.com

@assaf_carlsbad

itail@sentinelone.com

@liba2k

Agenda

01 A whirlwind tour of SMM

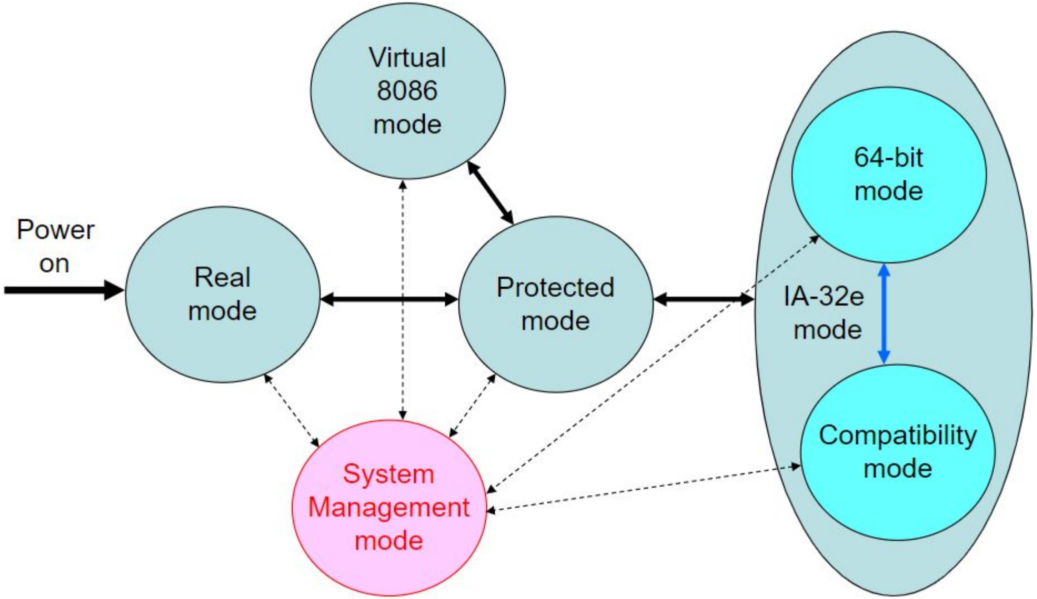
02 Summary of SMM bug classes and attacks

03 Automating bug hunting in SMM

A brief introduction to SMM

- System Management Mode

- A dedicated CPU mode for firmware handling low-level system-wide functions
  - Power management
  - Legacy device emulation
  - Proprietary OEM code

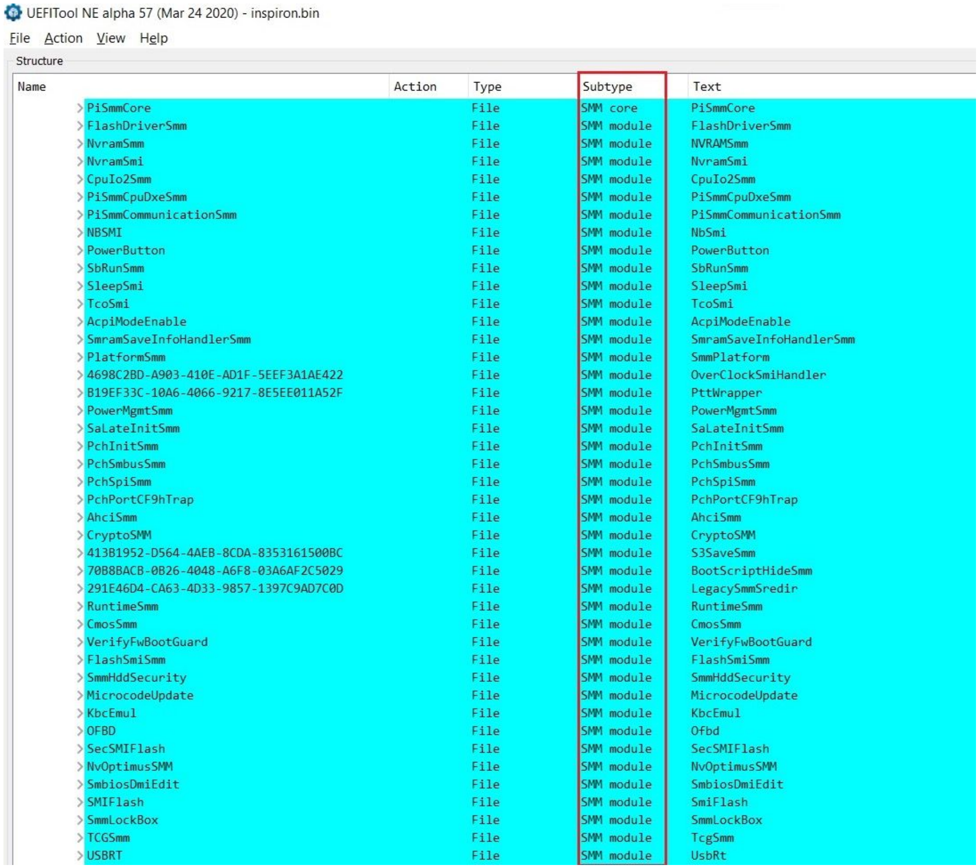


Venturing into the x86's System Management Mode

A brief introduction to SMM

- Originally introduced by the i386 CPU

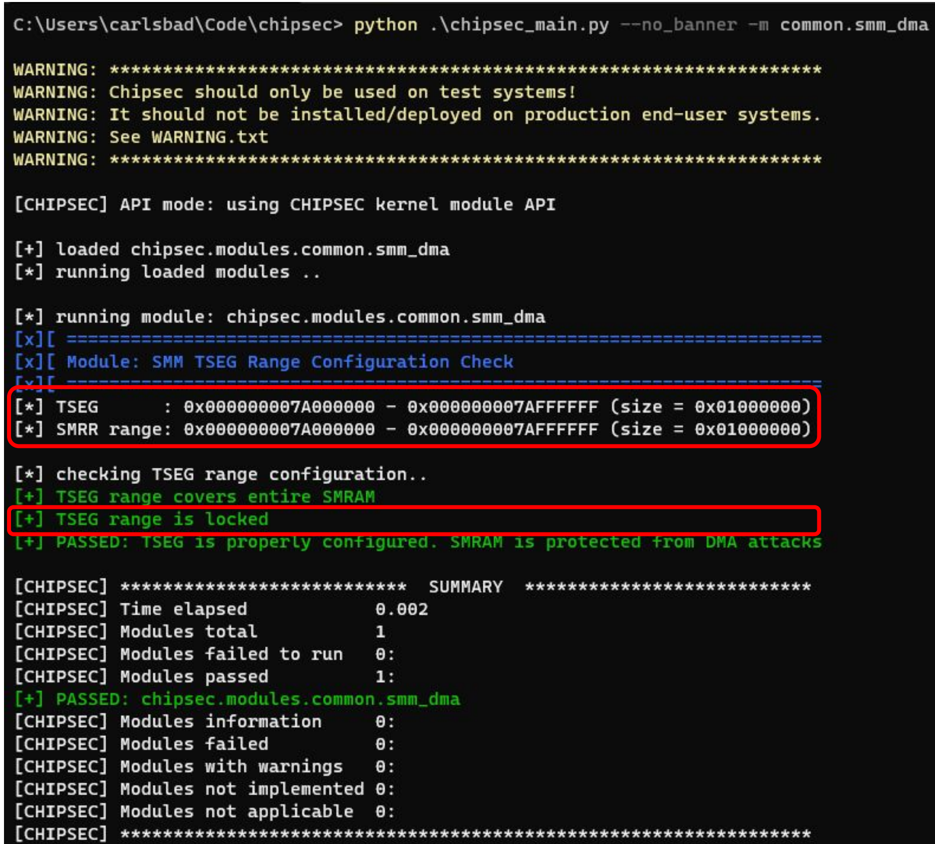
- Over the years, OEMs started shifting more and more functionality into it



Name	Action	Type	Subtype	Text
PiSmmCore		File	SMM core	PiSmmCore
FlashDriverSmm		File	SMM module	FlashDriverSmm
NvramSmm		File	SMM module	NVRAMSmm
NvramSmi		File	SMM module	NvramSmi
CpuIo2Smm		File	SMM module	CpuIo2Smm
PiSmmCpuDxeSmm		File	SMM module	PiSmmCpuDxeSmm
PiSmmCommunicationSmm		File	SMM module	PiSmmCommunicationSmm
NBSMI		File	SMM module	NbSmi
PowerButton		File	SMM module	PowerButton
SbRunSmm		File	SMM module	SbRunSmm
SleepSmi		File	SMM module	SleepSmi
TcoSmi		File	SMM module	TcoSmi
AcpiModeEnable		File	SMM module	AcpiModeEnable
SmramSaveInfoHandlerSmm		File	SMM module	SmramSaveInfoHandlerSmm
PlatformSmm		File	SMM module	SmmPlatform
4698C2BD-A903-410E-AD1F-5EEF3A1AE422		File	SMM module	OverClockSmiHandler
B19EF33C-10A6-4066-9217-8E5EE011A52F		File	SMM module	PttWrapper
PowerMgmtSmm		File	SMM module	PowerMgmtSmm
SaLateInitSmm		File	SMM module	SaLateInitSmm
PchInitSmm		File	SMM module	PchInitSmm
PchSmbusSmm		File	SMM module	PchSmbusSmm
PchSpiSmm		File	SMM module	PchSpiSmm
PchPortCF9hTrap		File	SMM module	PchPortCF9hTrap
AhciSmm		File	SMM module	AhciSmm
CryptoSMM		File	SMM module	CryptoSMM
413B1952-D564-4AEB-8CDA-8353161500BC		File	SMM module	S3SaveSmm
70B8BACB-0B26-4048-A6F8-03A6AF2C5029		File	SMM module	BootScriptHideSmm
291E46D4-CA63-4D33-9857-1397C9AD7C0D		File	SMM module	LegacySmmSredir
RuntimeSmm		File	SMM module	RuntimeSmm
CmosSmm		File	SMM module	CmosSmm
VerifyFwBootGuard		File	SMM module	VerifyFwBootGuard
FlashSmiSmm		File	SMM module	FlashSmiSmm
SmmHddSecurity		File	SMM module	SmmHddSecurity
MicrocodeUpdate		File	SMM module	MicrocodeUpdate
KbcEmul		File	SMM module	KbcEmul
OFBD		File	SMM module	Ofbd
SecSMIFlash		File	SMM module	SecSMIFlash
NvOptimusSMM		File	SMM module	NvOptimusSMM
SmbiosDmiEdit		File	SMM module	SmbiosDmiEdit
SMIFlash		File	SMM module	SmiFlash
SmmLockBox		File	SMM module	SmmLockBox
TCGSmm		File	SMM module	TcgSmm
USBRT		File	SMM module	UsbRt

SMRAM

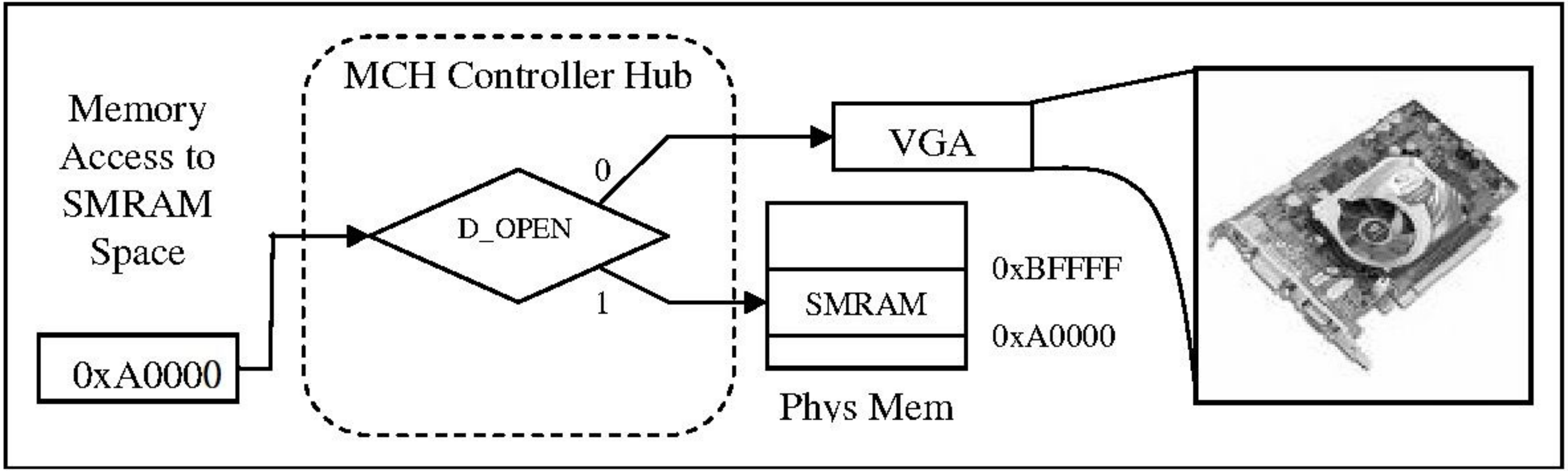
- SMM runs from its own address space called SMRAM
- A region of physical memory where SMM code and data lives
- Can be queried by reading a bunch of registers called SMRRs
- Can be closed & locked by hardware to isolate SMM from the “outside world”



```
C:\Users\carlsbad\Code\chipsec> python .\chipsec_main.py --no_banner -m common.smm_dma

WARNING: *******************************************************************
WARNING: Chipsec should only be used on test systems!
WARNING: It should not be installed/deployed on production end-user systems.
WARNING: See WARNING.txt
WARNING: *******************************************************************

[CHIPSEC] API mode: using CHIPSEC kernel module API

[+] loaded chipsec.modules.common.smm_dma
[*] running loaded modules ..

[*] running module: chipsec.modules.common.smm_dma
[x][ =======================================================================
[x][ Module: SMM TSEG Range Configuration Check
[x][ =======================================================================
[*] TSEG      : 0x000000007A000000 - 0x000000007AFFFFFF (size = 0x01000000)
[*] SMRR range: 0x000000007A000000 - 0x000000007AFFFFFF (size = 0x01000000)

[*] checking TSEG range configuration..
[+] TSEG range covers entire SMRAM
[+] TSEG range is locked
[+] PASSED: TSEG is properly configured. SMRAM is protected from DMA attacks

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Time elapsed            0.002
[CHIPSEC] Modules total           1
[CHIPSEC] Modules failed to run   0:
[CHIPSEC] Modules passed          1:
[+] PASSED: chipsec.modules.common.smm_dma
[CHIPSEC] Modules information     0:
[CHIPSEC] Modules failed          0:
[CHIPSEC] Modules with warnings   0:
[CHIPSEC] Modules not implemented 0:
[CHIPSEC] Modules not applicable  0:
[CHIPSEC] *****************************************************************
```

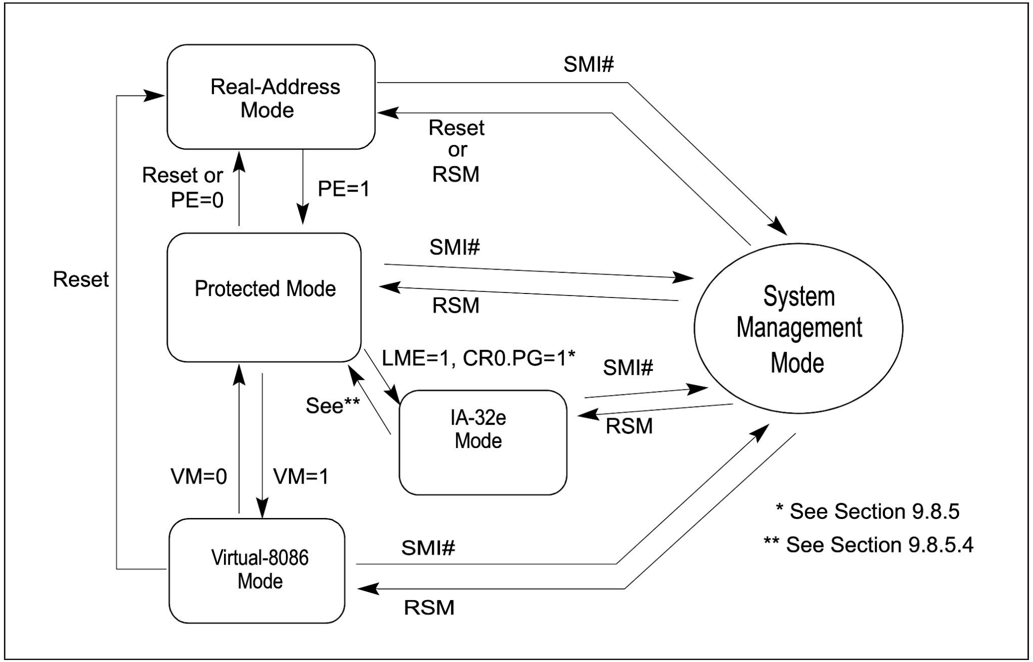
SMRAM

- Once closed, only code running in SMM can read/write SMRAM contents
- Attempts to read/write it from outside SMM (OS/hypervisor/DMA) would fail

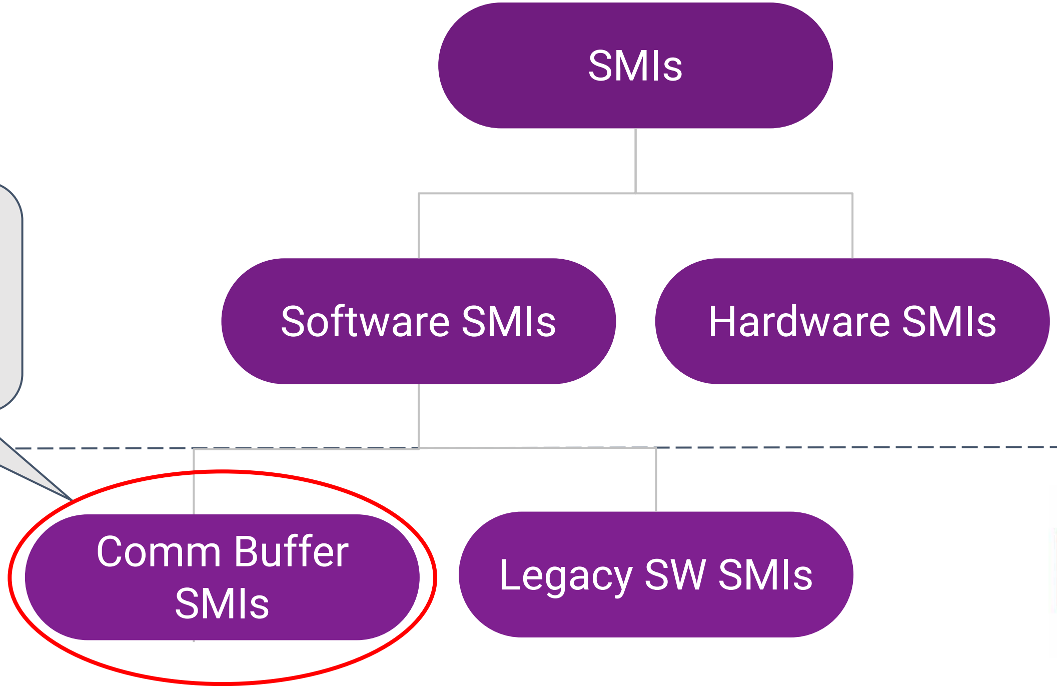


System Management Interrupts

- SMM is entered in response to an SMI
- Preempt (almost) all other code running on the CPU
- Execution jumps to an SMI handler
    - Firmware can install additional sub-handlers at boot time



Taxonomy of SMIs



SMIs

Software SMIs

Hardware SMIs

Comm Buffer SMIs

Legacy SW SMIs

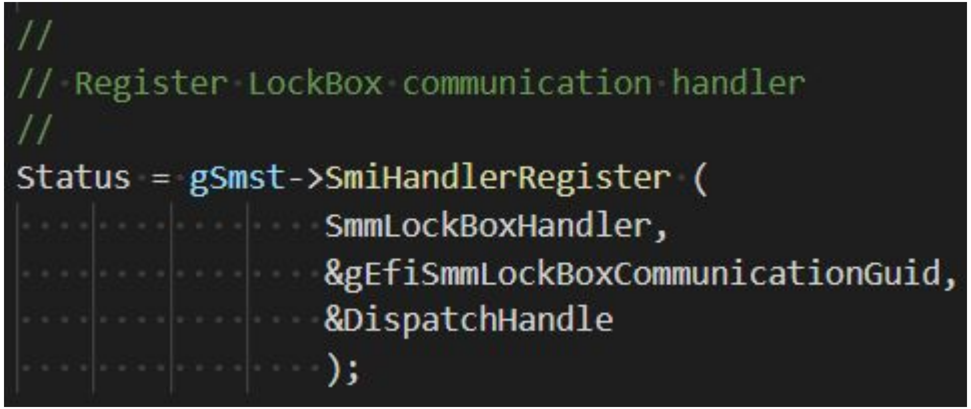
Our focal point for this talk

UNIFIED EXTENSIBLE
FIRMWARE INTERFACE

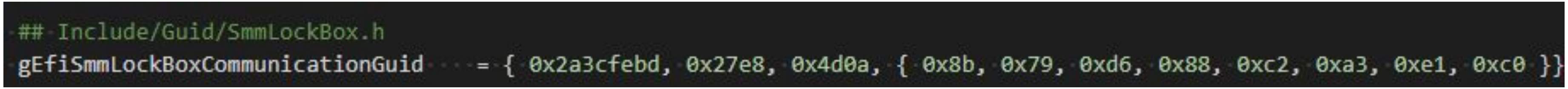
Invoking SMIs (1/9)

- In UEFI, handlers are registered via **`Smst->SmiHandlerRegister`**

- Each handler is identified by a GUID

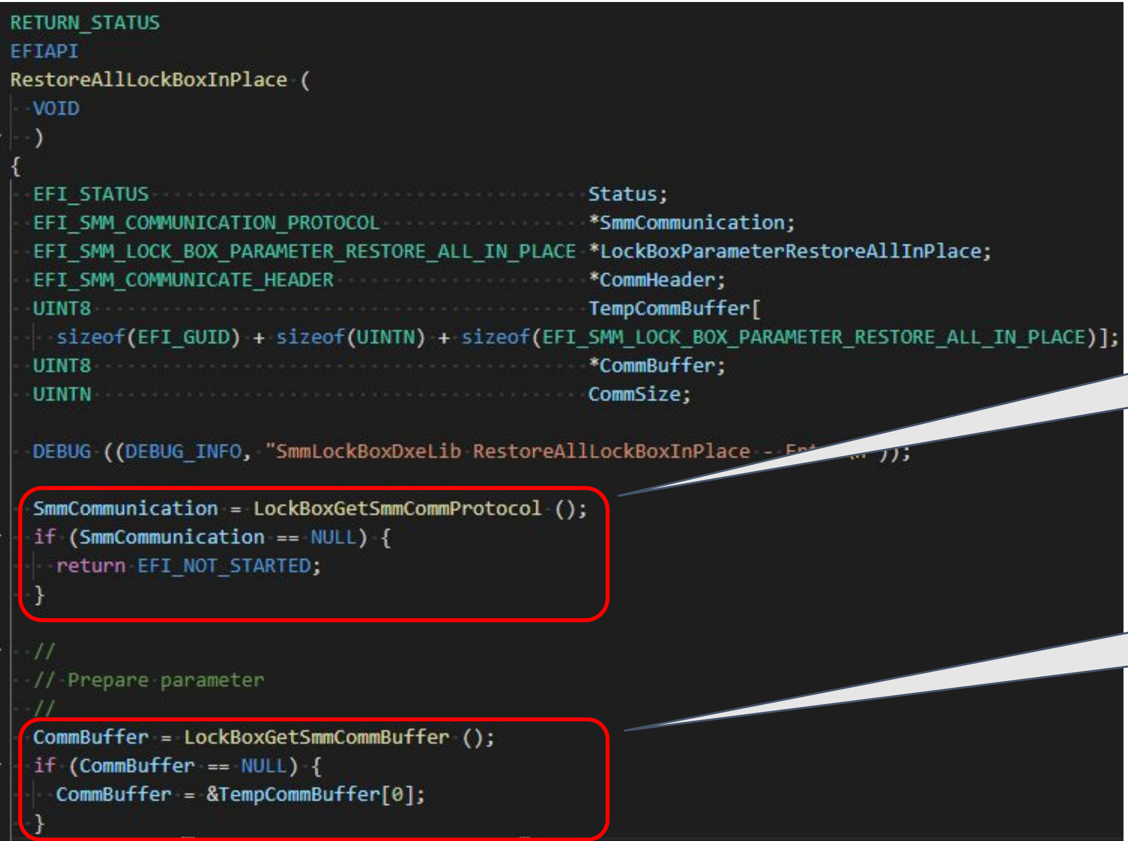


```
//
// Register LockBox communication handler
//
Status = gSmst->SmiHandlerRegister (
                  SmmLockBoxHandler,
                  &gEfiSmmLockBoxCommunicationGuid,
                  &DispatchHandle
                  );
```



```
## Include/Guid/SmmLockBox.h
gEfiSmmLockBoxCommunicationGuid    = { 0x2a3cfebd, 0x27e8, 0x4d0a, { 0x8b, 0x79, 0xd6, 0x88, 0xc2, 0xa3, 0xe1, 0xc0 }}
```

Invoking SMIs (2/9)



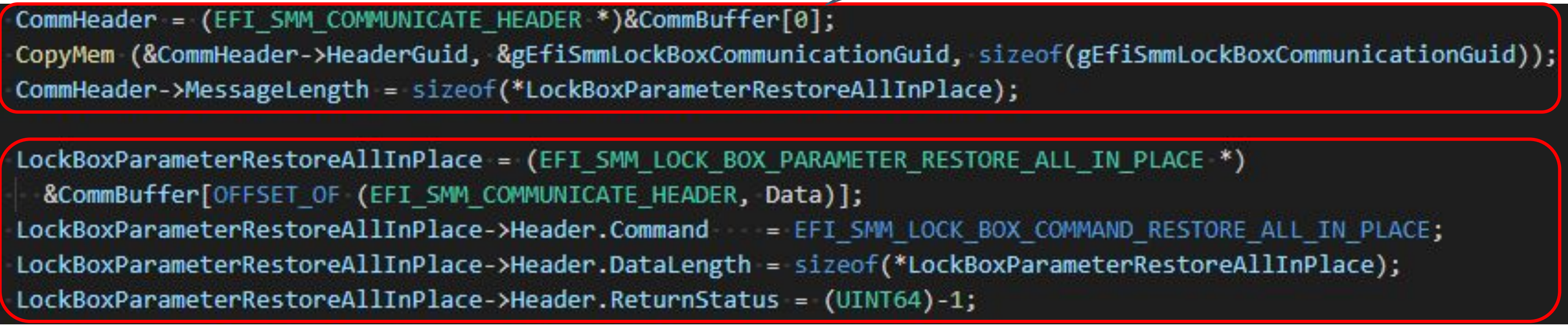
```
RETURN_STATUS
EFIAPI
RestoreAllLockBoxInPlace (
  VOID
  )
{
  EFI_STATUS                                Status;
  EFI_SMM_COMMUNICATION_PROTOCOL            *SmmCommunication;
  EFI_SMM_LOCK_BOX_PARAMETER_RESTORE_ALL_IN_PLACE *LockBoxParameterRestoreAllInPlace;
  EFI_SMM_COMMUNICATE_HEADER                *CommHeader;
  UINT8                                     TempCommBuffer[
    sizeof(EFI_GUID) + sizeof(UINTN) + sizeof(EFI_SMM_LOCK_BOX_PARAMETER_RESTORE_ALL_IN_PLACE)];
  UINT8                                     *CommBuffer;
  UINTN                                     CommSize;

  DEBUG ((DEBUG_INFO, "SmmLockBoxDxeLib RestoreAllLockBoxInPlace - Ent    "));

  SmmCommunication = LockBoxGetSmmCommProtocol ();
  if (SmmCommunication == NULL) {
    return EFI_NOT_STARTED;
  }

  //
  // Prepare parameter
  //
  CommBuffer = LockBoxGetSmmCommBuffer ();
  if (CommBuffer == NULL) {
    CommBuffer = &TempCommBuffer[0];
  }
```

Retrieves the **`EFI_SMM_COMMUNICATION_PROTOCOL`**

Returns a continuous chunk of physical memory

Invoking SMIs (3/9)

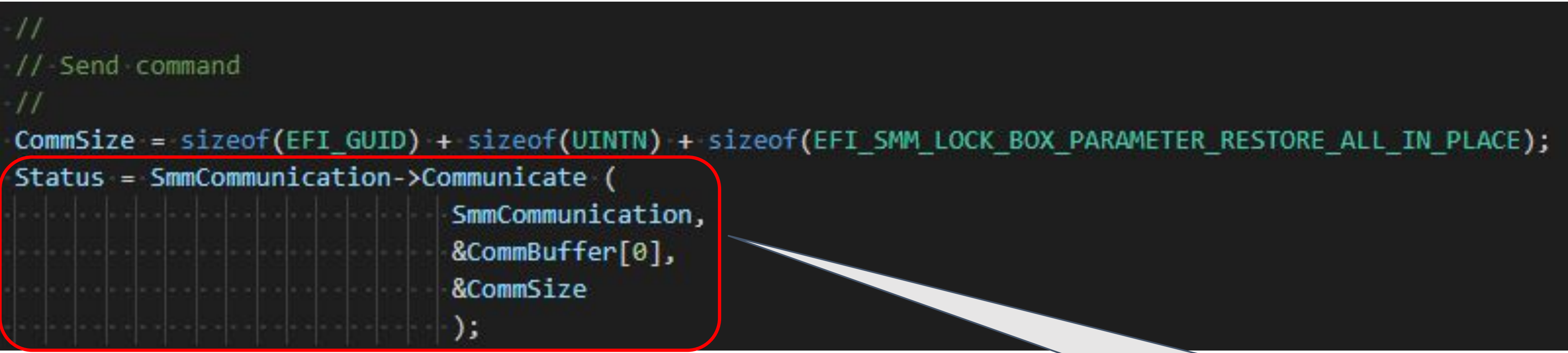
The CommBuffer is prefixed with the GUID identifying the handler and the size of data that follows



```
CommHeader = (EFI_SMM_COMMUNICATE_HEADER *)&CommBuffer[0];
CopyMem (&CommHeader->HeaderGuid, &gEfiSmmLockBoxCommunicationGuid, sizeof(gEfiSmmLockBoxCommunicationGuid));
CommHeader->MessageLength = sizeof(*LockBoxParameterRestoreAllInPlace);

LockBoxParameterRestoreAllInPlace = (EFI_SMM_LOCK_BOX_PARAMETER_RESTORE_ALL_IN_PLACE *)
  &CommBuffer[OFFSET_OF (EFI_SMM_COMMUNICATE_HEADER, Data)];
LockBoxParameterRestoreAllInPlace->Header.Command     = EFI_SMM_LOCK_BOX_COMMAND_RESTORE_ALL_IN_PLACE;
LockBoxParameterRestoreAllInPlace->Header.DataLength  = sizeof(*LockBoxParameterRestoreAllInPlace);
LockBoxParameterRestoreAllInPlace->Header.ReturnStatus = (UINT64)-1;
```

The specific argument for the SMI are placed after the header

Invoking SMIs (4/9)



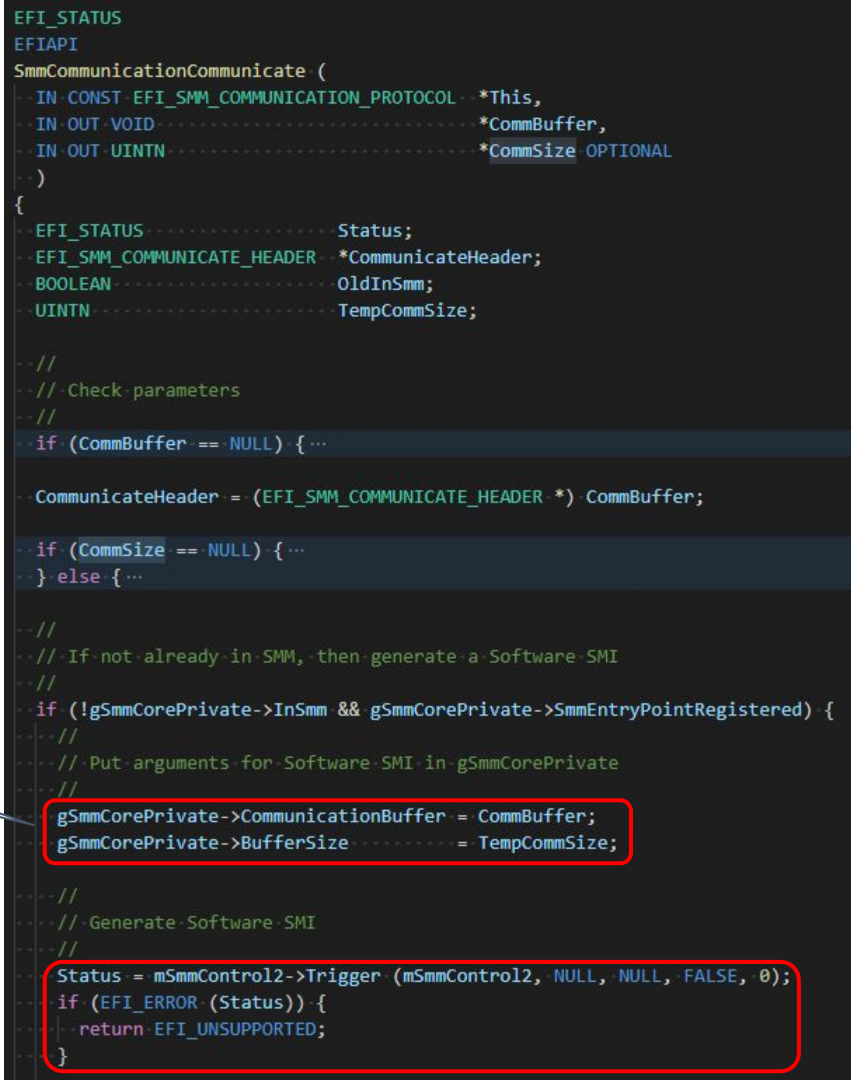
```
//
// Send command
//
CommSize = sizeof(EFI_GUID) + sizeof(UINTN) + sizeof(EFI_SMM_LOCK_BOX_PARAMETER_RESTORE_ALL_IN_PLACE);
Status = SmmCommunication->Communicate (
                             SmmCommunication,
                             &CommBuffer[0],
                             &CommSize
                             );
```

The **`Communicate()`** method of the protocol is called, which gets resolved to **`SmmCommunicationCommunicate()`**

Invoking SMIs (5/9)

Places the **`CommBuffer`** and its respective size in their designated places inside the **`gSmmCorePrivate`** structure

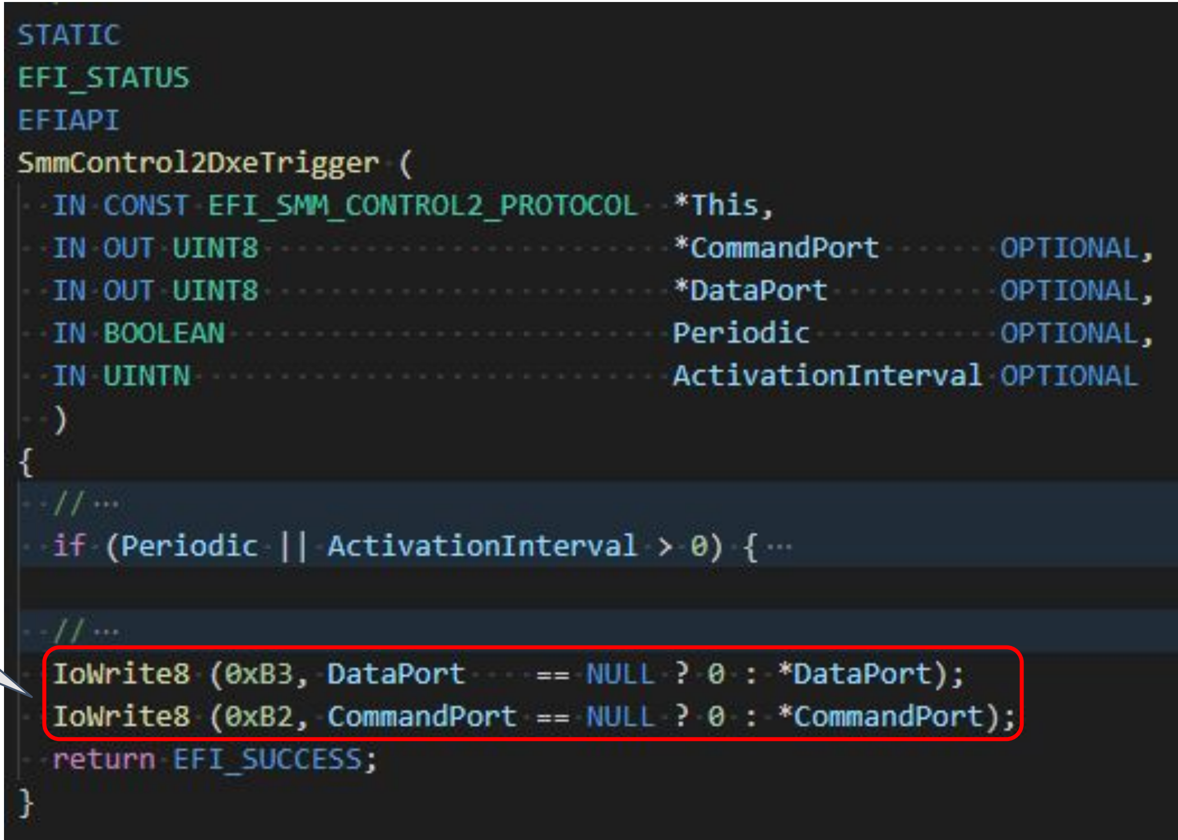
Generates a SW SMI using the **`EFI_SMM_CONTROL_PROTOCOL`**



```
EFI_STATUS
EFIAPI
SmmCommunicationCommunicate (
  IN CONST EFI_SMM_COMMUNICATION_PROTOCOL  *This,
  IN OUT VOID                              *CommBuffer,
  IN OUT UINTN                             *CommSize OPTIONAL
  )
{
  EFI_STATUS                 Status;
  EFI_SMM_COMMUNICATE_HEADER *CommunicateHeader;
  BOOLEAN                    OldInSmm;
  UINTN                      TempCommSize;

  //
  // Check parameters
  //
  if (CommBuffer == NULL) {···

  CommunicateHeader = (EFI_SMM_COMMUNICATE_HEADER *) CommBuffer;

  if (CommSize == NULL) {···
  } else {···

  //
  // If not already in SMM, then generate a Software SMI
  //
  if (!gSmmCorePrivate->InSmm && gSmmCorePrivate->SmmEntryPointRegistered) {
    //
    // Put arguments for Software SMI in gSmmCorePrivate
    //
    gSmmCorePrivate->CommunicationBuffer = CommBuffer;
    gSmmCorePrivate->BufferSize          = TempCommSize;

    //
    // Generate Software SMI
    //
    Status = mSmmControl2->Trigger (mSmmControl2, NULL, NULL, FALSE, 0);
    if (EFI_ERROR (Status)) {
      return EFI_UNSUPPORTED;
    }
```

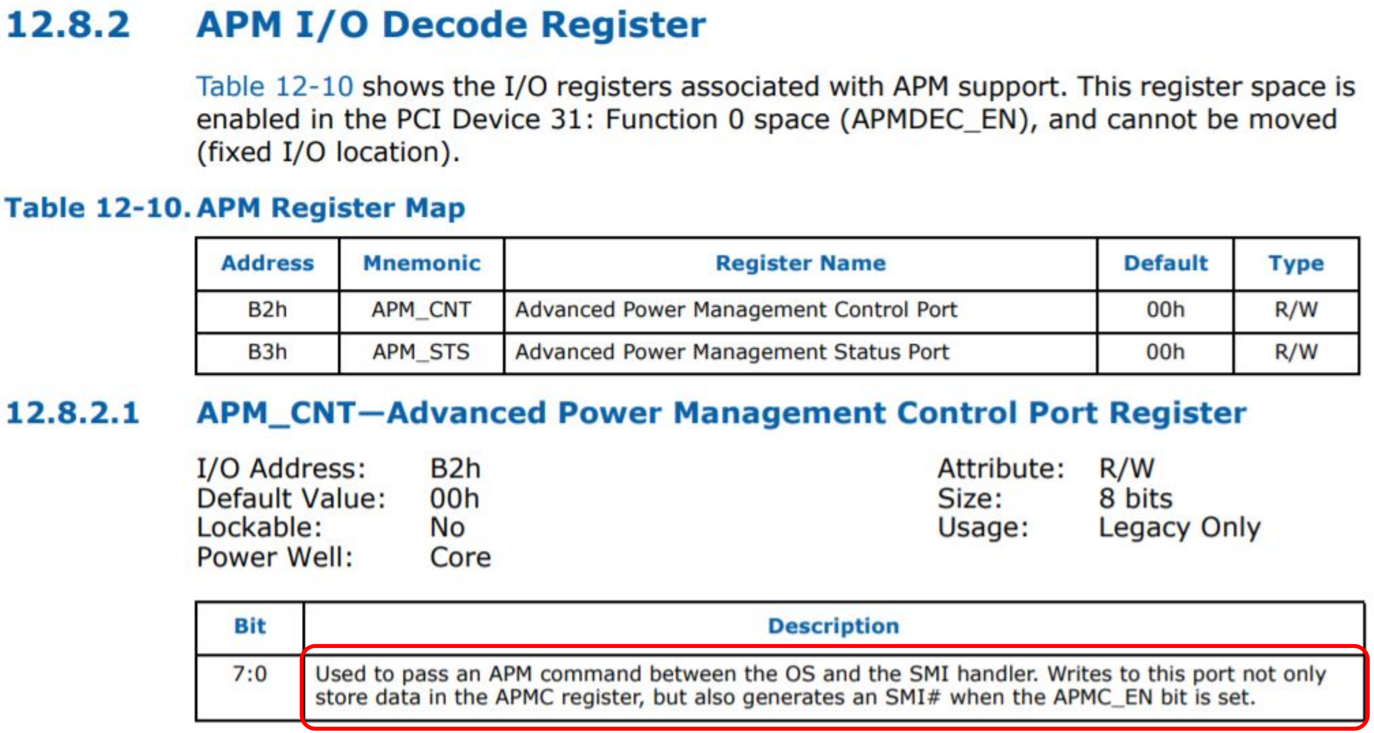
Invoking SMIs (6/9)

Writes to I/O port 0xB3 and 0xB2



```
STATIC
EFI_STATUS
EFIAPI
SmmControl2DxeTrigger (
  IN CONST EFI_SMM_CONTROL2_PROTOCOL  *This,
  IN OUT UINT8                        *CommandPort       OPTIONAL,
  IN OUT UINT8                        *DataPort          OPTIONAL,
  IN BOOLEAN                          Periodic           OPTIONAL,
  IN UINTN                            ActivationInterval OPTIONAL
  )
{
  //…
  if (Periodic || ActivationInterval > 0) {…

  //…
  IoWrite8 (0xB3, DataPort    == NULL ? 0 : *DataPort);
  IoWrite8 (0xB2, CommandPort == NULL ? 0 : *CommandPort);
  return EFI_SUCCESS;
}
```

Invoking SMIs (7/9)



12.8.2 APM I/O Decode Register

Table 12-10 shows the I/O registers associated with APM support. This register space is enabled in the PCI Device 31: Function 0 space (APMDEC_EN), and cannot be moved (fixed I/O location).

Table 12-10. APM Register Map

Address	Mnemonic	Register Name	Default	Type
B2h	APM_CNT	Advanced Power Management Control Port	00h	R/W
B3h	APM_STS	Advanced Power Management Status Port	00h	R/W

12.8.2.1 APM_CNT—Advanced Power Management Control Port Register

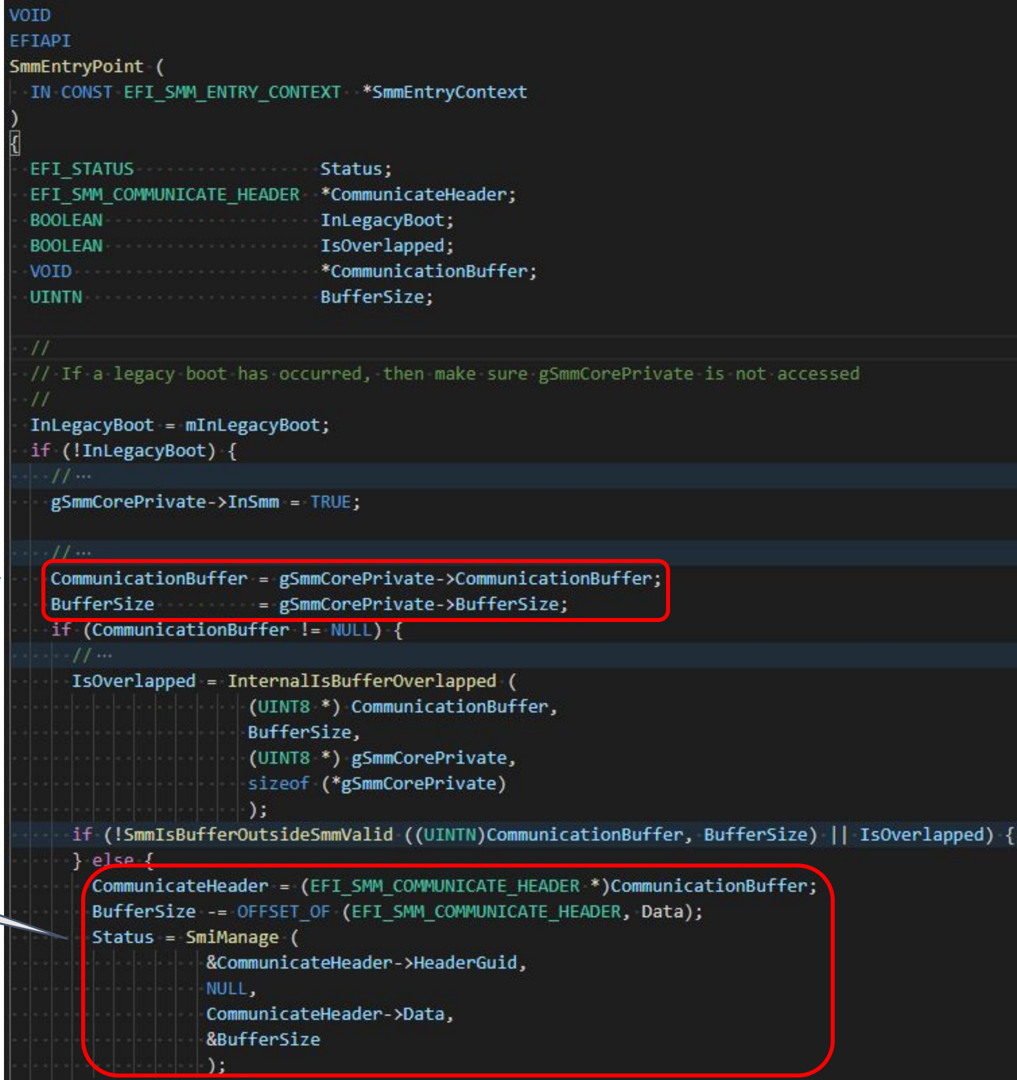
I/O Address:	B2h	Attribute:	R/W
Default Value:	00h	Size:	8 bits
Lockable:	No	Usage:	Legacy Only
Power Well:	Core		

Bit	Description
7:0	Used to pass an APM command between the OS and the SMI handler. Writes to this port not only store data in the APMC register, but also generates an SMI# when the APMC_EN bit is set.

Invoking SMIs (8/9)

The **`CommBuffer`** and its respective size are fetched from **`gSmmCorePrivate`**

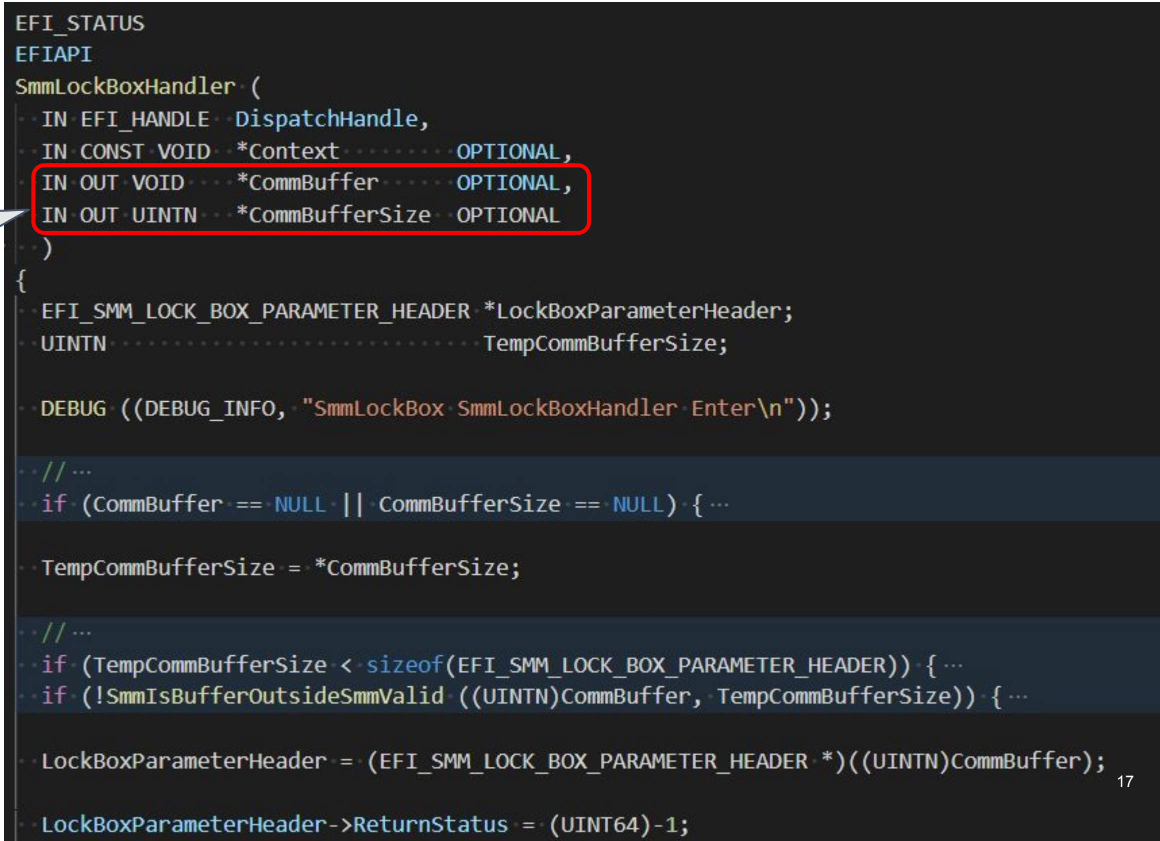
The SMI handler with the GUID found in the header is invoked



```
VOID
EFIAPI
SmmEntryPoint (
  IN CONST EFI_SMM_ENTRY_CONTEXT  *SmmEntryContext
)
{
  EFI_STATUS                  Status;
  EFI_SMM_COMMUNICATE_HEADER  *CommunicateHeader;
  BOOLEAN                     InLegacyBoot;
  BOOLEAN                     IsOverlapped;
  VOID                        *CommunicationBuffer;
  UINTN                       BufferSize;

  //
  // If a legacy boot has occurred, then make sure gSmmCorePrivate is not accessed
  //
  InLegacyBoot = mInLegacyBoot;
  if (!InLegacyBoot) {
    //…
    gSmmCorePrivate->InSmm = TRUE;

    //…
    CommunicationBuffer = gSmmCorePrivate->CommunicationBuffer;
    BufferSize          = gSmmCorePrivate->BufferSize;
    if (CommunicationBuffer != NULL) {
      //…
      IsOverlapped = InternalIsBufferOverlapped (
                       (UINT8 *) CommunicationBuffer,
                       BufferSize,
                       (UINT8 *) gSmmCorePrivate,
                       sizeof (*gSmmCorePrivate)
                       );
      if (!SmmIsBufferOutsideSmmValid ((UINTN)CommunicationBuffer, BufferSize) || IsOverlapped) {
      } else {
        CommunicateHeader = (EFI_SMM_COMMUNICATE_HEADER *)CommunicationBuffer;
        BufferSize -= OFFSET_OF (EFI_SMM_COMMUNICATE_HEADER, Data);
        Status = SmiManage (
                   &CommunicateHeader->HeaderGuid,
                   NULL,
                   CommunicateHeader->Data,
                   &BufferSize
                   );
```

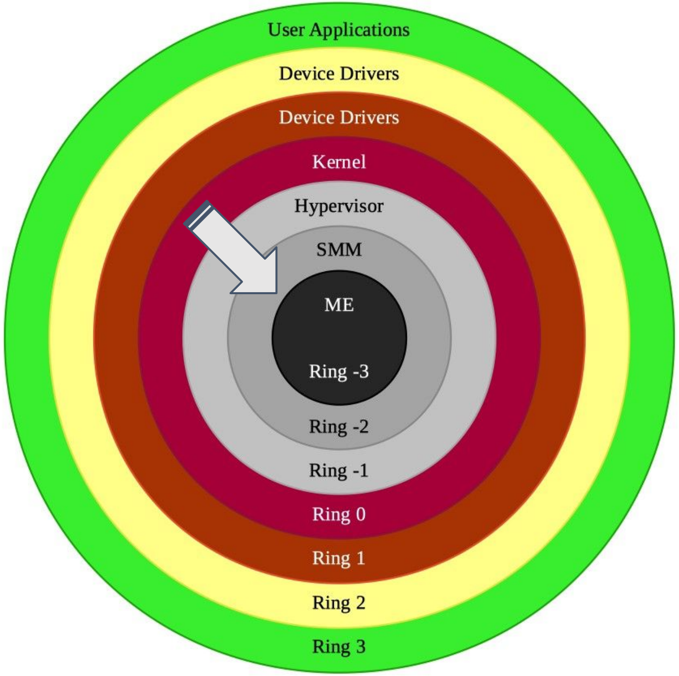
Invoking SMIs (9/9)

Handler can access the **`CommBuffer`** and **`CommBufferSize`**. Note that **`CommBuffer`** points outside of SMRAM!



```
EFI_STATUS
EFIAPI
SmmLockBoxHandler (
  IN EFI_HANDLE  DispatchHandle,
  IN CONST VOID  *Context         OPTIONAL,
  IN OUT VOID    *CommBuffer      OPTIONAL,
  IN OUT UINTN   *CommBufferSize  OPTIONAL
  )
{
  EFI_SMM_LOCK_BOX_PARAMETER_HEADER *LockBoxParameterHeader;
  UINTN                              TempCommBufferSize;

  DEBUG ((DEBUG_INFO, "SmmLockBox SmmLockBoxHandler Enter\n"));

  //…
  if (CommBuffer == NULL || CommBufferSize == NULL) {…

  TempCommBufferSize = *CommBufferSize;

  //…
  if (TempCommBufferSize < sizeof(EFI_SMM_LOCK_BOX_PARAMETER_HEADER)) {…
  if (!SmmIsBufferOutsideSmmValid ((UINTN)CommBuffer, TempCommBufferSize)) {…

  LockBoxParameterHeader = (EFI_SMM_LOCK_BOX_PARAMETER_HEADER *)((UINTN)CommBuffer);

  LockBoxParameterHeader->ReturnStatus = (UINT64)-1;
```

Attacks against SMM

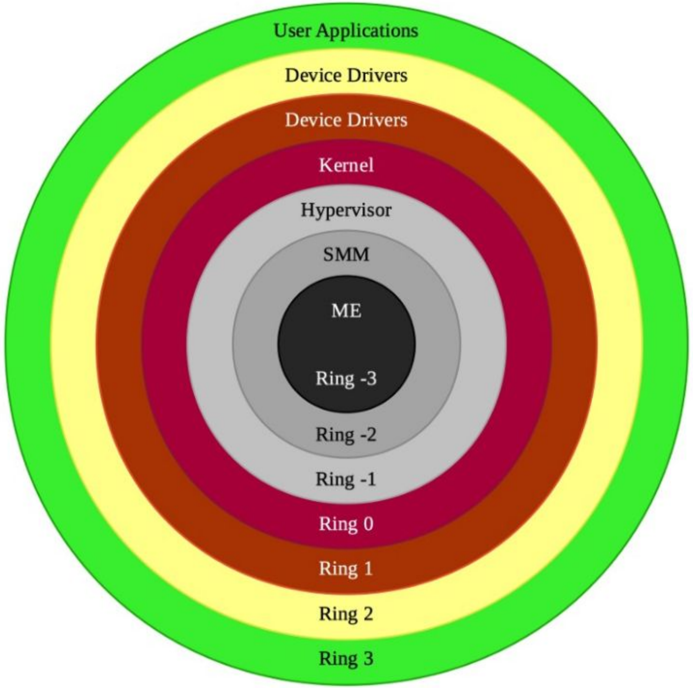


SMM privileges

- SMM code is highly privileged

- You can think of SMM code as “ring -2"
  - More powerful than the kernel (ring 0) and the hypervisor (ring -1)

- SMM “superpowers”
  - Invisible to all the layers above it (SMRAM)
  - Full access to all physical memory
  - Full access to all MSRs
  - Can write the BIOS region on the SPI flash

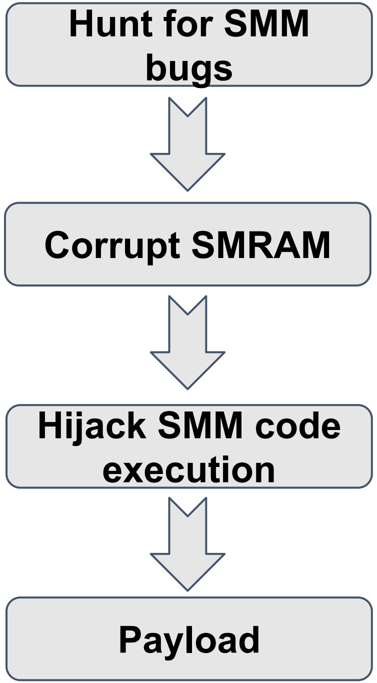


IA Negative Rings

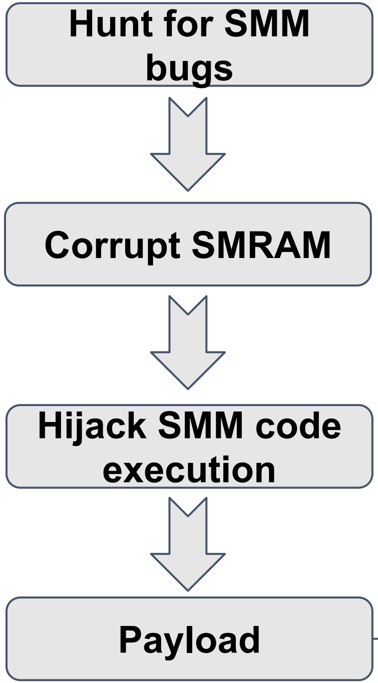
Attack scenario

- Goal: elevate privileges to ring -2
- Assumption: ring 0 privileges
  - We can freely issue SW SMIs
- Vector: confused deputy attack against SMI handlers
  - The privileged SMI handler will be “tricked” to corrupt/modify SMRAM contents

Full attack flow



Full attack flow

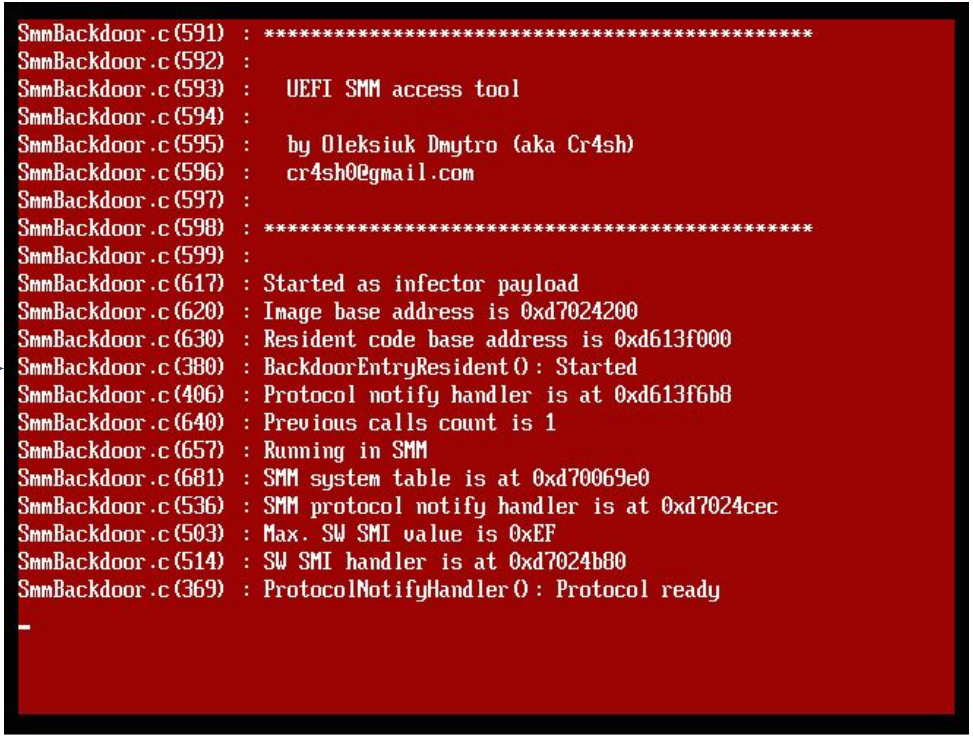


Hunt for SMM bugs

Corrupt SMRAM

Hijack SMM code execution

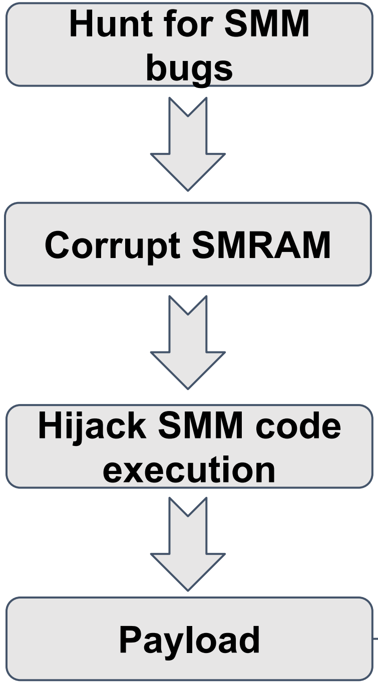
Payload

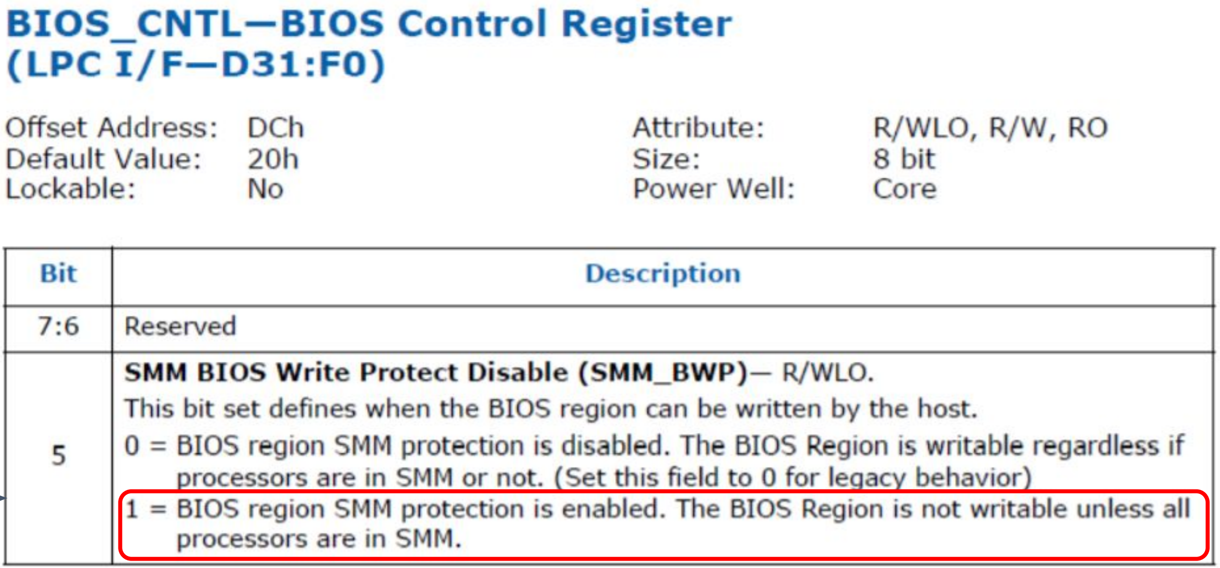


```
SmmBackdoor.c(591) : ************************************************
SmmBackdoor.c(592) :
SmmBackdoor.c(593) :   UEFI SMM access tool
SmmBackdoor.c(594) :
SmmBackdoor.c(595) :   by Oleksiuk Dmytro (aka Cr4sh)
SmmBackdoor.c(596) :   cr4sh0@gmail.com
SmmBackdoor.c(597) :
SmmBackdoor.c(598) : ************************************************
SmmBackdoor.c(599) :
SmmBackdoor.c(617) : Started as infector payload
SmmBackdoor.c(620) : Image base address is 0xd7024200
SmmBackdoor.c(630) : Resident code base address is 0xd613f000
SmmBackdoor.c(380) : BackdoorEntryResident() : Started
SmmBackdoor.c(406) : Protocol notify handler is at 0xd613f6b8
SmmBackdoor.c(640) : Previous calls count is 1
SmmBackdoor.c(657) : Running in SMM
SmmBackdoor.c(681) : SMM system table is at 0xd70069e0
SmmBackdoor.c(536) : SMM protocol notify handler is at 0xd7024cec
SmmBackdoor.c(503) : Max. SW SMI value is 0xEF
SmmBackdoor.c(514) : SW SMI handler is at 0xd7024b80
SmmBackdoor.c(369) : ProtocolNotifyHandler() : Protocol ready
_
```

SMM backdoor
https://github.com/Cr4sh/SmmBackdoor

Full attack flow





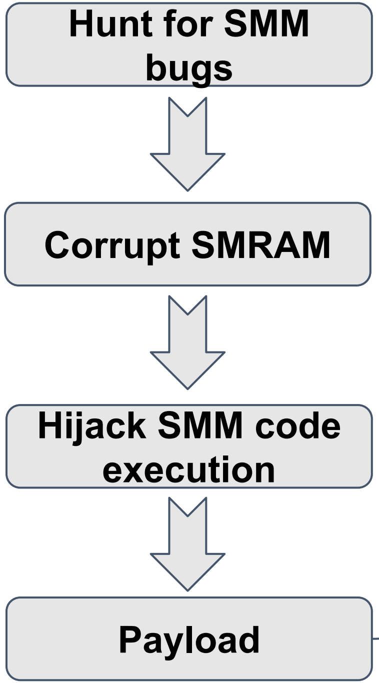
BIOS_CNTL—BIOS Control Register (LPC I/F—D31:F0)

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.

Re-flash the BIOS
https://opensecuritytraining.info/IntroBIOS_files/Day2_03_Advanced%20x86%20-%20BIOS%20and%20SMM%20Internals%20-%20SPI%20Flash%20Protection%20Mechanisms.pptx

Full attack flow

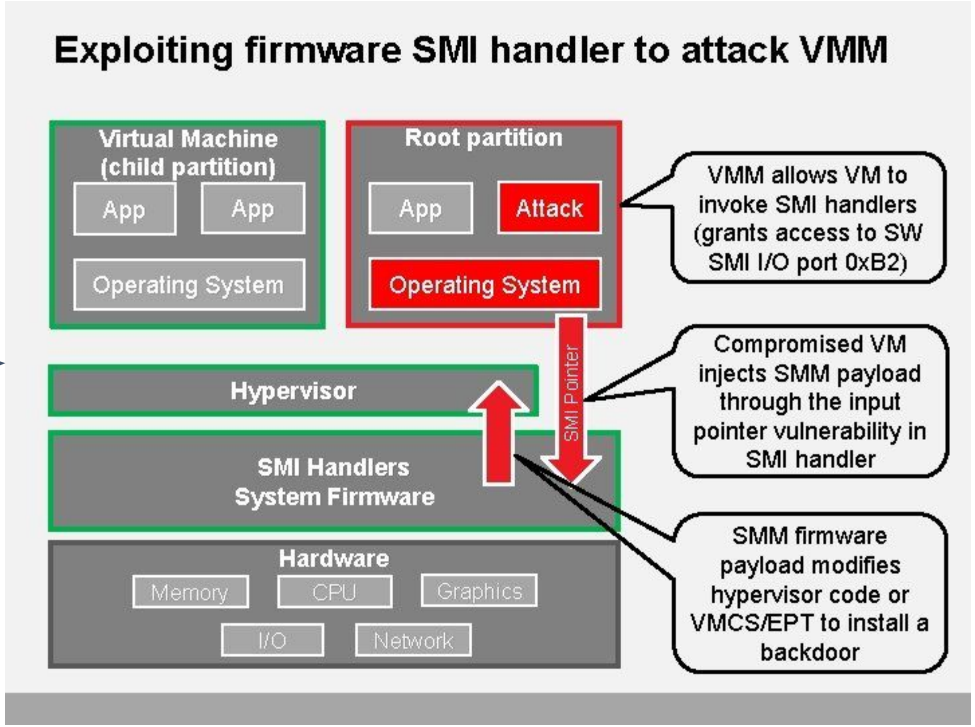


Hunt for SMM bugs

Corrupt SMRAM

Hijack SMM code execution

Payload



Exploiting firmware SMI handler to attack VMM

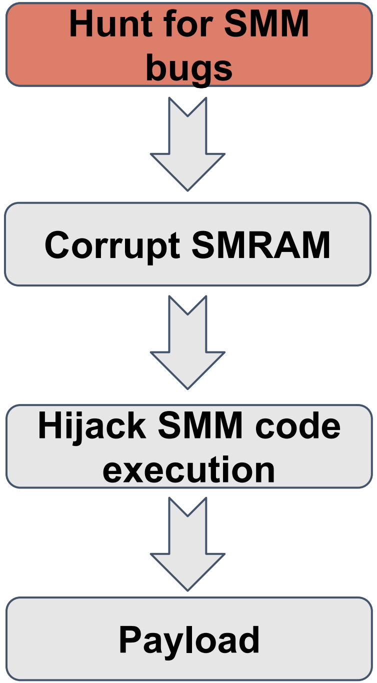
VMM allows VM to invoke SMI handlers (grants access to SW SMI I/O port 0xB2)

Compromised VM injects SMM payload through the input pointer vulnerability in SMI handler

SMM firmware payload modifies hypervisor code or VMCS/EPT to install a backdoor

Infect the hypervisor and guest VMs
http://c7zero.info/stuff/AttackingHypervisorsViaFirmware_bhusa15_dc23.pdf

Full attack flow



Hunt for SMM bugs

Corrupt SMRAM

Hijack SMM code execution

Payload

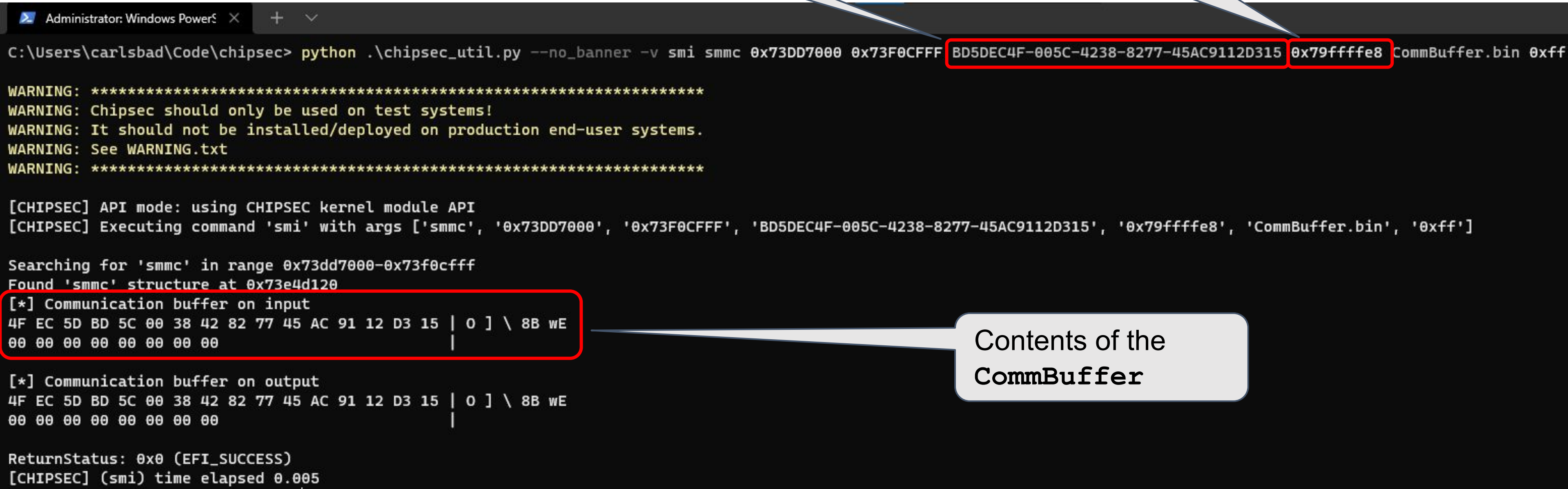
We’ll only focus on the first phase in this talk!

Attack surface

- A lot of attacker controlled parameters

GUID of the handler

Address of the **`CommBuffer`**



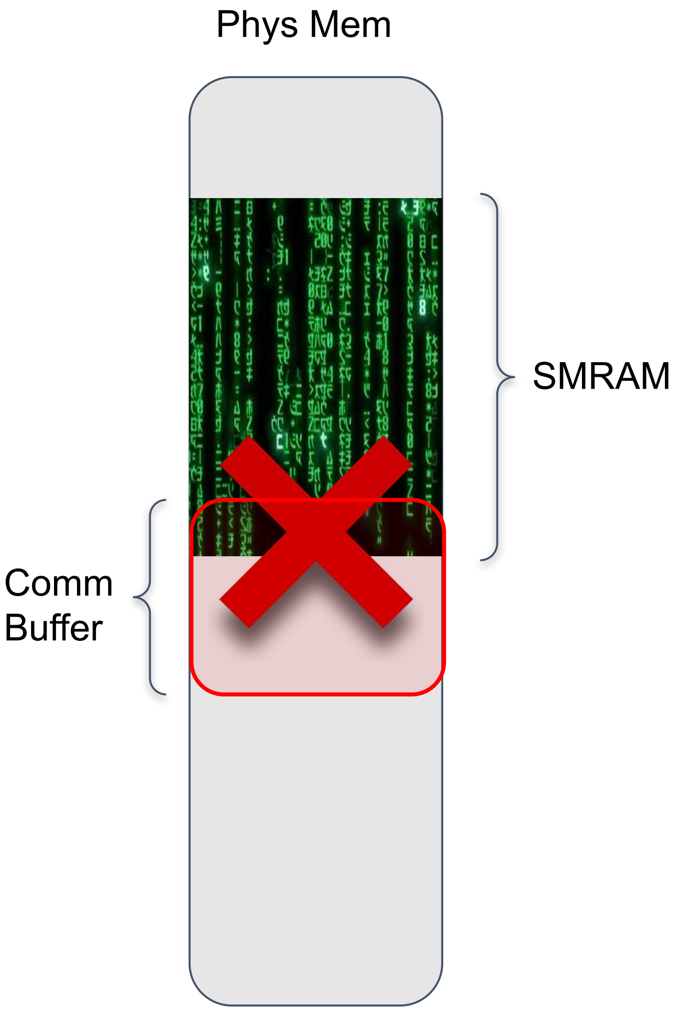
```
C:\Users\carlsbad\Code\chipsec> python .\chipsec_util.py --no_banner -v smi smmc 0x73DD7000 0x73F0CFFF BD5DEC4F-005C-4238-8277-45AC9112D315 0x79ffffe8 CommBuffer.bin 0xff

WARNING: *******************************************************************
WARNING: Chipsec should only be used on test systems!
WARNING: It should not be installed/deployed on production end-user systems.
WARNING: See WARNING.txt
WARNING: *******************************************************************

[CHIPSEC] API mode: using CHIPSEC kernel module API
[CHIPSEC] Executing command 'smi' with args ['smmc', '0x73DD7000', '0x73F0CFFF', 'BD5DEC4F-005C-4238-8277-45AC9112D315', '0x79ffffe8', 'CommBuffer.bin', '0xff']

Searching for 'smmc' in range 0x73dd7000-0x73f0cfff
Found 'smmc' structure at 0x73e4d120
[*] Communication buffer on input
4F EC 5D BD 5C 00 38 42 82 77 45 AC 91 12 D3 15 | O ] \ 8B wE
00 00 00 00 00 00 00 00                         |

[*] Communication buffer on output
4F EC 5D BD 5C 00 38 42 82 77 45 AC 91 12 D3 15 | O ] \ 8B wE
00 00 00 00 00 00 00 00                         |

ReturnStatus: 0x0 (EFI_SUCCESS)
[CHIPSEC] (smi) time elapsed 0.005
```

Contents of the **`CommBuffer`**

Restrictions

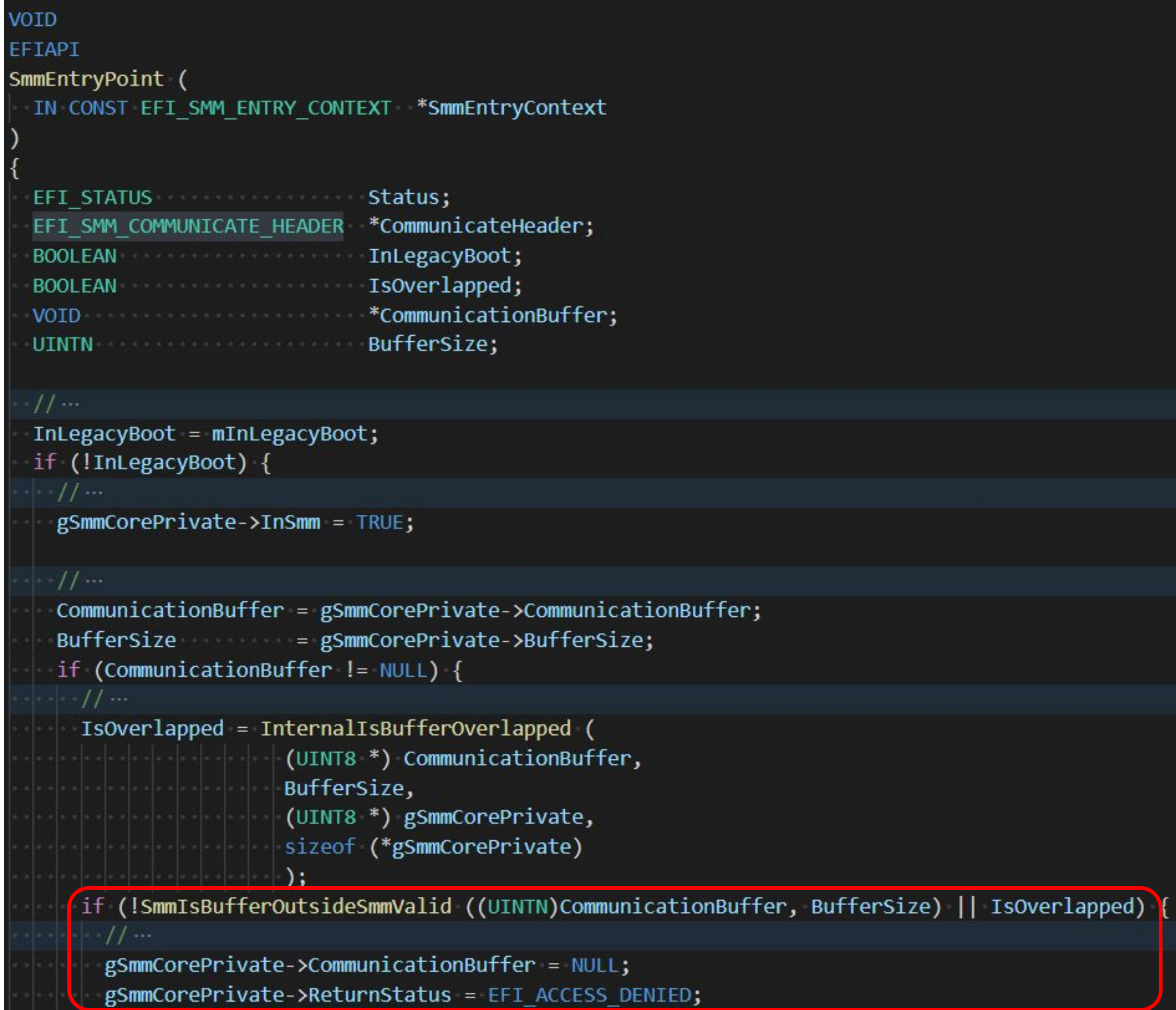
- To protect SMRAM, the Comm Buffer cannot overlap with SMRAM

- Otherwise, any handler that writes results to the CommBuffer will also modify SMRAM contents



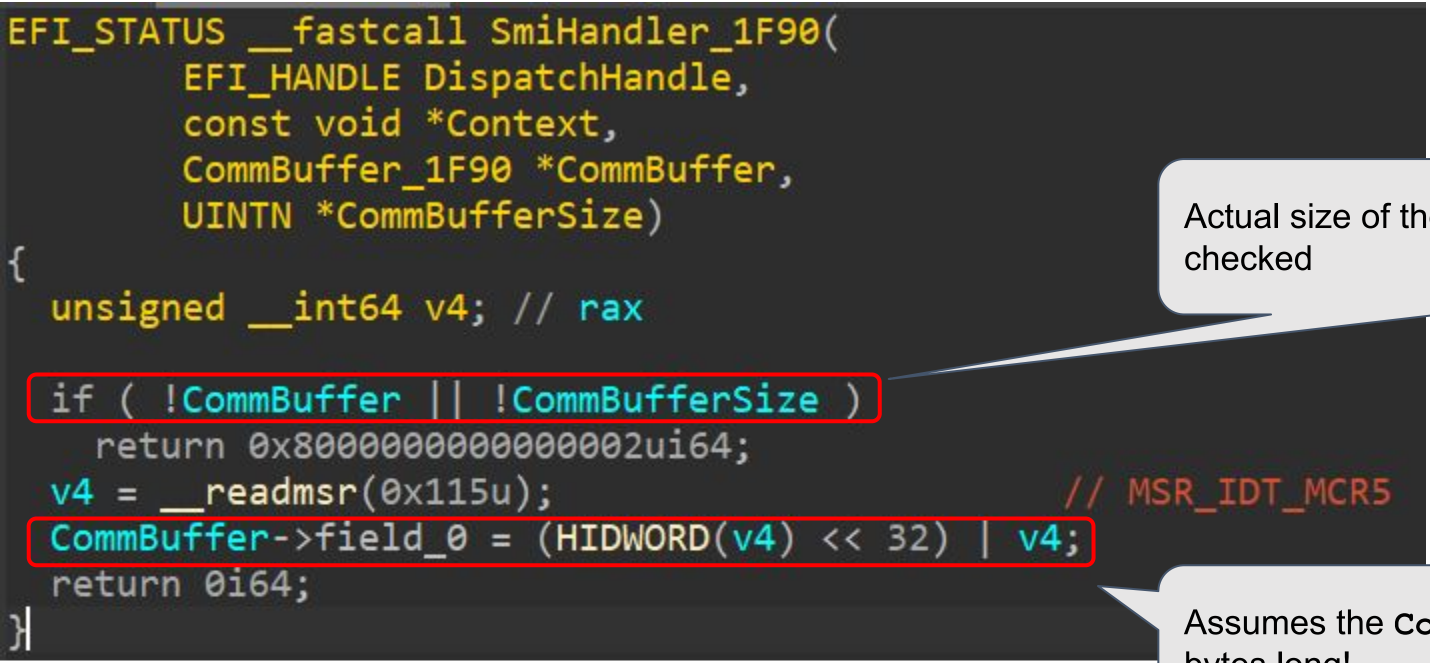
Restrictions

- Checked using **`SmmIsBufferOutsideSmmValid()`**
- However, some poorly written SMI handler allows us to bypass this restriction



```
VOID
EFIAPI
SmmEntryPoint (
  IN CONST EFI_SMM_ENTRY_CONTEXT  *SmmEntryContext
)
{
  EFI_STATUS                  Status;
  EFI_SMM_COMMUNICATE_HEADER  *CommunicateHeader;
  BOOLEAN                     InLegacyBoot;
  BOOLEAN                     IsOverlapped;
  VOID                        *CommunicationBuffer;
  UINTN                       BufferSize;

  //...
  InLegacyBoot = mInLegacyBoot;
  if (!InLegacyBoot) {
    //...
    gSmmCorePrivate->InSmm = TRUE;

    //...
    CommunicationBuffer = gSmmCorePrivate->CommunicationBuffer;
    BufferSize          = gSmmCorePrivate->BufferSize;
    if (CommunicationBuffer != NULL) {
      //...
      IsOverlapped = InternalIsBufferOverlapped (
                       (UINT8 *) CommunicationBuffer,
                       BufferSize,
                       (UINT8 *) gSmmCorePrivate,
                       sizeof (*gSmmCorePrivate)
                       );
      if (!SmmIsBufferOutsideSmmValid ((UINTN)CommunicationBuffer, BufferSize) || IsOverlapped) {
        //...
        gSmmCorePrivate->CommunicationBuffer = NULL;
        gSmmCorePrivate->ReturnStatus = EFI_ACCESS_DENIED;
```

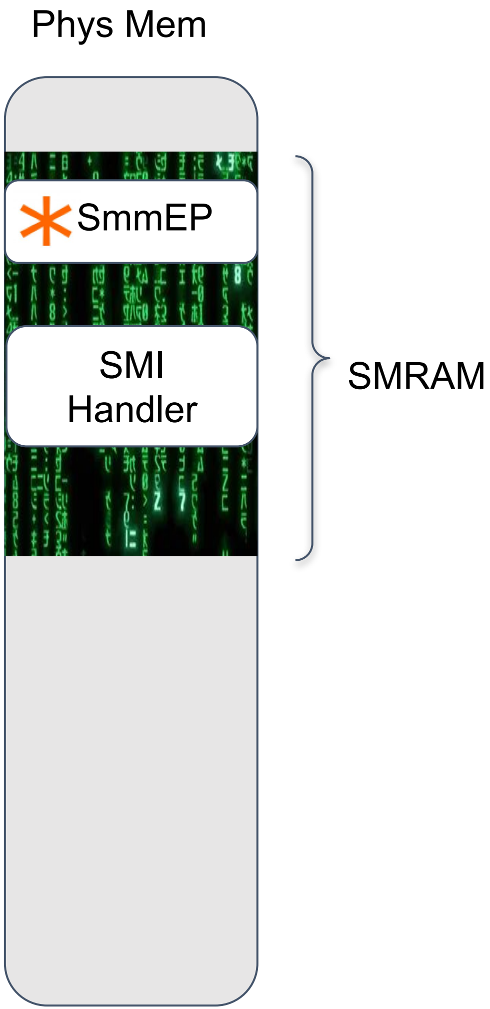
#1: Not validating CommBufferSize



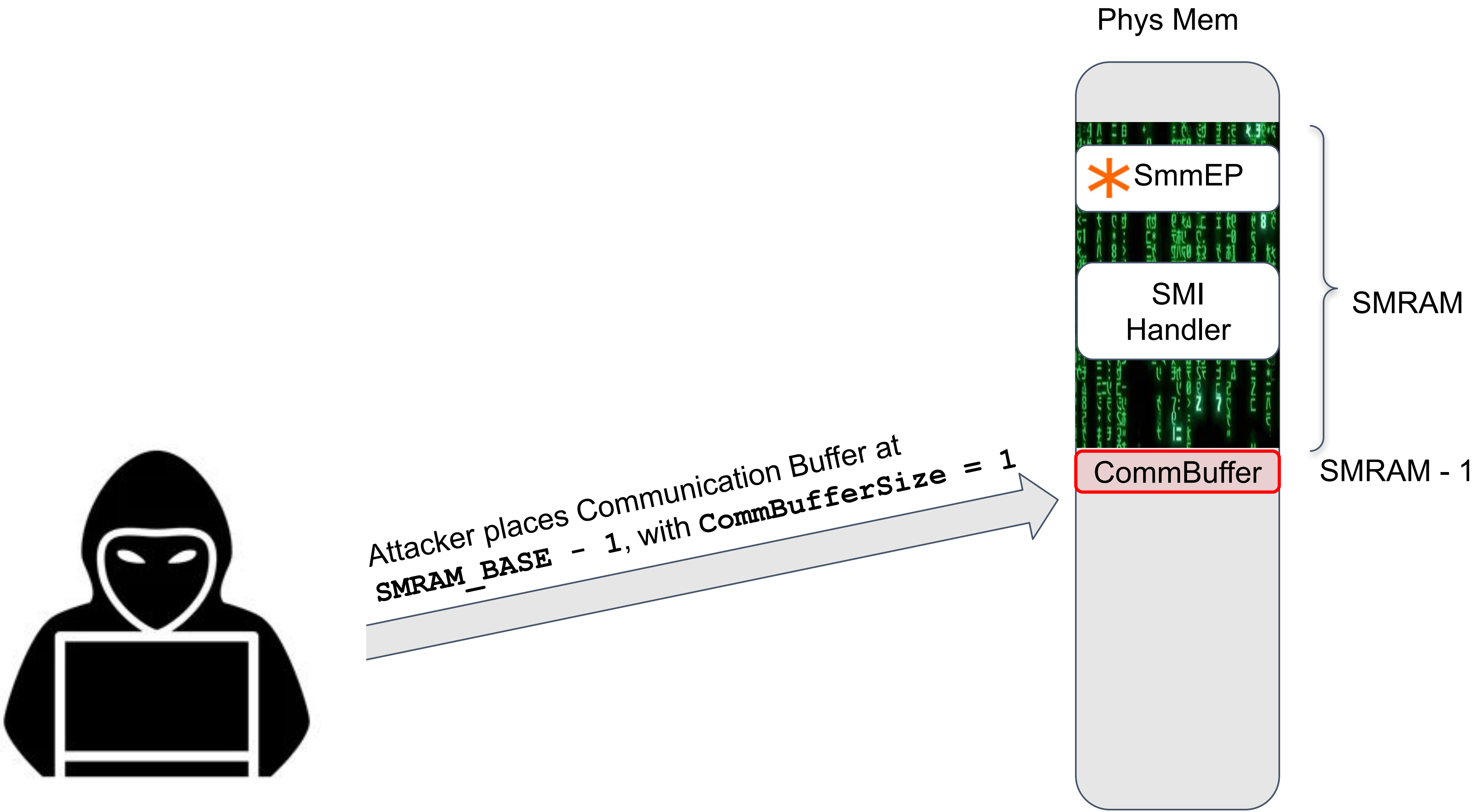
```
EFI_STATUS __fastcall SmiHandler_1F90(
        EFI_HANDLE DispatchHandle,
        const void *Context,
        CommBuffer_1F90 *CommBuffer,
        UINTN *CommBufferSize)
{
  unsigned __int64 v4; // rax

  if ( !CommBuffer || !CommBufferSize )
    return 0x8000000000000002ui64;
  v4 = __readmsr(0x115u);                       // MSR_IDT_MCR5
  CommBuffer->field_0 = (HIDWORD(v4) << 32) | v4;
  return 0i64;
}
```

Actual size of the **`CommBuffer`** is not checked

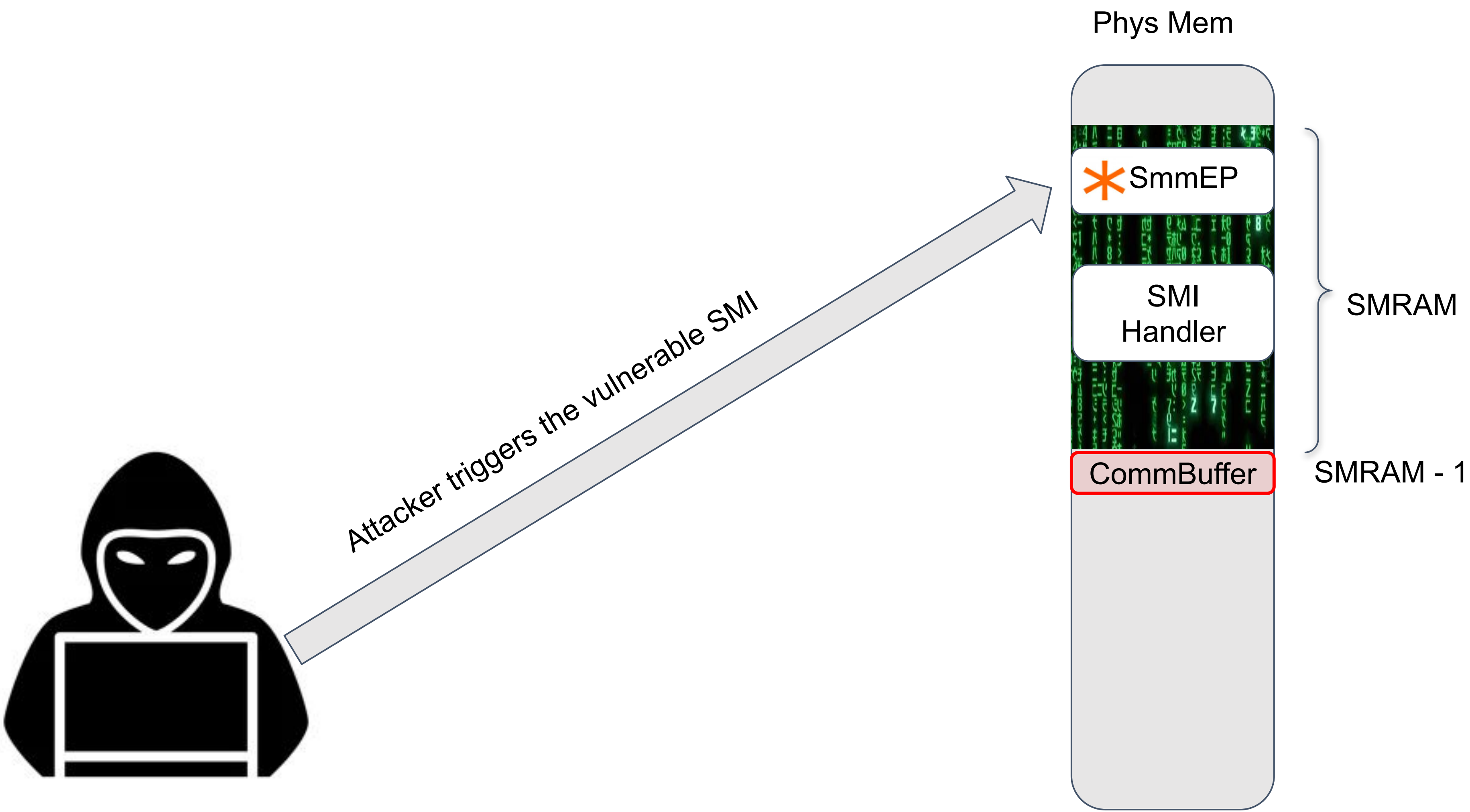
Assumes the **`CommBuffer`** is at least 8 bytes long!



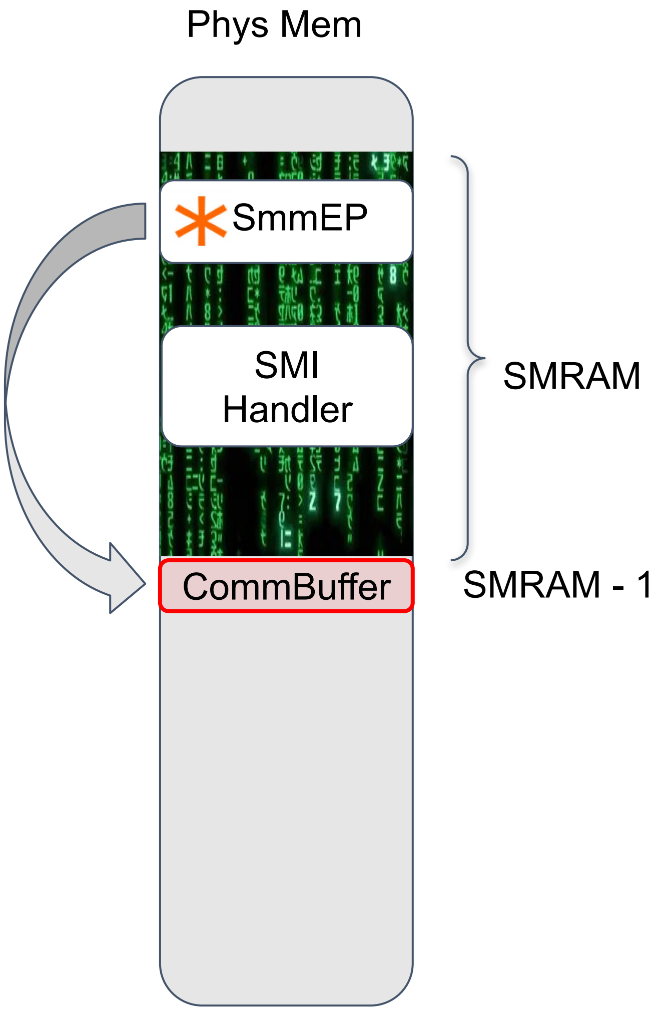
Phys Mem
SmmEP
SMI Handler
SMRAM



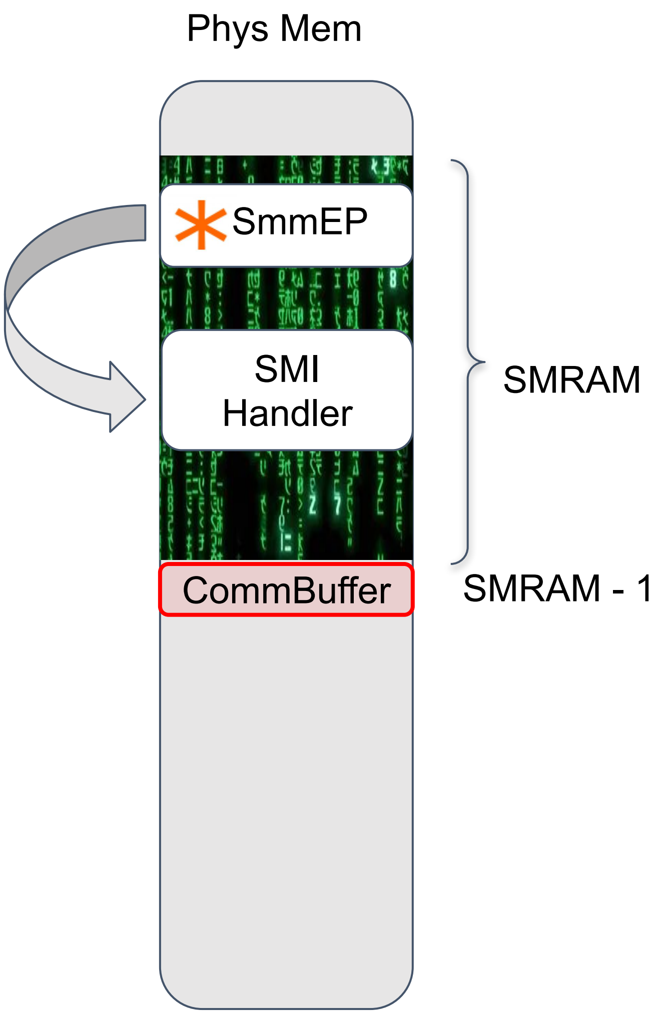
Attacker places Communication Buffer at **SMRAM_BASE - 1**, with **CommBufferSize = 1**


Phys Mem
Attacker triggers the vulnerable SMI
SmmEP
SMI
Handler
SMRAM
CommBuffer
SMRAM - 1

`SmmEntryPoint` checks that **`CommBuffer`** does not overlap with SMRAM



✓ Check is successful,
execute SMI handler



Phys Mem

SmmEP

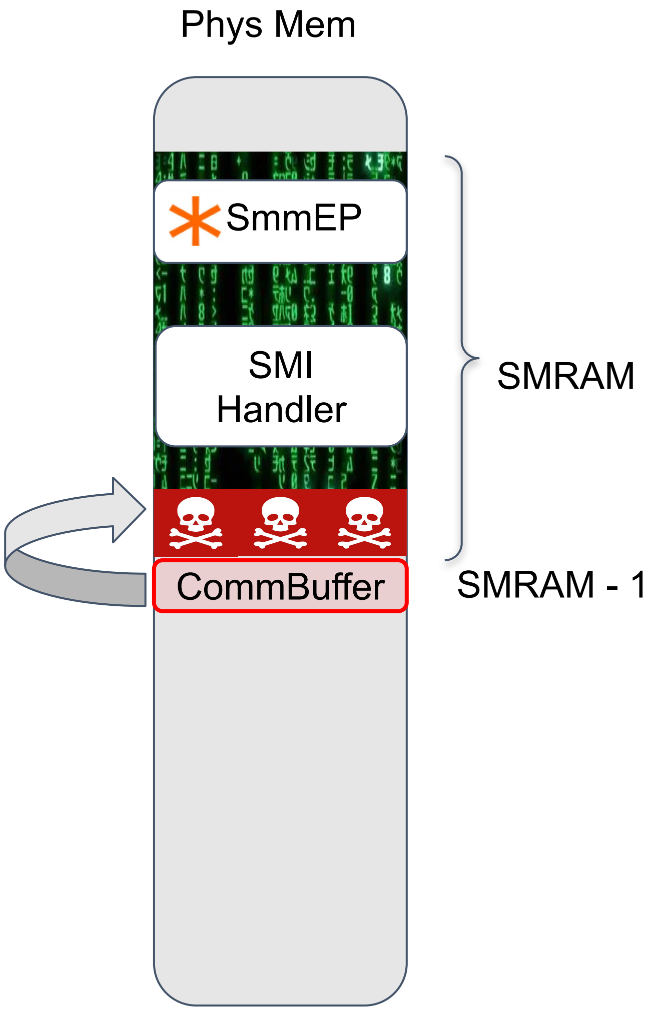
SMI
Handler

SMRAM

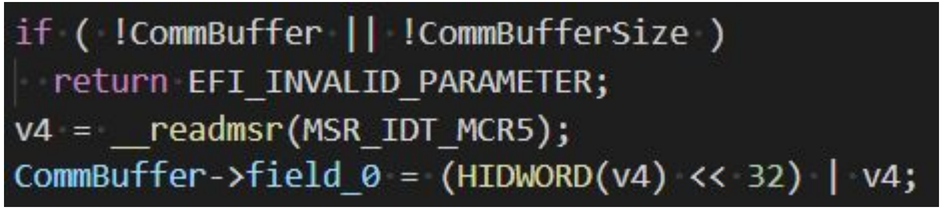
CommBuffer

SMRAM - 1

Handler blindly writes a **`QWORD`** to the **`CommBuffer`**, corrupting the lower portion of SMRAM



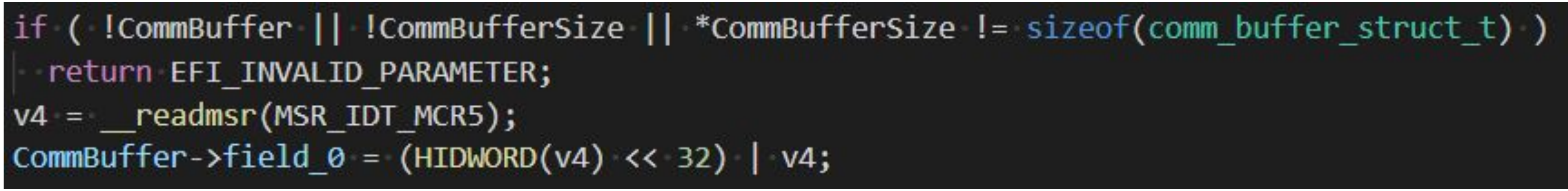
❌



```
if ( !CommBuffer || !CommBufferSize )
  return EFI_INVALID_PARAMETER;
v4 = __readmsr(MSR_IDT_MCR5);
CommBuffer->field_0 = (HIDWORD(v4) << 32) | v4;
```

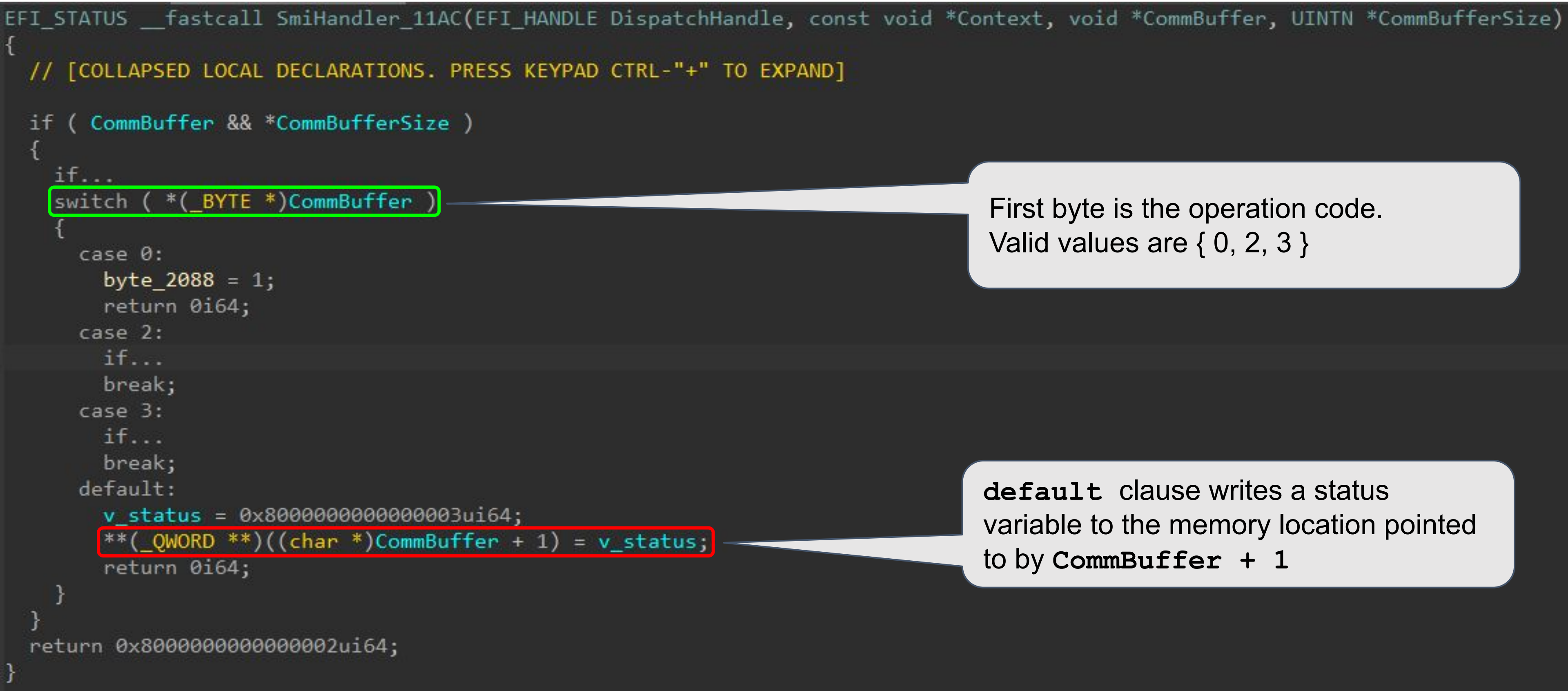
Handlers should explicitly check that **`CommBufferSize`** matches the expected size

✅



```
if ( !CommBuffer || !CommBufferSize || *CommBufferSize != sizeof(comm_buffer_struct_t) )
  return EFI_INVALID_PARAMETER;
v4 = __readmsr(MSR_IDT_MCR5);
CommBuffer->field_0 = (HIDWORD(v4) << 32) | v4;
```

#2: Unsanitized nested pointers

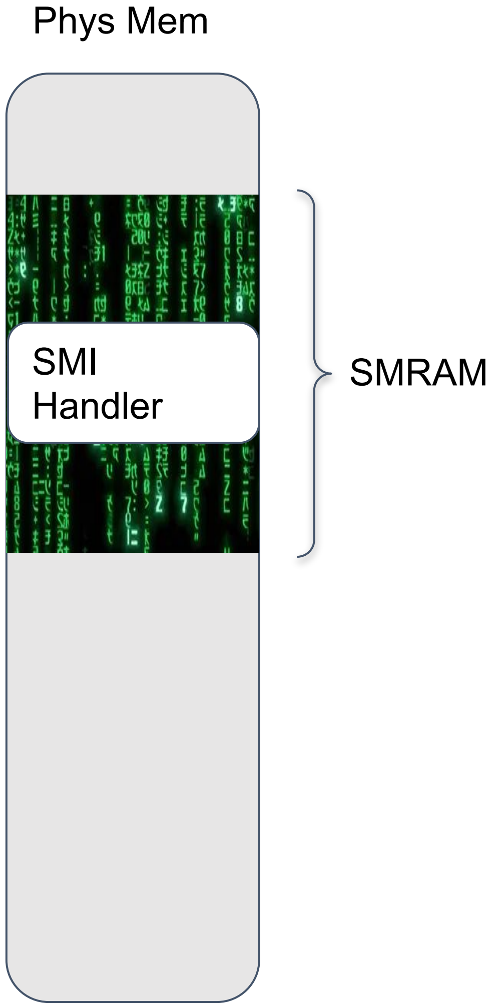


```
EFI_STATUS __fastcall SmiHandler_11AC(EFI_HANDLE DispatchHandle, const void *Context, void *CommBuffer, UINTN *CommBufferSize)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( CommBuffer && *CommBufferSize )
  {
    if...
    switch ( *(_BYTE *)CommBuffer )
    {
      case 0:
        byte_2088 = 1;
        return 0i64;
      case 2:
        if...
        break;
      case 3:
        if...
        break;
      default:
        v_status = 0x8000000000000003ui64;
        **(_QWORD **)((char *)CommBuffer + 1) = v_status;
        return 0i64;
    }
  }
  return 0x8000000000000002ui64;
}
```

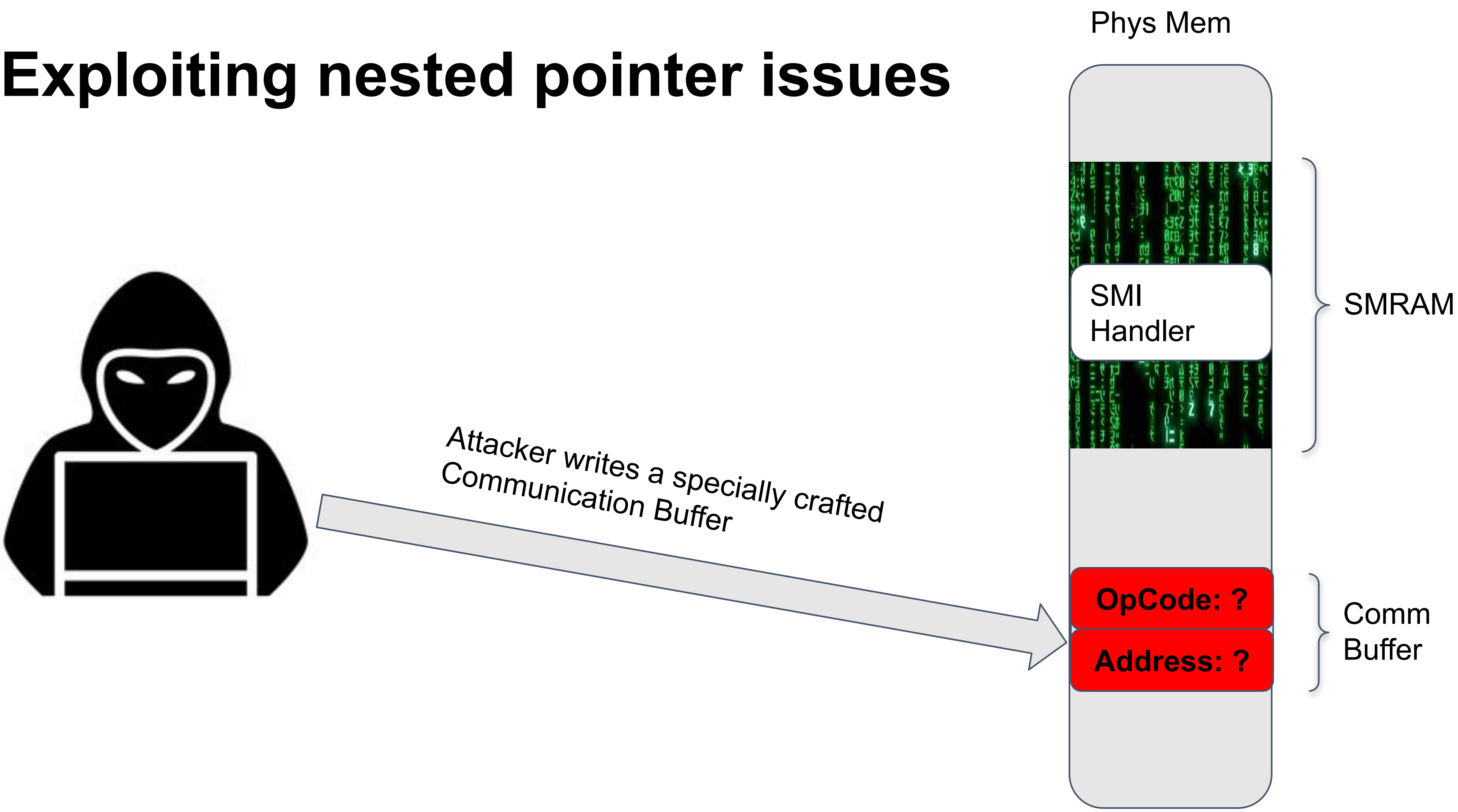
First byte is the operation code.
Valid values are { 0, 2, 3 }

`default` clause writes a status variable to the memory location pointed to by **`CommBuffer + 1`**

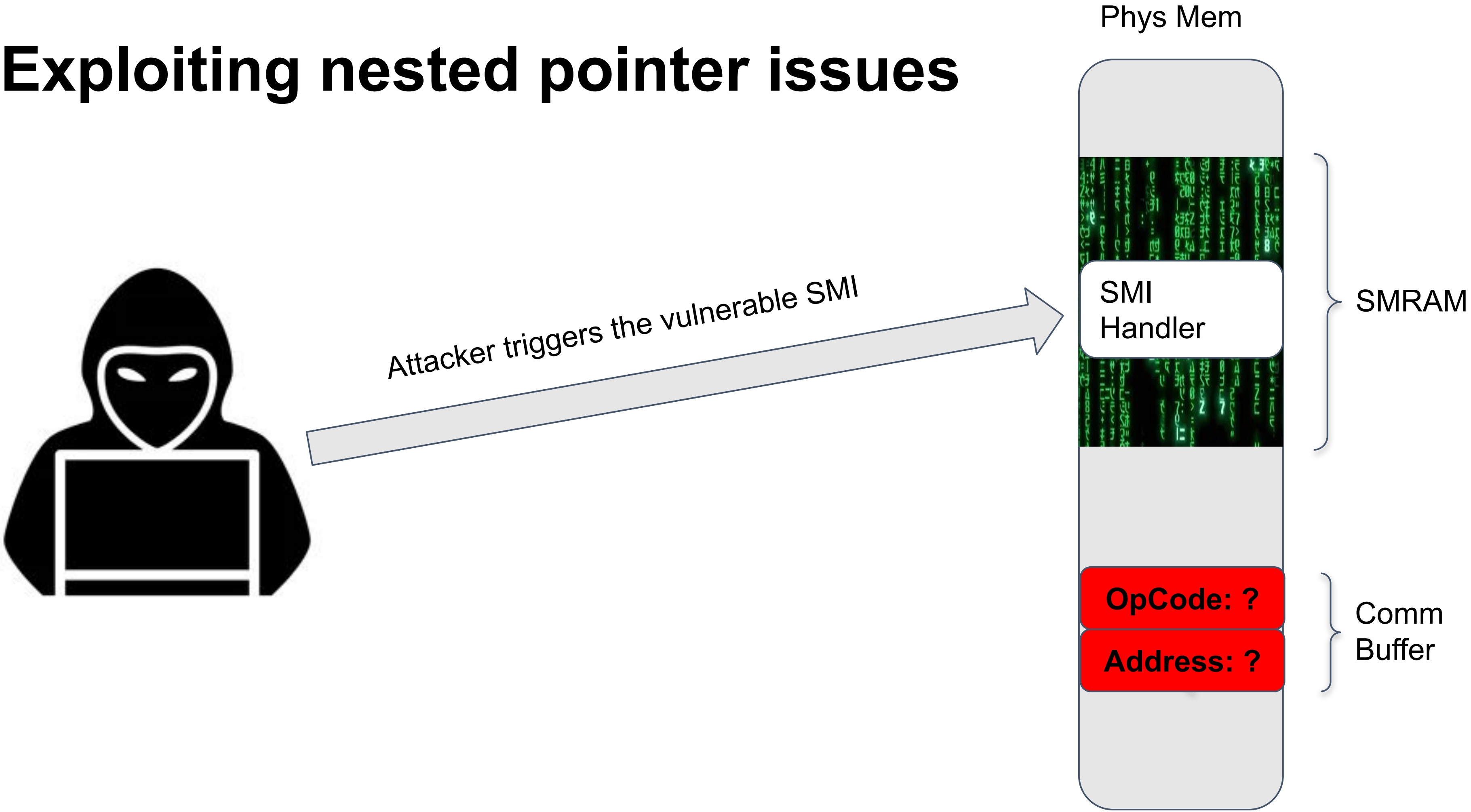
Exploiting nested pointer issues



Exploiting nested pointer issues

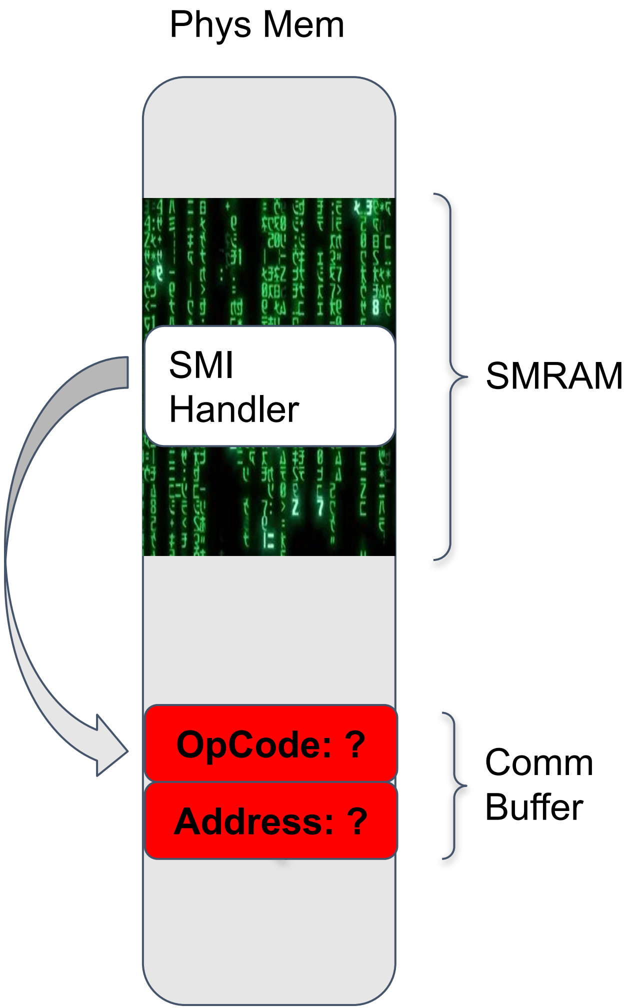


Exploiting nested pointer issues



Exploiting nested pointer issues

Handler inspects opcode field



Phys Mem

SMI Handler

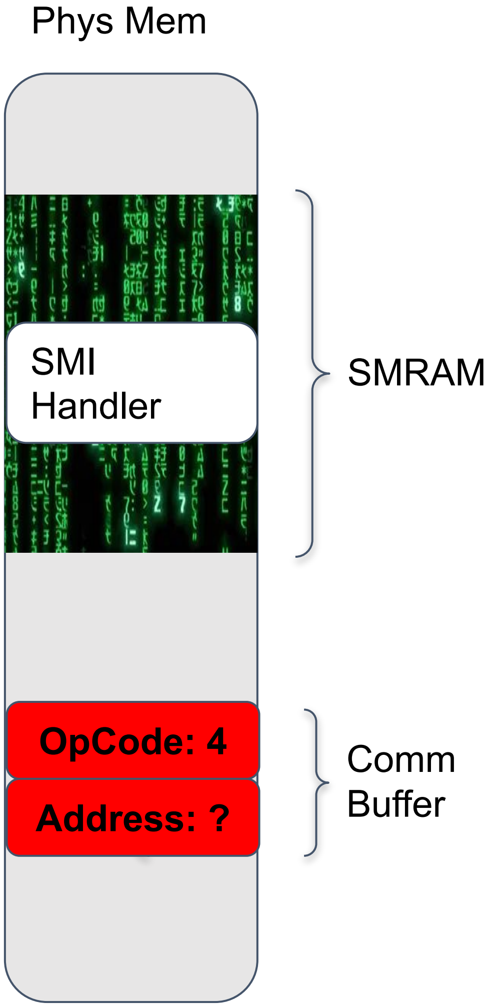
SMRAM

OpCode: ?

Address: ?

Comm Buffer

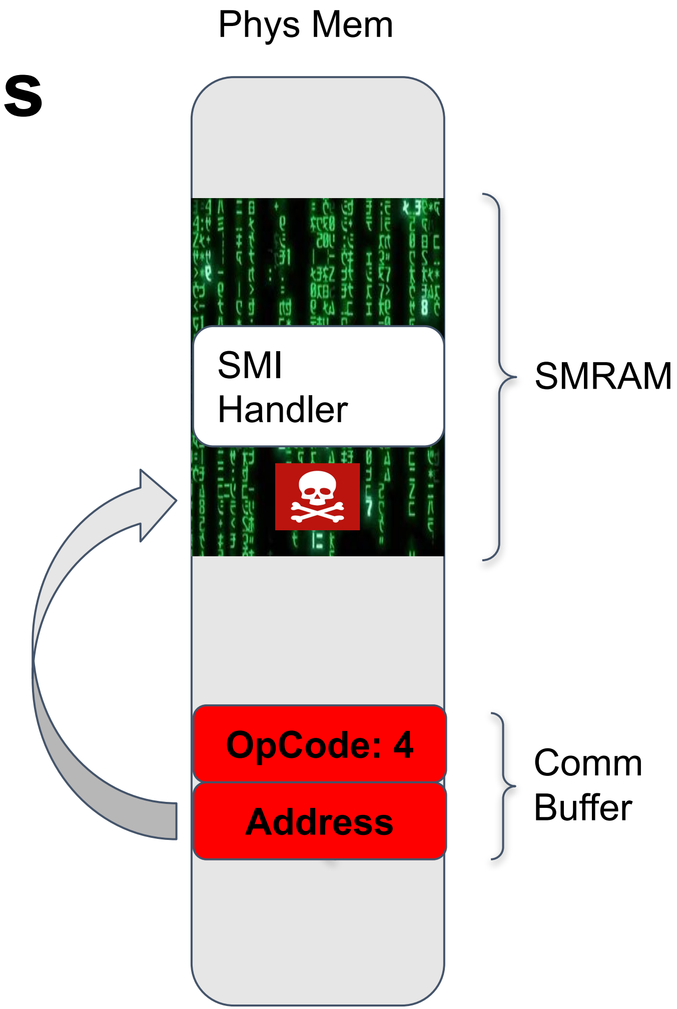
Exploiting nested pointer issues



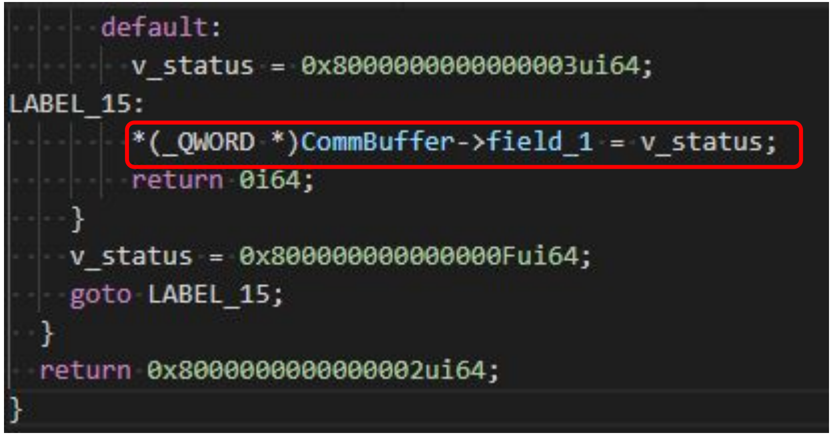
An invalid opcode values will force the handler to fallback into the `default` case

Exploiting nested pointer issues

Address is also attacker controlled, so we make it point to SMRAM



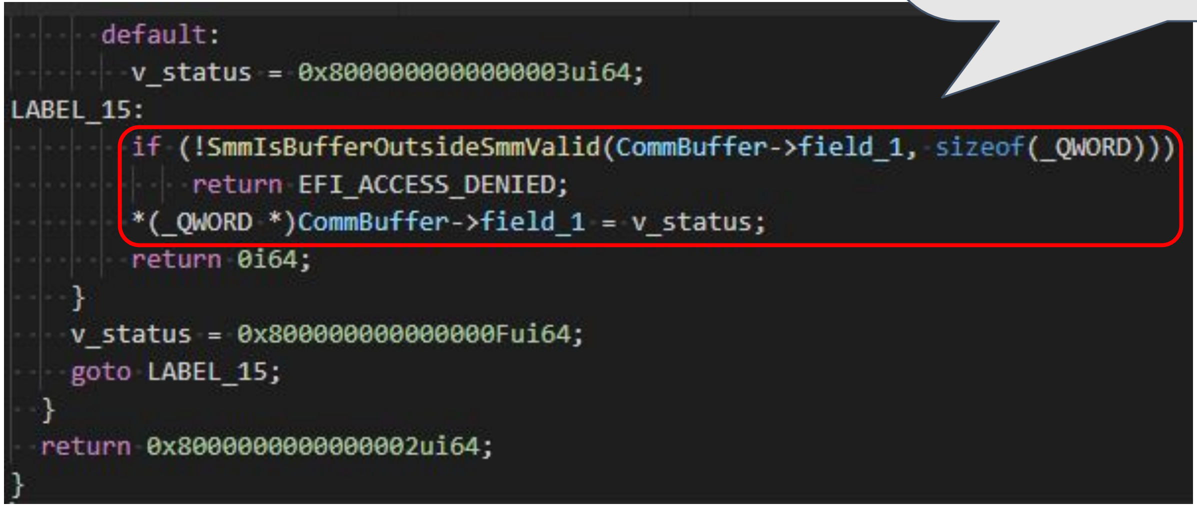
❌



```
            default:
              v_status = 0x8000000000000003ui64;
LABEL_15:
              *(_QWORD *)CommBuffer->field_1 = v_status;
              return 0i64;
        }
        v_status = 0x800000000000000Fui64;
        goto LABEL_15;
      }
      return 0x8000000000000002ui64;
}
```

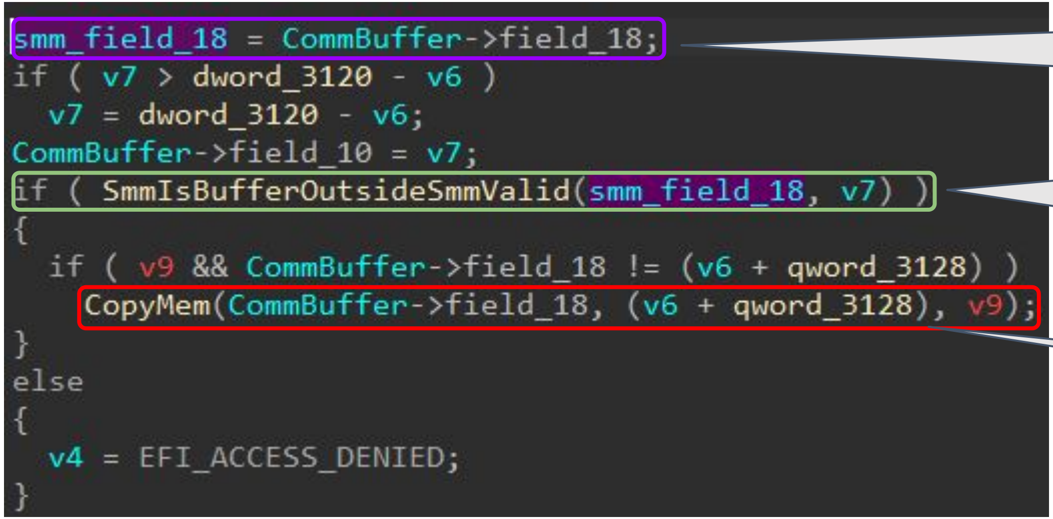
Handlers are expected to call **`SmmIsBufferOutsideSmmValid()`** to make sure client supplied pointers do not overlap with SMRAM

✅



```
            default:
              v_status = 0x8000000000000003ui64;
LABEL_15:
              if (!SmmIsBufferOutsideSmmValid(CommBuffer->field_1, sizeof(_QWORD)))
                  return EFI_ACCESS_DENIED;
              *(_QWORD *)CommBuffer->field_1 = v_status;
              return 0i64;
        }
        v_status = 0x800000000000000Fui64;
        goto LABEL_15;
      }
      return 0x8000000000000002ui64;
}
```

#3: Double-fetches from the CommBuffer



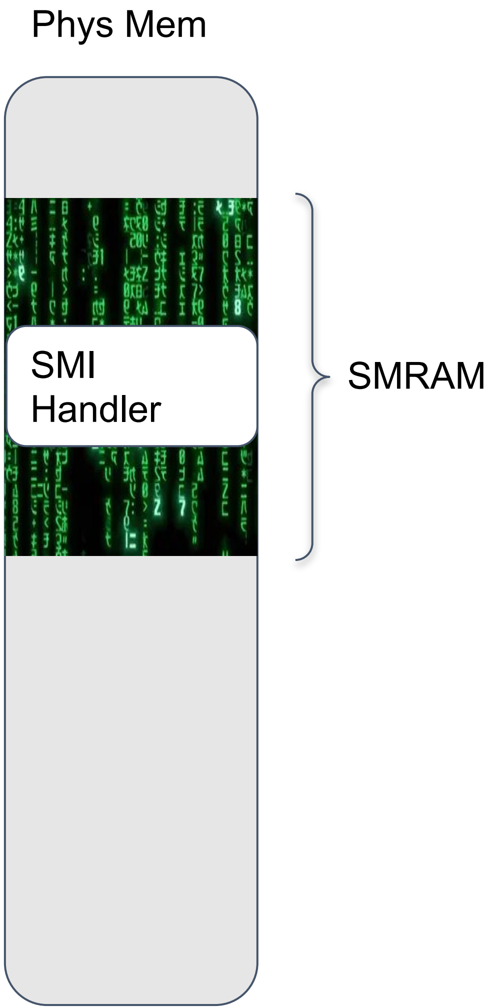
```
smm_field_18 = CommBuffer->field_18;
if ( v7 > dword_3120 - v6 )
  v7 = dword_3120 - v6;
CommBuffer->field_10 = v7;
if ( SmmIsBufferOutsideSmmValid(smm_field_18, v7) )
{
  if ( v9 && CommBuffer->field_18 != (v6 + qword_3128) )
    CopyMem(CommBuffer->field_18, (v6 + qword_3128), v9);
}
else
{
  v4 = EFI_ACCESS_DENIED;
}
```

CommBuffer->field_18 (not in SMRAM) is copied to a local variable in SMRAM

Handler checks that the copied pointer does not overlap with SMRAM

Memory is copied using the original pointer from the *CommBuffer*

Exploiting TOCTOU issues

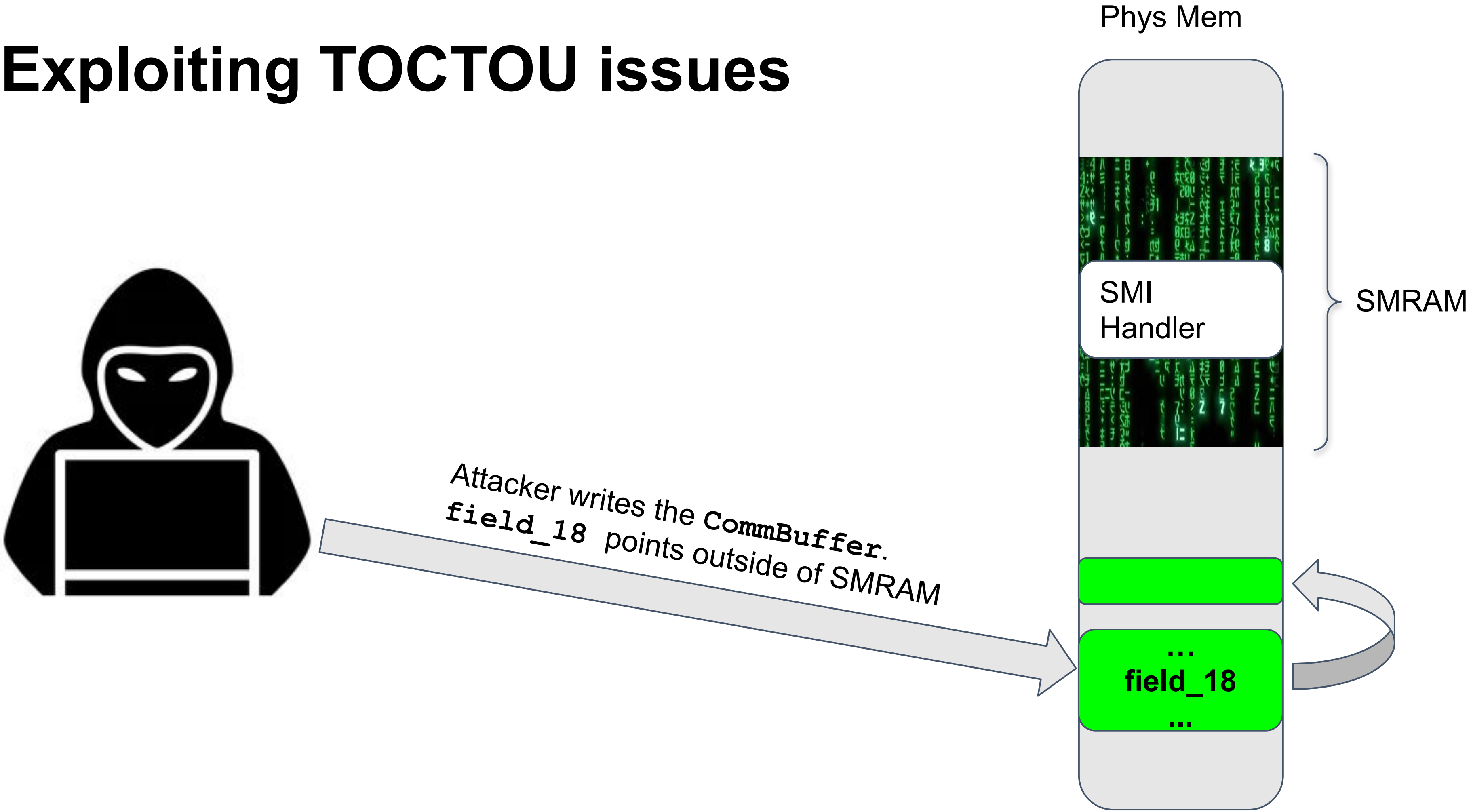


Phys Mem

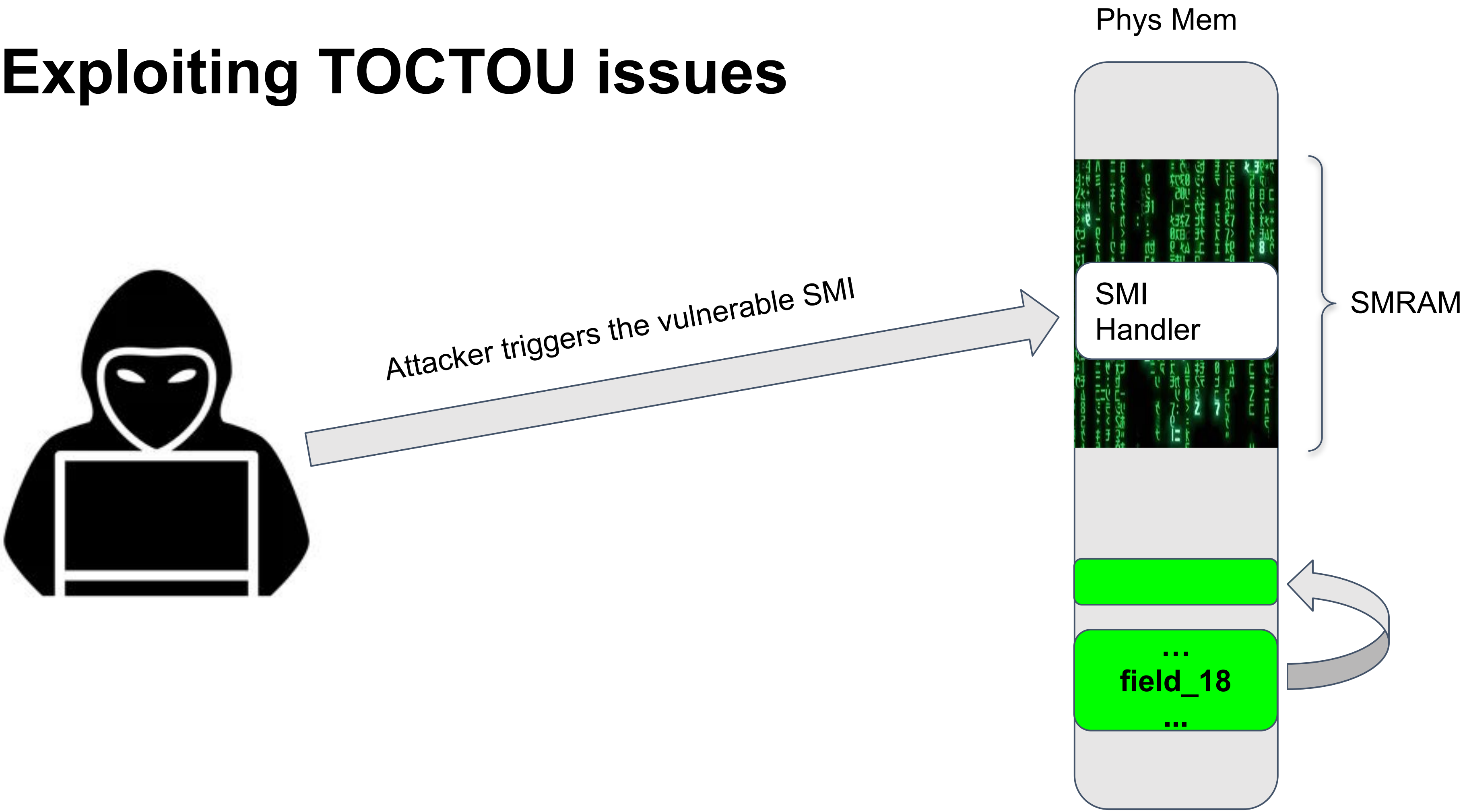
SMI Handler

SMRAM

Exploiting TOCTOU issues

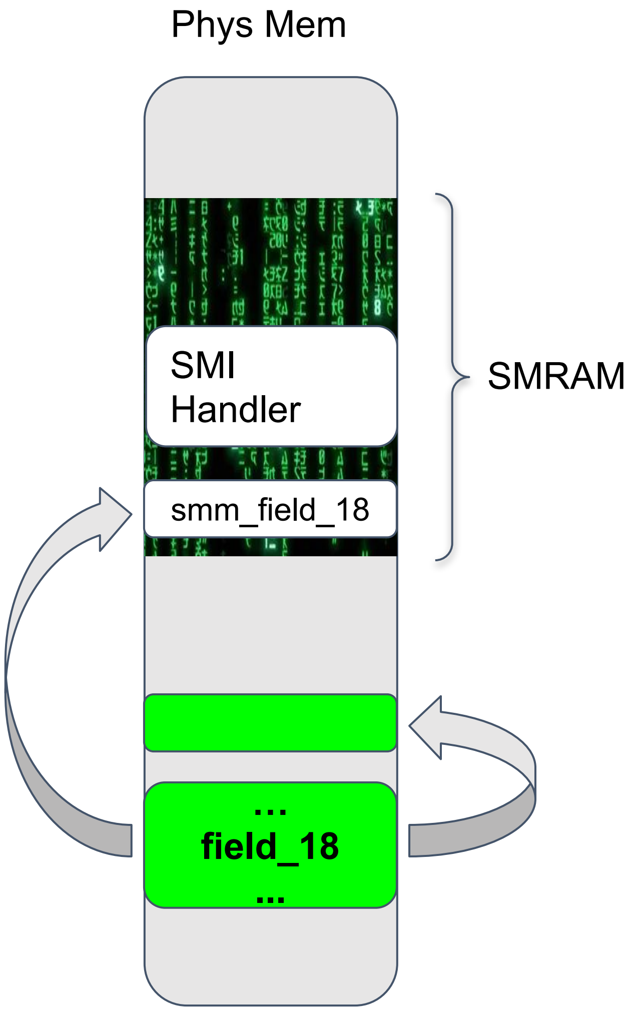

Phys Mem
SMI Handler
SMRAM
Attacker writes the CommBuffer. field_18 points outside of SMRAM
...
field_18
...

Exploiting TOCTOU issues



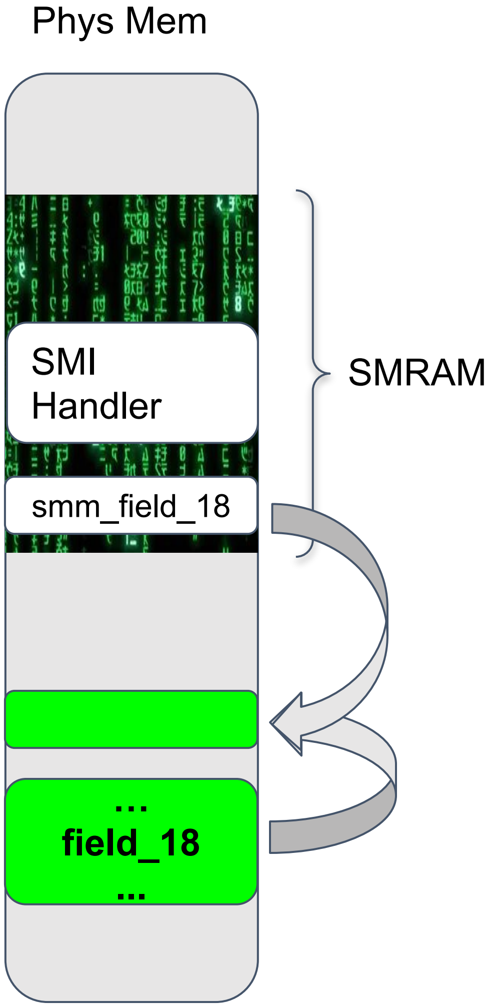
Exploiting TOCTOU issues

`CommBuffer->field_18`
is copied into a local
variable in SMRAM



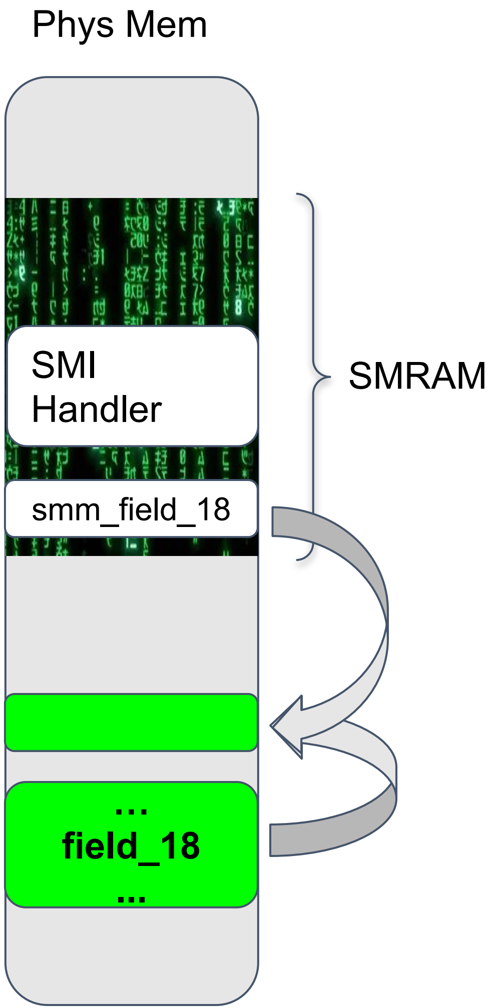
Exploiting TOCTOU issues

Both copies point to the same address



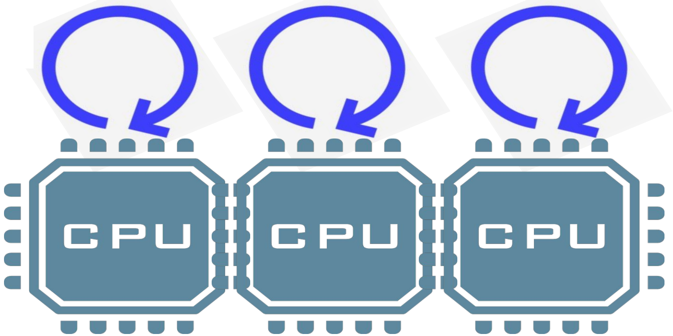
Exploiting TOCTOU issues

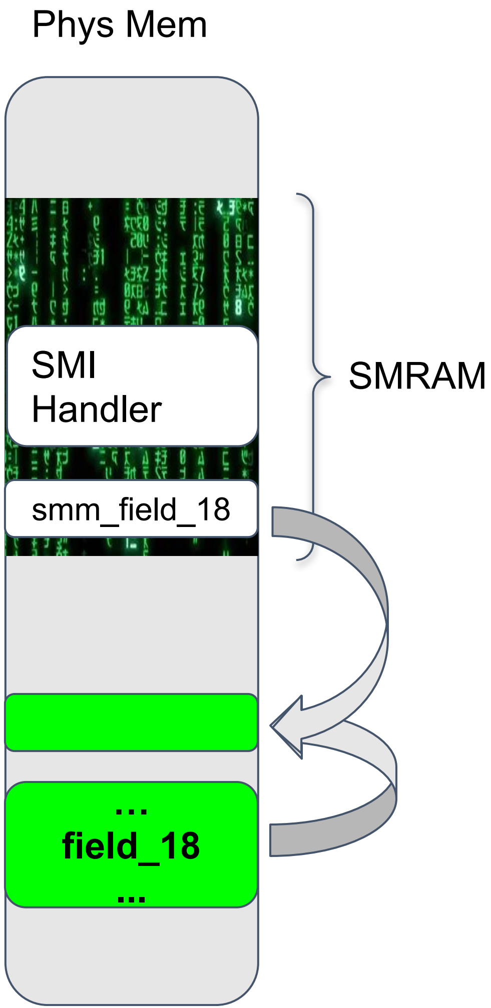
✓ **`SmmIsBufferOutsideSmmValid`** is called to make sure **`smm_field_18`** does not point to SMRAM



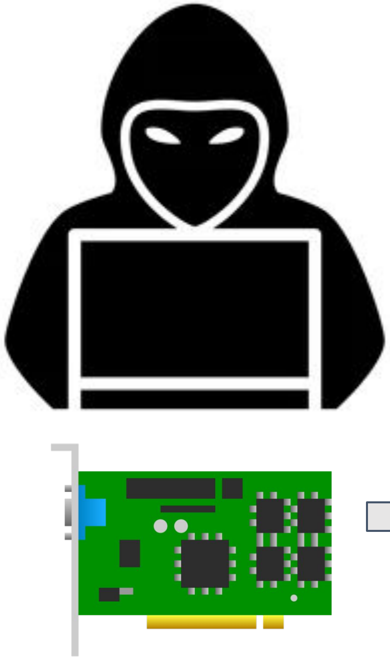
Exploiting TOCTOU issues

While one CPU executes the SMI handler, the other CPUs wait for it to finish in SMM (rendezvous)

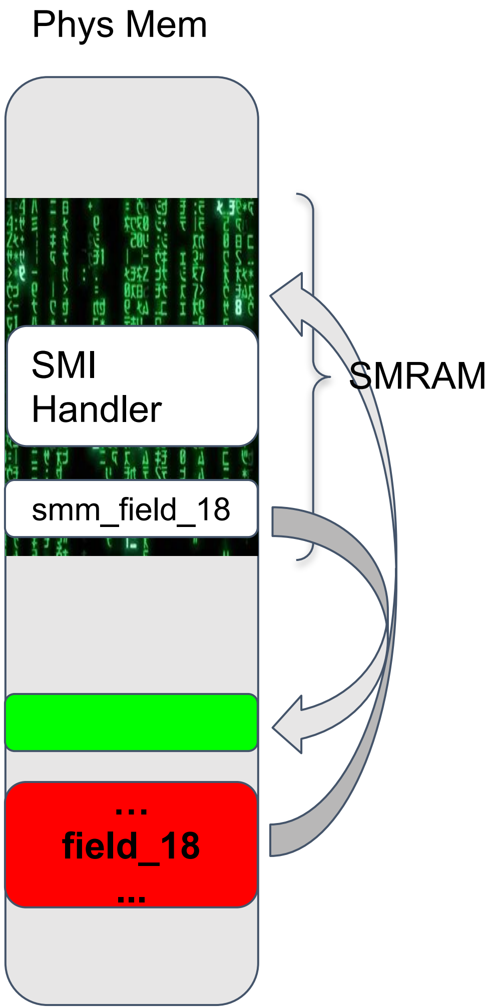




Exploiting TOCTOU issues

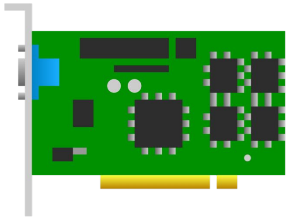


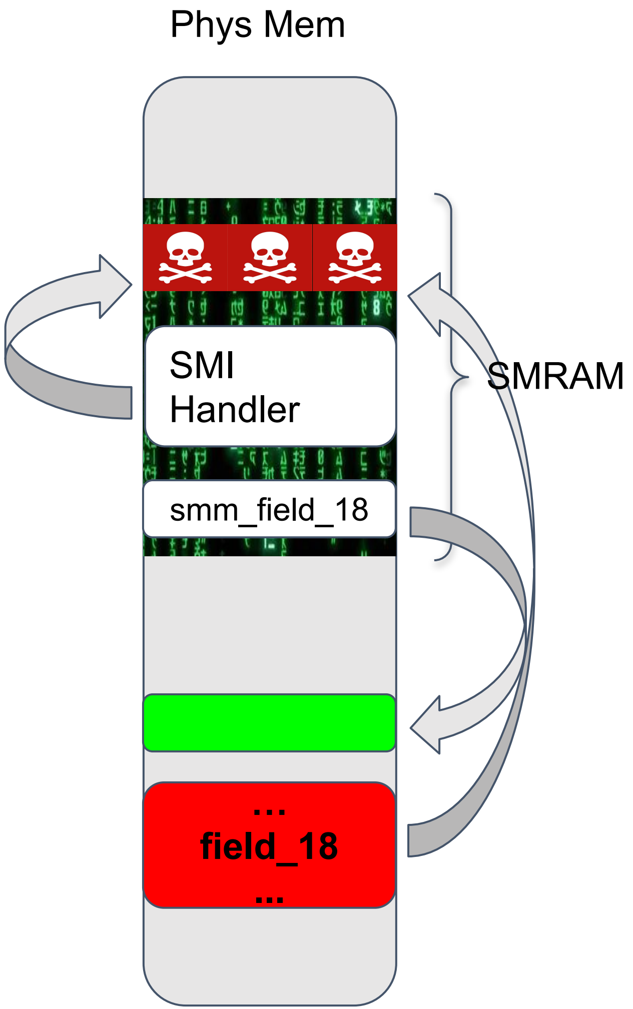
While the SMI handler executes, a DMA attack modifies **`CommBuffer->field_18`** to point to SMRAM



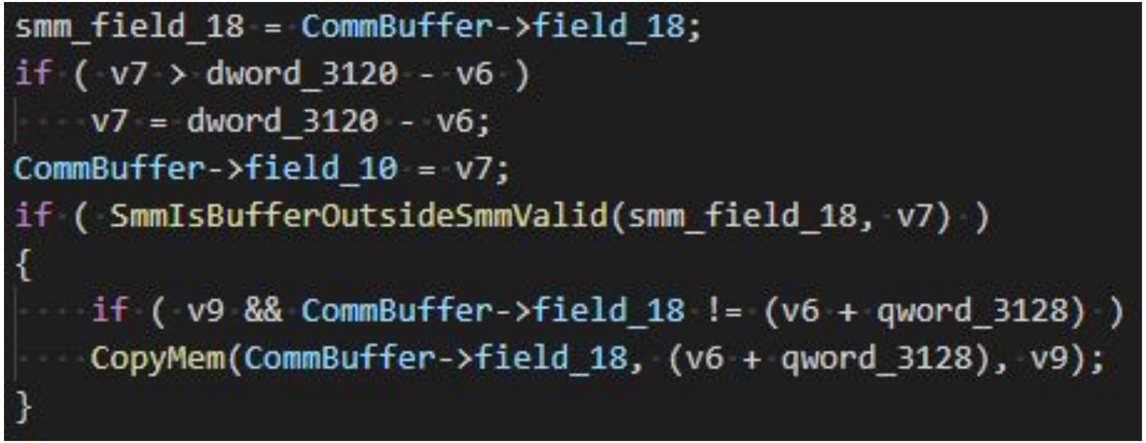
Exploiting TOCTOU issues

Handler calls
`CopyMem(CommBuffer->field_18, …)`



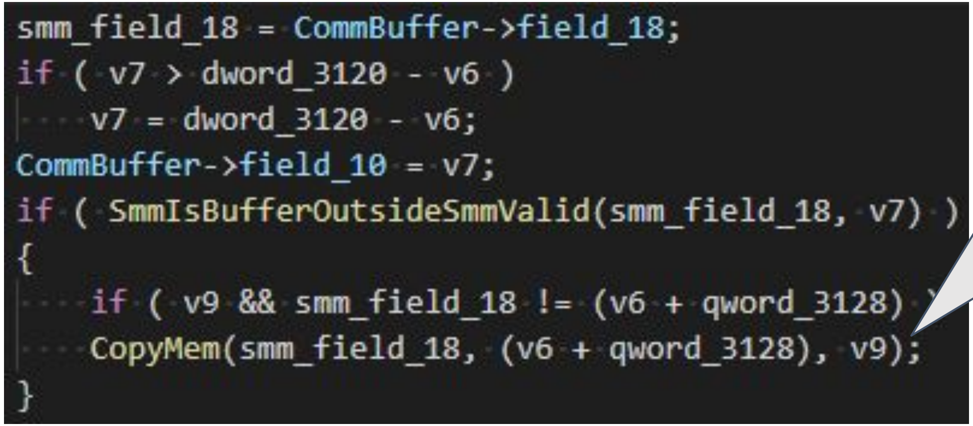


❌



```
smm_field_18 = CommBuffer->field_18;
if ( v7 > dword_3120 - v6 )
    v7 = dword_3120 - v6;
CommBuffer->field_10 = v7;
if ( SmmIsBufferOutsideSmmValid(smm_field_18, v7) )
{
    if ( v9 && CommBuffer->field_18 != (v6 + qword_3128) )
    CopyMem(CommBuffer->field_18, (v6 + qword_3128), v9);
}
```

✅



```
smm_field_18 = CommBuffer->field_18;
if ( v7 > dword_3120 - v6 )
    v7 = dword_3120 - v6;
CommBuffer->field_10 = v7;
if ( SmmIsBufferOutsideSmmValid(smm_field_18, v7) )
{
    if ( v9 && smm_field_18 != (v6 + qword_3128) )
    CopyMem(smm_field_18, (v6 + qword_3128), v9);
}
```

Double fetches from the `CommBuffer` are dangerous!

Handlers are expected to copy members of interest into SMRAM and use only the copy henceforth

Using Brick to automatically hunt SMM bugs

General

- Brick is an **automated, static** analysis tool for hunting SMM vulnerabilities

- Based on IDA
  - Rich ecosystem
  - Higher level analysis via the Hex-Rays decompiler

- Demo time!

Phases

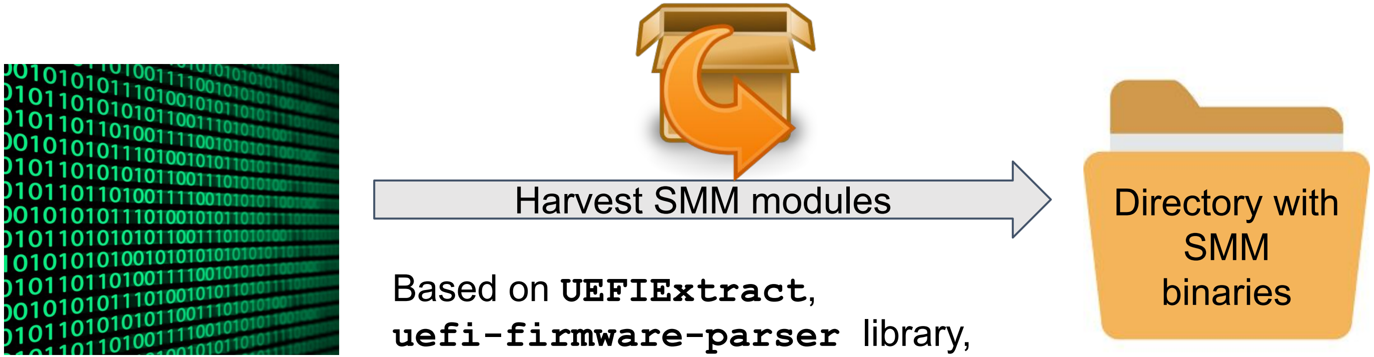
Harvest phase

Analysis phase

Summary phase

Harvest phase

- Extracts all the SMM binaries from the input file



Harvest SMM modules

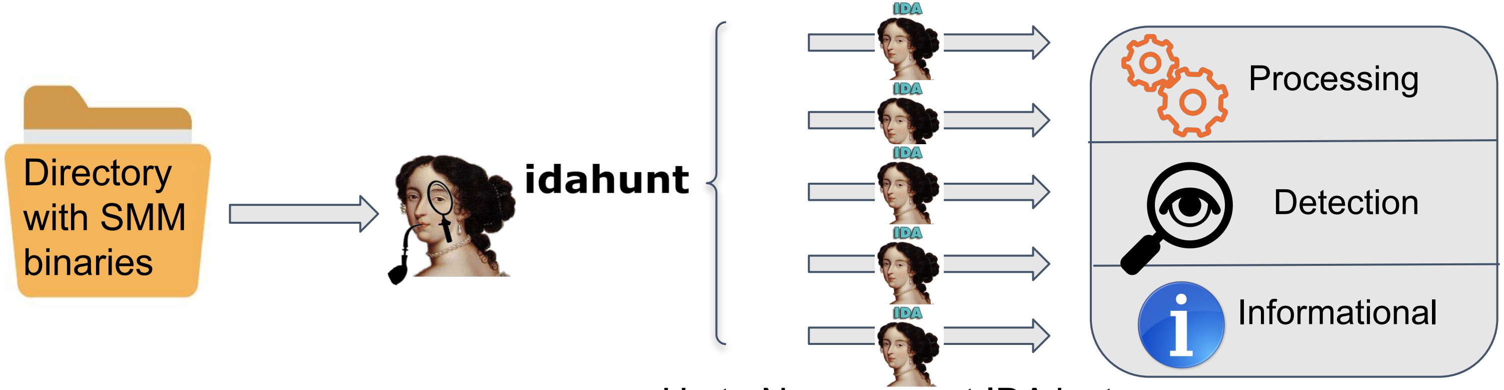
Directory with SMM binaries

Directory, SPI dump, capsule update, FV, BIOS image etc.

Based on **`UEFIExtract`**, **`uefi-firmware-parser`** library, etc.

Analysis phase

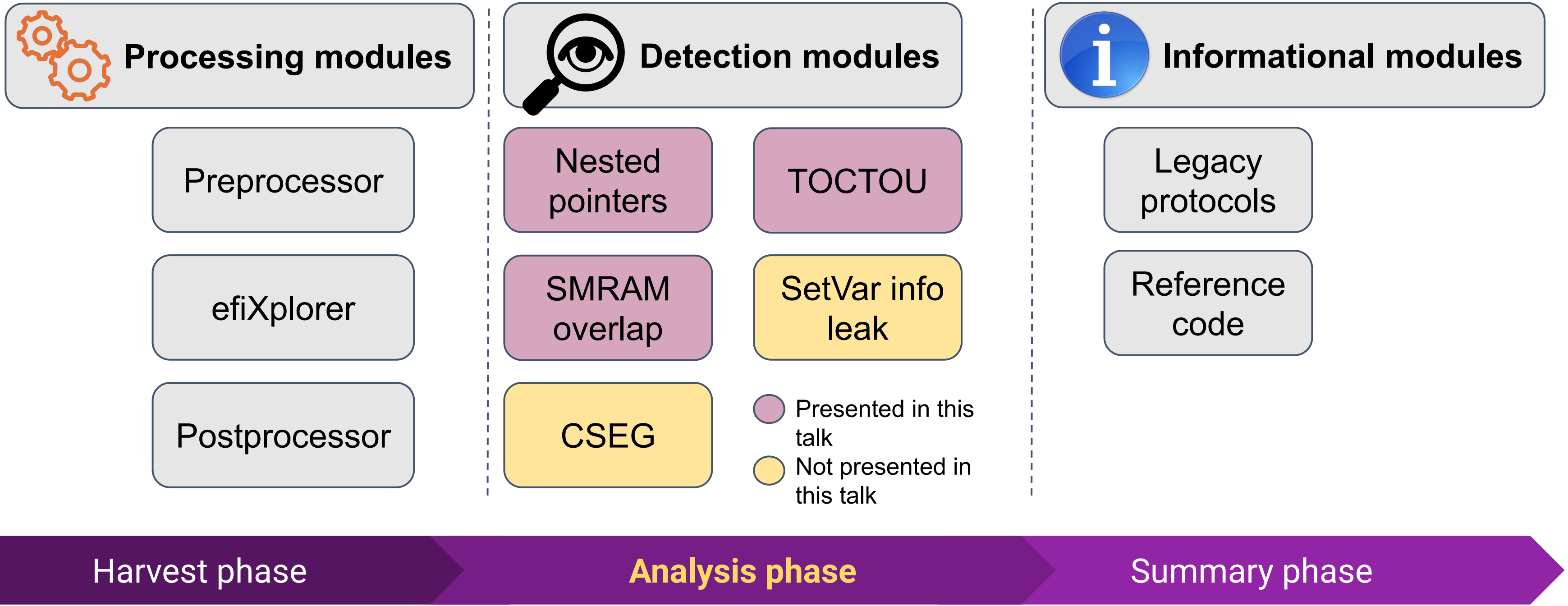
- Each SMM image is opened in IDA
- Runs a bunch of modules against each SMM binary:
  - Processing modules
  - Detection modules
  - Informational modules
- Uses **`idahunt`** to parallelize the process



Up to N concurrent IDA instances

Brick modules

- Implemented as **`IDAPython`** scripts
- Written on top of the **`Bip`** framework



Processing modules

Preprocessor

efiXplorer

Postprocessor

Detection modules

Nested pointers

TOCTOU

SMRAM overlap

SetVar info leak

CSEG

🟣 Presented in this talk

🟡 Not presented in this talk

Informational modules

Legacy protocols

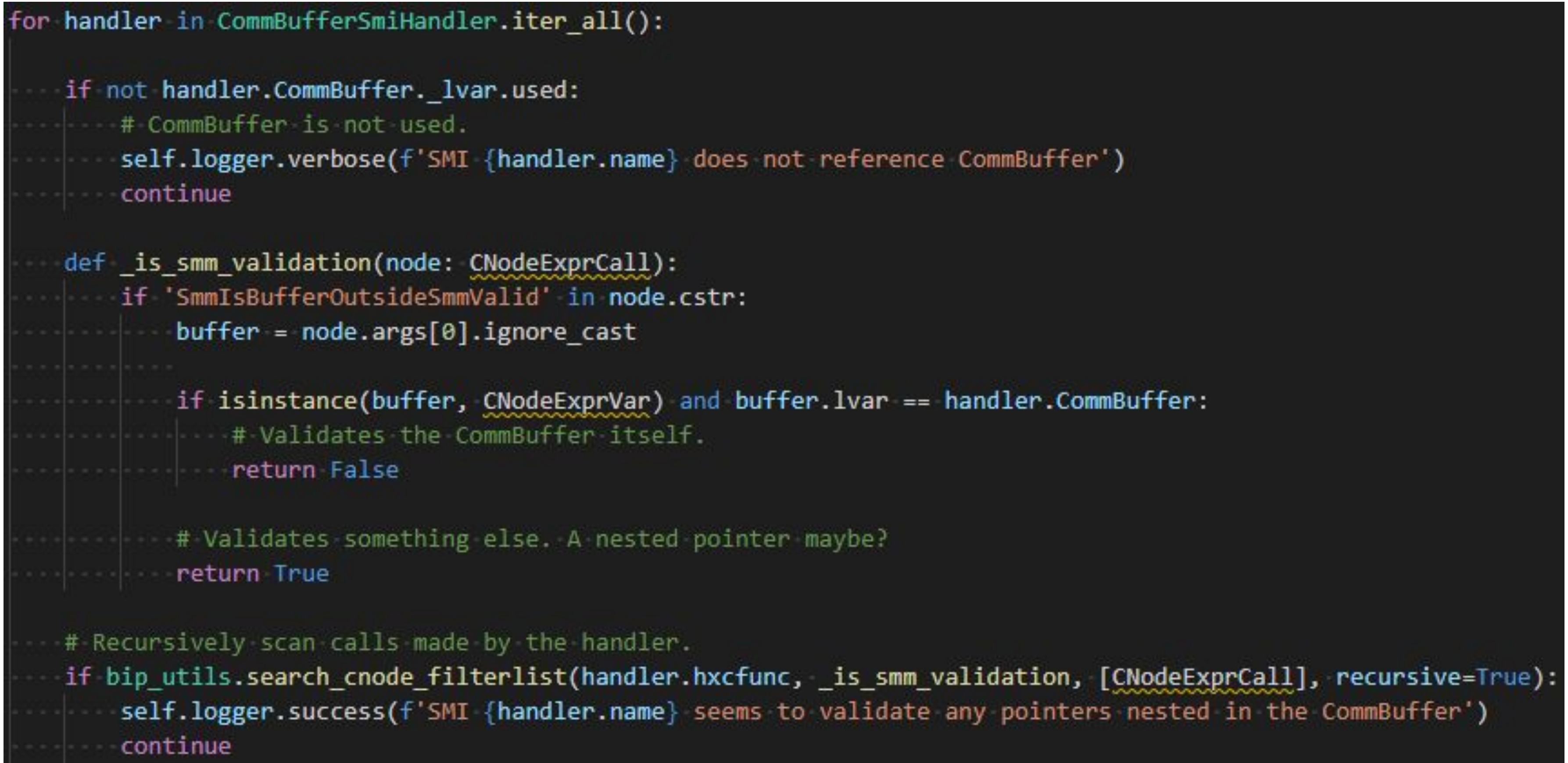
Reference code

Harvest phase

Analysis phase

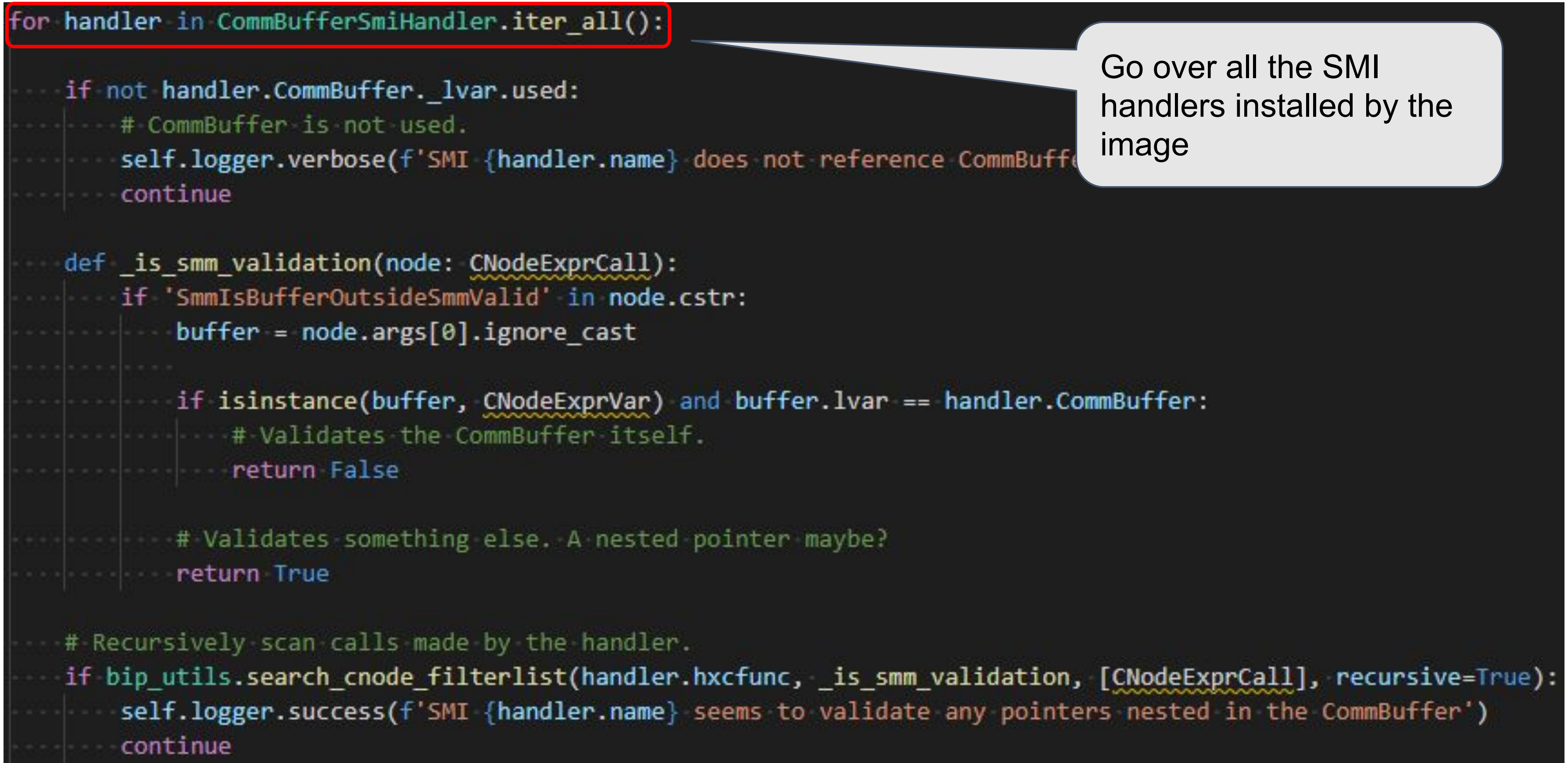
Summary phase

Detection heuristic - unsanitized nested pointers



```
for handler in CommBufferSmiHandler.iter_all():

    if not handler.CommBuffer._lvar.used:
        # CommBuffer is not used.
        self.logger.verbose(f'SMI {handler.name} does not reference CommBuffer')
        continue

    def _is_smm_validation(node: CNodeExprCall):
        if 'SmmIsBufferOutsideSmmValid' in node.cstr:
            buffer = node.args[0].ignore_cast

            if isinstance(buffer, CNodeExprVar) and buffer.lvar == handler.CommBuffer:
                # Validates the CommBuffer itself.
                return False

            # Validates something else. A nested pointer maybe?
            return True

    # Recursively scan calls made by the handler.
    if bip_utils.search_cnode_filterlist(handler.hxcfunc, _is_smm_validation, [CNodeExprCall], recursive=True):
        self.logger.success(f'SMI {handler.name} seems to validate any pointers nested in the CommBuffer')
        continue
```

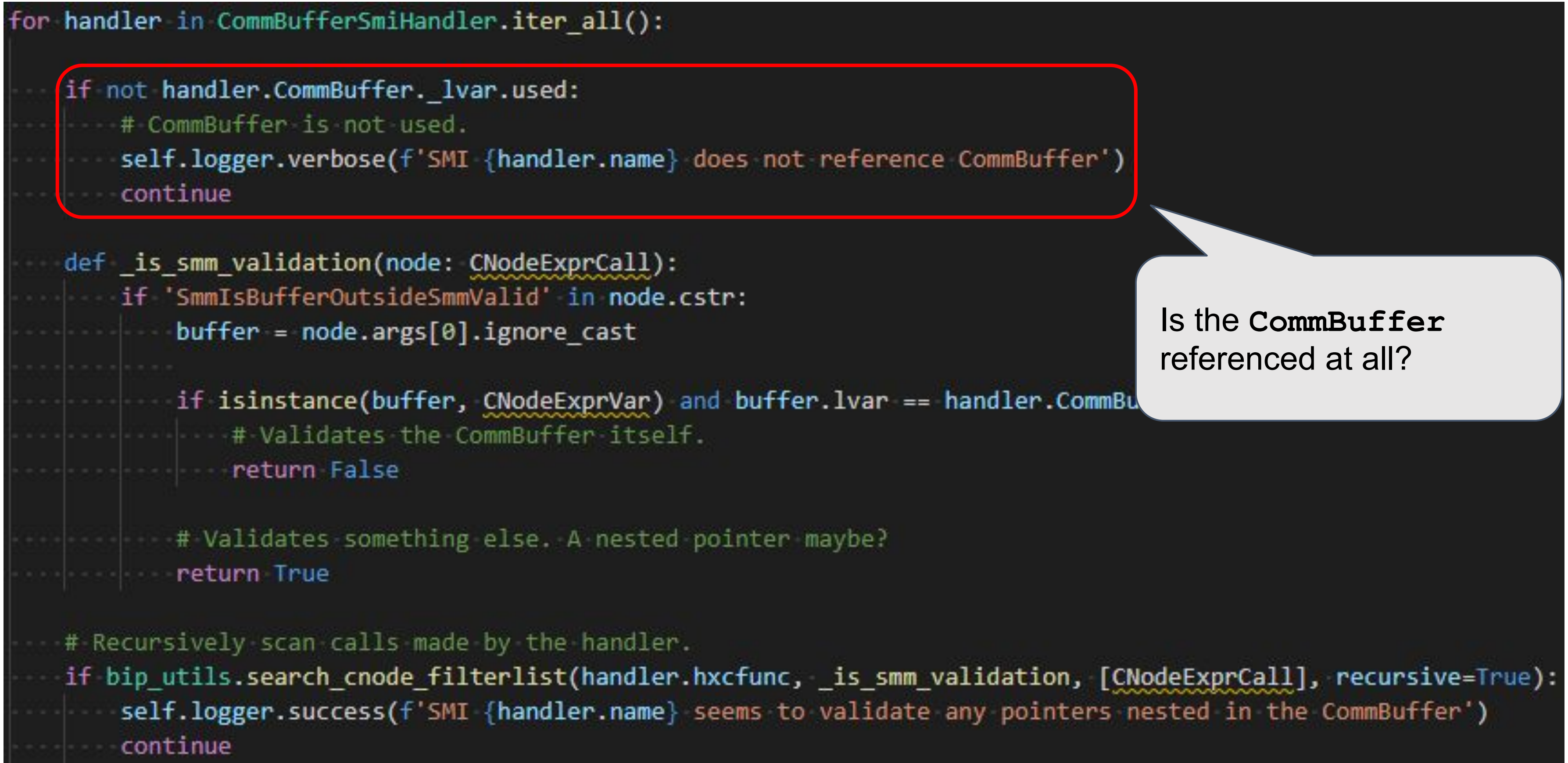
Detection heuristic - unsanitized nested pointers



```
for handler in CommBufferSmiHandler.iter_all():

    if not handler.CommBuffer._lvar.used:
        # CommBuffer is not used.
        self.logger.verbose(f'SMI {handler.name} does not reference CommBuffe
        continue

    def _is_smm_validation(node: CNodeExprCall):
        if 'SmmIsBufferOutsideSmmValid' in node.cstr:
            buffer = node.args[0].ignore_cast

            if isinstance(buffer, CNodeExprVar) and buffer.lvar == handler.CommBuffer:
                # Validates the CommBuffer itself.
                return False

            # Validates something else. A nested pointer maybe?
            return True

    # Recursively scan calls made by the handler.
    if bip_utils.search_cnode_filterlist(handler.hxcfunc, _is_smm_validation, [CNodeExprCall], recursive=True):
        self.logger.success(f'SMI {handler.name} seems to validate any pointers nested in the CommBuffer')
        continue
```

Go over all the SMI handlers installed by the image

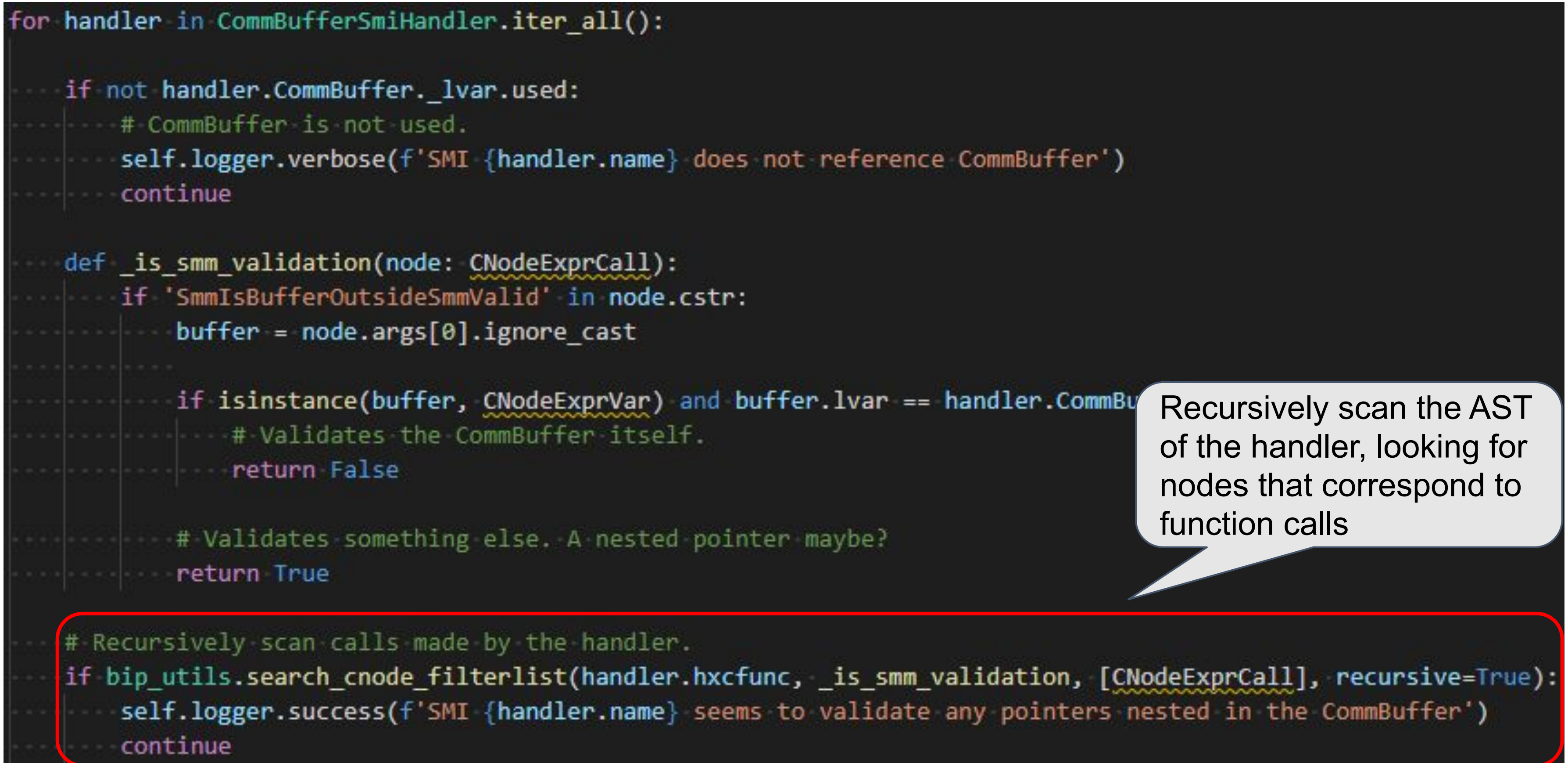
Detection heuristic - unsanitized nested pointers



```
for handler in CommBufferSmiHandler.iter_all():

    if not handler.CommBuffer._lvar.used:
        # CommBuffer is not used.
        self.logger.verbose(f'SMI {handler.name} does not reference CommBuffer')
        continue

    def _is_smm_validation(node: CNodeExprCall):
        if 'SmmIsBufferOutsideSmmValid' in node.cstr:
            buffer = node.args[0].ignore_cast

            if isinstance(buffer, CNodeExprVar) and buffer.lvar == handler.CommBu
                # Validates the CommBuffer itself.
                return False

            # Validates something else. A nested pointer maybe?
            return True

    # Recursively scan calls made by the handler.
    if bip_utils.search_cnode_filterlist(handler.hxcfunc, _is_smm_validation, [CNodeExprCall], recursive=True):
        self.logger.success(f'SMI {handler.name} seems to validate any pointers nested in the CommBuffer')
        continue
```

Is the **`CommBuffer`** referenced at all?

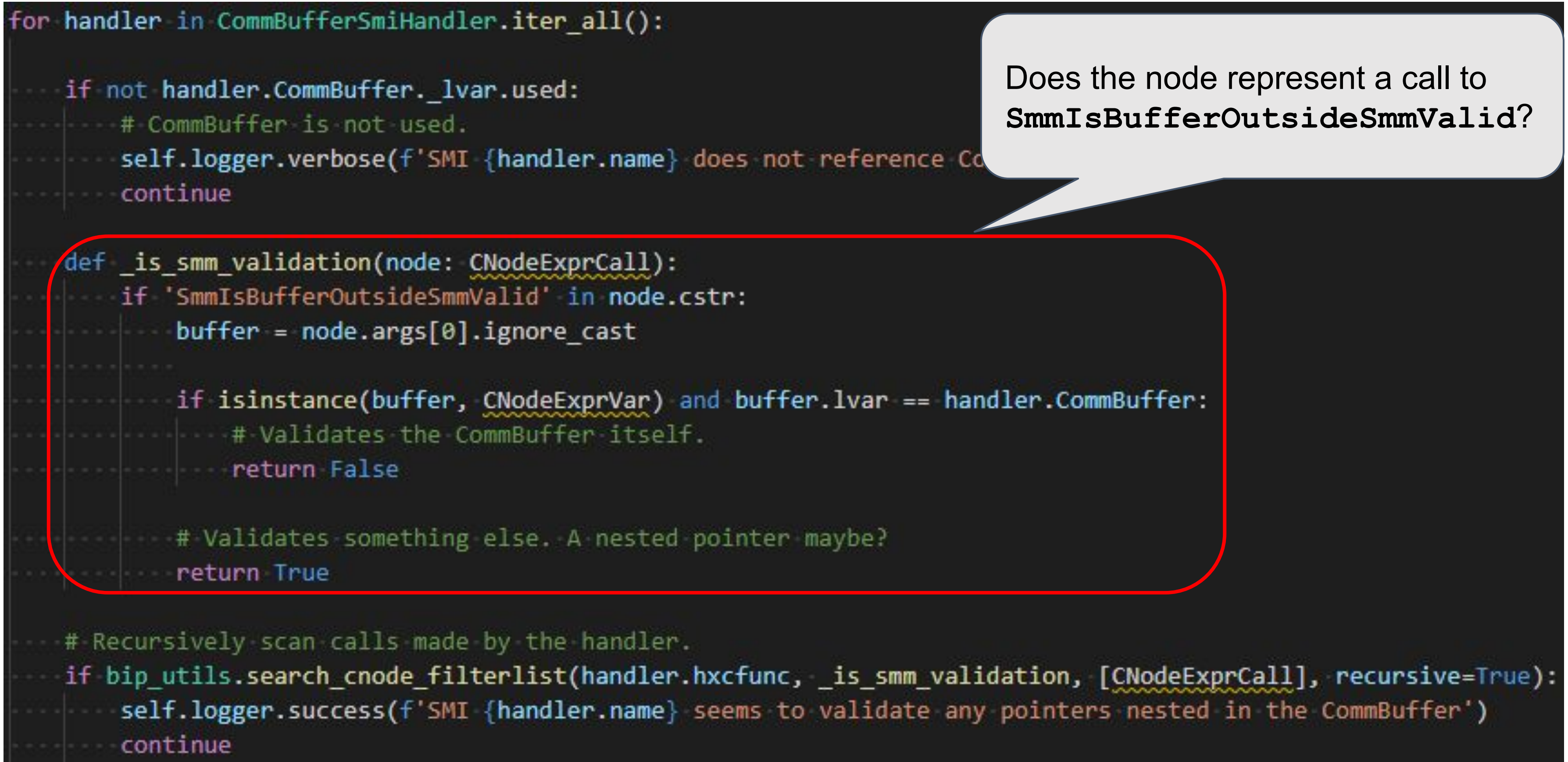
Detection heuristic - unsanitized nested pointers



```
for handler in CommBufferSmiHandler.iter_all():

    if not handler.CommBuffer._lvar.used:
        # CommBuffer is not used.
        self.logger.verbose(f'SMI {handler.name} does not reference CommBuffer')
        continue

    def _is_smm_validation(node: CNodeExprCall):
        if 'SmmIsBufferOutsideSmmValid' in node.cstr:
            buffer = node.args[0].ignore_cast

            if isinstance(buffer, CNodeExprVar) and buffer.lvar == handler.CommBu
                # Validates the CommBuffer itself.
                return False

            # Validates something else. A nested pointer maybe?
            return True

    # Recursively scan calls made by the handler.
    if bip_utils.search_cnode_filterlist(handler.hxcfunc, _is_smm_validation, [CNodeExprCall], recursive=True):
        self.logger.success(f'SMI {handler.name} seems to validate any pointers nested in the CommBuffer')
        continue
```

Recursively scan the AST of the handler, looking for nodes that correspond to function calls

Detection heuristic - unsanitized nested pointers



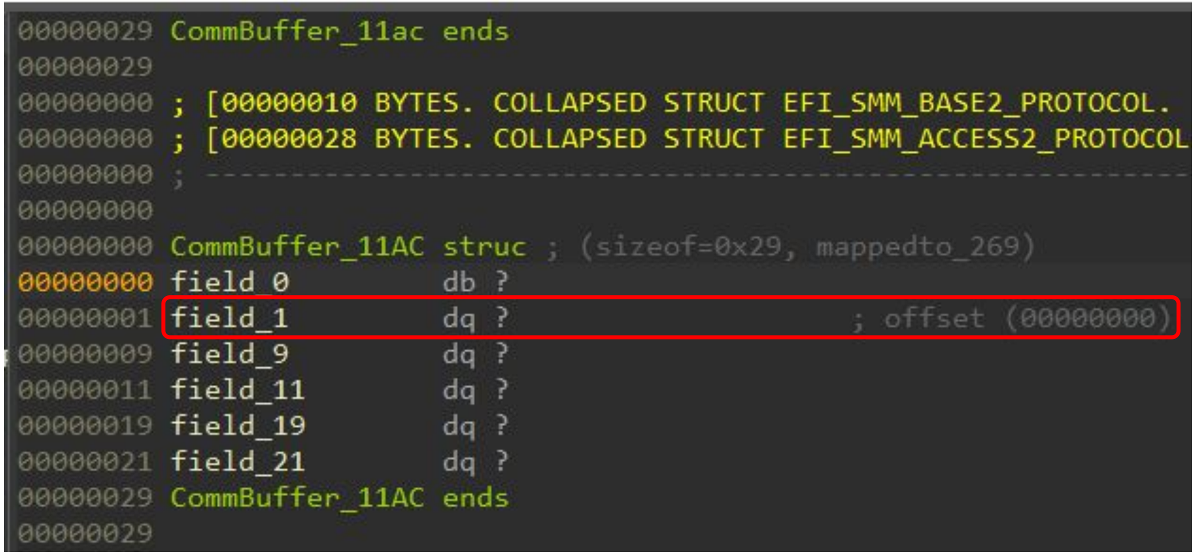
```
for handler in CommBufferSmiHandler.iter_all():

    if not handler.CommBuffer._lvar.used:
        # CommBuffer is not used.
        self.logger.verbose(f'SMI {handler.name} does not reference Co
        continue

    def _is_smm_validation(node: CNodeExprCall):
        if 'SmmIsBufferOutsideSmmValid' in node.cstr:
            buffer = node.args[0].ignore_cast

            if isinstance(buffer, CNodeExprVar) and buffer.lvar == handler.CommBuffer:
                # Validates the CommBuffer itself.
                return False

            # Validates something else. A nested pointer maybe?
            return True

    # Recursively scan calls made by the handler.
    if bip_utils.search_cnode_filterlist(handler.hxcfunc, _is_smm_validation, [CNodeExprCall], recursive=True):
        self.logger.success(f'SMI {handler.name} seems to validate any pointers nested in the CommBuffer')
        continue
```

Does the node represent a call to **`SmmIsBufferOutsideSmmValid`**?

Improving detection

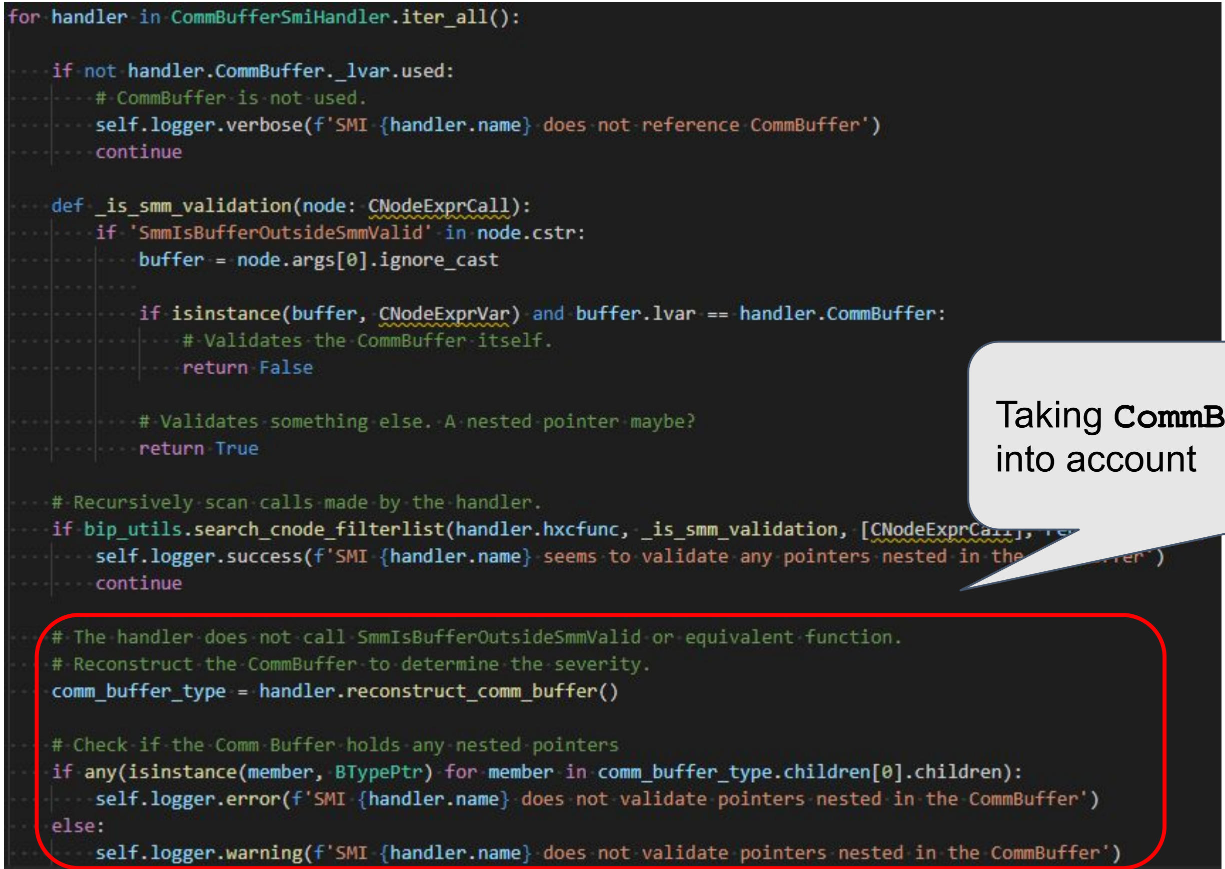
- Reconstructing the layout of the **`CommBuffer`** allows us to determine whether or not it holds nested pointers

- Can be done via **`HexRaysCodeXplorer`** https://github.com/REhints/HexRaysCodeXplorer



```
00000029 CommBuffer_11ac ends
00000029
00000000 ; [00000010 BYTES. COLLAPSED STRUCT EFI_SMM_BASE2_PROTOCOL.
00000000 ; [00000028 BYTES. COLLAPSED STRUCT EFI_SMM_ACCESS2_PROTOCOL
00000000 ; ---------------------------------------------------------------
00000000
00000000 CommBuffer_11AC struc ; (sizeof=0x29, mappedto_269)
00000000 field_0         db ?
00000001 field_1         dq ?                       ; offset (00000000)
00000009 field_9         dq ?
00000011 field_11        dq ?
00000019 field_19        dq ?
00000021 field_21        dq ?
00000029 CommBuffer_11AC ends
00000029
```

Improved heuristic - unsanitized nested pointers



```python
for handler in CommBufferSmiHandler.iter_all():

    if not handler.CommBuffer._lvar.used:
        # CommBuffer is not used.
        self.logger.verbose(f'SMI {handler.name} does not reference CommBuffer')
        continue

    def _is_smm_validation(node: CNodeExprCall):
        if 'SmmIsBufferOutsideSmmValid' in node.cstr:
            buffer = node.args[0].ignore_cast

            if isinstance(buffer, CNodeExprVar) and buffer.lvar == handler.CommBuffer:
                # Validates the CommBuffer itself.
                return False

            # Validates something else. A nested pointer maybe?
            return True

    # Recursively scan calls made by the handler.
    if bip_utils.search_cnode_filterlist(handler.hxcfunc, _is_smm_validation, [CNodeExprCall], re
        self.logger.success(f'SMI {handler.name} seems to validate any pointers nested in the        ffer')
        continue

    # The handler does not call SmmIsBufferOutsideSmmValid or equivalent function.
    # Reconstruct the CommBuffer to determine the severity.
    comm_buffer_type = handler.reconstruct_comm_buffer()

    # Check if the Comm Buffer holds any nested pointers
    if any(isinstance(member, BTypePtr) for member in comm_buffer_type.children[0].children):
        self.logger.error(f'SMI {handler.name} does not validate pointers nested in the CommBuffer')
    else:
        self.logger.warning(f'SMI {handler.name} does not validate pointers nested in the CommBuffer')
```

Taking **`CommBuffer`** reconstruction into account

A word of false { positives, negatives }

- Simple heuristics have many advantages, but also imply that false positives and false negatives will occur from time to time

- False positives (misleading alerts)
  - Brick is just a helper tool, so manual examination of the results is a must
  - A small degree of false positives is acceptable

- False negatives (misses)
  - Main use case is scanning the entire firmware image
  - Finding even a subset of all vulnerabilities might be good enough to compromise SMM

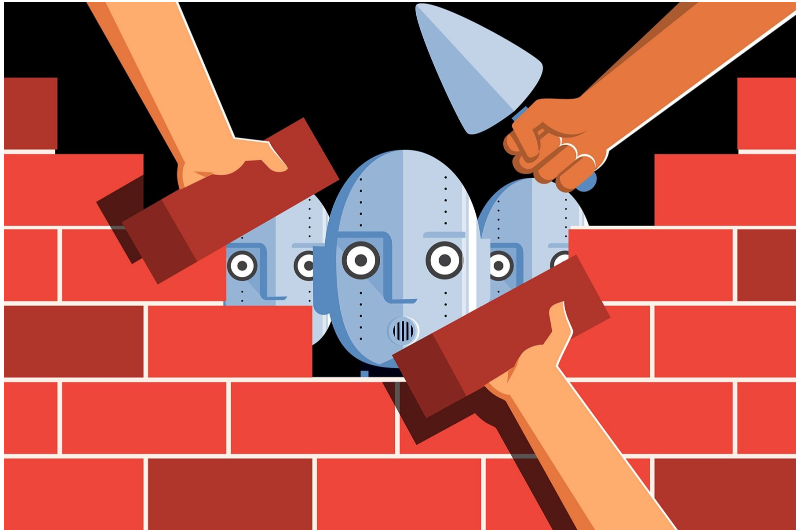
Results (so far)

- Two CVEs from Lenovo
  - CVE-2021-3599: A potential vulnerability in the SMI callback function used to access flash device in some ThinkPad models may allow an attacker with local access and elevated privileges to execute arbitrary code.

  - CVE-2021-3786: A potential vulnerability in the SMI callback function used in CSME configuration could be used to leak out data out of the SMRAM range.

- About a dozen of other vulnerabilities in various stages of the disclosure process
  - Affecting all major vendors and OEMs
  - Some affect the reference code shared between multiple vendors

Future work



- Add more detection modules
- Improve reliability
- Reduce running time
- You can contribute too!
  - https://github.com/Sentinel-One/brick

Thank you for your attention!