Exploring Control Flow Guard in Windows 10

Jack Tang, Trend Micro Threat Solution Team

As operating system developers are always keen on improving exploit mitigation technology, Microsoft has enabled a new mechanism in Windows 10 and in Windows 8.1 Update 3 (released last November) by default. This technology is called Control Flow Guard (CFG).

Like other exploit mitigation mechanisms, such as address space layout randomization (ASLR), and data execution prevention (DEP), it will be successful in making exploitation of vulnerabilities more difficult. Beyond doubt, it will greatly change the attacker's exploit technology. It's like that ALSR results in heap spray techniques showing up and DEP results in return-oriented-programming (ROP) techniques showing up in the exploit code.

To explore this particular technology, I used the Windows 10 Technical Preview (build 6.4.9841), with test applications built using the Visual Studio 2015 Preview. Because the CFG implementation of the latest Windows 10 technical preview build (10.0.9926) has a slight change at that same point, I will point out the difference.

To fully implement CFG, both the compiler and the operating system must support it properly. As an exploit mitigation mechanism in the system level, the CFG implementation requires cooperation from the compiler, the operating system user mode library, and the kernel mode module. A blog post on MSDN outlined the steps that developers need to do to support CFG.

Microsoft's implementation of CFG is focused on indirect call protection. Consider the following code in the test program I created:

```c
typedef int(*fun_t)(int);
int foo(int a)
{
    printf("hello jack: %d\n", a);
    return a;
}
class CTargetObject
{
public:
    fun_t _fun;
};
int _tmain(int argc, _TCHAR* argv[])
{
    int i = 0;
    CTargetObject* o_array = new CTargetObject[5];
    for (i = 0; i < 1000; i++)
        o_array[i]._fun = foo;
    o_array[0]._fun(1);
    return 0;
}
```

Figure 1. Code of test application

Let's take a look at what the encircled code compiles into if CFG is not enabled.

```asm
mov       ecx, 3E8h
rep stosd
mov       esi, [esi]
push      1
call      esi
add       esp, 4
xor       eax, eax
```

Figure 2. Assembly code of test program

In the above figure, there is one type of indirect call. Its target address is not decided at compilation and is instead decided at runtime. An exploit can abuse this as follows:

```asm
mov       ecx, 3E8h          Pointer to fake object
rep stosd                    constructed by attacker
mov       esi, [esi]
push      1
call      esi                Call to the 1st stage
add       esp, 4             shellcode
xor       eax, eax
```

Figure 3. How to abuse the indirect call

Microsoft's implementation of CFG focuses on mitigating problems if the indirect call is exploited and an invalid target is called. (In an exploit, this would be to first stage shell code, (e.g.,stack pivot gadget).

The invalid target has a distinguishing characteristic: in most cases it is not a valid function starting address. Microsoft's CFG implementation is based on the idea that an indirect call's target must be the start of a valid function. What is the resulting assembly code if CFG is enabled?

```
mov      ecx, 3E8h
rep stosd
mov      esi, [esi]
mov      ecx, esi          ; Target
push     1
call     @_guard_check_icall@4 ; _guard_check_icall(x)
call     esi
add      esp, 4
xor      eax, eax
```

Figure 4. Assembly code, with CFG enabled

Before the indirect call, the target address is passed to the _guard_check_icall function, which is where CFG is actually implemented. In versions of Windows without CFG support, this function does nothing. In Windows 10, which does have CFG support, it points to *ntdll!LdrpValidateUserCallTarget*. This function takes a target address as argument and does the following:

1. Access a bitmap (called *CFGBitmap*) which represents the starting location of all the functions in the process space. The status of every 8 bytes in the process space corresponds to a bit in *CFGBitmap*. If there is a function starting address in each group of 8 bytes, the corresponding bit in *CFGBitmap* is set to 1; otherwise it is set to 0. The figure below is an example of a portion of *CFGBitmap*:
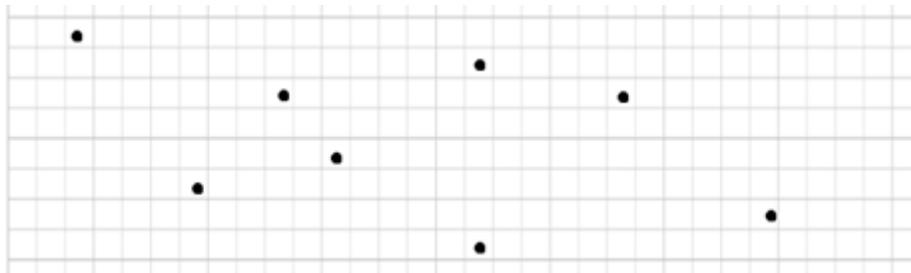
Figure 5. Representation of CFGBitmap

2. Convert the target address to one bit in *CFGBitmap*. Let's take 00b01030 as example:

Figure 6. Target address

The highest 3 bytes (the 24 bits encircled in blue) is the offset for *CFGBitmap* (unit is 4 bytes/32 bits). In this example, the highest three bytes are equal to to 0xb010. Therefore, the pointer to a four byte unit in *CFGBitmap* is the base address of *CFGBitmap* plus 0xb010.

Meanwhile, the fourth bit to the eighth bit is (the five bits encircled in red) have the value X. If target address is aligned with 0x10 (target address & 0xf == 0), then X is the bit offset value within the unit. If the target address is not aligned with 0x10 (target address & 0xf != 0), the X | 0x1 is the bit offset value.

In this example, the target address is 0x00b01030. X has the value of six. The formula *0x00b01030 & 0xf* has a result of zero; this means the bit offset is also six.

3. We look at the bit identified in Step 2. If the bit is equal to 1, it means the indirect call target is valid because it is a function's starting address. If the bit is 0, it means the indirect call target is invalid because it is not a function's starting address. If the indirect call target is valid, the function will do nothing and let it go. If the indirect call target is invalid, an exception will be raised which should prevent further exploit code from running.

00000100 00000000 00000000 01000000

Figure 7. Value in *CFGBitmap*

The value X is from the 4th bit to the 8th bit is (five bits in red circle). If the target address is aligned with 0x10 (target address & 0xf == 0), the X is the bit offset value in the unit. If the target address is not aligned with 0x10 (target address & 0xf != 0), the X | 0x1 is the bit offset value in the above 4 bytes. In this example, the target address is 0x00b01030, with X as 6 (red circle part in Figure 6).   *0x00b01030 & 0xf ==0*, so the bit offset is 6.

At Step 2, the bit offset is 6. Taking the Figure 7 as the example, the sixth bit (in red circle) is 1. It's mean the indirect call target is a valid function address.

Now, we have got a general idea of the workings of CFG. But this still raises the following questions:

1. Where is the bit information of *CFGBitmap* from?

2. When and how is the *CFGBitmap* generated?

3. How does the system handle the exception, which is caused by invalid indirect call target?

Looking into CFG Implementation

We can find additional CFG information in the PE file, which is compiled by VS2015 with CFG enabled. Let's take a look at the information of the PE file whose code can be seen in Figure.1. The information is dumped by VS2015's *dumpbin.exe*. In the PE file's *Load Config Table* part, we can find following content:

```
004020D4 Guard CF address of check-function pointer
00000000 Reserved
00402100 Guard CF function table
      11 Guard CF function count
00003500 Guard Flags
         CF Instrumented
         FID table present
         Protect delayload IAT
         Delayload IAT in its own section
```

Figure 8. PE information

- Guard CF address of check-function pointer: the address of *_guard_check_icall* (seen in Figure.4). On the Windows 10 preview, when the PE file is loaded, *_guard_check_icall* will be modified and point to *nt!LdrpValidateUserCallTarget*.

- Guard CF function table: pointer to list of functions' relative virtual address (RVA), which the application's code contains. Every function RVA will be converted to a "1" bit in the *CFGBitmap*. In other words, the *CFGBitmap*'s bit information will come from the Guard CF function table.

- Guard CF function count: the list count of function's RVA.

- CF Instrumented: indicates CFG is enabled for this application.

Here, the compiler does his entire job for CFG. What remains is the OS' support allowing the CFG mechanism to work.

1. In the OS boot phase, the first called CFG-related function is *MiInitializeCfg*. The process is system. The call stack is as follows:

```
# ChildEBP RetAddr  Args to Child
00 801a36b0 81b5713a 808fb378 808fb378 00000001 nt!MiInitializeCfg (FPO: [0,3,4])
01 801a3728 81b5706c 808fb378 c00000bb 00000000 nt!MiInitSystem+0xb4 (FPO: [0,25,4])
02 801a3738 81b51bf8 00000000 80263540 00000000 nt!MmInitSystem+0x66 (FPO: [0,0,4])
03 801a3c24 81978f66 00000000 801a3c70 81669583 nt!Phase1InitializationDiscard+0x6cc (FPO: [
04 801a3c30 81669583 808fb378 c6bf9ac6 00000000 nt!Phase1Initialization+0xe (FPO: [1,0,4])
05 801a3c70 8171f9b5 81978f58 808fb378 00000000 nt!PspSystemThreadStartup+0x5b (FPO: [SEH])
06 801a3c7c 00000000 00000000 ffffffff 000000ff nt!KiThreadStartup+0x15
```

Figure 9. Call stack

The primary job of the function *MiInitializeCfg* is to create shared memory to contain *CFG Bitmap*. The calling timing can be found in the NT kernel phase 1

initialization's memory manager component initialization (*MmInitSystem*). As you know, during the initialization of NT kernel phase 1, it will call *MmInitSystem* twice. The first *MmInitSystem* call will go to *MiInitializeCfg*. What will *MiInitializeCfg* do?
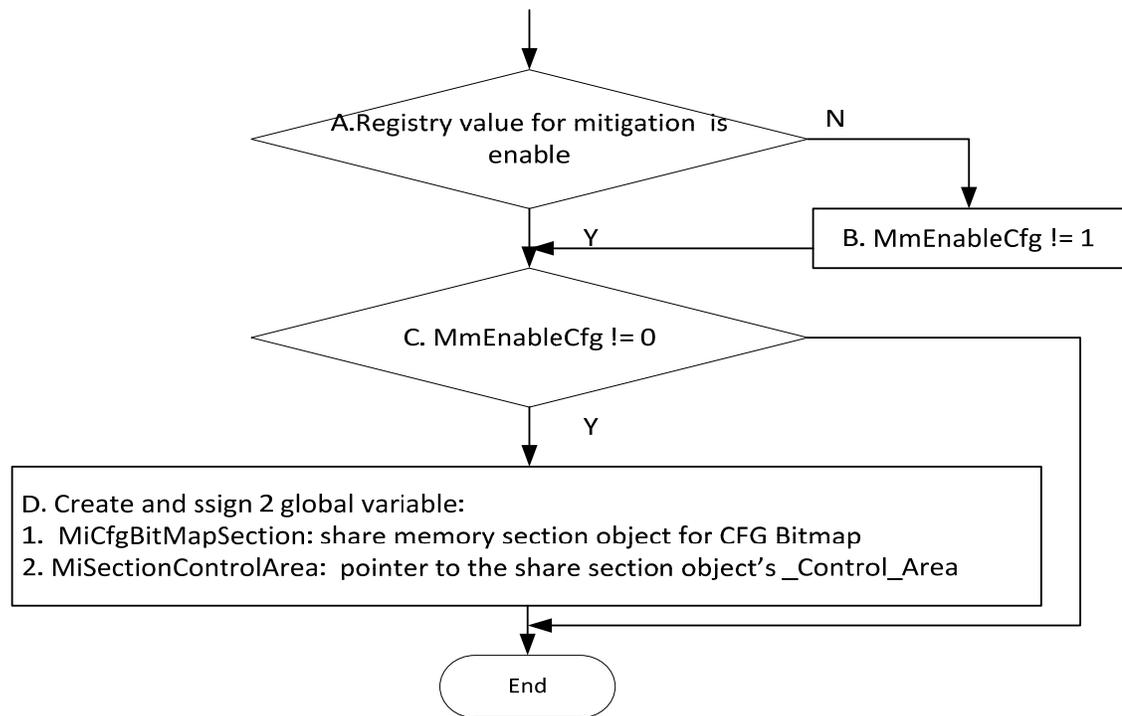

Figure 10. Function's main logic

Step A: the registry value is from *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\kernel: MitigationOptions*

Step B: *MmEnableCfg* is a global variable that is used to indicate whether the system enables CFG function

Step C: *MiCfgBitMapSection*'s DesiredAccess allows all access; its allocation type is "reserve." The shared memory size is different in build 10.0.9926 from build 6.4.9841. For build 6.4.9841, it is calculated from user mode space size. The detail formula is size = User Mode Space Size >> 6. (>> X : right shift X bits). For build 10.0.9926, the size is 0x3000000. It means the CFG bitmap can represent the entire

user mode space. The *MiCfgBitMapSection* is a core component in the CFG implementation, because it is used to contain *CFGbitmap*.

2. Get the functions compressed RVA list information and save to the image's *Control_Area* structure.

A PE image loaded into the system the first time. The NT kernel will call *MiRelocateImage* to relocate. *MiRelocateImage* will call *MiParseImageCfgBits*. In the function *MiParseImageCfgBits*, the PE image's compressed RVA list is calculated and saved into the PE image section's *Control_Area* data structure. This only happens one time for one PE image during one system booting.

When the PE is once again loaded into a process, the NT kernel will call *MiRelocateImageAgain*. Because its compressed RVA list is already saved (and does *not* need to be calculated again), *MiRelocateImageAgain* doesn't need call *MiParseImageCfgBits*, saving some processing time. *MiParseImageCfgBits* is used to calculate the compressed RVA list for the PE image in order to save the RAV list within a small space. Microsoft takes the time and space performance into consideration for CFG implementation. In the *MiRelocateImage*, its CFG-related part can be describe simply as following:

```
MiRelocateImage (X,X,X,X,X)
{
…
MiParseImageCfgBits();

//Save compressed function's RVA list into the image section's Control_Area structure
[//for 6.4.9841 build
_Control_Area ->SeImageStub->[+4]->[+24h] = Compressed function's RVA list
//for 10.0.9926 build
_Control_Area ->SeImageStub->[+0]->[+24h] = Compressed function's RVA list

…
MiSelectImageBase();    //It is ALSR core implement function
…
MiUpdateCfgSystemWideBitmap();
…


…
```

```
// modify the image's relocate information reference to the new loaded base
…
}
```

MiParseImageCfgBits is used to calculate the compressed RVA list from the module which is compiled with CFG enabled. Before diving into the function, we will look at the context of calling this function. The function *MiParseImageCfgBits* will be called in *MiRelocateImage* function.

The function *MiParseImageCfgBits* has five arguments:

a. Pointer to the image section's *Control_Area* structure

b. Pointer to the image file content

c. The image size

d. Pointer to a structure which contain part of PE Optional Header

e. Output pointer to a compressed CFG function RVA list

The main jobs of *MiParseImageCfgBits* are the following:

a. Get the function RVA list from the image's "Load Config Table" part , which I have described in the previous part

b. Use the compression algorithm to compress the list in order to save this list with small space

c. Create the compressed RVA list as its output

3. After the *CFGBitmap* shared memory section object is created, the *CFGBitmap* shared memory section object will be mapped for two uses:

a. For writing bits for shared module (.DLL files, etc.). This mapping is temporary; after the bits writing is finished, the mapping will be released. The bits information written by this mapping is shared, meaning it can be read from all processes on this system. The mapping happens in the *MiUpdateCfgSystemWideBitmap* function. The call stack is as follows:

```
 # ChildEBP RetAddr  Args to Child
00 9c7d973c 81d5e4fd cc26fa40 98daf893 cc19dd60 nt!MiUpdateCfgSystemWideBitmap (FPO: [1,14,4])
01 9c7d97fc 81d73678 00000124 00000000 00000000 nt!MiRelocateImage+0x25d (FPO: [SEH])
02 9c7d98a8 81cf6d35 00000002 00000000 9c7d99f0 nt!MiCreateNewSection+0x1cc (FPO: [6,35,4])
03 9c7d99d8 81cf6438 00000000 9c7d9a48 00000002 nt!MiCreateSection+0x8e5 (FPO: [8,65,4])
04 9c7d9a08 81cf62cb 9c7d9a8c 00000005 9c7d9aa4 nt!MmCreateSection+0x58 (FPO: [8,2,0])
05 9c7d9a74 81d5899e 9c7d9ae0 00000005 9c7d9aa4 nt!NtCreateSection+0x14b (FPO: [SEH])
06 9c7d9aec 81d58535 9c7d9b7c c8d0e490 00000003 nt!PfSnGetSectionObject+0x158 (FPO: [7,18,4])
07 9c7d9b9c 81ae209f 9c7d9c30 81ae2123 aec9d2f0 nt!PfSnPopulateReadList+0x2e5 (FPO: [1,33,4])
08 9c7d9c30 81ac3583 81c6c1a0 98daf31f 00000000 nt!ExpWorkerThread+0x1bf (FPO: [1,29,4])
09 9c7d9c70 81b799b5 81ae1ee0 81c6c1a0 00000000 nt!PspSystemThreadStartup+0x5b (FPO: [SEH])
0a 9c7d9c7c 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x15
```

Figure 11. Call stack for writing bits for shared module

b. For writing private bits and reading bits for checking indirect call target.
The bits written by this mapping is private, which means it can only be read
within the current process. This mapping's life cycle is as long as the
process' life cycle. The mapping occurs in the function *MiCfgInitializeProcess*.
The call stack is as follows:

```
 # ChildEBP RetAddr  Args to Child
00 a0b10240 81d574eb 00000000 00000000 b3bbbcc0 nt!MiCfgInitializeProcess (FPO: [0,7,4])
01 a0b10278 81d56df8 00c10000 00000000 00000000 nt!MiMapProcessExecutable+0x131 (FPO: [2,9,4])
02 a0b102f0 81d4b519 c42911b8 a0b104c0 00000000 nt!MmInitializeProcessAddressSpace+0x136 (FPO: [3,23,4])
03 a0b104a0 81d48fae 00000000 00000000 00000000 nt!PspAllocateProcess+0x839 (FPO: [SEH])
04 a0b10be0 81b74fe7 00e7ed84 00e7ed38 02000000 nt!NtCreateUserProcess+0x444 (FPO: [SEH])
05 a0b10be0 7786e240 00e7ed84 00e7ed38 02000000 nt!KiSystemServicePostCall (FPO: [0,3] TrapFrame @ a0b10c14)
```

Figure 12. Call stack for writing private bits and reading bits

Based on the call stack, we know that it is mapped at an initializing process.
The mapped size is different for builds 10.0.9926 and 6.4.9841. For build
6.4.9841, the size is calculated based onuser mode space size. The detail
formula is size = User Mode Space Size >> 6. (>> X : right shift X bits). For
build 10.0.9926, the size is 0x3000000. The mapped space always exists during
the process' life cycle. The mapped base address and length will be saved to a
global structure which type is *MI_CFG_BITMAP_INFO* and address is fixed
(For build 6.4.9841, the base address is 0xC0802144. For build 10.0.9926, the
base address is 0xC080214C). I will discuss how we write private bits into the
mapped space later on. Below is the structure of *MI_CFG_BITMAP_INFO*:

```
{
  Void* BaseAddress,          //mapped base address in current process
  UINT32 RegionSize,          //mapped length
  void* VadBaseAddress,       //the VAD's base address
  _MMVAD*  BitmapVad          //the VAD for the mapped address
}
```

4. Once the PE image RVA list is ready and the *CFGBitmap* section is mapped, it's time to translate the RVA list into bits in the *CFGBitmap*.
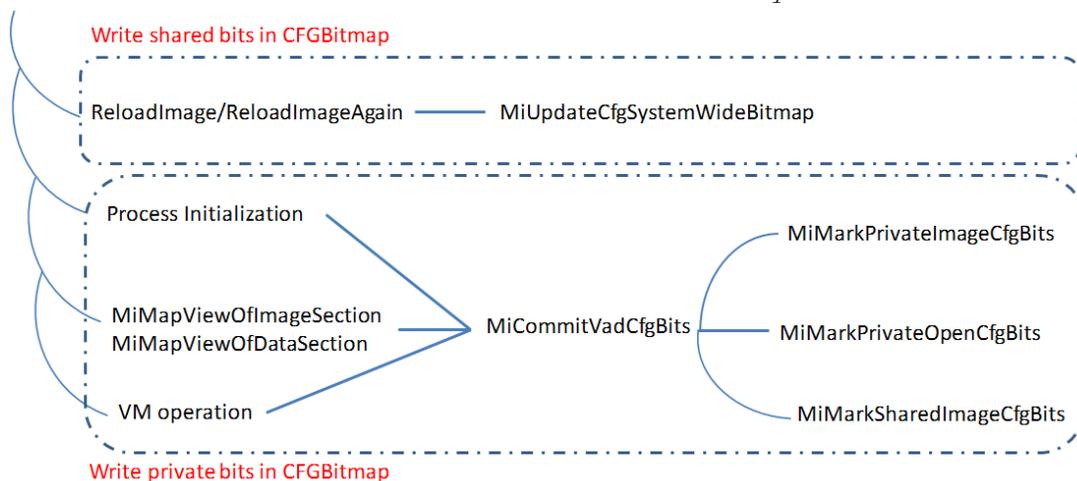

Figure 13. Updating bits into CFGBitmap

The progress differs in several scenarios:

- In the *ReloadImage/ReloadImageAgain*, writing bits for shared module for shared module (etc. dll) by MiUpdateCfgSystemWideBitmap

- Writing bits for private module (.EXE files, etc.) in the process initialization phase

- Writing bits for VM (virtual memory) operation

- Writing bits for view mapping for image and data section

Before we take a closer look at each scenario, we need to make clear some background information. In each process view, the space which contains *CFGBitmap* can be divided into two parts: shared and private.

MiCfgBitMapSection is a shared memory section object that contains *CFGBitmap*'s shared bitmap content. It is shared with every process. Each process sees the same content in the shared section when it maps *MiCfgBitMapSection* in its process virtual memory space. The shared module (.DLL files, etc.) bitmap information will be written by the mapping method described in Section 3.a.

However, each process requires a part in the *CFGBitmap* that is not shared among all processes. It needs to write some of the module's bitmap information into the *CFGBitmap* as private. The private part will not be shared to all processes. The EXE module's bitmap information will be written using the mapping method in Section 3.b. The figure below shows a common scenario.
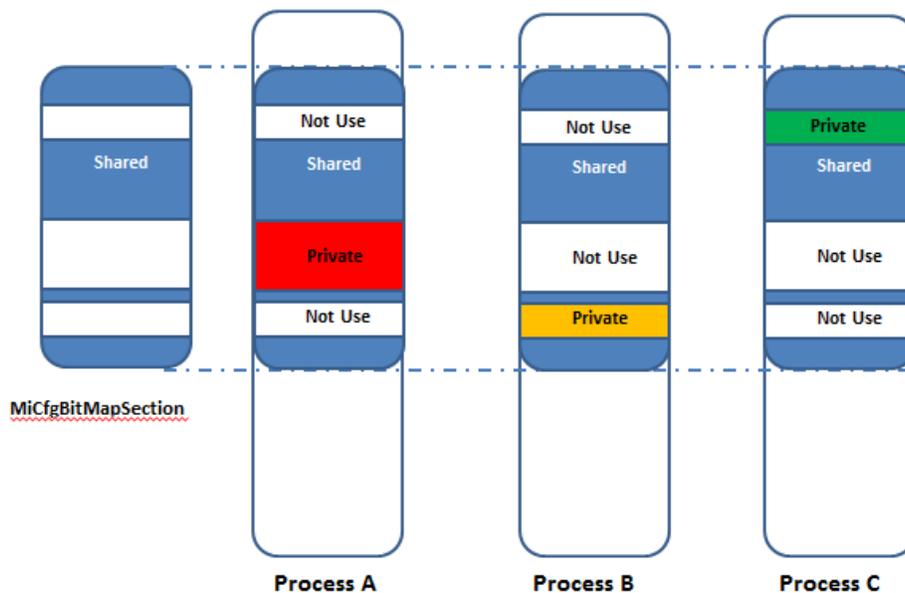

Figure 14. Three processes with the shared part bitmap content in the
MiCfgBitMapSection and their private sections

a. In the *ReloadImage/ReloadImageAgain*, writing bits for shared module for shared module (.DLL files, etc.) by *MiUpdateCfgSystemWideBitmap*.

As seen in Section 2, after getting the image's function RVA compressed list and saving it to the image's *Control_Area* data structure (In build 6.4. 9841 : _Control_Area ->SeImageStub->[+4]->[+24h]; in build 10.0.9926: _Control_Area ->SeImageStub->[+0]->[+24h]), it will call *MiSelectImageBase*. The function is the core function of ASLR implementation. It returns the final selected base address. The selected base address is very important for writing bit information into *CFGBitmap*. After getting the final decided base address, it will call *MiUpdateCfgSystemWideBitmap*.

The main task of *MiUpdateCfgSystemWideBitmap* is to translate compressed RVA list to "1" bit in the *CFGBitmap*. The bitmap content which is written by this

function is shared and will be shared by all processes in the system. The function only works for shared module (.DLL files, etc.).

MiUpdateCfgSystemWideBitmap takes 3 arguments:

- o Pointer to the image's *Control_Area* structure.

- o The image's selected base address

- o Pointer to compressed RVA list

*MiUpdateCfgSystemWideBitmap*'s main logic is as follows:
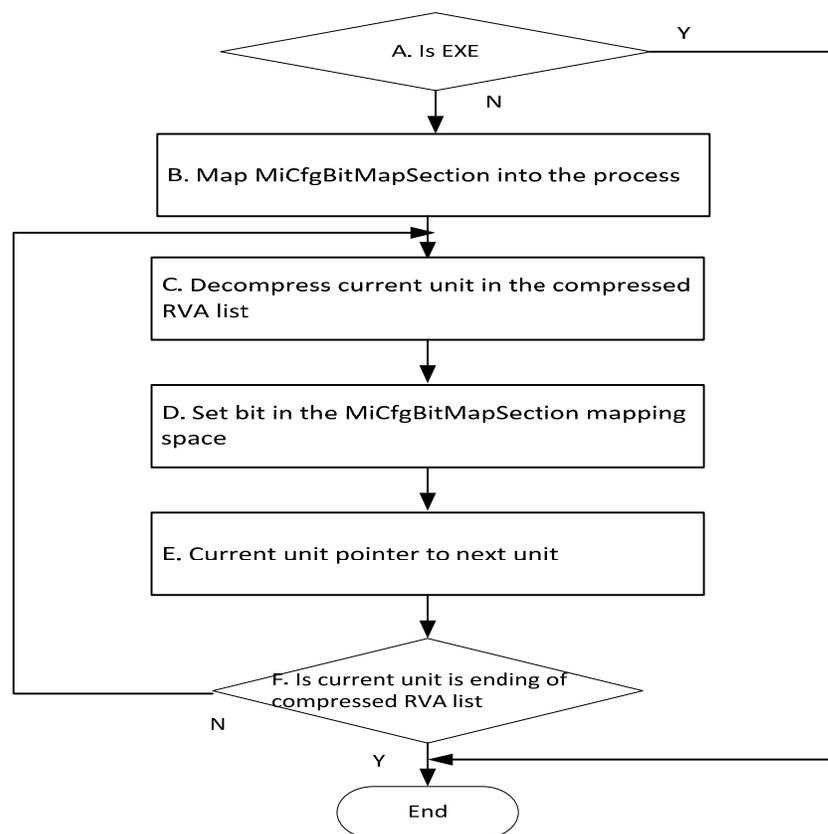
Figure 15. Main logic of MiUpdateCfgSystemWideBitmap

In Step B, it maps the *CFGBitmap* shared memory into the system process space. It doesn't map all of the shared memory's total size. It converts the selected image's base address to *CFGBitmap*'s offset and use the converted result as the starting offset of the mapping. The conversion is as follows:

Offset in bitmap = Image selected base address >> 6. In like manner, the mapping size is the image size >> 6.
The function also is called when the image need to be relocated again (ReloadImageAgain function).

b. Writing bits for private modules (.EXE files, etc.) in the process initialization phase. It calls the function *MiCommitVadCfgBits*, which is a dispatcher function. You may use Figure 13 as a reference. It is called in certain scenarios. The function's primary job is to write bits for the space that is described by the input Virtual Address Descriptor (VAD). The main logic is as follows:
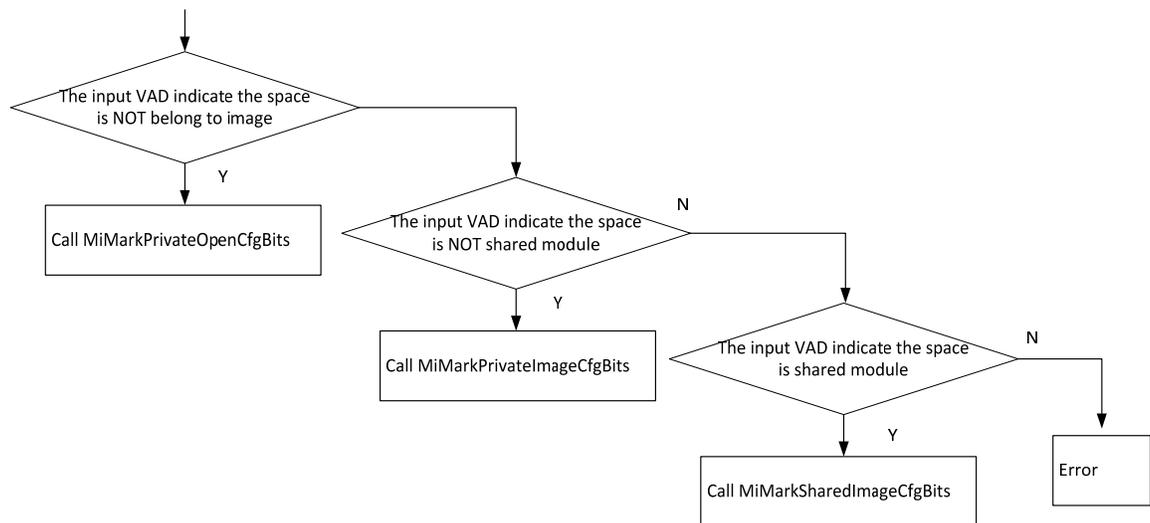
Figure 16. *MiMarkPrivateImageCfgBits* function handle for writing bits for private module

The function *MiMarkPrivateImageCfgBits* implements writing bit information into CFG Bitmap for private modules (EXE files, etc.). When the system maps a view of the EXE image section or starts a process, the function is being called.

The function takes two arguments:

1. Address of global variable for CFGBitmap information

2. The VAD for the image space

The VAD (Virtual Address Descriptor) is a structure which is used to describe a range of virtual memory space.

The function's primary job is to convert the input VAD's related compressed RVA list to the bitmap information, and write bits privately in the *CFGBitmap*. The main logic is as follows:
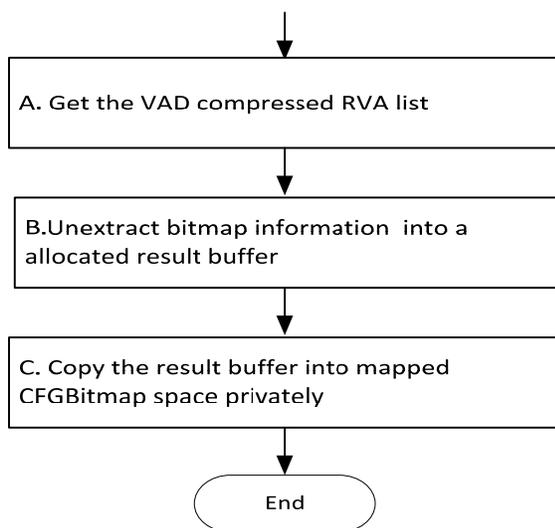

Figure 17. Main logic for *MiMarkPrivateImageCfgBits*

At step A, the related compressed RVA list can be retrieved from the input VAD's related *Control_Area* structure, which is saved in the *MiRelocateImage* (see Section 2).

This function's main step is Step C. It implements writing privately the *MiCfgBitMapSection32* section mapped space, which I described in Section 3.b. The mapping for write private bits is read-only. How do we write bits into the mapped space? Its key steps are the following:

i. Get the target mapped space address's physical address (PFN: Physical Frame Number)

ii. Apply a blank Page Table Entry (PTE) and fill the PTE with the physical address, which was acquired from the previous step. The new PTE will be mapped to the same physical page that includes the target *MiCfgBitMapSection32* mapped virtual address.

    iii. Copy the resulting buffer (Figure 12) into the new PTE represent virtual address. The physical page thus contains the resulting buffer content.

    iv. Release the new PTE.

After the steps mentioned above, the bitmap information is copied to the current process virtual memory space. But this will have no impact on *MiCfgBitMapSection*. In other words, *MiCfgBitMapSection* will not know the bitmap is changed. Other processes will not see changes; the newly added bitmap information is private for the current process.

c. Writing bits for virtual memory (VM) operations. If a process has a virtual memory operation, it may impact their bits status in *CFGBitmap*'s bitmap. From the scenario in Figure 13, it will call *MiMarkPrivateOpenCfgBits*. The function's primary job is to copy pages full of "1" or "0" to the *CFGBitmap* space privately.

    i. For the *NtAllocVirtualMemory* function

If a process calls the *NtAllocVirtualMemory* function to allocate virtual memory with executable attributes, the NT kernel will set all relative bit to "1" in *CFGBitmap* privately. But if the allocated memory's protect mask has *SEC_WRITECOMBINE*, the NT kernel will use "0" to set the bitmap.

    ii. For the *MiProtectVirtualMemory* function

If a process calls *MiProtectVirtualMemory* to change the range of virtual memory's protection to "executable," the NT kernel will set all relative bit to "1" in *CFGBitmap* privately.

d. Writing bits for view mapping for image and data section.

    i. For mapping view for image section (.DLL, .EXE, etc.), if the image not shared, the handling process is that of Section 4.b. If the image is shared, it will be handled by the *MiMarkSharedImageCfgBits* function (Figure 13). It traverses each page in the mapping space and converts the page address to the offset in the *CFGBitmap*.

i. If the offset in the *CFGBitmap* is not backed by *PrototypePTE*, the related bits information will be copied onto the *CFGBitmap* space privately.

ii. If the offset in the *CFGBitmap* already has bitmap information, the part of the *CFGBitmap* will be changed to *read-only*.

ii. For mapping view for the data section, the handling is the same as in Section 4.c.i.

5. The steps mentioned above occur in the kernel mode. But for the user mode, the *CFGBitmap* needs access to the *LdrpValidateUserCallTarget* function, which I described in the previous portion. How will the user mode be made aware of the *CFGBitmap* mapped address? When creating a process, the NT kernel calls the *PspPrepareSystemDllInitBlock* function to write the *CFGBitmap* mapped address and length to a global variable's fields, whose data structure is the *PspSystemDllInitBlock* structure. The *PspSystemDllInitBlock* is the fixed address and can be accessed from both the user mode and kernel mode code.

```
 # ChildEBP RetAddr  Args to Child
00 b9c5c5ec 81ce4ead 00000000 b0e87a80 000002aa nt!PspPrepareSystemDllInitBlock (FPO: [SEH])
01 b9c5c634 81ceb689 b9c5c794 b9c5cb40 9ba378bc nt!PspSetupUserProcessAddressSpace+0x10d (FPO: [2,13,4])
02 b9c5c7e0 81ce8fae 00000000 00000000 00000000 nt!PspAllocateProcess+0x9a9 (FPO: [SEH])
03 b9c5cf20 81b14fe7 000bec6c 000bec20 02000000 nt!NtCreateUserProcess+0x444 (FPO: [SEH])
04 b9c5cf20 774fe240 000bec6c 000bec20 02000000 nt!KiSystemServicePostCall (FPO: [0,3] TrapFrame @ b9c5cf54)
```

Figure 18. Call stack

The user mode code can access the *PspSystemDllInitBlock* global variable's *CFGBitmap* field by hard code.

6. In Figure 4, the *_guard_check_icall* function pointer will point to *LdrpValidateUserCallTarget* in *ntdll.dll*. When and how does this occur? The *LdrpCfgProcessLoadConfig* function performs this job. The process creating progress will call *LdrpCfgProcessLoadConfig* in user mode.

```
 # ChildEBP RetAddr  Args to Child
00 007df6d4 774c95b1 7748b308 7f1d4000 008e1680 ntdll!LdrpCfgProcessLoadConfig (FPO: [Non-Fpo])
01 007df700 77531f85 6cdf12c7 7f1d4000 00000000 ntdll!LdrpProcessMappedModule+0x117 (FPO: [0,5,4])
02 007df904 774cd8fb 6cdf1297 00000000 00000000 ntdll!LdrpInitializeProcess+0xdaf (FPO: [SEH])
03 007df954 774cd9a0 00000000 00000000 007df970 ntdll!_LdrpInitialize+0xa0 (FPO: [Non-Fpo])
04 007df95c 00000000 007df970 77480000 00000000 ntdll!LdrInitializeThunk+0x10 (FPO: [2,0,0])
```

Figure 19. In this function, it will call modify *_guard_check_icall* 's value to pointer to *LdrpValidateUserCallTarget*

7. After all preparations are set, if the indirect call's target address's related bit is not "1" in the *CFGBitmap*, it violates CFG. The process will take action to handle the violation. The handle function is *RtlpHandleInvalidUserCallTarget*. The function takes the indirect call target as the only argument. The function's main logic is as follows:
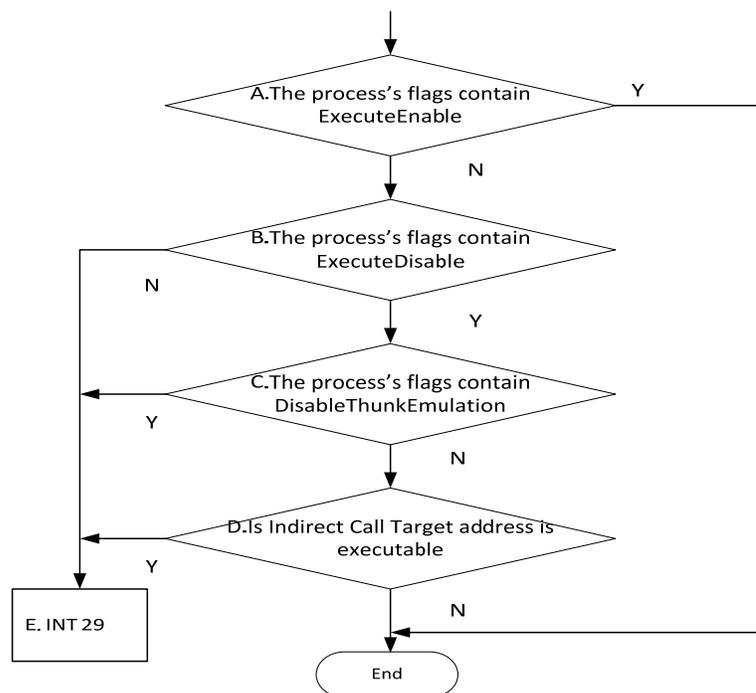

Figure 20. Main logic of *RtlpHandleInvalidUserCallTarget*

The function's main job is to check the Data Execution Prevention (DEP) status and raise *interrupt 29*, the kernel interrupt handle routine *KiRaiseSecurityCheckFailure*. Its behavior is stopping the process.

If an indirect call target address's related bit location in the *CFGBitmap* cannot be accessed (e.g., out of range of *CFGBitmap* space), it means that the target address is invalid. The system will throw the access violation exception. When the exception goes back to the user mode handle function *KiUserExceptionDispatcher*, it will call *RTLDispatchExeption*. In *RTLDispatchExeption*, it will check the exception's happen address. If the address is for instruction to access *CFGBitmap*, it will go to *RtlpHandleInvalidUserCallTarget* too.

8. If a process needs to customize its *CFGBitmap*, it can call *NtSetInformationVirtuaMemory* in *ntdll.dll*. The kernel implements the feature in the function *MiCfgMarkValidEntries*. *MiCfgMarkValidEntries* take a buffer and its length. Every unit in the buffer is eight bytes. The first four bytes is the target address, which wants to set the related bit in *CFGBitmap*, and the last four bytes is the flag to set the bit to "0" or "1." *MiCfgMarkValidEntries* customizes the *CFGBitmap* privately, which is only seen by the current process.

9. If an attacker needs to change the *CFGBitmap* content from the user mode code, it could be impossible. This is because the *CFGBitmap* is mapped *read-only*, which was discussed in Section 3.b. Either the changing the space protection or writing value to the space will fail.

Weaknesses of CFG

Of course, this mechanism is not without some weak points. We have outlined some of the weaknesses of CFG.

- The *CFGBitmap* space's base address is stored in a fixed addres,s which can be retrieved from user mode code. This was described in the implementation of CFG. This is important, security data but however, it can be easily gotten.

- If the main executable is not enabled for CFG, the process is not protected by CFG even if it loaded a CFG-enabled module.

- Based on Figure 20, if a process's main executable has disabled DEP (the process's *ExecuteEnable* is enabled by compiled with */NXCOMPAT:NO*), it will bypass the CFG violation handle, even if the indirect call target address is invalid.

- Every bit in the CFGBitmap represents eight bytes in the process space. So if an invalid target call address has less than eight bytes from the valid function address, the CFG will think the target call address is "valid."

- If the target function generated is dynamic (similar to JIT technology), the CFG implement doesn't protect it. This is because *NtAllocVirtualMemory* will

set all "1" in *CFGBitmap* for allocated executable virtual memory space (described in 4.c.i). It's possible that customizing the *CFGBitmap* via *MiCfgMarkValidEntries* can address this issue.

TREND MICRO™

225 E. John Carpenter Freeway
Suite 1500
Irving, Texas
75062 U.S.A.

Phone: +1.817.569,8900