White Paper

A Tour Beyond BIOS Launching a STM to Monitor SMM in EFI Developer Kit II

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

August 2015

Executive Summary

In the current UEFI PI infrastructure, System Management Mode (SMM) drivers are loaded into System Management Random Access Memory (SMRAM) and execute in a ring-0 privileged environment. However, there are several of SMRAM based attacks that have occurred [SMM01][SMM02][SMM03][SMM04][SMM05][SMM06]. As such, some platforms might need a way to monitor the SMM driver's actions and block some malicious or errant behavior. Some ideas to provide a least-privilege environment where SMM code behavior can be mediated have been presented before [SMM07][SMM08][SMM11]. In [SMM10], we discussed several possible implementations for a SMM monitor, including SMI Transfer Monitor (STM) and page tables. In this paper we will use a real open source example to show how STM works in BIOS and how to enable an STM in the system board BIOS.

Prerequisite

This paper assumes that audience has EDKII/UEFI firmware development experience [UEFI][UEFI PI Specification] and IA32 SMM knowledge [IA32 Manual]. He or she should be familiar with the UEFI/PI firmware infrastructure (e.g., PEI/DXE) and know the IA32 SMM driver flow. [UEFI Book]

Table of Contents

Overview

Introduction to SMM

System Management Mode (SMM) is a special-purpose operating mode in Intel® architecture based CPUs. SMM was designed to provide for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware from the manufacturer and not intended to be 3rd party extensible. [IA32 Manual]

Introduction to PI SMM

In order to support SMM in system firmware, [UEFI PI Specification] Volume 4 describes detailed infrastructure on how to support SMM in UEFI PI-based firmware. See below figure 1 for details.

The SMM Initial Program Loader (IPL) will load the SMM Foundation into SMM memory (SMRAM) and the SMM services will start at that time until a system shutdown.

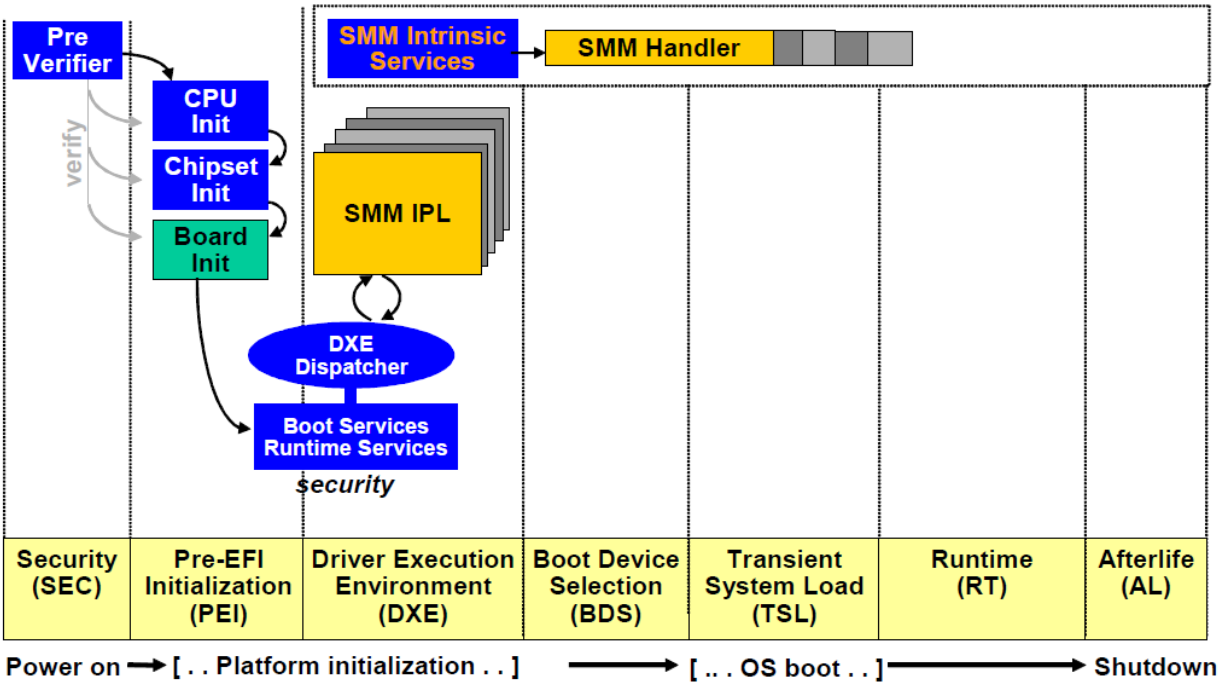

Figure 1 SMM architecture

We have discussed PI SMM Execution environment in [SMM10]. And more details can be found in [UEFI PI Specification] Volume 4, [IA32 manual] Volume 3 Chapter 34, and [SMM09].

Introduction to EDKII

EDKII is open source implementation of UEFI PI-based firmware which can boot multiple UEFI-aware operating systems. The EDKII open source project includes the SMM infrastructure which follows the PI specification to provide a capability for loading SMM drivers in the DXE phase of execution.

In this paper, we will use MinnowMax platform as real example to show how to build STM and enable STM on EDKII based IA platform.

Summary
This section provided an overview of SMM and EDKII.

STM Security Model

4 components in STM architecture

[IA32 Manual] Volume 3, 34.15 describes the Dual Monitor treatment of SMIs and SMM.
A dual-monitor treatment is activated through the cooperation of the **executive monitor** (the Virtual Machine Monitor (VMM) that operates outside of SMM to provide basic virtualization) and the **SMM-transfer monitor** (**STM**). The STM is the VMM that operates inside SMM, while in VMX operation, to support system-management functions. Control is transferred to the STM through VM exits. VM entries are used to return from SMM. For more information please also refer to [TrustedPlaform] and [SMM08].

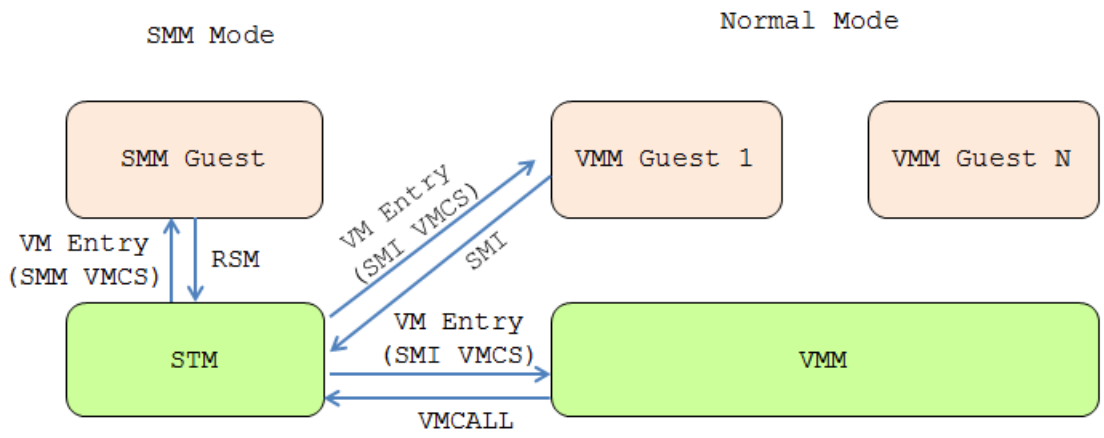

Figure 2 STM architecture

In order maintain the security for SMM, we need a set of rules for the 4 roles involved. These roles include: VMM Guest, VMM, SMM Guest, and STM. Components at the left hand side (SMM Guest and STM) are running in SMM mode. Components at the right handle side (VMM Guest and VMM) are running in the normal CPU mode. Components at the top (SMM Guest and VMM Guest) are considered as guest virtual machines of a virtual manchine monitor. Components at the bottom (STM and VMM) are considered as monitors.

Access Control Table

Accessed By	VMM Guest	VMM	SMM Guest	STM
VMM Guest	N/A	Legal	Controlled	Legal
VMM	Illegal	N/A	Controlled	Legal
SMM Guest	Illegal	Illegal	N/A	Legal
STM	Illegal	Illegal	Illegal	N/A

Table 1 Access Control Table

Table 1 shows an access control table for the 4 components. According to the priviliage control, the VMM should not be accessed by the VMM guest and the STM should not be accessed by the SMM guest. Also, the SMM area should not be access by a non-SMM component.
In contrast, the VMM can access the VMM guest, and the STM can access the SMM guest. Since the SMM component can access non-SMM component, the STM can access the VMM or the VMM guest.

The most important thing in the above table is the SMM Guest. According to the privilege control rule, the SMM guest can access the VMM or the VMM guest. However, this access should be strictly controled by the STM. And this strict access control is exactly the role of the STM.

The basic rules for a SMM guest are as follows:

- **The STM needs to protect itself**. The STM should determine the location and size of TSEG by consulting the SMRR and the STM must make sure that MSEG is covered by an SMRR so that MSEG is invisible to the VMM or the VMM guest. The STM ensure that MSEG remains invisible to the SMM guest.
- **The STM needs to protect the OS(VMM/VMM Guest)**. The STM is responsible for controlling SMM guest access to pages of memory in order to protect the integrity and confidentiality of the OS (VMM/VMM Guest). The STM must not map any OS protected pages into the SMM guest's address space.

Summary
This section introduced the security module of the STM.

Preparation of an STM in the BIOS

In this section, we will provide detailed information on what the BIOS should do to support the STM.

We use the open source STM package as example. The BIOS code can be found at <STM_SOURCE>\Bios directory. These sources are generic samples to demonstrate the concept, and they might be insufficient for production. For example, the PiSmmCpu module needs more restricted MSR programming and additional SmmPlatform modules need to register BIOS resources, like memory, MMIO, or IO.

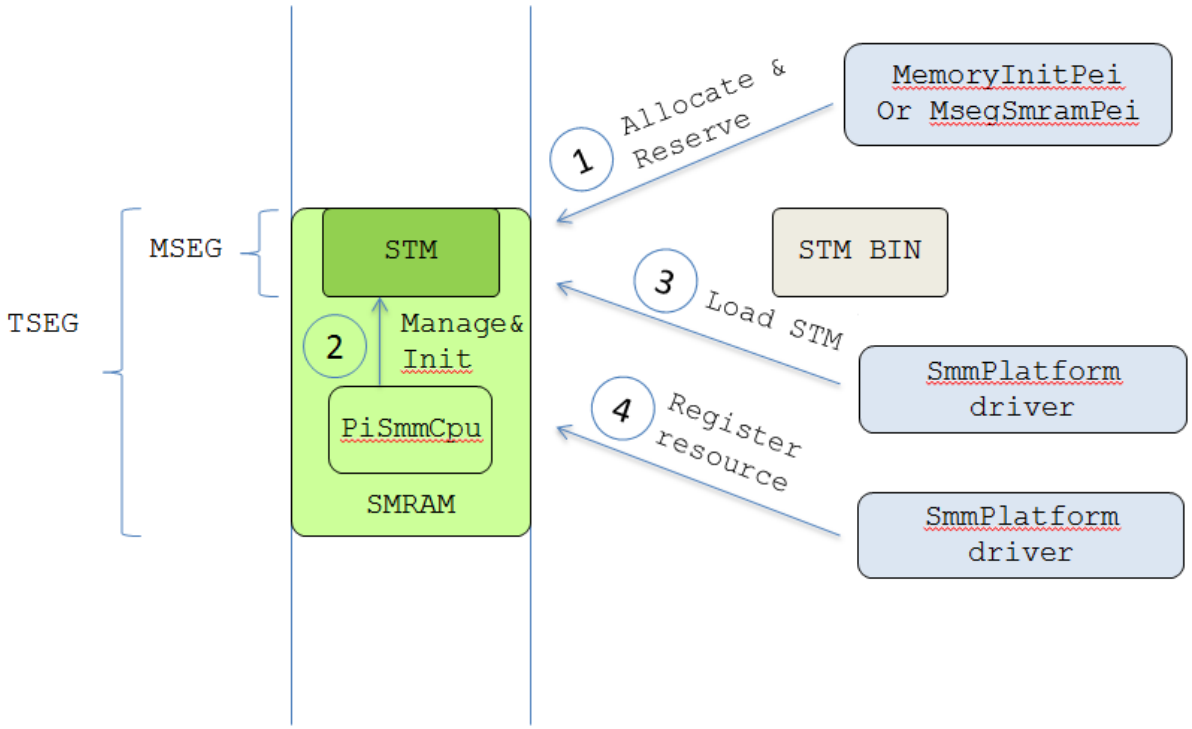

Figure 3 prepare the STM in BIOS

Responsibility of Memory Initialization module

In order to run the STM, STM must be stored in a special region of SMRAM – Monitor Segment (MSEG).
See figure 3, step 1. In most BIOSes, the SMRAM is reserved by the memory initialization driver, such as a PEIM. Then MSEG can be reserved by this memory initialization module or a standalone module. In the open source STM package, it is @ <STM_SOURCE>\Bios\StmPlatformSamplePkg\MsegSmramPei.

Responsibility of PI SMM CPU driver

See figure 3, step 2. If PcdCpuStmSupport is defined to be TRUE, the PI SMM CPU driver provides a service - EFI_SM_MONITOR_INIT_PROTOCOL, that initializes MSEG, and initializes **TXT_PROCESSOR_SMM_DESCRIPTOR** in SMRAM.

First, in the driver entrypoint, PI SMM CPU driver exposes a service - EFI_SM_MONITOR_INIT_PROTOCOL in PiCpuSmmEntry() @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\PiSmmCpuDxeSmm.c. The EFI_SM_MONITOR_INIT_PROTOCOL is provided @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\SmmStm.c.

Second, during SMM relocation, the PI SMM CPU driver initializes MSEG for IA32_SMM_MONITOR_CTL_MSR in NehalemInitMseg() @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\SmmFeatures.c.

Third, the PI SMM CPU driver initializes **TXT_PROCESSOR_SMM_DESCRIPTOR**, which is the data structure to pass information from a SMM guest to the STM. Data structure (gcPsd) is defined @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\X64\SmiException.asm.
gcPsd is initialized in InitializeMpServiceData() @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\MpService.c.

Finally, the PI SMM CPU driver should generate a 4G 1:1 virtual-to-physical mapping page table for the STM X64 mode, if STM is launched in non-TXT mode. The SINIT ACM should generate the 4G 1:1 mapping page table for the STM X64 mode, if STM is launch in TXT mode.

Responsibility of the Platform SMM driver

In order to support the STM, a platform STM driver must extract the STM binary from the BIOS and call EFI_SM_MONITOR_INIT_PROTOCOL to load it into MSEG. The Platform SMM drivers also need to report the required BIOS resources consumed by the SMI handler, for example, reserved memory, MMIO, IO, PCI resource, or MSR. See figure 3, step 3 and step 4.

Examples can be found @ <STM_SOURCE>\Bios\StmPlatformSamplePkg\StmPlatformSmm. StmPlatformSmm.c loads the STM, and registers BIOS resources at EndOfDxe callback function.

StmPlatformResource.c defines some resources required by BIOS SMI handler. Ideally a platform developer needs to check all SMI handlers to see how it accesses the system memory, Memory-Mapped I/O (MMIO) and IO resources during SMM runtime, and report the usage correctly.

Interface between the BIOS SMM Guest and the STM

TXT_PROCESSOR_SMM_DESCRIPTOR is the most information data structure to exchange information between the SMM guest and the STM.

For example, data includes:

- SMM Guest Entrypoint state: SmmEntryState, SmmCr3, SmmCs, SmmDs, SmmSs, SmmSmiHandlerRip, SmmSmiHandlerRsp, GdtPtr, GdtSize,
- SMM Guest callback function: SmmStmSetupRip, SmmStmTeardownRip, StmProtectionExceptionHandler.
- SMM Guest Information: LocalApicId, BiosHwResourceRequirementsPtr, AcpiRsdp, PhysicalAddressBit.

BiosHwResourceRequirementsPtr is the resource list reporting the resources needed by the BIOS SMI handler. It must be reported correctly, or the system will evolve into an unpredicable state.

The SMM Guest can also use the VMCALL instruction to communicate with the STM. `StmMapAddressRangeVMCALL`/ `StmUnmapAddressRangeVMCALL`/ `StmAddressLookupVMCALL` are only used if CPU does not have Extended Page Table (EPT) support. Since Intel Core i7 procrssor supports EPT, the above 3 VMCALLs are no longer needed. `StmReturnFromProtectionExceptionVMCALL` is still useful, in order to support StmProtectionExceptionHandler.

Summary

This section introduces the BIOS responsibility in deploying an STM solution.

Launch of the STM by a VMM

In this section, we will provide detailed information on what the VMM should do to launch the STM.

Specifically, if the VMM is launched by TXT, we refer to this VMM as a Measured Launched Environment (MLE).

We use the open source STM package as an example. The STM code can be found at <STM_SOURCE>\Stm directory. These are generic codes and cover most of the cases. There might be some additional restricted silicon specific CPU MSRs that need to be programmed in order to produce a fully secure solution.

STM image format and memory layout

See below figure 4. The DARK GREEN part is the static STM image, and the LIGHT GREEN part is the MSEG region used by the STM image.

The static STM image includes the following components:
• MSEG header (layout defined in [IA32 SDM]).
    It includes:
    • MSEG-header revision identifier.
    • SMM-transfer monitor features. (IA32 mode or X64 mode)
    • GDTR limit.
    • GDTR base offset.
    • CS selector.
    • EIP offset.
    • ESP offset.
    • CR3 offset.
The definition can be found @ <STM_SOURCE>\Stm\StmPkg\Include\StmApi.h - STM_HEADER.
• GDT
• STM code/data – PE/COFF image
The whole static image is generated by GenStm tool @ <STM_SOURCE>\Stm\StmPkg\Tool\GenStm.

The rest part of MSEG includes the following parts:
• Page Table for STM. It is created by the STM loader – the SMM CPU driver.
• Heap
• Stack – each processor has its own stack.
• SMI VMCS (VMCS for executive monitor) – each processor has its own SMI VMCS.
• SMM VMCS (VMCS for SMM guest) – each processor has its own SMM VMCS.

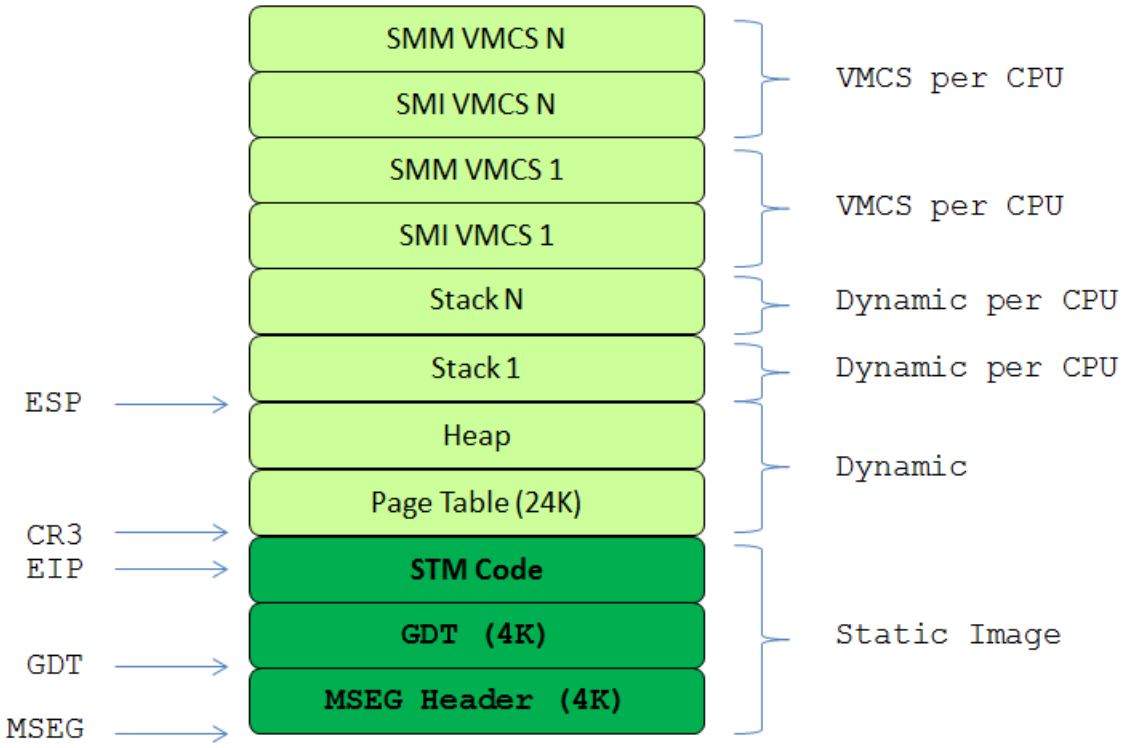

Figure 4 STM memory layout

Interface between the VMM and the STM

The STM needs to invoke a VMCALL for initialization. Some VMCALLs are used for STM configuration. The lifecycle of STM configuration is below.

1. System enters VMX root mode.
2. VMM invokes `InitializeProtectionVMCALL` to prepare the STM for setup of the initial protection profile. This is done on a single CPU and has global effect.
3. (Optional) VMM may invoke `GetBiosResourceVMCALL` to inspect the list of resources that the BIOS has access to. This can be done on a single CPU.
4. The VMM invokes `ProtectResourceVMCALL` to define the initial protection profile. The protection profile is global across all CPUs.
5. The VMM invokes `StartStmVMCALL` to enable the STM to begin receiving SMI events. This must be done on every logical CPU.
6. The VMM may invoke `ProtectResourceVMCALL` or `UnProtectResourceVMCALL` during runtime as many times as necessary.
7. The VMM invokes `StopStmVMCALL` to disable the STM. This must be done on every logical CPU.

STM initialization

Once the processor receives the first VMCALL (InitializeProtectionVMCALL for BSP and StartStmVMCALL for APs) from VMX root mode, the processor jumps to the EIP defined in the STM header, if IA32_SMM_MONITOR_CTL MSR is enabled.

In the open source STM, the entry point EIP is defined as _ModuleEntryPoint() @ <STM_SOURCE>\Stm\StmPkg\Core\Init\x64\AsmStmInit.asm.

The assembly code initializes the stack and saves the general purpose registers to the stack, then it jumps to InitializeSmmMonitor() @ <STM_SOURCE>\Stm\StmPkg\Core\Init\StmInit.c.

In InitializeSmmMonitor(), the STM creates the memory layout in Figure 4, initializes the VMCS for the SMM guest and the VMCS for SMIs, then it calls LaunchBack() to return to the VMM.

STM setup

Once the STM receives a StartStmVMCALL, the STM calls the SMM Guest handler SmmStmSetupRip to notify the SMM guest that the STM has commenced operation in SmmSetup(), @ <STM_SOURCE>\Stm\StmPkg\Core\Runtime\SmmSetup.c.

In the SMM guest, StmSetup entrypoint _OnStmSetup() is invoked @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\x64\SmiException.asm.
Then _OnStmSetup() calls SmmStmSetup() to record the information that the STM is activated, @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\SmmStm.c.
Finally, _OnStmSetup() uses RSM to return back to STM.

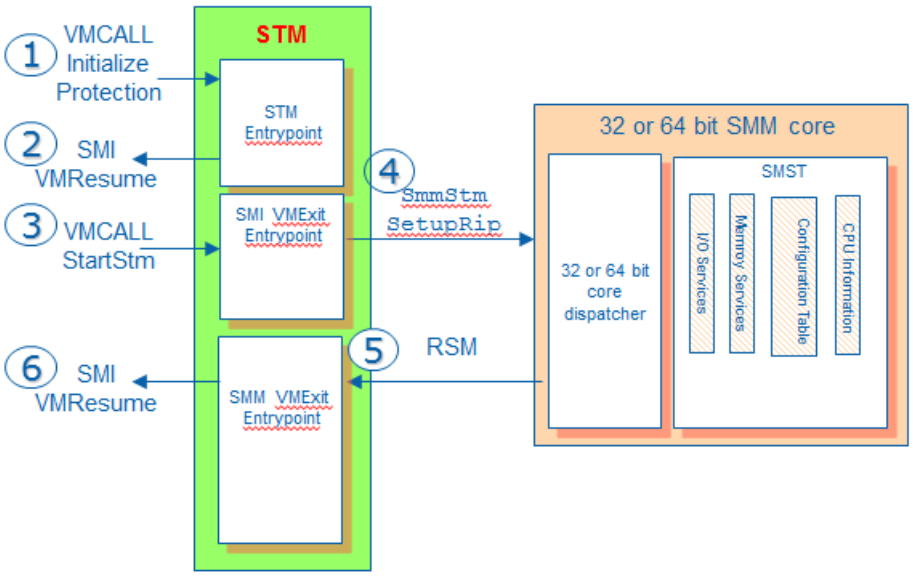

Figure 5 STM start

Launch of STM with TXT or without TXT

According to the IA32 SDM, the STM can be launched without TXT.
If STM needs to support a non-TXT launch, the STM should check TXT.STS.SENTER.DONE bit. If TXT.STS.SENTER.DONT is zero, it means the current launch is a non-TXT launch.
In non-TXT launches, the STM needs to find some information from different source.

Information	TXT launch	Non-TXT launch
CPU Number	BiosToOsData.NumLogProcs	ACPI MADT table
PCI Express Base	SinitToMleData.SinitMdrTable PCIe configuration region	ACPI MCFG table
Reset register	TXT.CMD.SYS_RESET	ACPI FADT table
Error Code	TXT.ERRORCODE	N/A

The open source STM supports both launch modes, and it uses IsSentryEnabled() function to check the current launch mode and get the CPU number or PCI express base from different sources.

Summary
This section introduces the STM image format, memory layout and how to initialize STM.

STM runtime

In this section, we will provide detailed information on the SMI flows after the STM is activated.

SMM runtime flow without STM

If there is no STM when an SMI occurs, then the CPU passes control into the real mode SMM entrypoint _SmiEntryPoint () @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\X64\SmiEntry.asm.
Then the PI SMM CPU switches to 32 or 64 bit mode and jumps to the PI SMM Core. The PI SMM core dispatches 32 or 64 bit SMI handlers, and returns back to the PI SMM CPU code. Finally PI SMM CPU code returns to the normal CPU mode via the RSM instruction.

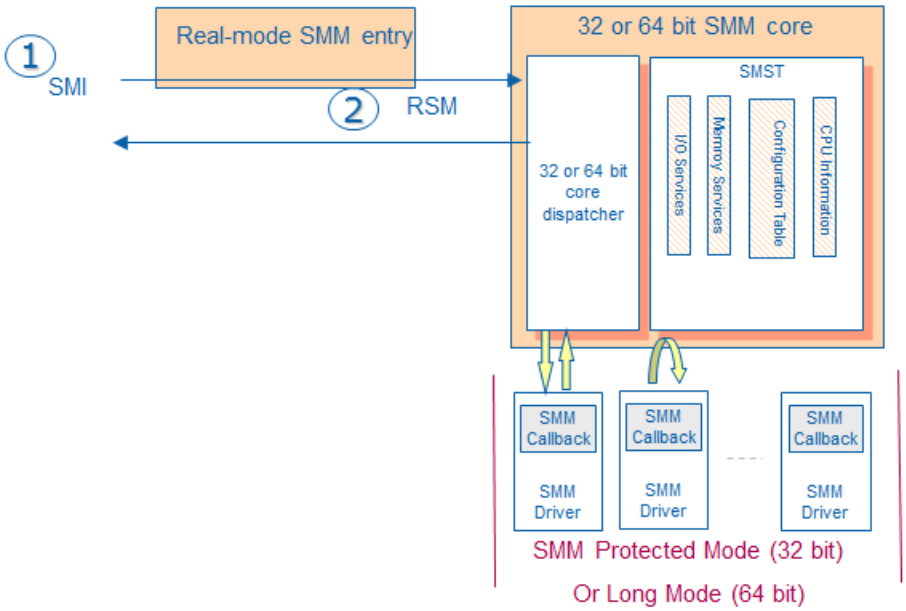

Figure 6 SMM flow without STM

SMM runtime flow with STM

If there is a STM, then when SMI happens, the processor passes control into the STM host entrypoint AsmHostEntrypointSmi() @ <STM_SOURCE>\Stm\StmPkg\Core\Runtime\x64\VmExit.asm.

AsmHostEntrypointSmi() calls StmHandlerSmi() to discover the VMExit reason. If the VMExit is caused by an SMI, then SmiEventHandler() will be called, @ <STM_SOURCE>\Stm\StmPkg\Core\Runtime\StmPkg\Core\SmiEventHandler.c.

SmiEventHandler() uses WriteSyncSmmStateSaveArea() to sync the SMM state registers from the VMCS to the SMM save state area, which could be memory based or MSR based. Then the SmiEventHandler() stores the current SMI VMCS and loads the SMM VMCS, and finally the code performs a VMResume to the SMM Guest.

The entrypoint of SMM Guest is _SmiHandler() @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\X64\SmiEntry.asm.

In normal conditions, the BIOS SMI handler will call RSM instruction after it handles an SMI.

The RSM will cause a VMExit in SMM, and the processor passes control to the STM host entrypoint AsmHostEntrypointSmm() @ <STM_SOURCE>\Stm\StmPkg\Core\Runtime\x64\VmExit.asm.

AsmHostEntrypointSmm() calls StmHandlerSmm() to discover the VMExit reason. If the VMExit is caused by a RSM, RsmHandler() will be called, @ <STM_SOURCE>\Stm\StmPkg\Core\Runtime\SmmRsmHandler.c.

RsmHandler() stores the current SMM VMCS and loads the SMI VMCS. Then RsmHandler() uses ReadSyncSmmStateSaveArea() to sync the SMM state registers from the SMM save state area, which could be memory based or MSR based, to the VMCS. After that RsmHandler() performs a VMResume to the SMI Guest.

See figure 7, the normal steps - 1, 2, 7, and 8.

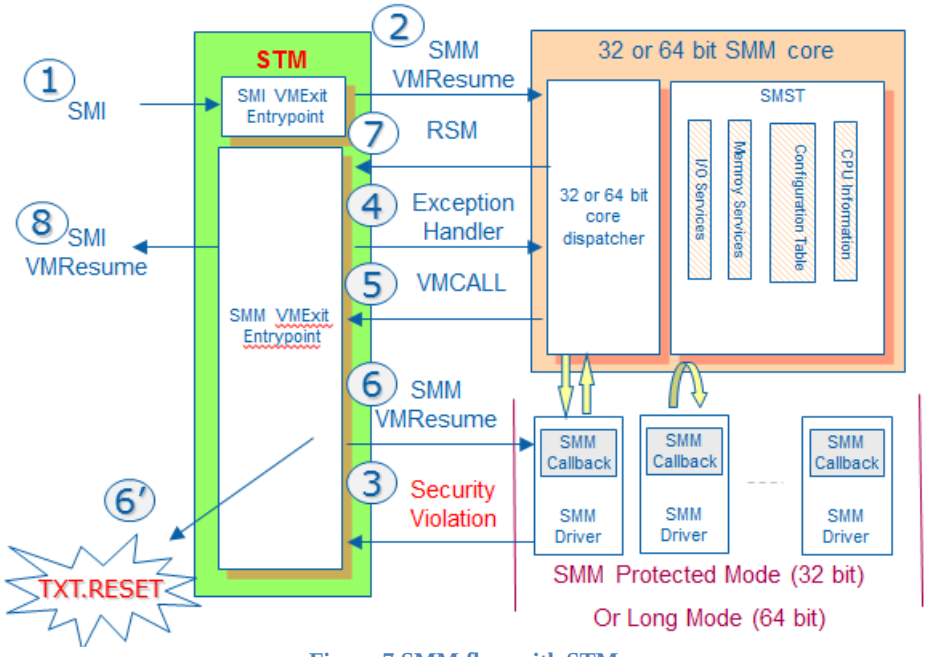

Figure 7 SMM flow with STM

SMM runtime violation flow with the STM

In order to maintain the integrity of the VMM, the STM may set some memory region or system resource to be invisible to the SMM Guest. If the SMM Guest accesses this resource, the SMM VMExit will occur because of this security access-control violation, and the processor passes control into the STM host entrypoint AsmHostEntrypointSmm().

AsmHostEntrypointSmm() calls StmHandlerSmm() to discover the VMExit reason. The reason might be: EptViolation, IoInstruction, or WRMSR. If the security violation is a memory access, the SmmEPTViolationHandler() is called @ <STM_SOURCE>\Stm\StmPkg\Core\Runtime\SmmEptHandler.c.

SmmEPTViolationHandler() checks if this violation is caused by memory protection, or PCI express protection. If the memory resource is declared by the VMM, the STM will inject an exception into SMM guest in SmmExceptionHandler() @ <STM_SOURCE>\Stm\StmPkg\Core\Runtime\SmmExceptionHandler.c.

Then the STM creates a BIOS exception stack in ResumeToBiosExceptionHandler() @ StmPkg\Core\Runtime\x64\SmmException.c, then VMResume to SMM Guest.

The exception entrypoint of SMM Guest is _OnException () @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\X64\SmiException.asm.
Finally, _OnException () uses VMCALL to return back to the STM.

SMRAM context handling

When the STM transfers an SMI to the SMM Guest, the STM needs to synchronize the SMM state registers between the VMCS and the SMM save state area, which could be memory based or MSR based.
The detailed state synchronization code is in WriteSyncSmmStateSaveArea() and ReadSyncSmmStateSaveArea() @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\SmmStateSync.

The general purpose Register, GDT/IDT, CR/DR can be sync directly. IOMisc should be from IoInstruction Qualification. AutoHALTRestart should be synced with GuestActivity in VMCS. If IORestart is filled, STM will update GuestRip in VMCS.

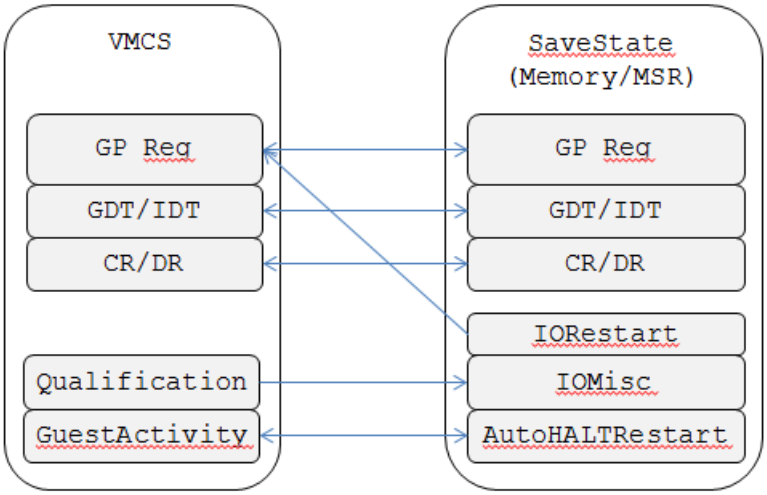

Figure 8 SMM Save State Sync

STM tear down

If the VMM decides to tear-down, the VMM calls StopStmVMCALL before invoking VMX OFF.
Once the STM receives StopStmVMCALL, the STM will call the SMM Guest handler SmmStmTeardownRip to notify the SMM guest that the STM is ceasing to monitor activities in SmmTeardown(), @ <STM_SOURCE>\Stm\StmPkg\Core\Runtime\SmmTearDown.c.

In the SMM guest, StmTeardown entrypoint _OnStmTeardown() is invoked @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\x64\SmiException.asm.
Then _OnStmTeardown() calls SmmStmTeardown() to record the information that the STM is deactivated, @ <STM_SOURCE>\Bios\IA32FamilyCpuStmSamplePkg\PiSmmCpuDxeSmm\SmmStm.c.
Finally, _OnStmTeardown() uses an RSM to return back to the STM.

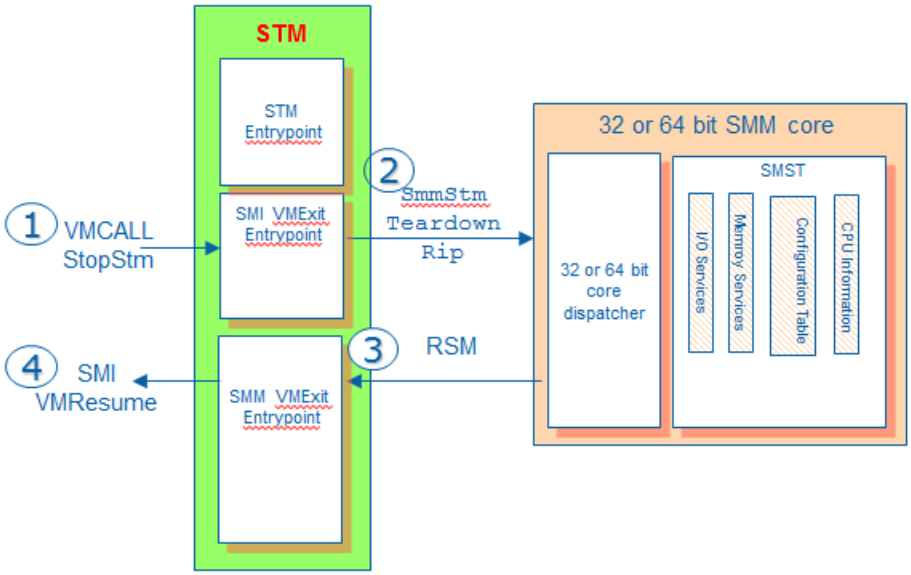

Figure 9 STM stop

Summary

This section introduces STM runtime flow and how to tear down STM.

STM as a monitor

The main purpose of STM is to be a reference monitor in order to maintain the integrity of the VMM.

Resource Protection

The STM can get BIOS resources in **TXT_PROCESSOR_SMM_DESCRIPTOR** BiosHwResourceRequirementsPtr. The VMM can register protected resource via `ProtectResourceVMCALL`. Then the STM sets the SMM VMCS based on BIOS resources and OS resources, respectively. Each of the VMCALL handlers are defined in mSmiVmcallHandler
@ <STM_SOURCE>\Stm\StmPkg\Core\Runtime\SmiVmcallHandler.c.

SmiVmcallProtectResourceHandler() will copy ResourceList to MSEG, validate if it is valid, and then call RegisterProtectedResource() @
<STM_SOURCE>\Stm\StmPkg\Core\StmResource.c.

• If the OS requests blocking some pages of memory, the STM will update EPT entry in the VMCS to remove the read access, write access, or execute access bits in EPTSetPageAttributeRange() @
<STM_SOURCE>\Stm\StmPkg\Core\Runtime\SmmEptHandler.c.

If the BIOS SMM handler still accesses these resources, an EPT violation will be triggered. This triggering is a VM exit event, and the system will return back from the BIOS SMM Guest to the STM SmmEPTViolationHandler() @
<STM_SOURCE>\Stm\StmPkg\Core\Runtime\SmmEptHandler.c.

• If the OS requests blocking some IO port, the STM will update the I/O bitmap of the VMCS to remove IO access in SetIoBitmapRange() @ <STM_SOURCE>\Stm\StmPkg\Core\Init\IoInit.c.

If the BIOS SMM handler attempts to access these resources, an IO instruction VM exit event will be triggered, and this will cause system to return back from the BIOS SMM Guest to the STM SmmIoHandler() @ StmPkg\Stm\Core\Runtime\SmmIoHandler.c.

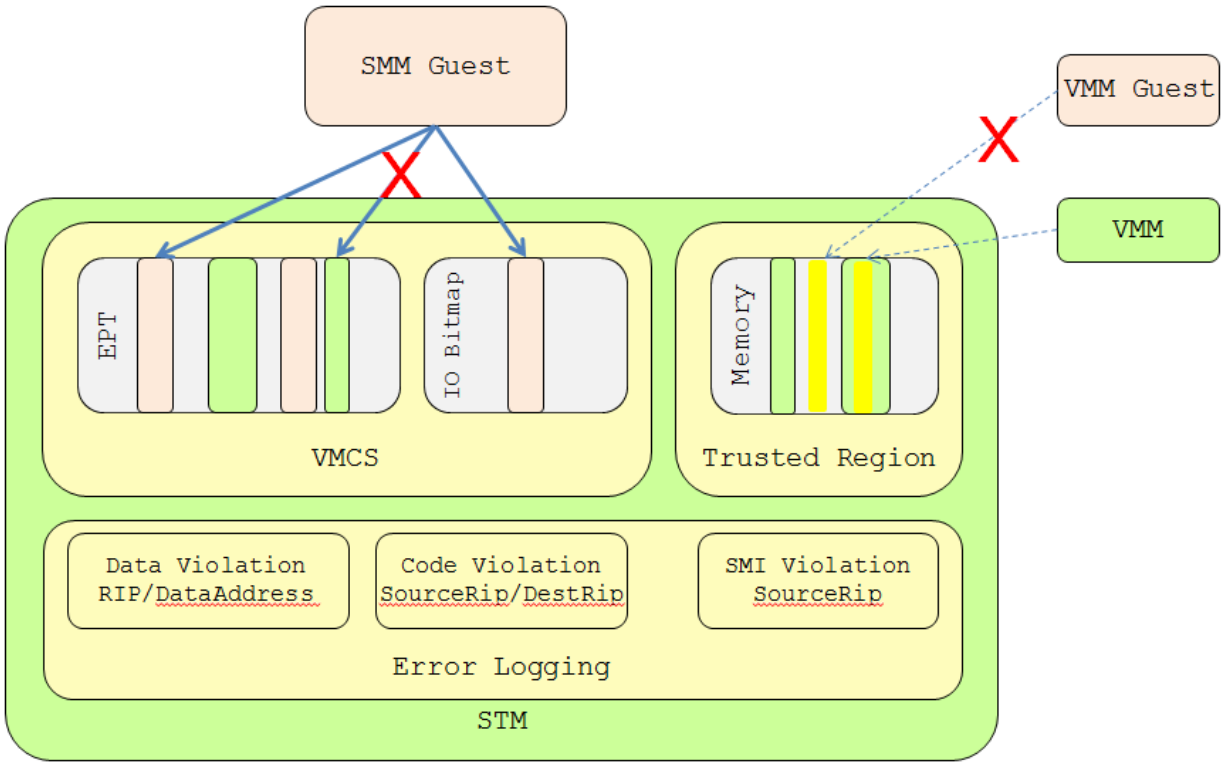

Figure 10 STM as Monitor

Event Log

The STM should implement an error log to record all violations. The VMM may call `ManageEventLogVMCALL` to control the logging feature. The sub-functions include NEW_LOG, CONFIGURE_LOG, START_LOG, STOP_LOG，CLEAR_LOG，and DELETE_LOG.
The log itself is a circular buffer that is allocated from VMM memory, so that VMM can refer to it at any time.

If the processor triggers a resource violation, for example, SmmEPTViolationHandler() or SmmIoHandler(), the STM will record an event in AddEventLogForResource() @ <STM_SOURCE>\Stm\StmPkg\Core\EventLog.c.

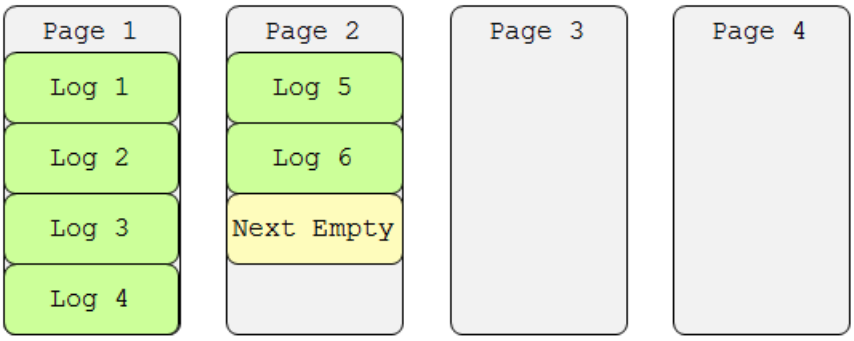

Figure 11 Event Log

Then in AddEventLog(), if the event log is started, this function will invoke GetNextEmpty() and copy log data to log area.

Take figure 11 as example, the VMM allocates a 4 page event log for the STM, and the STM records 6 logs – GREEN boxes. Then GetNextEmpty() will return the next slot in page 2 – YELLOW box.

Summary
This section introduces the STM as a monitor and typical usages thereof.

VMM responsibility

The STM cannot exist alone. It must be launched by a VMM in VMX-root mode. In order to demonstrate the concept, we provide a lightweight VMM for test purpose. We call it Firmware Reference Montiro (FRM). The FRM code can be found at <STM_SOURCE>\Test directory. It is generic code only for test purposes. It may not be used in any production. A production VMM must be used in a final secure solution, for example, that also includes setting up VT-d and EPT to protect VMM resources.

VMM initialization

After the FRM loader (<STM_SOURCE>\Test\FrmPkg\LoaderDriver) loads the FRM (<STM_SOURCE>\Test\FrmPkg\Core) into reserved memory, the FRM entry point is invoked with communication data, such as the FRM high memory region location, ACPI table pointer, FRM image base and size, and the STM services base and size. The FRM entry point is _ModuleEntryPoint() @ <STM_SOURCE>\Test\FrmPkg\Core\Init\FrmInit.c.

In _ModuleEntryPoint(), FRM initializes a heap and creates its own data structures like HostContext and GuestContext. The FRM Boot Strap Processor (BSP) need wakes up Application Processors (Aps) and put the APs into VMX root mode.
Finally, FRM launches back into the UEFI environment as a Guest.

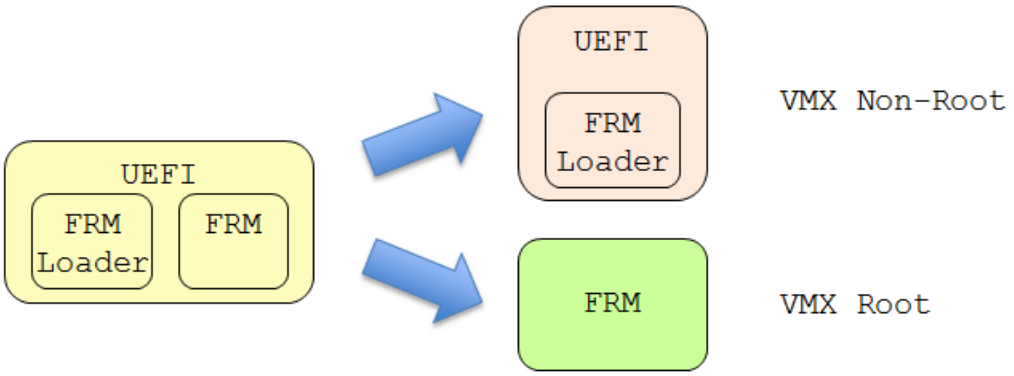

Figure 12 FRM

Before launching back to UEFI as a VM guest, the FRM LaunchGuestBsp() calls LaunchStm() @ <STM_SOURCE>\Test\FrmPkg\Core\Init\SrmInit.c.

LaunchStm() performs actions required in the “STM initialization” section by calling the interface between VMM and STM, e.g. `InitializeProtectionVMCALL`, `ProtectResourceVMCALL`, `StartStmVMCALL`.

Since these VMCALL APIs are defined in the STM User Guide only, we abstract those actions by defining a generic UEFI protocol. FRM can just call the UEFI protocol to perform the action,

e.g. `SmMonitorServiceProtocol->InitializeProtection(), SmMonitorServiceProtocol->ProtectOsResource(), SmMonitorServiceProtocol->Start()`.
<STM_SOURCE>\Test\FrmPkg\StmService driver produces all these services. These services must be in ReservedMemory region and must be protected by the FRM and the STM.

VMM tear down

During setup, the FRM registers a RESET and SHUTDOWN IO Port handler @ <STM_SOURCE>\Test\FrmPkg\Core\Init\IoInit.c. If FRM detects RESET or SHUTDOWN operations by the OS, FRM prepares to close FrmTeardownBsp() @ <STM_SOURCE>\Test\FrmPkg\Core\Runtime\FrmTeardown.c. Then TeardownStm() @ <STM_SOURCE>\Test\FrmPkg\Core\Runtime\SrmDeactivate.c is called to notify STM.

Summary
This section introduces the FRM – a lightweight test VMM used to launch the STM.

Conclusion

The STM can be one implementation of a reference monitor [MONITOR] for SMM drivers [SMM10]. This STM can maintain the integrity of the VMM and detect improper behavior on the part of SMM drivers. We use a real example to demonstrate how to enable the STM in a BIOS and perform both STM initialization and runtime flows.

Glossary

DEP – Data Execution Protection.

EBC – EFI Byte Code. See [UEFI Specification].

EPT – Extended Page Table. See [IA32 SDM]

IPL – Initial program loader.

MSEG – Monitor Segment. A special SMRAM for STM.

MLE – Measured Launched Environment

NX – No Execution. See DEP.

PI – Platform Initialization. Volume 1-5 of the UEFI PI specifications.

SMI – System Management Interrupt. The interrupt to trigger processor into SMM mode.

SMM – System Management Mode. x86 CPU operational mode that is isolated from and transparent to the operating system runtime.

SMRAM – System Management RAM. The memory reserved for SMM mode.

SMRR – System Management Range Register.

STM – SMI Transfer Monitor.

TXT – Intel Trusted Execution Enviornment

UEFI – Unified Extensible Firmware Interface. Firmware interface between the platform and the operating system.

XD – Execution Disable. See DEP.

VMCS – Virtual Machine Control Structure. See [IA32 SDM]

VT – Virtualization Technology. See [IA32 SDM]

WP – Write Protection.

References

[ACPI] Advanced Configuration and Power Interface, vesion 6.0, www.uefi.org

[APEI] Sakthikumar, Zimmer, "A Tour Beyond BIOS Implementing the ACPI Platform Error Interface with the Unified Extensible Firmware Interface," January 2013, https://firmware.intel.com/sites/default/files/resources/A_Tour_beyond_BIOS_Implementing_APEI_with_UEFI_White_Paper.pdf

[EDK2] UEFI Developer Kit www.tianocore.org

[EDKII specification] A set of specification describe EDKII DEC/INF/DSC/FDF file format, as well as EDKII BUILD. http://tianocore.sourceforge.net/wiki/EDK_II_Specifications

[EMBED] Sun, Jones, Reinauer, Zimmer, "Embedded Firmware Solutions: Development Best Practices for the Internet of Things," Apress, January 2015, ISBN 978-1-4842-0071-1

[IA32 Manual] Intel® 64 and IA-32 Architectures Software Developer Manuals http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[IA VMM] John Scott Robin & Cynthia E. Irvine , "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor", 2000, http://www.usenix.org/events/sec2000/full_papers/robin/robin.pdf

[MEMORY] Yao, Zimmer, Fleming, "A Tour Beyond BIOS Memory Practices in UEFI", June 2015 https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Memory_Practices_with_UEFI.pdf

[MLE] Intel® Trusted Execution Technology (Intel® TXT) Software Development Guide - Measured Launched Environment Developer's Guide http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf

[MONITOR] Anderson, "Computer Security Technology Planning Study," ESD-TR-73-51, US Air Force Electronic Systems Division, 1973 http://csrc.nist.gov/publications/history/ande72.pdf

[SMM01] Oleksandr Bazhaniuk, et al, "A New Class of Vulnerabilities in SMI Handlers", 2015, https://cansecwest.com/slides/2015/A%20New%20Class%20of%20Vulnin%20SMI%20-%20Andrew%20Furtak.pdf

[SMM02] Rafal Wojtczuk, et al, "Attacks on UEFI Security", 2015, https://bromiumlabs.files.wordpress.com/2015/01/attacksonuefi_slides.pdf

[SMM03] Corey Kallenberg, et al, "How many million BIOSes world you like to infect?", 2015, http://legbacore.com/Research_files/HowManyMillionBIOSWouldYouLikeToInfect_Full2.pdf

[SMM04] Loïc Duflot, et al, "System Management Mode Design and Security Issues", 2010, http://www.ssi.gouv.fr/uploads/IMG/pdf/IT_Defense_2010_final.pdf

[SMM05] Rafal Wojtczuk, et al, "Attacking Intel BIOS", 2009, http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf

[SMM06] "ASUS Eee PC and other series: BIOS SMM Privilege Escalation Vulnerabilities", 2009, http://www.securityfocus.com/archive/1/505590

[SMM07] Dick Wilkins, "UEFI Firmware –Securing SMM", 2015, http://www.uefi.org/sites/default/files/resources/UEFI_Plugfest_May_2015%20Firmware%20-%20Securing%20SMM.pdf

[SMM08] Rafal Wojtczuk, et al, "Attacking Intel® Trusted Execution Technology", 2009, http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf

[SMM09] Jiewen Yao, Zimmer Vincent, "A_Tour_Beyond_BIOS_Launching_Standalone_SMM_Drivers_in_PEI_using_the_EFI_Developer_Kit_II", 2015, http://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Creating_the_Intel_Firmware_Support_Package_Version_1_1_with_the_EFI_Developer_Kit_II.pdf

[SMM10] Jiewen Yao, Zimmer Vincent, "A_Tour_Beyond_BIOS_Supporting_SMM_Resource_Monitor_using_the_EFI_Developer_Kit_II", 2015, https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Supporting_SMM_Resource_Monitor_using_the_EFI_Developer_Kit_II.pdf

[SMM11] Steve Weis, "Protecting Data In Use From Firmware And Physical Attacks", 2014, https://www.blackhat.com/docs/us-14/materials/us-14-Weis-Protecting-Data-In-Use-From-Firmware-And-Physical-Attacks.pdf

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

[TrustedPlatform] David Grawrock, "Dynamics of a Trusted Platform: A Building Block Approach", 2009, http://www.amazon.com/Dynamics-Trusted-Platform-Building-Approach/dp/1934053171

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.5 www.uefi.org

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

[UEFI Overview] Zimmer, Rothman, Hale, "UEFI: From Reset Vector to Operating System," Chapter 3 of *Hardware-Dependent Software*, Springer, February 2009

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

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 Software and Services Group (SSG) at Intel Corporation. Jiewen is a BIOS security researcher, co-invented a VMM in BIOS in patent US007827371 using PI DXE infrastructure.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer with the Software and Services Group (SSG) at Intel Corporation. Vincent chairs the UEFI Security and Networking Subteams in the UEFI Forum.