分析及防护：Win10 执行流保护绕过问题

Content

内容摘要

Black Hat USA 2015 正在进行，在微软安全响应中心公布的最新贡献榜单中，绿盟科技安全研究员张云海位列第 6 位，绿盟科技安全团队（NSFST）位列 28 位，绿盟科技安全团队（NSFST）常年致力于发现并解决计算机以及网络系统中存在的各种安全缺陷。这篇《Windows 10 执行流保护绕过问题及修复》是团队在此次大会上分享的主要内容。

1. 1 月 22 日，微软发布 Windows 10 技术预览版，Build 号 9926；
2. 2 月，绿盟科技安全团队（NSFST）展开对其安全机制的研究，发现并与微软一起解决了 CFG 绕过问题；
3. 3 月，微软发布补丁修复了 CFG 绕过问题；
4. 7 月 21 日，在绿盟科技 Techworld 技术大会上分享了此次研究成果；
5. 8 月 7 日，在 Black Hat US 2015 上进行演讲并发布分析文章。

绿盟科技安全团队 NSFST 一直努力发现及修复计算机以及网络系统中存在的各种安全缺陷，如果您需要了解更多信息，请联系：

- 绿盟科技微博
- http://weibo.com/nsfocus
- 绿盟科技微信号
- 搜索公众号 绿盟科技

执行流保护（CFG）

攻击者常常溢出覆盖或者直接篡改寄存器 EIP 的值，篡改间接调用的地址，进而控制了程序的执行流程。执行流保护（CFG，Control Flow Guard）是微软从 Windows 8.1 update 3 及 Windows 10 技术预览版开始，默认启用的一项缓解技术。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。

这种思想及技术在业界有了较为成熟的应用，此次 Windows 10 将其引入，以便提高其安全性。但是绿盟科技安全团队（NSFST）在分析 CFG 的实现机制过程中，发现了 CFG 存在全面绕过的方法，随即向微软提报，并在随后的一段时间内，配合微软修复了这个问题。

CFG 原理

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。

```
0:006> dds jscript9!_load_config_used + 48 l5
62b21048  62f043fc jscript9!__guard_check_icall_fptr  Guard CF Check Function Pointer
62b2104c  00000000                                    Reserved
62b21050  62b2105c jscript9!__guard_fids_table        Guard CF Function Table
62b21054  00001d54                                    Guard CF Function Count
62b21058  00003500                                    Guard Flags
```

同时，编译器还会在所有间接函数调用之前插入一段校验代码，以确保调用的目标地址是预期中的地址。这是未启用 CFG 的情况：

```
jscript9!Js::JavascriptOperators::HasItem+0x15:
66ee9558 8b03            mov     eax,dword ptr [ebx]
66ee955a 8bcb            mov     ecx,ebx
66ee955c 56              push    esi
66ee955d ff507c          call    dword ptr [eax+7Ch]
66ee9560 85c0            test    eax,eax
66ee9562 750b            jne     jscript9!Js::JavascriptOperators::HasItem+0x2c (66ee956f)
```

这是启用 CFG 的情况：

```
jscript9!Js::JavascriptOperators::HasItem+0x1b:
62c31e13 8b03            mov     eax,dword ptr [ebx]
62c31e15 8bfc            mov     edi,esp
62c31e17 52              push    edx
```

```
5   62c31e18 8b707c          mov     esi,dword ptr [eax+7Ch]
6   62c31e1b 8bce            mov     ecx,esi
7   62c31e1d ff15fc43f062    call    dword ptr [jscript9!__guard_check_icall_fptr (62f043fc)]
8   62c31e23 8bcb            mov     ecx,ebx
9   62c31e25 ffd6            call    esi
10  62c31e27 3bfc            cmp     edi,esp
11  62c31e29 0f8514400c00    jne     jscript9!Js::JavascriptOperators::HasItem+0x33 (62cf5e43)
```

操作系统在创建支持 CFG 的进程时，将 CFG Bitmap 映射到其地址空间中，并将其基址保存在 ntdll!LdrSystemDllInitBlock+0x60 中。

CFG Bitmap 是记录了所有有效的间接函数调用目标地址的位图，出于效率方面的考虑，平均每 1 位对应 8 个地址（偶数位对应 1 个 0x10 对齐的地址，奇数位对应剩下的 15 个非 0x10 对齐的地址）。

提取目标地址对应位的过程如下：

- 取目标地址的高 24 位作为索引 i；
- 将 CFG Bitmap 当作 32 位整数的数组，用索引 i 取出一个 32 位整数 bits；
- 取目标地址的第 4 至 8 位作为偏移量 n；
- 如果目标地址不是 0x10 对齐的，则设置 n 的最低位；
- 取 32 位整数 bits 的第 n 位即为目标地址的对应位。

操作系统在加载支持 CFG 的模块时，根据其 Guard CF Function Table 来更新 CFG Bitmap 中该模块所对应的位。同时，将函数指针_guard_check_icall_fptr 初始化为指向 ntdll!LdrpValidateUserCallTarget。

ntdll!LdrpValidateUserCallTarget 从 CFG Bitmap 中取出目标地址所对应的位，根据该位是否设置来判断目标地址是否有效。若目标地址有效，则该函数返回进而执行间接函数调用；否则，该函数将抛出异常而终止当前进程。

```
1   ntdll!LdrpValidateUserCallTarget:
2   774bd970 8b1570e15377    mov     edx,dword ptr [ntdll!LdrSystemDllInitBlock+0x60 (7753e170)]
3   774bd976 8bc1            mov     eax,ecx
4   774bd978 c1e808          shr     eax,8
5   774bd97b 8b1482          mov     edx,dword ptr [edx+eax*4]
6   774bd97e 8bc1            mov     eax,ecx
7   774bd980 c1e803          shr     eax,3
8   774bd983 f6c10f          test    cl,0Fh
9   774bd986 7506            jne     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (774bd98e)
10
11  ntdll!LdrpValidateUserCallTargetBitMapCheck+0xd:
12  774bd988 0fa3c2          bt      edx,eax
13  774bd98b 730a            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0xa (774bd997)
14
15  ntdll!LdrpValidateUserCallTargetBitMapRet:
16  774bd98d c3              ret
```

```
17
18      ntdll!LdrpValidateUserCallTargetBitMapRet+0x1:
19      774bd98e 83c801          or      eax,1
20      774bd991 0fa3c2          bt      edx,eax
21      774bd994 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0xa (774bd997)
22
23      ntdll!LdrpValidateUserCallTargetBitMapRet+0x9:
24      774bd996 c3              ret
```

绕过问题

通过上面的原理分析，我们发现 CFG 的实现中存在一个隐患，校验函数 ntdll!LdrpValidateUserCallTarget 是通过函数指针 _guard_check_icall_fptr 来调用的。

如果我们修改_guard_check_icall_fptr，将其指向一个合适的函数，就可以使任意目标地址通过校验，从而全面的绕过 CFG。通常情况下，_guard_check_icall_fptr 是只读的：

```
1   0:006> x jscript9!___guard_check_icall_fptr
2   62f043fc          jscript9!__guard_check_icall_fptr = <no type information>
3   0:006> !address 62f043fc
4   Usage:                  Image
5   Base Address:           62f04000
6   End Address:            62f06000
7   Region Size:            00002000
8   State:                  00001000    MEM_COMMIT
9   Protect:                00000002    PAGE_READONLY
10  Type:                   01000000    MEM_IMAGE
11  Allocation Base:        62b20000
12  Allocation Protect:     00000080    (null)
13  Image Path:             C:\Windows\System32\jscript9.dll
14  Module Name:            jscript9
15  Loaded Image Name:      C:\Windows\System32\jscript9.dll
16  Mapped Image Name:
```

但如果利用 jscript9 中的 CustomHeap::Heap 对象将其变成可读写的，那么就会出现问题了。

CustomHeap::Heap 对象

CustomHeap::Heap 是 jscript9 中用于管理私有堆的类，其结构如下：

```
CustomHeap::Heap
+0x000  HeapPageAllocator       :   PageAllocator
+0x060  HeapArenaAllocator      :   Ptr32  ArenaAllocator
+0x064  PartialPageBuckets      :   [7]  DListBase<CustomHeap::Page>
+0x09c  FullPageBuckets         :   [7]  DListBase<CustomHeap::Page>
+0x0d4  LargeObjects            :   DListBase<CustomHeap::Page>
+0x0dc  DecommittedBuckets          :   DListBase<CustomHeap::Page>
+0x0e4  DecommittedLargeObjects :   DListBase<CustomHeap::Page>
+0x0ec  CriticalSection         :   LPCRITICAL_SECTION
```

当 CustomHeap::Heap 对象析构时，其析构函数会调用 CustomHeap::Heap::FreeAll 来释放所有分配的内存。

```
int __thiscall CustomHeap::Heap::~Heap(CustomHeap::Heap *this)
{
  CustomHeap::Heap *v1; // esi@1
  v1 = this;
  CustomHeap::Heap::FreeAll(this);
  DeleteCriticalSection((LPCRITICAL_SECTION)((char *)v1 + 0xEC));
  `eh vector destructor iterator'((int)((char *)v1 + 0x9C), 8u, 7, sub_10010390);
  `eh vector destructor iterator'((int)((char *)v1 + 0x64), 8u, 7, sub_10010390);
  return PageAllocator::~PageAllocator(v1);
}
```

CustomHeap::Heap::FreeAll 为每个 Bucket 对象调用 CustomHeap::Heap::FreeBucket。

```
void __thiscall CustomHeap::Heap::FreeAll(CustomHeap::Heap *this)
{
  CustomHeap::Heap *v1; // esi@1
  signed int v2; // ebx@1
  int v3; // edi@1
  int v4; // ecx@2
  v1 = this;
  v2 = 7;
  v3 = (int)((char *)this + 0x9C);
  do
  {
```

```
        CustomHeap::Heap::FreeBucket(v1, v3 - 0x38, (int)this);
        CustomHeap::Heap::FreeBucket(v1, v3, v4);
        v3 += 8;
        --v2;
    }
    while ( v2 );
    CustomHeap::Heap::FreeLargeObject<1>(this);
    CustomHeap::Heap::FreeDecommittedBuckets(v1);
    CustomHeap::Heap::FreeDecommittedLargeObjects(v1);
}
```

CustomHeap::Heap::FreeBucket 遍历 Bucket 的双向链表，为每个节点的 CustomHeap::Page 对象调用 CustomHeap::Heap::EnsurePageReadWrite<1,4>。

```
int __thiscall CustomHeap::Heap::FreeBucket(PageAllocator *this, int a2, int a3)
{
    PageAllocator *v3; // edi@1
    int result; // eax@2
    int v5; // esi@3
    int v6; // [sp+8h] [bp-8h]@1
    int v7; // [sp+Ch] [bp-4h]@1

    v3 = this;
    v6 = a2;
    v7 = a2;
    while ( 1 )
    {
        result = SListBase<Bucket<AddPropertyCacheBucket>,FakeCount>::Iterator::Next(&v6);
        if ( !(_BYTE)result )
            break;
        v5 = v7 + 8;
        CustomHeap::Heap::EnsurePageReadWrite<1,4>(v7 + 8);
        PageAllocator::ReleasePages(v3, *(void **)(v5 + 0xc), 1u, *(struct PageSegment **)(v5 + 4));
        DListBase<CustomHeap::Page>::EditingIterator::RemoveCurrent<ArenaAllocator>(*((ArenaAllocator **)
v3 + 0x18));
    }
    return result;
}
```

CustomHeap::Heap::EnsurePageReadWrite<1,4> 用以下参数调用 VirtualProtect：

- lpAddress: CustomHeap::Page 对象的成员变量 address
- dwSize: 0x1000
- flNewProtect: PAGE_READWRITE

```
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
6    if ( *(_BYTE *)(a1 + 1) || *(_BYTE *)a1 )
7    {
8      result = 0;
9    }
10   else
11   {
12     flOldProtect = 0;
13     VirtualProtect(*(LPVOID *)(a1 + 0xC), 0x1000u, 4u, &flOldProtect);
14     result = flOldProtect;
15     *(_BYTE *)(a1 + 1) = 1;
16   }
17   return result;
18 }
```

将内存页面标记为 PAGE_READWRITE，这正是出现问题的关键地方。

绕过 CFG

通过修改 CustomHeap::Heap 对象，我们可以将一个只读页面变成可读写的，从而可以改写函数指针 _guard_check_icall_fptr 的值。观察 ntdll!LdrpValidateUserCallTarget 在目标地址有效时执行的指令：

```
1  mov     edx,dword ptr [ntdll!LdrSystemDllInitBlock+0x60 (7753e170)]
2  mov     eax,ecx
3  shr     eax,8
4  mov     edx,dword ptr [edx+eax*4]
5  mov     eax,ecx
6  shr     eax,3
7  test    cl,0Fh
8  jne     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (774bd98e)
9  bt      edx,eax
10 jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0xa (774bd997)
```

```
ret
```

从调用者的角度来看，上述指令与单条 ret 指令之间并没有本质区别。因此，将函数指针_guard_check_icall_fptr 改写为指向 ret 指令，就可以使任意的目标地址通过校验，从而全面的绕过 CFG。

问题修复

绿盟科技安全团队（NSFST）发现这一问题后，立即向微软报告了相关情况。微软很快修复了这一问题，并在 2015 年 3 月发布了相关的补丁。在该补丁中，微软引入了一个新的函数 HeapPageAllocator::ProtectPages。

HeapPageAllocator::ProtectPages 函数

```
int __thiscall HeapPageAllocator::ProtectPages(HeapPageAllocator *this, LPCVOID lpAddress, unsigned int a3, struct Segment *a4, DWORD flNewProtect, unsigned __int32 *a6, unsigned __int32 a7)
{
  unsigned __int32 v7; // ebx@1
  unsigned int v8; // edx@2
  int result; // eax@7
  struct _MEMORY_BASIC_INFORMATION Buffer; // [sp+Ch] [bp-20h]@4
  DWORD flOldProtect; // [sp+28h] [bp-4h]@7

  v7 = (unsigned __int32)this;
  if ( (unsigned __int16)lpAddress & 0xFFF
    || (v8 = *((_DWORD *)a4 + 2), (unsigned int)lpAddress < v8)
    || (unsigned int)((char *)lpAddress - v8) > (*((_DWORD *)a4 + 3) - a3) << 12
    || !VirtualQuery(lpAddress, &Buffer, 0x1Cu)
    || Buffer.RegionSize < a3 << 12
    || a7 != Buffer.Protect )
  {
    CustomHeap_BadPageState_fatal_error(v7);
    result = 0;
  }
  else
  {
    *a6 = Buffer.Protect;
    result = VirtualProtect((LPVOID)lpAddress, a3 << 12, flNewProtect, &flOldProtect);
  }
  return result;
}
```

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。

修复机制

CustomHeap::Heap::EnsurePageReadWrite<1,4> 改为调用 HeapPageAllocator::ProtectPages 而不再直接调用 VirtualProtect。

```
1   unsigned __int32 __thiscall CustomHeap::Heap::EnsurePageReadWrite<1,4>(HeapPageAllocator *this, in
    t a2)
2   {
3     unsigned __int32 result; // eax@2
4     unsigned __int32 v3; // [sp+4h] [bp-4h]@5
5
6     if ( *(_BYTE *)(a2 + 1) || *(_BYTE *)a2 )
7     {
8       result = 0;
9     }
10    else
11    {
12      v3 = 0;
13      HeapPageAllocator::ProtectPages(this, *(LPCVOID *)(a2 + 12), 1u, *(struct Segment **)(a2 + 4),
    4u, &v3, 0x10u);
14      result = v3;
15      *(_BYTE *)(a2 + 1) = 1;
16    }
17    return result;
18  }
```

这里参数中指定的访问权限是 PAGE_EXECUTE，从而防止了利用 CustomHeap::Heap 将只读内存页面变成可读写内存页面。

参考文献

[1] MJ0011. Windows 10 Control Flow Guard Internals

http://www.powerofcommunity.net/poc2014/mj0011.pdf

[2] Jack Tang. Exploring Control Flow Guard in Windows 10

http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

[3] Francisco Falcón. Exploiting CVE-2015-0311, Part II: Bypassing Control Flow Guard on Windows 8.1 Update 3

https://blog.coresecurity.com/2015/03/25/exploiting-cve-2015-0311-part-ii-bypassing-control-flow-guard-on-windows-8-1-update-3/

[4] Yuki Chen. The Birth of a Complete IE11 Exploit under the New Exploit Mitigations

https://www.syscan.org/index.php/download/get/aef11ba81927bf9aa02530bab85e303a/SyScan15%20Yuki%20Chen%20-%20The%20Birth%20of%20a%20Complete%20IE11%20Exploit%20Under%20the%20New%20Exploit%20Mitigations.pdf

关于绿盟科技 NSFST

SECURITY

为了更有效的保护计算机系统、消除安全风险，必须对系统本身有深入了解。为此，绿盟安全研究院成立了 NSFOCUS 安全小组（NSFST），由一批高水平的安全专家组成的。绿盟安全研究院一直努力发现及修复计算机以及网络系统中存在的各种安全缺陷，这些缺陷可能被攻击者利用来破坏正常的工作。绿盟安全研究院通过与相关产品厂商合作来解决存在的安全问题，以保护广大用户的合法利益。绿盟安全研究院将研究成果以安全公告或者公开文档的形式发布，以使我们的客户、系统管理员、普通用户、安全研究人员、产品厂商能了解安全缺陷发生的原因、找到解决问题的方法，从而可以从我们的研究成果中受益。

关于绿盟科技

NSFOCUS

北京神州绿盟信息安全科技股份有限公司（简称绿盟科技）成立于 2000 年 4 月，总部位于北京。在国内外设有 30 多个分支机构，为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户，提供具有核心竞争力的安全产品及解决方案，帮助客户实现业务的安全顺畅运行。

基于多年的安全攻防研究，绿盟科技在网络及终端安全、互联网基础安全、合规及安全管理等领域，为客户提供入侵检测/防护、抗拒绝服务攻击、远程安全评估以及 Web 安全防护等产品以及专业安全服务。

北京神州绿盟信息安全科技股份有限公司于 2014 年 1 月 29 日起在深圳证券交易所创业板上市交易，股票简称：绿盟科技，股票代码：300369。