A Tour Beyond BIOS - Memory Protection in UEFI BIOS

TABLE OF CONTENTS

A Tour Beyond BIOS - Memory Protection in UEFI BIOS

Revision 1.0

04/30/2025 09:32:55

Jiewen Yao, Intel Corporation

Vincent J. Zimmer , Intel Corporation

Acknowledgements

Revision History

Revision	Revision History	Date
1.0	Initial release.	March 2017

EXECUTIVE SUMMARY

Introduction

Data execution protection (DEP) is intended to prevent an application or service from executing code from a non-executable memory region. This helps prevent certain exploits that store code via a buffer overflow. [WindowsHeap] shows 4 of 7 exploitation techniques that can be mitigated by DEP and ASLR (Address Space Layout Randomization). [DEP] also shows 14 of 19 exploits from popular exploit kits that fail with DEP enabled. Besides Windows, the Unix/Linux community also has similar non-executable protection [PaX].

In the white paper [MemMap], we discussed DEP and the limitation of enabling DEP in UEFI firmware. In [SecurityEnhancement], we only discussed the DEP for protecting the stack and setting the not-present page for detecting `NULL` address accesses and as the guard page. In this document we will have a more comprehensive discussion of the DEP adoption in the current UEFI firmware to harden the pre-boot phase.

MEMORY PROTECTION IN SMM

The SMM is an isolated execution environment according to Intel(R) 64 and IA-32 Architectures Software Developer's Manual [IA32SDM]. The UEFI Platform Initialization [PI] specification volume 4 defines the SMM infrastructure. Figure 1 shows the SMM memory protection. **RO** designates read-only memory. **XD** designates execution-disabled memory.

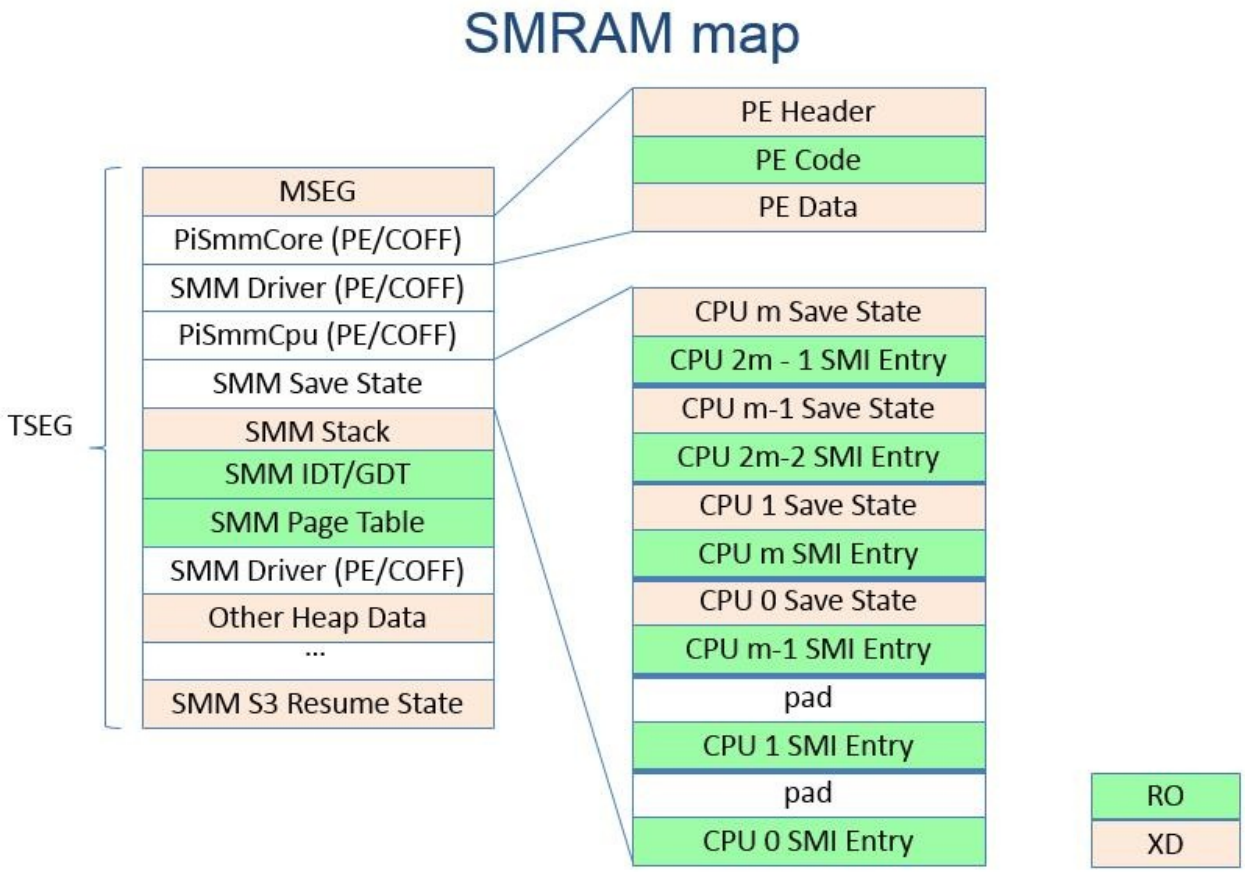

Figure 1 - SMRAM memory protection

Protection for PE image

In UEFI/PI firmware, the SMM image is a normal PE/COFF image loaded by the SmmCore. If a given section of the SMM image is page aligned, it may be protected according to the section attributes, such as read-only for the code and non-executable for data. See the top right of Figure 1.

In EDK II, the PiSmmCore (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/PiSmmCore/MemoryAttributesTable.c) checks the PE image alignment and builds an `EDKII_PI_SMM_MEMORY_ATTRIBUTES_TABLE` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/PiSmmMemoryAttributesTable.h) to record such information. If the PI SMM image is not page aligned, this table will not be published. If the `EDKII_PI_SMM_MEMORY_ATTRIBUTES_TABLE` is published, that means the `EfiRuntimeServicesCode` contains only code and it is `EFI_MEMORY_RO`, and the `EfiRuntimeServicesData` contains only data and it is `EFI_MEMORY_XP`.

Later the PiSmmCpu driver (https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/SmmCpuMemoryManagement.c)` SetMemMapAttributes()` API consumes the `EDKII_PI_SMM_MEMORY_ATTRIBUTES_TABLE` and sets the page

table attribute.

There are several assumptions to support the PE image protection in SMM:

1. The PE code section and data sections are not merged. If those 2 sections are merged, a #PF exception might be generated because the CPU might try to write a RO data item in the data section or execute a non-executable (NX) instruction in code section.
2. The PE image can be protected if it is page aligned. There should not be any self-modified-code in the code region. If there is, a platform should not set this PE image to be page aligned.

A platform may disable the XD in the UEFI environment, but this does not impact the SMM environment. The SMM environment may choose to always enable the XD upon SMM entry, and restore the XD state at the SMM exit point.

Protection for stack and heap

The PiSmmCore maintains a memory map internally. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/PiSmmCore/Page.c) If an SMM module allocates the data with `EfiRuntimeServicesCode`, this data is marked as the code page. If the SMM module allocates the data with `EfiRuntimeServicesData`, this data is marked as the data page. This information is also exposed via the `EDKII_PI_SMM_MEMORY_ATTRIBUTES_TABLE`.

The same RO and XD policy is also applied to the normal SMM data region, such as stack and heap.

Protection for critical CPU status

Besides the PE image, the Intel X86 architecture has some special architecture-specific regions that need to be protected as well.

SMM EntryPoint

When a hardware SMI occurs, the Intel X86 CPU jumps to an SMM entry point in order to execute the code at this location. This SMM entry point is not inside of a normal PE image, so we also need to protect this region. See the bottom right of figure 2.

According to [IA32SDM], the SMM entry point is at a fixed offset from SMBASE. In EDK II, the SMBASE and the SMM save state area are allocated at https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/PiSmmCpuDxeSmm.c `PiCpuSmmEntry()`. Both the SMM entry point and the SMM save state are allocated as a CODE page. Later https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/SmmCpuMemoryManagement.c `PatchSmmSaveStateMap()` patches the SMM entry point to be read-only and the SMM save state to be non-executable.

GDT/IDT

The GDT defines the base address and the limit of a code segment or a data segment. If the GDT is updated, the code might be redirected to a malicious region. As such, the GDT should be set to read-only.

The IDT defines the entry point of the exception handler. If the IDT is updated, the malicious code may trigger an exception and jump to a malicious region. As such, the IDT should be set to read-only as well.

This work is done by `PatchGdtIdtMap()` at https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/X64/SmmFuncsArch.c.

However, the IA32 version GDT cannot be set to read-only if the stack guard feature is enabled. (https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/Ia32/SmmFuncsArch.c) The reason is that the IA32 stack guard needs to use a "task switch" to switch the stack, and the task switch needs to write the GDT and Task-State Segment (TSS). The X64 version of the GDT does not have such a problem because the X64 stack guard uses "interrupt stack table (IST)" to switch the stack. For details of the stack switch and exceptions, please refer to [IA32SDM].

Page Table

In an X86 CPU, we rely on the page table to set up the read-only or non-executable region. In order to prevent the page table itself from being updated, we may need to set the page table itself to be read-only.

The work is done at
https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/X64/PageTbl.c
`SetPageTableAttributes()`.

However, setting a page table to be read-only may break the original dynamic paging feature in SMM. There is a (PCD) `PcdCpuSmmStaticPageTable` to determine if the platform wants to enable the static page table or the dynamic page table.

If `PcdCpuSmmStaticPageTable` is FALSE, the PiSmmCpu uses the original dynamic paging policy, namely the the PiSmmCpu only sets 4GiB paging by default. If the PiSmmCpu needs to access above 4GiB memory locations, a page fault exception (#PF) exception is triggered and an above-4GiB mapping is created in the page fault handler.

If `PcdCpuSmmStaticPageTable` is TRUE, the PiSmmCpu will try to set the read-only attribute for the page table.

Figure 2 shows the mapping of the protection.

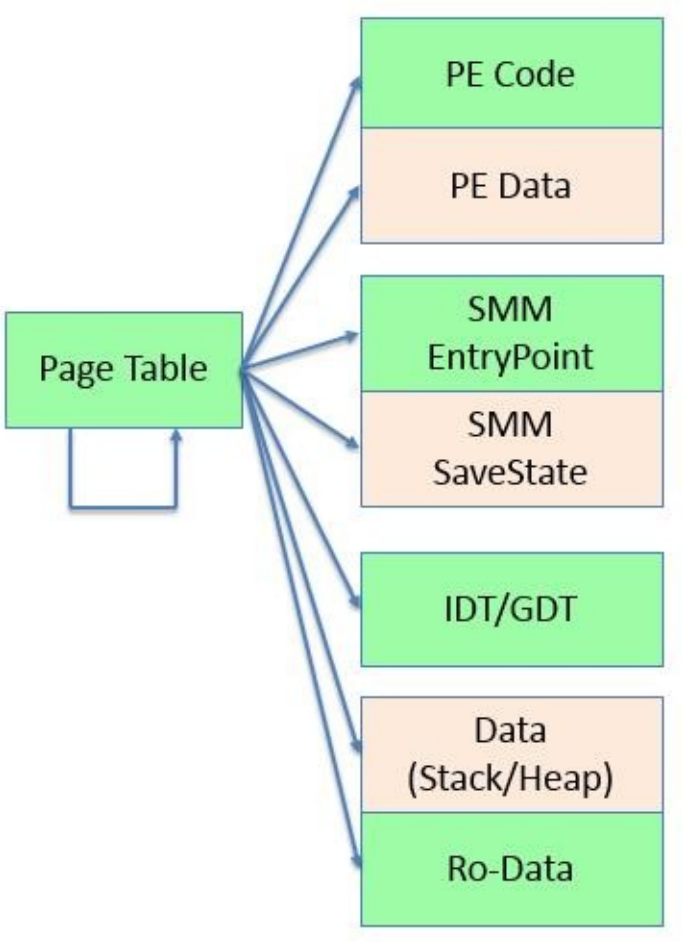

Figure 2 - Mapping of Protection in SMM

Life cycle of the protection

In a normal boot, the page table based protection is configured by the PiSmmCpu driver just after the SmmReadyToLock event by `PerformRemainingTasks()` at https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/PiSmmCpuDxeSmm.c. All read-only data must be ready before `SmmReadyToLock`.

In an S3 resume, the protection is disabled during SMBASE relocation because the PiSmmCpu needs to set up the environment. The PiSmmCpu uses SmmS3Cr3, which is generated by `InitSmmS3Cr3()` at https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/X64/SmmProfileArch.c with 4G paging only. After the SMBASE relocation is done, all the protection takes effect up receipt of the next SMI by `PerformPreTasks()` at https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/PiSmmCpuDxeSmm.c.

If there is an additional lock that needs to be set, it can be done in `SmmCpuFeaturesCompleteSmmReadyToLock()` API (defined in https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/Include/Library/SmmCpuFeaturesLib.h).

SMRAM Size Overhead

PE image

In order to protect the PE code and data sections, we must set the PE image section alignment to be 4K.

In EDK II, the default PE image alignment is 0x20 bytes. Assuming one PE image has 3 sections (1 header, 1 code section, 1 data section), average overhead for one PE image is `(4K * 3) / 2 = 6K`.

If a platform has n SMM images, the average of the overhead is `6K * n`.

Page Table

In order to protect the page table itself, we must use the static page table instead of the dynamic on-demand page table.

The size of the dynamic paging is fixed. We need 6 fixed pages (24K) and 8 on-demand pages (32K). The total size of the page table is 56K in this case.

The size of the static page table depends upon 2 things: 1) 1G paging capability, 2) max supported address bit. A rough estimation is below:

1. If 1G paging is supported,
2. 32 bit addressing need (1+1+4) pages = 24K. (still use 2M paging for below 4G memory)
3. 39 bit addressing need (1+1+4) pages = 24K.
4. 48 bit addressing need (1+512) pages = 2M.
5. If 1G paging is not supported, 2M paging is used.
6. 32 bit addressing need (1+1+4) pages = 24K.
7. 39 bit addressing need (1+1+512) pages = 2M.
8. 48 bit addressing need (1+512+512512) pages = 1G. < - This seems ***not** acceptable.

The maximum address bit is determined by the (CPU_HOB) if it is present, or the physical address bit returned by the CPUID instruction if the CPU_HOB is not present.
(https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/X64/PageTbl.c,

`CalculateMaximumSupportAddress()`) A platform may set the CPU_HOB based upon the addressing capability of the memory controller or the CPU.

Performance Overhead

1. The SMRAM protection setup is a one-time activity. It happens just after the SmmReadyToLock event. We do not observe too much impact to the system firmware boot performance. The activity only takes some small number of milliseconds.
2. The SMRAM runtime protection is based upon the page table. No additional CPU instruction is needed. As such, there is zero SMM runtime performance impact to have this protection.

Non SMRAM access in SMM

Besides the SMRAM, the SMM memory protection also limits the access to the non-SMRAM region.

First, the non-SMRAM region must be set to be non-executable because the SMM entities should not call any code outside SMRAM. Code outside of SMRAM might be controlled by malicious software.

This protection work is done by `InitPaging()` at https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/SmmProfile.c

Second, because of the security concerns regarding SMM entities accessing VMM memory, [WindowsWSMT] [Wsmt.docx] and [MicrosoftHV] introduced the Windows SMM Security Mitigations Table (WSMT). A platform needs to report the WSMT table in order to declare that the SMI handler will validate the SMM communication buffer.

As we discussed in [SecureSmmComm], the SMI handler should check if the SMM communication buffer is from a fixed region, (EfiReservedMemoryType/ EfiACPIMemoryNVS/ EfiRuntimeServicesData/ EfiRuntimeServicesCode). However, this is a passive check. If a SMI handler does not include such a check, it is hard to detect.

A better way is to use an active check. The PiSmmCpu driver sets the non-fixed DRAM region (EfiLoaderCode/ EfiLoaderData/ EfiBootServicesCode/ EfiBootServicesData/ EfiConventionalMemory/ EfiUnusableMemory/ EfiACPIReclaimMemory) to be not-present in the page tables after the SmmReadyToLock event.

As such, if a platform SMI handler does not include the check recommended in [SecureSmmComm], the system will get #PF exception within SMM on such an attack.

This protection work is done by `SetUefiMemMapAttributes()` at https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/SmmCpuMemoryManagement.c.

Figure 3 shows final image layout.

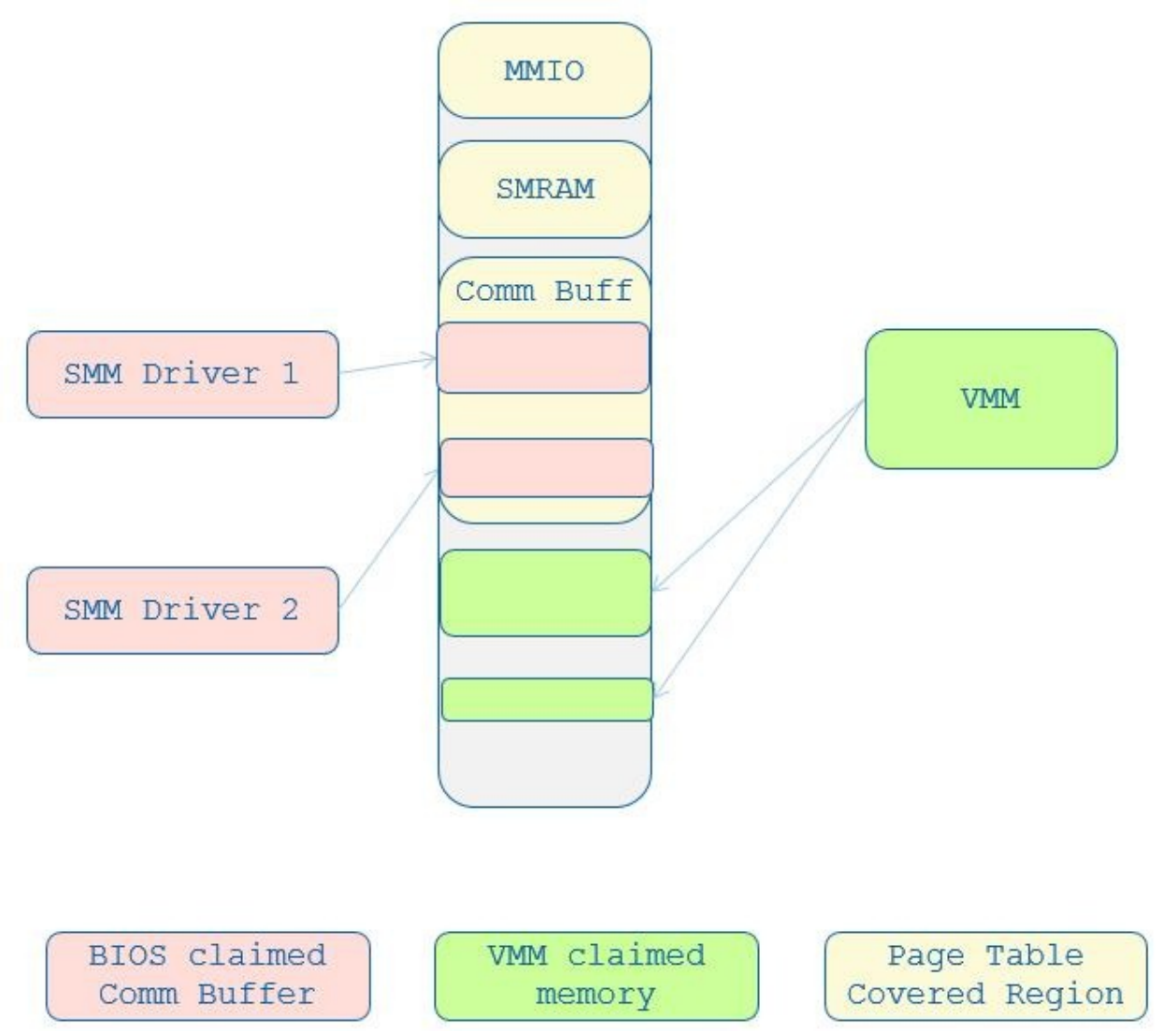

Figure 3 - Page table enforced memory layout

The assumption for non-SMRAM access in SMM is described in [SecureSmmComm]. Besides that, this solution assumes that all DRAM regions are added to the Global Coherency Domain (GCD) management before EndOfDxe, so that the UEFI memory map can return all DRAM regions. If there are more regions added to the GCD after EndOfDxe, those regions are not set to not-present in the page table. NOTE: The SMM does not set the not-present bit for the GCD **EfiGcdMemoryTypeNonExistent** memory, because this type of memory may be converted to the other types, such as **EfiGcdMemoryTypeReserved**, or **EfiGcdMemoryTypeMemoryMappedIo**, which might be accessed by the SMM later.

Limitation

Setting up RO and NX attribute for SMRAM is a good enhancement to prevent a code overriding attack. However it has some limitations:

1. It cannot resist a Return-Oriented-Programming (ROP) attack. [ROP]. We might need ASLR to mitigate the ROP attack. [ASLR] With the code region randomized, an attacker cannot accurately predict the location of instructions in order to leverage gadgets.
2. Not all important data structure are set to Read-Only. This is the current SMM driver limitation. The SMM driver can be updated to allocate the important structures to be read-only instead of a read-write global variable.

To set not-present bit for non-fixed DRAM region in SmmReadyToLock is a good enhancement to enforce the protection policy. However, it cannot cover below cases:

1. Memory Hot Plug. Take a server platform as the example, A RAS server may hot plug more DRAM during OS runtime, and rely on SMM to initialize those DRAM. This SMM Memory Initialization module may need access the DRAM for the memory test.
2. Memory Mapped IO (MMIO). Ideally, not all MMIO regions are configured to be accessible to SMM. Some MMIO BARs are important such as VTd or SPI controller. VTd BAR is important because OS need setup VTd to configuration the DMA protection. SPI controller BAR is important because BIOS SMM handler need access it to program the flash device. It should be a platform policy to configure which one should be accessible. The SMI handler must consider the case that the MMIO BAR might be modified by the malicious software and check if the MMIO BAR is in the valid region.

Compatibility Considerations

1. So far, we have not observed self-modified-code in SMM image or executable code in data section. As such, we believe the PE image protection is compatible.
2. The protection for the SMM communication buffer may cause a #PF exception in SMM if the SMI handler does not perform the check recommended in [SecureSmmComm].
3. Some legacy Compatibility Support Module (CSM) drivers may need co-work with SMM module. Then the SMM driver need access the legacy region. As such these memory regions should be allocated as ReservedMemory, such as BIOS data area (BDA) or extended BIOS data area (EBDA).

Call for action

In order to support SMM memory protection, the firmware need configure SMM driver to be page aligned:

1. Override link flags below to support SMM memory protection.

```
    [BuildOptions.common.EDKII.DXE_SMM_DRIVER,
    BuildOptions.common.EDKII.SMM_CORE]
    MSFT:*_*_*_DLINK_FLAGS = /ALIGN:4096
  GCC:*_*_*_DLINK_FLAGS = -z common-page-size=0x1000
```

2. Evaluate if SMRAM size is big enough.

Summary

This section introduces the memory protection in SMM.

MEMORY PROTECTION IN UEFI

In the white paper [MemMap], we discussed to how to report the runtime memory attribute by using `EFI_MEMORY_ATTRIBUTES_TABLE`, so that OS can apply the protection for the runtime code and data. This may bring some compatibility concerns if we choose to adopt the full DEP protection for the entire UEFI memory.

In order to resolve the compatibility concerns, we can define a policy-based setting to enable partial NX and RO protection for the UEFI memory region. The detailed information will be discussed below.

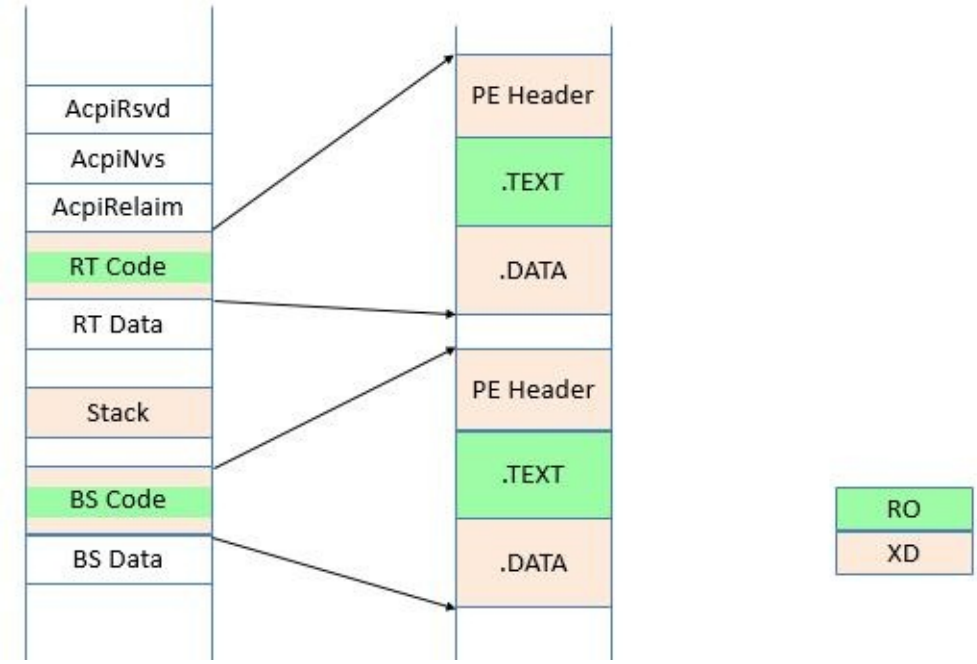

Figure 4 - UEFI memory protection

Protection for PE image

The DXE core may apply a pre-defined policy to set up the NX attribute for the PE data region and the RO attribute for the PE code region.

1. The image is loaded by the UEFI boot service - `LoadImage()`. If an image is loaded in some other way, the DXE core does not have such knowledge and the DXE core cannot apply any protection.
2. The image section is page aligned. If an image is not page aligned, the DXE core cannot apply the page level protection.
3. The protection policy can be based upon a PCD `PcdImageProtectionPolicy`. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/MdeModulePkg.dec) Whenever a new image is loaded, the DxeCore checks the source of the image and then decides the policy of the protection. The policy could be to enable the protection if the sections are aligned, or disable the protection. The platform may choose the policy based upon the need. For example, if a platform thinks the image from the firmware volume should be capable of being protection, it can set protection for IMAGE_FROM_FV. But if a platform is not sure about a PCI option ROM or a file system on disk, it can set no-protection.

There are assumptions for the PE image protection in UEFI:

1. [Same as SMM] The PE code section and data sections are not merged. If those 2 sections are merged, a #PF exception might be generated because the CPU may try to write a RO data in data

section or execute a NX instruction in the code section.
2. [Same as SMM] The PE image can be protected if it is page aligned. There should not be any self-modifying-code in the code region. If there is, a platform should not set this PE image to be page aligned.
3. A platform may not disable the XD in the DXE phase. If a platform disables the XD in the DXE phase, the X86 page table will become invalid because the XD bit in page table becomes a RESERVED bit. The consequence is that a #PF exception will be generated. If a platform wants to disable the XD bit, it must happen in the PEI phase.

In EDK II, the DXE core image services calls `ProtectUefiImage()` on image load and `UnprotectUefiImage()` on image unload. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Image/Image.c) Then `ProtectUefiImageCommon()` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Misc/MemoryProtection.c) calls `GetUefiImageProtectionPolicy()` to check the image source and protection policy and parses PE alignment. If all checks pass, `SetUefiImageProtectionAttributes()` calls `SetUefiImageMemoryAttributes()`. Finally, `gCpu->SetMemoryAttribute()` sets **EFI_MEMORY_XP** or **EFI_MEMORY_RO** for the new loaded image , or clears the protection for the old unloaded image. When the CPU driver gets the memory attribute setting request, it updates page table.

The X86 CPU driver https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/CpuDxe/CpuDxe.c `CpuSetMemoryAttributes ()` calls https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/CpuDxe/CpuPageTable.c, `AssignMemoryPageAttributes()` to setup page table.

The ARM CPU driver https://github.com/tianocore/edk2/blob/master/ArmPkg/Drivers/CpuDxe/CpuMmuCommon.c `CpuSetMemoryAttributes()` also has similar capability.

If an image is loaded before CPU_ARCH protocol is ready, the DXE core just skips the setting. Later these images protection will be set in CPU_ARCH callback function - `MemoryProtectionCpuArchProtocolNotify()` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Misc/MemoryProtection.c).

In `ExitBootServices` event, `MemoryProtectionExitBootServicesCallback()` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Misc/MemoryProtection.c) is invoked to unprotect the runtime image, because the runtime image code relocation need write code segment at `SetVirtualAddressMap()`.

Protection for stack and heap

[UEFI] specification allows

> "Stack may be marked as non-executable in identity mapped page tables."

As such, we set up the NX stack (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/DxeIplPeim/X64/VirtualMemory.c, `CreateIdentityMappingPageTables()`).

The heap protection is based upon the policy, because we already observed some unexpected usage in [MemMap] white paper. A platform needs to configure a PCD `PcdDxeNxMemoryProtectionPolicy` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/MdeModulePkg.dec) to indicate which type of memory can be set to NX in the page table. The DxeCore `ApplyMemoryProtectionPolicy()` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Misc/MemoryProtection.c) consumes the PCD after the memory allocation service and sets NX attribute for the allocated memory by using CPU_ARCH protocol.

Before CPU_ARCH protocol is ready, the protection takes no effect. In CPU_ARCH callback function - `MemoryProtectionCpuArchProtocolNotify()` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Misc/MemoryProtection.c), the `InitializeDxeNxMemoryProtectionPolicy()` is called to get current memory map and setup the NX protection.

In addition, we may use some special techniques, such as the guard page, to apply the protection for the allocated memory in order to detect a buffer overflow. This is discussed in [SecurityEnhancement] white paper.

Life cycle of the protection

The UEFI image protection starts when the CpuArch protocol is ready. The UEFI runtime image protection is torn down at `ExitBootServices()`, the runtime image code relocation need write code segment at `SetVirtualAddressMap()`. We cannot assume OS/Loader has taken over page table at that time.

The UEFI heap protection also starts when the `CpuArch` protocol is ready.

The UEFI stack protection starts in `DxeIpl`, because the region is fixed and it can set directly.

The UEFI firmware does not own page tables after `ExitBootServices()`, so the OS would have to relax protection of runtime code pages across `SetVirtualAddressMap()`, or delay setting protections on runtime code pages until after `SetVirtualAddressMap()`. OS may set protection on runtime memory based upon EFI_MEMORY_ATTRIBUTES_TABLE later.

Size Overhead

1. Runtime memory overhead (visible to OS) : The size overhead of the runtime PE image is the same as the overhead of the SMM PE image. If a platform has n runtime images, the average amount overhead is `6K * n`.
2. Boot time memory overhead (invisible to OS) : The size of the overhead for the boot time PE image is the same as the overhead of the SMM PE image. If a platform has n boot time images, the average overhead is `6K * n`.

If the NX protection for data is enabled, the size of the page table is increased because we need set fine granularity page level protection.

The size overhead of the boot time page table is also same as for the SMM static page table. Please refer to the SMM section for the size calculation based upon the 1G paging capability and max supported address bit.

Limitation

The protection in the UEFI is limited to the PE image and the stack at this moment because of the compatibility concerns. The limitations of the UEFI memory protection are:

1. Not all images are protected to be NX and RO. The protection is based upon the policy.
2. Not all heap regions are protected to be NX due to the compatibility concern. We observed that both Windows boot loader and Linux boot loader may use the LoaderData type for the code. The heap protection is based upon the policy.
3. [Same as SMM] The protection cannot resist ROP attack.
4. [Same as SMM] Not all important data structures are set to ReadOnly.

Compatibility Consideration

A platform may need to evaluate and select the image protection policy based upon the capability of the platform image, Option ROM, and OS loader. For platform images, the Compatibility Support Module (CSM) and the EDK-I Compatibility Package (ECP) modules should be considered. If a platform observes the compatibility issues, it should choose 1) to disable the protection, or 2) to fix the compatibility issue and enable the protection.

Call for action

In order to support UEFI memory protection, the firmware need configure UEFI driver to be page aligned:

1. Override link flags below to support UEFI runtime attribute table, so that OS can protect the runtime memory.

```
[BuildOptions.IA32.EDKII.DXE_RUNTIME_DRIVER, BuildOptions.X64.EDKII.DXE_RUNTIME_DRIVER]
MSFT:*_*_*_DLINK_FLAGS = /ALIGN:4096
GCC:*_*_*_DLINK_FLAGS = -z common-page-size=0x1000
```

2. Override link flags below to support UEFI memory protection.

```
[BuildOptions.common.EDKII.DXE_DRIVER,
BuildOptions.common.EDKII.DXE_CORE,
BuildOptions.common.EDKII.UEFI_DRIVER, BuildOptions.common.EDKII.UEFI_APPLICATION]
MSFT:*_*_*_DLINK_FLAGS = /ALIGN:4096
GCC:*_*_*_DLINK_FLAGS = -z common-page-size=0x1000
```

3. Evaluate if the UEFI memory size is big enough to hold the split page table.
4. Evaluate if the DXE image can be protected.
5. Set proper `gEfiMdeModulePkgTokenSpaceGuid.PcdImageProtectionPolicy`.
6. Set proper `gEfiMdeModulePkgTokenSpaceGuid.PcdDxeNxMemoryProtectionPolicy`.

Summary

This section introduces the memory protection in UEFI.

GLOSSARY

ASLR - Address Space Layout Randomization.

BDA - BIOS Data Area.

CSM - Compatibility Support Module.

DEP - Data Execution Protection.

EBDA - Extended BIOS Data Area

HOB - Hand off block. See [PI].

MMIO - Memory Mapped I/O.

NX - No Execution. See DEP.

PE/COFF - Portable Executable and Common Object File Format. The executable file format for UEFI.

ROP - Return-oriented programming

RO - Read Only.

RW - Read/Write.

PCD - Platform configuration database. See [PI].

PF - Page Fault Exception.

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

SPI - Serial Peripheral Interface.

TSS - Task-state segment. See [IA32 SDM].

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.

VTd - Virtualization for Directed IO. See [VTd]

WP - Write Protect.

XD - Execution Disable. See DEP.

XP - Execution Protected. See DEP.

REFERENCES

[ASLR] Address Space Layout Randmization, https://en.wikipedia.org/wiki/Address_space_layout_randomization

[DEP] Exploit Mitigation Improvements in Windows 8, Ken Johnson, Ma, Miller, http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

[IA32SDM] Intel(R) 64 and IA-32 Architectures Software Developer's Manual, www.intel.com https://software.intel.com/en-us/articles/intel-sdm

[MemMap] A Tour Beyond BIOS Memory Map And Practices in UEFI BIOS, Jiewen Yao, Vincent Zimmer, 2016 https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Memory_Map_And_Practices_in_UEFI_BIOS_V2.pdf

[PaX] PaX Home Page, https://pax.grsecurity.net/

[ROP] Return-oriented programming, https://en.wikipedia.org/wiki/Return-oriented_programming

[SecureSmmComm] A Tour Beyond BIOS Secure SMM Communication, Jiewen Yao, Vincent Zimmer, Star Zeng, 2016, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Secure_SMM_Communication.pdf

[SecurityEnhancement] A Tour Beyond BIOS Securiy Enhancement to Mitigate Buffer Overflow in UEFI, Jiewen Yao, Vincent Zimmer, 2016, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Securiy_Enhancement_to_Mitigate_Buffer_Overflow_in_UEFI.pdf

[SecurityDesign] A Tour Beyond BIOS Security Design Guide in EDK II, Jiewen Yao, Vincent Zimmer, 2016, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Security_Design_Guide_in_EDK_II.pdf

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

[VTd] Intel(R) Virtualization Technology for Directed I/O: Spec, http://www.intel.com/content/www/us/en/embedded/technology/virtualization/vt-directed-io-spec.html

[WindowsHeap] Preventing the exploitation of user mode heap corruption vulnerabilities, 2009, https://blogs.technet.microsoft.com/srd/2009/08/04/preventing-the-exploitation-of-user-mode-heap-corruption-vulnerabilities/

[WindowsInternal] Windows Internals, 6th edition, Mark E. Russinovich, David A. Solomon, Alex Ionescu, 2012, Microsoft Press. ISBN-13: 978-0735648739/978-0735665873

[WindowsWSMT] 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

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

Authors

Jiewen Yao jiewen.yao@intel.com is EDK II BIOS architect, EDK II FSP package maintainer, EDK II TPM2 module maintainer, EDK II ACPI S3 module maintainer, with 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 with the Software and Services Group at Intel Corporation. Vincent chairs the UEFI Security and Network Sub-teams in the UEFI Forum.