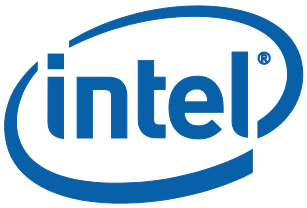


White Paper

A Tour Beyond BIOS – Security Enhancement to Mitigate Buffer Overflow in UEFI

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

October 2016

Executive Summary

Introduction

A buffer overflow is “one of the most important exploitation techniques in the history of computer security.” [Tanenbaum] “Buffer overflows are ideally suited for introducing three of the most important protection mechanisms available in most modern systems: stack canaries, data execution protection, and address-space layout randomization.” [Tanenbaum] However, the current UEFI firmware implementation only adopted a few of these mechanisms. This paper will introduce how to enable the protection mechanisms in UEFI firmware to harden the pre-boot phase.

Table of Contents

Stack Canaries

One of the most important buffer overflow attacks is the first Internet worm, written by Robert Morris Jr. in 1988. It modified the return address on the stack, injected malicious code, then it controlled the system. (See figure 1-1 Stack Smashing Buffer Overflow Attack)

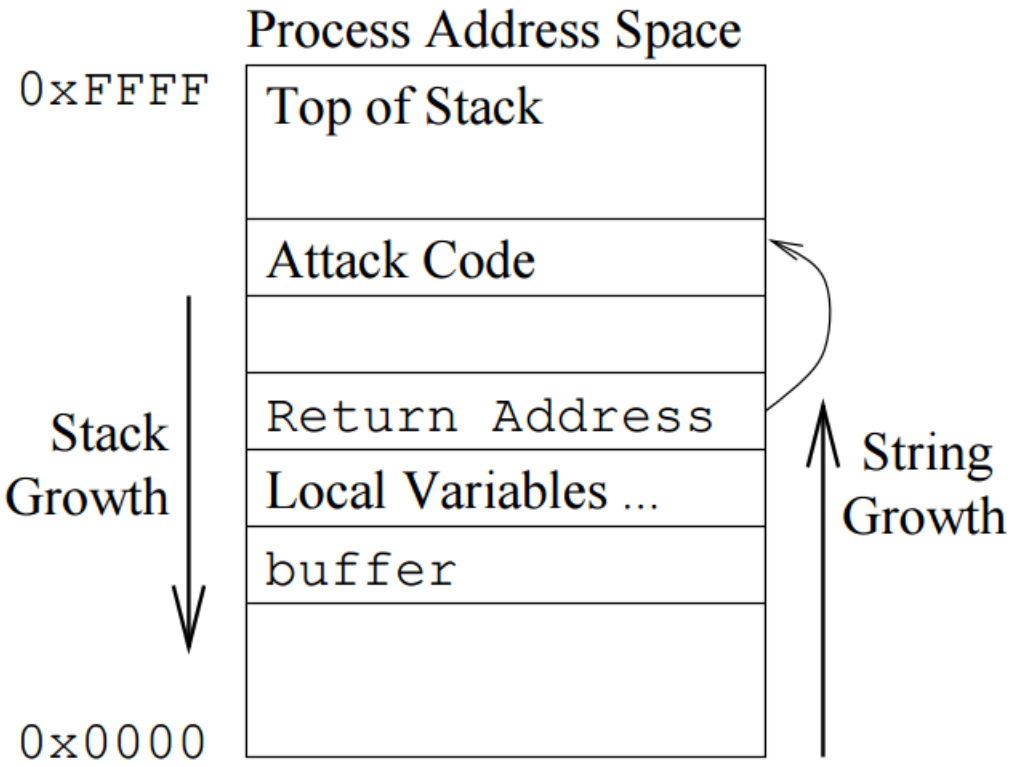


Figure 1-1 Stack Smashing Buffer Overflow Attack (Source: [StackCheck])

In 1999, StackCheck was introduced to prevent buffer overflow attack on a stack. [StackCheck]
When the program makes a function call, it puts a random digital canary on top of the return address on the stack. When the function returns, the program checks if the canary data is modified. If it is modified, there must be something wrong and a special failure reporting function is invoked before the program returns back to the function address on the stack. (See figure 1-2 Canary Word Next to Return Address)

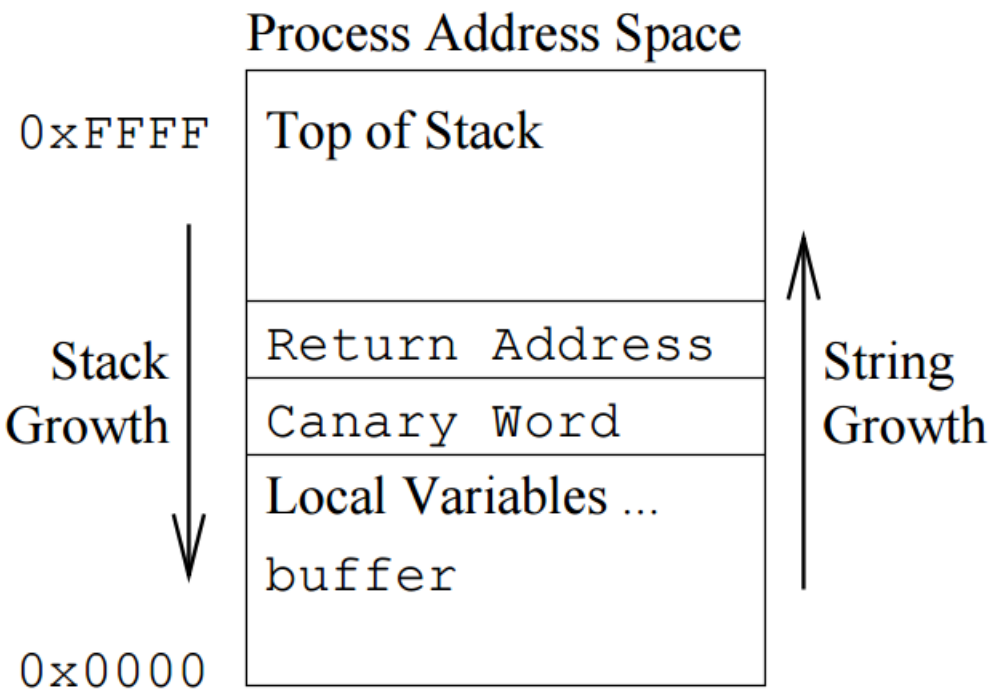


Figure 1-2 Canary Word Next to Return Address (Source: [StackGuard])

Stack canary is a software feature. It is supported by most compilers.

Stack Check Support in Microsoft Visual Studio

Microsoft Visual Studio supports a stack guard function. “Compiler Security Checks In Depth” [MSVC] introduces the detail on how it works. There are 2 compiler options related: `/GS` and `/RTCs`.

`/GS` [MSVC_GS] is designed to detect some buffer overruns that overwrite a function's return address. It is similar to the stack guard feature described above.

`/RTCs` [MSVC_RTC] is designed to put 2 tags around (before and after) all individual buffers allocated on the stack. Therefore, both overruns and underflows can be caught.

See figure 1-3 Microsoft Stack Check.

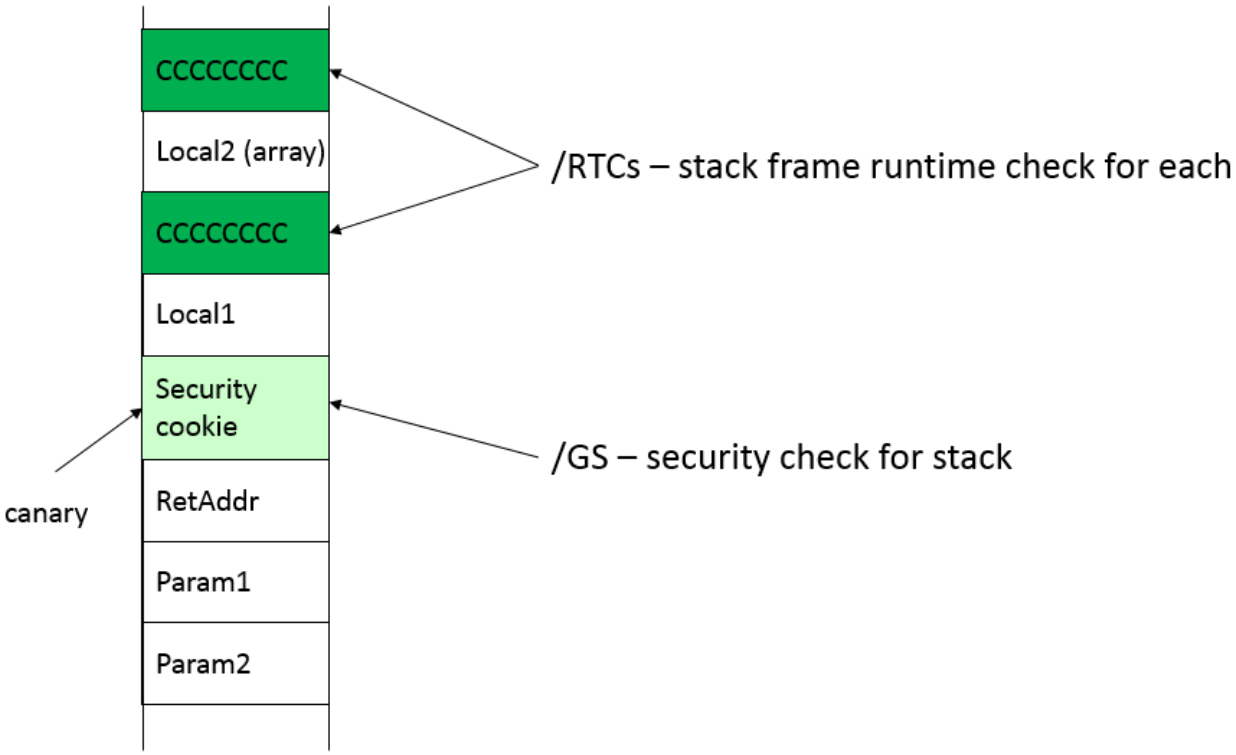


Figure 1-3 – Microsoft Stack Check

Stack Check Support in GCC

The first stack guard is supported in GCC. Current GCC supports: `-fstack-protector`, `-fstack-protector-all`, and `-fstack-protector-strong`. [StackCanaries].

`-fstack-protector-strong` is recommended because "this option tries to hit the balance between an over-simplified version and an over-killing protection schema". [GCC]

Enable Stack Check in EDK II

Current EDK II uses `/GS-` for MSVC and `-fno-stack-protector` for GCC. The stack check feature is disabled by default. The reason is that EDK II does not link against any compiler provided libraries. If `/GS` or `-fstack-protector` is enabled, the link will fail due to no symbol detected for `__security_cookie/ __security_check_cookie()` or `__stack_chk_guard/__stack_chk_fail()`.

In order to enable a stack check, we provide an implementation for the above symbols at https://github.com/jyao1/SecurityEx/tree/master/StackCheckPkg/Library/StackCheckLib.

As such, any drivers or applications can use `/GS` or `-fstack-protector-strong` to prevent the stack smash attack. The sample driver https://github.com/jyao1/SecurityEx/tree/master/StackCheckPkg/Test/StackCookieTest shows how stack check works in UEFI.

Future work

The current stack error handler just uses `CpuDeadLoop()`. It is enough in preboot phase, but it is a bad idea in OS runtime. If a UEFI runtime module enables stack check, we need figure out a better way to signal the error to operating system. For example, we can use `__fastfail` to notify OS kernel. "The **__fastfail** intrinsic provides a mechanism for a *fast fail* request—a way for a potentially corrupted process to request immediate process termination." [MSVC_FASTFAIL]

If a SMM module enables stack check, the fail program cannot use `__fastfail` directly. It can choose deadloop, reset system, or inject System Control Interrupt (SCI) and do a long jump to a known good point to finish resource cleanup and execute RSM instruction to return to OS, then fail in OS eventually.

From a software engineering standpoint, the UEFI Platform Initialization [PI] Specification defines a ReportStatusCode service for SMM, generalized to "MM" for SMM and TrustZone, and DXE, which extends into UEFI runtime. Exception handling code can invoke ReportStatusCode so that a platform can provide market or product-specific behavior, including a while (1) loop that pulses an LED for a closed box client, all the way up to a multi-socket server that might record the result of the failure in a baseboard management controller (BMC).

Summary

This section introduces the concept of stack canaries and how to enable this feature in EDK II.

Data Execution Protection

Stack smash attacks may inject code. The other possible way to prevent such an attack is to prevent malicious code from executing. Some modern OS's already have Data Execution Protection (DEP) support [DEP][PaX]. DEP may be applied to: [WindowsInternal]

- User mode stacks
- User mode pages not specifically marked as executable
- Kernel mode Stacks
- kernel paged pool (X64)
- kernel session pool (X64)

Research shows 14 of 19 exploits from popular exploit kits fail with DEP enabled. [DEP].

DEP in X86 Processor

DEP is a hardware feature. Intel X86 processor supports the XD (eXecution Disable) bit in the page table. [IA32SDM] This XD bit can be used to indicate that a page is an Execute-Disable Page. In order to enable Data Execution Protection, the operating system needs to set the IA32_EFER.NXE (No-eXecution Enable) bit in IA32_EFER MSR, and then set the XD bit in the CPU PAE page table.

DEP in UEFI specification

[UEFI] specification allows "Stack may be marked as non-executable in identity mapped page tables." UEFI also defines **EFI_MEMORY_ATTRIBUTES_TABLE** to let the OS know which addresses represent runtime code pages and runtime data pages, respectively. As such, the OS may refer to this information in order to setup the protection during OS runtime. The details of this design are discussed in [MemMap].

Enable DEP stack in EDK II

EDK II follows the UEFI specification and defines a PCD for non-executable stack. The DxeIplPeim
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/DxeIplPeim
`HandOffToDxeCore()` function sets up the stack for UEFI. This function references `gEfiMdeModulePkgTokenSpaceGuid.PcdSetNxForStack` to decide if it needs to configure the non-executable stack.

Future work

As discussed in [MemMap], there are some limitations if one wants to enable an entire DEP environment in UEFI pre-boot environment. If the limitation is eliminated in the future, we may configure a DEP environment in the pre-boot environment.

Summary

This section introduces the data execution protection and how to enable it in EDK II. For more details, please refer to [MemMap], where we have provided a detailed discussion before.

Address Space Layout Randomization

Another possible way to prevent such these attacks is to provide a random address. With address space layout randomization (ASLR), the address of every function or data between every run of program is random so that it is hard for an attacker to exploit the system to know where to return.

ASLR is a software feature. It is supported by operating systems today.

ASLR in Windows

Windows supports ASLR. [WindowsInternal] desribes the detail on how executable images, DLL, stack, heap are randomized.

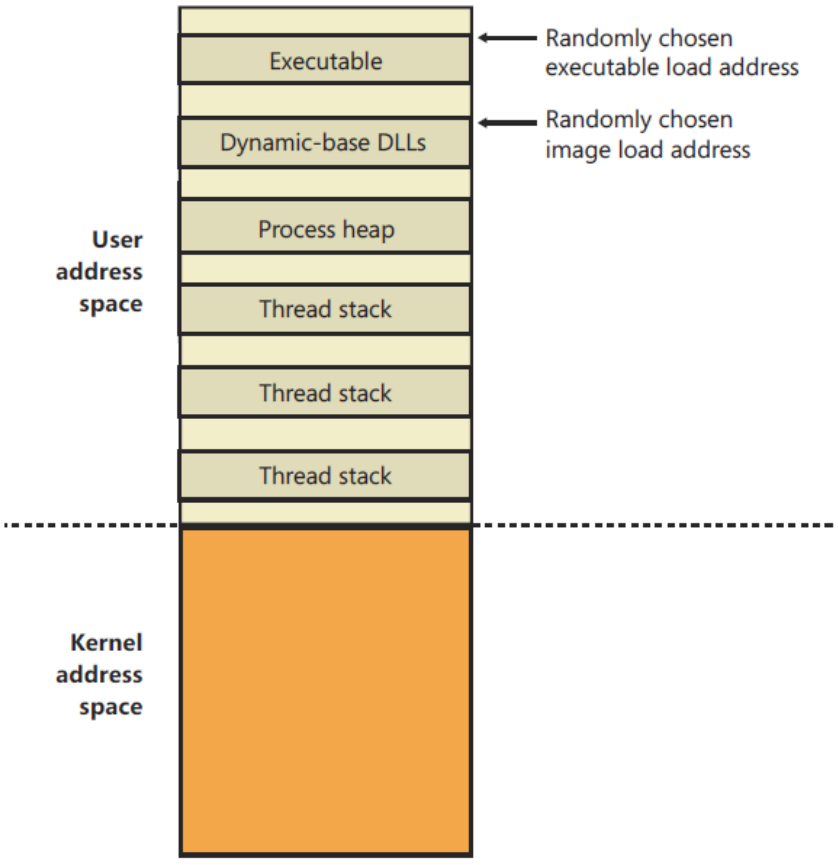


Figure 3-1 Windows OS space layout, source: [WindowsInternal]

[ASLR1] shows the Windows8 HE-ASLR design and entropy number.

Entropy (in bits) by region	Windows 7		Windows 8		
	32-bit	64-bit	32-bit	64-bit	64-bit (HE)
Bottom-up allocations (opt-in)	0	0	8	8	24
Stacks	14	14	17	17	33
Heaps	5	5	8	8	24
Top-down allocations (opt-in)	0	0	8	17	17
PEBs/TEBs	4	4	8	17	17
EXE images	8	8	8	17*	17*
DLL images	8	8	8	19*	19*
Non-ASLR DLL images (opt-in)	0	0	8	8	24

* 64-bit DLLs based below 4GB receive 14 bits, EXEs below 4GB receive 8 bits

ASLR entropy is the same for both 32-bit and 64-bit processes on Windows 7

64-bit processes receive much more entropy on Windows 8, especially with high entropy (HE) enabled

Figure 3-2 Win8 HE-ASLR, source: [ASLR1]

[ASLR2] shows different image layouts during boot.

Figure 3-3 Image layout during boot, source: [ASLR2]

ASLR in *nix

[PaX] also provides an ASLR patch for Linux and OpenBSD. [OpenBSD] and [PIE] provide detailed information on the randomized image layout.

ASLR requirement in UEFI firmware

The current EDK II code does not support address space randomization. The memory allocation algorithm is top-down. In order to support the randomization in the preboot environment, we define below requirement:

1) **The randomization algorithm should keep UEFI firmware boot behavior consistent**. If a system has enough memory for boot, it should be able to boot at any time. If a system is out of memory, it should be out of memory for any boot. If a UEFI firmware boots at some time because the memory is enough, but it may fail at some other time because the randomization cause memory being exhausted, then it is not acceptable solution. This is very important for a resource constrained environment, such as SMM or the PEI phase.
2) **The randomization algorithm should keep OS resume from sleep states required memory being consistent**. An OS may needs to support S4 or S3 resume feature. The OS resume may have some assumption that some special memory is unchanged, such as the runtime memory, or the ACPI memory.
3) **Any individual component may have its own randomization algorithm.** For example, the GDT, IDT, or Page Table are allocated from heap memory. Even if the heap has randomized, the CPU driver can have additional randomization for GDT/IDT/PageTable specifically.

Enable ASLR for UEFI in EDK II

In order to enable address space layout randomization, we provide a sample implementation for randomization in UEFI.

- Randomization control.
  The `gEfiAslrPkgTokenSpaceGuid.PcdASLRMinimumEntropyBits` (https://github.com/jyao1/SecurityEx/blob/master/AslrPkg/AslrPkg.dec) to indicate the ASLR entropy bits. 0 means no randomization.
  The entropy bit controls the how much randomness we want to achieve.

- UEFI stack randomization.
  The stack for DxeCore is allocated in `HandOffToDxeCore()` https://github.com/jyao1/SecurityEx/blob/master/AslrPkg/Override/MdeModulePkg/Core/DxeIplPeim/DxeLoad.c. Before the DXE stack is allocated from heap in the PEI phase, `AllocateRandomPages()` is called to allocate some random pages to shift the PEI heap.

- DxeCore randomization.
  The DxeCore is also loaded from PEI heap by `DxeIplFindDxeCore()` https://github.com/jyao1/SecurityEx/blob/master/AslrPkg/Override/MdeModulePkg/Core/DxeIplPeim/DxeLoad.c. `AllocateRandomPages()` also helps shift the DxeCore memory.

- UEFI heap randomization.
  The heap for DxeCore is reported by PEI and discovered by DxeCore in https://github.com/jyao1/SecurityEx/blob/master/AslrPkg/Override/MdeModulePkg/Core/Dxe/Gcd/Gcd.c. After `CoreInitializeMemoryServices()` adds available memory to UEFI heap, `AllocateRandomPages()` is called to allocate some random pages to shift the UEFI heap.

  NOTE: The OS aware memory, such as runtime, ACPI, and reserved memory are pre-reserved in PEI phase. They are not impacted by the UEFI heap randomization.

  The final memory layout in UEFI is shown in figure 3-4.

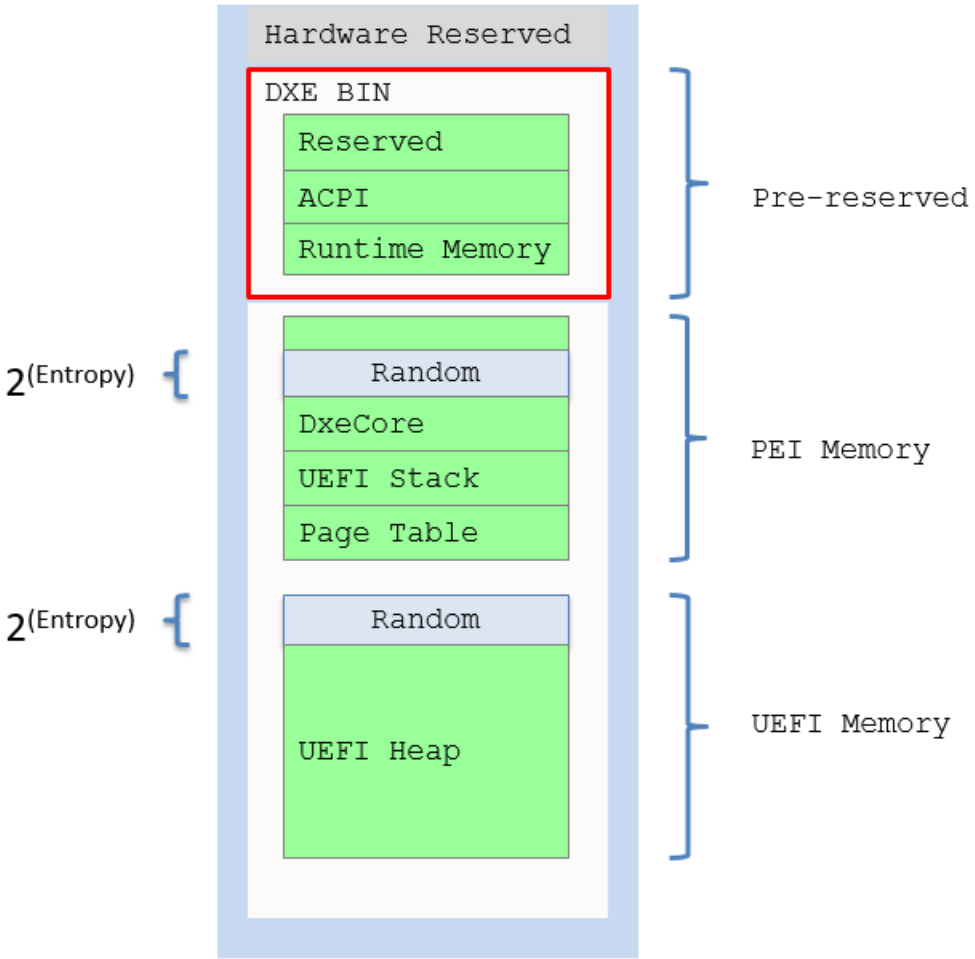


Figure 3-4 UEFI memory layout

- UEFI image randomization.

  The UEFI randomized heap shifts are implemented with a fixed offset. As such, even memory allocation shifts occur with the fixed offset. It is not good enough for a PE/COFF image load.

  For PE/COFF images we use “image shuffle” to randomize the image load order. Whenever the DxeCore discovers a new firmware volume (FV), the DxeCore unconditionally load all the images in this FV with a random order. See figure 3-5 Image shuffle.

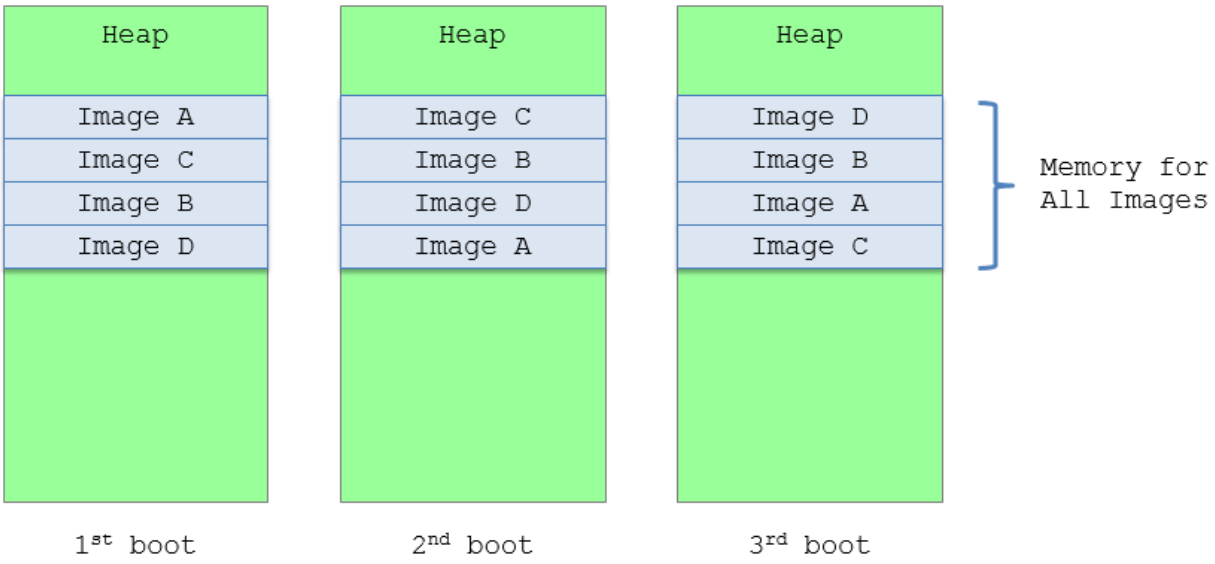


Figure 3-5 Image Shuffle

For example, if an FV contains 4 images – A, B, C, D. The loaded image order in memory is different among the 1st boot, the 2nd boot, and the 3rd boot.

Now let’s see how the Core shuffles images.

The DxeCore maintains the image information in below data structure:

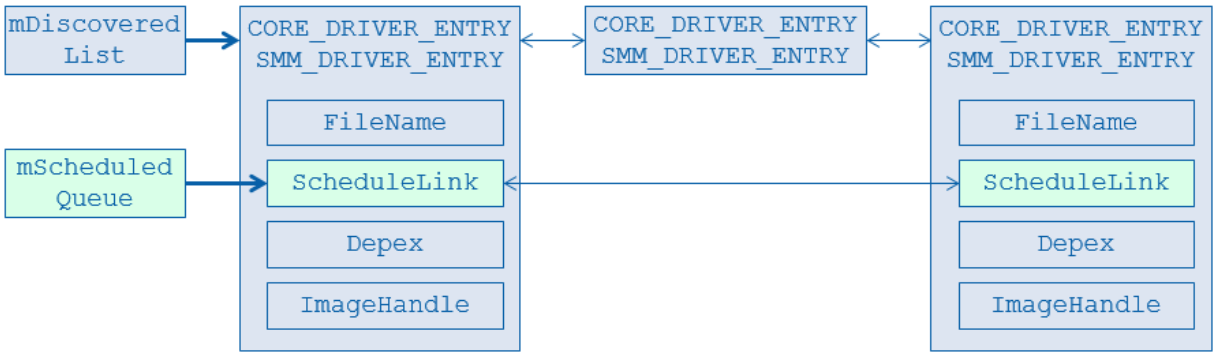


Figure 3-6 Core Image Database

`mDiscoveredList` is a linked list for all discovered images in the firmware volume. `mScheduledQueue` is a subset of `mDiscoveredList`. and `mScheduledQueue` records the linked list of the image whose dependency is satisfied and ready to run.

The pseudo code for current core dispatch is below:

```
============================
Scan FV, put to DiscoveredList.
Check Apriori, put to Scheduled List.
While (TRUE) {
    For image in ScheduledList {
        LoadImage() ←
        call entrypoint // StartImage()
    }
    Check dependency, put to Scheduled List.
}
============================
```

With ASLR capability, the core dispatch logic is updated to below:

```
============================
Scan FV, put to DiscoveredList.
For image in DiscoveredList {
    Copy Information to local cache
}
Shuffle image order in local cache
For image in local cache {
    LoadImage() ←
}
Check Apriori, put to Scheduled List.
While (TRUE) {
    For image in ScheduledList {
        call entrypoint // StartImage()
    }
    Check dependency, put to Scheduled List.
}
```

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

The RED part is the additional step to implement the image shuffle. The `LoadImage()` is moved earlier.

The image shuffle capability is controlled by `gEfiAslrPkgTokenSpaceGuid.PcdImageShuffleEnable` (https://github.com/jyao1/SecurityEx/blob/master/AslrPkg/AslrPkg.dec). When this PCD is TRUE, the DxeCore dispatcher function `CoreFwVolEventProtocolNotify()` (https://github.com/jyao1/SecurityEx/blob/master/AslrPkg/Override/MdeModulePkg/Core/Dxe/Dispatcher/Dispatcher.c) calls `DxeCoreLoadImages()` to load all images with shuffled order before the dependency section is evaluated, as we discussed above.

Image shuffle just controls *image load*, it does not control *image start*. The image start process is unchanged. DxeCore only starts an image after its dependency is satisfied.

Enable ASLR for SMM in EDK II

System Management Mode (SMM) is a resource constrained environment.

- SmmCore randomization.

  The SmmCore is loaded by SmmIpl, and SmmIpl need find all SMRAM and allocate the top of SMRAM for SmmCore. `ExecuteSmmCoreFromSmram()` (https://github.com/jyao1/SecurityEx/blob/master/AslrPkg/Override/MdeModulePkg/Core/PiSmmCore/PiSmmIpl.c) allocates the SMRAM for SmmCore, plus the maximum pages needed by randomization. Then it shifts the SmmCore inside of the whole allocated memory. This is designed to meet the requirement 1 – to make sure the SMRAM consumption is consistent.

- SMM heap randomization.

  The SMM heap randomization is handled in https://github.com/jyao1/SecurityEx/blob/master/AslrPkg/Override/MdeModulePkg/Core/PiSmmCore/Page.c. When a new SMRAM block is found, the `SmmAddMemoryRegion()` function reserves the some fixed length pages and shift the valid SMRAM inside of whole SMRAM.

  The same design philosophy can be adopted by any randomization in SMM, such as stack, page table, GDT, IDT, etc. The key is to allocate MAX fixed length pages, and shift content inside of it, in order to ensure that the SMRAM consumption is consistent in

every boot.

Figure 3-7 shows the SMM memory layout.

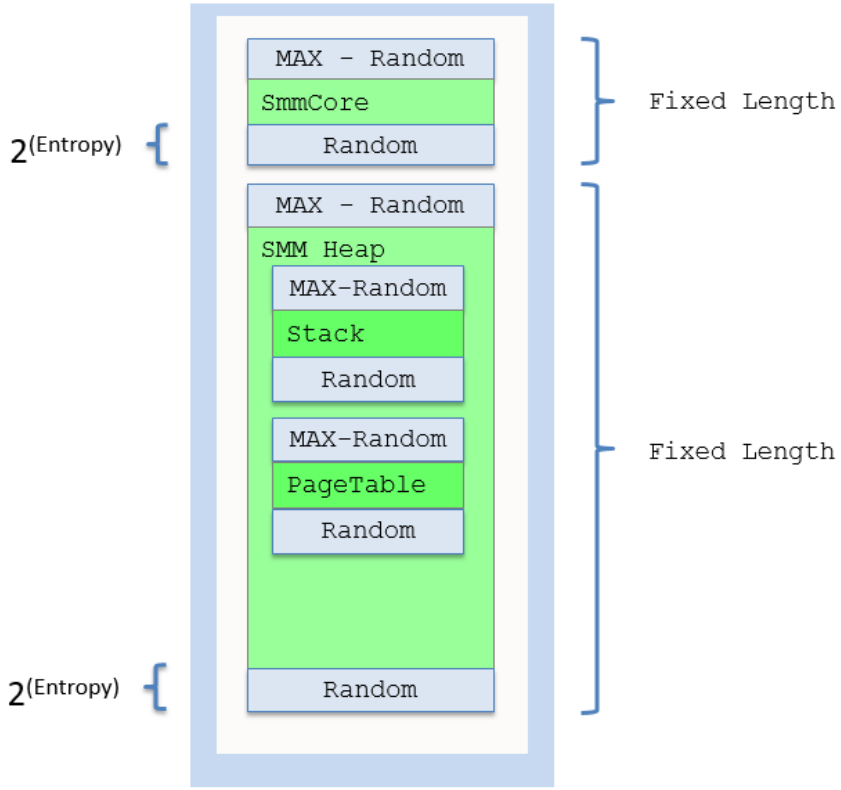


Figure 3-7 SMM memory layout

- SMM image randomization.
  SMM image randomization is similar to UEFI image randomization. When `PcdImageShuffleEnable` is TRUE, the SmmCore dispatcher function `SmmDriverDispatchHandler()` (https://github.com/jyao1/SecurityEx/blob/master/AslrPkg/Override/MdeModulePkg/Core/PiSmmCore/Dispatcher.c) calls `SmmCoreLoadImages()` to load all images with shuffled order before the dependency section is evaluated as we discussed above.

  Just as for UEFI, the SMM image shuffle only controls *image load*. It does not control *image start*. The image start process is unchanged. SmmCore only starts an image after its dependency is satisfied.

- SMM information leak prevention.
  SMM is considered as an isolated and secure execution environment. We randomize the component in SMM to prevent attacks. However, if the randomized information is exposed, it is considered as an information leak.
  In the current EDK II, the SmmCore installs an EFI_LOADED_IMAGE_PROTOCOL into DXE protocol database for each SMM images. This EFI_LOADED_IMAGE_PROTOCOL

contains the SMM image base and size information. This is a typical SMM information leak and make SMM image randomization useless.

In order to mitigate this, `SmmLoadImage()` (https://github.com/jyao1/SecurityEx/blob/master/AslrPkg/Override/MdeModulePkg/Core/PiSmmCore/Dispatcher.c) installs the EFI_LOADED_IMAGE_PROTOCOL into SMM protocol database to make SMM information self-contained.

Future work

Entropy bit selection is one important task for randomization. Insufficient entropy may not be able to resist an attack. Too strong entropy may easily exhaust the memory. We might need more work to see what the best entropy bit is for the different execution environments.

Current AslrPkg just selects some important components for randomization as a sample. We might need to evaluate if we need to randomize more components, such as the core protocol handle database, system table, etc.

Summary

This section introduces the address space layout randomization technique and how to enable different randomization features in EDK II.

Additional Overflow Detection

Besides the mechanism discussed above, we may use other mechanisms to detect buffer overflow.

Stack Overflow Detection

The UEFI specification defined 128 KiB stack size as the minimal requirement. There is no explicit requirement for SMM. A PCD `gUefiCpuPkgTokenSpaceGuid.PcdCpuSmmStackSize` (https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/UefiCpuPkg.dec) defines the SMM stack size for each processor. The default size is 8KiB, and we observed some platforms set it to be 128KiB.

Since the stack size is not so large, there is risk that stack overflows and overlaps with the data in heap below stack. We need to devise an effective mechanism to detect if the stack is healthy in order to assist the developer in debugging potential issues. To that end we use the stack guard page. See figure 4-1.

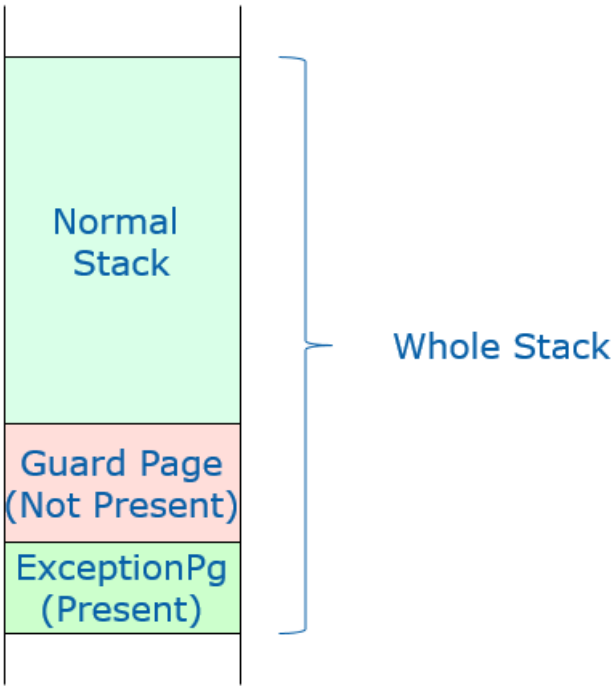


Figure 4-1 StackGuard for Overflow Detection

The core uses 2 pages in the bottom of stack. The second-to-last page is the “GuardPage”, which works as a guard. The GuardPage is set be NOT PRESENT in the page table. When the stack overlaps with the GuardPage, an exception will be triggered.
An interesting thing is that if the current stack is NOT PRESENT, the CPU cannot push the error code and architecture status (CS/RIP/RFLAGS/SS/RSP) to the current stack. The core must

setup a special stack for this exception, which is the “ExceptionPage”.

The last page is the “ExceptionPage”. This page is used in by the exception handler. It guarantees the stack is always valid when an exception happens.

In IA32 protected mode, the core sets up an exception TSS and puts the exception TSS segment in the page fault exception entry. This is done so that when the exception happens, the CPU does a task switch to the new stack. In X64 long mode, the core just reuses the TSS and sets up the IST bit in the page fault exception entry to indicate a stack switch.

In SMM, this stack guard feature is already done in https://github.com/tianocore/edk2/tree/master/UefiCpuPkg/PiSmmCpuDxeSmm and controlled by `gUefiCpuPkgTokenSpaceGuid.PcdCpuSmmStackGuard` in https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/UefiCpuPkg.dec

In UEFI, https://github.com/jyao1/SecurityEx/tree/master/OverflowDetectionPkg/StackGuard enables stack guard feature. It is controlled by `gEfiOverflowDetectionPkgTokenSpaceGuid.PcdCpuStackGuard` in https://github.com/jyao1/SecurityEx/blob/master/OverflowDetectionPkg/OverflowDetectionPkg.dec . If the `PcdCpuStackGuard` is TRUE, this module can be loaded after any CPU DXE driver. `InitStackGuard()` calls `EnableStackGuard()` to clear the PRESENT bit in the page table and then extends the GDT data with the TSS segment and updates the IDT with an exception stack.

https://github.com/jyao1/SecurityEx/tree/master/OverflowDetectionPkg/Test/StackOverflow is the unit test driver. It just pushes data on the stack until the stack reaches the guard page, thus triggering an exception.

Heap Management in EDK II

In UEFI, the DxeCore maintains the heap usage. The UEFI driver or application may call `AllocatePages/FreePages/AllocatePool/FreePool` to allocate or free the resource, or call `GetMemoryMap()` to review all of the memory usage.

[Heap Initialization]

When DxeIpl transfers control to the DxeCore, all of the resource information is reported in a Hand-off-Block (HOB) [PI] list. The DxeCore constructs the heap based upon the HOB information. See figure 4-2 Heap Initialization.

1) The DxeCore needs to find one region to serve as the initial memory in **`CoreInitializeMemoryServices()`** (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Gcd/Gcd.c). The function is responsible for priming the memory map so that memory allocations and resource allocations can be made. If the memory region described by the PHIT HOB is

big enough to hold BIN and minimum initial memory, this memory region is used as highest priority. It can make the memory BIN allocation to be at the same memory region with PHIT that has better compatibility to avoid memory fragmentation. Usually the BIN size is already considered by platform PEIM when the platform PEIM calls `InstallPeiMemory()` to PEI core.

2) Then the DxeCore allocates runtime memory for EFI system table and runtime services table. It triggers BIN initialization in `CoreAddMemoryDescriptor()` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/Page.c), when mMemoryTypeInformationInitialized is FALSE. The memory type information HOB is consumed to pre-allocate memory region for each memory type defined in this HOB. [MemMap] described the purpose and usage of the BIN.
3) Other small DXE service allocation also happed in this region, before full memory is ready. For example, `CoreInitializeImageServices()` installs EFI_LOADED_IMAGE_PROTOCOL for DxeCore, so that we can have ImageHandle for DxeCore, which will be used for GCD.
4) Now DxeCore **`CoreInitializeGcdServices()`** (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Gcd/Gcd.c) need figure out all memory and IO resources, add them to `mGcdMemorySpaceMap` and `mGcdIoSpaceMap`. The tested memory is marked as **`EfiGcdMemoryTypeSystemMemory`** or **`EfiGcdMemoryTypeMoreReliable.`** The other memory is marked to be **`EfiGcdMemoryTypeReserved`** or **`EfiGcdMemoryTypePersistentMemory`**. After the GCD memory map is constructed, the DxeCore calls `CoreAddMemoryDescriptor()` to add all **`EfiGcdMemoryTypeSystemMemory`** and **`EfiGcdMemoryTypeMoreReliable`**. Now all available memory is ready for use.
5) The last step in `CoreInitializeGcdServices()` is to relocate the HOB List to an allocated pool buffer. The relocation should be at after all the tested memory resources are added because the memory resource found in `CoreInitializeMemoryServices()` may have not enough remaining resource for the HOB List.

Now the DxeCore heap initialization is done. The rest of DxeCore and any drivers may use the UEFI services AllocatePages/AllocatePool to allocate a chunk of memory. The DxeCore uses the below priority in `FindFreePages()` to find a free memory location.

- If the memory type matches the one described in the memory type information, the memory in BIN is used as first priority.
- If the memory type is not in memory type information, or there is no enough memory in the pre-allocated BIN, the DxeCore looks for the free memory with top-down algorithm.
- If there is not enough memory, the DxeCore does a special **"memory promotion"**. `PromoteMemoryResource()` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/Page.c) is called to add UNTESTED memory region to be system memory. Then `FindFreePages()` tries to find some free memory again.

In the late DXE/BDS phase, there might be a memory test driver, such as https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/MemoryTest, to test all untested memory and add them to the memory map. After this

point all the memory is added to UEFI memory map.

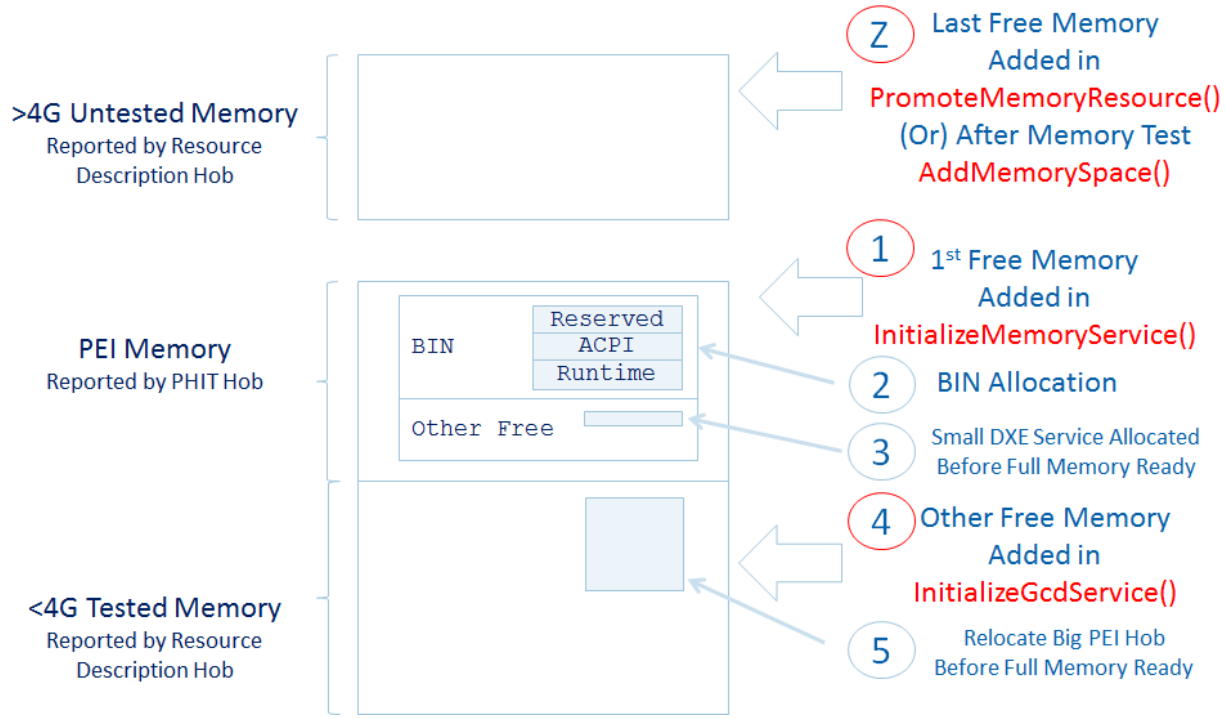


Figure 4-2 Heap Initialization

[Page Management]

After the heap is initialized, the DxeCore maintains a list of memory map entries. See figure 4-3 Page management. This is a linked list with the memory addresses in ascending order. It contains memory address, length, type and attribute at the page level. When the UEFI service `GetMemoryMap()` is called, the contents of this linked list are returned, together with other MMIO with `EFI_MEMORY_RUNTIME` attribute and the reserved memory in the GCD resource list.

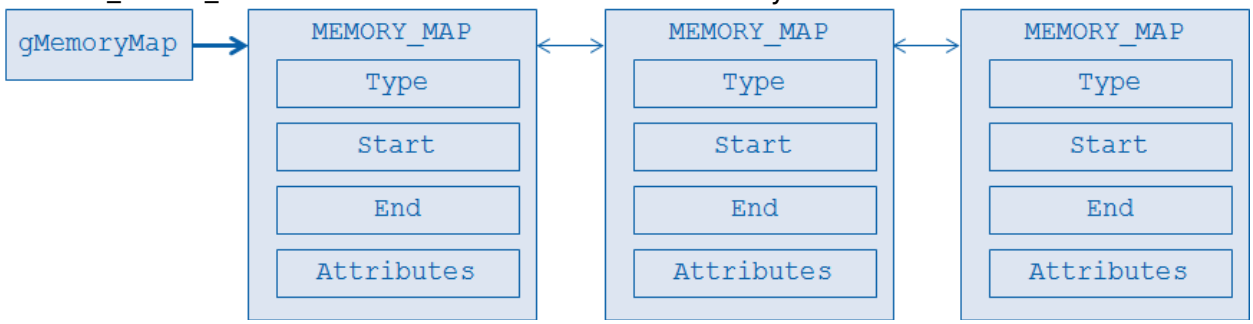
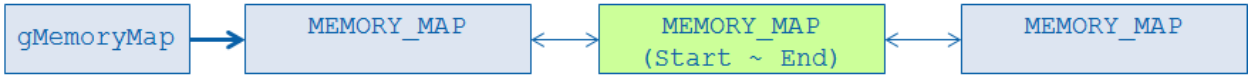


Figure 4-3 Page Management

When `gBS->AllocatePages()` is called, `CoreInternalAllocatePages()` calls `FindFreePages()` to find out which address can be allocated. The `gMemoryMap` linked list is traversed in `CoreFindFreePagesI()` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/Page.c). If the Type of an entry is `EfiConventionalMemory`, and the Length field of the entry is larger than the requested length, and the Start field of the entry is the highest, then the target

allocated address is found in this entry. At this point `CoreInternalAllocatePages()` calls `CoreConvertPages()` to update this memory map linked list entry.
The original entry which contains the target allocated address will be separated (or CLIP operation will occur) into 2 or more entries because the memory type is now different.



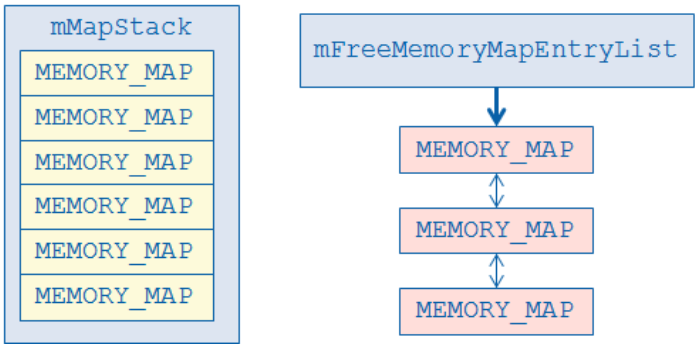


Figure 4-4.1 Page Management – Clip: Step 1

Besides the `gMemoryMap` linked list, there are 2 data structures involved in the CLIP. `mMapStack` contains 6 `MEMORY_MAP` entries as a global variable. `mFreeMemoryMapEntryList` maintains a list of unused `MEMORY_MAP` entries. The details of the "**CLIP**" process is shown at figure 4-4.1 ~ 4-4.5.
Step 1: The `CoreConvertPagesEx()` discovers which memory map entry (the GREEN one) contain the target allocated address.

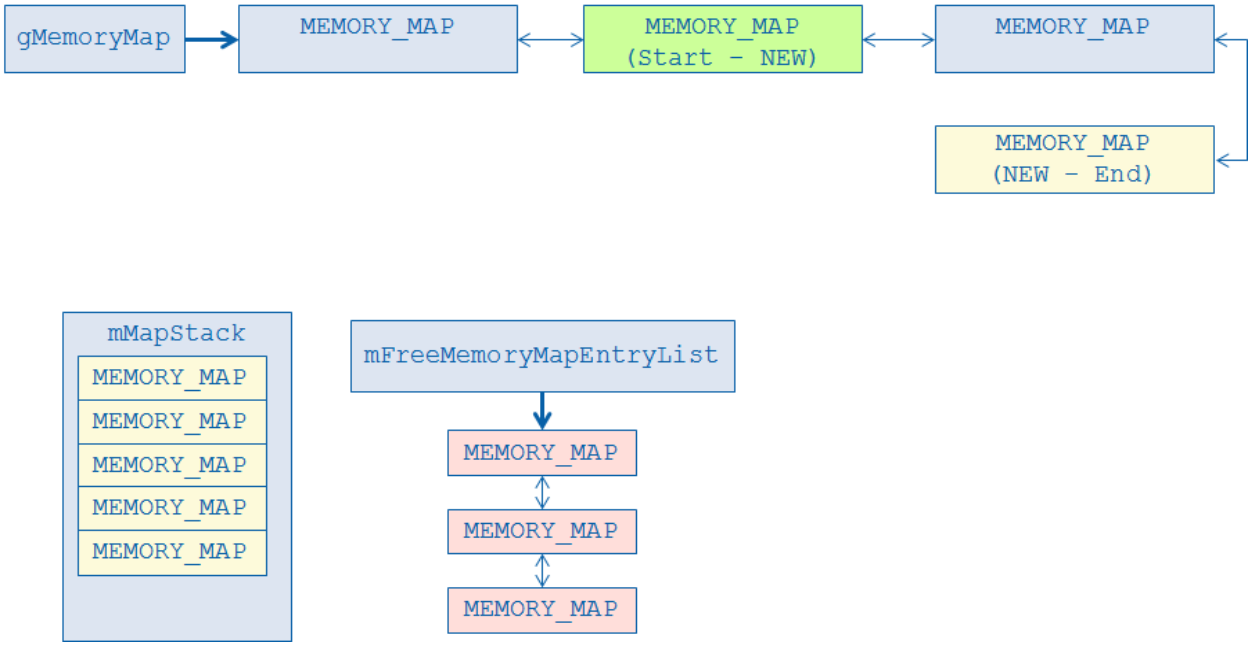


Figure 4-4.2 Page Management – Clip: Step 2

Step 2: The content of this target `MEMORY_MAP` entry is updated to contain the part of reset memory after allocation. If the allocated memory is in the middle of the entry, one entry (the YELLOW one) in `mMapStack` is popped and added to the memory map linked list. This entry covers the rest of memory. This new `MEMORY_MAP` entry must be from `mMapStack` because we are not able to *allocate* memory again in the *allocate* process. We cannot get an entry from `mFreeMemoryMap`, because this linked list might be empty at that time.

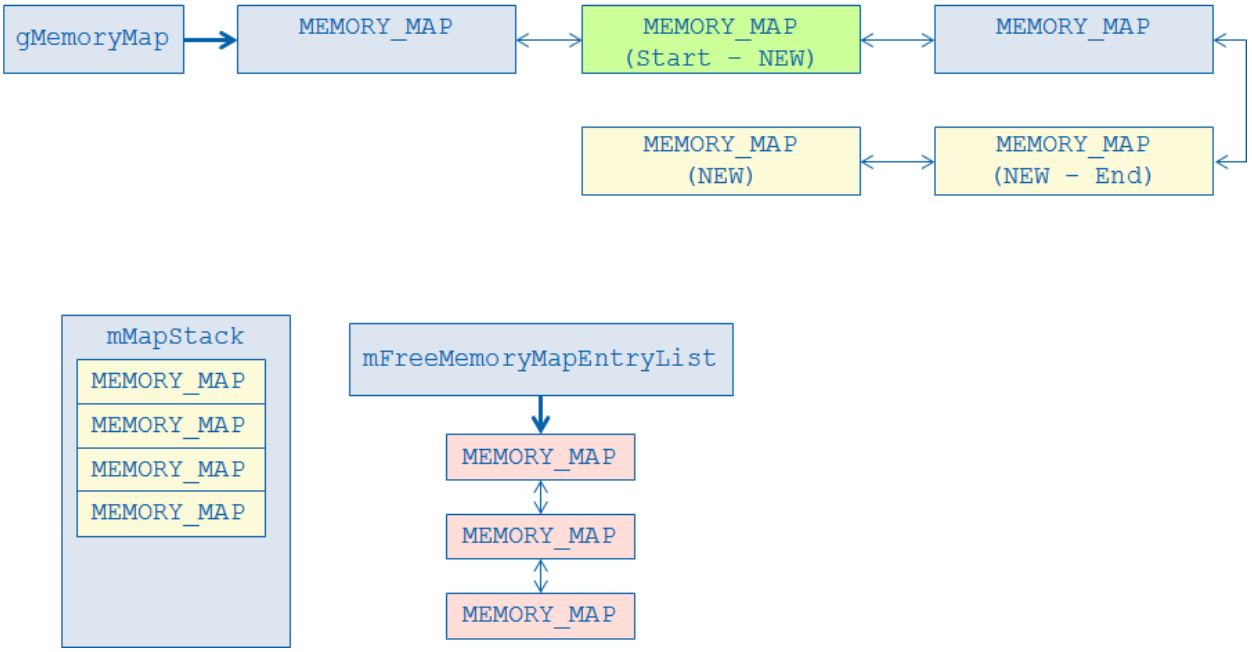


Figure 4-4.3 Page Management – Clip: Step 3

Step 3: `CoreConvertPagesEx()` calls `CoreAddRange()` to add the new `MEMORY_MAP` entry to the memory map linked list. This new entry contains the allocated memory information. For the same reason listed earlier, this entry is also from `mMapStack`.

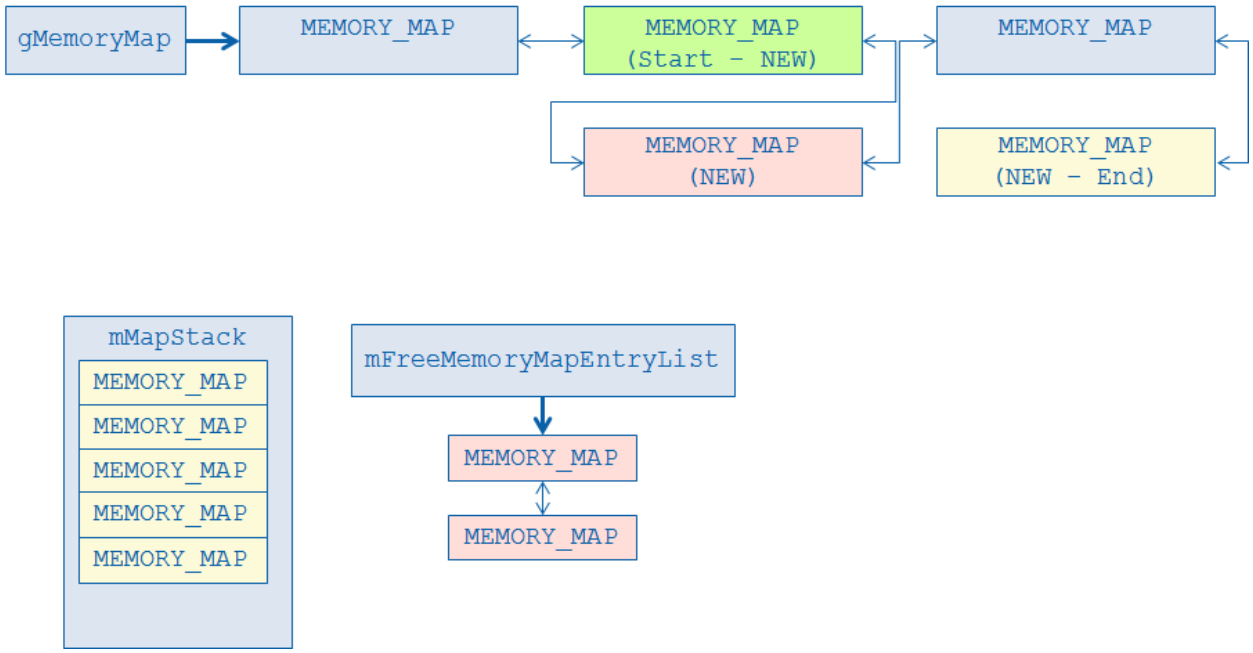


Figure 4-4.4 Page Management – Clip: Step 4

Step 4: Last but not of least importance, `CoreConvertPagesEx()` calls `CoreFreeMemoryMapStack()` to move the `MEMORY_MAP` entry from `mMapStack` to pool. `CoreFreeMemoryMapStack()` calls `AllocateMemoryMapEntry()` to de-queue a `MEMORY_MAP` entry (the RED one) from `mFreeMemoryMapEntryList`. It copies the content to the new entry and finds the correct insertion location with ascending order. The memory allocate may happen in `AllocateMemoryMapEntry()` if the `mFreeMemoryMapEntryList` is empty.

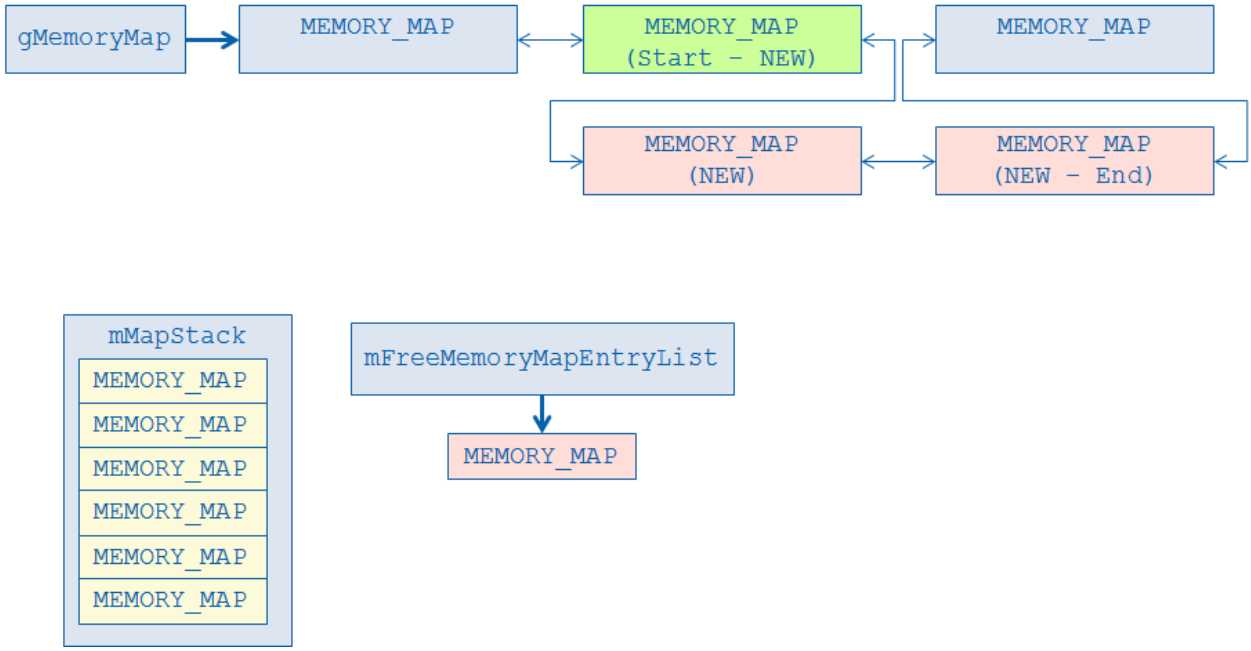


Figure 4-4.5 Page Management – Clip: Step 5

Step 5: Finally, after all entries from `mMapStack` are moved to pool, the memory map linked list CLIP is finished. The `mMapStack` is kept unchanged. The `mFreeMemoryMapEntry` is updated.

Later, when `FreePages()` happens, then 2 or more `MEMORY_MAP` entries can be merged into one. The unused `MEMORY_MAP` entries are returned to `mFreeMemoryMapEntryList`. Those entries can be used in a subsequent `AllocatePages()`.

[Pool Management]
Besides page management, DxeCore maintains the pool. A typical UEFI driver or application calls `gBS->AllocatePool()` for memory allocation.

DxeCore assigns one `mPoolHead` for each UEFI specification defined memory type (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/Pool.c). Each `mPoolHead` includes a set of `FreeList`. Each `FreeList` is a linked list for the fixed size free pool. The size of each `FreeList` is defined in `mPoolSizeTable`. It is a Fibonacci sequence, which allows us to migrate blocks between bins by splitting them up, while not wasting too much memory. See figure 4-5 Pool Management.

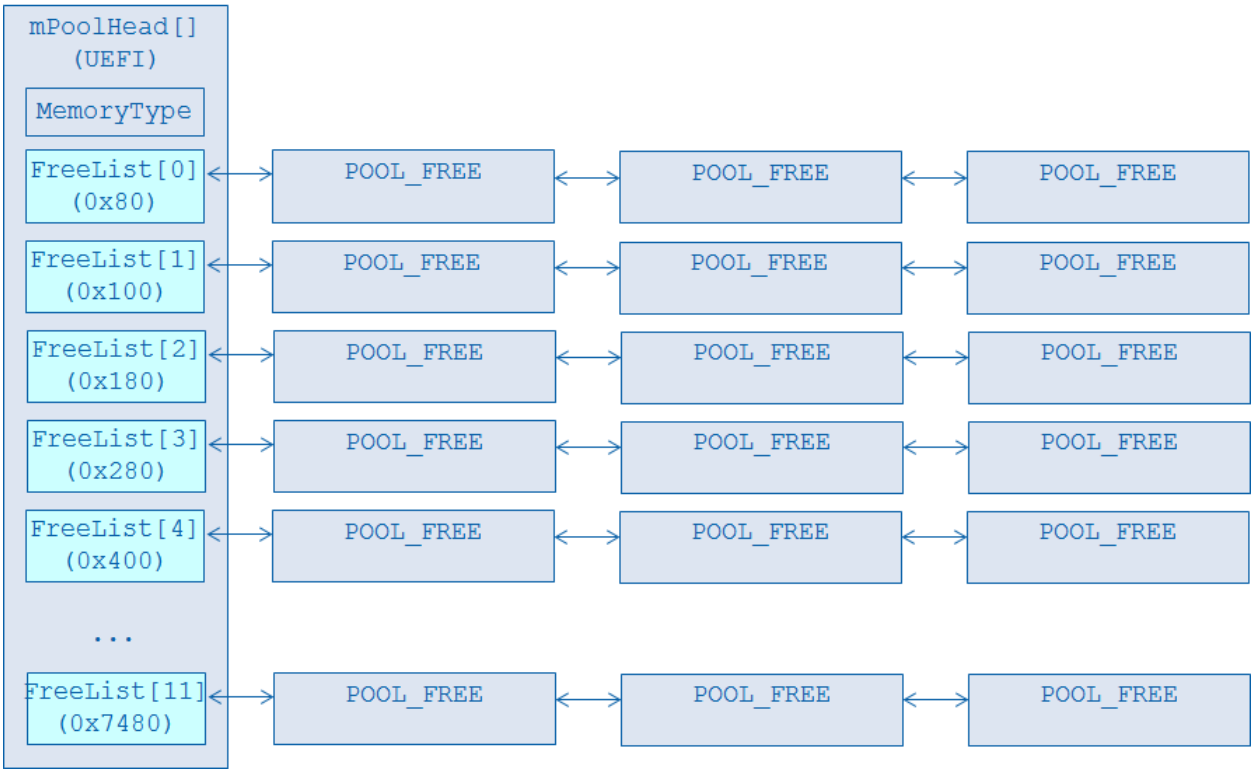


Figure 4-5 Pool Management

When `gBS->AllocatePool()` is called, `CoreInternalAllocatePool()` calls `CoreAllocatePoolI ()` to find out which `FreeList` should be used. If the requested pool size is too big to fit in all `FreeList`, a PoolPage is allocated and returned directly. The `FreeList` is untouched.
If the requested pool size matches one of the `FreeList`, one entry in this `FreeList` is dequeued and returned as the free memory.
If the matched `FreeList` is empty, `CoreAllocatePoolI()` checks the bins holding larger blocks, and will **CARVE** one. The detail "**CARVE**" process is shown at figure 4-6.1 ~ 4-6.2.

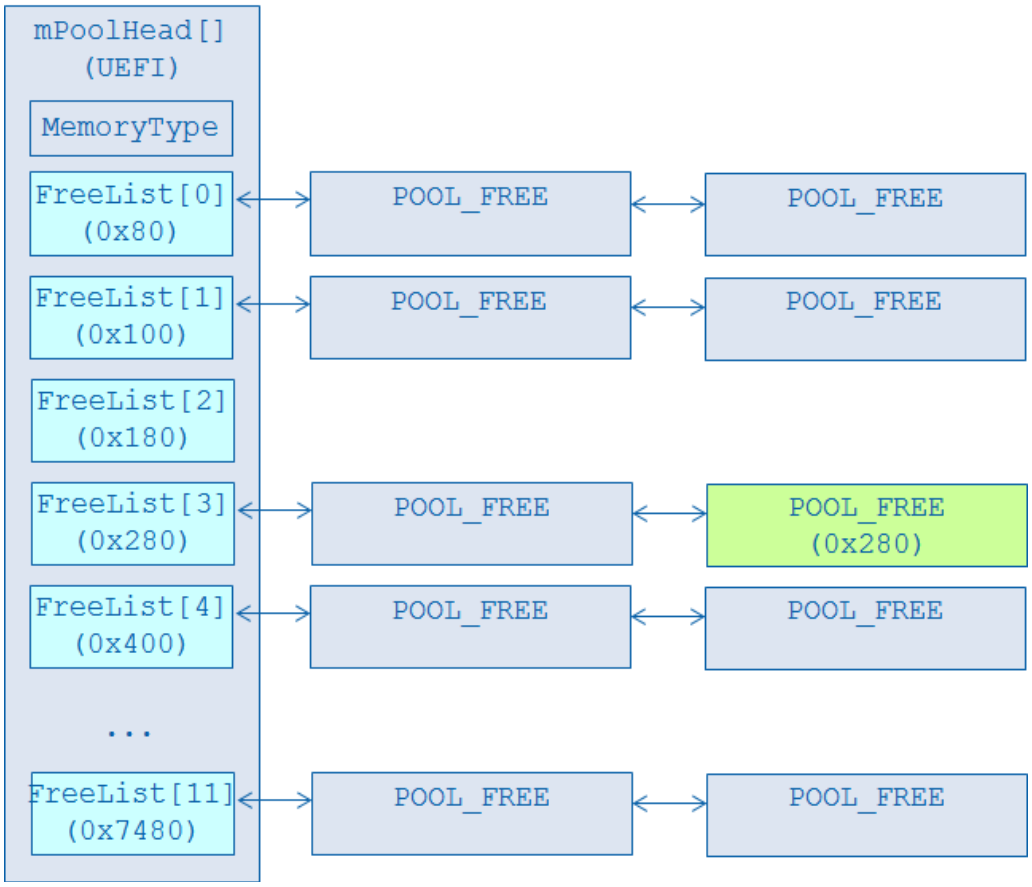


Figure 4-6.1 Pool Management – Carve: Step 1

Step 1: If the matched `FreeList` is empty (such as FreeList[2]), `CoreAllocatePoolI()` increments the Index of `FreeList` one by one and checks if the next `FreeList` is empty. If there is one `FreeList` that is not empty (such as FreeList[3]), one `POOL_FREE` entry in this linked list is dequeued (the GREEN one) and goes to the CARVE process.
If there is no available entry in all `FreeList`, `CoreAllocatePoolI()` calls `CoreAllocatePoolPages()` to allocate new pages as the candidate and goes to the CARVE process.

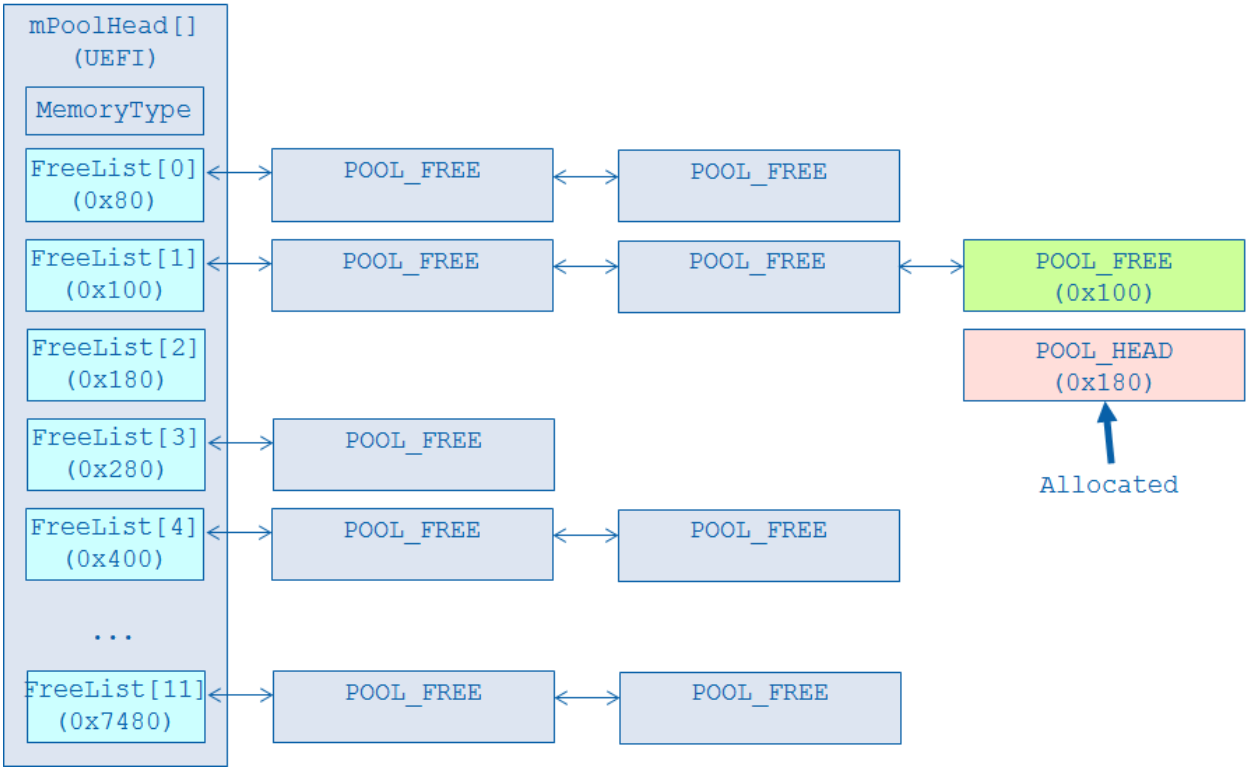


Figure 4-6.2 Pool Management – Carve: Step 2

Step 2: Since the size of this `POOL_FREE` candidate entry is bigger than the one requested, `CoreAllocatePoolI()` records the requested entry (the RED one) and splits up the remaining space (the GREEN one) into free pool blocks (such as FreeList[1]). The head of the request entry is changed from `POOL_FREE` to `POOL_HEAD` to record the pool information, such as size and type.

Later, when `FreePool()` is called, the information in `POOL_HEAD` can be used to discover into which `FreeList` it should be returned. If the pool size is too big to fit in all `FreeList`, it is a PoolPage and freed by `CoreFreePoolPages()`. Alternately, this `POOL_HEAD` is converted to `POOL_FREE` and is inserted into one of the proper `FreeList`. `CoreFreePoolI()` also makes an additional check to see if all the pool entries in the same page as Free are freed pool entries. If so, all of these pool entries are removed from the free loop lists, and `CoreFreePoolPages()` is called to free the entire pages.

Heap Overflow Detection (for Page)

Heap overflow is a big problem. [WindowsHeap] discussed some machenism to detect heap overflow.

In UEFI, we may setup a guard page around the allocated pages. The concept is similar to the guard page for stack.

Whenever there is an `AllocatePage()` request, the core allocates 2 more pages. One page is before the allocated pages and the other is after. Both are set be NOT PRESENT in the page table. If the overflow happens, the page fault exception is triggered immediately.
See figure 4-7.

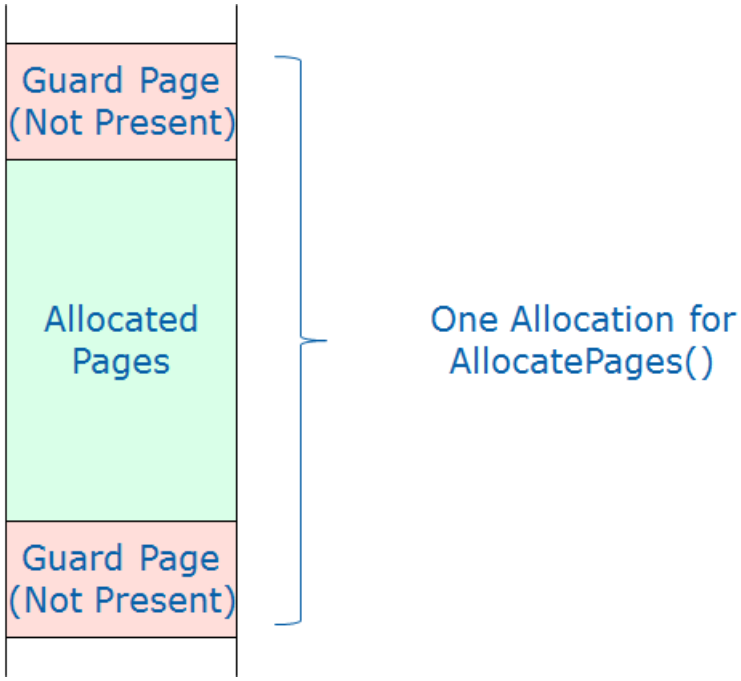


Figure 4-7 HeapGuard for Page Overflow Detection

Such enhancement is added to https://github.com/jyao1/SecurityEx/blob/master/OverflowDetectionPkg/Override/MdeModulePkg/Core/Dxe/Mem/Page.c. The HeapGuard feature is controlled by `gEfiOverflowDetectionPkgTokenSpaceGuid.PcdHeapPageGuard` in https://github.com/jyao1/SecurityEx/blob/master/OverflowDetectionPkg/OverflowDetectionPkg.dec . If the `PcdHeapPageGuard` is TRUE, `CoreInternalAllocatePages()` allocates 2 more pages and call `SetGuardPageOnAllocatePages()`. The later calls `SetGuardPage()` twice to set the guard page before and after. `SetGuardPage()` constructs a linked list (`GUARD_HEAD` and `GUARD_TAIL` with a special signature) in the guard page, before it is set to be not present. See figure 4-8 HeapGuard structure.
This is designed to detect heap overflow in non-paging environment, such as IA32 with paging disabled, or NT32 emulation. `mGuardPageList` is the linked list header. `CheckGuardPages()` can be used to detect heap overflow, even with paging disabled.

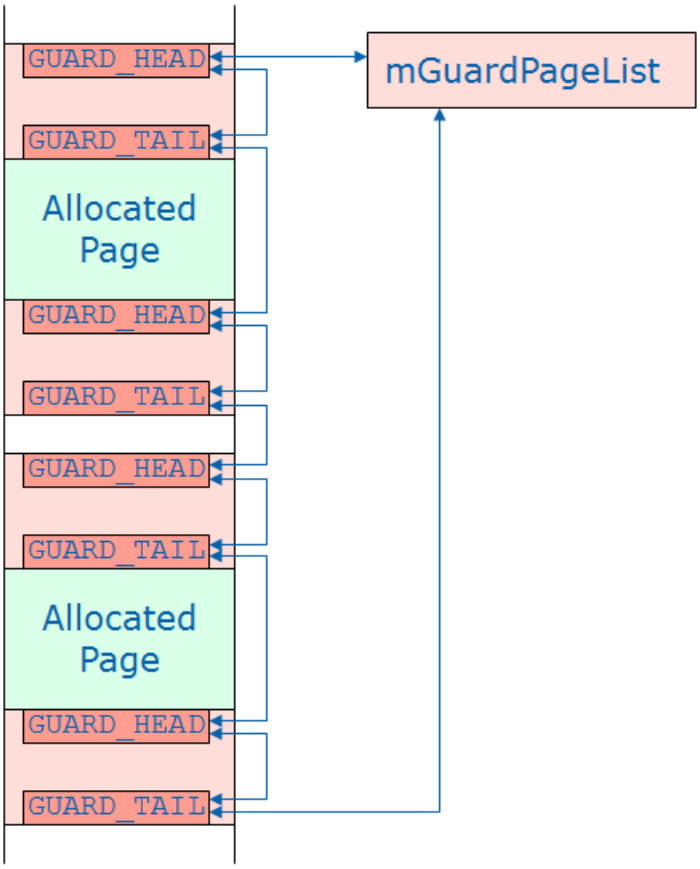


Figure 4-8 HeapGuard structure.

After `CoreInternalAllocatePages()` sets up the GuardPage and returns, `CoreAllocatePages()` calls `SetMemoryPageAttributes()` (https://github.com/jyao1/SecurityEx/blob/master/OverflowDetectionPkg/Include/Library/PageTableLib.h) to clear PRESENT flag. This must be done here to avoid a re-entry issue. This re-entry risk stems from the fact that `SetMemoryPageAttributes()` may need to split a page table entry, which in turn needs to call the `AllocatePages()` API again.

One interesting feature for `AllocatePages()` is that the pages can be freed partially. Figure 4-9 shows a possible AllocatePages/FreePages sequence. After the caller frees the middle of the pages, the core needs to insert the GuardPages to protect the rest of the allocated pages. This work is done by `ClearGuardPageOnFreePages()`. It returns the guard operations such as `SetGuard/ClearGuard/NoAction`, so that `CoreFreePages()` can perform the GuardOperation later.

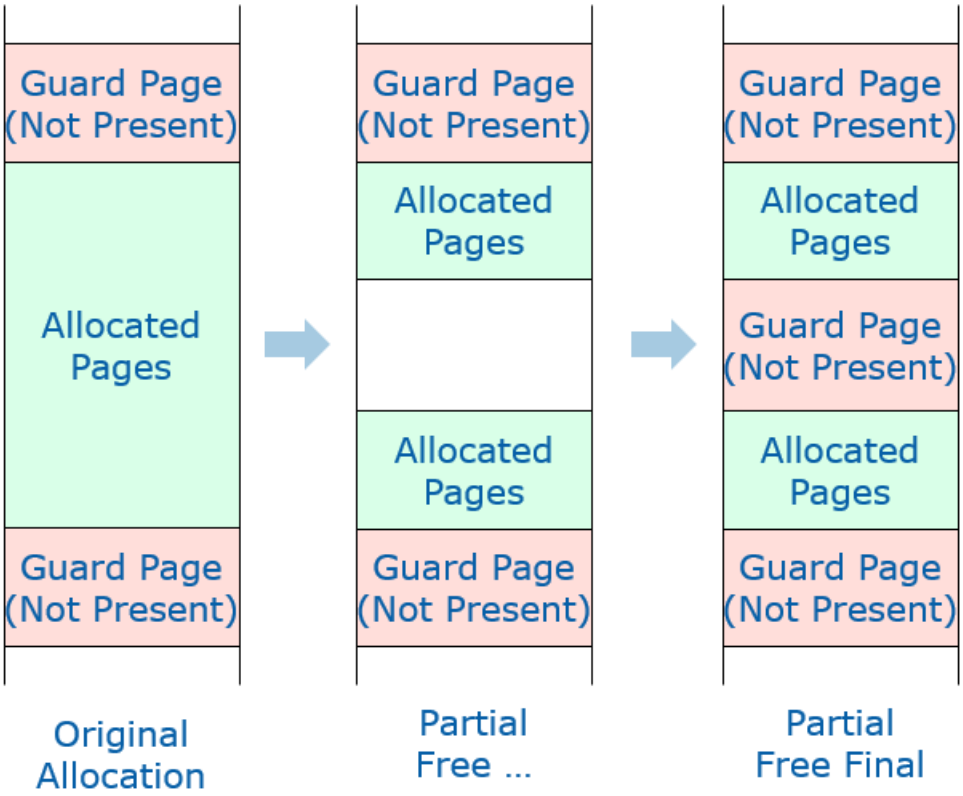


Figure 4-9 HeapGuard in Partial Page Free

The unit test https://github.com/jyao1/SecurityEx/tree/master/OverflowDetectionPkg/Test/HeapOverflow can be used to trigger heap overflow and the page fault exception is triggered eventually.

Heap Overflow Detection (for Pool)

We can use GuardPage for pages. What about pool?

If the requested pool is over max size, then `CoreAllocatePoolI()` (https://github.com/jyao1/SecurityEx/blob/master/OverflowDetectionPkg/Override/MdeModulePkg/Core/Dxe/Mem/Pool.c) just allocate pages for this request – `CoreAllocatePoolPages()`.

In this case, the same guard page mechanism is used to protect such PoolPages. See figure 4-10.

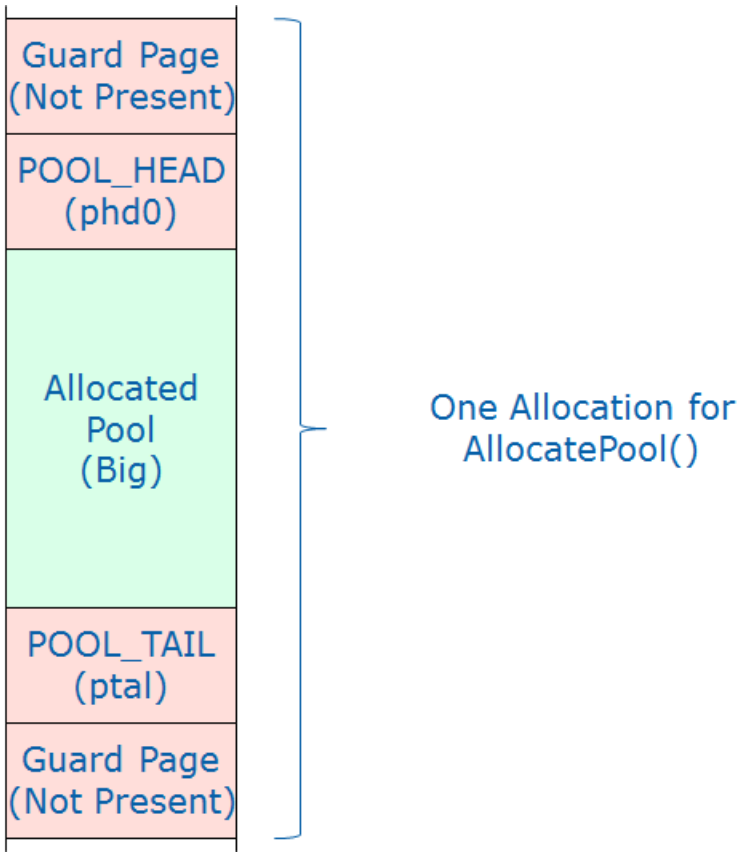


Figure 4-10 Big Pool Overflow Detection

However, if the pool is small and there is a free entry for the small pool, the DxeCore always adds `POOL_HEAD` and `POOL_TAIL` with a special signature around the allocated pool buffer. See figure 4-11. If there is any pool overflow, then the signature is changed, and this change will be caught at `CoreFreePool()`.

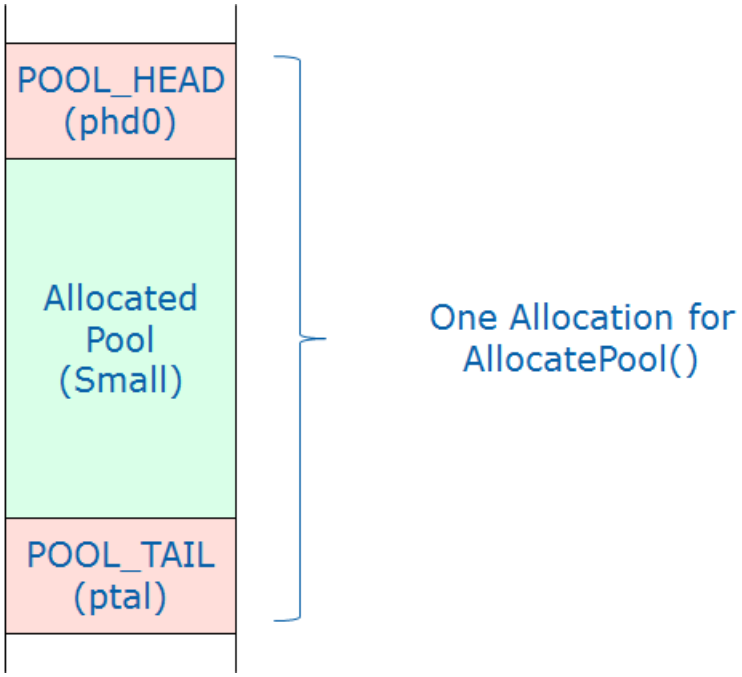


Figure 4-11 Small Pool Overflow Detection

NULL Pointer Protection in EDK II

Zero address is considered as an invalid address in most programs. However, in x86 systems, the zero address is valid address in legacy BIOS because the 16bit interrupt vector table (IVT) is at address zero. In current UEFI firmware, zero address is always mapped.
We can do some enhancement here. Once the 16bit legacy support is dropped in UEFI firmware, it is possible to mark the first 4K page at address zero to be invalid for X86 system. Then, we can catch the zero address reference if a program does not check memory allocation successful or not.

Since CSM/legacy boot needs to be disabled for OS compliance when using UEFI Secure Boot, few systems are seen in the market requiring this memory to be mapped at zero.

We define a `gEfiOverflowDetectionPkgTokenSpaceGuid.PcdNullPointerProtection` (https://github.com/jyao1/SecurityEx/blob/master/OverflowDetectionPkg/OverflowDetectionPkg.dec). If `PcdNullPointerProtection` is TRUE, https://github.com/jyao1/SecurityEx/tree/master/OverflowDetectionPkg/NullPointerProtection will call `SetMemoryPageAttributes()` (https://github.com/jyao1/SecurityEx/blob/master/OverflowDetectionPkg/Include/Library/PageTableLib.h) to update the page table to remove PRESENT bit of the address zero page. As such, a Page Fault exception will be generated if some program access to address zero.

Future work

Both software and hardware-based advances can be added to strengthen code against this class of issue. On the software front, language-based security may offer some relief. This can span using a safer variant of C [CHECKED__C] to refactoring code to a type-safe language [RUST]. These are huge tasks, though, given the existing software catalog and would challenge software portability. As such, a language-based approach is not a near term option.

As we go from software to hardware, though, one option appears feasible. Specifically, recent hardware advances include the Intel® Memory Protection Extensions (Intel® MPX). This is a new capability introduced into Intel Architecture [IA32SDM][MPX]. Intel MPX can help detect the buffer overflow or underflow with a set of new MPX instructions and the compiler support. When MPX is enabled, a Bounds Table is constructed to store the pointer value, lower bound of the buffer, and the upper bound of the buffer. See figure 4-12.

The BNDMK instruction can create LowerBound (LB) and UpperBound (UB) in bounds register. The BNDCL/BNDCU/BNDCN instruction can check the address of a memory reference or address against the LB or UB. A BOUND Range Exceeded exception (#BR) is raised if any of the bounds compare instructions fail.

Intel MPX need compiler support. MSVC 2015 Update 1 and GCC 5.1 supports Intel MPX.

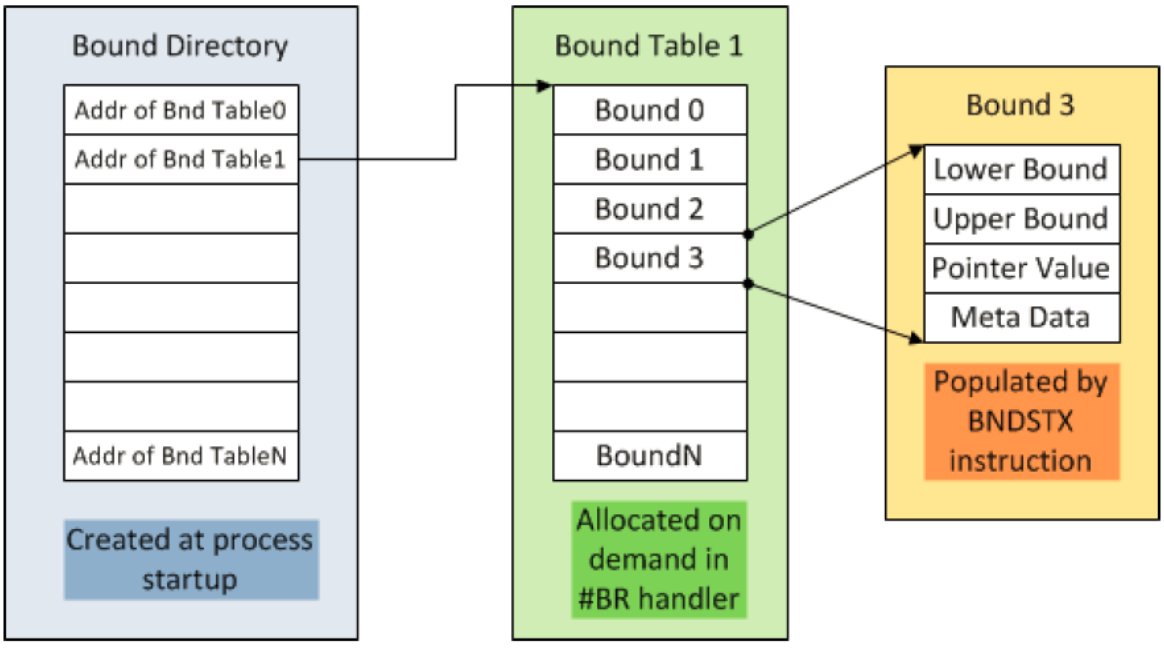


Figure 4-12 Bound Table, (source: [MPX])

Intel MPX may also be considered to add to a UEFI firmware to help catch the heap pool overflow or the global variable overflow.

Summary

This section discussed some other mechanism which can be used to detect stack overflow or heap overflow in EDK II.

References

General

[Tanenbaum] *Modern Operating Systems, 4th edition,* Andrew S. Tanenbaum, Herbert Bos, Pearson, 2014, ISBN: 978-0133591620

[Veen] *Memory Errors: the Past, the Present, and the Future*, Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos, 2012, Research in Attacks, Intrusions, and Defenses, Volume 7462 of the series Lecture Notes in Computer Science pp 86-106. Springer, ISBN 978-3-642-33337-8

Stack Canaries

[StackCanaries] http://en.wikipedia.org/wiki/Buffer_overflow_protection

[StackCheck] *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.* Cowan, C., Pu, C., Maier, D., Hintongif, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q. Proceedings of the 7th USENIX Security Symposium (January 1998), https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

[MSVC] *Compiler Security Checks In Depth*, https://msdn.microsoft.com/library/aa290051.aspx

[MSVC_GS] */GS (Buffer Security Check)*, https://msdn.microsoft.com/en-us/library/8dbf701c.aspx

[MSVC_RTC] */RTC (Run-Time Error Checks)*, https://msdn.microsoft.com/en-US/library/8wtf2dfz.aspx

[MSVC_FASTFAIL] *_fastfail*, https://msdn.microsoft.com/en-us/library/dn774154.aspx

[GCC] *Proposal to add a new stack-smashing-attack protection mechanism "-fstack-protector-strong"*, https://docs.google.com/document/d/1xXBH6rRZue4f296vGt9YQcuLVQHeE516stHwt8M9xyU/edit

[PI] UEFI Platform Initialization Specification, Version 1.5 http://www.uefi.org/sites/default/files/resources/PI%201.5.zip

Data Execution Protection

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

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

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

[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

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

[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

Address Space Randomization

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

[ASLR1] *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

[ASLR2] *Enhance Memory Protections in IE10*, http://blogs.msdn.com/b/ie/archive/2012/03/12/enhanced-memory-protections-in-ie10.aspx

[OpenBSD] *Exploit Mitigation Techniques*, Theo de Raadt, http://www.openbsd.org/papers/ven05-deraadt

[PIE] *OpenBSD's Position Independent Executable (PIE) Implementation*, Kurt Miller, http://www.openbsd.org/papers/nycbsdcon08-pie/

[PaX] *PaX presentation*, Brad Spengler, https://grsecurity.net/PaX-presentation.ppt

Additional Overflow Detection

[CHECKED_C] Checked C https://www.microsoft.com/en-us/research/project/checked-c/

[RUST] Rust language https://www.rust-lang.org/en-US/

[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/

[WindowsHeap2] *Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass*, Alexander Anisimov, 2004, https://www.ptsecurity.com/ww-en/download/defeating-xpsp2-heap-protection.pdf

[WindowsHeap3] *Attacking the Vista Heap*, Ben Hawkes, 2008, http://www.blackhat.com/presentations/bh-usa-08/Hawkes/BH_US_08_Hawkes_Attacking_Vista_Heap.pdf

[WindowsHeap4] *Practical Windows XPSP3/2003 Heap Exploitation*, John McDonald and Christopher Valasek, 2009, http://www.blackhat.com/presentations/bh-usa-09/MCDONALD/BHUSA09-McDonald-WindowsHeap-PAPER.pdf

[MPX] *Intel® Memory Protection Extensions Enabling Guide* https://software.intel.com/en-us/articles/intel-memory-protection-extensions-enabling-guide

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.