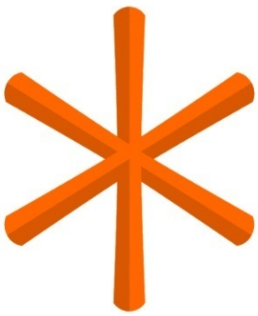


tianocore

A Tour Beyond BIOS -Security Enhancement to Mitigate Buffer Overflow in Unified Extensible Interface (UEFI)

TABLE OF CONTENTS

- A Tour Beyond BIOS - Security Enhancement to Mitigate Buffer Overflow in UEFI
  - Executive Summary
- Stack Canaries
  - Stack Check Support in Microsoft Visual Studio
  - Stack Check Support in GCC
  - Enable Stack Check in EDK II
  - Future work
- Data Execution Protection
  - DEP in X86 Processor
  - DEP in UEFI specification
  - Enable DEP in EDK II
  - Future work
- Address Space Layout Randomization
  - ASLR in Windows
  - ASLR in *nix
  - ASLR requirement in UEFI firmware
  - Enable ASLR for UEFI in EDK II
  - Enable ASLR for SMM in EDK II
  - Future work
- Additional Overflow Detection
  - Stack Overflow Detection
  - Heap Management in EDKII
  - Heap Overflow Detection (for Page)
  - Heap Overflow Detection (for Pool)
  - NULL Pointer Protection in EDK II
  - Read-only page table
  - Limitation
  - Compatibility Consideration
  - Call for action
  - Future work
- Summary
  - Policy Control
- References

A TOUR BEYOND BIOS - SECURITY ENHANCEMENT TO MITIGATE BUFFER OVERFLOW IN UNIFIED EXTENSIBLE INTERFACE (UEFI)

WHITEPAPER

DRAFT FOR REVIEW

04/30/2025 11:44:18

Revision 02.0

Contributed by

Jiewen Yao, Intel Corporation

Vincent J. Zimmer, Intel Corporation

Jian Wang, Intel Corporation

Acknowledgements

Revision History

Revision	Revision History	Date
01.0	Initial release.	Oct 2016
02.0	Added: enabling of a special pool feature, how does it work, how to debug, etc....,	March 2018

	Convert to Gitbook	2018

Introduction A buffer overflow is “one of the most important exploitation techniques in the history of computer security.” [Tanenbaum][1] “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] [1] 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.

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

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)

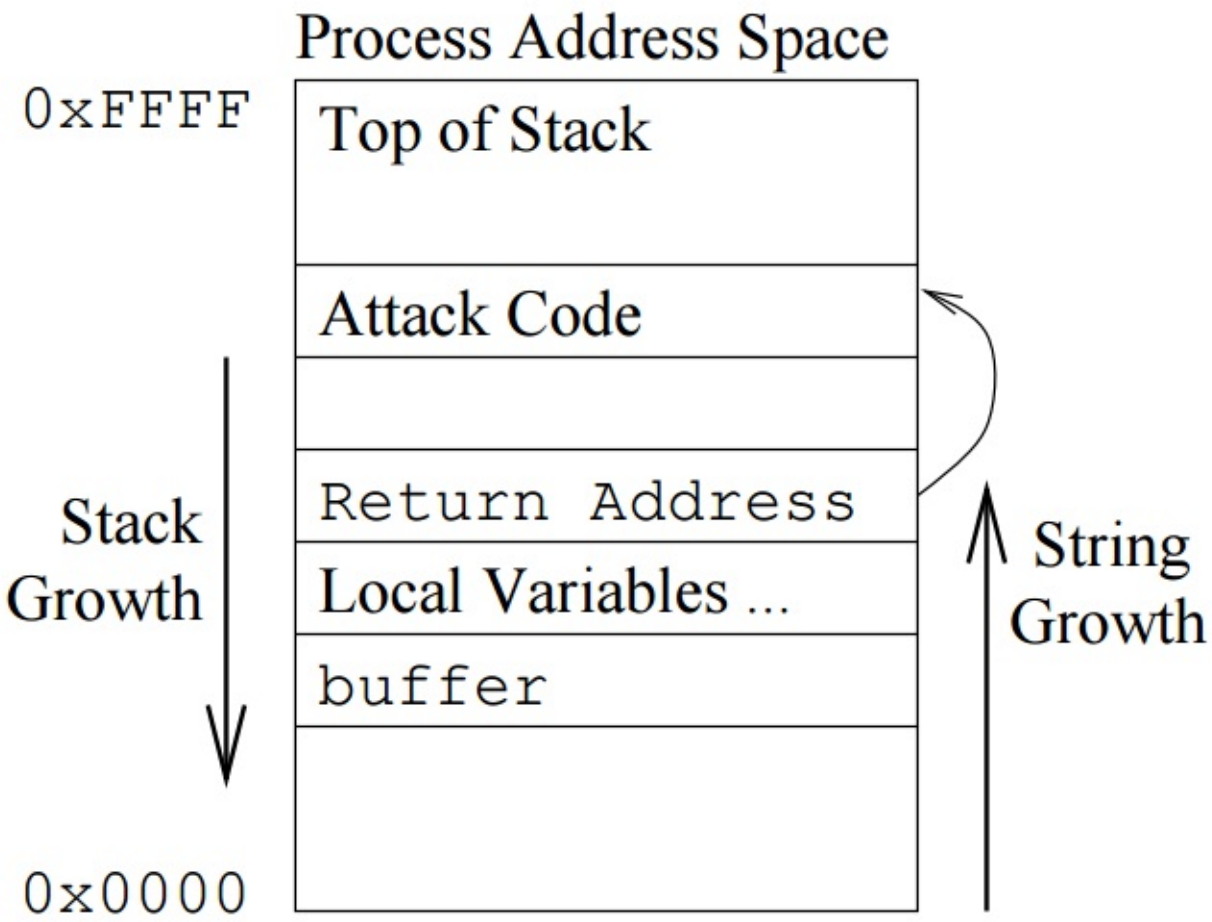


Figure 1-1 Stack Smashing Buffer Overflow Attack (Source: [StackCheck][1])

In 1999, StackCheck was introduced to prevent buffer overflow attack on a stack.[StackCheck][1]>

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)

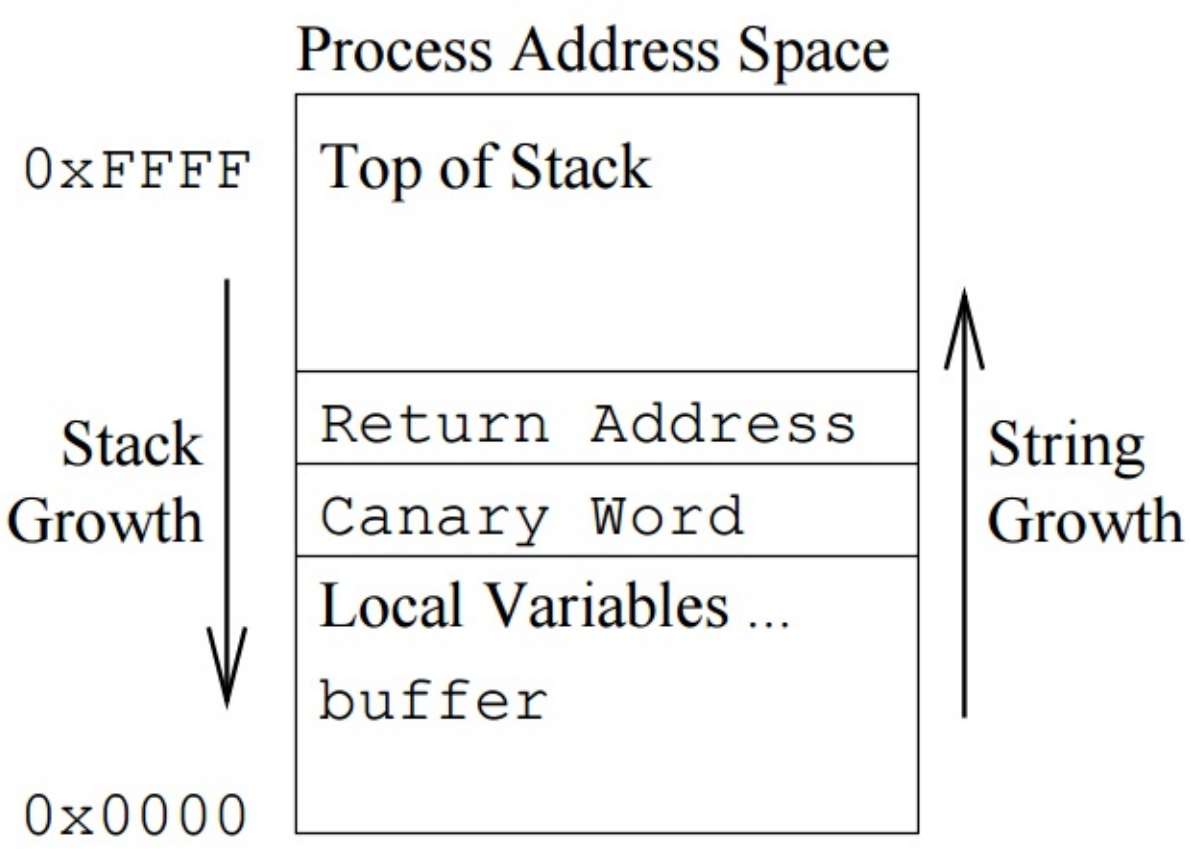


Figure 1-2 Canary Word Next to Return Address (Source: [StackCheck][1])

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

[1] [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)

Stack Check Support in Microsoft Visual Studio*

Microsoft Visual Studio supports a stack guard function. “Compiler Security Checks In Depth” [MSVC] [1] introduces the detail on how it works. There are 2 compiler options related: `/GS` and `/RTC`.

`/GS` [MSVC_GS] [2] 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] [3] 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.

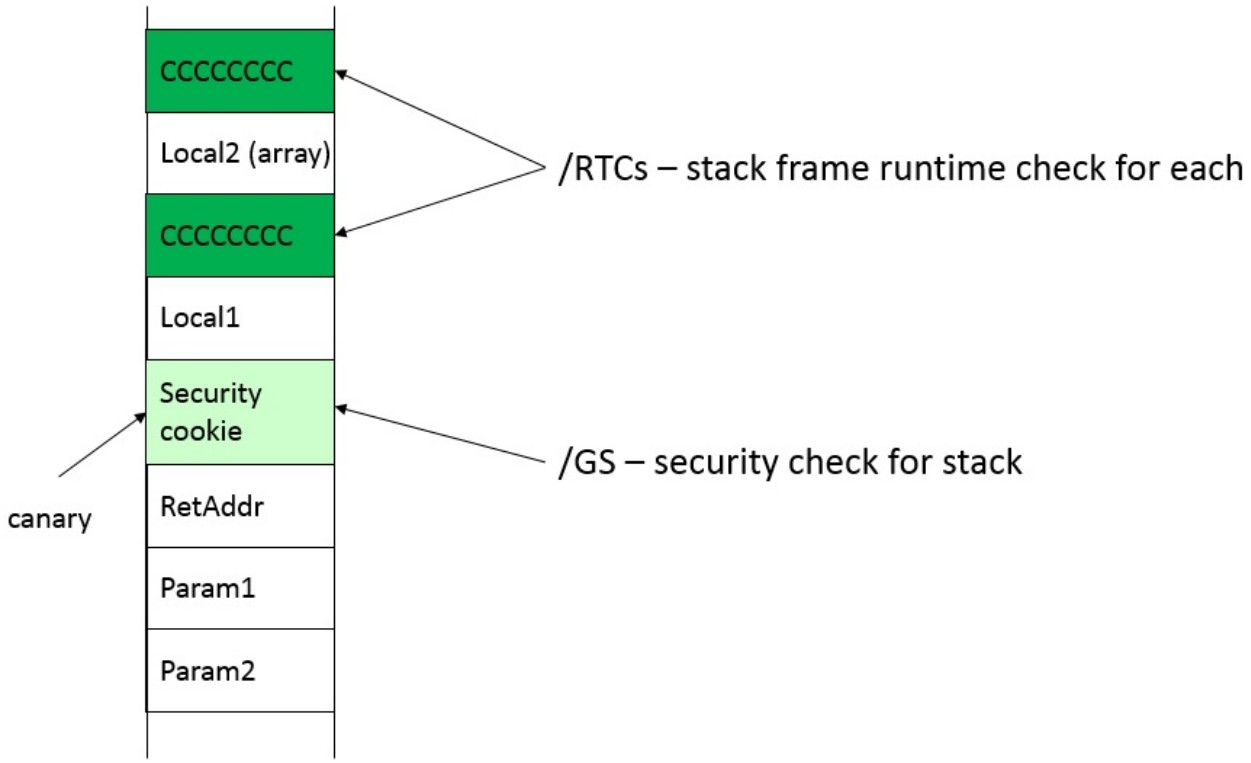


Figure 1-3 – Microsoft Stack Check

[1] [MSVC] Compiler Security Checks In Depth

[2] [MSVC_GS] /GS (Buffer Security Check)

[3] [MSVC_RTC] /RTC (Run-Time Error Checks)

Stack Check Support in GNU Compiler Collection* (GCC)

The first stack guard is supported in GCC. Current GCC supports: `-fstack-protector, -fstack-protector-all, and -fstack-protector-strong` .[StackCanaries] [1] .

`-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] [2]

---

[1] [StackCanaries] Buffer overflow protection - Wikipedia.org

[2] [GCC] Proposal to add a new stack-smashing-attack protection mechanism “-fstack-protector-strong”

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] [1]
If a System Management Mode (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] [2] 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 Baseboard Management Controller (BMC).

Summary

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

---

[1] [MSVC_FASTFAIL] Microsoft.com- _fastfail, [2] [PI] UEFI Platform Initialization Specification, Version 1.5

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] [1] [PaX] [2]. DEP may be applied to: [WindowsInternal][3]

- 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] [1].

---

[1][DEP] Exploit Mitigation Improvements in Windows 8, Ken Johnson, Ma, Miller

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

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

DEP in X86 Processor

Data Execution Protection (DEP) is a hardware feature. Intel X86 processor supports the `XD` (eXecution Disable) bit in the page table. [IA32SDM][1] 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` model specific register (MSR), and then set the `XD` bit in the CPU physical address extensions (PAE) page table.

[1][IA32SDM] Intel® 64 and IA-32 Architectures Software Developer’s Manual,

DEP in UEFI specification

The Unified Extensible Firmware Interface (UEFI) [www.uefi.org] 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 the white paper, A Tour Beyond BIOS Memory Map And Practices in UEFI BIOS .

Enable DEP in EDKII

In the white paper, A Tour Beyond BIOS- Memory Protection in UEFI BIOS, we discussed the how to enable DEP for stack, heap, and PE image in Driver eXecution Environment (DXE) and System Management Mode (SMM) environment.

We support Non-Executable stack, ReadOnly Portable Executable (PE) image code, Non-Executable PE image data.

Future work

As discussed in the white paper, A Tour Beyond BIOS Memory Map And Practices in UEFI BIOS, 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 the white paper A Tour Beyond BIOS Memory Map And Practices in UEFI BIOS, 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 Sace 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.

Address Sace Layout Randomization (ASLR) in Windows*

Windows supports Address Sace Layout Randomization (ASLR). [WindowsInternal] [1] desribes the detail on how executable images, DLL, stack, heap are randomized.

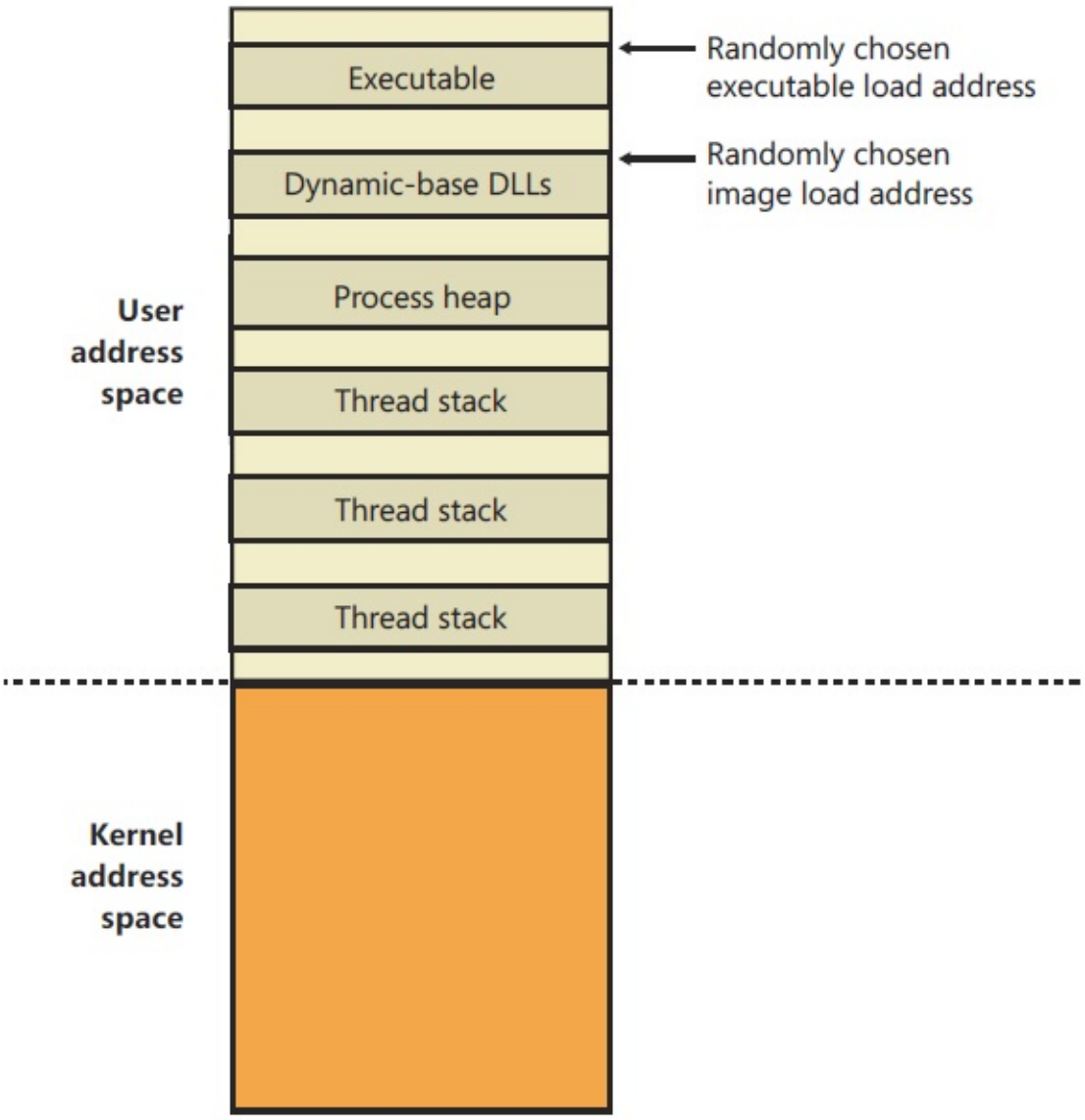


Figure 3-1 Windows OS space layout, source: [WindowsInternal] [1]

[ASLR1][2] 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][2]

[ASLR2][3] shows different image layouts during boot.

The following Diagram showing how the physical memory location of various system DLLs changes between a first and second boot

[3]

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

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

[2][ASLR1] Exploit Mitigation Improvements in Windows 8, Ken Johnson, Ma, Miller

[3][ASLR2] Enhance Memory Protections in IE10

Address Sace Layout Randomization (ASLR) in *nix

[PaX] [1] also provides an ASLR patch for Linux and OpenBSD. [OpenBSD][2] and [PIE][3] provide detailed information on the randomized image layout.

---

[1] [PaX] PaX presentation, Brad Spengler,

[2][OpenBSD] Exploit Mitigation Techniques, Theo de Raadt

[3][PIE] OpenBSD's Position Independent Executable (PIE) Implementation, Kurt Miller

Address Sace Layout Randomization (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 pre-boot 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 System Management Mode (SMM) or the Pre-EFI Initialization (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 Global Descriptor Table (GDT), Interrupt Descriptor Table (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 Address Sace Layout Randomization (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 Driver eXecution Environment (DXE) stack is allocated from heap in the Pre-EFI Initialization (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.

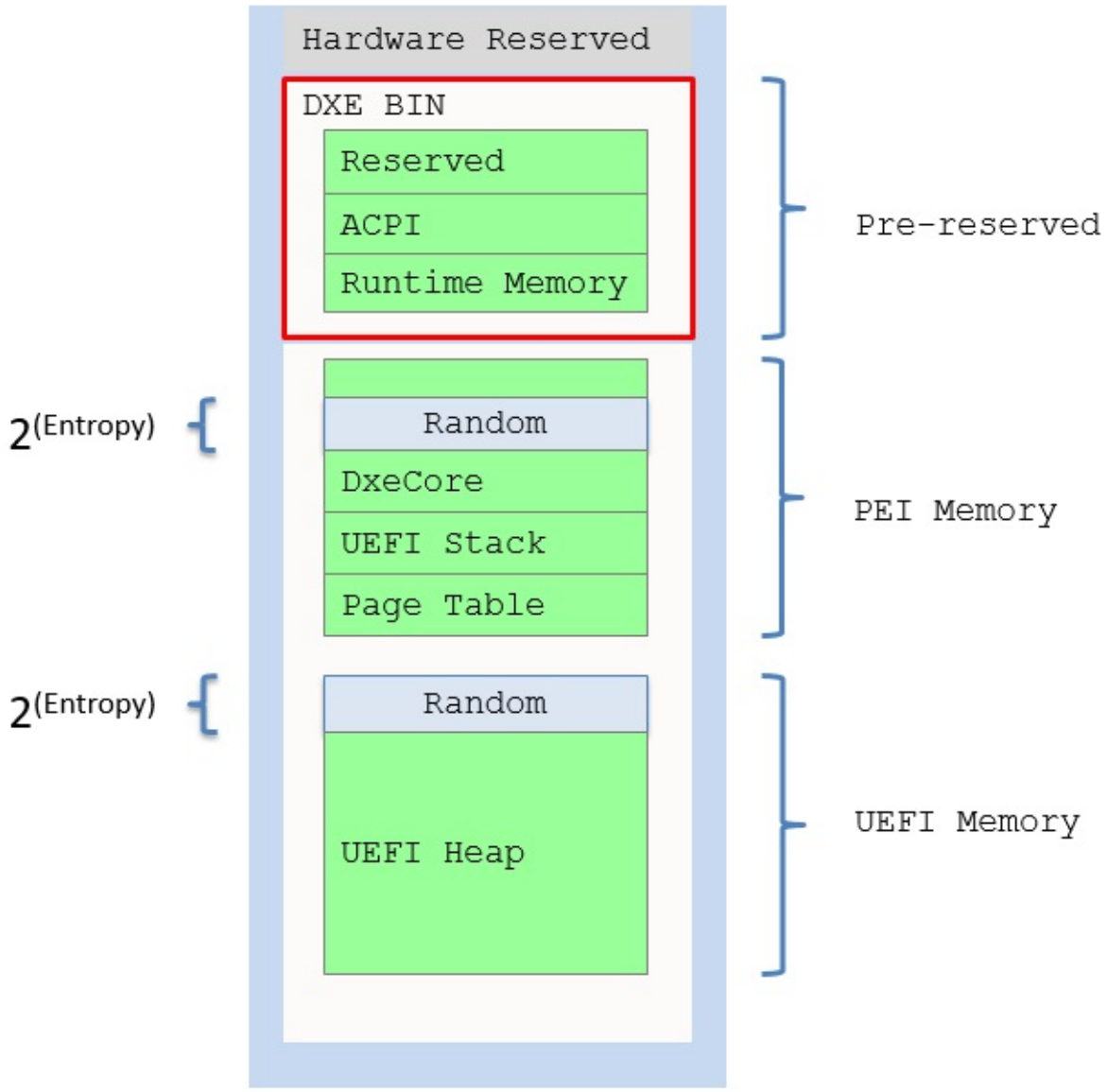


Figure 3-4 UEFI memory layout

- **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 Portable Executable (PE) Common Object File Format(COFF) (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.

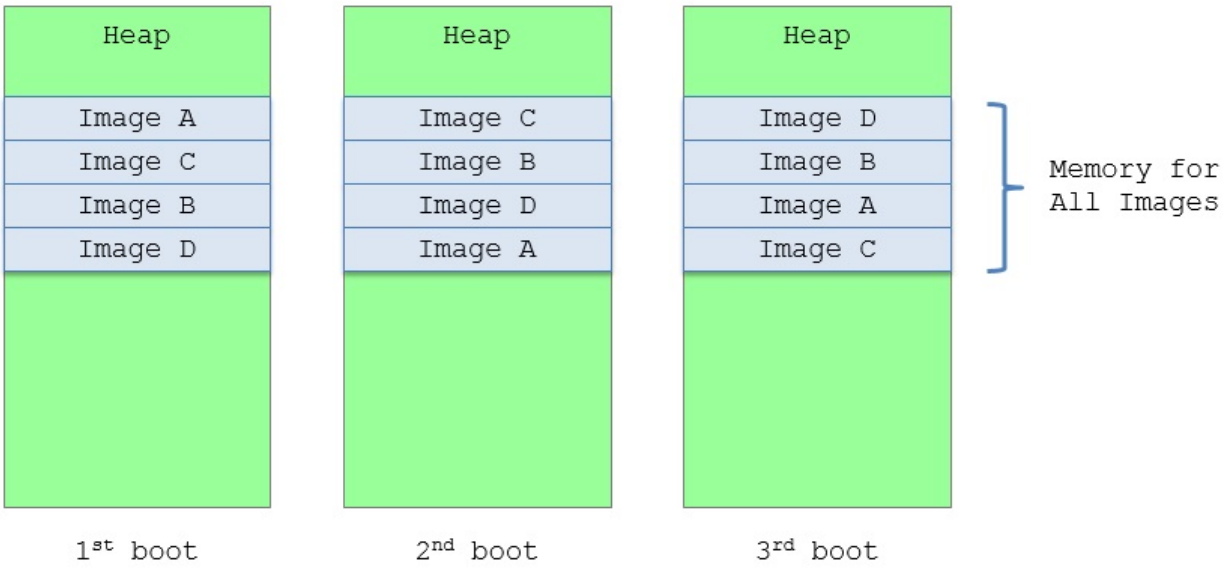


Figure 3-5 Image Shuffle

For example, if a 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:

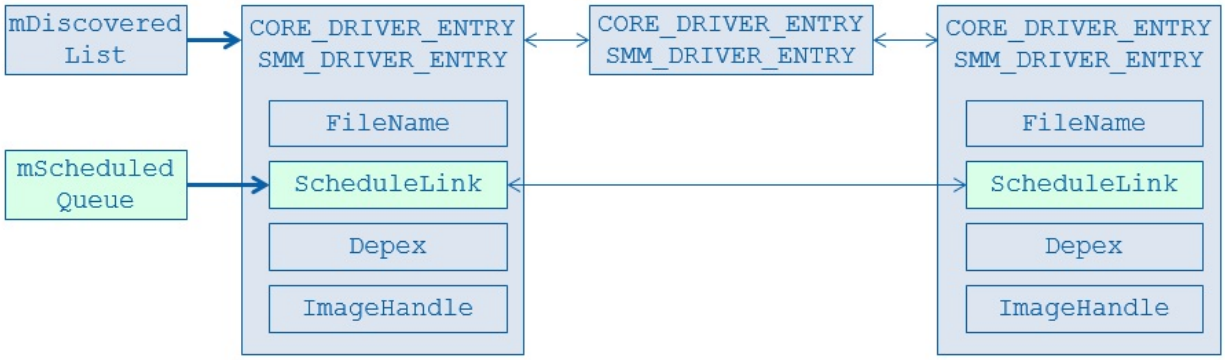


Figure 3-6 Core Image Database

The top left most `mDiscoveredList` is a linked list for all discovered images in the firmware volume. The `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()
}
```

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

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

The image shuffle capability is controlled by the Platform Configuration Database (PCD) Variable: `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 Enable Address Sace Layout Randomization (ASLR) for System Mamangement Mode (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.

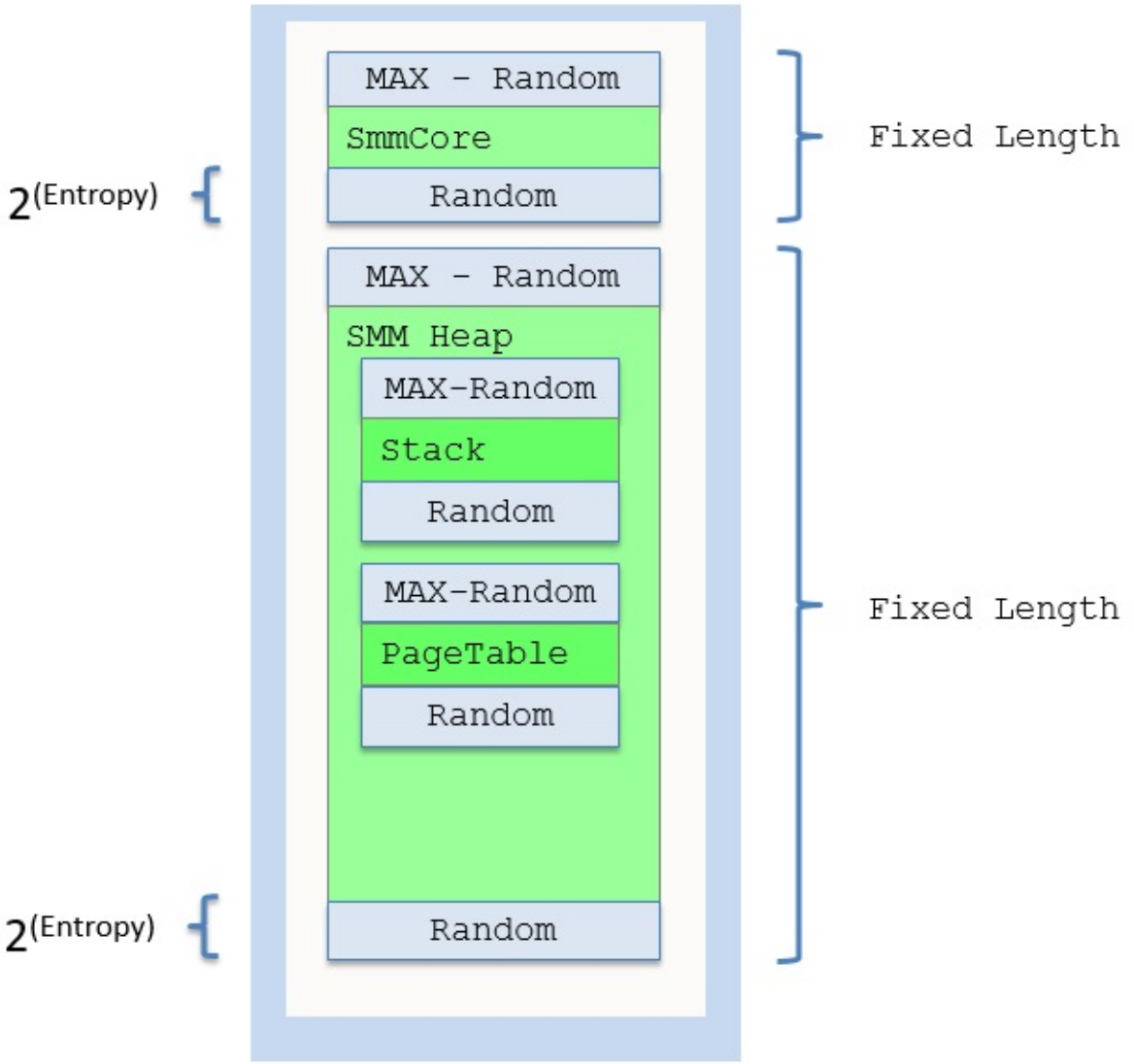


Figure 3-7 SMM memory layout

- **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 System Management Mode (SMM). A Platform Configuration Database (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 marks the last page in the bottom of stack as a “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 exception stack for the exception handler, which is the “ExceptionStack”. This page is reserved separately and only used by the exception handler. It guarantees the stack is always valid when an exception happens. For a multi-processor system, each processor has its own stack, its own guard page and its own exception stack.

In IA32 protected mode, the core sets up an exception task state segment (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 PCD `gUefiCpuPkgTokenSpaceGuid.PcdCpuSmmStackGuard` in https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/UefiCpuPkg.dec

In UEFI, this stack guard feature is done in https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/DxeIplPeim and controlled by PCD `gEfiMdeModulePkgTokenSpaceGuid.PcdCpuStackGuard` in https://github.com/tianocore/edk2/blob/master/MdeModulePkg/MdeModulePkg.dec. If the `PcdCpuStackGuard` is TRUE, the `DxeIpl` clears the PRESENT bit in the page table for the guard page of the BSP stack. The guard page of the AP stack is initialized in `CpuDxe` driver, https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/Library/MpInitLib/DxeMpLib.c by using DXE service `SetMemorySpaceAttributes()`.

Besides CPU driver, the `CpuExceptionHandlerLib` (https://github.com/tianocore/edk2/tree/master/UefiCpuPkg/Library/CpuExceptionHandlerLib) is also updated to support Stack Overflow detection. The new API – `InitializeCpuExceptionHandlersEx()` is introduced to initialize the exception TSS. `InitializeCpuExceptionHandlersEx()` is invoked by `DxeCore` to setup the exception TSS for the Boot Strap Processor (BSP), and `InitializeCpuExceptionHandlersEx()` is invoked by `CpuDxe` driver to setup the exception TSS for the Application Processor (AP).

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][1] 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 Phase Handoff Information Table (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 Pre-EFI Initialization Module (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 an Image Handle for `DxeCore`, which will be used for Global Coherency Domain (GCD).
4. Now `DxeCore` **`CoreInitializeGcdServices()`** (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Gcd/Gcd.c) needs 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 l
ooks for the free memory with top-down algorithm.

*   If there is not enough memory, the DxeCore does a special **“memory promotion”**. `PromoteMemoryResource()`([https://githu
```

```
b.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/Page.c](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](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.
```

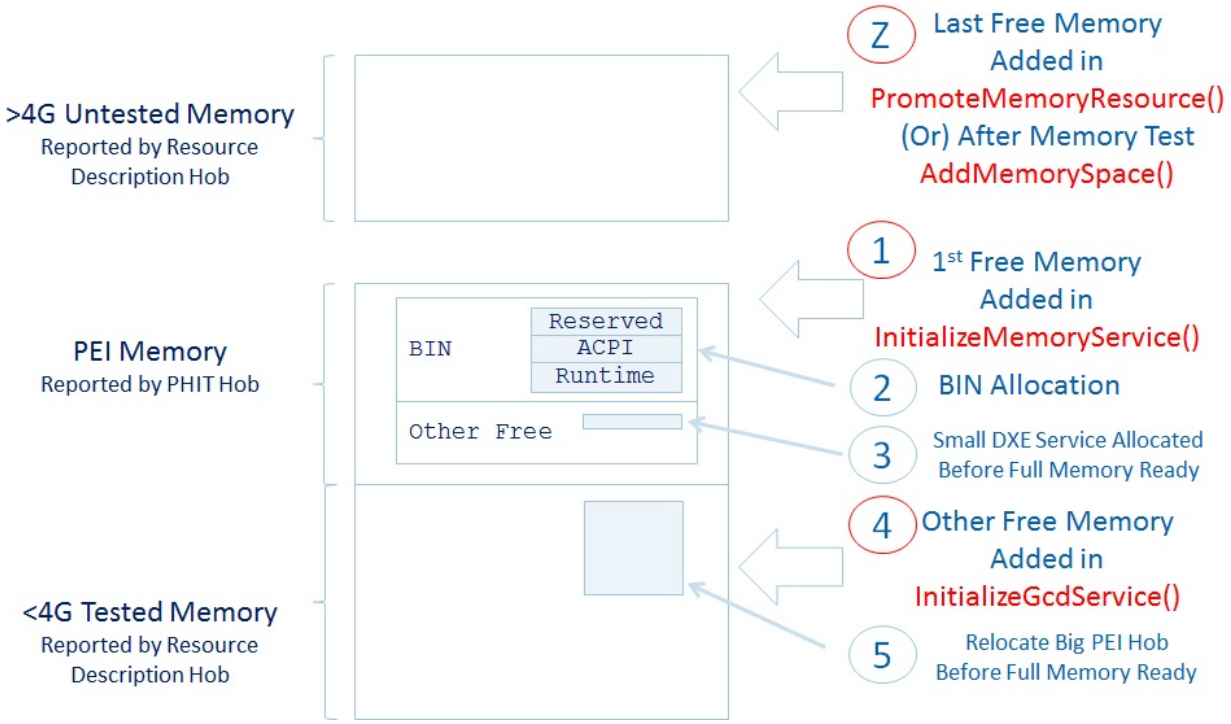


Figure 4-2 Heap Initialization

[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 Memory Mapped I/O (MMIO) with `EFI_MEMORY_RUNTIME` attribute and the reserved memory in the GCD resource list.

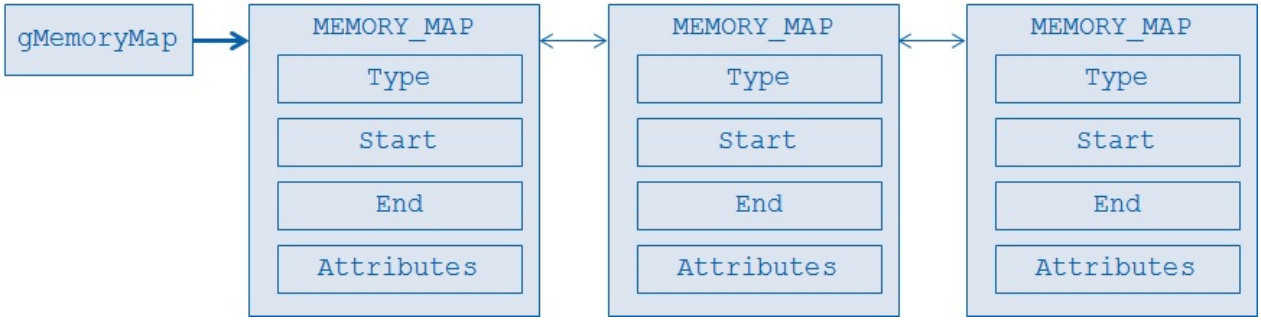


Figure 4-3 Page Management

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.

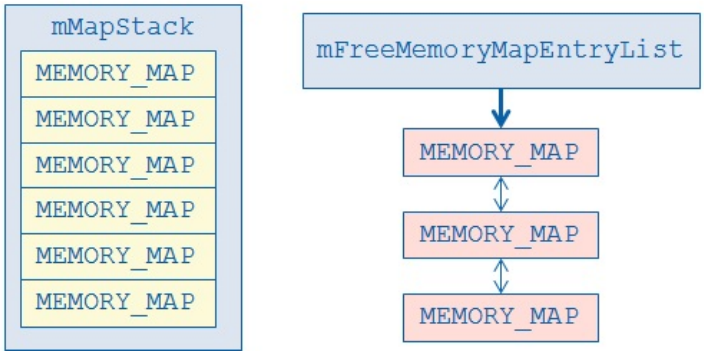
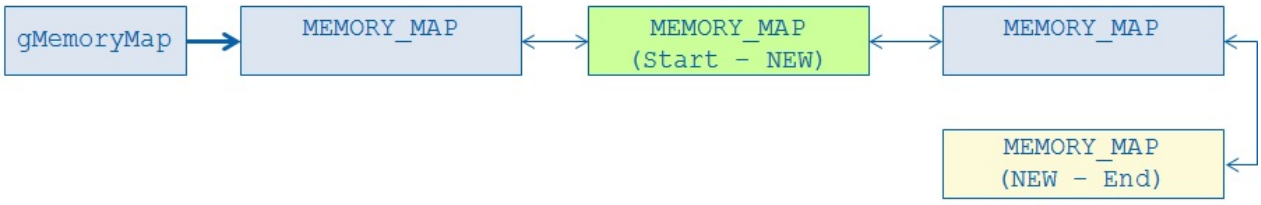


Figure 4-4.1 Page Management – Clip: Step 1

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.



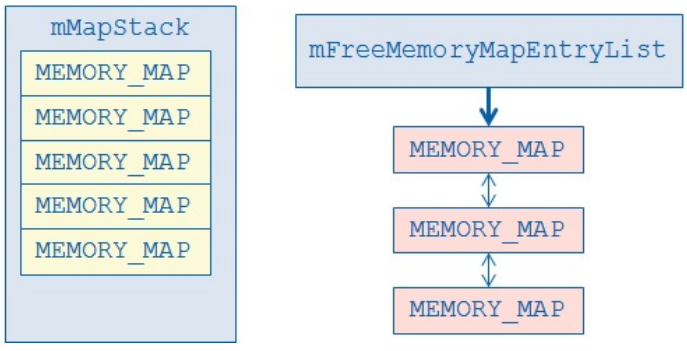
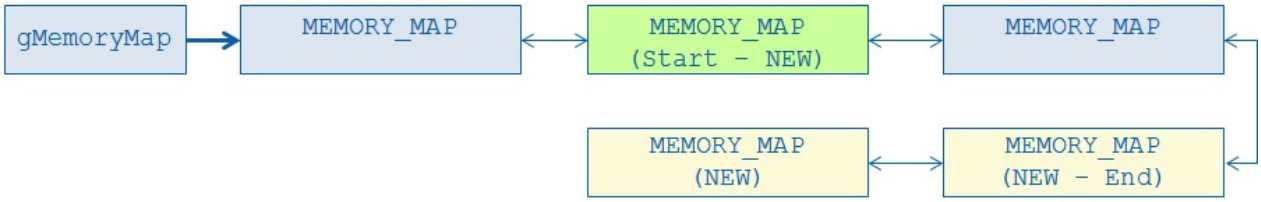
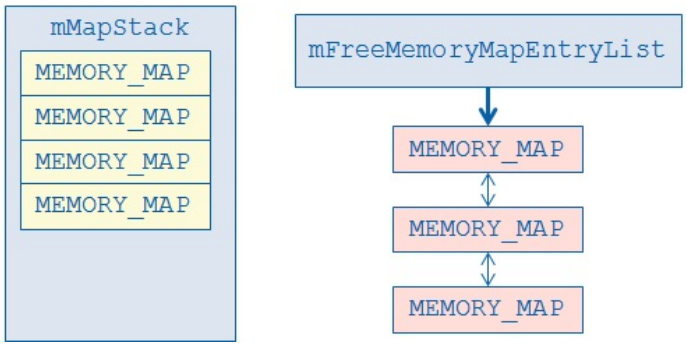


Figure 4-4.2 Page Management – Clip: Step 2

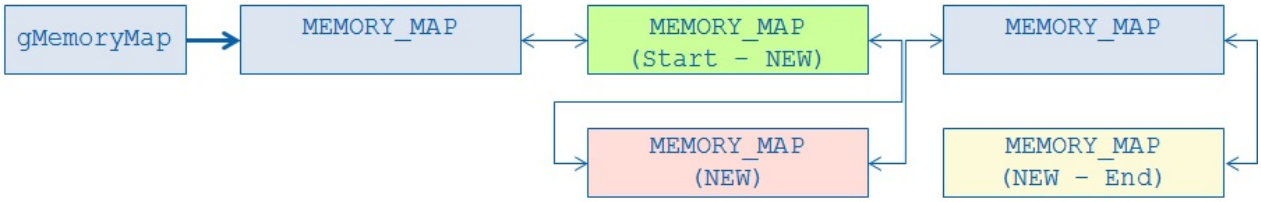
Step 2: The content of this target MEMORY_MAP entry is updated to contain the part of reset memory after allocation. If the allocated memory is in the middle of the entry, one entry (the YELLOW one) in `mMapStack` is popped and added to the memory map linked list. This entry covers the rest of memory. 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.



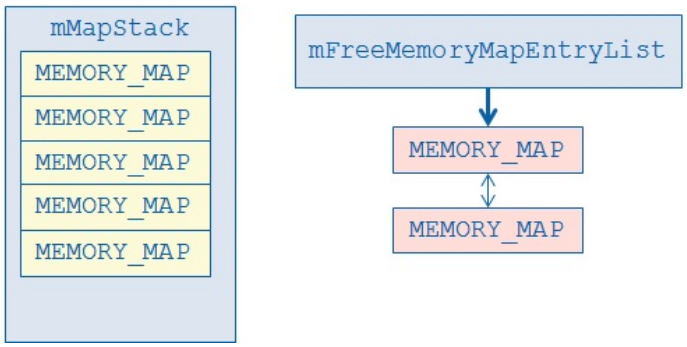


Figure 4-4.4 Page Management – Clip: Step 4

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 dequeue 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 allocation may occur in `AllocateMemoryMapEntry()` if the `mFreeMemoryMapEntryList` is empty.

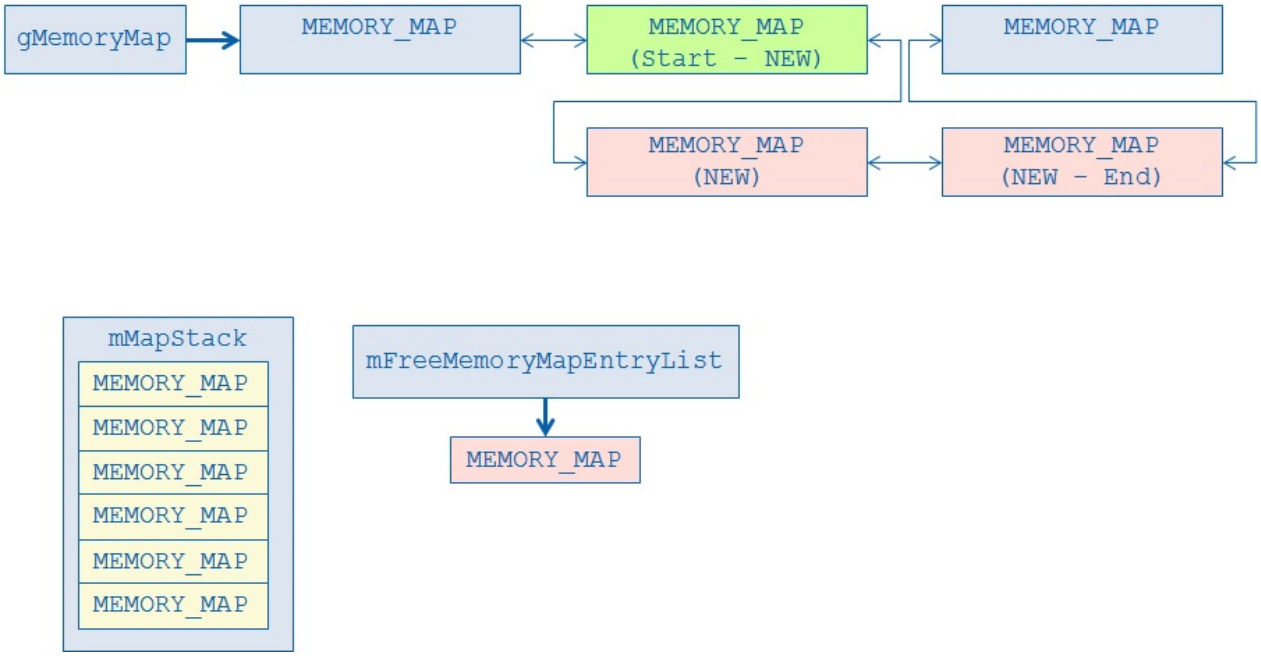


Figure 4-4.5 Page Management – Clip: Step 5

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, 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.

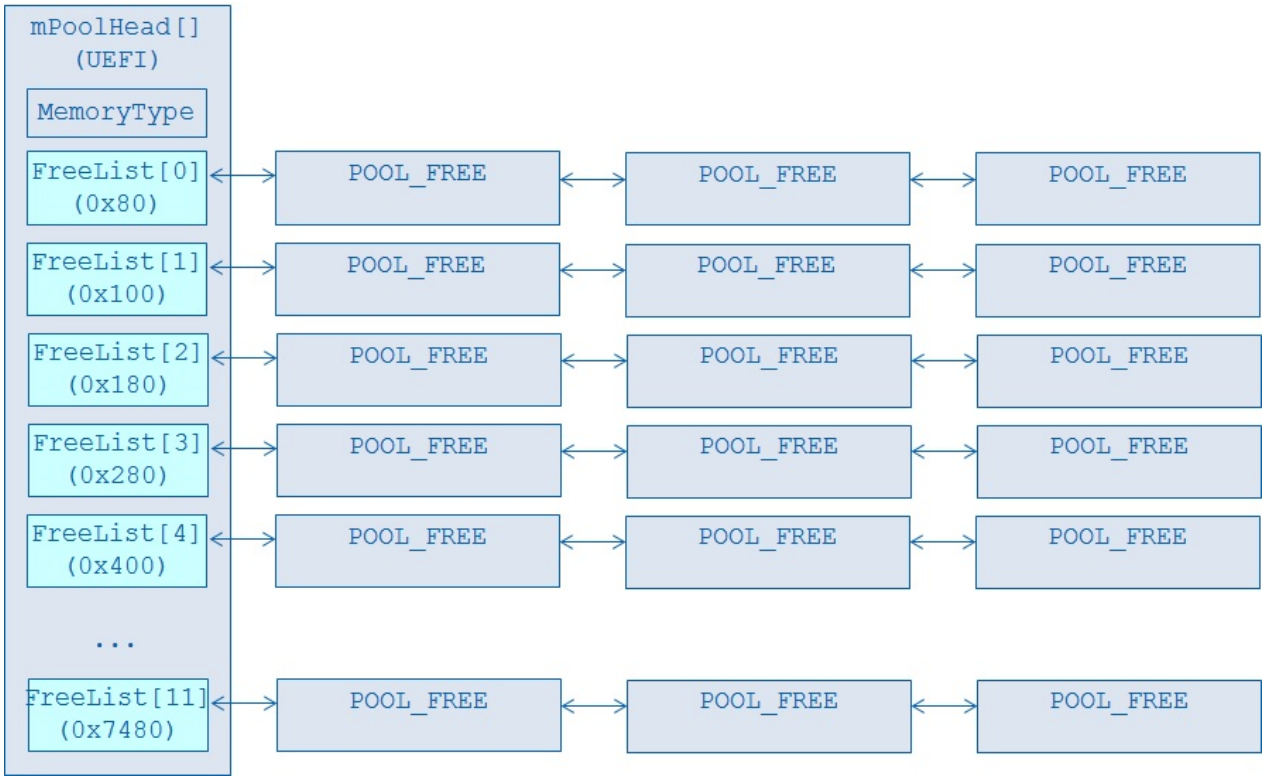


Figure 4-5 Pool Management

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.

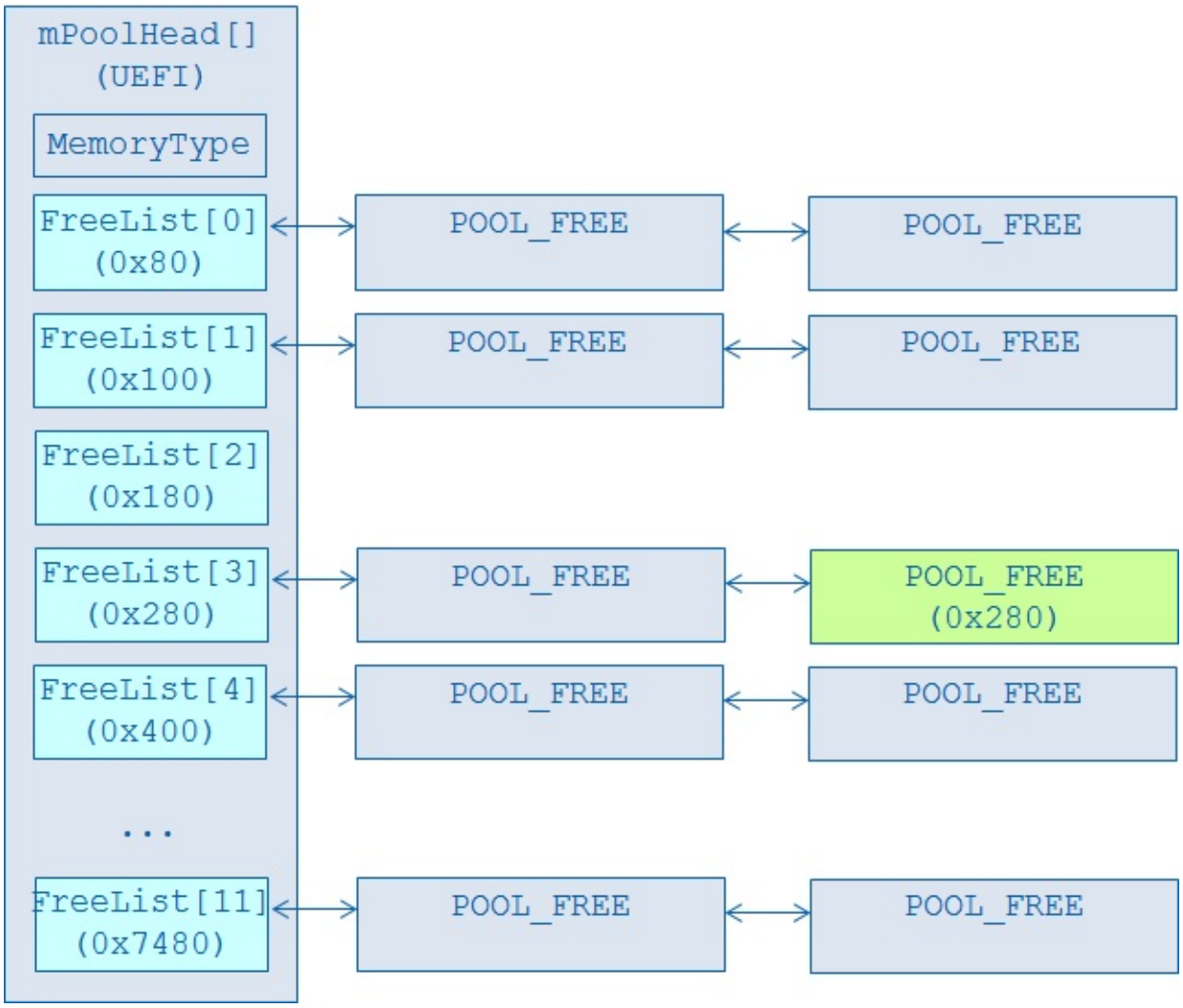


Figure 4-6.1 Pool Management – Carve: Step 1

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.

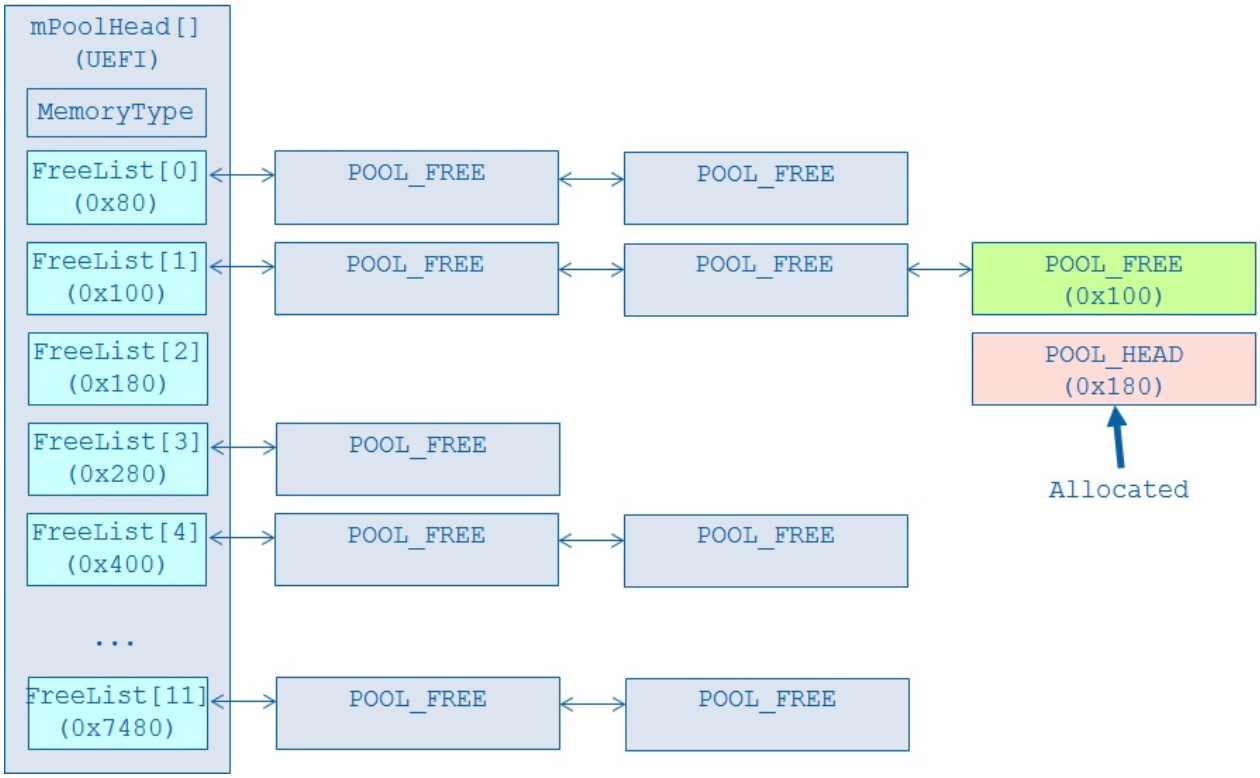


Figure 4-6.2 Pool Management – Carve: Step 2

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.

[1][PI] UEFI Platform Initialization Specification, Version 1.5

Heap Overflow Detection (for Page)

Heap overflow is a big problem. [WindowsHeap][1]discussed some machenisms to detect heap overflow.

In EDK II, we may setup a guard page around the allocated pages. The concept is similar to the guard page for stack. The `gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPageType` indicates which type page allocation need guard page. The `gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPoolType` indicates which type pool allocation need guard page. The `gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPropertyMask` is a mask to control Heap Guard behavior. All these Platform Configuration Database (PCDs) are defined in https://github.com/tianocore/edk2/blob/master/MdeModulePkg/MdeModulePkg.dec.

If heap guard for page allocation is enabled, 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.

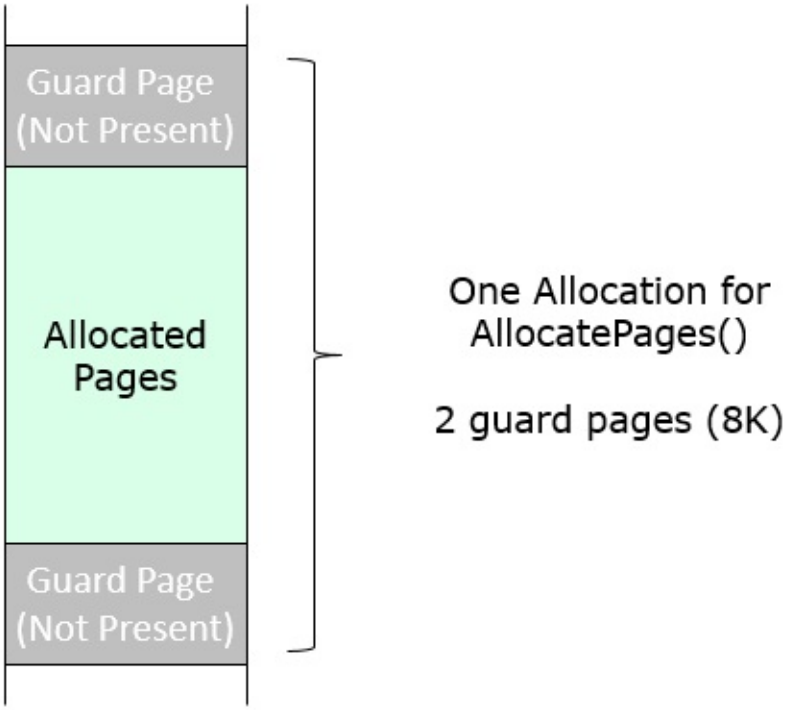


Figure 4-7 HeapGuard for Page Overflow Detection

This enhancement is in https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/Page.c. If the `IsPageTypeToGuard()` returns TRUE, `CoreInternalAllocatePages()` uses `CoreConvertPagesWithGuard()` to allocate 2 more pages and calls `SetGuardForMemory()`. The later calls `SetGuardPage()` twice to set the guard page before and after. `SetGuardPage()` calls `SetMemoryPageAttributes()` to clear PRESENT flag.

In the last step, `SetGuardForMemory()` calls `SetGuardedMemoryBits()` to mark the memory range as guarded. This bitmask will be checked in `UnsetGuardForMemory()` when `FreePages()` is called.

Care must be taken to avoid the 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. The global variable `mOnGuarding` is used to record the protection state.

One interesting feature for `AllocatePages()` is that the pages can be freed partially. Figure 4-8 shows a possible `AllocatePages/FreePages` sequence. `AdjustMemoryF()` is used to adjust the start address and number of pages to free according to Guard. The purpose of this function is to keep the shared Guard page with adjacent memory block if it's still in guard, or free it if no more sharing. Another purpose is to reserve pages as Guard pages in partial page free situation.

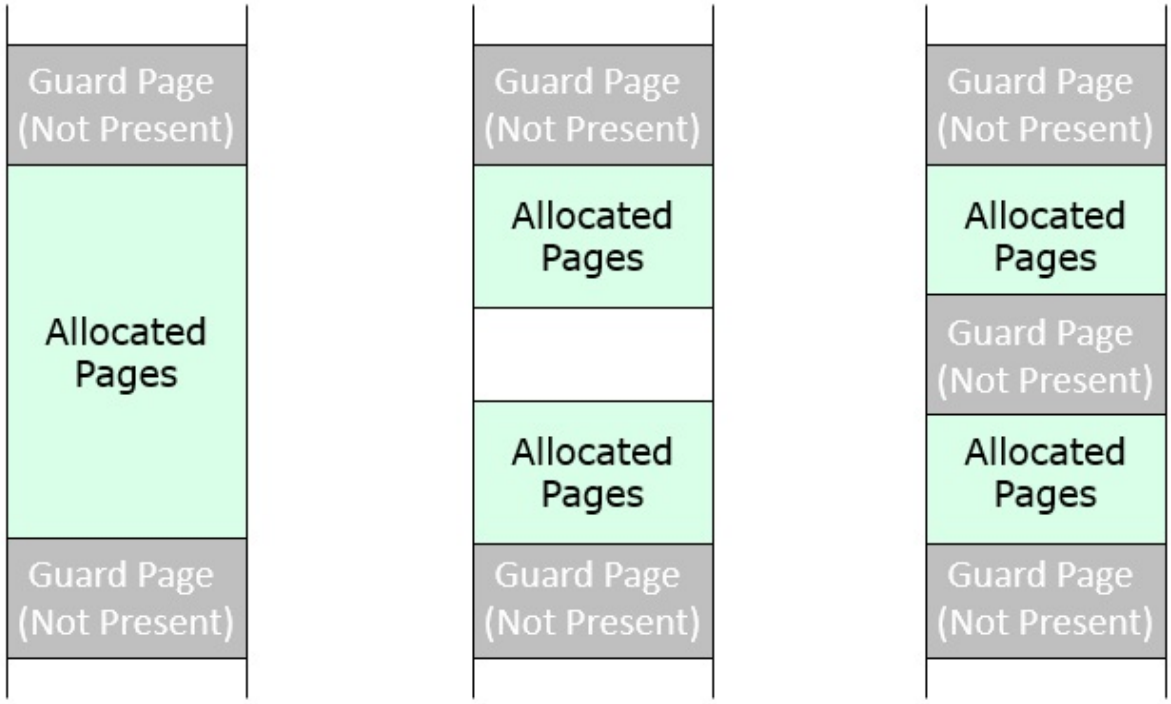


Figure 4-8 HeapGuard in Partial Page Free

Guarded Heap Map:

In order to track if a page is guarded or not, the core defines a guarded memory bitmap table. To simplify the access and reduce the memory used for this table, the table is constructed in the similar way as page table structure but in reverse direction, i.e. from bottom growing up to top. See Figure 4-9.

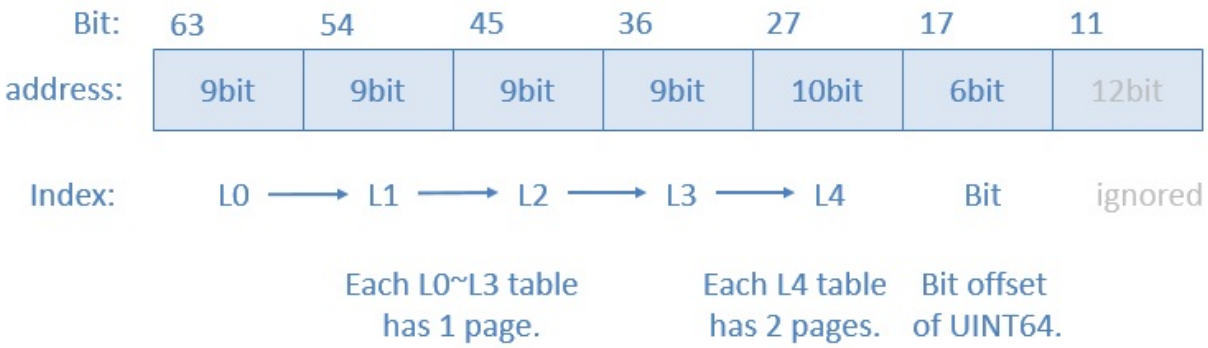


Figure 4-9 Guarded Heap Map

This table uses 1 bit to track 1 page. 1 `UINT64` entry tracks 256K memory. 1024 `UINT64` map table tracks 256M memory. The 5 levels of table can track any address of memory in 64bit system.

For a system with 4G memory, two levels of tables can track the whole memory, because two levels (L3+L4) of map tables have already covered 37-bit of memory address. That means we just need 1-page (L3) + 2-page (L4) memory (3 pages) to track the memory allocation works. In this case, there's no

need to setup L0-L2 tables. Figure 4-10 shows a real example on how to get the guarded page info from an address.

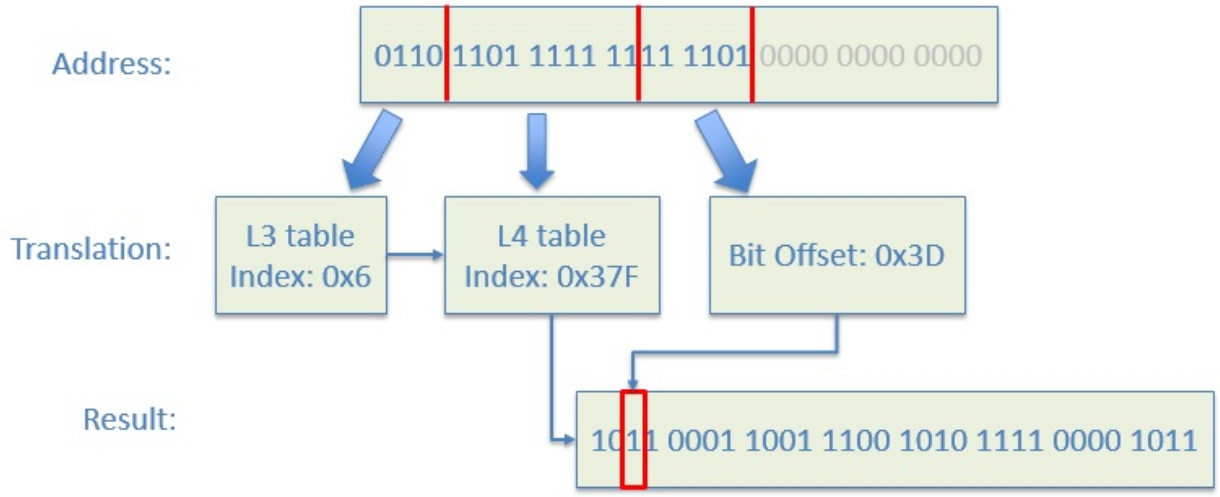


Figure 4-10 Guarded Heap Map (example)

Given the memory address `0x6DFFD000`, the **L3** table index is first 4 bits, the **L4** table index is the 10 bits followed by. The corresponding guarded heap entry in **L4** table index `0x37F – 0xB19CAF0B`. The bit offset is the 6 bits followed by **L4** table index. In this case, it is `0x3D`. Since bit `0x3D` of `0xB19CAF0B` is 1, we can know the page `0x6DFFD000` is guarded.

The guarded heap map management is at https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/HeapGuard.c. `mGuardedMemoryMap` is the pointer to table tracking the Guarded memory with bitmap, `mMapLevel` is the Current depth level of map table pointed by `mGuardedMemoryMap. SetGuardForMemory()` is to set head Guard and tail Guard for the given memory range. `UnsetGuardForMemory()` is to clear head Guard and tail Guard for the given memory range.

[1][WindowsHeap] Preventing the exploitation of user mode heap corruption vulnerabilities, 2009

Heap Overflow Detection (for Pool)

We can use GuardPage for pages. What about pool?

If the pool guard is enabled, each pool allocation becomes the page allocation with the guarded page. Because the pool size might not be a multiple of the page size, we can only set guard page at the head of pool or at the tail of pool. This behavior is controlled by `BIT7` of `gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPropertyMask`. By default the returned pool is near the tail guard page to check overflow. If the `BIT7` is set, the returned pool is near the head guard page to check underflow. See figure 4-11.

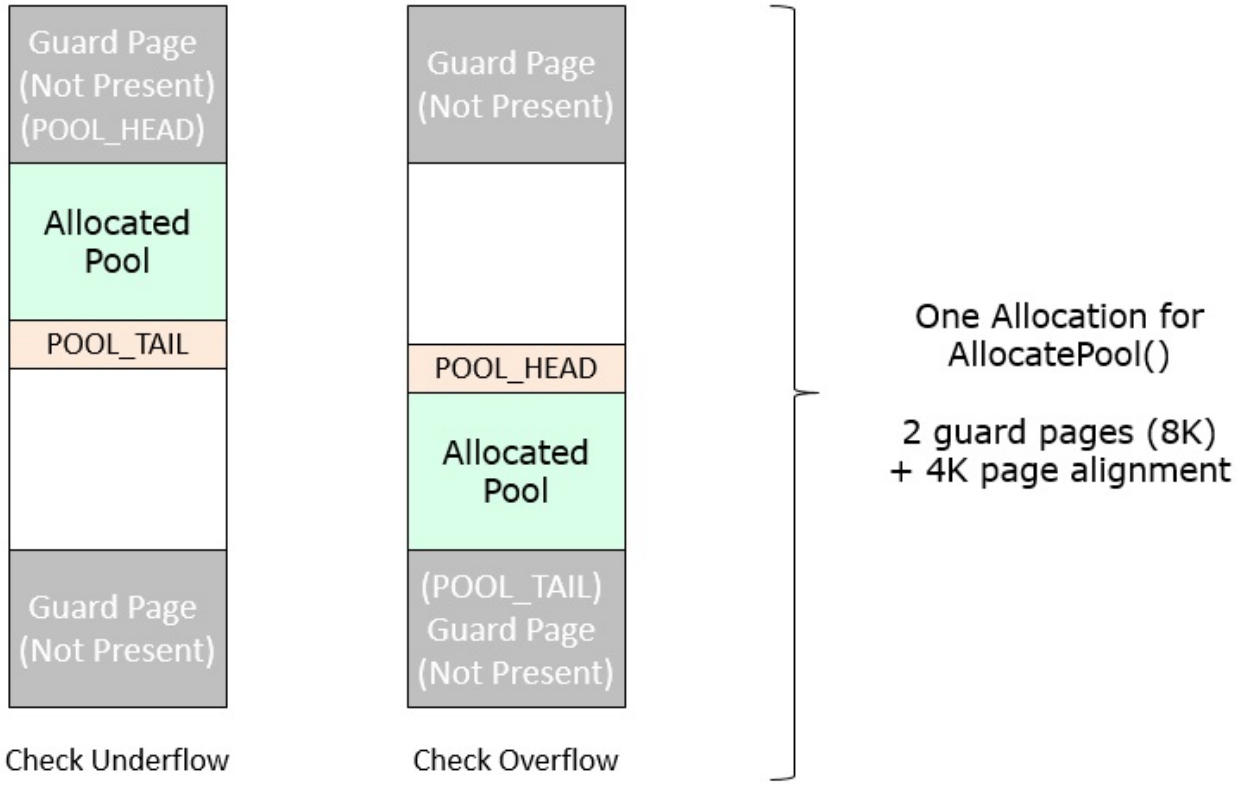


Figure 4-11 HeapGuard for Pool Overflow Detection

This enhancement for the guarded pool is in https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Mem/Pool.c. If the `IsPoolTypeToGuard()` returns TRUE, `CoreAllocatePoolI()` uses `CoreAllocatePoolPagesI()` directly to allocate pages for pool with the guarded pages. In the last step, `SetGuardForMemory()` is invoked to set the guard page around the allocated pool.

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 Compatible Support Module(CSM) or 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 `gEfiMdeModulePkgTokenSpaceGuid.PcdNullPointerDetectionPropertyMask` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/MdeModulePkg.dec). If the `BIT0` of `PcdNullPointerDetectionPropertyMask` is set, the https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/DxeIplPeim clears the PRESENT bit of the address zero page. As such, a Page Fault exception will be generated if some program access to address zero. The `BIT1` of `PcdNullPointerDetectionPropertyMask` controls the NULL pointer detection in System Management Mode (SMM) environment, it is referred by https://github.com/tianocore/edk2/tree/master/UefiCpuPkg/PiSmmCpuDxeSmm. The `BIT7` of `PcdNullPointerDetectionPropertyMask` disables NULL pointer detection just after `EndOfDxe`. This is a workaround for those unsolvable NULL access issues in OptionROM, boot loader, etc. It can also help to avoid unnecessary exception caused by legacy memory (`0-4095`) access after EndOfDxe, such as Windows 7* boot on QEMU*. `BIT7` is checked by https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Core/Dxe.

Read-only page table

Because we use CPU page table to provide such detection, `DxeIpl` does one more enhancement to set page table itself to be read only.
(https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/DxeIplPeim/X64/VirtualMemory.c `SetPageTablePoolReadOnly()`) If the buffer overflow impacts the page table, it can be detected immediately.

The CPU driver needs to be aware of this and carefully clear `CR0_WP` bit before modifying the page table and restore `CR0` after modification.
(https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/CpuDxe/CpuPageTable.c)

Limitation

1. **Size**

   If the heap page guard is enabled, the core allocates additional 2 pages for each `AllocatePages()`. If the heap pool guard is enabled, each `AllocatePool()` becomes `AllocatePages()`. Even 1 byte allocation need 12K memory. The heap guard feature will increase memory consumption and may cause memory out of resource. Especially, the System Management Mode (SMM) code runs in the limited System Management Mode RAM (SMRAM) (4M or 8M).

   We have observed SMRAM out of resource when this feature is turned on. The solution could be:

   - Enlarge SMRAM
   - Enable the partial of PageGuard or PoolGuard feature, via `gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPageType` and `gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPoolType` PCD.
   - Enable feature one-by-one. The user can build 2 or more BIOS images, and each image include a subset of full features. For example, if the user wants to detect overflow in USB drivers, there is no need to include network drivers.

2. **Performance**

   If the heap guard is enabled, the CPU driver need update the page table for every `AllocatePool()`, flush translation lookaside buffer (TLB). It brings overhead.

   We have observed the performance downgrade in UEFI Shell, such as the `DEVTREE` command or shell script.

   Some of the performance drop can be resolved by removing the page table synchronization from Boot Strap Processor (BSP) to Application Processors (AP)s. When BSP wakes up APs later, the page table is always rewritten. Paging synchronization is unnecessary.

3. **PEI Phase**

   The guard in Pre-EFI Initialization (PEI) phase is not supported yet, because most Intel&reg Architecture (IA) platforms only supports 32bit PEI and paging is not enabled.

   From technical perspective, we can add paging-based guard after the permanent memory is initialized in PEI. Stack guard, heap guard or NULL pointer detection can be enabled.

   Before the permanent memory is initialize, if we need enabling paging, we can only set paging table on A) Flash Region with Access and Dirty bit set. B) Cache as Ram. C) Static RAM. `#A` can only support read-only paging, while `#B` and `#C` has size limitation.

   In the EDK II community, Brian Johnson also provided another way to enable the NULL pointer detection (https://bugzilla.tianocore.org/show_bug.cgi?id=687). We can set “the GDT descriptor for the data segment from a "extend up" type based at 0, to an "extend down" type with a limit of 0. This disables access to the 4k page at 0 due to the way the limit math works with "extend down" descriptors.”

4. **Pool Underflow/Overflow detection**

   For heap pool detection, we cannot enable both underflow and overflow detection in one image, because the guard page must be 4K aligned and the allocated pool is either adjacent to head guard page or tail guard page.

   In order to detect underflow and overflow, the user must build 2 different images with `BIT7` of `gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPropertyMask` set or cleared.

5. **Emulation Environment**

Current emulation environment (such as NT32) does not have capability to modify CPU page table directly. As such we are not able to enable such page table based protection.

In the future, we may be able to set up memory for NT32 application process and then use this function to change page protections for pages within that process to match what we have done for the real hardware. For example: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa366898(v=vs.85).aspx]

```
BOOL WINAPI VirtualProtect(
_In_  LPVOID lpAddress,
_In_  SIZE_T dwSize,
_In_  DWORD  flNewProtect,
_Out_ PDWORD lpflOldProtect
);
```

`FlNewProtect` values: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa366786(v=vs.85).aspx]

Compatibility Consideration

1. **OS compatibility**

   If the heap guard is enabled (especially pool guard), the UEFI memory map is fragmented. We notice that the fragmentation may cause OS boot or installation fail. For example, Windows Boot Manager sets the maximum number of global memory descriptor for a 64-bit UEFI system at 512. (https://support.microsoft.com/en-us/help/4020050/blinitializelibrary-failed-xxx-error-when-you-install-or-start-an-oper)

   For NULL pointer detection, `BIT7` of `PcdNullPointerDetectionPropertyMask` (disables NULL pointer detection just after EndOfDxe) is a workaround for those unsolvable NULL access issues in Option ROM, OS boot loader, such as Windows 7* boot on Qemu*.

2. **3rd part driver (PCI Option ROM)**

   When the heap guard is enabled, we observed the `#PF` (Page Fault) exception happens and the instruction address is within a 3rd part PCI Option ROM.

3. **Legacy Compatible Support Module (CSM)**

   The legacy CSM module may need access the zero address and the first 4KiB memory, such as Interrupt Vector Table(IVT) and BIOS Data Area (BDA). Whenever the legacy module accesses the first 4KiB memory, the code must be included by `ACCESS_PAGE0_CODE()` macro. (https://github.com/tianocore/edk2/blob/master/IntelFrameworkPkg/Include/Protocol/LegacyBios.h). For example, when the `LegacyBios` driver accesses IVT region (https://github.com/tianocore/edk2/blob/master/IntelFrameworkModulePkg/Csm/LegacyBiosDxe/Thunk.c) or when the `LegacyKeyboard` driver accesses BDA region (https://github.com/tianocore/edk2/blob/master/IntelFrameworkModulePkg/Csm/BiosThunk/KeyboardDxe/BiosKeyboard.c).

   The platform specific `LegacyBiosPlatform` driver also need update to add `ACCESS_PAGE0_CODE()` macro around the code to update IVT or BDA.

4. **CPU driver**

   The EDK II `DxeCore` relies on CPU driver `EFI_CPU_ARCH_PROTOCOL.SetMemoryAttributes` (https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Protocol/Cpu.h) to update the page table (https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/CpuDxe/CpuPageTable.c). The EDK II `PiSmmCore` also relies on `PiSmmCpu` driver `EDKII_SMM_MEMORY_ATTRIBUTE_PROTOCOL` (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Protocol/SmmMemoryAttribute.h)to update the page table (https://github.com/tianocore/edk2/blob/master/UefiCpuPkg/PiSmmCpuDxeSmm/SmmCpuMemoryManagement.c). If a platform uses its own CPU driver instead of the open source one, this platform need add the page table update capability in the CPU driver.

Call for action

1. Enable the **Stack Overflow detection** to catch stack overflow (especially in System Management Mode (SMM))
    i. `gEfiMdeModulePkgTokenSpaceGuid.PcdCpuStackGuard`
    ii. `gUefiCpuPkgTokenSpaceGuid.PcdCpuSmmStackGuard`
2. Enable the **Heap Overflow detection** in a debug BIOS to catch potential buffer overflow issue.
    i. `gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPageType`
    ii. `gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPoolType`
    iii. `gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPropertyMask`
3. Enable the **NULL pointer detection** to catch the NULL address access.
    i. `gEfiMdeModulePkgTokenSpaceGuid.PcdNullPointerDetectionPropertyMask`

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] [1] to refactoring code to a type-safe language [RUST][2]. 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][3][MPX][4]. Intel MPX can help detect the buffer overflow or underflow with a set of new Intel® MPX instructions and the compiler support. When Intel® 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. Microsoft MSVC 2015* Update 1 and GCC 5.1* supports Intel MPX.

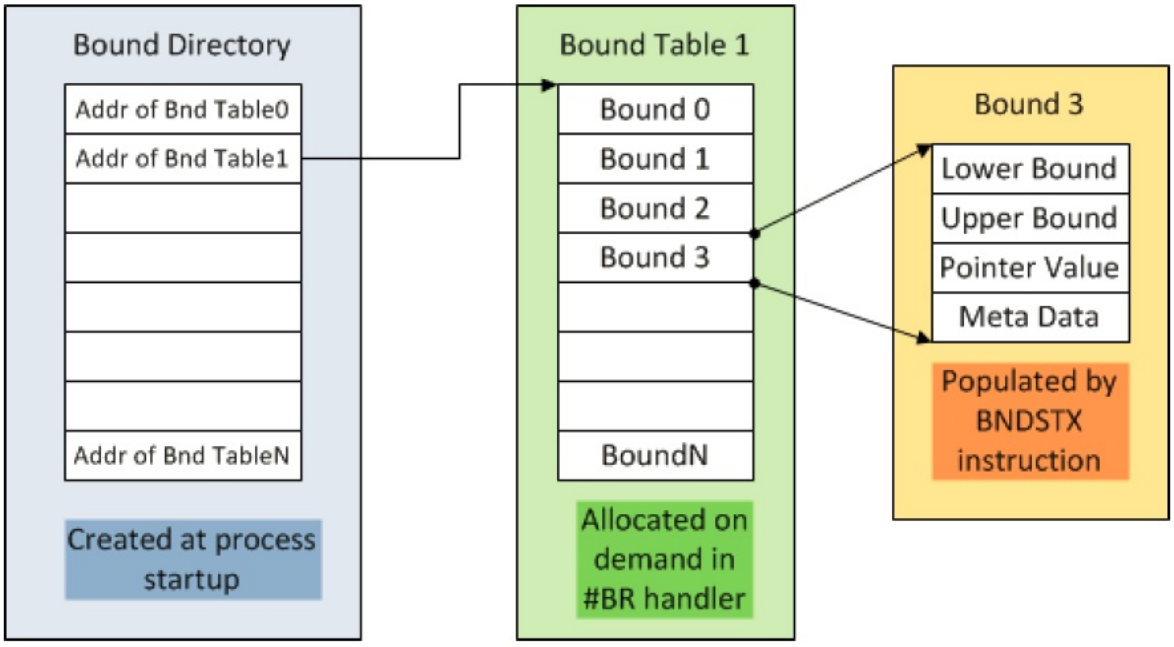


Figure 4-12 Bound Table, (source: [MPX][4]

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.

[1][CHECKED_C] Checked C [2] [RUST] Rust language [3][IA32SDM]Intel® 64 and IA-32 Architectures Software Developer’s Manual [4][MPX] Intel® Memory Protection Extensions Enabling Guide

SUMMARY

Below table list a set of core security related feature and their compatibility.

NOTE:

- BLUE means a production feature which might be enabled in the final production.
- YELLOW means a debug feature which need be disabled in the final production.
- “V” means 2 features can be enabled together.
- “N” means 2 feature must not be enabled together.
- (*) means this feature is for System Management Mode (SMM) only.
- No (*) means this feature can be enabled for DXE or SMM.

Compatbility with other features

	Stack Guard	NULL Ptr	Heap Guard	Mem Profile	NX Stack	NX/RO Mem	Image Protect	Static Paging	SMI Profile	SMM Profile
Stack Guard	N/A	V	V	V	V	V	V	V	V	V
NULL Pointer	V	N/A	V	V	V	V	V	V	V	V
Heap Guard	V	V	N/A	V	V	V	V	N	V	V
Mem Profile	V	V	V	N/A	V	V	V	V	V	V
NX Stack	V	V	V	V	N/A	V	V	V	V	V
NX/RO Memory	V	V	V	V	V	N/A	V	V	V	V
Image Protect	V	V	V	V	V	V	N/A	V	V	V
SMM Static Paging (*)	V	V	N	V	V	V	V	N/A	V	N
SMI Profile (*)	V	V	V	V	V	V	V	V	N/A	V
SMM Profile (*)	V	V	V	V	V	V	V	N	V	N/A

Production Feature	Debug Feature

System Management Mode (SMM) static paging feature is the only one what cannot be enabled with other features if the other features need modify the page table.

Policy Control

Stack Guard: Detect Stack Overflow

```
gEfiMdeModulePkgTokenSpaceGuid.PcdCpuStackGuard
  ## Indicates if UEFI Stack Guard will be enabled.
  #  If enabled, stack overflow in UEFI can be caught, preventing chaotic consequences.<BR><BR>
  #   TRUE  - UEFI Stack Guard will be enabled.<BR>
  #   FALSE - UEFI Stack Guard will be disabled.<BR>

gUefiCpuPkgTokenSpaceGuid.PcdCpuSmmStackGuard
  ## Indicates if SMM Stack Guard will be enabled.
  #  If enabled, stack overflow in SMM can be caught, preventing chaotic consequences.<BR><BR>
  #   TRUE  - SMM Stack Guard will be enabled.<BR>
  #   FALSE - SMM Stack Guard will be disabled.<BR>
```

NULL pointer detection: Detect NULL pointer access

```
gEfiMdeModulePkgTokenSpaceGuid.PcdNullPointerDetectionPropertyMask
  ## Mask to control the NULL address detection in code for different phases.
  #  If enabled, accessing NULL address in UEFI or SMM code can be caught.<BR><BR>
  #    BIT0    - Enable NULL pointer detection for UEFI.<BR>
  #    BIT1    - Enable NULL pointer detection for SMM.<BR>
  #    BIT2..6 - Reserved for future uses.<BR>
  #    BIT7    - Disable NULL pointer detection just after EndOfDxe. <BR>
  #              This is a workaround for those unsolvable NULL access issues in
  #              OptionROM, boot loader, etc. It can also help to avoid unnecessary
  #              exception caused by legacy memory (0-4095) access after EndOfDxe,
  #              such as Windows 7 boot on Qemu.<BR>
```

Heap Guard: Detect Heap Overflow.

```
gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPageType
  ## Indicates which type allocation need guard page.
  #
  # If a bit is set, a head guard page and a tail guard page will be added just
  # before and after corresponding type of pages allocated if there's enough
  # free pages for all of them. The page allocation for the type related to
  # cleared bits keeps the same as ususal.
  #
  # Below is bit mask for this PCD: (Order is same as UEFI spec)<BR>
  #  EfiReservedMemoryType             0x0000000000000001<BR>
  #  EfiLoaderCode                     0x0000000000000002<BR>
  #  EfiLoaderData                     0x0000000000000004<BR>
  #  EfiBootServicesCode               0x0000000000000008<BR>
  #  EfiBootServicesData               0x0000000000000010<BR>
  #  EfiRuntimeServicesCode            0x0000000000000020<BR>
  #  EfiRuntimeServicesData            0x0000000000000040<BR>
  #  EfiConventionalMemory             0x0000000000000080<BR>
  #  EfiUnusableMemory                 0x0000000000000100<BR>
  #  EfiACPIReclaimMemory              0x0000000000000200<BR>
  #  EfiACPIMemoryNVS                  0x0000000000000400<BR>
  #  EfiMemoryMappedIO                 0x0000000000000800<BR>
  #  EfiMemoryMappedIOPortSpace        0x0000000000001000<BR>
  #  EfiPalCode                        0x0000000000002000<BR>
  #  EfiPersistentMemory               0x0000000000004000<BR>
  #  OEM Reserved                      0x4000000000000000<BR>
  #  OS Reserved                       0x8000000000000000<BR>
  # e.g. LoaderCode+LoaderData+BootServicesCode+BootServicesData are needed, 0x1E should be used.<BR>
```

```
gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPoolType
  ## Indicates which type allocation need guard page.
```

```
  #
  # If a bit is set, a head guard page and a tail guard page will be added just
  # before and after corresponding type of pages which the allocated pool occupies,
  # if there's enough free memory for all of them. The pool allocation for the
  # type related to cleared bits keeps the same as ususal.
  #
  # Below is bit mask for this PCD: (Order is same as UEFI spec)<BR>
  #  EfiReservedMemoryType             0x0000000000000001<BR>
  #  EfiLoaderCode                     0x0000000000000002<BR>
  #  EfiLoaderData                     0x0000000000000004<BR>
  #  EfiBootServicesCode               0x0000000000000008<BR>
  #  EfiBootServicesData               0x0000000000000010<BR>
  #  EfiRuntimeServicesCode            0x0000000000000020<BR>
  #  EfiRuntimeServicesData            0x0000000000000040<BR>
  #  EfiConventionalMemory             0x0000000000000080<BR>
  #  EfiUnusableMemory                 0x0000000000000100<BR>
  #  EfiACPIReclaimMemory              0x0000000000000200<BR>
  #  EfiACPIMemoryNVS                  0x0000000000000400<BR>
  #  EfiMemoryMappedIO                 0x0000000000000800<BR>
  #  EfiMemoryMappedIOPortSpace        0x0000000000001000<BR>
  #  EfiPalCode                        0x0000000000002000<BR>
  #  EfiPersistentMemory               0x0000000000004000<BR>
  #  OEM Reserved                      0x4000000000000000<BR>
  #  OS Reserved                       0x8000000000000000<BR>
  # e.g. LoaderCode+LoaderData+BootServicesCode+BootServicesData are needed, 0x1E should be used.<BR>

gEfiMdeModulePkgTokenSpaceGuid.PcdHeapGuardPropertyMask
  ## This mask is to control Heap Guard behavior.
  # Note that due to the limit of pool memory implementation and the alignment
  # requirement of UEFI spec, BIT7 is a try-best setting which cannot guarantee
  # that the returned pool is exactly adjacent to head guard page or tail guard
  # page.
  #   BIT0 - Enable UEFI page guard.<BR>
  #   BIT1 - Enable UEFI pool guard.<BR>
  #   BIT2 - Enable SMM page guard.<BR>
  #   BIT3 - Enable SMM pool guard.<BR>
  #   BIT7 - The direction of Guard Page for Pool Guard.
  #          0 - The returned pool is near the tail guard page.<BR>
  #          1 - The returned pool is near the head guard page.<BR>
```

Memory Profile: Provide memory usage information, detect memory leak

```
gEfiMdeModulePkgTokenSpaceGuid.PcdMemoryProfilePropertyMask
  ## The mask is used to control memory profile behavior.<BR><BR>
  #  BIT0 - Enable UEFI memory profile.<BR>
  #  BIT1 - Enable SMRAM profile.<BR>
  #  BIT7 - Disable recording at the start.<BR>

gEfiMdeModulePkgTokenSpaceGuid.PcdMemoryProfileMemoryType
  ## This flag is to control which memory types of alloc info will be recorded by DxeCore & SmmCore.<BR><BR>
  # For SmmCore, only EfiRuntimeServicesCode and EfiRuntimeServicesData are valid.<BR>
  #
  # Below is bit mask for this PCD: (Order is same as UEFI spec)<BR>
  #  EfiReservedMemoryType          0x0001<BR>
  #  EfiLoaderCode                  0x0002<BR>
  #  EfiLoaderData                  0x0004<BR>
  #  EfiBootServicesCode            0x0008<BR>
  #  EfiBootServicesData            0x0010<BR>
  #  EfiRuntimeServicesCode         0x0020<BR>
  #  EfiRuntimeServicesData         0x0040<BR>
  #  EfiConventionalMemory          0x0080<BR>
  #  EfiUnusableMemory              0x0100<BR>
  #  EfiACPIReclaimMemory           0x0200<BR>
  #  EfiACPIMemoryNVS               0x0400<BR>
  #  EfiMemoryMappedIO              0x0800<BR>
  #  EfiMemoryMappedIOPortSpace     0x1000<BR>
  #  EfiPalCode                     0x2000<BR>
  #  EfiPersistentMemory            0x4000<BR>
```

```
  #  OEM Reserved       0x4000000000000000<BR>
  #  OS Reserved        0x8000000000000000<BR>
  #
  # e.g. Reserved+ACPINvs+ACPIReclaim+RuntimeCode+RuntimeData are needed, 0x661 should be used.<BR>

gEfiMdeModulePkgTokenSpaceGuid.PcdMemoryProfileDriverPath
  ## This PCD is to control which drivers need memory profile data.<BR><BR>
  # For example:<BR>
  # One image only (Shell):<BR>
  #     Header                    GUID<BR>
  #     {0x04, 0x06, 0x14, 0x00,  0x83, 0xA5, 0x04, 0x7C, 0x3E, 0x9E, 0x1C, 0x4F, 0xAD, 0x65, 0xE0, 0x52, 0x68, 0xD0, 0xB4, 0x
D1,<BR>
  #      0x7F, 0xFF, 0x04, 0x00}<BR>
  # Two or more images (Shell + WinNtSimpleFileSystem):<BR>
  #     {0x04, 0x06, 0x14, 0x00,  0x83, 0xA5, 0x04, 0x7C, 0x3E, 0x9E, 0x1C, 0x4F, 0xAD, 0x65, 0xE0, 0x52, 0x68, 0xD0, 0xB4, 0x
D1,<BR>
  #      0x7F, 0x01, 0x04, 0x00,<BR>
  #      0x04, 0x06, 0x14, 0x00,  0x8B, 0xE1, 0x25, 0x9C, 0xBA, 0x76, 0xDA, 0x43, 0xA1, 0x32, 0xDB, 0xB0, 0x99, 0x7C, 0xEF, 0x
EF,<BR>
  #      0x7F, 0xFF, 0x04, 0x00}<BR>
```

NX stack: Prevent code execution in stack

```
gEfiMdeModulePkgTokenSpaceGuid.PcdSetNxForStack
  ## Indicates if to set NX for stack.<BR><BR>
  #  For the DxeIpl and the DxeCore are both X64, set NX for stack feature also require PcdDxeIplBuildPageTables be TRUE.<BR>
  #  For the DxeIpl and the DxeCore are both IA32 (PcdDxeIplSwitchToLongMode is FALSE), set NX for stack feature also require
  #  IA32 PAE is supported and Execute Disable Bit is available.<BR>
  #   TRUE  - to set NX for stack.<BR>
  #   FALSE - Not to set NX for stack.<BR>
```

DXE NX/RO Protection: Prevent code injection

```
gEfiMdeModulePkgTokenSpaceGuid.PcdDxeNxMemoryProtectionPolicy
  ## Set DXE memory protection policy. The policy is bitwise.
  #  If a bit is set, memory regions of the associated type will be mapped
  #  non-executable.<BR><BR>
  #
  # Below is bit mask for this PCD: (Order is same as UEFI spec)<BR>
  #  EfiReservedMemoryType          0x0001<BR>
  #  EfiLoaderCode                  0x0002<BR>
  #  EfiLoaderData                  0x0004<BR>
  #  EfiBootServicesCode            0x0008<BR>
  #  EfiBootServicesData            0x0010<BR>
  #  EfiRuntimeServicesCode         0x0020<BR>
  #  EfiRuntimeServicesData         0x0040<BR>
  #  EfiConventionalMemory          0x0080<BR>
  #  EfiUnusableMemory              0x0100<BR>
  #  EfiACPIReclaimMemory           0x0200<BR>
  #  EfiACPIMemoryNVS               0x0400<BR>
  #  EfiMemoryMappedIO              0x0800<BR>
  #  EfiMemoryMappedIOPortSpace     0x1000<BR>
  #  EfiPalCode                     0x2000<BR>
  #  EfiPersistentMemory            0x4000<BR>
  #  OEM Reserved       0x4000000000000000<BR>
  #  OS Reserved        0x8000000000000000<BR>
  #
  # NOTE: User must NOT set NX protection for EfiLoaderCode / EfiBootServicesCode / EfiRuntimeServicesCode. <BR>
  #       User MUST set the same NX protection for EfiBootServicesData and EfiConventionalMemory. <BR>
  #
  # e.g. 0x7FD5 can be used for all memory except Code. <BR>
  # e.g. 0x7BD4 can be used for all memory except Code and ACPINVS/Reserved. <BR>
```

DXE image Protection: Prevent code injection

```
gEfiMdeModulePkgTokenSpaceGuid.PcdImageProtectionPolicy
  ## Set image protection policy. The policy is bitwise.
  #  If a bit is set, the image will be protected by DxeCore if it is aligned.
  #   The code section becomes read-only, and the data section becomes non-executable.
  #  If a bit is clear, the image will not be protected.<BR><BR>
  #    BIT0       - Image from unknown device. <BR>
  #    BIT1       - Image from firmware volume.<BR>
```

System Management Mode (SMM) static paging: Provide code injection in SMM

```
gUefiCpuPkgTokenSpaceGuid.PcdCpuSmmStaticPageTable
  ## Indicates if SMM uses static page table.
  #  If enabled, SMM will not use on-demand paging. SMM will build static page table for all memory.
  #  This flag only impacts X64 build, because SMM always builds static page table for IA32.
  #  It could not be enabled at the same time with SMM profile feature (PcdCpuSmmProfileEnable).
  #  It could not be enabled also at the same time with heap guard feature for SMM
  #  (PcdHeapGuardPropertyMask in MdeModulePkg).<BR><BR>
  #    TRUE  - SMM uses static page table for all memory.<BR>
  #    FALSE - SMM uses static page table for below 4G memory and use on-demand paging for above 4G memory.<BR>
```

System Management Mode Interrupt (SMI) Handler Profile: Provide SMI handler information

```
gEfiMdeModulePkgTokenSpaceGuid.PcdSmiHandlerProfilePropertyMask
  ## The mask is used to control SmiHandlerProfile behavior.<BR><BR>
  #  BIT0 - Enable SmiHandlerProfile.<BR>
```

SMM Profile: Provide non-SMRAM access in SMM

```
gUefiCpuPkgTokenSpaceGuid.PcdCpuSmmProfileEnable
  ## Indicates if SMM Profile will be enabled.
```

REFERENCES

[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

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

[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

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

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

[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

[MemProtection] A Tour Beyond BIOS- Memory Protection in UEFI BIOS, Jiewen Yao, Vincent Zimmer, 2017 https://www.gitbook.com/book/edk2-docs/a-tour-beyond-bios-memory-protection-in-uefi-bios/details

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

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

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

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

[MSVCFASTFAIL] __fastfail, [https://msdn.microsoft.com/en-us/library/dn774154.aspx](https://msdn.microsoft.com/en-us/library/dn774154.asp

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

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

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

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

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

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

[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

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

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

[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

[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

[WindowsInternal] Windows Internals, 6th edition, Mark E. Russinovich, David A. Solomon, Alex Ionescu, 2012, Microsoft Press. ISBN-13: 978-0735648739/978-0735665873 https://www.amazon.com/Windows-Internals-Part-Developer-Reference/dp/0735648735