

Windows 10 Control Flow Guard Internals

MJ0011

Agenda

- Introduction to Control Flow Guard
- How CFG Works: User Mode Part
- How CFG Works: Kernel Mode Part
- The Weakness of CFG

Intro to Control Flow Guard

- New security mitigation introduced in Windows 8.1 Preview
 - Then disabled in Windows 8.1 RTM because of compatibility issues
- Re-enabled in Windows 10 Technical Preview
 - With some minor changes
- An imperfect implementation of Control-Flow Integrity (CFI)
 - Prevent exploits which attempts to subvert machine code execution

Control-Flow Integrity

- “Control-Flow Integrity - Principles, Implementations, and Applications”
 - <http://research.microsoft.com/pubs/69217/ccs05-cfi.pdf>
- “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”
 - <http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/34913.pdf>
- “Practical Control Flow Integrity & Randomization for Binary Executables”
 - <http://www.cs.berkeley.edu/~dawnsong/papers/Oakland2013-CCFIR-CR.pdf>

Control Flow Guard

- CFG prevents untrusted indirect call
 - Also called “icall guard” in project code
- It relies on compile and link level processing on binary
 - Enforce additional calls target check before each indirect calls in machine code
- Windows adds some kernel mechanisms to improve its performance
 - Build shared function bitmap table into protected process

How CFG Works: User Mode Part

- New load config structure
- Initialize SystemDllInitBlock and load config
- Function bitmap layout and target validation logics
- Add CFG exception

New Load Config Structure

```
dd offset __security_cookie ; SecurityCookie
dd offset __safe_se_handler_table ; SEHandlerTable
dd 47h ; SEHandlerCount
dd offset __guard_check_icall_fptr ; GuardCFCheckFunctionPointer
dd 0 ; Reserved2
dd offset __guard_fids_table ; GuardCFFunctionTable
dd 1929 ; GuardCFFunctionCount
dd 3500h ; GuardFlags
```

- New load config structure adds 5 new fields
- Including key data for CFG which is generated in build processing
 - CFG check function pointer(point to null subroutine)
 - CFG function table(used by NT kernel)
 - CFG flags

Init LdrSystemDllInitBlock and Load Config

- Initialize LdrSystemDllInitBlock
 - +0x60 : Bitmap Address
 - +0x68 : Bitmap Size
 - Initialized by PspPrepareSystemDllInitBlock
 - NtCreateUserProcess->PspAllocateProcess->PspSetupUserProcessAddressSpace
- LdrpCfgProcessLoadConfig
 - Check PE Headers->OptionalHeader.DllCharacteristics
 - IMAGE_DLLCHARACTERISTICS_GUARD_CF flag
 - Set LoadConfig->GuardCFCheckFunctionPointer
 - LdrpValidateUserCallTarget

Call Target Validation Logics

- LdrpValidateUserCallTarget

```
    mov     edx, dword ptr ds:GuardCFBitMapAddress
    mov     eax, ecx
    shr     eax, 8
    mov     edx, [edx+eax*4]
    mov     eax, ecx
    shr     eax, 3
    test    cl, 0Fh
    jnz     short not_aligned_adress
    bt      edx, eax
    jnb     short invalid_target
    retn

not_aligned_adress
    or      eax, 1
    bt      edx, eax
    jnb     short invalid_target
    retn
```

- It only executes 10 instructions in most cases

Call Target Validation Logics

- LdrpValidateUserCallTarget

```
mov     edx, dword ptr ds:GuardCFBitMapAddress  
mov     eax, ecx  
shr     eax, 8  
mov     edx, [edx+eax*4]
```



```
mov     eax, ecx  
shr     eax, 3  
test    cl, 0Fh  
jnz     short not_aligned_adress  
bt      edx, eax  
jnb     short invalid_target  
retn
```

```
not_aligned_adress  
or      eax, 1  
bt      edx, eax  
jnb     short invalid_target  
retn
```

- Use (Address / 0x100) as index to get 32 bits from function bitmap
 - So one bit in function bitmap will indicates 8 bytes address range

Call Target Validation Logics

- LdrpValidateUserCallTarget

```
mov     edx, dword ptr ds:GuardCFBitMapAddress
mov     eax, ecx
shr     eax, 8
mov     edx, [edx+eax*4]
mov     eax, ecx
shr     eax, 3
test    cl, 0Fh
jnz     short not_aligned_adress
bt     edx, eax
jnb     short invalid_target
retn

not_aligned_adress
or     eax, 1
bt     edx, eax
jnb     short invalid_target
retn
```

- Clean low 3 bits of address and use bit3~bit7 as index in 32 bits bitmap
 - So address must at least aligned to 0x8

Call Target Validation Logics

- LdrpValidateUserCallTarget

```
mov     edx, dword ptr ds:GuardCFBitMapAddress
mov     eax, ecx
shr     eax, 8
mov     edx, [edx+eax*4]
mov     eax, ecx
shr     eax, 3
test    cl, 0Fh
jnz     short not_aligned_adress
bt     edx, eax
jnb     short invalid_target
retn
```

```
not_aligned_adress
or     eax, 1
bt     edx, eax
jnb     short invalid_target
retn
```

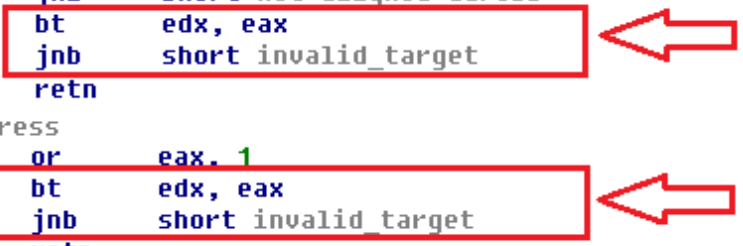
- Actually in most cases valid call target is aligned to 0x10
 - Address which is not aligned to 0x10 will always use odd bit
 - In most cases there are only half bits used in bitmap

Call Target Validation Logics

- LdrpValidateUserCallTarget

```
mov     edx, dword ptr ds:GuardCFBitMapAddress
mov     eax, ecx
shr     eax, 8
mov     edx, [edx+eax*4]
mov     eax, ecx
shr     eax, 3
test    cl, 0Fh
jnz     short not_aligned_adress
bt     edx, eax
jnb     short invalid_target
retn

not_aligned_adress
or     eax, 1
bt     edx, eax
jnb     short invalid_target
retn
```



- Finally, bit test to see if there is a valid function at this location

Function Bitmap Layout

- Guard function bitmap is mapping into every protected process

Address	Type	Size	Commi...	Private	Total WS	Priva...	Share...	Sha...	Loc...	Blocks	Protection
01130000	Private Data	4 K	4 K	4 K						1	Read/Write
01140000	Heap (Private...	16 K	4 K	4 K	4 K	4 K				2	Read/Write
01150000	Shareable	4 K	4 K		4 K		4 K	4 K		1	Read
01160000	Image	44 K	44 K	12 K	12 K		12 K	12 K		6	Execute/Read
01170000	Shareable	32,768 K	7,268 K		376 K		376 K	352 K		82	Read
01170000	Shareable	276 K				4 K					Reserved
011B5000	Shareable	4 K	4 K			4 K		4 K			Read
011B6000	Shareable	23,000 K									Reserved
0282C000	Shareable	3,860 K	3,860 K								No access
02BF1000	Shareable	12 K	12 K			8 K		8 K	8 K		Read
02BF4000	Shareable	864 K	864 K								No access
02CCC000	Shareable	4 K	4 K			4 K		4 K	4 K		Read
02CCD000	Shareable	176 K	176 K								No access
02CF9000	Shareable	20 K	20 K		12 K		12 K	12 K			Read
02CFE000	Shareable	24 K	24 K								No access



Guard Function Bitmap

- Every bit in the bitmap indicates 8 bytes in address space
 - Bitmap size = $\text{HighestUserAddress} / 8 / 8 = 0x80000000 / 0x40 = 0x20000000$
 - It will use 32MB user address space and about 7MB are committed(non-3GB Mode)
 - There are about only 200~300KB remaining in working set (physical memory)
 - Bitmap is mapped into every process and shared with each other

Unmapped Bitmap Processing

- RtlDispatchException adds a mechanism to process the case when call target validation tries to access unmapped bitmap area
 - When exception raised and dispatched to user mode exception handler
 - KiUserExceptionDispatcher-> RtlDispatchException
 - It will check whether Eip is the instruction inside LdrpValidateUserCallTarget
 - Then it will call RtlpHandleInvalidUserCallTarget to avoid invalid call
 - This is why LdrpValidateUserCallTarget doesn't need its own exception handler

Add CFG Exception

- CFG allows user process to add some exceptions for compatibility
- `Kernelbase!SetProcessValidCallTargets`
- It will call `NtSetInformationVirtualMemory->MiCfgMarkValidEntries` to add valid call bits into bitmap

How CFG Works: Kernel Mode Part

- CFG Initialization in Booting Process
- CFG Bitmap Mapping in Process Creation Process
- CFG Bitmap Building in Image Loading Process
- Shared Bitmap VS. Private Bitmap

Booting Process

- **MiInitializeCfg**
 - Check PspSystemMitigationOptions from CCS\Session Manager\Kernel: MitigationOptions
 - Calculate CFG bitmap section size using MmSystemRangeStart
 - Create CFG bitmap section object(MiCfgBitMapSection32)

Process Creation Process

- PspApplyMitigationOptions
 - PspAllocateProcess
 - Check mitigation options and set Process->Flags.ControlFlowGuardEnabled
- MiCfgInitializeProcess
 - MmInitializeProcessAddressSpace-> MiMapProcessExecutable
 - After map system dlls , map CFG bitmap section into process
 - Reference and commit CFG VAD bits in bitmap
 - Write bitmap mapping information to hyper space
 - 0xC0802144: bitmap mapped address
 - 0xC0802148: bitmap size
 - 0xC0802150: bitmap VAD

Image Loading Process

- MiParseImageCfgBits + MiUpdateCfgSystemWideBitmap
 - MiRelocateImage/MiRelocateImageAgain
- When system relocates image, NT kernel will parse new image's guard function table and update it into bitmap
- Compress guard function RVA list and set it to global bitmap

Shared Bitmap VS. Private Bitmap

- NT Kernel will check the behaviors which try to modify mapped bitmap
 - NtAllocateVirtualMemory
 - NtMapViewOfSection(Data/Image/Physical section)
 - NtProtectVirtualMemory
- If user mode code tries to modify mapped bitmap page, kernel will mark this page into private process memory
- So that process can change bitmap locally or globally
- But so far this feature doesn't work on my VM(win10 9860) , it's always blocked when acquiring VAD's push lock☹

The Weakness of CFG

- Rely on Security of Stack Address
- Unaligned Guard Functions
- Unprotected Images and Processes

Stack Address

- If we know thread stack address, we can bypass CFG in many ways
 - Overwrite return address on the stack
 - CFG only checks indirect call target , does not validate “ret” instruction
 - Bypass some checks on trusted functions and still achieve ROP
 - Bypass some checks on trusted function to achieve our own purpose
- And stack address is not difficult to obtain 😊
- Also if you can leak some important data location, you can control program behavior indirectly

Unaligned Guard Functions

- CFG only use 32MB address space on X86 machine because of memory limit
 - One bit indicates 8bytes address and actually in most cases 16bytes
 - Every guard function address needs to be aligned to 0x10
 - If function address is not aligned to 0x10 , it will use the odd bit only
- Unaligned guard function will allow untrusted function call near the trusted function address

Unaligned Guard Functions

- Is every guard function well aligned?
 - I created a tool to parse every binary on Windows10
 - The result has many binary with unaligned guard functions!

```
C:\Windows\system32\cmd.exe
check file: c:\windows\system32\d3d8thk.dll enabled control flow guard
guard function number :62
check file: c:\windows\system32\d3d9.dll enabled control flow guard
guard function number :2364
check file: c:\windows\system32\D3DCompiler_47.dll enabled control flow guard
guard function number :2862
check file: c:\windows\system32\d3dim.dll enabled control flow guard
guard function number :1117
function not aligned to 0x10: 00046efc 74046efc
function not aligned to 0x10: 0004700f 7404700f
function not aligned to 0x10: 00047122 74047122
function not aligned to 0x10: 000475c6 740475c6
function not aligned to 0x10: 00047944 74047944
function not aligned to 0x10: 00047cc3 74047cc3
function not aligned to 0x10: 0004846d 7404846d
function not aligned to 0x10: 0004853d 7404853d
function not aligned to 0x10: 00048783 74048783
function not aligned to 0x10: 000487f2 740487f2
```

Unprotected Images and Processes

- CFG relies on compile and link level processing
- So third party modules and even old version MS binary are not protected
- If the main executable image is not made for CFG, CFG will be disabled in full process even it will load some system modules that support CFG

Summary

- CFG is a well designed and implemented mitigation.
- Performance loss and memory consumption are controlled precisely
- It will significantly raise the bar on memory bug exploitation
- Hope it will finally ship to RTM version of Windows10 😊

Acknowledgement

- Yuki Chen
- Vangelis

Question?

- Thank you!