The Power of Data-Oriented Attacks: Bypassing Memory Mitigation Using Data-Only Exploitation Technique
Part II

Bing Sun, Chong Xu
About Speaker

• Bing Sun
  • Bing Sun is a senior information security researcher, and leads the IPS security research team of McAfee. He has extensive experiences in operating system kernel layer and information security R&D, with especially deep diving in advanced vulnerability exploitation and detection, Rootkits detection, firmware security, and virtualization technology.

• Chong Xu
  • Chong Xu received his PhD degree from Duke University with networking and security focus. He is currently a director leading McAfee Labs IPS team, which leads the McAfee Labs vulnerability research, malware and APT detection, botnet detection, and feeds security content and advanced detection features to McAfee's network IPS, host IPS, and firewall products, as well as global threat intelligence.
Abstract

• As Control Flow Integrity (CFI) enforcement solutions are widely adopted by major applications, traditional memory vulnerability exploitation techniques aiming to hijack the control flow have become increasingly difficult. For example, Microsoft’s Control Flow Guard (CFG) is an effective CFI solution against traditional memory exploits. However, due to the CFG implementation limitations, we have seen new exploitation techniques such as using the out-of-context function call to bypass CFG. We believe eventually these limitations could all be overcome or improved, and ultimately we expect a fine-grained CFG solution to completely defeat control-flow hijacking. Consequently, attackers have begun to seek alternatives to exploit memory vulnerabilities without diverting the control flow. As a result of this trend, the data-oriented attacks have emerged. As its name suggests, a data-oriented attack focuses on altering or forging the critical data of an application, rather than attempting to alter its control flow. The data-oriented attack may allow the attacker to do some powerful things, such as loading certain unwanted or disabled modules or changing the attributes of certain memory pages. Sometimes this can be achieved by changing only a few bits of data. Today, most successful memory exploits can gain some level of memory read/write primitives during exploitation of memory corruption vulnerability, which makes data-oriented attacks possible. In this talk, we will present some interesting examples that show the power of data-oriented attacks. We then discuss ways to prevent such attacks. We conclude by live demonstrations of CFG/DEP bypass on Windows 10’s Edge using data-only exploitation technique.
Agenda

• The Fundamental of CFG
• Known CFG Bypass Methods
  • Corrupt function’s return address on stack
  • Transit via unguarded trampoline code
  • Call function out-of-context
• Call Function Out-of-Context
  • Bypass CFG by calling ntdll!LdrResolveDelayLoadsFromDII
• Data-only Attack
  • Leak stack address from LdrpWork structure
  • Bypass CFG/DEP by abusing LdrpWork mechanism
• Suggestions for Preventing Data-only Attack
• Conclusion
• Acknowledgement
The Fundamental of CFG

• About CFG (Control Flow Guard)
  • A compiler-aided exploitation mitigation mechanism that prevents exploit from hijacking the control flow.
  • Compiler inserts CFG check before each indirect control transfer instruction (call/jmp), and at runtime the CFG check will validate the call target address against a pre-configured CFG bitmap to determine whether the call target is valid or not. The process will be terminated upon an unexpected call target being identified.
  • The Relative Virtual Address (RVA) of all valid call targets determined at the time of compilation are kept in a Guard CF Function table in PE file. During the PE loading process, the loader will read CF info from guard CF function table and update the CFG bitmap.
  • The read-only CFG bitmap is maintained by the OS, and part of the bitmap is shared by all processes. An even bit in CFG bitmap corresponds to one 16-bytes aligned address, while an odd bit corresponds to 15 non 16-bytes aligned addresses.
  • When the PE file is loaded, \texttt{\_\_guard\_check\_icall\_fp} will be resolved to point to \texttt{ntdll\!LdrpValidateUserCallTarget}. (on x64, \texttt{\_\_guard\_dispatch\_icall\_fp} -> \texttt{ntdll\!LdrpDispatchUserCallTarget})
The Fundamental of CFG (Continued)

Compiler inserts a call target check before each indirect function call/jmp

CFG bitmap base

High 55-bit of call target address is used as an index into the bitmap to get a 64-bit bitmap entry

Bit 3 ~ 8 of target address is used as an offset

Test the bit "offset" of that 64-bit bitmap entry. Target address is valid if bit is set, otherwise trigger INT 29h

Compiler inserts a call target check before each indirect function call/jmp

CFG bitmap base

High 55-bit of call target address is used as an index into the bitmap to get a 64-bit bitmap entry

Bit 3 ~ 8 of target address is used as an offset

Test the bit "offset" of that 64-bit bitmap entry. Target address is valid if bit is set, otherwise trigger INT 29h

Non 16-byte aligned, set bit 0 of offset
Known CFG Bypass Methods

• Corrupt function’s return address on stack
  • “Bypassing Control Flow Guard in Windows 10”
  • Mitigation: RFG (Return Flow Guard), Intel’s CET

• Transit via unguarded trampoline code (mostly involving dynamic code, such as JIT)
  • “Use Chakra engine again to bypass CFG”
  • “Chakra JIT CFG Bypass”
  • Mitigation: JIT security improvement (JIT code checksum, remote JIT etc)

• Call function out-of-context
  • “Bypass Control Flow Guard Comprehensively”
  • “Mitigation bounty — 4 techniques to bypass mitigations”
  • Mitigation: Fine-grained CFI (improvement on CFG after WIP build 14986)
Call Function Out-of-context

• The basic idea of calling function out-of-context
  • Issue a function call to certain unexpected target via memory indirect call instruction; however from the program’s logic perspective such a call is not supposed to happen from that call site. This is essentially one type of execution control hijacking.

• How to make an out-of-context call
  • Overwrite an existing function pointer (such as in an object’s vftable) with the target function of out-of-context call.
  • The target function needs to be able to pass the CFG check because almost all memory indirect calls have CFG check prior to them.
  • The call to the overwritten function can be reliably and repeatedly triggered from the context of scripting language.
  • The number and order of arguments to the target function should be (at least partially) controllable to the scripting language.
  • It’s preferable to be able to get the target function’s return value in its original form.

• Example
  • “From read-write anywhere to controllable calls”. This is a very good example of calling function out-of-context, controlling all arguments and getting the return value back.
The Delay Load Import Mechanism

- The Delay Load Import mechanism enables fast loading of a module by allowing its imported DLL to be loaded only on the first call to a function in that DLL.

- When the loading of an imported DLL is delayed, the imported functions in that DLL are temporarily pointing to some stub functions. Later when such an imported function is called, its stub function will use a helper function (__delayLoadHelper2) to load the imported DLL, and resolve the imported function to its real entry point at run time. Afterwards, the call to the imported function will happen naturally, without the intervention of stub and helper function.

- __delayLoadHelper2 calls the native API ntdll!ResolveDelayLoadedAPI to do the actual job. In order for ntdll!ResolveDelayLoadedAPI to know how to perform the delay load import, a critical data structure (DELAY_IMPORT_DESCRIPTOR) needs to be passed to it in the 2nd argument.
Example of Delay Load Import

Disassembly:

Offset: @$scopeip

WININET\_tailMerge\_bcrypt\_dll
WININET\_CreateFileWithRandomFilename+0x81
WININET\_CreateUniqueCacheFile+0x289
WININET\_CCacheClientFileManager::CreateUniqueFile+0xc8
WININET\_CCacheClientContainer::CreateUniqueFile+0x65

```
6a0fccb2 b824c21d6a mov eax.offset WININET\_imp\_nthohl (6a1dc224)
6a0fcb7 e96cfeffff jmp WININET\_tailMerge\_WS2\_32\_dll (6a0fcb28)
WININET\_imp\_load\_CertVerifyCertificateChainPolicy:
6a0fccc6 b818c01d6a mov eax.offset WININET\_imp\_CertVerifyCertificateChainPolicy (6a1dc018)
6a0fccc1 e996dfff jmp WININET\_tailMerge\_CRYPT32\_dll (6a0fca5c)
WININET\_imp\_load\_BCryptCreateHash:
6a0fccc5 b824c41d6a mov eax.offset WININET\_imp\_BCryptCreateHash (6a1dc42c)
6a0fccc6 e900000000 jmp WININET\_tailMerge\_bcrypt\_dll (6a0fcc0)
WININET\_tailMerge\_bcrypt\_dll:
6a0fced0 51 push ecx
6a0fcc01 52 push edx
6a0fcc02 50 push eax
6a0fcc03 68c0361c6a push offset WININET\_DELAY\_IMPORT\_DESCRIPTOR\_bcrypt\_dll (6a1c36c0)
6a0fced8 e8cd361fff call WININET\_delayLoadHelper2 (6a0f03aa)
6a0fced5 5a pop edx
6a0fcc0e 59 pop ecx
6a0fcedf ffe0 jmp eax

Command:
```
0:008> u poi(WININET\_imp\_BCryptGenRandom) L2
WININET\_imp\_load\_BCryptGenRandom:
6a0fced8 b824c41d6a mov eax.offset WININET\_imp\_BCryptGenRandom (6a1dc424)
6a0fcedd e9eefeffff jmp WININET\_tailMerge\_bcrypt\_dll (6a0fcc0)
```
Example of Delay Load Import (Continued)

```
WININET!_tailMerge bcrypt_dll+0xf
WININET!_CreateFileWithRandomFilename+0x81
WININET!_CreateUniqueCacheFile+0x289
WININET!CCacheClientFileManager::CreateUniqueFile+0xc8
WININET!CCacheClientContainer::CreateUniqueFile+0x65

Disassembly
Offset: @$scopeip

WININET!_tailMerge bcrypt_dll:
6a0fcd0 51 push ecx
6a0fcd1 52 push edx
6a0fcd2 50 push eax
6a0fcd3 680361c6a push offset WININET!_DELAY_IMPORT_DESCRIPTOR_bcrypt_dll (6a1c36c0)
6a0fcd8 e8cd36ffff call WININET!__delayLoadHelper2 (6a0f03aa)
6a0fcd9 5a pop edx
6a0fcdde 59 pop ecx
6a0fcdff ffe0 jmp eax {bcrypt!BCryptGenRandom (745b5160)}
WININET!_imp_load_BCryptHashData:
6a0fccc1 b834c41d6a mov eax, offset WININET!_imp_BCryptHashData (6a1dc434)
6a0fccc6 e9f5ffff jmp WININET!_tailMerge bcrypt_dll (6a0fcd0)
WININET!_imp_load_BCryptFinishHash:
6a0fccb b830c41d6a mov eax, offset WININET!_imp_BCryptFinishHash (6a1dc430)
6a0fccf 99dfbfff jmp WININET!_tailMerge bcrypt_dll (6a0fcd0)
WININET!_imp_load_BCryptDestroyHash:
6a0fccf5 b820c41d6a mov eax, offset WININET!_imp_BCryptDestroyHash (6a1dc420)

Command
0:008> u poi(WININET!_imp__BCryptGenRandom) L2
bcrypt!BCryptGenRandom:
745b5160 8bdf mov edi,edi
745b5162 55 push ebp
```

Real entry point
Bypass CFG By Calling `ntdll!LdrResolveDelayLoadsFromDll`

- `ntdll!LdrResolveDelayLoadedAPI` and `ntdll!LdrResolveDelayLoadsFromDll` can be leveraged to overwrite any read-only function pointer, and both are valid for CFG (in WIP build 14986 and before). In terms of making out-of-context function call, the latter is much easier as it has fewer arguments to control.

- A fake PE carrying a malformed `DELAY_IMPORT_DESCRIPTOR` structure is created in memory and fed to `ntdll!LdrResolveDelayLoadsFromDll`. The `ImportAddressTableRVA` field of delay import descriptor points to the function pointer to be overwritten (could be outside the fake PE).

- In order for the operating system’s module loader to believe the fake PE is a loaded module, some existing entry in `LDR_DATA_TABLE` needs to be corrupted (and will be restored later). In addition, the `SizeOfImage` field of corrupted entry needs to be set big enough to pass the range check.
The Diagram of CFG Bypass Scheme

- Fake PE
  - DELAYLOAD DESCRIPTOR
  - ImportNameTableRVA
  - ImportAddressTableRVA

- Corrupted _LDR_DATA_TABLE_ENTRY
  - DllBase: base address of Fake PE
  - SizeOfImage: 0x7fffffff

- ntdll.dll
  - Export table NtCurrentTeb

- chakra.dll
  - Import table __guard_check_icall_fptr
The Original Call Stack of array.push Method
Call ntdll!LdrResolveDelayLoadsFromDll
Out-of-context

out-of-context call via
chakraJs::JavascriptArray::EntryPush

1st argument: fake PE
2nd argument: DLL name
3rd argument: 0
ntdll!LdrResolveDelayLoadsFromDll

Native API ntdll!LdrResolveDelayLoadsFromDll

.text:6A2CB470 ; int __stdcall LdrResolveDelayLoadsFromDll(PVOID ImageBase, int, int)
...
.text:6A2CB475  cmp   [ebp+arg_8], 0
.text:6A2CB479  jz    short loc_6A2CB482  // The 3\textsuperscript{rd} argument must be 0
...
.text:6A2CB482  mov   edx, [ebp+arg_4]
.text:6A2CB485  mov   ecx, [ebp+ImageBase] ; ImageBase
.text:6A2CB488  call  _LdrpGetDelayloadDescriptor@8 ; LdrpGetDelayloadDescriptor(x,x)  // Get delay load descriptor from the fake PE
.text:6A2CB48D  test  eax, eax  // return delay load descriptor
.text:6A2CB498  jnz   short loc_6A2CB498
...
.text:6A2CB498  mov   ecx, [ebp+ImageBase]  // fake PE
.text:6A2CB49B  mov   edx, eax  // delay load descriptor
.text:6A2CB49D  call  _LdrpResolveDelayLoadDescriptor@8 ; LdrpResolveDelayLoadDescriptor(x,x)
...
ntdll!LdrpResolveDelayLoadDescriptor

.ntext:6A243F9F ; __stdcall LdrpResolveDelayLoadDescriptor(x, x)
...
text:6A243FA8      mov    ebx, [edx+0Ch]    // ImportAddressTableRVA
.text:6A243FAB      push   esi
.text:6A243FAC      add    ebx, ecx    // the address of function pointer to be overwritten
...
text:6A243FB9      cmp    [ebx], esi
.text:6A243FBB      jz     short loc_6A243FE0    // the current function pointer must not be null
.text:6A243FBD      mov    eax, ebx
.text:6A243FBF      push   0
.text:6A243FC1      push   eax    // the address of function pointer to be overwritten
.text:6A243FC2      push   0
.text:6A243FC4      push   0
.text:6A243FC6      push   edx    // delay load descriptor
.text:6A243FC7      push   ecx    // fake PE
.text:6A243FC8      call   _LdrResolveDelayLoadedAPI@24 ; LdrResolveDelayLoadedAPI(x,x,x,x,x,x)
...
ntdll!LdrResolveDelayLoadedAPI

.text:6A243D40 ; __stdcall LdrResolveDelayLoadedAPI(x, x, x, x, x, x)
...
.text:6A243D7F          lea    edx, [ebp+Address]
.text:6A243D82          mov    ecx, [ebp+arg_0]       // fake PE
.text:6A243D85          call   _LdrpFindLoadedDllByHandle@12 ; LdrpFindLoadedDllByHandle(x,x,x)    // find LDR_DATA_TABLE_ENTRY of the fake PE
.text:6A243D8A          mov    ecx, eax
...
.text:6A243D97          mov    ecx, [ebp+arg_10]
.text:6A243D9A          mov    esi, [ecx]        // get the current function pointer
.text:6A243D9C          mov    eax, esi
.text:6A243D9E          sub    eax, [ebp+arg_0]   // delta = the current function pointer – fake PE
.text:6A243DA1          mov    edi, [ebp+Address]  // LDR_DATA_TABLE_ENTRY of the fake PE
.text:6A243DA4          cmp    eax, [edi+20h]    // delta must be less than SizeOfImage field of LDR_DATA_TABLE_ENTRY
.text:6A243DA7          jb     short loc_6A243DC2
...
.text:6A243DD8          call   _LdrpHandleProtectedDelayload@24 ; LdrpHandleProtectedDelayload(x,x,x,x,x,x)
Corrupt LDR Data Table to Pass the Range Check
Corrupted LDR_DATA_TABLE_ENTRY
ntdll!LdrpHandleProtectedDelayload

.text:6A241C60 ; __stdcall LdrpHandleProtectedDelayload(x, x, x, x, x, x)

...  
.text:6A241CCD  push   edi              // the address of function pointer to be overwritten  
.text:6A241CCE  push    [ebp+arg_C]   
.text:6A241CD1  lea    eax, [ebp+var_228] 
.text:6A241CD7  push   eax 
.text:6A241CD8  call    _LdrpGetDelayloadExportDll@20 ; LdrpGetDelayloadExportDll(x,x,x,x,x)   // get LDR_DATA_TABLE_ENTRY of ntdll.dll  
.text:6A241CDD  mov     [ebp+statusCode], eax  
.text:6A241CE3  test   eax, eax 
.text:6A241CE5  js     loc_6A241FA2  
.text:6A241CEB  mov     ecx, [ebp+var_228]            // LDR_DATA_TABLE_ENTRY of ntdll.dll 
.text:6A241CF1  mov     ecx, [ecx+18h] ; PcValue  // DllBase of ntdll.dll  
.text:6A241CF4  call    _RtlGuardCheckImageBase@4 ; RtlGuardCheckImageBase(x) 
.text:6A241CF9  mov     eax, [esi+18h] 

...
Malformed IMAGE_DELAYLOAD_DESCRIPTOR

```
0:009> dt _IMAGE_DELAYLOAD_DESCRIPTOR 12010140
twinapi!_IMAGE_DELAYLOAD_DESCRIPTOR
+0x000 Attributes : __IMAGE_DELAYLOAD_DESCRIPTOR::<unnamed-type-Attributes>
+0x004 DllNameRVA : 0x180
+0x008 ModuleHandleRVA : 0x160
+0x00c ImportAddressTableRVA : 0x4ce654dc
+0x010 ImportNameTableRVA : 0x168
+0x014 BoundImportAddressTableRVA : 0
+0x018 UnloadInformationTableRVA : 0
+0x01c TimeDateStamp : 0

0:009> da 12010000 + 180
12010180 "ntdll.dll"
0:009> db 12010000 + 160 L4
12010160 00 00 00 00
0:009> ln 12010000 + 4ce654dc
Browse module
Set bu breakpoint

(5ee754dc) chakra!__guard_check_icall_icall_fptr | (5ee754e0) chakra!_IMPORT_DESCRIPTOR_msvcr
Exact matches:
  chakra!__guard_check_icall_icall_fptr = <no type information>
0:009> db 12010000 + 168 L4
12010168 39 01 00 80
Exported entry 0x139 NtCurrentTeb in ntdll.dll
```
ntdll!LdrpHandleProtectedDelayload (Continued)

ntdll!LdrpHandleProtectedDelayload

... 
.lea ecx, [ebp+var_230]
push ecx
push 0
push eax // 313, the ordinal of the function to be resolved
mov ecx, [ebp+var_228] // LDR_DATA_TABLE_ENTRY of ntdll.dll
call _LdrpResolveProcedureAddress@20 ; LdrpResolveProcedureAddress(x,x,x,x,x) // resolve function by its ordinal
... 
push eax
push edi
push [ebp+Address] // the address of resolved function pointer
mov edx, [ebp+var_23C] // the address of function pointer to be overwritten
mov ecx, esi // LDR_DATA_TABLE_ENTRY of the fake PE
call _LdrpWriteBackProtectedDelayLoad@20 ; LdrpWriteBackProtectedDelayLoad(x,x,x,x) // overwrite the function pointer
...
ntdll!LdrpWriteBackProtectedDelayLoad

```
ntdll! LdrpWriteBackProtectedDelayLoad

.text:6A2742D7 ; __stdcall LdrpWriteBackProtectedDelayLoad(x, x, x, x, x)

...  

.text:6A274310 push 4 ; NewProtect // remove write protection on the function pointer
.text:6A274312 lea eax, [ebp+ProtectSize]
.text:6A274315 push eax ; ProtectSize

...  

.text:6A27431C call _ZwProtectVirtualMemory@20 ; ZwProtectVirtualMemory(x,x,x,x,x)

...  

.text:6A274338 mov [edi+eax], ecx // overwrite the function pointer

...  

.text:6A274343 push 2 ; NewProtect // restore write protection on the function pointer
.text:6A274345 lea eax, [ebp+ProtectSize]
.text:6A274348 push eax ; ProtectSize

...  

.text:6A27434F call _ZwProtectVirtualMemory@20 ; ZwProtectVirtualMemory(x,x,x,x,x)

...```
Corrupted chakra!__guard_check_icall_icall_fptr

```
ntdll!LdrpWriteBackProtectedDelayLoad+0x7d
ntdll!LdrpHandleProtectedDelayLoad+0x230
ntdll!LdrResolveDelayLoadedAPI+0x9d
ntdll!LdrpResolveDelayLoadDescriptor+0x2e
ntdll!LdrResolveDelayLoadsFromDll+0x32

Disassembly
Offset: @$scopeip

77a8434d 6aff  push  0FF000000
77a8434f e86cc0100  call  ntdll!NtProtectVirtualMemory (77aa0ec0)
77a84354 ff75f0  push  dword ptr [ebp-10h] ss:0023:04efcba0=02c0284c
77a84357 e8a478fcff  call  ntdll!RtlReleaseSRWLockExclusive (77a4bc00)
77a8435c 5f  pop  edi

Command
0:009> u poi(chakra!__guard_check_icall_icall_fptr) L2
ntdll!NtdllNtCurrentTeb: Exported entry 313 NtCurrentTeb in ntdll.dll
77add080 64a180000000  mov  eax, dword ptr fs:[00000018h]
77add086 c3  ret
0:009> !address chakra!__guard_check_icall_icall_fptr

Usage: Image
Base Address: 5ee75000
End Address: 5eece000
Region Size: 00055000 ( 340.000 KB)
State: 00001000  MEM_COMMIT
Protect: 00000002  PAGE_READONLY
Type: 01000000  MEM_IMAGE
Allocation Base: 5e830000
Allocation Protect: 0000000000 PAGE_EXECUTE_WRITECOPY
Image Path: C:\WINDOWS\SYSTEM32\chakra.dll
```
Bypass CFG By Calling ntdll!LdrResolveDelayLoadsFromDll

This site says...
Go check the replaced guard_check_icall_fptr @ 0x605484d8

☐ Don't let this page create more messages

OK
CFG Improvement in WIP 15048
Significant improvement was made on CFG after WIP build 14986, many previous valid call targets now are no longer valid (see previous two slides)! Obviously, MS has been working hard to make its CFG implementation more fine-grained to defeat calling function out-of-context. As a consequence, we’ll have to find new way to exploit memory vulnerability, and this is really where data-only attack come into play!

“Microsoft Mitigation Bypass Bounty”

<table>
<thead>
<tr>
<th>Mitigation</th>
<th>In scope</th>
<th>Out of scope</th>
</tr>
</thead>
</table>
| Control Flow Guard(CFG) | Techniques that make it possible to gain control of the instruction pointer through an indirect call in a process that has enabled CFG. | - Hijacking control flow viare turn address corruption  
- Bypasses related to limitations of coarse-grained CFI (e.g. calling functions out of context)  
- Leveraging non-CFG images  
- Bypasses that rely on modifying or corrupting read-only memory |
Data-only Attack

• The fundamental of data-only attack
  • Change the program’s default behavior by manipulating the data it depends on without divert the program’s execution control flow. Change in program’s data may lead to the change of program’s original execution path, therefore achieve some powerful things, such as bypassing certain restriction or protection. Please note that the change in program’s code path caused by the change of data is still within the program’s normal logic, so it won’t have problem when CFI is enforced.

• Example of successful data-oriented attack
  • Vital Point Strike (JavaScript god mode)
  • EMET bypass (Replacing EnableProtectionPtr in CONFIG_STRUCT)

• What kind of data will be targeted
  • Any data that can be leveraged to alter the program’s default behavior will be of interest to the data-oriented attack, such as certain global flag in data section, or field in an object etc.
  • Unprotected .data section, unprotected heap/private data, stack.

• How to perform the data-only attack
  • Most data-oriented attacks require the ability of arbitrary address read/write.
  • Being able to accurately locate the targeted data at runtime is the key to success.
  • The existence of certain data is transient or time-sensitive, so timing may play an important role in these cases.
LdrpWork Mechanism

• The LdrpWork mechanism is responsible for performing various loader works related to the loading of a module, and these works can be processed either synchronously or asynchronously. The asynchronous mode is specifically used to support “parallel loading” of dependent modules.

• In memory, a LdrpWork structure contains a fixed size header and a variable length part (module name). In the asynchronous mode, all LdrpWork instances are organized using a FIFO doubly-linked list.

• LdrpWork mechanism consists of four major functions:
  • ntdll!LdrpAllocatePlaceHolder: allocate a new work from the loader’s private heap
  • ntdll!LdrpQueueWork: insert a new work at the tail of list
  • ntdll!LdrpDrainWorkQueue: remove the head node and send it to process routine
  • ntdll!LdrpProcessWork: process one single work, such as module mapping or snapping. This function can be invoked in both synchronous and asynchronous mode

• “Snapping” is one of the important loader works, and it’s fulfilled by:
  • ntdll!LdrpSnapModule: load dependent modules and resolve imported functions
  • ntdll!LdrpDoPostSnapWork: restore the memory attribute of .idata section
ntdll!LdrpAllocatePlaceHolder

ntdll!LdrpAllocatePlaceHolder

.text:00000001800158EC LdrpAllocatePlaceHolder proc near
...

.text:0000000180015911    mov    edx, cs:NtdllBaseTag
.text:0000000180015917    mov    r14d, r8d
.text:000000018001591A    add    edx, 40000h
.text:0000000180015920    mov    rsi, rcx    // pointer to UNICODE_STRING
.text:0000000180015923    xor    r12d, r12d
.text:0000000180015926    or     edx, 8    // Flags
.text:0000000180015929    mov    [rdi], r12
.text:000000018001592C    mov    ebp, r9d
.text:000000018001592F    movzx   r8d, word ptr [rcx]    // the path length of the module being loaded
.text:0000000180015933    mov    rcx, cs:LdrpHeap    // Heap handle
.text:000000018001593A    add    r8, 0B2h    // add the size of LdrpWork header (0xB0) and a null terminator (2)
.text:0000000180015941    call   RtlAllocateHeap
...
.text:0000000180015A00 LdrpAllocatePlaceHolder endp
ntdll!LdrpQueueWork

.Ltext:0000000180077260 LdrpQueueWork  proc near
...
.Ltext:0000000180077275    lea   rcx, LdrpWorkQueueLock
.Ltext:000000018007727C    call  RtlEnterCriticalSection
.Ltext:0000000180077281    mov   rcx, cs:qword_1801542D8       // the tail of LdrpWorkQueue
.Ltext:0000000180077288    lea   rdx, LdrpWorkQueue         // the head of LdrpWorkQueue
.Ltext:000000018007728F    lea   rax, [rbx+38h]            // LIST_ENTRY field of LdrpWork structure
.Ltext:0000000180077293    cmp   [rcx], rdx                // verify the integrity of the doubly-linked list
.Ltext:0000000180077296    jnz   loc_1800C79A8
.Ltext:000000018007729C    mov   [rax+8], rcx            // insert the new LdrpWork at the tail of LdrpWorkQueue
.Ltext:00000001800772A0    mov   [rax], rdx
.Ltext:00000001800772A3    mov   [rcx], rax
.Ltext:00000001800772A6    lea   rcx, LdrpWorkQueueLock
.Ltext:00000001800772AD    mov   cs:qword_1801542D8, rax
.Ltext:00000001800772B4    call  RtlLeaveCriticalSection
...
.Ltext:00000001800772E7 LdrpQueueWork  endp
ntdll!LdrpDrainWorkQueue

```
.text:000000018003B3A0  LdrpDrainWorkQueue  proc near
...
.text:000000018003B3BF  lea   r12, LdrpWorkQueue
...
.text:000000018003B3F1  mov   rbx, cs:LdrpWorkQueue           // get the 1st node from the head of LdrpWorkQueue
.text:000000018003B3F8  mov   rax, [rbx]                      // get the 2nd node
.text:000000018003B3F8  cmp   [rbx+8], r12                  // verify the integrity of the doubly-linked list
.text:000000018003B3FF  jnz   loc_1800BAD6C
.text:000000018003B405  cmp   [rax+8], rbx                   // verify the integrity of the doubly-linked list
.text:000000018003B409  jnz   loc_1800BAD6C
.text:000000018003B40F  mov   cs:LdrpWorkQueue, rax       // remove the 1st node from LdrpWorkQueue
.text:000000018003B416  mov   [rax+8], r12
.text:000000018003B41A  cmp   r12, rbx
.text:000000018003B41D  jnz   loc_18003B4F7                  // make sure the removed node is a LdrpWork to be processed
.text:000000018003B423  cmp   cs:LdrpWorkInProgress, edi
.text:000000018003B429  jnz   short loc_18003B438
```
ntdll!LdrpDrainWorkQueue

.text:000000018003B42B mov cs:LdrpWorkInProgress, 1
.text:000000018003B435 mov sil, 1
...
.text:000000018003B444 test sil, sil
.text:000000018003B447 jz loc_18003B4E0
.text:000000018003B44D test edi, edi
.text:000000018003B44F jnz short loc_18003B47F
...
.text:000000018003B4E0 cmp r12, rbx
.text:000000018003B4E3 jnz short loc_18003B50C // make sure the removed node is a LdrpWork to be processed
...
.text:000000018003B50C lea rcx, [rbx-38h] // LdrpWork structure to be processed
.text:000000018003B510 mov dl, bpl
.text:000000018003B513 call LdrpProcessWork // Process the LdrpWork node that is just removed from the head of LdrpWorkQueue
.text:000000018003B518 jmp loc_18003B3D5
.text:000000018003B518 LdrpDrainWorkQueue endp
ntdll!LdrpProcessWork

.text:0000000180010328 LdrpProcessWork proc near

... 

.text:000000018001033B mov rbx, rcx

.text:000000018001033E mov rax, [rcx+20h]

.text:0000000180010342 cmp dword ptr [rax], 0

.text:0000000180010345 jl loc_1800104AD

.text:000000018001034B mov rax, [rcx+30h] // _LDR_DATA_TABLE_ENTRY of current module being loaded

.text:000000018001034F mov rcx, [rax+98h] // _LDR_DDAG_NODE

.text:0000000180010356 cmp dword ptr [rcx+38h], 0 // test State._LDR_DDAG_NODE to determine which work to do

.text:000000018001035A mov rcx, rbx

.text:000000018001035D jz short loc_180010373

.text:000000018001035F call LdrpSnapModule // load dependent modules and resolved imported functions

... 

.text:0000000180010373 test dword ptr [rbx+18h], 100000h

.text:000000018001037A jnz loc_18001042F

.text:0000000180010380 test dword ptr [rbx+18h], 200h

.text:0000000180010387 jnz short loc_1800103B1

.text:0000000180010389 call LdrpMapDllSearchPath
ntdll!LdrpProcessWork (Continued)

ntdll!LdrpProcessWork

.text:000000018001038E  mov   edi, eax
.text:0000000180010390  mov   [rsp+58h+arg_10], eax
.text:0000000180010394  mov   ecx, 80000000h
.text:0000000180010399  lea   eax, [rdi+rcx]
.text:000000018001039C  test  ecx, eax
.text:000000018001039E  jnz   loc_1800104AD

... 

.text:00000001800103B1  call  LdrpMapDllFullPath
.text:00000001800103B6  jmp  short loc_18001038E

... 

.text:000000018001042F  call  LdrpMapDllRetry
.text:0000000180010434  mov   edi, eax
.text:0000000180010436  mov   [rsp+58h+arg_10], eax
.text:000000018001043A  jmp  loc_180010394

... 

.text:000000018001050D LdrpProcessWork endp
.text:0000000180031940 LdrpSnapModule  proc near

...  

.text:00000001800319A0   mov    eax, [rdi+70h]     // the number of already loaded dependent modules
.text:00000001800319A3   cmp    eax, [rdi+58h]     // the total number of all dependent modules
.text:00000001800319A6   jnb    loc_1800320FC

...  

=text:0000000180031D07   inc    dword ptr [rdi+70h] // increase the counter
.text:0000000180031D0A   jmp    loc_1800319A0

...  

.text:00000001800320FC   test   esi, esi
.text:00000001800320FE   js     loc_1800322B7
.text:0000000180032104   mov    rcx, rdi
.text:0000000180032107   call   LdrpDoPostSnapWork     // restore .idata section’s memory attribute

...  

.text:00000001800322C9 LdrpSnapModule  endp
ntdll!LdrpDoPostSnapWork

ntdll!LdrpDoPostSnapWork

.text:000000018004C30C LdrpDoPostSnapWork proc near

...  

.text:000000018004C316  mov  rbx, rcx
.text:000000018004C319  xor  edi, edi
.text:000000018004C31B  mov  ecx, edi
.text:000000018004C31D  lea  rdx, [rbx+60h]  // the start address of .idata section
.text:000000018004C321  cmp  [rdx], rdi
.text:000000018004C324  jz   short loc_18004C34A
.text:000000018004C326  mov  r9d, [rbx+80h]  // the original memory attribute of .idata section in PE file
.text:000000018004C32D  lea  rax, [rsp+38h+arg_0]
.text:000000018004C332  lea  r8, [rbx+68h]  // the size of .idata section
.text:000000018004C336  mov  [rsp+38h+var_18], rax
.text:000000018004C33B  or   rcx, 0FFFFFFFFFFFFFFFh
.text:000000018004C33F  call  ZwProtectVirtualMemory  //restore the .idata section’s original memory attribute

...
ntdll!LdrpDoPostSnapWork (Continued)

ntdll!LdrpDoPostSnapWork

...
An Instance of LdrpWork in Memory

_UNICODE_STRING: module name

_LDR_DATA_TABLE_ENTRY: module

_LIST_ENTRY

_info about dependent modules

start address of .idata section

memory attribute of .idata section

size of .idata section

variable length part: module name
_LDR_DATA_TABLE_ENTRY

0:018 dt _LDR_DATA_TABLE_ENTRY 14f88917b40
Windows_01LDR_DATA_TABLE_ENTRY
+0x0000 LoadOrderLinks ▴  ▴ ▴ LIST_ENTRY [ 0x00007ff9-577043f0 - 0x000014f-08316a60 ]
+0x0100 LoadHashLinks ▴  ▴ ▴ LIST_ENTRY [ 0x00007ff9-57704200 - 0x000014f-88916a70 ]
+0x0200 InitializationOrderLinks ▴  ▴ ▴ LIST_ENTRY [ 0x0000ff0-00000000 - 0x00000000-00000000 ]
+0x0300 DLLBase ▴  ▴ ▴ 36c58000 Void
+0x0400 EntryPoint ▴  ▴ ▴ 0x00007ff9-36c93d10 Void
+0x0500 SizeOfImage ▴  ▴ 0x36c8000
+0x0600 FullDllName ▴  ▴ UNIQUE STRING "C:WindowsSYSTEM32D USER.dll"
+0x0700 BaseDllName ▴  ▴ UNIQUE STRING "D USER.dll"
+0x0800 Flags ▴  ▴ 0x00000000
+0x0900 Flags2 ▴  ▴ 0x00000000
+0x0a00 InterfaceTLS ▴  ▴ 0x00000000
+0x0b00 CritSectCount ▴  ▴ 0x00000000
+0x0c00 Reserved ▴  ▴ 0x00000000
+0x0d00 InternedStringTable ▴  ▴ 0x00000000
+0x0e00 DosStubDll ▴  ▴ 0x00000000
+0x0f00 EncryptedBinary ▴  ▴ 0x00000000
+0x1000 Indirect ▴  ▴ 0x00000000
+0x1100 IsFullDll ▴  ▴ 0x00000000
+0x1200 LoadNotificationsSent ▴  ▴ 0x00000000
+0x1300 TelemetryEntryProcessed ▴  ▴ 0x00000000
+0x1400 ProcessStaticImport ▴  ▴ 0x00000000
+0x1500 InLegacyLists ▴  ▴ 0x00000000
+0x1600 InIndexes ▴  ▴ 0x00000000
+0x1700 InMemory ▴  ▴ 0x00000000
+0x1800 SizeOfImage ▴  ▴ 0x00000000
+0x1900 InExecutionTable ▴  ▴ 0x00000000
+0x1a00 ReservedFlags1 ▴  ▴ 0x00000000
+0x1b00 LoadInProcess ▴  ▴ 0x00000000
+0x1c00 LoadConfigProcessed ▴  ▴ 0x00000000
+0x1d00 EntryProcessed ▴  ▴ 0x00000000
+0x1e00 ProtectDelayLoad ▴  ▴ 0x00000000
+0x1f00 ReservedFlags2 ▴  ▴ 0x00000000
+0x2000 DontCallForThreads ▴  ▴ 0x00000000
+0x2100 ProcessAttachCalled ▴  ▴ 0x00000000
+0x2200 ProcessAttachFailed ▴  ▴ 0x00000000
+0x2300 CorDebugValidate ▴  ▴ 0x00000000
+0x2400 CorUnmanaged ▴  ▴ 0x00000000
+0x2500 DontRelocate ▴  ▴ 0x00000000
+0x2600 CorILOnly ▴  ▴ 0x00000000
+0x2700 ReservedFlags3 ▴  ▴ 0x00000000
+0x2800 Redirection ▴  ▴ 0x00000000
+0x2900 DeferredRefs ▴  ▴ 0x00000000
+0x2a00 Compartment ▴  ▴ 0x00000000
+0x2b00 ObsoleteTocCount ▴  ▴ 6
+0x2c00 Oid ▴  ▴ 0x00000000
+0x2d00 HashLinks ▴  ▴ ▴ LIST_ENTRY [ 0x00007ff9-577040d0 - 0x000014f-9occ263d0 ]
+0x2e00 TimeDateStamp ▴  ▴ 0x1741d3cf
+0x2f00 EntryPointActivationContext ▴ (null)
+0x3000 Id ▴ (null)
+0x3100 LoadModuleDir ▴  ▴ ▴ LIST_ENTRY [ 0x00007ff9-889a3680 - 0x000014f-889a3680 ]
+0x3200 EntryPointActivationContext ▴ (null)
+0x3300 ParentDllBase ▴  ▴ 0x00007ff9-34993000 Void
enum State {
  LdrModulesPlaceHolder = 0,
  LdrModulesMapping,
  LdrModulesMapped,
  LdrModulesWaitingForDependencies,
  LdrModulesSnapping,
  LdrModulesSnapped,
  LdrModulesCondensed,
  LdrModulesReadyToInit,
  LdrModulesInitializing
};
Leak Stack Address From LdrpWork Structure

- Two stack addresses are saved in LdrpWork structure, and they belong to the thread on which the loading of module operates.
- LdrpWork structure stays in LdrpWorkQueue only for a very short time, and it will soon be removed from the queue for processing, which makes it impossible to trace it by reading it from the queue.
- The size of LdrpWork structure (PlaceHolder) is variable due to the difference in length of module name, so its location (such as LFH bucket) in LdrpHeap can’t be accurately predicted.
- But what if we can force the LdrpWork to be allocated in an expected size? 😊 By combining data-only attack technique with the knowledge about Windows 10 segment Heap Fengshui, it will not be difficult for us to reliably locate one LdrpWork instance in memory and leak the stack address out of it.
CFG Bypass: Abuse LdrpWork Mechanism

• No protection on LdrpWork doubly-linked list (ntdll!LdrpWorkQueue) to prevent it from being modified (such as inserting a new node).

• LdrModuleSnapping work can be leveraged to change the memory attribute of arbitrary address. ntdll!LdrpDoPostSnapWork fails to enforce a check on the target address to make sure it indeed belongs to .idata section of certain module that is in LdrModuleSnapping state.

• The whole exploitation process is virtually as simple as manipulating a linked list: first fake a LdrpWork structure (LdrModuleSnapping work) that carries all the information needed for the memory attribute change, then insert it to doubly-linked list and wait for it to be processed. Normally this won’t take long as LdrpDrainWorkQueue operation will be triggered at many places. Please note that the data in the fake LdrpWork must be correctly populated to ensure it can eventually reach ntdll!LdrpDoPostSnapWork.
The Diagram of CFG Bypass Scheme

Fake LdrpWork

- LIST_ENTRY
- address of .idata section
- size of .idata section: 0x1000
- attribute of .idata section: PAGE_READWRITE

ntdll.dll

- .data section
- LdrpWorkQueue

chakra.dll

- Import table
- __guard_check_icall_fptr
CFG Bypass: Abuse LdrpWork Mechanism

```
ntdll!LdrpDoPostSnapWork:
00007ff9 575fc30c 43898c=2410 mov qword ptr [rsp+10h],rbx
00007ff9 575fc311 57 push rdi
00007ff9 575fc312 4803e9=30 sub rsp,30h
00007ff9 575fc316 483bd9 mov rbx,rcx
00007ff9 575fc319 33ff xor edi,edi
00007ff9 575fc31b 3bcf mov ecx,edi
00007ff9 575fc31d 483d5360 lea rdx,[rbx+60h]
00007ff9 575fc321 48393a cmp qword ptr [rdx],rdi
00007ff9 575fc324 7424 je ntdll!LdrpDoPostSnapWork+0x3e (00007ff9 575fc34a)
00007ff9 575fc325 449b8b80000000 mov r9d,qword ptr [rbx+80h]
00007ff9 575fc32d 483d442440 lea rax,[rsp+40h]
00007ff9 575fc332 4c0d4586 lea r3,[rbx+60h]
00007ff9 575fc335 4839442420 mov qword ptr [rbx+20h],rax
00007ff9 575fc33b 4839c3ff or rcx,0FFFFFFFFFFFFFFFh
00007ff9 575fc33f 85e8=20500 call ntdll!RtlProtectVirtualMemory (00007ff9 576565b0)
```

Command

```
0:017> r
rax=000000b715f7e2a0 rbx=00000027d2d455730 rcx=ffffffffffffffff
rdx=00000027d2d455720 rsi=0000000000000000 rdi=0000000000000000
r8=0000000000000000 r9=0000000000000000 r10=0000000000000000
r11=0000000000000000 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
icpl=0
nv up ei ng nz na po nc
```

```
ntdll!LdrpDoPostSnapWork+0x33:
00007ff9 575fc33f e85ca20500 call ntdll!RtlProtectVirtualMemory (00007ff9 576565b0)
```
CFG Bypass: Abuse LdrpWork Mechanism
## Solution for CFG Bypass Abusing LdrpWork

<table>
<thead>
<tr>
<th>Bypass</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-enlightened Just-in-Time (JIT) compilers</td>
<td>Mitigated in latest version of Edge on Windows 10 (Chakra, Adobe Flash, and WARP)</td>
</tr>
<tr>
<td>Multiple non-instrumented indirect calls reported to our Mitigation Bypass Bounty</td>
<td>Mitigated in latest version of Edge on Windows 10</td>
</tr>
<tr>
<td>Calling sensitive APIs out of context</td>
<td>NtContinue/longjmp – mitigated for all CFG enabled apps on Windows 10</td>
</tr>
<tr>
<td></td>
<td>VirtualProtect/VirtualAlloc – mitigated in latest version of Microsoft Edge on Windows 10</td>
</tr>
<tr>
<td></td>
<td>LoadLibrary – mitigated in latest version of Microsoft Edge on Windows 10 via CIG</td>
</tr>
<tr>
<td></td>
<td>WinExec – mitigated in Edge on Windows 10 anniversary edition via child process policy</td>
</tr>
<tr>
<td>Corrupting mutable read-only memory</td>
<td>Known limitation that we are exploring solutions for</td>
</tr>
</tbody>
</table>
Suggestions for Preventing Data-only Attack

• Never leave critical data unprotected!
  • If possible, apply write protection on the critical data page. Remove the write protection only when it needs to be updated, and be sure to lock the data page immediately after the updating finishes.
  • If for some reason the scheme above is not feasible (such as granularity, performance etc), the critical data can probably be stored in a dynamically allocated memory (ASLR enabled), and its address needs to be encrypted and stored separately.
  • For the encryption scheme mentioned above, the secret key of encryption has to be in a protected area, such as in kernel space, to prevent the access from user-mode. Moreover, the strength of encoding or encryption algorithm needs to enhanced to prevent brute-force attack.

• Always try verifying the integrity of critical data before using it. With such extra logic being introduced, many data-only attacks can be detected in their early stage.
Conclusion

• With the emergence of fine-grained CFI solution, the approach of calling function out-of-context will gradually lose its effectiveness.

• Today, application programs and operating systems have a lot of unprotected data, which can be leveraged to conduct powerful attack without the need of altering the program’s execution flow.

• Even if people have already been aware of the danger of data-only attacks, it’s still very difficult to prevent.
  • In some cases, it’s almost impossible to distinguish an attack from a legitimate access. Therefore, data-only attacks can’t always be resolved from the program’s logical perspective.
  • Due to performance consideration, OS/Application can’t move all its critical data into kernel space. In most cases, such user-space data will be protected by either “read-only” memory attribute (such as PE module’s import/export table section, .mrdata section of ntdll.dll) or simple encoding (RtlEncodePointer).
  • The battle of contending for the protected memory will continue.

• Data-only attack may have some variations, and it can be combined with some other exploitation techniques, such as race condition. We have discovered a couple of such bugs, and we would like to share the details after they are fixed by the vendor.
Q & A

• This concludes part II of our data-only attack series. In our future conference talks, we are looking to present the advanced data-only attack techniques (data-only combined with other exploitation techniques). Stay tuned!

• You are welcomed to send questions to
  • Bing Sun @ bing_sun@mcafee.com
  • Chong Xu @ chong_xu@mcafee.com

• Thank MSRC for getting the issues fixed in a timely manner.

• Special thanks to Stanley Zhu, Haifei Li and the McAfee IPS Vulnerability Research team.
References

- From read-write anywhere to controllable calls
- Mitigation bounty — 4 techniques to bypass mitigations
- Bypassing Control Flow Guard in Windows 10
- Use Chakra engine again to bypass CFG
- Chakra JIT CFG Bypass
- Bypass Control Flow Guard Comprehensively
- JIT Spraying Never Dies - Bypass CFG By Leveraging WARP Shader JIT Spraying
- Look Mom, I don’t use Shellcode
- Write Once, Pwn Anywhere