# 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.

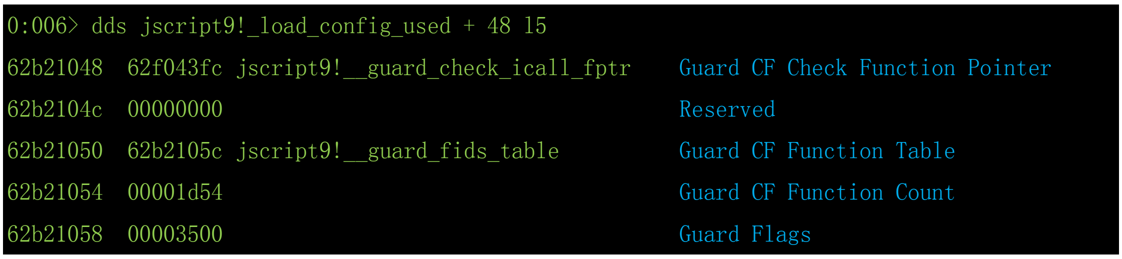
This article introduce a universal bypass technique, which bypass CFG protection comprehensively, and thus make those mitigated techniques exploitable again.

## 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.



```
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
```

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 00009360 00009450 000094e0
62b2106c  0000a270 0000a2f0 0000a770 0000a7e0
62b2107c  0000a7f0 0000a810 0000a860 0000aa70
62b2108c  0000ab40 0000ab90 0000d240 0000d2b0
62b2109c  000102a0 00010350 00010390 00010460
62b210ac  00010710 000107a0 00010980 00010be0
62b210bc  00010c50 00010ce0 00010dc0 000110c0
62b210cc  00011250 000112c0 00011910 00011930
0:006> lmm jscript9
start    end         module name
62b20000 62f42000    jscript9   (private pdb symbols)
0:006> ln 62b20000 + 00009330
(62b29330)   jscript9!__security_check_cookie   |  (62b2933d)   jscript9!_SEH_epilog4
Exact matches:
0:006> ln 62b20000 + 00009360
(62b29360)   jscript9!_EH_epilog3   |  (62b2937b)   jscript9!_EH_prolog3
Exact matches:
0:006> ln 62b20000 + 00009450
(62b29450)   jscript9!memcmp   |  (62b29458)   jscript9!IID_IUnknown
Exact matches:
0:017> ln 62b20000 + 000094e0
(62b294e0)   jscript9!_DllMainCRTStartup   |  (62b294f5)   jscript9!__DllMainCRTStartup
Exact matches:
```

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
```

```
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     jscript9!Js::JavascriptOperators::HasItem+0x33 (62cf5e43)
```

**Indirect call with CFG enabled**

## 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.

```
0:017> dd ntdll!LdrSystemDllInitBlock+0x60 l1
77b4e170  00920000      address
0:006> dd ntdll!LdrSystemDllInitBlock+0x68 l1
77b4e178  02000000      size
0:017> !address
    ...
+    920000   941000    21000 MEM_MAPPED  MEM_RESERVE                 MappedFile "PageFile"
     941000   945000     4000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY   MappedFile "PageFile"
     945000   aad000   168000 MEM_MAPPED  MEM_RESERVE                 MappedFile "PageFile"
     aad000   aaf000     2000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY   MappedFile "PageFile"
     aaf000   b6b000    bc000 MEM_MAPPED  MEM_RESERVE                 MappedFile "PageFile"
     b6b000   b6c000     1000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY   MappedFile "PageFile"
     b6c000  1e0e000  12a2000 MEM_MAPPED  MEM_RESERVE                 MappedFile "PageFile"
    1e0e000  1fda000   1cc000 MEM_MAPPED  MEM_COMMIT  PAGE_NOACCESS   MappedFile "PageFile"
    1fda000  2073000    99000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY   MappedFile "PageFile"
    ...
    269f000  26a0000     1000 MEM_MAPPED  MEM_COMMIT  PAGE_NOACCESS   MappedFile "PageFile"
    26a0000  270f000    6f000 MEM_MAPPED  MEM_COMMIT  PAGE_READONLY   MappedFile "PageFile"
    270f000  2920000   211000 MEM_MAPPED  MEM_RESERVE                 MappedFile "PageFile"
    ...
```

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

```
0:006> x jscript9!__security_check_cookie
62b29330          jscript9!__security_check_cookie = <no type information>
```

```
0:006> dd 00920000 + 62b293 * 4 l1
021cca4c  00001040
```

The check function pointer is pointed to ntdll!LdrpValidateUserCallTarget at the same time.

```
0:017> dds jscript9!__guard_check_icall_fptr l1
62f043fc  77acd970 ntdll!LdrpValidateUserCallTarget
```

### 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 nth 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.

```
0:006> u slc!SLConsumeWindowsRight+0xe3
slc!SLConsumeWindowsRight+0xe3:
73127463 8b06            mov     eax,dword ptr [esi]
```

```
73127465 56               push    esi
73127466 ff5008           call    dword ptr [eax+8]
```

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.

```
0:006> lmm slc
start    end        module name
73110000 73138000   slc        (deferred)
0:006> dd poi(ntdll!LdrSystemDllInitBlock+0x60) + 731100 * 4
025e4400  ffffffff ffffffff ffffffff ffffffff
025e4410  ffffffff ffffffff ffffffff ffffffff
025e4420  ffffffff ffffffff ffffffff ffffffff
025e4430  ffffffff ffffffff ffffffff ffffffff
025e4440  ffffffff ffffffff ffffffff ffffffff
025e4450  ffffffff ffffffff ffffffff ffffffff
025e4460  ffffffff ffffffff ffffffff ffffffff
025e4470  ffffffff ffffffff ffffffff ffffffff
```

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
```

```
62b3c5ef ffe0            jmp     eax
```

Indirect jump can be protected using the same mechanism as indirect call. The fore mentioned one is protected in the latest version of Edge.

```
chakra!NativeCodeGenerator::CheckCodeGenThunk:
621b4020 55              push    ebp
621b4021 8bec            mov     ebp,esp
621b4023 ff742408        push    dword ptr [esp+8]
621b4027 e8c4b4f6ff      call    chakra!NativeCodeGenerator::CheckCodeGen (6211f4f0)
621b402c 50              push    eax
621b402d 8bc8            mov     ecx,eax
621b402f ff1504154f62    call    dword ptr [chakra!__guard_check_icall_fptr (624f1504)]
621b4035 58              pop     eax
621b4036 5d              pop     ebp
621b4037 ffe0            jmp     eax
```

## 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
- msvcrt!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.

```
62c31e18 8b707c          mov     esi,dword ptr [eax+7Ch]
62c31e1b 8bce            mov     ecx,esi
62c31e1d ff15fc43f062    call    dword ptr [jscript9!__guard_check_icall_fptr (62f043fc)]
```

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.

```
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
```

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>
```

```
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:
```

Therefore, we need to make it writeable before overwrite it.

## Make Read-only Memory Writeable

We archive this through JScript9 CustomHeap::Heap.

### CustomHeap::Heap

```
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
```

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
+0x000   Flag          :   Uint4B
+0x004   Segment       :   Ptr32 PageSegment
+0x008   Bitmap        :   Uint4B
+0x00c   Address       :   Ptr32
+0x010   BucketID      :   Uint4B
```

## Destructor Behavior

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

```
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.

```
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.

```
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>(*((ArenaAllocato
r **)v3 + 0x18));
  }
  return result;
}
```

CustomHeap::Heap::EnsurePageReadWrite<1,4> call VirtualProtect with the following arguments:

- lpAddress: CustomHeap::Page 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;
}
```

## Locate the Heap

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

```
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.

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

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

```
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.

```
0:018> dd 0c629d18 l4
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.

```
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;
}
```

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.

```
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;
}
```

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

https://www.blackhat.com/docs/us-14/materials/us-14-Yu-Write-Once-Pwn-Anywhere-wp.pdf

[2] MJ0011. Windows 10 Control Flow Guard Internals

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

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

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

[4] 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/

[5] 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