BYPASS CONTROL FLOW GUARD

COMPREHENSIVELY

Zhang Yunhai

Overview

Control Flow Guard (CFG) is a mitigation that prevents redirecting control flow to unexpected location.

It was first introduced in Windows 8.1 Preview, but disabled in Windows 8.1 RTM for compatibility reason. Then, it was improved and enabled in Windows 10 Technical Preview and Windows 8.1 Update.

CFG operates by injecting a check before every indirect call to ensure that the target address is valid. If the check fail at runtime on a CFG-aware operating system, the operating system close the program. Therefore, CFG can mitigate quite a lot common exploit techniques, which overwrite a pointer to redirect control flow.

This article introduce a universal bypass technique, which bypass CFG protection comprehensively, and thus make those mitigated techniques exploitable again.

CFG Internals

MJ0011 [2] and Jack Tang [3] have analyzed CFG implementation in detail. The following is a brief description.

Compile Stage

The compiler append five Load Configuration Table entries.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:0060</td>
<td>dds jscript9!_load_config_used + 48</td>
<td>Guard CF Check Function Pointer</td>
</tr>
<tr>
<td>62b21048</td>
<td>62f043fc jscript9!_guard_check_icall_fptr</td>
<td>Reserved</td>
</tr>
<tr>
<td>62b2104c</td>
<td>00000000</td>
<td>Guard CF Function Table</td>
</tr>
<tr>
<td>62b21050</td>
<td>62b2105c jscript9!_guard_fids_table</td>
<td>Guard CF Function Count</td>
</tr>
<tr>
<td>62b21054</td>
<td>0001d54</td>
<td>Guard Flags</td>
</tr>
<tr>
<td>62b21058</td>
<td>00003500</td>
<td></td>
</tr>
</tbody>
</table>

The Guard CF Check Function Pointer entry store the address of the check function pointer, which point to the check function ntdll!LdrpValidateUserCallTarget.
The Guard CF Function Table and Guard CF Function Count entries locate the Guard CF Function Table, which is a RVA list of valid target address.

```
0:017> dd jscript9!__guard_fids_table
62b2105c 00009330 00009350 00009450 000094e0
62b2106c 0000a270 0000a2f0 0000a770 0000a7e0
62b2107c 0000a7f0 0000a810 0000a860 0000a890
62b2108c 0000a9b0 0000d240 0000d2b0
62b2109c 000102a0 00010350 00010390 00010460
62b210ac 00010710 000107c0 00010810 00010860
62b210bc 000109b0 00010a70 00010b20 00010b30
62b210cc 00011250 000112c0 00011910 00011930
```

The compiler also inject a check before every indirect call to ensure that the target address is valid.

```
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)

Indirect call without CFG
```

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

CFG use a Bitmap to track all valid target address.

CFG-aware operating system create a section object for the CFG Bitmap when it is booting.

When creating a process, the operating system map the CFG Bitmap into process memory address space, and store the address and size in ntdll!LdrSystemDllInitBlock at offset 0x60 and 0x68.

!address

When loading a CFG enabled module, the operating system update the CFG Bitmap according to the Guard CFG Function Table.

!security_check_cookie

js9!__security_check_cookie = <no type information>
The check function pointer is pointed to ntdll!LdrpValidateUserCallTarget at the same time.

Runtime

Therefore, ntdll!LdrpValidateUserCallTarget is called before indirect call to verify the target address at runtime.

It test a bit of the CFG Bitmap that correspond to the target address as follows:

- Extract the highest 24 bit of the target address to form an index
- Fetch a 32-bit DWORD from the CFG Bitmap using the index
- Extract the 4th to 8th bits of the target address to form an offset n
- Set the lowest bit of offset n if the target address is not 0x10 aligned
- Test the n\textsuperscript{th} bit of the 32-bit DWORD

If the bit is not set, which means that the target address is invalid, it call ntdll!RtlpHandleInvalidUserCallTarget. Otherwise, it return to execute the indirect call.

ntdll!RtlpHandleInvalidUserCallTarget raise interrupt 0x29, unless special conditions are met, as follows:

- Call ntdll!NtQueryInformationProcess to fetch ProcessExecuteFlags
- If MEM_EXECUTE_OPTION_DISABLE is set, return
- If MEM_EXECUTE_OPTION_ENABLE is not set, raise interrupt 0x29
- If MEM_EXECUTE_OPTION_ATL7_THUNK_EMULATION is set, raise interrupt 0x29
- Call ntdll!NtQueryVirtualMemory to fetch target address’s Protect
- If target address is not executable, raise interrupt 0x29
- Otherwise, return to execute the indirect call

Attack surface

Non-CFG Module

Non-CFG module affect CFG in two aspects.

First, it may contain unprotected indirect call, whose target can be overwrite to redirect control flow.
Secondly, all its corresponding bits in the CFG Bitmap are set. This means that any address inside its .text section is a valid indirect call target. Therefore, protected indirect call target can be overwrite with these address.

However, non-CFG module will exhaust eventually, since vendors usually trend to compile new modules with CFG enable.

**JIT Generated Code**

JIT generated code is just like a non-CFG module. It also may contain unprotected indirect call, and all its corresponding bits in the CFG Bitmap are set.

Francisco Falcón [4] show how to use an unprotected indirect call in Flash JIT generated code to bypass CFG.

Note that both are no longer the case in the latest version of Edge, JIT code is instrumented, and JIT code pages don't have all bits set.

**Indirect Jump**

Indirect jump also redirect control flow like indirect call, and it is not protected by CFG in most case.

Yuki Chen [5] show a controllable indirect jump, as follows:

```
jscript9!NativeCodeGenerator::CheckCodeGenThunk:  
62b3c5e2 55              push    ebp  
62b3c5e3 8bec            mov     ebp,esp  
62b3c5e5 ff742408        push    dword ptr [esp+8]  
62b3c5e9 e812ffffff      call    jscript9!NativeCodeGenerator::CheckCodeGen (62b3c500)  
62b3c5ee 5d              pop     ebp  
```
Indirect jump can be protected using the same mechanism as indirect call. The forementioned one is protected in the latest version of Edge.

Return Address

Return address can be overwritten to redirect control flow too, as stack buffer overflow has shown.

Mitigations such as GS, SAFESEH, and SEHOP have prevent stack buffer overflow. However, it is still possible to overwrite the return address directly with the capability of arbitrary address read and write.

- First, we need to locate the stack. Yuki Chen [5] discuss several way to archive this.
- Then, we search the stack for an appropriate frame.
- Finally, we replace the stack frame with crafted one.

Unlike indirect jump, return address cannot be protected using the same mechanism as indirect call.

Valid API Function

Yang Yu [1] show how to use ntdll!NtContinue to bypass DEP & ASLR, which is a valid indirect call target.

There are some similar API functions, as follows:

- KERNELBASE!SetThreadContext
- msvcr71!longjmp
- KERNEL32!WinExec
- SHELL32!ShellExecuteExA
- KERNEL32!LoadLibraryA

Note that three of them, those will modify the context, are no longer valid in the latest version of Windows 10.
Universal Bypass

Objective

The objective of Universal Bypass is to bypass CFG comprehensively, thus allow overwriting protected indirect call target with any address.

The ideal way is to let any address pass Guard CF Check Function.

Overwrite Guard CF Check Function Pointer

Guard CF Check Function is called through Guard CF Check Function Pointer, which is initialized with the address of ntdll!LdrpValidateUserCallTarget when a module is loaded.

If we overwrite Guard CF Check Function Pointer with the address of a special function, which always behave as what ntdll!LdrpValidateUserCallTarget will do for valid target address, any address will pass Guard CF Check Function. The question is where is such a function?

Review the instructions that is executed in ntdll!LdrpValidateUserCallTarget when the target address is valid.

From the aspect of caller, there is no difference between these instructions and a single ret instruction.

Therefore, overwrite Guard CF Check Function Pointer with the address of a ret instruction will let any address pass Guard CF Check Function, and thus bypass CFG.

However, Guard CF Check Function Pointer is read-only.

0:006> x jscript9!__guard_check_icall_fptr
62f043fc  jscript9!__guard_check_icall_fptr = <no type information>
Therefore, we need to make it writeable before overwrite it.

Make Read-only Memory Writeable

We archive this through JScript9 CustomHeap::Heap.

CustomHeap::Heap

Offset 0x64 – 0x9c are 7 Buckets for partially used Page.
Offset 0x9c – 0xd4 are 7 Buckets for fully used Page.

Each Bucket is a double linked list of CustomHeap::Page.

CustomHeap::Page

Offset 0x00 – 0x04 are 7 Buckets for partially used Page.
Offset 0x04 – 0x08 are 7 Buckets for fully used Page.

Each Bucket is a double linked list of CustomHeap::Page.
**Destructor Behavior**

When `CustomHeap::Heap` is destructed, the destructor call `CustomHeap::Heap::FreeAll` to free all the memory it has allocated.

```cpp
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` call `CustomHeap::Heap::FreeBucket` for each Bucket to free it.

```cpp
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` traverse the double linked list, and call `CustomHeap::Heap::EnsurePageReadWrite<1,4>` for each `CustomHeap::Page`.

```cpp
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]01
int v7; // [sp+Ch] [bp-4h]01

v3 = this;
v6 = a2;
v7 = a2;
while ( 1 )
{
    result = SLListBase<AddPropertyCacheBucket,FakeCount>::Iterator::Next(&v6);
    if ( !(BYTE)result )
        break;
    v5 = v7 + 8;
    CustomHeap::Heap::EnsurePageWrite<1,4>(v7 + 8);
    PageAllocator::ReleasePages(v3, *(void **)(v5 + 0xc), 1u, *(struct PageSegment **)(v5 + 4));
}
return result;

CustomHeap::Heap::EnsurePageWrite<1,4> call VirtualProtect with the following arguments:
- lpAddress: CustomHeap::Page address
- dwSize: 0x1000
- flNewProtect: PAGE_READWRITE

DWORD __stdcall CustomHeap::Heap::EnsurePageWrite<1,4>(int a1)
{
    DWORD result; // eax@3
    DWORD f101dProtect; // [sp+4h] [bp-4h]03

    if ( !(BYTE *)(a1 + 1) || *(BYTE *)a1 )
    {
        result = 0;
    }
    else
    {
        f101dProtect = 0;
        VirtualProtect(*(LPVOID *)(a1 + 0xC), 0x1000u, 4u, &f101dProtect);
        result = f101dProtect;
        *(BYTE *)(a1 + 1) = 1;
Locate the Heap

CustomHeap::Heap is a member of InterpreterThunkEmitter at offset 0xc.

```plaintext
0:018> dd 0441d380 l4
0441d380 00000000 0c6cdd28 00000000 679521d8
0:018> dds 0441d380 + c l1
0441d38c 679521d8 jscript9!HeapPageAllocator::`vftable'
```

InterpreterThunkEmitter is pointed by a member of Js::ScriptContext at offset 0x4b0.

```plaintext
0:018> dd 0c6cdb80 + 4b0 l4
0c6ce030 0441d380 0c66e860 00000000 00000000
```

Js::ScriptContext is pointed by a member of ScriptEngine at offset 0x4.

```plaintext
0:018> dds 0441d138 l1
0441d138 6794a5f4 jscript9!ScriptEngine::`vftable'
0:018> dd 0441d138 l4
0441d138 6794a5f4 0c6cdb80 00000009 043591a8
```

ScriptEngine is pointed by a member of ScriptSite at offset 0x4.

```plaintext
0:018> dd 0c629d18 14
0c629d18 00000003 0441d138 0441d138 00000000
```

Decommit Issue

Normally, when CustomHeap::Heap::~Heap is called, all Buckets are empty, thus no VirtualProtect is called as expect.

This is because all CustomHeap::Page in Buckets are decommitted in Js::ScriptContext::Close. Decommitted CustomHeap::Page is removed from Bucket. And Js::ScriptContext::Close is called before CustomHeap::Heap::~Heap in Js::ScriptContext::~ScriptContext.

There are two ways to resolve this issue: insert a fake CustomHeap::Page object into the Bucket, or modify CustomHeap::Heap to prevent a CustomHeap::Page from being decommitted.

Fix for the Issue

Microsoft fix this issue soon after it is reported, and release the patch in 2015 March.

The patch introduce a new function HeapPageAllocator::ProtectPages.
HeapPageAllocator::ProtectPages is a wrapper of VirtualProtect with the following check:

- lpAddress is 0x1000 aligned
- lpAddress is not less than Segment address
- lpAddress plus dwSize is not greater than Segment address plus Segment size
- dwSize is not greater than RegionSize
- Protect equal to the expected one

If any check fail, it calls CustomHeap_BadPageState_fatal_error to raise an exception.

CustomHeap::Heap::EnsurePageReadWrite<1,4> call HeapPageAllocator::ProtectPages instead of VirtualProtect, and the required Protect is PAGE_EXECUTE.
if ( *(BYTE *)(a2 + 1) || *(BYTE *)a2 )
{
    result = 0;
}
else
{
    v3 = 0;
    HeapPageAllocator::ProtectPages(this, *(LPVOID *)(a2 + 12), 1u, *(struct Segment **)(a2 + 4), 4u, &v3, 0x10u);
    result = v3;
    *(BYTE *)(a2 + 1) = 1;
}
return result;

Therefore, we can not make read-only memory writeable any more, since it will fail the Protect check.

Conclusion

Jscript9 CustomHeap::Heap can be used to make read-only memory writeable.

Therefore, we can overwrite Guard CF Check Function Pointer to bypass CFG.

Microsoft fix this issue soon after it is reported.

References

[1] Yang Yu. WRITE ONCE, PWN ANYWHERE