Control Flow Guard Teleportation

Control Flow Guard (CFG) is a Windows' security feature that aims to mitigate the redirection of the execution flow, for example, by checking if the target address for an indirect call is a valid function. We can abuse this for funny obfuscation tricks.

How does CFG works?

With that example, let's compile an exe file with MSVC compiler to see what code is produced and executed before calling main():

```
call    __scrt_get_dyn_tls_init_callback
mov     esi, eax
...
mov     esi, [esi]
mov     ecx, esi
call    ds:__guard_check_icall_fptr
call    esi
```

The function *__scrt_get_dyn_tls_init_callback* gets a pointer to a TLS callback table to call the first entry. The callback's function is protected by CFG so the compiler adds code to check if the function address is valid before executing the target address in *ESI*. Let's follow the call:

```
__guard_check_icall_fptr dd offset _guard_check_icall_nop
```

```
_guard_check_icall_nop proc near
      retn
_guard_check_icall_nop endp
```

Just *RETN*. Why? So that the program can run in older OS versions that do not support CFG. In a system that does supports it the *_guard_check_icall_nop* address is replaced with *LdrpValidateUserCallTarget* from NTDLL:

```
ntdll!LdrpValidateUserCallTarget:
mov     edx,[ntdll!LdrSystemDllInitBlock+0xb0 (76fb82e8)]
mov     eax,ecx
shr     eax,8
ntdll!LdrpValidateUserCallTargetBitMapCheck:
mov     edx,[edx+eax*4]
mov     eax,ecx
shr     eax,3
```

Introducing the Bitmap

For CFG they added a bunch of new fields to the PE in the Load Config Directory: *GuardCFCheckFunctionPointer* which points to *__guard_check_icall_ptr*, the function address to replace; and *GuardCFFunctionTable*. The table contains the RVAs of all the functions to be set as valid targets. But set where? In a Bitmap that is created when loading the PE. *LdrpValidateUserCallTarget* gets the address of the Bitmap from *LdrSystemDllInitBlock+0xb0 in that first instruction.*

The Bitmap contains (2 bit) "states" for every 16 bytes in the entire process: yes, it's big. When the PE is loaded, the RVAs from the table are converted to offsets, then the state at that offset is set accordingly.

Beam me up, CFG!

My idea is to use the *GuardCFFunctionTable* to populate the Bitmap with chosen states, and regenerate our code inside it, then at the entrypoint we copy it into our image and execute it. I was able to figure out some of the states before, now thanks to Alex Ionescu's (et al) research in *Windows Internals 7th Edition* book, I completed the list, including their meaning:

00b	Invalid target
01b	Valid and aligned target
10b	Same as 01b? (See below)
11b	Valid but unaligned

Say that the first byte in our code is 0x10 (010000b), our region to transfer our code from the Bitmap begins at 0x402000 (RVA: 0x2000), just for clarity we will use that same region for our fake RVAs. To generate 0x10 we need only 1 entry in the table: 0x2020, skipping the first 32 bytes so that the states are set to **0000**b, 0x2020 sets the next state to **01**b and the Bitmap becomes **01****0000**b.

Now to get the state **11**b, say that we want the byte 0x1D (011101b), we use an unaligned RVA, the table would be: 0x2000 (sets to **01**b), 0x2012 (sets to **11**b), 0x2020 (sets to **01**b). It's easy!

To get **10**b, we need to use a special type of RVA with metadata, but it's simple, we append a byte to the RVA that we use to generate the **10**b. The metadata is a flag: IMAGE_GUARD_FLAG_FID_SUPPRESSED (1) or IMAGE_GUARD_FLAG_EXPORT_SUPPRESSED (2). So say we want to generate 0x86 (**10****00****01****10**b), we use: 0x2000 with 0x2 (sets to **10**b), 0x2010 (sets to **01**b), 0x2030 with 0x2 (sets to **10**b).

Transfer from the Bitmap

```
mov esi, 0DEADh     ;GuardCFCheckFunctionPointer points here
mov esi, [esi + 2] ;get LdrSystemDllInitBlock+0xb0 address
mov esi, [esi]      ;get the Bitmap address
mov eax, [ebx + 8] ;ebx=fs:[30h] at start time
lea edi, [eax + xxxxxxxx] ;imagebase + buffer rva
add ah, 20h         ;imagebase + 0x2000
shr eax, 8          ;shift-right 8 bits to make the offset
lea esi, [esi + eax*4] ;esi=our code in the Bitmap
mov ecx, xxxxxxxx  ;size of code
rep movsb
```

We let the loader replace the *0DEADh* with the address to *LdrpValidateUserCallTarget* from which we can get the address of the Bitmap. We calculate the offset to the region in the Bitmap (0x402000) and copy the regenerated code from it.

Bonus fun facts

So what happens when an invalid address is detected? The program is terminated with an exception. It's funny because most tools or codes that alter PE files don't support CFG: any address that you alter to execute your code somewhere else, must be in the table. This has the effect of killing many viruses that alter *AddressOfEntryPoint*, or use EntryPoint Obscuring (EPO) techniques. But if you disable CFG in the PE, you can replace *GuardCFCheckFunctionPointer* with your own address for a nice EPO technique. :-)

Outro

This was an idea of which I wrote two texts about failure and success. This article is a better explanation of it for the people who don't know it yet. Maybe now you want to look at my demo and try it: https://github.com/86hh/cfg-teleport-demo