全面绕过执行流保护

安全研究部 张云海

执行流保护（CFG，Control Flow Guard）是微软在 Windows 10 技术预览版与 Windows 8.1 update 3 中，默认启用的一项缓解技术。在分析 CFG 的实现机制的过程中，我们发现了一种全面绕过 CFG 的方法，并配合微软修复了这一问题。

背景

CFG 是微软在 Windows 10 技术预览版与 Windows 8.1 update 3 中默认启用的一项缓解技术。通过在间接跳转前插入校验代码来检查目标地址的有效性，执行流保护可以阻止执行流跳转到预期之外的地点，从而有效的防止修改函数指针来控制 EIP 的漏洞利用技术。

在分析 CFG 的实现机制的过程中，我们发现了一种全面绕过 CFG 的方法，并配合微软修复了这一问题。

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
62c31e18 8b707c        mov     esi,dword ptr [eax+7Ch]
62c31e1b 8bce          mov     ecx,esi
62c31e1d ff15fc43f062  call    dword ptr [jscript9!__guard_check_icall_fptr
(62f043fc)]
62c31e23 8bcb          mov     ecx,ebx
62c31e25 ffd6          call    esi
62c31e27 3bfc          cmp     edi,esp
62c31e29 0f8514400c00  jne     jsc
ript9!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。

```
ntdll!LdrpValidateUserCallTarget:
774bd970 8b1570e15377    mov     edx,dword ptr [ntdll!LdrSystemDllInitBlock+0x60 (7753e170)]
774bd976 8bc1            mov     eax,ecx
774bd978 c1e808          shr     eax,8
774bd97b 8b1482          mov     edx,dword ptr [edx+eax*4]
774bd97e 8bc1            mov     eax,ecx
774bd980 c1e803          shr     eax,3
774bd983 f6c10f          test    cl,0Fh
774bd986 7506            jne     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1 (774bd98e)
ntdll!LdrpValidateUserCallTargetBitMapCheck+0xd:
774bd988 0fa3c2          bt      edx,eax
774bd98b 730a            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0xa (774bd997)
ntdll!LdrpValidateUserCallTargetBitMapRet:
774bd98d c3              ret
ntdll!LdrpValidateUserCallTargetBitMapRet+0x1:
774bd98e 83c801          or      eax,1
774bd991 0fa3c2          bt      edx,eax
774bd994 7301            jae     ntdll!LdrpValidateUserCallTargetBitMapRet+0xa (774bd997)
ntdll!LdrpValidateUserCallTargetBitMapRet+0x9:
774bd996 c3              ret
```

绕过问题

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：

```
0:006> x jscript9!___guard_check_icall_fptr
62f043fc        jscript9!__guard_check_icall_fptr = <no type information>
0:006> !address 62f043fc
Usage:                  Image
Base Address:           62f04000
End Address:            62f06000
Region Size:            00002000
State:                  00001000          MEM_COMMIT
```

```
Protect:              00000002          PAGE_READONLY
Type:                 01000000          MEM_IMAGE
Allocation Base:      62b20000
Allocation Protect:   00000080          (null)
Image Path:           C:\Windows\System32\jscript9.dll
Module Name:          jscript9
Loaded Image Name:    C:\Windows\System32\jscript9.dll
Mapped Image Name:
```

但是，我们可以利用 jscript9 中的 CustomHeap::Heap 对象将其变成可读写的。

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<CustomH
eap::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::He
ap *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::H
eap *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

```
DWORD __stdcall CustomHeap::Heap::EnsurePageReadWrite<1,4>(int a1)
{
DWORD result; // eax@3
DWORD flOldProtect; // [sp+4h] [bp-4h]@3
if ( *(_BYTE *)(a1 + 1) || *(_BYTE *)a1 )
{
result = 0;
}
else
{
flOldProtect = 0;
VirtualProtect(*(LPVOID *)(a1 + 0xC), 0x1000u, 4u, &flOldProtect);
result = flOldProtect;
*(_BYTE *)(a1 + 1) = 1;
}
return result;
}
```

将内存页面标记为 PAGE_READWRITE，这正是我们需要的行为。

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：

```
mov     edx,dword ptr [ntdll!LdrSystemDllInitBlock+0x60(7753e170)]
mov     eax,ecx
shr     eax,8
mov     edx,dword ptr [edx+eax*4]
mov     eax,ecx
shr     eax,3
test    cl,0Fh
jne     ntdll!LdrpValidateUserCallTargetBitMapRet+0x1(774bd98e)
bt      edx,eax
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

我们发现这一问题后，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。

```
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。

CustomHeap::Heap::EnsurePageReadWrite<1,4> 改为调用 HeapPageAllocator::ProtectPages 而不再

直接调用 VirtualProtect。

```
unsigned __int32 __thiscall CustomHeap::Heap::EnsurePageReadWrite<1,4>(HeapPageAllocator *this, int a2)
{
unsigned __int32 result; // eax@2
unsigned __int32 v3; // [sp+4h] [bp-4h]@5

if ( *(_BYTE *)(a2 + 1) || *(_BYTE *)a2 )
{
result = 0;
}
else
{
v3 = 0;
HeapPageAllocator::ProtectPages(this, *(LPCVOID *)(a2 + 12), 1u, *(struct Segment **)(a2 + 4), 4u, &v3, 0x10u);
result = v3;
*(_BYTE *)(a2 + 1) = 1;
}
return result;
}
```

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