CFG防护机制的简要分析

f0**** / 2018-08-16 13:37:52 / 浏览数 4152 安全技术 技术讨论 顶(0) 踩(0)

0．概述

本文将介绍一下Windows上的CFG防护技术。
主要会讲一个 demo，自己动手写一个有CFG保护的程序，然后跟踪调试。
接着会调试 CFG 在IE ， Edge上出现的情况。
最后提一些曾经所使用的bypass方法。

1.CFG简介

微软在Windows10和Windows8.1Update3（2014年11月发布）系统中已经默认启用了一种新的机制- Control Flow Guard （控制流防护）。
这项技术通过在间接跳转前插入校验代码，检查目标地址的有效性，进而可以阻止执行流跳转到预期之外的地点，
最终及时并有效的进行异常处理，避免引发相关的安全问题。
简单的说，就是在程序间接跳转之前，会判断这个将要跳转的地址是否是合法的。
如下：
① 没有CFG保护

```
.text:004015E9                  mov     eax, [ebp+var_8]
.text:004015EC                  mov     ecx, [eax+edx]
.text:004015EF                  call    ecx
.text:004015F1                  add     esp, 4
.text:004015F4                  cmp     esi, esp
.text:004015F6                  call    j___RTC_CheckEsp
```

可以看到有段 call ecx 的间接调用，而ecx 中地址，由：
mov eax , [ebp+var_8]
mov ecx , [eax+edx]
得： ecx = [[ebp+var_8] + edx]
注：下面还有一个 call j_RTC_CheckEsp ，这个RTC_CheckEsp 函数是用来检查堆栈是否平衡的。简单的说，函数在调用之前会把 esp 保存在edi/esi
等寄存器中；当函数调用完之后，RTC_CheckEsp 会去检查这个时候的esp值与之前保存在edi/esi
中的值是否一致；不一致说明esp被改动了，堆栈上存在数据溢出，就会丢出一个错误。
②启用 CFG 保护：

```
.text:004021CE                  mov     ecx, [ebp+var_14]
.text:004021D1                  call    ds:___guard_check_icall_fptr
.text:004021D7                  cmp     edi, esp
.text:004021D9                  call    j___RTC_CheckEsp
.text:004021DE                  call    [ebp+var_14]
```

这里的间接调用是最后一句 call [ebp+var_14] 。在这个调用之前，可以看到：
mov ecx , [ebp+var_14]
call ds: guard_check_icall_fptr
这个里先把下面要调用的地址[ebp+var_14] 放到了 ecx 里面，然后去调用 guard_check_icall_fptr；而 guard_check_icall_fptr
就是CFG保护开启才有的保护函数；这个函数里面，将会去判断 ecx 这个地址里的调用函数是不是一个合法的函数。
然后有 RTC_CheckEsp ，最后才是间接调用。

2.demo

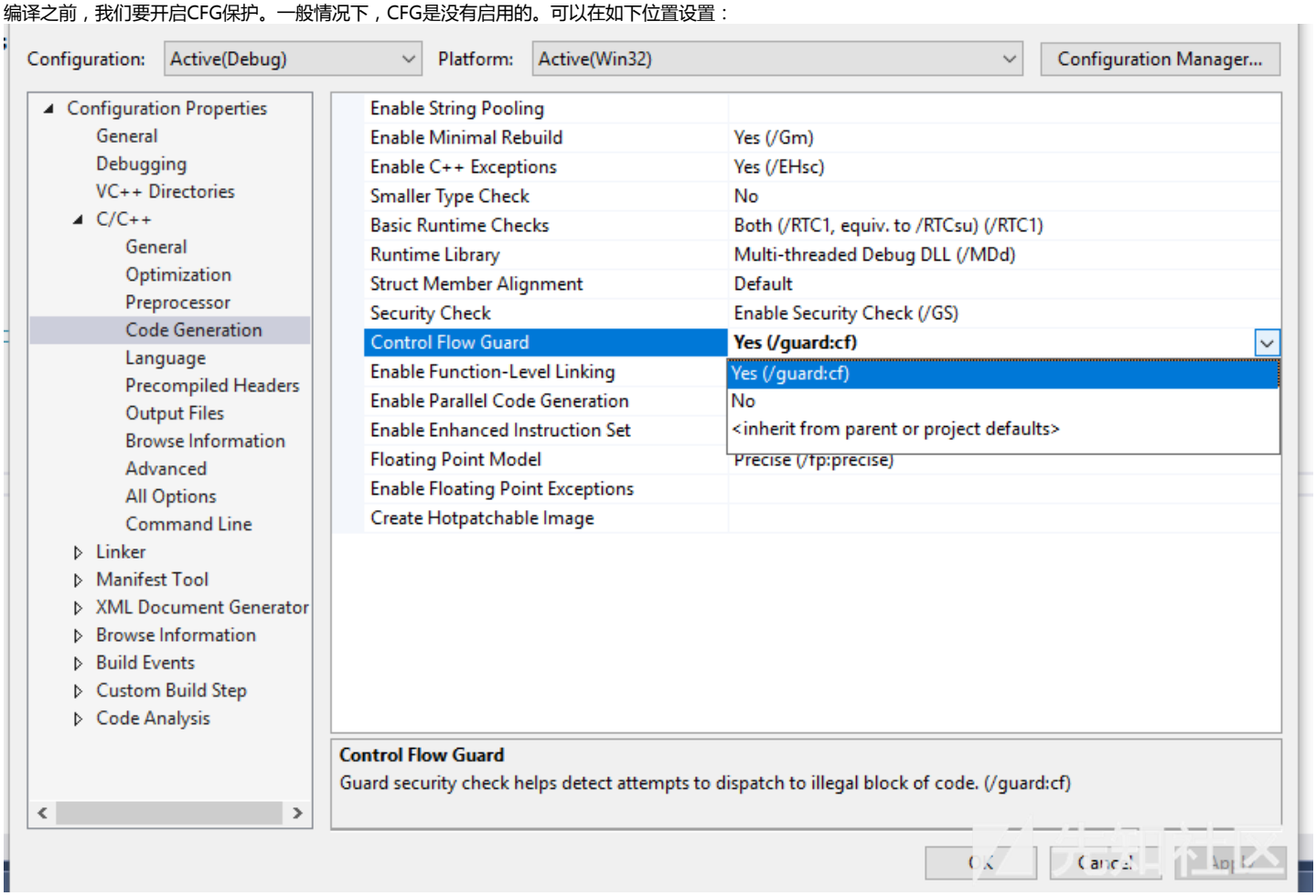
2.1 环境&工具&源码

前面了解了CFG的大概情况，那么这里通过自己写一个简单的程序，进行调用调试分析。
环境与工具：windows 10 pro , Visual Studio 2017 , windbg , IDA pro
源码: 出自这里

```
typedef int(*fun_t)(int);

int foo(int a)
{
   printf("hellow world %d\n",a);
   return a;
}
class CTargetObject
{
public:
   fun_t fun;
```

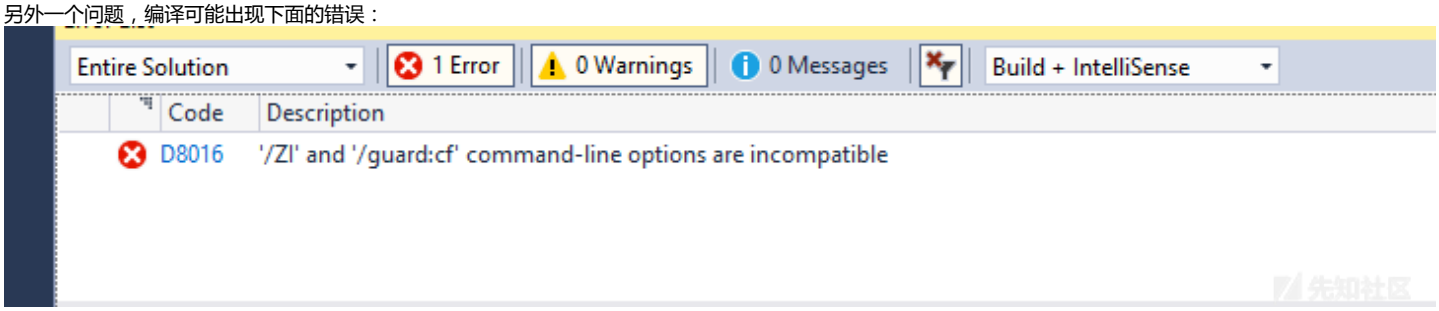
```
};
int main()
{
   int i = 0;
   CTargetObject *o_array = new CTargetObject[5];
   for (i = 0; i < 5 ; i++)
       o_array[i].fun = foo;
   o_array[0].fun(1);
   return 0;
}
```

编译之前，我们要开启CFG保护。一般情况下，CFG是没有启用的。可以在如下位置设置：

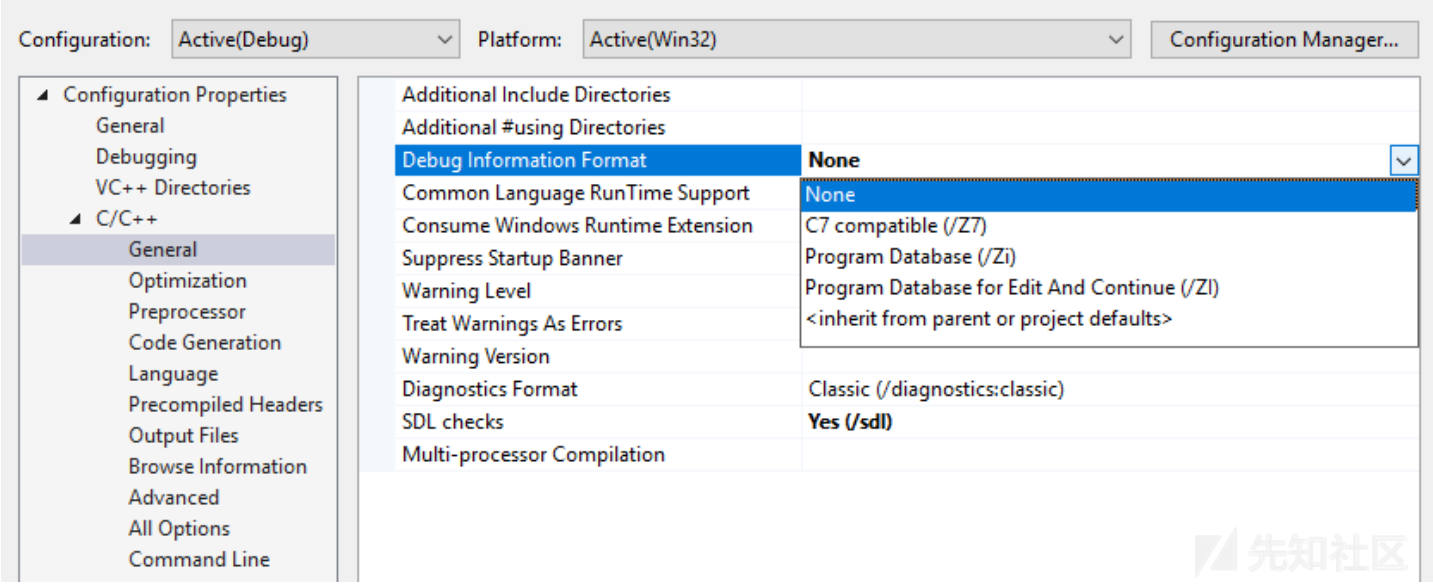


需要注意的是，高版本的Visual Studio 才有这个选项，笔者使用的是2017。

另外一个问题，编译可能出现下面的错误：



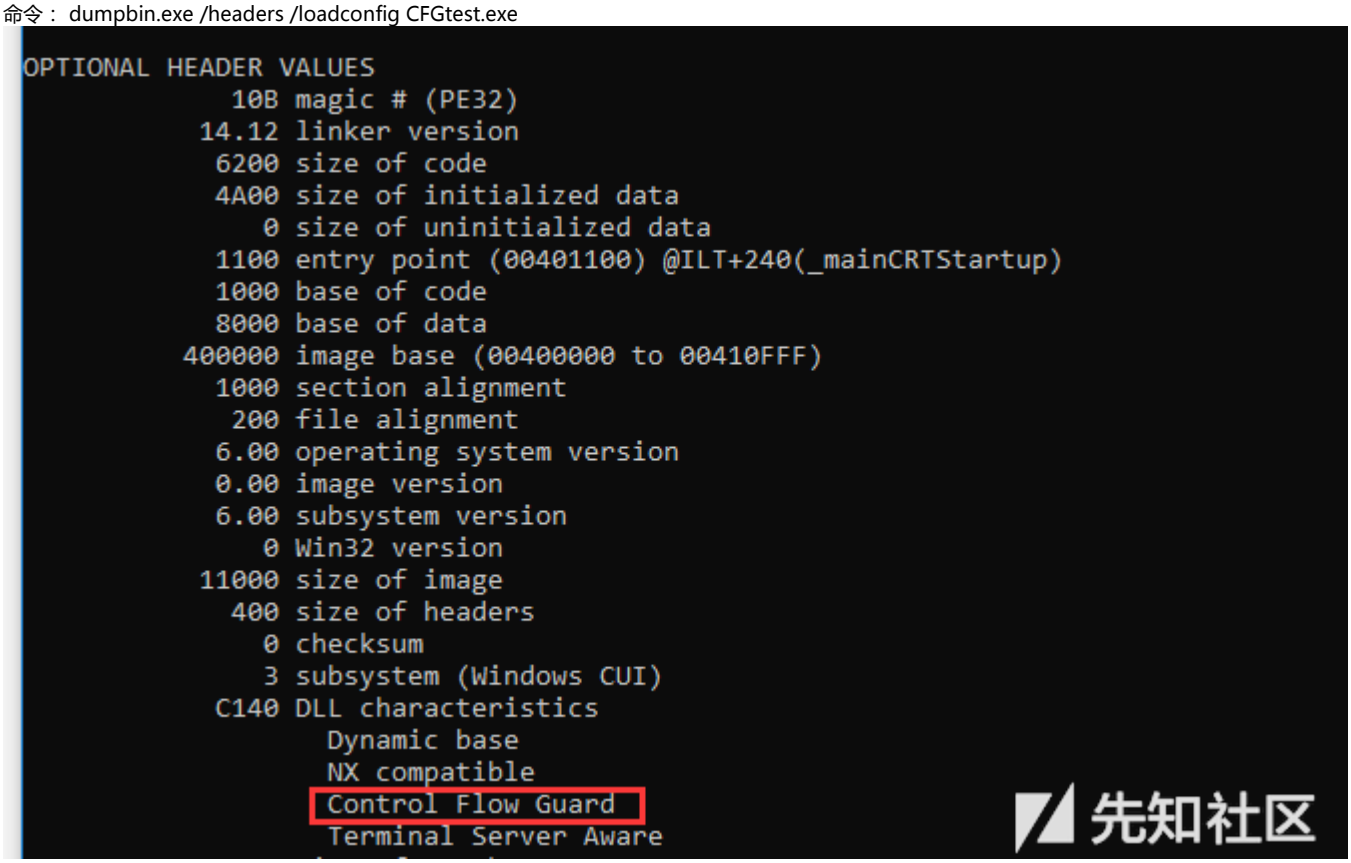
这是说 /guard:cf 选项与 /ZI模式不兼容。我们可以在如下位置把/ZI修改一下，改为None：



最后编译一下工程就行了。

2.2 dumpbin & IDA

编译完成后，查看一下是否真正开启了CFG；使用VS自带的dumpbin.exe 查看一下 header 和 LoadConfig 部分。
命令： dumpbin.exe /headers /loadconfig CFGtest.exe



红框处，说明成功开启CFG。
继续向下看Load Config 部分：

```
Section contains the following load config:

          000000A0 size
                 0 time date stamp
              0.00 Version
                 0 GlobalFlags Clear
                 0 GlobalFlags Set
                 0 Critical Section Default Timeout
                 0 Decommit Free Block Threshold
                 0 Decommit Total Free Threshold
          00000000 Lock Prefix Table
                 0 Maximum Allocation Size
                 0 Virtual Memory Threshold
                 0 Process Heap Flags
                 0 Process Affinity Mask
                 0 CSD Version
              0000 Dependent Load Flag
          00000000 Edit List
          0040B000 Security Cookie
          00409A30 Safe Exception Handler Table
                 1 Safe Exception Handler Count
          0040E000 Guard CF address of check-function pointer
          00000000 Guard CF address of dispatch-function pointer
          0040D000 Guard CF function table
                1C Guard CF function count
          00010500 Guard Flags
                     CF instrumented
                     FID table present
                     Long jump target table present
```


• Guard CF address of check-function pointer：_guard_check_icall_fptr的地址，在调试的时候可以发现它其实是指向ntdll!LdrpValidateUserCallTarget。
• Guard CF address of dispatch-function pointer：在VS2015编译出来上是个保留字段，直译是保护调度函数指针，在IDA中可看到代码就一句 jmp rax 。
• Guard CF function table： RVA列表的指针，其包含了程序的代码。每个函数的RVA将转化为 CFGBitmap中的“1”位。CFGBitmap 的位信息来自Guard CF function table。
• Guard CF function count： RVA的个数。

当然也可以用IDA验证一下：
在编译启用了CFG的模块时，编译器会分析出该模块中所有间接函数调用可达的目标地址，并将这一信息保存在Guard CF Function Table中。
Guard CF function table 表，一个有1C = 28个rva

```
0040D000 Guard CF function table
      1C Guard CF function count
```


IDA中：0x0040D000

```
.gfids:0040D000 ; Segment type: Pure data
.gfids:0040D000 ; Segment permissions: Read
.gfids:0040D000 _gfids          segment para public 'DATA' use32
.gfids:0040D000                 assume cs:_gfids
.gfids:0040D000                 ;org 40D000h
.gfids:0040D000 ___guard_fids_table dd rva j_unknown_libname_3
.gfids:0040D000                                         ; DATA XREF: .rdata:00409608↑o
.gfids:0040D004                 dd rva sub_401090
.gfids:0040D008                 dd rva start
.gfids:0040D00C                 dd rva sub_401160
.gfids:0040D010                 dd rva sub_401240
.gfids:0040D014                 dd rva j___RTC_Shutdown
.gfids:0040D018                 dd rva j___RTC_InitBase
.gfids:0040D01C                 dd rva sub_401330
.gfids:0040D020                 dd rva TopLevelExceptionFilter
.gfids:0040D024                 dd rva sub_4013A0
.gfids:0040D028                 dd rva sub_4013E0
.gfids:0040D02C                 dd rva sub_401400
.gfids:0040D030                 dd rva sub_401430
.gfids:0040D034                 dd rva j_??_Gexception@std@@UAEPAXI@Z ; std::exception::`scalar deleting destructor'(uint)
.gfids:0040D038                 dd rva sub_4015B0
.gfids:0040D03C                 dd rva sub_401630
.gfids:0040D040                 dd rva j_@__security_check_cookie@4 ; __security_check_cookie(x)
.gfids:0040D044                 dd rva j_nullsub_1
.gfids:0040D048                 dd rva j_??_Gexception@std@@UAEPAXI@Z_0 ; std::exception::`scalar deleting destructor'(uint)
.gfids:0040D04C                 dd rva sub_401760
.gfids:0040D050                 dd rva sub_4017B0
.gfids:0040D054                 dd rva j_??0exception@std@@QAE@ABV01@@Z ; std::exception::exception(std::exception const &)
.gfids:0040D058                 dd rva j_?what@exception@std@@UBEPBDXZ ; std::exception::what(void)
.gfids:0040D05C                 dd rva ?pre_c_initialization@@YAHXZ ; pre_c_initialization(void)
.gfids:0040D060                 dd rva ?post_pgo_initialization@@YAHXZ ; post_pgo_initialization(void)
.gfids:0040D064                 dd rva ?pre_cpp_initialization@@YAXXZ ; pre_cpp_initialization(void)
.gfids:0040D068                 dd rva _CrtDbgReport_0
.gfids:0040D06C                 dd rva _CrtDbgReportW_0
.gfids:0040D070                 align 1000h
.gfids:0040D070 _gfids          ends
```

我们去看一下 _guard_check_icall_fptr：

```
.00cfg:0040E000 ; Section 6. (virtual address 0000E000)
.00cfg:0040E000 ; Virtual size                  : 00000104 (    260.)
.00cfg:0040E000 ; Section size in file          : 00000200 (    512.)
.00cfg:0040E000 ; Offset to raw data for section: 00009E00
.00cfg:0040E000 ; Flags 40000040: Data Readable
.00cfg:0040E000 ; Alignment     : default
.00cfg:0040E000 ; ===========================================================================
.00cfg:0040E000
.00cfg:0040E000 ; Segment type: Pure data
.00cfg:0040E000 ; Segment permissions: Read
.00cfg:0040E000 _00cfg          segment para public 'DATA' use32
.00cfg:0040E000                 assume cs:_00cfg
.00cfg:0040E000                 ;org 40E000h
.00cfg:0040E000 ___guard_check_icall_fptr dd offset j_nullsub_1
.00cfg:0040E000                                         ; DATA XREF: _main_0+81↑r
.00cfg:0040E000                                         ; __guard_icall_checks_enforced↑r ...
.00cfg:0040E004                 align 1000h
.00cfg:0040E004 _00cfg          ends
.00cfg:0040E004
.00cfg:0040E004
.00cfg:0040E004                 end start
```

这里的j_nullsub_1 最终是指向ntdll!LdrpValidateUserCallTarget，在ntdll中，IDA再跟进就看不到什么了。

3.原理

在调试demo之前，有必要先了解一下CFG的验证过程。
它会以我们将要跳转的目标函数地址作为自己的参数，并进行以下的操作：

1. 访问CFGbitmap，它表示在进程空间内所有函数的起始位置。在进程空间内每8个字节的状态对应CFGBitmap中的一位。如果函数的地址是合法有效的，那么这个函数对
   进行函数地址校验，会把这个要校验的地址通过计算转换为CFGbitmap中的一位。
   计算的过程，以一个实例来说明：

```
Mov ecx, 0x00b613a0
Mov esi , ecx
Call _guard_check_icall_fptr
call esi
```

2. 上面这段汇编代码，间接跳转的地址是： 0x00b613a0。把这个地址放到了ecx，和 esi中，然后调用 _guard_check_icall_fptr函数。
   在这个函数之中，操作如下：

目标地址 ：
0x00b613a0 = □00000000 10110110 00010011 10100000□ (b)
首先取高位的3个字节：00000000 10110110 00010011 = 0x00b613
那么CFGbitmap的基地址加上 0x00b613 就是指向一个字节单元的指针。
这个指针最终取到的假设是：
CFGBitmap ：0x10100444 = 0001 0000 0001 0000 0000 0100 0100 0100(b)
接着判断目标地址是否以0x10对齐：地址 & 0xf 是否等于 0 ；

- 如果等于0，那偏移是：最低字节二进制位的前5位，这里就是 10100；
- 若不等于0，则偏移是： 10100|0x1。
  这里的例子，0x00b613a0 & 0xf = 0 ，所以偏移： 1010 0 = 20 (d)

这个偏移就是该函数在CFGbitmap中第20位的位置上，若这个位置是1，说明该函数调用是合法有效的，反之则不是。
那么：
CFGBitmap ：0x10100444 = 0001 0000 0001 0000 0000 0100 0100 0100(b)
第20位加粗的 1 ，说明这个调用合法有效。

4.执行流程的跟踪

4.1 正常情况

Windbg版本：6.12.0002.633 AMD64/X86
将前面写的demo程序载入windbg后，下断点。

```
.text:004021CE                  mov     ecx, [ebp+var_14]
.text:004021D1                  call    ds:___guard_check_icall_fptr
.text:004021D7                  cmp     edi, esp
.text:004021D9                  call    j___RTC_CheckEsp
.text:004021DE                  call    [ebp+var_14]
.text:004021E1                  add     esp, 4
.text:004021E4                  cmp     esi, esp
.text:004021E6                  call    j___RTC_CheckEsp
```

断点的位置，结合IDA计算一下，打在如下位置：
X86 ： 00b60000 + 21ce = 00b621ce

```
CommandLine: C:\Users\PC\Desktop\CFG\CFGtest\CFGtest2\Debug\CFGtest2.exe
Symbol search path is: srv*c:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00b60000 00b71000   CFGtest2.exe
ModLoad: 77200000 7738d000   ntdll.dll
ModLoad: 73e00000 73ed0000   C:\Windows\SysWOW64\KERNEL32.DLL
ModLoad: 76b40000 76d17000   C:\Windows\SysWOW64\KERNELBASE.dll
ModLoad: 57ae0000 57afa000   C:\Windows\SysWOW64\VCRUNTIME140D.dll
ModLoad: 0fa10000 0fb82000   C:\Windows\SysWOW64\ucrtbased.dll
ModLoad: 009d0000 00b42000   C:\Windows\SysWOW64\ucrtbased.dll
(1824.2fc8): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000010 ecx=fd8d0000 edx=00000000 esi=003c9000 edi=77206964
eip=772adbcf esp=0019f324 ebp=0019f350 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2b:
772adbcf cc              int     3
```

运行程序，停在断点位置：

```
00b621cc 8bfc            mov     edi,esp
00b621ce 8b4dec          mov     ecx,dword ptr [ebp-14h] ss:002b:0019f80c=00b613a0
00b621d1 ff1500e0b600    call    dword ptr [CFGtest2+0xe000 (00b6e000)]
00b621d7 3bfc            cmp     edi,esp
00b621d9 e802f1ffff      call    CFGtest2+0x12e0 (00b612e0)
00b621de ff55ec          call    dword ptr [ebp-14h]
00b621e1 83c404          add     esp,4
00b621e4 3bf4            cmp     esi,esp
```

这里可以看到 ecx 就是函数间接调用的地址，下面就是 Call dword ptr(00b6e000)
先去看看这个ecx 的地址 0x00b613a0 是不是demo中我们说写的那个函数。
查看 00b631a0 ：

```
0:000> u 00b613a0
CFGtest2+0x13a0:
00b613a0 e98b0d0000      jmp     CFGtest2+0x2130 (00b62130)
00b613a5 cc              int     3
00b613a6 cc              int     3
00b613a7 cc              int     3
00b613a8 cc              int     3
00b613a9 cc              int     3
00b613aa cc              int     3
00b613ab cc              int     3
```

偏移 0x2130：

```
.text:00402130
.text:00402130 ; =============== S U B R O U T I N E =======================================
.text:00402130
.text:00402130 ; Attributes: bp-based frame
.text:00402130
.text:00402130 sub_402130      proc near               ; CODE XREF: sub_4013A0↑j
.text:00402130
.text:00402130 arg_0           = dword ptr  8
.text:00402130
.text:00402130                 push    ebp
.text:00402131                 mov     ebp, esp
.text:00402133                 mov     eax, [ebp+arg_0]
.text:00402136                 push    eax
.text:00402137                 push    offset aHellowWorldD ; "hellow world %d\n"
.text:0040213C                 call    sub_401810
.text:00402141                 add     esp, 8
.text:00402144                 mov     eax, [ebp+arg_0]
.text:00402147                 cmp     ebp, esp
.text:00402149                 call    j___RTC_CheckEsp
.text:0040214E                 pop     ebp
.text:0040214F                 retn
.text:0040214F sub_402130      endp
.text:0040214F
```

可以看到 hellow world 的字符串，说明0x00b631a0 确实是一个正常的间接调用。

那么，继续执行

```
00b621ce 8b4dec          mov     ecx,dword ptr [ebp-14h]
00b621d1 ff1500e0b600    call    dword ptr [CFGtest2+0xe000 (00b6e000)] ds:002b:00b6e000={ntdll!LdrpValidateUserCallTarget (77289be0)}
00b621d7 3bfc            cmp     edi,esp
00b621d9 e802f1ffff      call    CFGtest2+0x12e0 (00b612e0)
00b621de ff55ec          call    dword ptr [ebp-14h]
00b621e1 83c404          add     esp,4
```

跟进: call dword ptr [CFGtest2+0xe000 (00b6e000)]

```
ntdll!LdrpValidateUserCallTarget:
77289be0 8b15e8923177    mov     edx,dword ptr [ntdll!LdrSystemDllInitBlock+0xb0 (773192e8)]
77289be6 8bc1            mov     eax,ecx
77289be8 c1e808          shr     eax,8
ntdll!LdrpValidateUserCallTargetBitMapCheck:
77289beb 8b1482          mov     edx,dword ptr [edx+eax*4] ds:002b:00bad84c=10100444
77289bee 8bc1            mov     eax,ecx
77289bf0 c1e803          shr     eax,3
77289bf3 f6c10f          test    cl,0Fh
77289bf6 7506            jne     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (77289bfe)
77289bf8 0fa3c2          bt      edx,eax
77289bfb 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (77289bfe)
ntdll!LdrpValidateUserCallTargetBitMapRet:
77289bfd c3              ret
77289bfe 83c801          or      eax,1
77289c01 0fa3c2          bt      edx,eax
77289c04 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0xa (77289c07)
77289c06 c3              ret
77289c07 51              push    ecx
77289c08 8d642480        lea     esp,[esp-80h]
77289c0c 0f110424        movups  xmmword ptr [esp],xmm0
77289c10 0f114c2410      movups  xmmword ptr [esp+10h],xmm1
77289c15 0f11542420      movups  xmmword ptr [esp+20h],xmm2
77289c1a 0f115c2430      movups  xmmword ptr [esp+30h],xmm3
77289c1f 0f11642440      movups  xmmword ptr [esp+40h],xmm4
77289c24 0f116c2450      movups  xmmword ptr [esp+50h],xmm5
77289c29 0f11742460      movups  xmmword ptr [esp+60h],xmm6
77289c2e 0f117c2470      movups  xmmword ptr [esp+70h],xmm7
77289c33 e81de60400      call    ntdll!RtlpHandleInvalidUserCallTarget (772[illegible]55)
77289c38 0f100424        movups  xmm0,xmmword ptr [esp]
```

来到了前面讲到的：ntdll!LdrpValidateUserCallTarget

```
ntdll!LdrpValidateUserCallTarget:
77289be0 8b15e8923177    mov     edx,dword ptr [ntdll!LdrSystemDllInitBlock+0xb0 (773192e8)]
77289be6 8bc1            mov     eax,ecx
77289be8 c1e808          shr     eax,8
ntdll!LdrpValidateUserCallTargetBitMapCheck:
77289beb 8b1482          mov     edx,dword ptr [edx+eax*4] ds:002b:00bad84c=10100444
```

```
0x77289be0 ■■ edx ■■■■■■■■ ■■CFGbitmap■■■■■■■■■
edx = 0x00b80000
eax = ecx =  0x00b613a0
shr  eax >> 8 = 0x0000b613 //■■■■■8■■■■■■■■■3■■■
edx = [ 0x00b80000 + 0x0000b613*4 ] = 0x10100444
■■■CFGbitmap = 0x10100444 ■■■■■■■■■■■■■■■■
```

下面，来验证：

```
77289bf0 c1e803          shr     eax,3
77289bf3 f6c10f          test    cl,0Fh
77289bf6 7506            jne     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (77289bfe)
77289bf8 0fa3c2          bt      edx,eax
77289bfb 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (77289bfe)
ntdll!LdrpValidateUserCallTargetBitMapRet:
77289bfd c3              ret
```

```
eax = ecx = 0x00b613a0   ->   eax  >> 3 = 0x0016c274
test cl ,0Fh    //   0xa0  &  0x0F  = 0
ZF = 0           // ZF ■0 ,jne■■■■■■■■0x0f■■
bt edx, eax
/*
bt ■■■■■■■■■■■CF■■■■■■■■■■■■■■■ edx ■■■■■■■CF■■■■■■■■■eax■■■■■■■eax■■■■■■■ eax mod 0x20 ■■■
■■■■■mod 0x20■■■■■■■■■■■■■■■■■■■■■■edx, edx ■32■■■■■■0x20 = 32 (d)■
■■■■■eax■■■■■mod■■■■■■■■■■■■■■■■■■■■■16/32■■■■■■■■■■■■■
■■■eax ■■3■■■■■■bt ■■■■■■■■■■■■■■■5■ ■■■■■■■■■■■
eax = 0x00b613a0
0000 0000 1011 0110 0001 0011 1010 0000
eax >> 3  // ■■■■■■■■8■■■■3■ ■■■■5■
0000 0000 0001 0110 1100 0010 0111 0100
■■■■■5■■■■■■■■■5■1■ 11111  =  31 (■■■■) ■■■■■■ eax■ 0x20 ■■■■■■■■■■■■
■■:
shr eax, 3
bt  edx, eax

■2■■■■■■■■■■■■■■■5■■■■■■■■■■■■■CF■■■■■■■■■■■■■■■jae■■■■■■■■■■■■■■■■■■■■■CF = 0■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
*/
```

4.2 错误处理

让程序断在相同的位置：

```
00b621cc 8bfc            mov     edi,esp
00b621ce 8b4dec          mov     ecx,dword ptr [ebp-14h] ss:002b:0019f80c=00b613a0
00b621d1 ff1500e0b600    call    dword ptr [CFGtest2+0xe000 (00b6e000)]
00b621d7 3bfc            cmp     edi,esp
00b621d9 e802f1ffff      call    CFGtest2+0x12e0 (00b612e0)
00b621de ff55ec          call    dword ptr [ebp-14h]
00b621e1 83c404          add     esp,4
00b621e4 3bf4            cmp     esi,esp
```

修改ecx : 00b613b0

```
0:000> r ecx=00b613b0
0:000> r
eax=051842d8 ebx=02c22000 ecx=00b613b0 edx=00000000 esi=02f7fd9c edi=02f7fd98
eip=77289be0 esp=02f7fd94 ebp=02f7fdb8 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
ntdll!LdrpValidateUserCallTarget:
77289be0 8b15e8923177    mov     edx,dword ptr [ntdll!LdrSystemDllInitBlock+0xb0 (773192e8)] ds:002b:773192e8=00bf0000
0:000>
```

修改ecx 为00b613b0 之后，可以发现下面的情况：

```
ntdll!LdrpValidateUserCallTargetBitMapCheck:
77289beb 8b1482          mov     edx,dword ptr [edx+eax*4]
77289bee 8bc1            mov     eax,ecx
77289bf0 c1e803          shr     eax,3
77289bf3 f6c10f          test    cl,0Fh
77289bf6 7506            jne     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (77289bfe)
77289bf8 0fa3c2          bt      edx,eax
77289bfb 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (77289bfe) [br=1]
ntdll!LdrpValidateUserCallTargetBitMapRet:
77289bfd c3              ret
77289bfe 83c801          or      eax,1
77289c01 0fa3c2          bt      edx,eax
77289c04 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0xa (77289c07)
77289c06 c3              ret
```

也就是说当正常的地址00b613a0，被改为
00b613b0的时候，虽然这不是一个合法的地址，但是在CFG检查的时候，第一次CF标志位被设为0，通过jae跳转到到了0x77289bfe Or eax，1

但是，根据我们前面的分析，当地址被修改为 00b613a0
时，这不是一个正常的函数地址。CFG函数检查之后，CF标志位被置0，接下来应该是抛出异常。但根据windbg调试来看，并不是这样的流程。而是跳转到0x77289bfe，然后继续执行：

```
Or eax■ 1
// eax  = 0x0016c276
// eax = 0x0016c276 or 1 = 0x0016c277
bt  edx, eax          - >  CF = 0
jae ■■
```

```
77289bfe 83c801          or      eax,1
77289c01 0fa3c2          bt      edx,eax
77289c04 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0xa (77289c07)
77289c06 c3              ret
77289c07 51              push    ecx
77289c08 8d642480        lea     esp,[esp-80h]
77289c0c 0f110424        movups  xmmword ptr [esp],xmm0
77289c10 0f114c2410      movups  xmmword ptr [esp+10h],xmm1
77289c15 0f11542420      movups  xmmword ptr [esp+20h],xmm2
77289c1a 0f115c2430      movups  xmmword ptr [esp+30h],xmm3
77289c1f 0f11642440      movups  xmmword ptr [esp+40h],xmm4
77289c24 0f116c2450      movups  xmmword ptr [esp+50h],xmm5
77289c29 0f11742460      movups  xmmword ptr [esp+60h],xmm6
77289c2e 0f117c2470      movups  xmmword ptr [esp+70h],xmm7
77289c33 e81de60400      call    ntdll!RtlpHandleInvalidUserCallTarget (772d8255)
```

这里为什么会在进行一次or 1的操作呢？
后面笔者参考了很多其他资料，得出下面的结论（仅个人理解，若出现错误望大家斧正）：

- win10新增一个功能可以抑制导出，意思是现在导出函数能在CFG保护的调用位置被标记为非法目的地址。注意，导出函数本身是正常合法的函数。这一功能的实现需要
  Bitmap中每一个地址的第二位，以及在初始化每个进程的bitmap时guard_fids_table中每一个RVA条目的一个标记字节。
- 正常的受限的导出函数现在用的2个标志位来检查的。
  就像上面的情况，第一次检查CF = 0
  ，这个时候检查的地址可能是非法的，但也可能是正常的受限的导出函数，若是受限的导出函数，那它在CFGbitmap中用2位标识，且这2位的情况就是 10
  ，所以进行第二次检查的时候，就会检查到"1"，那这就是个正常的函数调用过程。所以，不会抛出异常。这样一些导出表将会在进行创建时以不合法的间接调用开始，但
  另外：根据上面的这种情况分析可知00b613a0 = 00b613e0 = 00b61330
  这3个地址是等价的，虽然正确的地址是00b613a0 ，但它们都能通过CFG的检查，不影响程序的执行，并且打印出hello world字符串。

4.3 int 29

当我们提供一个错误的地址，我们尝试跟踪完整的中断过程，看看CFG是怎么做的。
地址：0x77289c07

```
77289c07 51              push    ecx
77289c08 8d642480        lea     esp,[esp-80h]
77289c0c 0f110424        movups  xmmword ptr [esp],xmm0
77289c10 0f114c2410      movups  xmmword ptr [esp+10h],xmm1
77289c15 0f11542420      movups  xmmword ptr [esp+20h],xmm2
77289c1a 0f115c2430      movups  xmmword ptr [esp+30h],xmm3
77289c1f 0f11642440      movups  xmmword ptr [esp+40h],xmm4
77289c24 0f116c2450      movups  xmmword ptr [esp+50h],xmm5
77289c29 0f11742460      movups  xmmword ptr [esp+60h],xmm6
77289c2e 0f117c2470      movups  xmmword ptr [esp+70h],xmm7
77289c33 e81de60400      call    ntdll!RtlpHandleInvalidUserCallTarget (772d8255)
```

```
Command
0093fd74  009742d8 0093fd84 00cc237c 00000014
0:000> dd 0093fd04+10
0093fd14  0093fd38 559c9876 00000014 00000001
0093fd24  00000000 00000000 3b616b50 009742d8
0093fd34  0000d6a0 0093fd50 559cc00a 00000014
0093fd44  00000001 00000000 00000000 0093fd68
0093fd54  559cc914 00000014 00000001 00000000
0093fd64  00000000 0093fd78 00cc2add 00000014
0093fd74  009742d8 0093fd84 00cc237c 00000014
0093fd84  00cc13b0 00cc21d7 00000001 00cc1100
0:000> p
eax=00198277 ebx=00795000 ecx=00cc13b0 edx=10100444 esi=0093fd90 edi=0093fd8c
eip=77289c15 esp=0093fd04 ebp=0093fdac iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
ntdll!LdrpValidateUserCallTargetBitMapRet+0x18:
77289c15 0f11542420      movups  xmmword ptr [esp+20h],xmm2 ss:002b:0093fd24=009742d83b616b500000000000000000
0:000> dd 0093fd04+10
0093fd14  00000000 00000000 00000000 00000000
0093fd24  00000000 00000000 3b616b50 009742d8
0093fd34  0000d6a0 0093fd50 559cc00a 00000014
```

跟进 ntdll!RtlpHandleInvalidUserCallTarget：

```
ntdll!RtlpHandleInvalidUserCallTarget:
772d8255 8bff            mov     edi,edi
772d8257 55              push    ebp
772d8258 8bec            mov     ebp,esp
772d825a 83ec24          sub     esp,24h
772d825d 8d45fc          lea     eax,[ebp-4]
772d8260 56              push    esi
772d8261 6a00            push    0
772d8263 6a04            push    4
772d8265 50              push    eax
772d8266 6a22            push    22h
772d8268 6aff            push    0FFFFFFFFh
772d826a 8bf1            mov     esi,ecx
772d826c e87f66f9ff      call    ntdll!NtQueryInformationProcess (7
```

继续 ntdll!NtQueryInformationProcess:

```
ntdll!NtQueryInformationProcess:
7726e8f0 b819000000      mov     eax,19h
7726e8f5 e800000000      call    ntdll!ZwQueryInformationProcess+0xa (7726e8fa)
7726e8fa 5a              pop     edx
7726e8fb 807a144b        cmp     byte ptr [edx+14h],4Bh
7726e8ff 750e            jne     ntdll!ZwQueryInformationProcess+0x1f (7726e90f)
7726e901 64ff15c0000000  call    dword ptr fs:[0C0h]
7726e908 c21400          ret     14h
```

ntdll!ZwQueryInformationProcess+0x1f (7726e90f):

```
7726e90f ba209d2877      mov     edx,offset ntdll!Wow64SystemServiceCall (77289d20)
7726e914 ffd2            call    edx
7726e916 c21400          ret     14h
```

最终调用的函数层次比较深，用下图进行说明：

```
7726e916 c21400          ret     14h
...
772d8271 85c0            test    eax,eax
772d8273 7829            js      ntdll_77200000!RtlpHandleInvalidUserCallTarget+0x49 (772d829e)
772d8275 8b45fc          mov     eax,dword ptr [ebp-4]
772d8278 a802            test    al,2
772d827a 7540            jne     ntdll_77200000!RtlpHandleInvalidUserCallTarget+0x67 (772d82bc)
772d827c 2405            and     al,5
772d827e 3c01            cmp     al,1
772d8280 751c            jne     ntdll_77200000!RtlpHandleInvalidUserCallTarget+0x49 (772d829e) [br=1]
...
772d829e 803d2692317700  cmp     byte ptr [ntdll_77200000!RtlGuardAllowSuppressedCalls (77319226)],0
772d82a5 741a            je      ntdll_77200000!RtlpHandleInvalidUserCallTarget+0x6c (772d82c1) [br=1]

...
772d82c1 e85b32f7ff      call    ntdll_77200000!LdrControlFlowGuardEnforcedWithExportSuppression (7724b521)
772d82c6 85c0            test    eax,eax
772d82c8 7416            je      ntdll_77200000!RtlpHandleInvalidUserCallTarget+0x8b (772d82e0) [br=1]

..
772d82e0 6a0a            push    0Ah
772d82e2 8bd6            mov     edx,esi
772d82e4 59              pop     ecx
772d82e5 e8261afbff      call    ntdll_77200000!RtlFailFast2 (77289d10)
ntdll_77200000!RtlFailFast2:
77289d10 cd29            int     29h
```

最后中断在：
0x77289d10 int 29h

5.Edge-x64 / IE 11(32/64)

下面就简单的调试下CFG在Edge，IE11下的情况。

5.1 Edge x64

使用dumpbin /headers /loadconfig xxx.exe ，查看Edge开启的保护：

```
            D160 DLL characteristics
                   High Entropy Virtual Addresses
                   Dynamic base
                   NX compatible
                   App Container
                   Control Flow Guard
                   Terminal Server Aware
    0000000140778970 Guard CF address of check-function pointer
    0000000140778978 Guard CF address of dispatch-function pointer
    0000000140778980 Guard CF function table
                7D98 Guard CF function count
            00017500 Guard Flags
```

在IDA的情况下：

```
.rdata:000000014088C6A0                 dq offset __guard_check_icall_fptr ; GuardCFCheckFunctionPointer
.rdata:000000014088C6A8                 dq offset __guard_dispatch_icall_fptr ; Reserved2
.rdata:000000014088C6B0                 dq offset __guard_fids_table ; GuardCFFunctionTable
.rdata:000000014088C6B8                 dq 7D98h                ; GuardCFFunctionCount
.rdata:000000014088C6C0                 dd 17500h               ; GuardFlags

.rdata:00000001407787D8 __guard_check_icall_fptr dq offset ?DependencyPropertyChanged@PdfSeekBar_obj1_Bindings@PdfSeek
.rdata:00000001407787D8                                         ; DATA XREF: .rdata:000000014088C6A0↓o
.rdata:00000001407787D8                                         ; SpartanXAML::PdfSeekBar::PdfSeekBar_obj1_Bindings::D
.rdata:00000001407787E0 __guard_dispatch_icall_fptr dq offset _guard_dispatch_icall_nop
```

下断点跟踪：

```
ntdll!LdrpValidateUserCallTarget:
00007ff8`967d0da0 488b15c9e50d00  mov     rdx,qword ptr [ntdll!LdrSystemDllInitBlock+0xb0 (00007ff8`968af370)] ds:00007ff8`968af370=00007df5ff3e000
00007ff8`967d0da7 488bc1          mov     rax,rcx
00007ff8`967d0daa 48c1e809        shr     rax,9
00007ff8`967d0dae 488b14c2        mov     rdx,qword ptr [rdx+rax*8]
00007ff8`967d0db2 488bc1          mov     rax,rcx
00007ff8`967d0db5 48c1e803        shr     rax,3
00007ff8`967d0db9 f6c10f          test    cl,0Fh
00007ff8`967d0dbc 7507            jne     ntdll!LdrpValidateUserCallTarget+0x25 (00007ff8`967d0dc5)
00007ff8`967d0dbe 480fa3c2        bt      rdx,rax
00007ff8`967d0dc2 7301            jae     ntdll!LdrpValidateUserCallTarget+0x25 (00007ff8`967d0dc5)
00007ff8`967d0dc4 c3              ret
00007ff8`967d0dc5 4883c801        or      rax,1
00007ff8`967d0dc9 480fa3c2        bt      rdx,rax
00007ff8`967d0dcd 7301            jae     ntdll!LdrpValidateUserCallTarget+0x30 (00007ff8`967d0dd0)
00007ff8`967d0dcf c3              ret
00007ff8`967d0dd0 488bc1          mov     rax,rcx
```

从上图看汇编代码，可以发现与我们的demo程序的情况是一致的，区别就是64位的系统，地址的第9-63位被用于在CFGbimap中检索一个qword，第3-10位被用于（模64

```
ntdll!LdrpValidateUserCallTarget:
00007ff8`967d0da0 488b15c9e50d00  mov     rdx,qword ptr [ntdll!LdrSystemDllInitBlock+0xb0 (00007ff8`968af370)]
00007ff8`967d0da7 488bc1          mov     rax,rcx
00007ff8`967d0daa 48c1e809        shr     rax,9
00007ff8`967d0dae 488b14c2        mov     rdx,qword ptr [rdx+rax*8]
00007ff8`967d0db2 488bc1          mov     rax,rcx
00007ff8`967d0db5 48c1e803        shr     rax,3
00007ff8`967d0db9 f6c10f          test    cl,0Fh
00007ff8`967d0dbc 7507            jne     ntdll!LdrpValidateUserCallTarget+0x25 (00007ff8`967d0dc5)
00007ff8`967d0dbe 480fa3c2        bt      rdx,rax
00007ff8`967d0dc2 7301            jae     ntdll!LdrpValidateUserCallTarget+0x25 (00007ff8`967d0dc5)
00007ff8`967d0dc4 c3              ret
00007ff8`967d0dc5 4883c801        or      rax,1
00007ff8`967d0dc9 480fa3c2        bt      rdx,rax
00007ff8`967d0dcd 7301            jae     ntdll!LdrpValidateUserCallTarget+0x30 (00007ff8`967d0dd0)
00007ff8`967d0dcf c3              ret
00007ff8`967d0dd0 488bc1          mov     rax,rcx
00007ff8`967d0dd3 4d33d2          xor     r10,r10
00007ff8`967d0dd6 e905ffffff      jmp     ntdll!LdrpHandleInvalidUserCallTarget (00007ff8`967d0ce0)
00007ff8`967d0ddb cc              int     3
```

5.2 IE32

调试时候先找一个有CFG保护的函数，确定一下函数名，然后下断点：

xrefs to __guard_check_icall_fptr

Directio	Typ	Address	Text
Up	o	.text:00401050	dd offset ___guard_check_icall_fptr; GuardCFCheckFunctionPointer
Up	r	wWinMain(x,x,x,x)+276	call ds:___guard_check_icall_fptr; _guard_check_icall_nop(x)
	r	sub_4034B3+F5	call ds:___guard_check_icall_fptr; _guard_check_icall_nop(x)
D...	r	__initterm_e+28	call ds:___guard_check_icall_fptr; _guard_check_icall_nop(x)
D...	r	__onexit+28	call ds:___guard_check_icall_fptr; _guard_check_icall_nop(x)
D...	r	wil::details::ThreadFailure...	call ds:___guard_check_icall_fptr; _guard_check_icall_nop(x)
D...	r	wil::details::ThreadFailure...	call ds:___guard_check_icall_fptr; _guard_check_icall_nop(x)
D...	r	wil::GetFailureLogString(us	call ds:___guard_check_icall_fptr; _guard_check_icall_nop(x)

笔者找的红框的2个函数，下面用initterm_e 说明就行。
iexplore!_initterm_e 的情况如下：

```
001236a4 8b3e            mov     edi,dword ptr [esi]
001236a6 85ff            test    edi,edi
001236a8 740a            je      iexplore!_initterm_e+0x30 (001236b4)
001236aa 8bcf            mov     ecx,edi
001236ac ff15cc911200    call    dword ptr [iexplore!__guard_check_icall_fptr (001291cc)]
001236b2 ffd7            call    edi {iexplore!operator new+0x35 (001233b0)}
001236b4 83c604          add     esi,4
ntdll!LdrpValidateUserCallTarget:
77289be0 8b15e8923177    mov     edx,dword ptr [ntdll!LdrSystemDllInitBlock+0xb0 (773192e8)]
77289be6 8bc1            mov     eax,ecx
77289be8 c1e808          shr     eax,8
ntdll!LdrpValidateUserCallTargetBitMapCheck:
77289beb 8b1482          mov     edx,dword ptr [edx+eax*4]
77289bee 8bc1            mov     eax,ecx
77289bf0 c1e803          shr     eax,3
77289bf3 f6c10f          test    cl,0Fh
77289bf6 7506            jne     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (77289bfe)
77289bf8 0fa3c2          bt      edx,eax
77289bfb 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (77289bfe) [br=0]
ntdll!LdrpValidateUserCallTargetBitMapRet:
77289bfd c3              ret
```

进行简单的计算一下：

```
iexplore!_onexit■
eax = ecx = 0x73e18630
…
edx = 0x80080082 =1000 0000 0000 1000 0000 0000 1000 0010
eax >>  3 = 0x0e7c30c6
0x0e7c30c6 mod 0x20 = 0x00000006
CF = 0
```

```
ntdll!LdrpValidateUserCallTargetBitMapCheck:
77289beb 8b1482          mov     edx,dword ptr [edx+eax*4]
77289bee 8bc1            mov     eax,ecx
77289bf0 c1e803          shr     eax,3
77289bf3 f6c10f          test    cl,0Fh
77289bf6 7506            jne     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (77289bfe)
77289bf8 0fa3c2          bt      edx,eax
77289bfb 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (77289bfe) [br=1]
ntdll!LdrpValidateUserCallTargetBitMapRet:
77289bfd c3              ret
77289bfe 83c801          or      eax,1
77289c01 0fa3c2          bt      edx,eax
77289c04 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0xa (77289c07)
77289c06 c3              ret
```

同样，这里也会比对CFGbitmap中的2位。道理如同上面所讲，这里不再赘述。

5.3 IE64

依然查看它的保护情况：18 h = 24d

```
0000000140006480 Guard CF address of check-function pointer
0000000140006488 Guard CF address of dispatch-function pointer
0000000140006528 Guard CF function table
              18 Guard CF function count
```

在有CFG保护的位置下断，由于我是直接开启iexplore.exe，能断下的函数只有下图bsearch的位置，其他在开启IE过程中，都断不下来，不过并不影响什么：

```
00007ff8`967d3435 c3              ret
00007ff8`967d3436 4d85ff          test    r15,r15
00007ff8`967d3439 74c6            je      ntdll!bsearch+0x41 (00007ff8`967d3401)
00007ff8`967d343b 4c39742470      cmp     qword ptr [rsp+70h],r14
00007ff8`967d3440 74bf            je      ntdll!bsearch+0x41 (00007ff8`967d3401)
00007ff8`967d3442 488b4c2470      mov     rcx,qword ptr [rsp+70h]
00007ff8`967d3447 e8103b0000      call    ntdll!guard_check_icall (00007ff8`967d6f5c)
00007ff8`967d344c eb3d            jmp     ntdll!bsearch+0xcb (00007ff8`967d348b)
00007ff8`967d344e 488bee          mov     rbp,rsi
00007ff8`967d3451 48d1ed          shr     rbp,1
```

跟踪情况：

```
ntdll!LdrpValidateUserCallTarget:
00007ff8`967d0da0 488b15c9e50d00  mov     rdx,qword ptr [ntdll!LdrSystemDllInitBlock+0xb0 (00007ff8`968af370)]
00007ff8`967d0da7 488bc1          mov     rax,rcx
00007ff8`967d0daa 48c1e809        shr     rax,9
00007ff8`967d0dae 488b14c2        mov     rdx,qword ptr [rdx+rax*8]
00007ff8`967d0db2 488bc1          mov     rax,rcx
00007ff8`967d0db5 48c1e803        shr     rax,3
00007ff8`967d0db9 f6c10f          test    cl,0Fh
00007ff8`967d0dbc 7507            jne     ntdll!LdrpValidateUserCallTarget+0x25 (00007ff8`967d0dc5)
00007ff8`967d0dbe 480fa3c2        bt      rdx,rax
00007ff8`967d0dc2 7301            jae     ntdll!LdrpValidateUserCallTarget+0x25 (00007ff8`967d0dc5)
00007ff8`967d0dc4 c3              ret
```

从上面的跟踪可以知道，无论是demo，Edge，或是 IE ，最终都会来到ntdll中实现CFG检查，无论是32位还是64位，检查过程都是一致的。

6.bypass部分

这里bypass只列出一部分，也会简单的说明一下。

6.1 CVE-2015-0311

利用这个CVE可以先达到任意内存读写。
然后Adobe Flash Player二进制文件中有很多间接调用，但是都有CFG保护。
到目前为止，CFG在保护这个Adobe Flash Player二进制文件中的29,000多个间接调用方面做得非常出色。
但是需要找一个不受CFG保护的间接调用。
Flash Player包含一个即时（JIT）编译器，该编译器将虚拟机字节码转换为本地代码以提高执行速度。由JIT编译器生成的代码包括间接调用，
并且由于此代码是在运行时生成的，这意味着其中的间接调用不受CFG的保护。

```
0864D88C   8B01          MOV EAX,DWORD PTR DS:[ECX]          EAX = ByteArray object
0864D88E   8B50 08       MOV EDX,DWORD PTR DS:[EAX+8]        EDX = VTable object
0864D891   8B8A D4000000 MOV ECX,DWORD PTR DS:[EDX+D4]       ECX = MethodEnv object from VTable_object + 0xD4
0864D897   8D55 FC       LEA EDX,DWORD PTR SS:[EBP-4]
0864D89A   8945 FC       MOV DWORD PTR SS:[EBP-4],EAX
0864D89D   8B41 04       MOV EAX,DWORD PTR DS:[ECX+4]        EAX = function pointer from MethodEnv_object + 4
0864D8A0   83EC 04       SUB ESP,4
0864D8A3   52            PUSH EDX
0864D8A4   6A 00         PUSH 0
0864D8A6   51            PUSH ECX
0864D8A7   FFD0          CALL EAX                            call the function pointer! No CFG here!
0864D8A9   83C4 10       ADD ESP,10
0864D8AC   8B4D F0       MOV ECX,DWORD PTR SS:[EBP-10]
0864D8AF   890D 50406908 MOV DWORD PTR DS:[8694050],ECX
0864D8B5   8BE5          MOV ESP,EBP
0864D8B7   5D            POP EBP
0864D8B8   C3            RETN
```

首先有一个Vector容器，且长度很大，这个长度将允许我们从进程的地址空间内的任何内存地址读/写。
然后有一个ByteArray对象，我们的ROP链存储在Vector的第一个元素。
这个ByteArray对象+ 8的位置是一个指向VTable对象的指针
VTable_object 里面又包含很多其他指针。
在 VTable_object + 0xD4 处有一个指针，指向MethodEnv对象
MethodEnv对象第一个双字是一个指向它自己的的vtable的指针，而第二个双字是一个函数指针。
通过观察发现 MethodEnv+4处的指针，是间接调用实现的。取消引用了存储在VTable_object + 0xD4 处的 MethodEnv 对象的指针
也就是说，本应该是 通过 VTable_object + 0xD4 + 偏移 去调用 MethodEnv 第二个双字处的函数指针，然后去使用。
可以从图上看到，它是通过mov mov 然后取到了这些需要调用的位置。
此间接调用来自Flash JIT编译器生成的代码，通过调用ByteArray对象上的toString()方法，可以可靠地触发这种无保留的间接调用。

6.2 BlackHat 2015

CustomHeap::Heap对象 ， 改写guard_check_icall_fptr，指向合适的函数。

```
1  DWORD __stdcall CustomHeap::Heap::EnsurePageReadWrite<1,4>(int a1)
2  {
3    DWORD result; // eax@3
4    DWORD flOldProtect; // [sp+4h] [bp-4h]@3
5    if ( *(_BYTE *)(a1 + 1) || *(_BYTE *)a1 )
6    {
7      result = 0;
8    }
9    else
10   {
11     flOldProtect = 0;
12     VirtualProtect(*(LPVOID *)(a1 + 0xC), 0x1000u, 4u, &flOldProtect);
13     result = flOldProtect;
14     *(_BYTE *)(a1 + 1) = 1;
15   }
16   return result;
17 }
```

利用jscript9中的CustomHeap::Heap对象将 guard_check_icall_fptr 变成可读写的:

```
CustomHeap::Heap::FreeAll ■■■ Bucket ■■■■ CustomHeap::Heap::FreeBucket
CustomHeap::Heap::FreeBucket ■■ Bucket ■■■■■■■■■■■■ CustomHeap::Page ■■■■CustomHeap::Heap::EnsurePageReadWrite <1,4>■
CustomHeap::Heap::EnsurePageReadWrite <1,4>■
■■■■■■■■ VirtualProtect ■■■■ VirtualProtect ■■■■■■■■■■■■■■■■■■■■■■
■4■■■■■■
```

```
lpAddress: CustomHeap::Page ████████address
dwSize: 0x1000
flNewProtect: PAGE_READWRITE   // ████████
flodlProtect:                  // █████████
█████████PAGE_READWRITE ████████CFG███ptr███████████CFG■
```

patch：补丁如下图

```
1  int __thiscall HeapPageAllocator::ProtectPages(HeapPageAllocator *this, LPCVOID lpAddress, unsigned int a3, s
2  {
3    unsigned __int32 v7; // ebx@1
4    unsigned int v8; // edx@2
5    int result; // eax@7
6    struct _MEMORY_BASIC_INFORMATION Buffer; // [sp+Ch] [bp-20h]@4
7    DWORD flOldProtect; // [sp+28h] [bp-4h]@7
8    v7 = (unsigned __int32)this;
9    if ( (unsigned __int16)lpAddress & 0xFFF
10     || (v8 = *((_DWORD *)a4 + 2), (unsigned int)lpAddress < v8)
11     || (unsigned int)((char *)lpAddress - v8) > (*((_DWORD *)a4 + 3) - a3) << 12
12     || !VirtualQuery(lpAddress, &Buffer, 0x1Cu)
13     || Buffer.RegionSize < a3 << 12
14     || a7 != Buffer.Protect )
15   {
16     CustomHeap_BadPageState_fatal_error(v7);
17     result = 0;
18   }
19   else
20   {
21     *a6 = Buffer.Protect;
22     result = VirtualProtect((LPVOID)lpAddress, a3 << 12, flNewProtect, &flOldProtect);
23   }
24   return result;
25 }
```


微软引入了一个新的函数HeapPageAllocator::ProtectPages。
这个函数是VirtualProtect的一个封装，在调用VirtualProtect之前对参数进行校验，如下：
检查lpAddress是否是0x1000对齐的；
检查lpAddress是否大于Segment的基址；
检查lpAddress加上dwSize是否小于Segment的基址加上Segment的大小；
检查dwSize是否小于Region的大小；
检查目标内存的访问权限是否等于指定的（通过参数）访问权限；
任何一个检查项未通过，都会调用CustomHeap_BadPageState_fatal_error抛出异常而终止进程

6.3 MS16-063

微软在MS16-063 中发布的补丁：修复 jscript9.dll 与 TypedArray 和 DataView 相关的内存损坏漏洞。
这个漏洞简单来说： 就是先泄漏 vftable 的基地址，构建一个包含里 shellcode 的 rop 链， 用假的返回地址覆盖 vftable 的地址。然后 call Uint8Array.subarray ,就执行了 shellcode
但是在windows10 里面， 调用 vftable 前会检查这个地址是不是合法的，我们之前替换这个地址，所以当然不合法。就会抛出异常。那么绕过的方法：

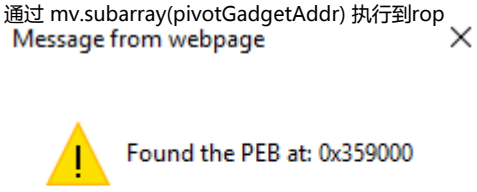
```
CFG ████████████
███████CFG████ ███████ RtlCaptureContext,██████Context████████
██Context████████████████████████████
██ TypedArray ■ 0x7c ■■ RtlCaptureContext ■■■■■ 0x20 ■■ TypedArray ■■■■■■■■■■■■
■■ TypedArray ■■■ RtlCaptureContext ■■■■■■■ RtlCaptureContext ■ ntdll ■■■■■■■■■ ntdll ■■■■■
■■■■■■■■■■■■■■■vtable■■■■■■■■■■jscript9.dll■■■■■■■■■■■■■0x1000■■■■■■■■MZ■■■■■■■ kernelbase.dll ■■■■■■■■■■■■■■■■■■ ker

■■■■■■ ■■ TypedArray ■■ ■■ RtlCaptureContext ■■■ -> Context■■■ -> ■■■■■■■■■
■■ RtlCaptureContext -> ■■ ntdll ■■■■■ -> kernelbase.dll ■■■■■  -> ■■■■■■■  ->  ■■■■■■■■■■■■■
■■■■■■■■■■■■■■■
```

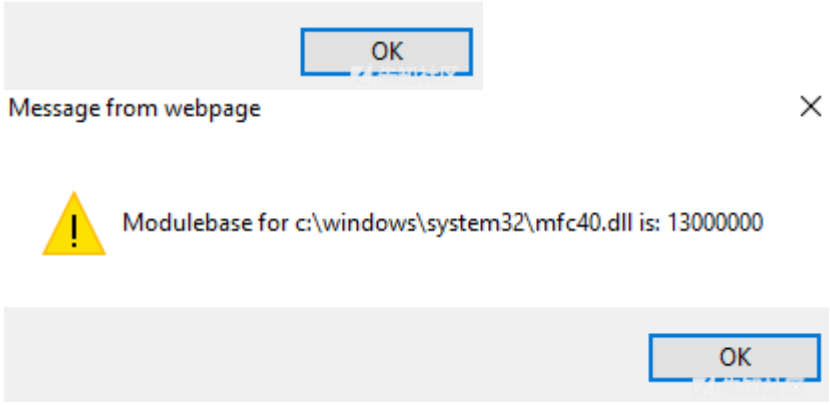
```
var stackAddr = readDWORD(envaddr + 0x9C);

writeDWORD(stackAddr + 0x40, 0x12345678);
writeDWORD(stackAddr + 0x44, 0xABCDEF00);
//overwrite current return addr
alert("Just before overwrite");
writeDWORD(stackAddr + 0x34, 0x11111111); //overwrite retn of write
```


6.4 MFC40.dll

采用 stack pivot 绕过 CFG
先让程序加载没有使用CFG保护的 dll -> MFC40.dll

通过 PEB 去寻找 MFC40.dll 的基地址
构造 rop 链
通过 mv.subarray(pivotGadgetAddr) 执行到rop





7. 参考文献

1.如何绕过Windows 10的CFG机制
http://www.freebuf.com/articles/system/126007.html
2.探索Windows 10的CFG机制
https://www.anquanke.com/post/id/85493
3.使用最新的代码重用攻击绕过执行流保护
https://bbs.pediy.com/thread-217335.htm
4.About CVE-2015-0311
https://www.coresecurity.com/blog/exploiting-cve-2015-0311-part-ii-bypassing-control-flow-guard-on-windows-8-1-update-3
http://www.freebuf.com/vuls/57925.html
5.http://blog.nsfocus.net/win10-cfg-bypass/
6.Bypassing Control Flow Guard in Windows 10
https://improsec.com/blog/bypassing-control-flow-guard-in-windows-10
https://improsec.com/blog/bypassing-control-flow-guard-on-windows-10-part-ii
7.敏感的API绕过CFG
https://blog.trendmicro.com/trendlabs-security-intelligence/control-flow-guard-improvements-windows-10-anniversary-update/