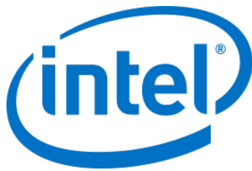

White Paper

A Tour Beyond BIOS Secure SMM Communication in the EFI Developer Kit II

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

Star Zeng
Intel Corporation

April 26, 2016

This paper is for informational purposes only. THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE
ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel, the Intel logo, Intel. leap ahead. and Intel. Leap ahead. logo, and other Intel product name are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright 2016 by Intel Corporation. All rights reserved

Executive Summary

This paper introduces how we can do secure SMM communication in a UEFI BIOS.

Prerequisite
This paper assumes that audience has basic EDKII/UEFI firmware development experience, and basic knowledge of SMM.

Table of Contents

Executive Summary ***3***

Prerequisite 3

Table of Contents ***4***

Table of Figures: 5

Overview ***5***

Introduction to SMM Communication ***5***

Summary 5

SMM Communication in UEFI BIOS ***6***

PI EFI_SMM_COMMUNICATION_PROTOCOL ***6***

UEFI ACPI table ***7***

Pre-defined location ***8***

General purpose register ***9***

Pointer in communication buffer ***10***

Summary 10

Vulnerability of SMM communication ***11***

Attack on SMM communication ***11***

Windows SMM Security Mitigations Table ***13***

Summary 14

Secure SMM Communication ***15***

Protect SMM itself to prevent an attack from the OS or VMM. ***15***

Protect VMM to prevent an attack from a VM. ***15***

Communication Buffer Location Information. ***16***

Communication Buffer Type ***18***

(General purpose register) 19

EDKII_PI_SMM_COMMUNICATION_REGION_TABLE ***19***

CSM SMI handler consideration ***20***

Fixed Communication Buffer Check ***22***

Summary 23

Assumption and Recommendation ***24***

Call for action ***25***

Future work ***26***

Summary 27

Conclusion ***28***

Glossary ***29***

References ***30***

Table of Figures:

Figure 1 – SMM Variable 6
Figure 2 – SMM Communication 7
Figure 3 – SMM ACPI Table 8
Figure 4 – TCG SMM 9
Figure 5 – GP register as communication 9
Figure 6 – Pointer in communication buffer 10
Figure 7 – source: [SMM Attack2] 11
Figure 8 – source: [SMM Attack3] 12
Figure 9 – source: [SMM Attack3] 12
Figure 10 – PI-SMM Monitor resource declaration 16
Figure 11 – Type based fixed communication buffer 17
Figure 12 – PI-SMM Communication region table 19
Figure 13 – Dynamic CommBuffer to Statis CommBuffer 20
Figure 14 – Legacy agent – GP register as pointer 21
Figure 15 – Reverse Thunk solution to help buffer move 21
Figure 16 –memory layout with MemoryTypeInformation enabled 25
Figure 17 – Page table enforced memory layout 26

Overview

The main job of BIOS is to initialize the platform hardware and report information to a generic operating system (OS). Some BIOS's also have runtime code to provide services to the OS, such as UEFI runtime services [UEFI], ACPI ASL code [ACPI], as well as SMI handlers [IA32SDM].

System Management Mode (SMM) is defined in IA CPU architecture. SMM firmware infrastructure is defined in UEFI PI specification volume 4, and SMM core is implemented in EDKII. The goal of SMM is to provide an isolated execution environment, so that it can provide some secure services. SMM has higher privilege than normal OS kernel (ring 0) or event hypervisor.

Introduction to SMM Communication

SMM is invoked through a system management interrupt (SMI). When SMI happened, the processor saves the current state of the processor, and then switches to a separate operating environment defined by a new address space. This behavior is defined in [IA32SDM]. But IA32SDM does not define how to do data communication between non-SMM and SMM. For SMM communication, [UEFI PI Specification] defines EFI_SMM_COMMUNICATION_PROTOCOL and [UEFI] defines SMM Communication ACPI Table. These are 2 generic ways for OS communicate with BIOS SMI handler. OEM BIOS may also define his own way for SMM communication.

Summary

This section provided an overview of the SMM communication.

SMM Communication in UEFI BIOS

In this chapter, we will discuss how SMM communication works in UEFI BIOS.

PI EFI_SMM_COMMUNICATION_PROTOCOL

[UEFI PI Specification] defines EFI_SMM_COMMUNICATION_PROTOCOL. This protocol provides runtime services for communicating between drivers outside of SMM and a registered SMI handler inside of SMM. For example, BIOS DXE driver, BIOS runtime driver, or OS agent can consume this protocol.

See figure 1 [Variable] below. The EDKII UEFI variable driver implementation (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Variable/RuntimeDxe) is a typical SMM communication example. During driver initialization, the SMM agent registers a communication handler with the GUID gEfiSmmVariableProtocolGuid in VariableServiceInitialize(). The DXE RUNTIME driver initializes the SMM communication buffer with same GUID, gEfiSmmVariableProtocolGuid, in InitCommunicateBuffer(), and then calls SendCommunicateBuffer() at runtime to communicate with the SMI handler SmmVariableHandler().

All parameters like Name, Guid, Attributes Data, DataSize are in the communication buffer.

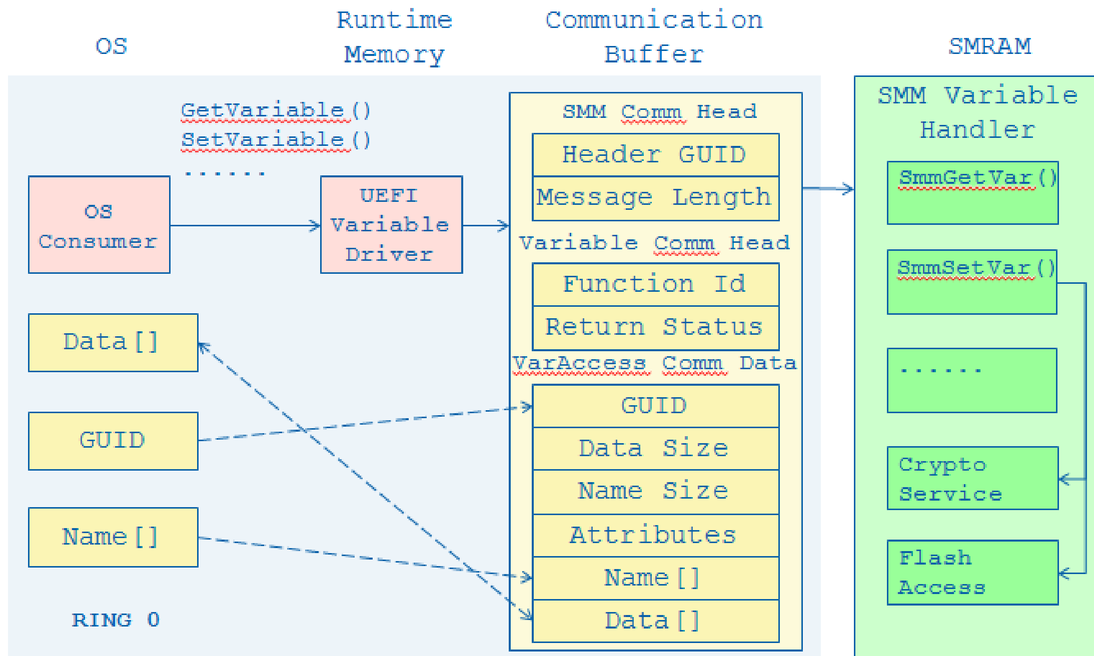

Figure 1 – SMM Variable

The EFI_SMM_COMMUNICATION_PROTOCOL is installed by SmmIpl (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/PiSmmCore). PiSmmIpl and PiSmmCore use SMM_CORE_PRIVATE_DATA structure as share buffer for data

exchange. See below figure 2. The GREEN portions of the figure (SmmIplImageHandle/ SmramRangeCount/ SmramRanges pointer/ SmmEntryPoint, SmstPtr) are used in driver initialization. After driver initialization, the PiSmmCore should never touch those regions and always use the internal SMRAM copy. The YELLOW portion (SmmEntryPointRegistered) is used before SmmReadyToLock event. The PiSmmCore should never touch it after SmmReadyToLock. The RED portion (InSmm, SmmCommunicationBuffer pointer, size, return status) are used after SmmReadyToLock. Care must be taken when the PiSmmCore refers to those fields because they might be filled by malicious code. The SMM communication buffer information is set by PiSmmIpl in SmmCommunicationCommunicate(), and is retrieved by the PiSmmCore in SmmEntryPoint().

The SmmEntryPoint() routine calls SmiManage(Guid) to handle a specific child SMI handler, and it then calls SmiManager(NULL) to handle the root SMI handler.

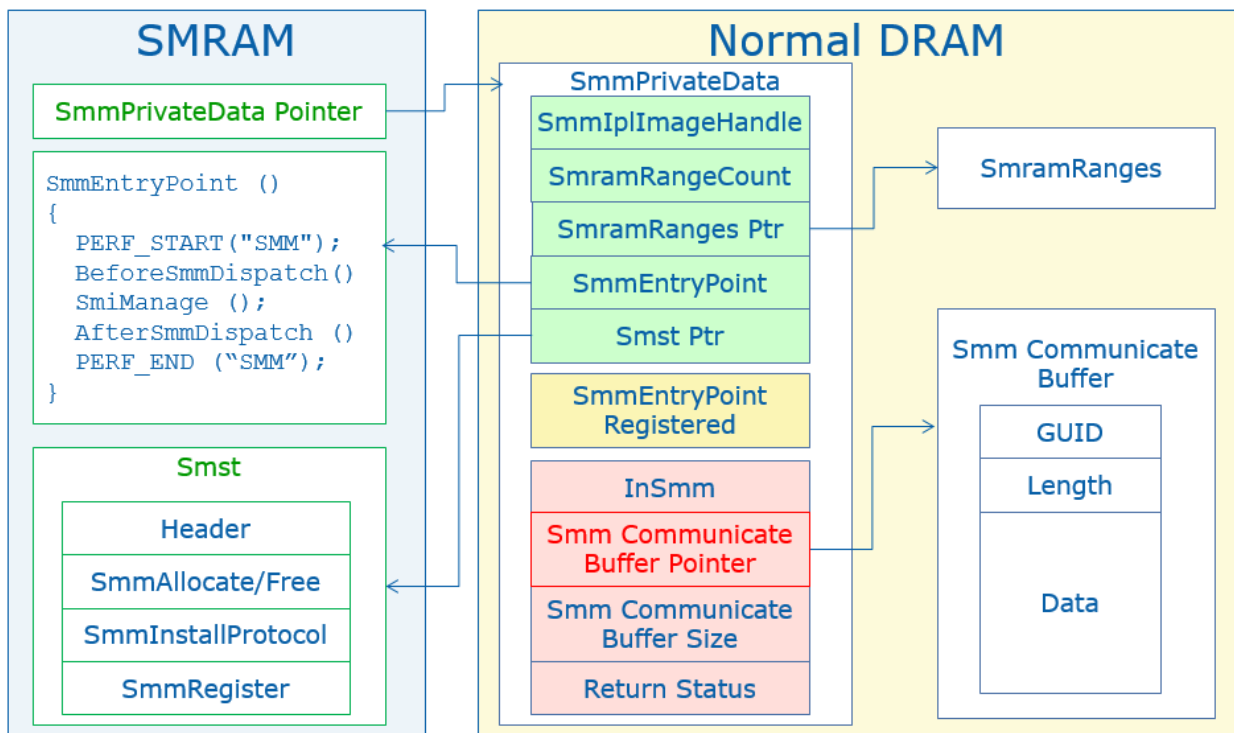

Figure 2 – SMM Communication

UEFI ACPI table

[UEFI] defines the SMM Communication ACPI Table in Appendix O. This table describes a special software SMI that can be used to initiate inter-mode communication in the OS present environment by non-firmware agents with SMM code. The benefit is that the OS agent does not need to locate a UEFI Runtime protocol. Instead, the OS agent just needs to report the SMM COMMUNICATION buffer address in an ACPI table and then trigger communication by using the invocation register or SWSMI. This SWSMI handler, PiSmmCommunicationHandler() in https://github.com/tianocore/edk2/tree/master/UefiCpuPkg/PiSmmCommunication, will handle this request and forward the request by calling gSmst->SmiManage(Guid). The final result is the same as the EFI_SMM_COMMUNICATION_PROTOCOL.

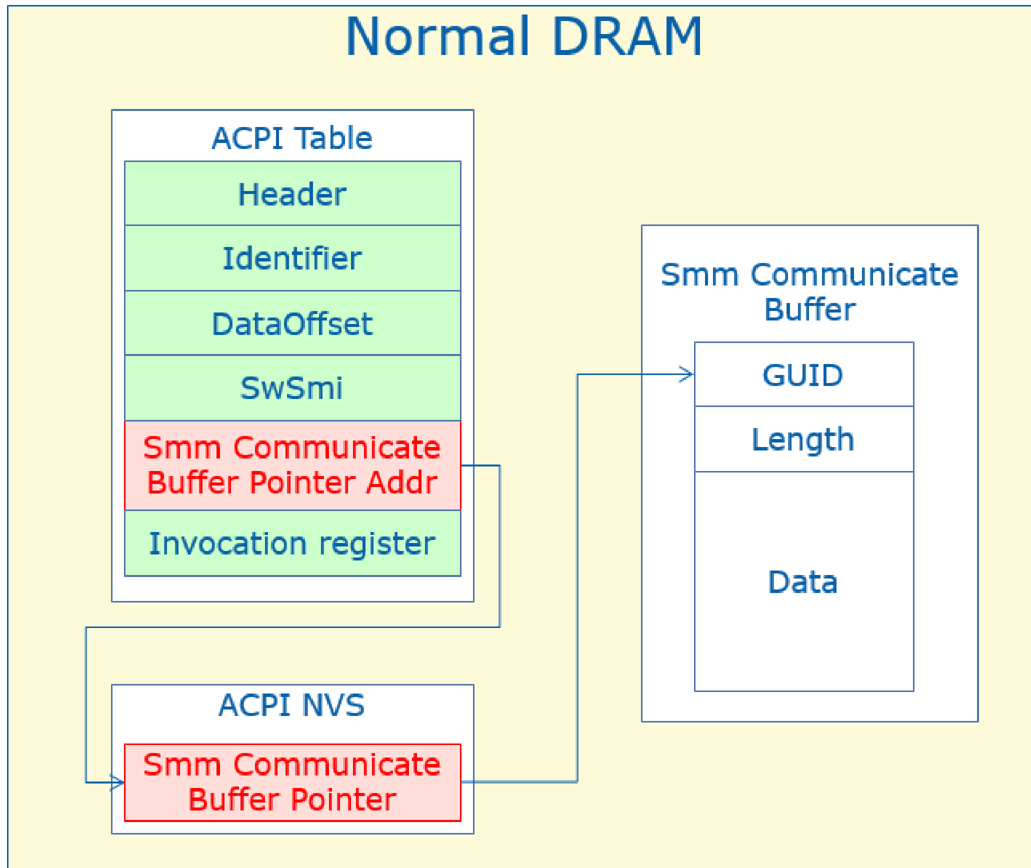

Figure 3 – SMM ACPI Table

Pre-defined location

Sometimes when a specific SWSMI occurs, the SMI handler may refer to data in a pre-defined location during driver initialization. A typical example is the SWSMI activation via ASL code.

See figure 4 for an example in the TCG SMM driver.
https://github.com/tianocore/edk2/tree/master/SecurityPkg/Tcg/Tcg2Smm

Tcg2Smm.h defines the TCG_NVS data structure. Tpm.asl defines the same TNVS data structure. This structure is allocated in ACPI NVS region by the PublishAcpiTable() routine in Tcg2Smm.c, and the pointer to TCG_NVS is saved in SMRAM. At runtime, Tpm.asl can fill the TCG_NVS according to the TCG Physical Presence (PP) request or memory clear (MC) request, and then trigger a SWSMI. Then the SMI Handler PhysicalPresenceCallback() or MemoryClearCallback() will be called to process this request.

The communication buffer TCG_NVS is predefined and has no need for a runtime allocation.

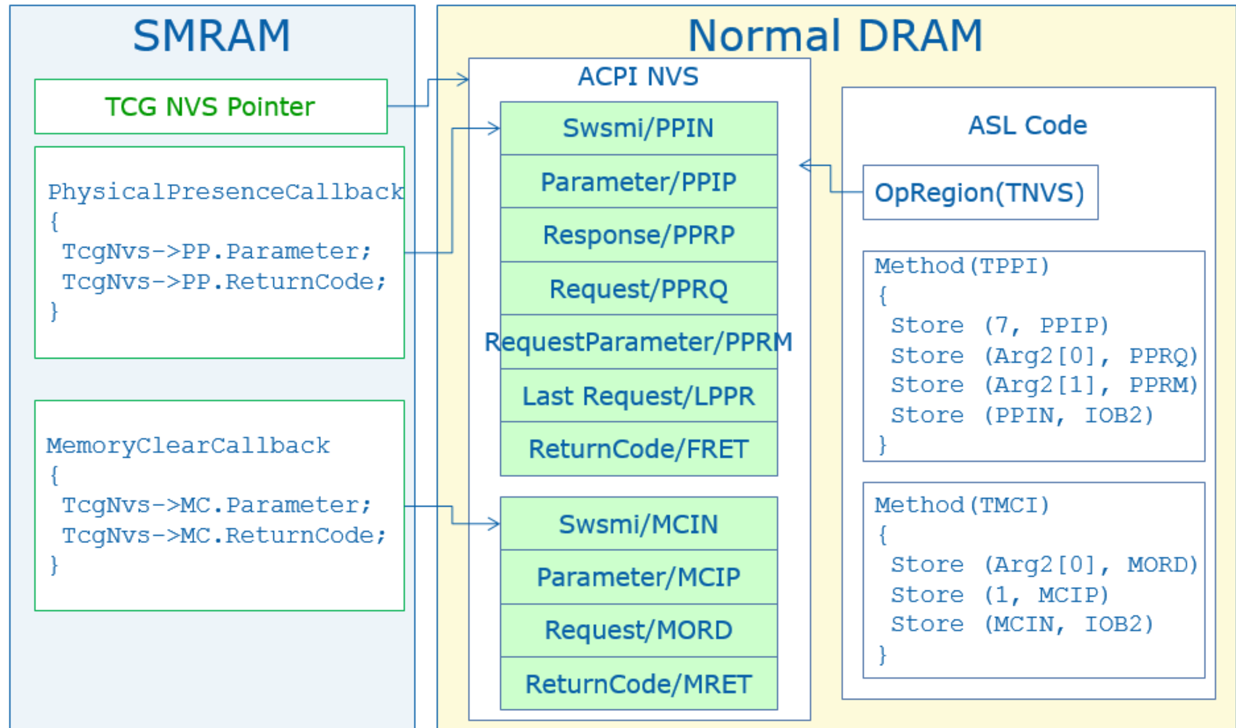

Figure 4 – TCG SMM

General purpose register

In a CSM ON system, the non-SMM agent may use a general purpose register to pass information to the SMI handler. For example, [IA32SDM] defines the INT15 D042 interface for Microcode update. [TCG Legacy] defines INT1A 0xBB interface for TCG legacy services. In most cases, these general purpose registers contain the address of an input buffer or output buffer. See figure 5 as an example.

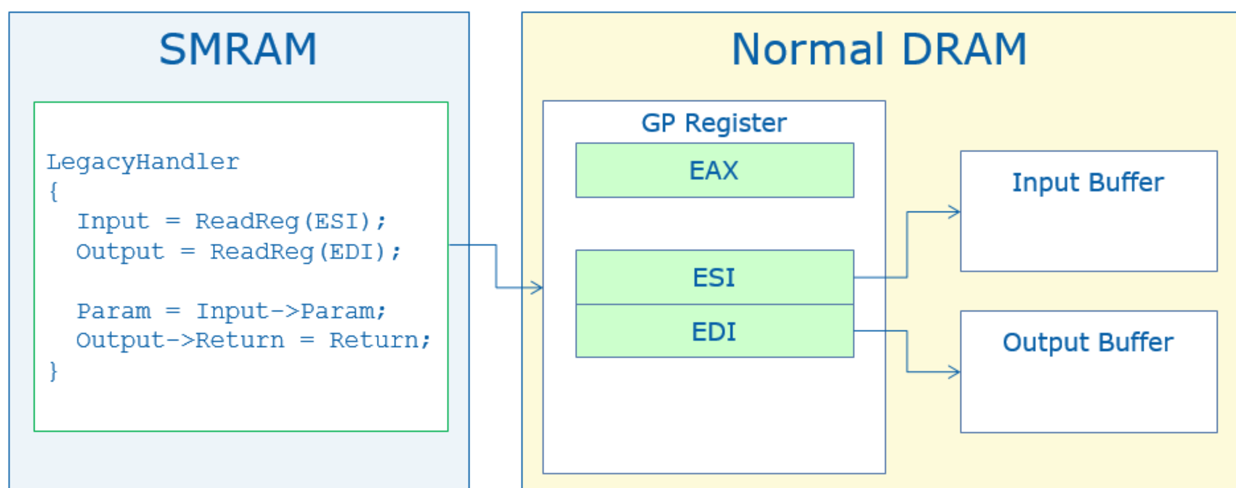

Figure 5 – GP register as communication

Pointer in communication buffer

In most cases, the communication buffer is used to hold data. But in some special case, there can be a data pointer in the communication buffer. As an example, see SMM_PERF_COMMUNICATE in https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/Performance.h .

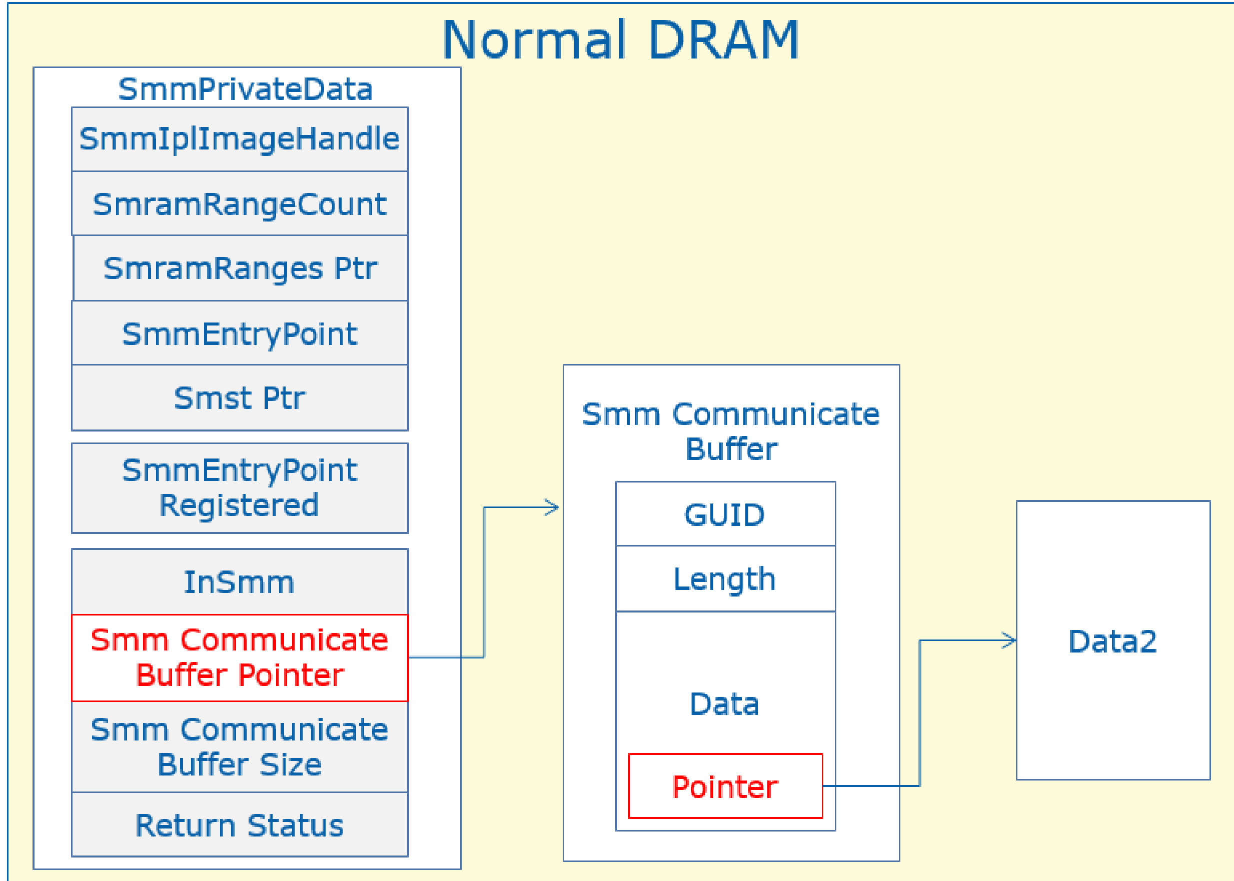

Figure 6 – Pointer in communication buffer

Summary

This section introduces several typical ways to perform data exchange between a non-SMM agent and a SMI handler.

Vulnerability of SMM communication

Attack on SMM communication

[SMM Attack1] [SMM Attack2] [SMM Attack 3] discuss several types of SMM attack via the SMM communication buffer.

Based on the security model, the attack can cause:

1) **Tampering** by breaking the **Integrity**. Data in protected entity is written by untrusted entity.
2) **Information Disclosure** by breaking the **Confidentiality**. Data in protected entity is read by untrusted entity.

Based on the target entity, we categorize these attacks into 2 groups:

1. OS/VMM attacks SMM.
2. **VM attacks VMM via SMM.**

[SMM Attack1] [SMM Attack2] discuss several types of SMM attack via the SMM communication buffer to attack SMM. For example:

1) SMM communication buffer overrides SMRAM.
2) SMM communication checker has Time-of-Check to Time-of-Use (TOC/TOU) issue.

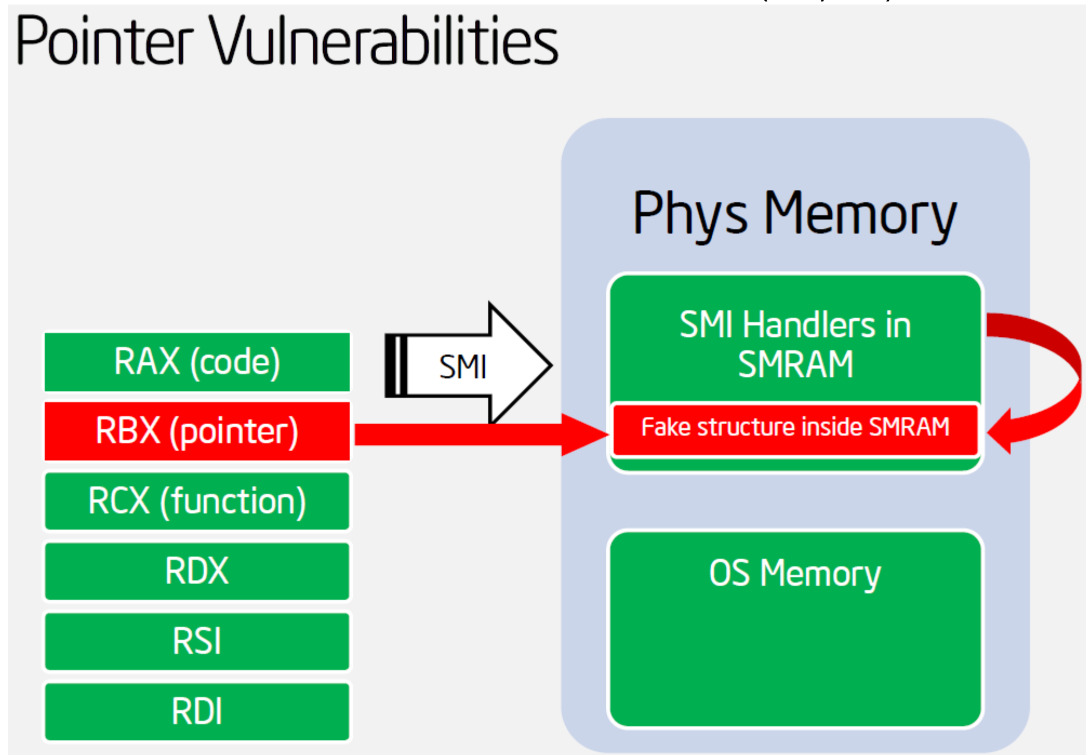

Figure 7 – source: [SMM Attack2]

For example, the malicious OS may pass a communication buffer pointer to SMRAM. If the SMI handler does not check the buffer location, this SMI handler may write some SMRAM content, or read SMRAM

content out.

[SMM Attack3] discusses a new way to attack a Virtual Machine Monitor (VMM) via SMM.

1) SMM communication buffer overwrites a VMM memory page.

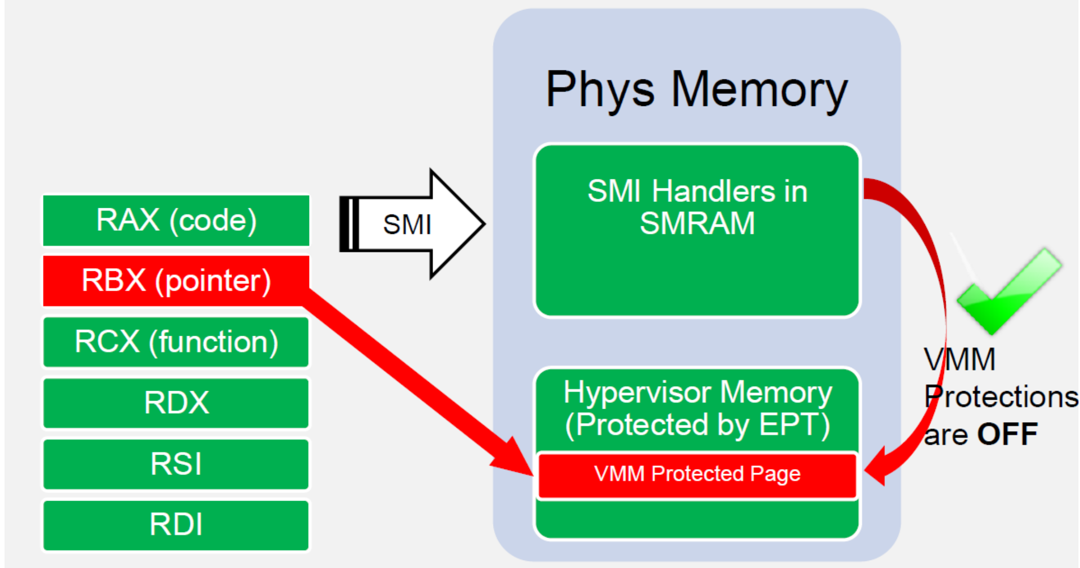

Figure 8 – source: [SMM Attack3]

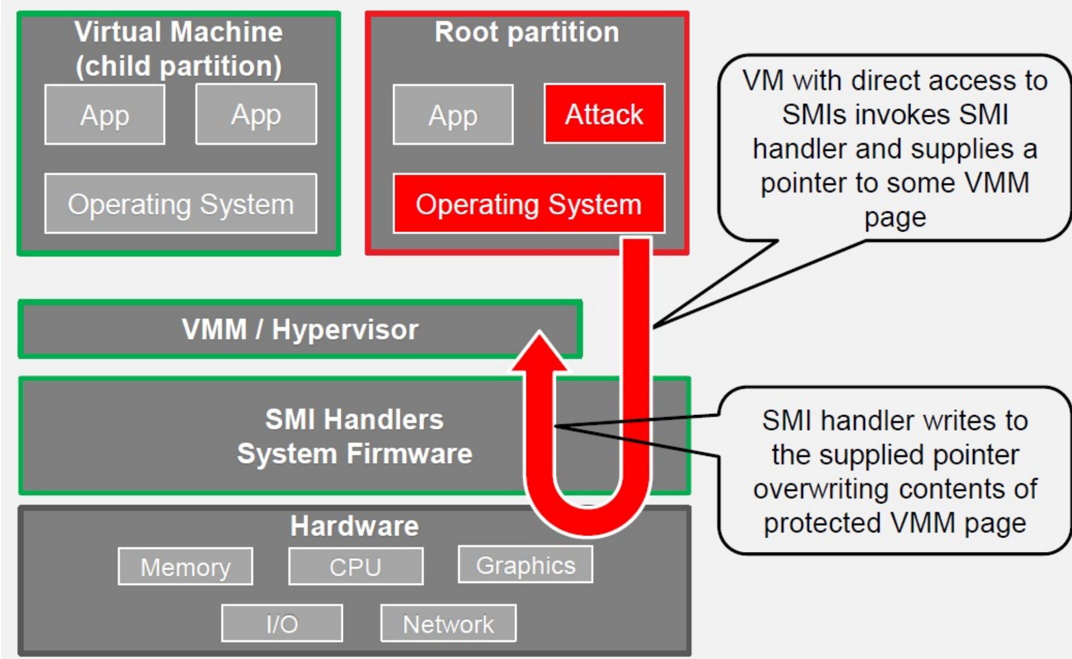

Figure 9 – source: [SMM Attack3]

Because the VMM allows a Virtual Machine (VM) guest to invoke SMI handlers, the malicious VM can invoke a SMI handler and supply a pointer to the VMM page. If SMI handler only checks if communication buffer overrides SMRAM but not if it overwrites arbitrary VMM pages, then this malicious request will be granted. The consequence is that the SMI handler helps overwrite contents of protected VMM pages, essentially acting as a 'confused deputy.'

Windows SMM Security Mitigations Table

Microsoft introduced Virtualization Based Security (VBS) in Windows 10[WSMT]. Unfortunately, [SMM Attack3] is a typical attack against VBS. In order to mitigate the problem, a number of changes to the SMM programming practices and assumptions must be introduced. Without these changes, the Windows OS may choose to degrade or disable specific security features or capabilities in order to ensure the integrity of platform security assets and provide a robust and reliable platform on which to develop new end user scenarios.

Since SMM is opaque to the OS, and since the OS must assume SMM is within the same trust domain as the OS itself, the OS must rely on SMM firmware to accurately self-report which of the Microsoft recommended security best practices it has implemented. To accomplish this, Microsoft has defined the Windows Security Mitigations Table (WSMT) for Windows SMM.

WSMT is one type of ACPI table. It defines the following Protection Flags.

- BIT0 - **FIXED_COMM_BUFFERS**. It means all SMM communication should be in an expected fixed memory region. Here *communication* means all other possible data exchanges between SMM and non-SMM entities, including but not limited to EFI_SMM_COMMUNICATION_PROTOCOL, ACPINVS in ASL code, general purpose registers as buffer pointers, etc.

- BIT1 - **COMM_BUFFER_NESTED_PTR_PROTECTION**. This is also for SMM communication. It is used to ensure that if there is a pointer in the communication buffer, the memory pointed should also be considered as an SMM communication buffer and located in a fixed memory region.

Both BIT0 and BIT1 are related to the SMM communication buffer. The EDKII generic core module can help check **FIXED_COMM_BUFFERS** because it knows the buffer address and length of communication buffer. But the generic core module does not know the internal data structure inside of the communication buffer. As such, each SMI handler should check **COMM_BUFFER_NESTED_PTR_PROTECTION** because only the SMI handler itself knows the meaning of communication buffer structure and has the capability to check for a nested pointer.

- BIT2 - **SYSTEM_RESOURCE_PROTECTION**. This is not related to the SMM communication buffer. Instead, it is used to indicate the platform has locked some system resource to prevent it from being modified by malicious software.

The WSMT table is a Windows requirement for VBS. [WSMT2][WSMT3].

We observed that previous EDKII BIOS's cannot set BIT0 and BIT1 because some DXE agents may allocate a SMM communication buffer in BootServicesData memory, and the SMI handlers do not check for this condition, leading to potential data writes into this memory that is owned by the OS or VMM after ExitBootServices().

In order to set both BIT0 and BIT1 of the Protection Flag in WSMT, we need to introduce a secure SMM communication mechanism in EDKII. This mechanism will be introduced in the next section.

Summary
This section introduces attacks on SMM communications.

Secure SMM Communication

Since the SMM communication buffer is an untrusted resource, the SMI handler must validate the contents before processing it. The SMI handler should consider 2 rules:

1) SMM should protect SMM itself to prevent an attack from the OS or VMM.
2) SMM should protect the VMM to prevent an attack from the VM.

The protection should consider the below areas:

1) **Integrity**. Data in a protected entity should NOT be written by an untrusted entity or it will cause **Tampering** and/or **Elevation of Privilege**.
2) **Confidentiality**. Data in a protected entity should NOT be read by an untrusted entity or it will cause an **Information Disclosure** and/or **Elevation of Privilege**.
3) **Availability**. Services in a protected entity should NOT be impacted by an untrusted entity or it will cause a **Denial of Service**.

Protect SMM itself to prevent an attack from the OS or VMM.

In order to follow this rule, the SMI handler needs to check if the SMM communication buffer override SMRAM.

EDKII provides an API SmmIsBufferOutsideSmmValid() in SmmMemLib (https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Library/SmmMemLib.h) to check if a communication buffer is outside SMM and valid.

For example, the SmmEntryPoint() in the PiSmmCore (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/PiSmmCore) will call SmmIsBufferOutsideSmmValid() to check if CommunicationBuffer is safe. Each individual SMI handler should also do the check based on its own need.

Also, in order to prevent a Time-of-Check/Time-of-Use (TOC/TOU) attack, the SMI handler should copy the communication buffer to SMRAM before checking the data fields. For example, SmmVariableHandler() in SmmVariable (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Variable/RuntimeDxe) will call CopyMem (mVariableBufferPayload, SmmVariableFunctionHeader->Data, CommBufferPayloadSize), then check the DataSizea and NameSize field.

Protect VMM to prevent an attack from a VM.

In order to follow this rule, the SMI handler needs to check if the SMM communication buffer conflicts with VMM memory.

We can use 2 techniques to handle this issue:

1) A **black list** of memory which is consumed by the VMM. Communication is denied if the buffer is in black list, otherwise communication is allowed.

2) A **white list** of memory which is consumed by the SMI handler. Communication is allowed if the buffer is in the white list, otherwise communication is denied.

[STM][STM2][SMM Monitor] discussed how to support #1 the black list solution. The VMM declares a list of protected memory resources via `ProtectOsResource` API (See figure 10, the GREEN part), so that the BIOS SMM monitor can check the list. However, most current BIOS's do not support the STM or other SMM monitors. There is no interface in the UEFI or PI specification to let the VMM pass such information at the current point of time. To support this mechanism, we need to update both the VMM and BIOS. This solution has big compatibility issues because an old VMM cannot support the new interface to interact with the STM or an SMM monitor.

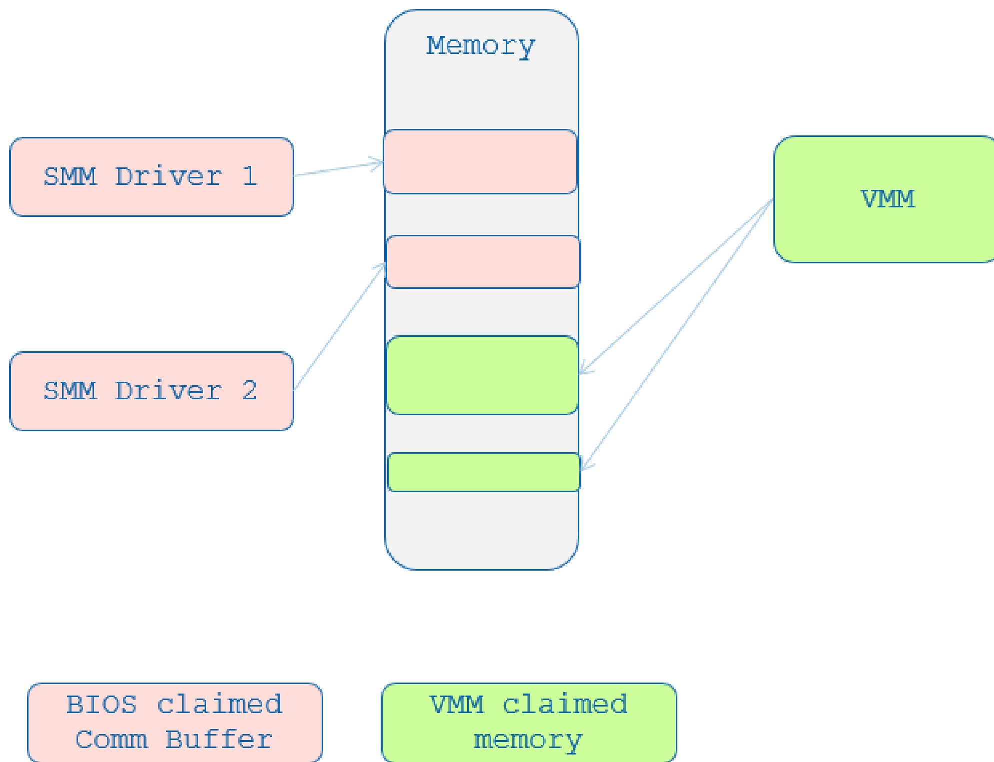

Figure 10 – PI-SMM Monitor resource declaration

Current EDKII implementations choose the #2 white list solution. It only impacts the BIOS SMM handler. It does not require a VMM change.

Now the question becomes: Who constructs the white list and how?

Communication Buffer Location Information.

We can define a *valid* communication buffer location in 2 ways:

1) Based on each SMI handler declaration.

2) Based on the memory type. (EfiReservedMemoryType/ EfiACPIMemoryNVS/ EfiRuntimeServicesData/ EfiRuntimeServicesCode)

[STM][STM2][SMM Monitor] discuss how to support #1. Each SMI handler needs to declare the valid SMM communication buffer region via `AddBiosResource` API (See figure 10, the RED part), so that the SMM core can know a full list of valid communication buffers. This information is precise. However, this solution has a big compatibility issue because each SMI handler needs to be updated in order to declare its own resource.

The current EDKII implementation chooses approach #2. Since the VMM should not put its own resource into EfiReservedMemoryType/ EfiACPIMemoryNVS/ EfiRuntimeServicesData/ EfiRuntimeServicesCode memory, only these 4 types of memory regions are treated as valid communication buffer locations. As long as the SMI hander finds the SMM communication buffer in those 4 regions, the SMI handler allows this communication request. Otherwise, the SMM communication buffer is invalid. Since most SMI handlers already follow this rule, the impact of #2 is much smaller than the impact of #1.

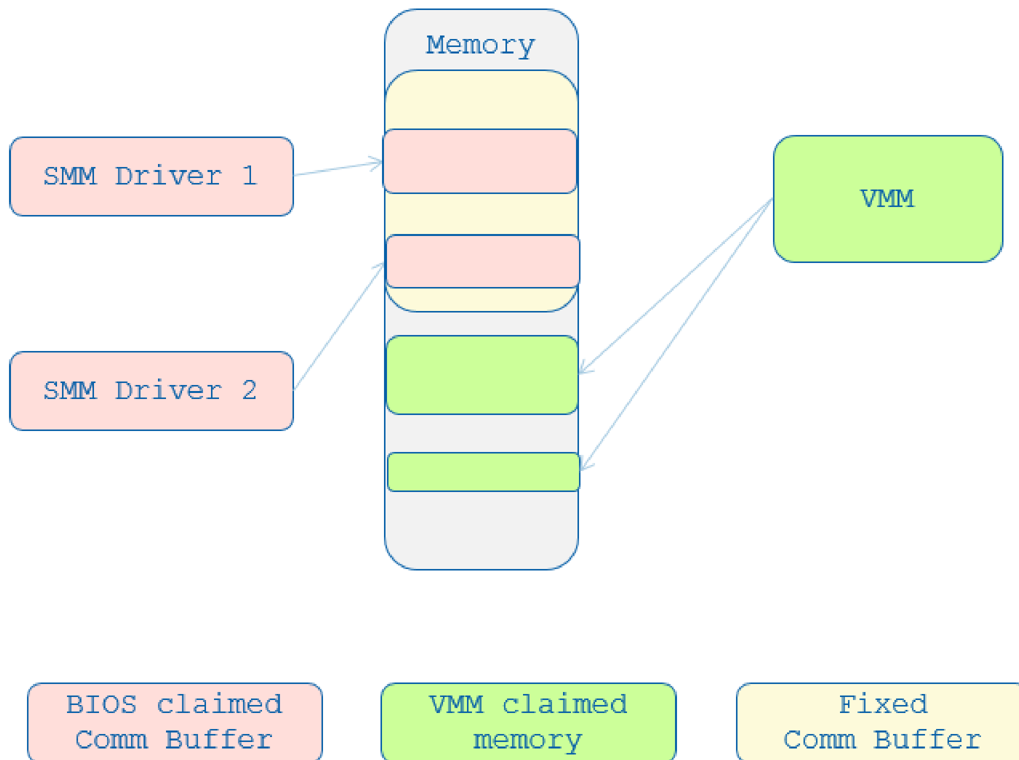

Figure 11 – Type based fixed communication buffer

NOTE: This SMM communication buffer type check only happens after SmmReadyToLock. There is no need to check before SmmReadyToLock because SMM environment is still in construction phase.

NOTE: This *valid* communication buffer includes more than the buffer needed for communication. It may include some other valid code or data, like RuntimeServiceCode, or data in the ACPINvs region. But that should not impact VMM. The VMM should not put its own data into EfiReservedMemoryType/ EfiACPIMemoryNVS/ EfiRuntimeServicesData/ EfiRuntimeServicesCode, Also, the VMM should protect its own resource and validate data from other regions.

Now the question is: Is there a SMM communication issue in using approach #2?

Communication Buffer Type

We analyze the current EDKII BIOS SMI handlers. The communication buffer can be categorized into 2 groups:

1) Static communication buffer. The communication buffer location is decided at BIOS boot phase prior to the EndOfDxe. It is not changed across each SMM communication instance. For example:
   a) VariableRuntimeDxe: mVariableBuffer (**EfiRuntimeServicesData**) (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Variable/RuntimeDxe)
   b) PiSmmIpl: mCommunicateHeader in global data region (**EfiRuntimeServicesCode**) (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/PiSmmCore)
   c) Tcg2Smm: mTcgNvs (**EfiACPIMemoryNVS**) (https://github.com/tianocore/edk2/tree/master/SecurityPkg/Tcg/Tcg2Smm)
   d) SmmLockBoxPeiLib: CommBuffer in stack (**EfiReservedMemoryType**) (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/SmmLockBoxLib)

2) **Dynamic** communication buffer. The communication buffer location is NOT decided at BIOS boot phase. It might be changed across each SMM communication instance. For example:
   a) MemoryProfileInfo: SMRAM_PROFILE_PARAMETER_GET_PROFILE_DATA.ProfileBuffer **(Allocated pointer in communication buffer)** (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Application/MemoryProfileInfo)
   b) FirmwarePerformanceDataTable: SMM_BOOT_RECORD_COMMUNICATE.BootRecordData **(Allocated pointer in communication buffer)** (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Acpi/FirmwarePerformanceDataTableDxe)
   c) DxeSmmPerformanceLib: SMM_PERF_COMMUNICATE.GaugeData **(Allocated pointer in communication buffer)** (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/DxeSmmPerformanceLib)
   d) OpalPasswordSupportLib: OPAL_SMM_COMMUNICATE_HEADER **(Allocated communication buffer)**

(https://github.com/tianocore/edk2/tree/master/SecurityPkg/Library/OpalPasswordSupportLib)

e) TcgInt1A services (close source): RSI/RDI as INT1A input buffer and output buffer.

(General purpose register)

The static communication buffer group follows the rules listed above.

The previous dynamic communication buffer group breaks the rule. Previous examples 2.a)~2.d) allocate EfiBootServicesData. Previous example 2.e) just passes a buffer pointer to SMI handler, which is allocated by an OS application and points to OS memory.

So, how to fix?

EDKII_PI_SMM_COMMUNICATION_REGION_TABLE

We can fix the Dynamic Communication Buffer issue by converting the dynamic buffer into a static buffer.
We notice that it is a burden to let each dynamic DXE agent to allocate Reserved/ACPINvs/RuntimeData/RuntimeCode for SMM communication. The alternative is to let a universal driver define a common static communication buffer to share with all agents. See Figure 12.

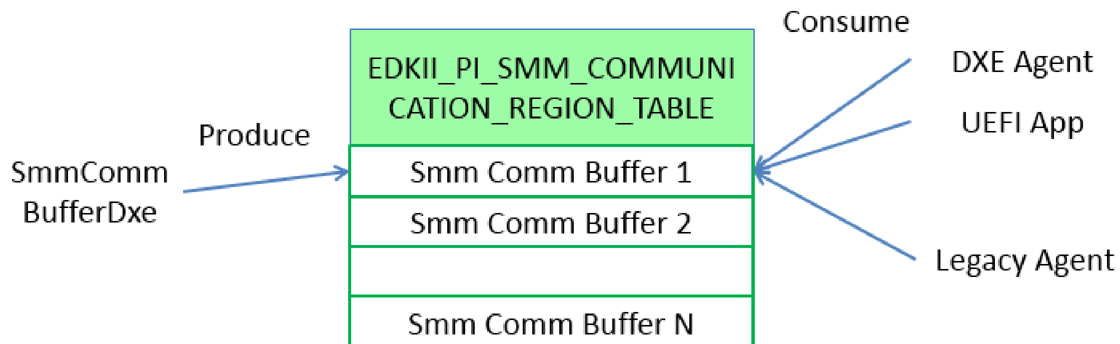

Figure 12 – PI-SMM Communication region table

In EDKII, the SmmCommunicationBufferDxe (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/SmmCommunicationBufferDxe) module produces this EDKII_PI_SMM_COMMUNICATION_REGION_TABLE. MemoryProfileInfo/ FirmwarePerformanceDataTable/ DxeSmmPerformanceLib/ OpalPasswordSupportLib consume this table to get the common SMM communication buffer in order to avoid buffer allocation.

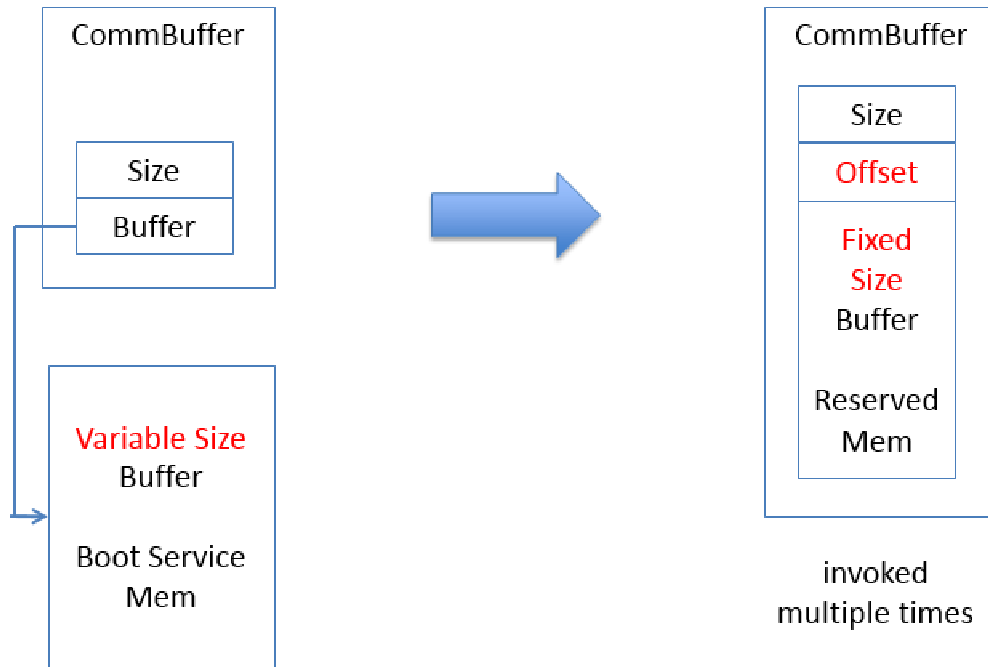

Figure 13 – Dynamic CommBuffer to Statis CommBuffer

If there is a pointer in the dynamic communication buffer, such as MemoryProfileInfo/ FirmwarePerformanceDataTable/ DxeSmmPerformanceLib, the DXE agent needs to be updated to use a fixed communication buffer instead of allocating new memory. If the size is not big enough, the DXE agent needs to invoke the Communication() API multiple times to retrieve the data one smaller element per transaction. See figure 13 for an example of this behavior.

CSM SMI handler consideration

If a platform only supports a UEFI OS, this section can be skipped.

If a platform still needs to support a legacy OS like Windows 7, then the BIOS needs a Compatibility Support Module (CSM), or PC/AT BIOS services. Some CSM SMI handlers still use general purpose registers to pass data, such as the TCG INT1A services.

Since the original buffer is allocated in the OS, we need a “Runtime Agent” to copy input data to a static communication buffer, trigger an SMI to process the requests, and copy output data from static communication buffer. See figure 14 for an example of this behavior.

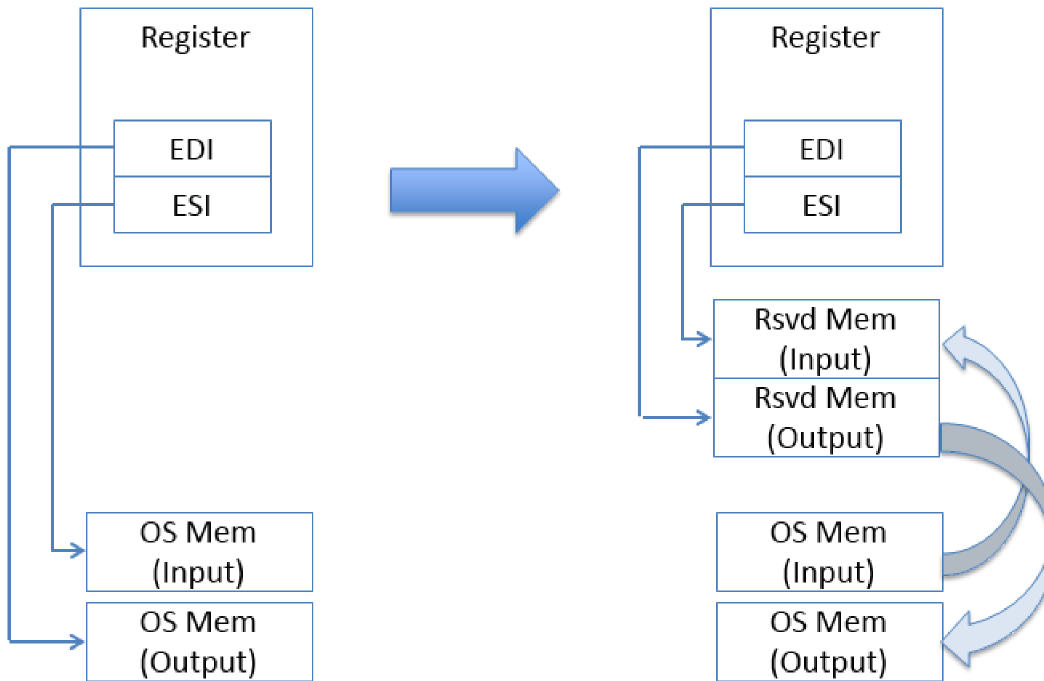

Figure 14 – Legacy agent – GP register as pointer

The TCG INT1A services are 16bit code. Originally, the SMI is triggered in 16bit code. However, it might be a huge burden if we to need write 16bit code to parse the TCG INT1A service data structure and move the data buffer to reserved memory.

A better way is to use reverse thunk to switch from 16bit code to native code (32bit or 64bit). The INT1A native handler is written in C in this case, so it is easy to parse INT1A services, move data and trigger SMI. See figure 15 for an example of this usage.

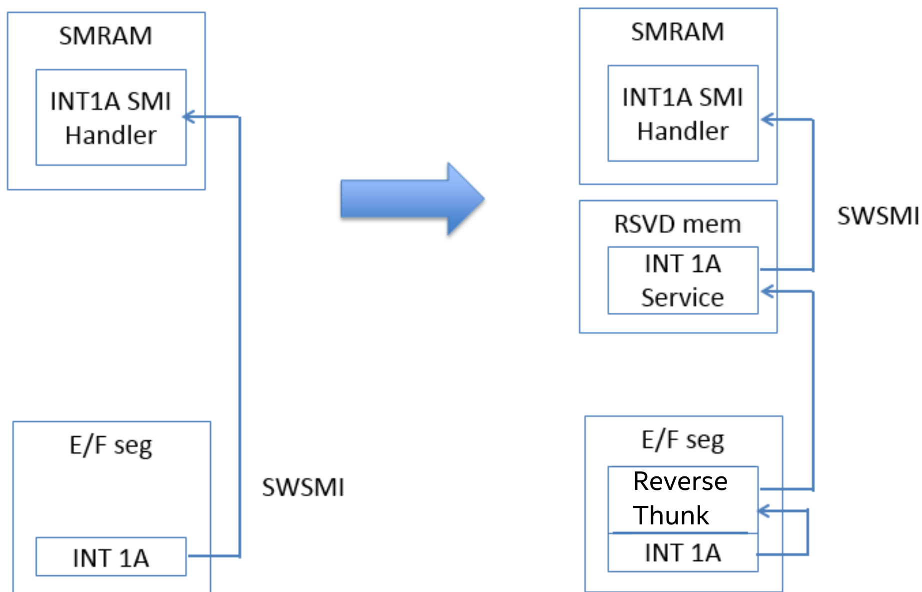

Figure 15 – Reverse Thunk solution to help buffer move

Fixed Communication Buffer Check

After a platform adopts the secure SMM communication solution, it should update all DXE agents to use the fixed communication buffer for functionality consideration.

For security considerations, the SMI handle should check if CommBuffer is a fixed communication buffer. To support this we updated the existing API SmmIsBufferOutsideSmmValid() in SmmMemLib (https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Library/SmmMemLib.h) to check if a communication buffer is in a fixed location. This check should be done not only for the communication buffer, but also for the pointers within the communication buffer.

The SmmMemLib implementation SmmLibInternalEndOfDxeNotify () (https://github.com/tianocore/edk2/blob/master/MdePkg/Library/SmmMemLib/SmmMemLib.c) calls gBS->GetMemoryMap() to collect Reserved/ACPINvs/RuntimeData/RuntimeCode information at EndOfDxe callback. Then this information will be used to check if the communication buffer is valid in SmmIsBufferOutsideSmmValid().

This information collection work must be done at EndOfDxe, because:

1) It is last time to allow an SMM driver call outside SMRAM. After that, an SMM call out is forbidden.
2) It is last time that a SMM driver trusts data from DXE. After that, SMM should not trust any data from DXE.

See below example on how an SMI handler calls SmmIsBufferOutsideSmmValid() to check a buffer. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/Acpi/FirmwarePerformanceDataTableSmm/FirmwarePerformanceSmm.c)

```
===========================
EFI_STATUS
EFIAPI
FpdtSmiHandler (
  IN     EFI_HANDLE                   DispatchHandle,
  IN     CONST VOID                   *RegisterContext,
  IN OUT VOID                         *CommBuffer,
  IN OUT UINTN                        *CommBufferSize
  )
{

//
// 1. Check full fixed buffer
//
  if (!SmmIsBufferOutsideSmmValid ((UINTN)CommBuffer, TempCommBufferSize)) {
    DEBUG ((EFI_D_ERROR, "FpdtSmiHandler: SMM communication data buffer in SMRAM
or  overflow!\n"));
    return EFI_SUCCESS;
  }

//
// 2. Copy data to local to avoid TOC-TOU attack
//
      BootRecordData  = SmmCommData-
      >BootRecordData; BootRecordSize =
      SmmCommData->BootRecordSize;

//
// 3. Check nested pointer
//
      if (!SmmIsBufferOutsideSmmValid ((UINTN)BootRecordData, BootRecordSize)) {
        DEBUG ((EFI_D_ERROR, "FpdtSmiHandler: SMM Data buffer in SMRAM or
        overflow!\n")); Status = EFI_ACCESS_DENIED;
        break;
      }

===========================
```

Summary

This section introduces the secure SMM communication mechanism to meet WSMT BIT0/BIT1 requirements.

Assumption and Recommendation

The current EDKII secure SMM communication solution has the below assumption:

1) **All SMM communication buffers are in Reserved/ ACPINvs/ RuntimeData/ RuntimeCode region.**

If SMM communication is triggered by a DXE agent at boot time or UEFI runtime, the DXE agent can allocate a communication buffer at its entry point, or reuse the common communication buffer in the EDKII_PI_SMM_COMMUNICATION_REGION_TABLE.

If the SMM communication is triggered by an OS agent, the OS agent must have the capability to get an agent specific reserved region, such as reported by an ASL code OpRegion, or to get a common communication buffer region in EDKII_PI_SMM_COMMUNICATION_REGION_TABLE.

2) **All SMM communication buffers are allocated before EndOfDxe.**

If the SMM communication buffer is allocated late, this information cannot be recorded by SmmMemLib. The consequence is that SmmIsBufferOutsideSmmValid() check will return FALSE and communication will fail.

If a DXE agent needs to allocate a permanent SMM communication buffer, we recommend the DXE agent do the allocation in the driver entry point.

If a DXE agent needs a temporary SMM communication buffer, we recommend that the DXE agent gets a common communication buffer region from EDKII_PI_SMM_COMMUNICATION_REGION_TABLE.

3) **The Reserved/ ACPINvs/ RuntimeData/ RuntimeCode allocated before EndOfDxe will not disappear in the memory map after EndOfDxe.**

The SmmMemLib records the SMM communication information based upon the type. If any of these memory types are freed later, they might be used by OS. But the SmmMemLib still treats this special memory region as valid and passes the check. As such, it becomes a potential vulnerability.

This is rare case, but it might happen. For example, the DxeCore may allocate RuntimeData for the configuration table. If one driver allocates more configuration tables after EndOfDxe, the DxeCore might do a RuntimeData reallocation. The previous RuntimeData region is freed. The same reallocation action also exists in HiiDatabase driver and SmbiosTable driver.

The easiest way to mitigate this is to enable the MemoryTypeInformation. In [MemMap], we discussed why MemoryTypeInformation is useful for the S4 resume. The DXE core will pre- allocate the BIN for each of these types of memory. If the BIN size is NOT big enough in this

boot, the system will record information, reset, and then the BIN size will become big enough in next boot.

If the memory type information feature is enabled, the Reserved/ ACPINvs/ RuntimeCode/ RuntimeData memory freed in the reallocation action will not become usable memory in the final memory map. The memory is always in a BIN and marked as the original type. As such, the OS will not use these freed memory locations to store its own data.

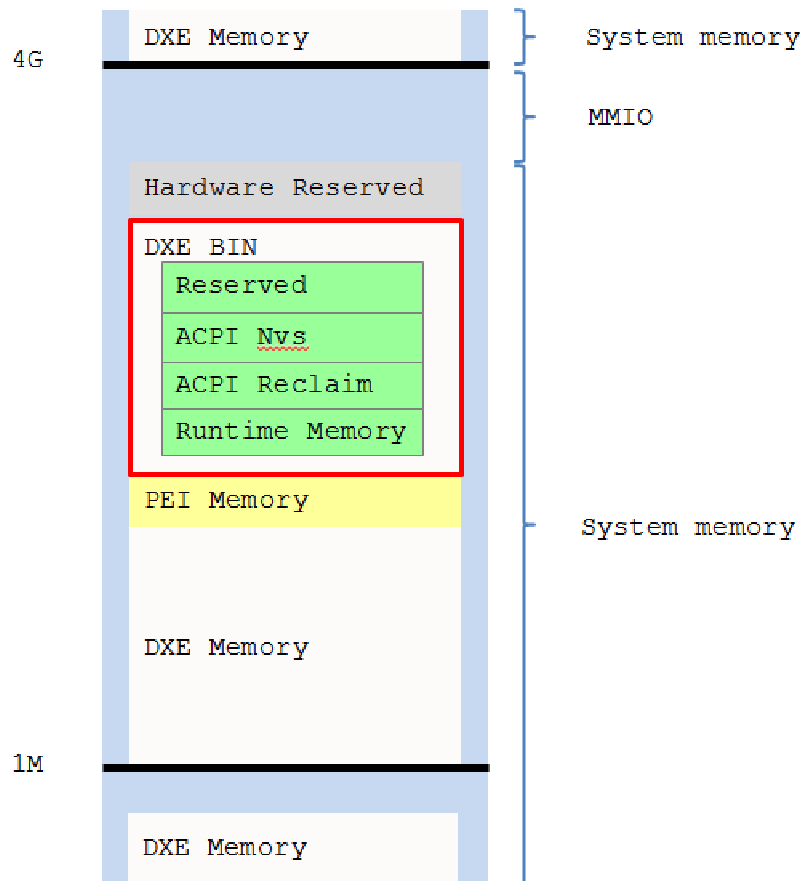

Figure 16 –memory layout with MemoryTypeInformation enabled

We strongly recommend a platform enable the MemoryTypeInformation feature, not only for S4 resume, but also for secure SMM communication.

Call for action

1) A platform should check its own SMM communication buffer.
2) The DXE agent should allocate a fixed communication buffer or use the EDKII_PI_SMM_COMMUNICATION_REGION_TABLE.
3) The SMI handler should call SmmIsBufferOutsideSmmValid() to validate the communication buffer.
4) A platform should enable Memory Type Information table.
5) A platform should report the WSMT table to indicate the capability.

NOTE: There is no core driver to publish WSMT table, because a core driver does not have knowledge on if a platform SMI handle is well designed and reviewed. Only a platform has knowledge on both core and platform SMI handler. We hope every platform owner can check that and publish WSMT table based on analysis result.

Future work

The SmmMemLib SmmIsBufferOutsideSmmValid() check is a *passive* check. It highly depends upon the SMI handler invoking this API to do the check. If a SMI handler does not invoke the call, the check is not performed.

A better way is to have an *active* check in an SMM monitor, such as discussed in [SMM Monitor]. One feasible solution is to enable page tables in the PiSmmCpu driver to enforce policy. The page table only covers SMRAM, the valid SMM communication buffer, and valid MMIO. If the SMI handler omits the check and accesses invalid communication buffer, then a Page Fault exception will be generated.

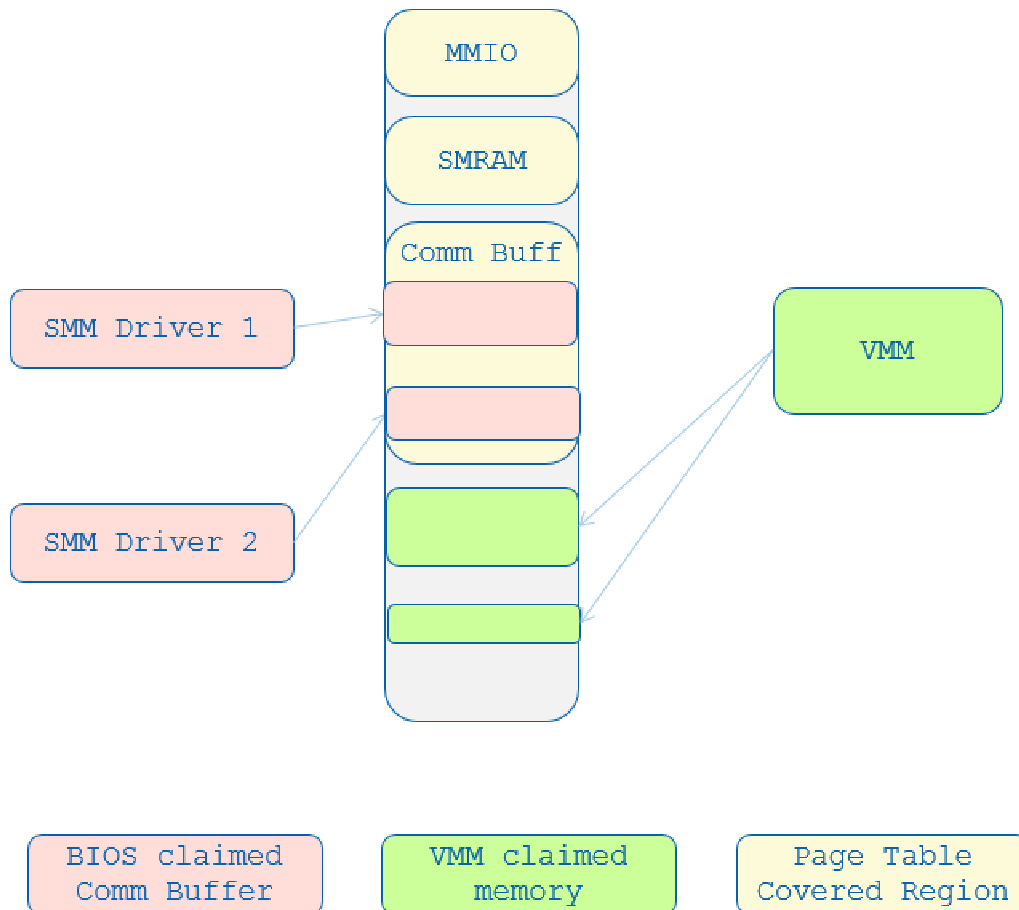

Figure 17 – Page table enforced memory layout

Summary
This section introduces the assumptions and recommendations for the secure SMM communication feature in EDKII.

Conclusion

Secure SMM communication is important for SMM as well as the VMM. Microsoft Windows has a requirement for a fixed communication buffer and the BIOS needs to report the capability via the WSMT ACPI table. This paper describes the design of secure SMM communications and provides recommendations on how a platform can enable the secure SMM communications feature.

Glossary

ACPI – Advanced Configuration and Power Interface. The specification defines a new interface to the system board that enables the operating system to implement operating system-directed power management and system configuration.

CSM – Compatibility Support Module MMIO – Memory Mapped I/O.
PI – Platform Initialization. Volume 1-5 of the UEFI PI specifications. SMM – System Management mode.
UEFI – Unified Extensible Firmware Interface. Firmware interface between the platform and the operating system. Predominate interfaces are in the boot services (BS) or pre-OS. Few runtime (RT) services.

WSMT – Windows SMM Security Mitigation Table. [WSMT]

References

[ACPI] ACPI specification, Version 6.1 www.uefi.org

[EDK2] UEFI Developer Kit www.tianocore.org

[IA32SDM] Intel® 64 and IA-32 Architectures Software Developer's Manual, www.intel.com

[MemMap] Jiewen Yao, Vincent Zimmer, A Tour Beyond BIOS Memory Map and Practices in UEFI BIOS, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Memory_Map_And_Practices_in_UEFI_BIOS_V2.pdf

[SMM Attack1] Oleksandr Bazhaniuk,Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alex Matrosov, Mickey Shkatov, Attacking and Defending BIOS in 2015 http://www.intelsecurity.com/advanced-threat-research/content/AttackingAndDefendingBIOS-RECon2015.pdf

[SMM Attack2] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alexander Matrosov, Mickey Shkatov, A New Class of Vulnerabilities in SMI Handlers http://www.intelsecurity.com/advanced-threat-research/content/ANewClassOfVulnInSMIHandlers_csw2015.pdf

[SMM Attack3] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, Yuriy Bulygin, Attacking Hypervisors via Firmware and Hardware http://www.intelsecurity.com/advanced-threat-research/content/AttackingHypervisorsViaFirmware_bhusa15_dc23.pdf

[SMM Monitor] Jiewen Yao, Vincent Zimmer, A Tour Beyond BIOS Supporting an SMM Resource Monitor using the EFI Developer Kit II, https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Supporting_SMM_Resource_Monitor_using_the_EFI_Developer_Kit_II.pdf

[STM] STM User Guide, https://firmware.intel.com/content/smi-transfer-monitor-stm

[STM2] Jiewen Yao, Vincent Zimmer, A Tour Beyond BIOS Launching a STM to Monitor SMM in EFI Developer Kit II, https://firmware.intel.com/sites/default/files/A_Tour_Beyond_BIOS_Launching_STM_to_Monitor_SMM_in_EFI_Developer_Kit_II.pdf

[TCG Legacy] TCG PC Client Specific Implementation Specification for Conventional BIOS, http://www.trustedcomputinggroup.org

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.6 www.uefi.org

[UEFI Book] Zimmer, et al, "Beyond BIOS: Developing with the Unified Extensible Firmware Interface," 2nd edition, Intel Press, January 2011

[UEFI PI Specification] UEFI Platform Initialization (PI) Specifications, volumes 1-5, Version 1.4 www.uefi.org

[Variable] Jiewen Yao, Vincent Zimmer, Star Zeng, A Tour Beyond BIOS Implementing UEFI Authenticated Variables in SMM with EDKII, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Implementing_UEFI_Authenticated_Variables_in_SMM_with_EDKII_V2.pdf

[WSMT] Windows SMM Security Table, https://msdn.microsoft.com/en-us/library/windows/hardware/dn495660(v=vs.85).aspx#wsmt
http://download.microsoft.com/download/1/8/A/18A21244-EB67-4538-BAA2-1A54E0E490B6/WSMT.docx

[WSMT2] Microsoft Update for Windows Security,
http://www.uefi.org/sites/default/files/resources/UEFI_Plugfest_March2016.pdf

[WSMT3] Microsoft Hypervisor Requirements, https://msdn.microsoft.com/en-us/library/windows/hardware/dn614617

Authors

Jiewen Yao (jiewen.yao@intel.com) is an EDKII BIOS architect, EDKII TPM2 module maintainer, ACPI/S3 module maintainer, and FSP package owner with the Software and Services Group at Intel Corporation. Jiewen is member of UEFI Security Sub-team and PI Security Sub-team in the UEFI Forum.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer and chairs the UEFI networking and security sub-team with the Software and Services Group at Intel Corporation.

Star Zeng (star.zeng@intel.com) is EDKII BIOS engineer with Software and Services Group at Intel Corporation.