

Shellcodes Are for the 99%

Bing Sun, Sr. Security Researcher @ McAfee

Stanley Zhu, Sr. Security Researcher @ Didi Chuxing

Chong Xu, Sr. Director @ McAfee

CanSecWest 2018

Abstract

- Today most memory vulnerability exploits execute their second-stage exploit or final payload either by creating an executable memory page with shellcode or by loading a third-party executable module. However, as the new exploitation mitigation features ACG (Arbitrary Code Guard) and CIG (Code Integrity Guard) are introduced to high-risk applications on modern operating systems (for example, Microsoft Edge on Windows 10 x64), current memory vulnerability exploitation methods that rely on executing customized code will no longer function. As a result, a code-reuse attack (such as ROP, COOP, etc.) seems to be the only option that can survive these new mitigations. The main drawback of a code-reuse attack, however, lies in its limited flexibility and extensibility. Due to these limitations, it is extremely difficult to implement complex logic, such as multithreading, with pure code reuse. As a countermeasure to the new mitigation, we propose a new browser exploitation framework that leverages and extends JavaScript to complete tasks that would otherwise need to be done by native code. With this framework, we can basically achieve almost anything that native code can do. Moreover, unlike previous works that abuse JavaScript, such as "JS god mode" or "interdimensional execution," our new technique covers the whole exploitation kill chain (from initial RCE to EoP stage to final payload), without using any native code. In this talk, we will discuss various challenges of implementing this new exploitation framework in detail. In the demo section, we will present some examples that show how this new exploitation framework can be easily adopted in real-world exploitation scenarios.

About the Speakers

- Bing Sun is a senior security researcher. He leads the IPS security research team of McAfee. He has extensive experiences in operating system low-level and information security technique R&D, with especially deep diving in advanced vulnerability exploitation and detection, rootkits detection, firmware security, and virtualization technology. Bing is a regular speaker at international security conference such as XCon, Black Hat, and CanSecWest.
- Stanley Zhu is a security researcher of Didi Chuxing. A designer of the Bytehero Heuristic Detection Engine, he provided the BDV engine with VirusTotal and OPSWAT platform. He has many years of experience in information security techniques research and is interested in advanced vulnerability exploitation and detection, virus and rootkits reverse engineering. He has spoken at security conferences such as CanSecWest2014, AVAR2012, XCon2010, XCon2015, and XCon2016.
- Chong Xu received his Ph.D. degree in networking and security from Duke University. His current focus includes research and innovation on intrusion and prevention techniques as well as threat intelligence. He is a senior director of the McAfee IPS team, which leads McAfee vulnerability research, malware and APT detection, and botnet detection. The team feeds security content and innovative protection solutions into McAfee's network IPS, host IPS, and sandbox products, as well as McAfee Global Threat Intelligence (GTI).

Agenda

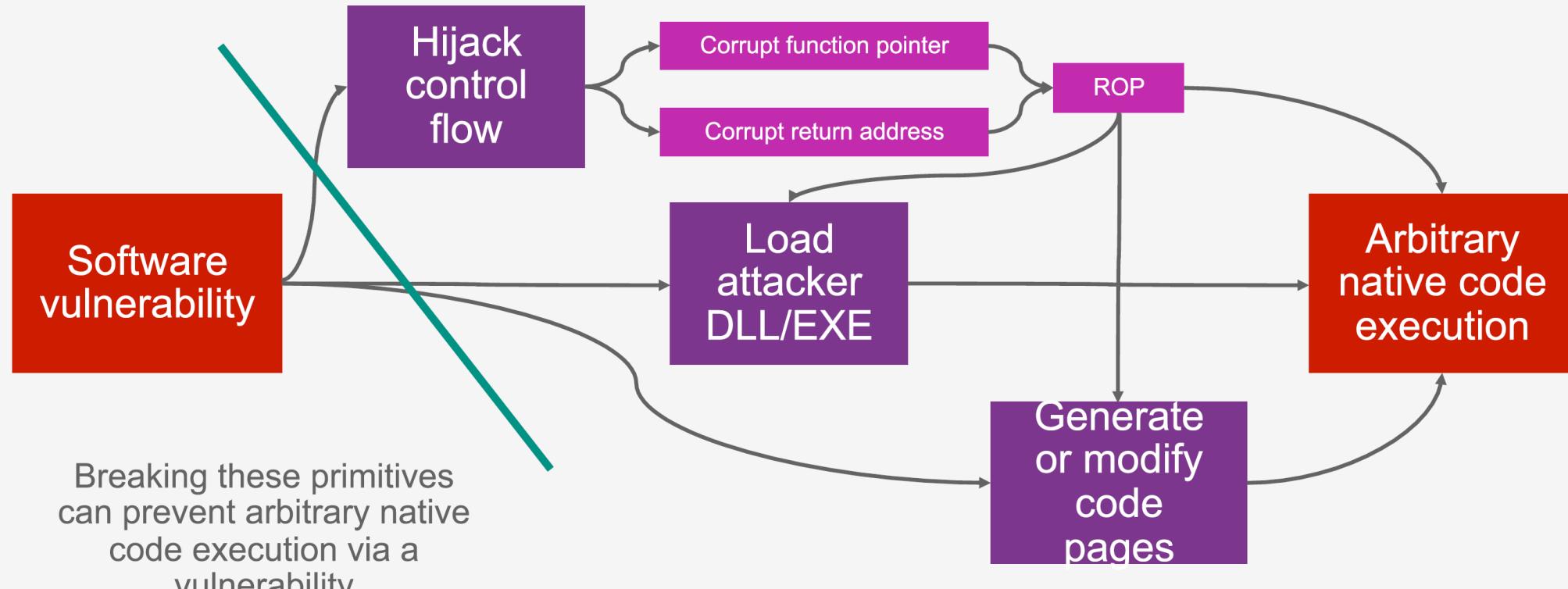
- 1. An Overview of Microsoft Exploitation Mitigation Features and Bypasses**
- 2. A JavaScript-Based Browser Exploitation Framework**
- 3. Demo**
- 4. Conclusion**
- 5. Q&A**

1. An Overview of Microsoft Exploitation Mitigation Features and Bypasses

Paths to Achieve Native Code Execution

The paths to arbitrary native code execution

There are a finite number of ways to transform a vulnerability into arbitrary native code execution



Microsoft's Technologies for Mitigating Code Execution

Technologies for mitigating code execution

Prevent arbitrary code generation

Code Integrity Guard

Images must be signed and load from valid places

Arbitrary Code Guard

Prevent dynamic code generation, modification, and execution

Prevent control-flow hijacking

Control Flow Guard

Enforce control flow integrity on indirect function calls

???

Enforce control flow integrity on function returns

- ✓ Only valid, signed code pages can be mapped by the app
- ✓ Code pages are immutable and cannot be modified by the app
- ✓ Code execution stays “on the rails” per the control-flow integrity policy

ACG (Arbitrary Code Guard)

- ACG prevents a process from generating dynamic code or modifying existing executable code, and is configured via `ProcessDynamicCodePolicy`
- Two W^X policies:
 - ✓ Existing code pages cannot be made writable
 - ✓ New, unsigned code pages cannot be created
- Enforced in all kernel functions that deal with memory protection:
 - ✓ `NtProtectVirtualMemory`, `NtAllocateVirtualMemory`, `NtMapViewOfSection`, etc.

CIG (Code Integrity Guard)

- CIG prevents a process from loading unsigned images, and is configured via ProcessSignaturePolicy
- Enforced in the kernel function that creates the section object for the image (NtCreateSection)
- In addition to ProcessSignaturePolicy, another mitigation policy ProcessImageLoadPolicy (NoRemoteImages, NoLowMandatoryLabelImages) and a process creation attribute (CHILD_PROCESS_POLICY) are used to prevent loading untrusted images.

CFG (Control Flow Guard)

- CFG prevents an exploit from hijacking the program's control flow, and is configured via ProcessControlFlowGuardPolicy
- The call target check is enforced at each indirect control transfer instruction (call and jmp). The check is performed by routines in ntdll.dll (LdrpValidateUserCallTarget, LdrpDispatchUserCallTarget etc).
- CFG does not protect control transfers via “ret” that are expected to be addressed by a hardware-based solution (such as Intel CET, etc.)

Mitigation Policy and Representation in EPROCESS

```
kd> dt nt!_EPROCESS
...
+0x300 Flags2          : Uint4B
...
+0x300 DisableDynamicCode : Pos 10, 1 Bit
...
+0x304 Flags           : Uint4B
...
+0x304 ControlFlowGuardEnabled : Pos 4, 1 Bit
...
+0x6cc Flags3          : Uint4B
...
+0x6cc ProhibitRemoteImageMap : Pos 11, 1 Bit
+0x6cc ProhibitLowTLLImageMap : Pos 12, 1 Bit
+0x6cc SignatureMitigationOptIn : Pos 13, 1 Bit
+0x6cc DisableDynamicCodeAllowOptOut : Pos 14, 1 Bit
...
+0x6cc PreferSystem32Images : Pos 17, 1 Bit
...
+0x6cc DisableDynamicCodeAllowRemoteDowngrade : Pos 23, 1 Bit
...
+0x6cc ControlFlowGuardExportSuppressionEnabled : Pos 26, 1 Bit
...
+0x6cc ControlFlowGuardStrict : Pos 29, 1 Bit
...
+0x810 Flags4          : Uint4B
...
+0x810 RestrictSetThreadContext : Pos 1, 1 Bit
```

```
PS C:\Users\bing> Get-ProcessMitigation -Id 1672
```

```
ProcessName: MicrosoftEdgeCP
Source      : Running Process
Id         : 1672

DEP:
    Enable                         : on
    Disable ATL                     : off

ASLR:
    BottomUp                      : on
    ForceRelocate                 : on
    HighEntropy                   : on
    DisallowStripped              : off

StrictHandle:
    RaiseExceptionOnInvalid       : on
    HandleExceptionsPermanently   : on

System Call:
    DisallowWin32kSysCalls        : off

ExtensionPoint:
    DisableExtensionPoints       : off

DynamicCode:
    ProhibitDynamicCode          : on
    AllowThreadOpt               : on
    AllowRemoteDowngrade         : off

CFG:
    EnableCFG                    : on
    EnableExportSuppression      : off
    StrictMode                   : off

BinarySignature:
    MicrosoftSignedOnly          : off
    StoreSignedOnly               : on
    MitigationOptIn              : off

FontDisable:
    DisableNonSystemFonts        : off
    AuditNonSystemFontLoading    : off

ImageLoad:
    NoRemoteImages                : on
    NoLowMandatoryLabelImages     : off
    PreferSystem32Images         : off
```

Achieving Arbitrary Code Execution With ACG+CIG+CFG All in Place

- ACG and CIG together ensure only trusted code can be executed. They are enforced in kernel mode, so instead of attacking them directly, we choose to circumvent them.
- CFG does a good job in stopping most code-reuse attacks through control-flow hijacking (ROP etc.). However, because CFG is a coarse-grained CFI implementation and the check itself is primarily enforced in user mode, it is subjected to various bypasses.
- Once CFG is breached, we have two options to achieve arbitrary code execution:
 - ROP only
 - Interactive runtime (such as a script engine) + minimal ROP (if needed)

Option I: ROP Only

- Constructing the second-stage exploit or the final payload with ROP-only shellcode is possible, but could be very painful
- Example: [The first ROP-Only, Sandbox-Escaping PDF Exploit \(CVE-2013-0641\).](#)
- Pros:
 - ✓ The native code has direct access to the memory space and system APIs
- Cons:
 - ✓ It is not always possible to find suitable ROP gadgets
 - ✓ It is difficult to implement complex logic
 - ✓ The code is platform and version specific, thus not easy to maintain

Option II: Interactive Runtime + Minimal ROP

- By leveraging and extending the capability of scripting with interactive runtime, we can create a script-based exploit/payload that can achieve almost anything that its native code counterpart can do
- Pros:
 - ✓ It is convenient to implement complex logic with a high-level language
 - ✓ The script-based code is human readable and portable
- Cons:
 - ✓ The data type and execution mechanism of script language are not compatible with that of native code
 - ✓ The script language has some restrictions that are imposed by its script engine
 - ✓ The script language is, by design, not granted the full access to the underlying operating system

2. A JavaScript-Based Browser Exploitation Framework (Microsoft Edge)

Leveraging Chakra JS Engine in Edge to Achieve Arbitrary Code Execution

- The Chakra JS engine in Edge provides a strong scripting environment that allows us to implement complex logic with JavaScript
- However, to achieve arbitrary code execution (go beyond the JS VM), there are some challenges to overcome:
 - ✓ Limited data type
 - ✓ Call system APIs (JS -> native code)
 - ✓ Multithreading and thread communication
 - ✓ Callback mechanism (native code -> JS)

Limited Data Type

- JavaScript has no true 64-bit integer
 - ✓ Max safe value is $2^{53}-1$, bitwise operators deal only with 2^{32} integer values
 - ✓ [long.js](#)
- JavaScript does not have the concept of pointer and & operator
 - ✓ Modern exploits can convert a memory vulnerability to the primitives of AAR/AAW and addressOf (leaking the address of any JS object)
 - ✓ [Mitigation bounty toolkit by @mxatone \(Thomas Garnier\)](#)
- JavaScript does not support complex byte structure (such as C-style)
 - ✓ Not a big problem if working with AAR/AAW and addressOf primitives
 - ✓ [jsStruct](#)

Calling System APIs

- Call function from JS layer with the ability of controlling all arguments and obtaining the return value
 - ✓ << Mitigation bounty—From read-write anywhere to controllable calls >>
 - ✓ The target function needs to be a valid CFG target
- If CFG is bypassed, the same goal can be accomplished using the code-reuse technique
 - ✓ An ROP gadget chain sets up arguments, transfers control to the target function, and retrieves the return value
 - ✓ Certain existing functions can complete the above operation (such as `Js::JavascriptFunction::CallAsmJsFunction<int>` in WIP 17101)

Js::JavascriptFunction::CallAsmJsFunction<int>

```
.text:0000000180280AC0 __int64__fastcall Js_JavascriptFunction_CallAsmJsFunction_int_Js_this_ proc near
.text:0000000180280AC0 ; CODE XREF: j_Js_JavascriptFunction_CallAsmJsFunction+j
.text:0000000180280AC0 ; j_Js_JavascriptFunction_CallAsmJsFunction_0+j ...
.text:0000000180280AC0
.text:0000000180280AC0 var_s0      = qword ptr 0
.text:0000000180280AC0 arg_0       = qword ptr 20h
.text:0000000180280AC0 arg_8       = qword ptr 28h
.text:0000000180280AC0 arg_10      = qword ptr 30h
.text:0000000180280AC0 arg_18      = qword ptr 38h
.text:0000000180280AC0 arg_20      = qword ptr 40h
.text:0000000180280AC0
.text:0000000180280AC0         mov    [rsp-18h+arg_0], rcx
.text:0000000180280AC5         mov    [rsp-18h+arg_8], rdx
.text:0000000180280ACA         mov    [rsp-18h+arg_10], r8
.text:0000000180280ACF         mov    [rsp-18h+arg_18], r9
.text:0000000180280AD4         push   rsi
.text:0000000180280AD5         push   rdi
.text:0000000180280AD6         push   rbp
.text:0000000180280AD7         mov    rbp, rsp
.text:0000000180280ADA         and   rbp, 0xFFFFFFFFFFFFFFF0h
.text:0000000180280ADE         lea    rax, [r9+10h]
.text:0000000180280AE2         cmp   rax, 2000h
.text:0000000180280AE8         jl    short loc_180280AEF
.text:0000000180280AEA         call  __alloca_probe
.text:0000000180280AEF
.text:0000000180280AEF loc_180280AEF: : CODE XREF: __int64__fastcall Js_JavascriptFunction_CallAsmJsFunction_int_Js_this_+28+j
.subs
.text:0000000180280AEF         sub   rsp, rax
.text:0000000180280AF2         mov   [r8], rcx
.text:0000000180280AF5         mov   rcx, rax
.text:0000000180280AF8         shr   rcx, 3
.text:0000000180280AFC         mov   rsi, r8
.text:0000000180280AFF         mov   rdi, rsp
.text:0000000180280B02         rep   movsq
.text:0000000180280B05         mov   rax, rdx
.text:0000000180280B08         mov   rcx, [rsp+var_s0]
.text:0000000180280B0C         mov   r10, [rbp+arg_20]
.text:0000000180280B10         mov   rdx, [r10]
.text:0000000180280B13         movaps xmm1, xmmword ptr [r10]
.text:0000000180280B17         mov   r8, [r10+10h]
.text:0000000180280B1B         movaps xmm2, xmmword ptr [r10+10h]
.text:0000000180280B20         mov   r9, [r10+20h]
.text:0000000180280B24         movaps xmm3, xmmword ptr [r10+20h]
.text:0000000180280B29         call  cs:_guard_dispatch_icall_fptr
.text:0000000180280B2F         lea    rsp, [rbp+0]
.text:0000000180280B33         pop   rbp
```

Prepare arguments and
call the AsmJs function

Calling Function From JS Layer via RPC Mechanism

Disassembly

Offset: RPCRT4!Invoke+0x4f

00007ffe`976aeb45 f348a5	rep movs qword ptr [rdi],qword ptr [rsi]
00007ffe`976aeb48 498bfa	mov rdi,r10
00007ffe`976aeb4b 498bca	mov rcx,r10
00007ffe`976aeb4e e89dffffff	call RPCRT4!RpcInvokeCheckICall (00007ffe`976aeaf0)
00007ffe`976aeb53 4c8bd7	mov r10,rdi
00007ffe`976aeb56 488b0c24	mov rcx,qword ptr [rsp]
00007ffe`976aeb5a f30f7e0424	movq xmm0,mmword ptr [rsp]
00007ffe`976aeb5f 488b542408	mov rdx,qword ptr [rsp+8]
00007ffe`976aeb64 f30f7e4c2408	movq xmm1,mmword ptr [rsp+8]
00007ffe`976aeb6a 4c8b442410	mov r8,qword ptr [rsp+10h]
00007ffe`976aeb6f f30f7e542410	movq xmm2,mmword ptr [rsp+10h]
00007ffe`976aeb75 4c8b4c2418	mov r9,qword ptr [rsp+18h]
00007ffe`976aeb7a f30f7e5c2418	movq xmm3,mmword ptr [rsp+18h]
00007ffe`976aeb80 41ffd2	call r10

A CFG check is enforced prior to invoking RPC

Command

```
0:018> kv
# Child-SP      RetAddr          : Args to Child
00 00000069`571f83f8 00007ffe`976aeb83 : 00000247`0d3557c8 00000000`00000000 00000000`00000000 00000000`00000000 : Call Site
01 00000069`571f8400 00007ffe`9764a737 : 00000069`571f88f0 00000247`0d354258 00000069`571f8b98 00000069`571f88f0 : KERNEL32!LoadLibraryAStub
02 00000069`571f84a0 00007ffe`976951e3 : 0000024f`67c71e60 00000247`15f3655b 00000000`00000193 00007ffe`7a13128a : RPCRT4!Invoke+0x73
03 00000069`571f8b40 00007ffe`7a11f2d3 : 00000069`571f8d60 0000024f`00000001 0000024f`67c5ddb0 00007ffe`7a11f25d : RPCRT4!NdrStubCall2+0x397
04 00000069`571f8b70 00007ffe`79fd0ddc : 00000247`0d354258 00000000`00000008 0000024f`67cb4690 0000024f`67c5ddb0 : RPCRT4!NdrServerCall2+0x23
05 00000069`571f8bc0 00007ffe`7a022629 : 00000247`0d353048 00007ffe`976951c0 00000069`571f8c40 0000024f`64632890 : chakra!amd64_CallFunction+0x93
06 00000069`571f8c20 00007ffe`7a0273bd : 00000069`571f8d60 00000247`15f36569 00000247`149d5b70 00000069`571f8d60 : chakra!Js::JavascriptFunction::CallFunction<1>+0x8c
07 00000069`571f8c70 00007ffe`7a02704c : 00000069`571f8d60 00000000`00000000 00000000`00000000 00000000`ffffffff : chakra!Js::InterpreterStackFrame::OP_CallI<Js::OpLayoutDynamic>+0x10
08 00000069`571f8cd0 00007ffe`79fa548a : 00000069`571f8d60 0000024f`77f27450 00000069`571f8f00 0000024f`64632890 : chakra!Js::InterpreterStackFrame::Process+0x1dc
09 00000069`571f8d30 00007ffe`79fa4fbe : 0000024f`67c71380 00000069`571f90d0 0000024f`77f70dd2 00000069`571f90e8 : chakra!Js::InterpreterStackFrame::InterpreterHelper+0x4aa
0a 00000069`571f90a0 0000024f`77f70dd2 : 00000069`571f9120 00000069`571f94a0 00000001`00070001 0000024f`77f70000 : chakra!Js::InterpreterStackFrame::InterpreterThunk+0x5e
0b 00000069`571f90f0 00007ffe`7a11f2d3 : 0000024f`67c71380 00000000`10000002 0000024f`67c5ddb0 00000247`118c9bc0 : chakra!Js::InterpreterStackFrame::InterpreterThunk+0x5e
0c 00000069`571f9120 00007ffe`79fd0ddc : 0000024f`64632890 00000000`00000010 00000247`00000002 0000024f`67c5ddb0 : chakra!amd64_CallFunction+0x93
```

Calling Arbitrary System APIs

- Because a CFG check (RPCRT4!InvokeCheckICall) is enforced in the RPCRT4!Invoke function, mxatone's method can call only a CFG-friendly function
- When the CFG check in RPC runtime module (RPCRT4.dll) is completely disarmed, we can call an arbitrary function through RPC
 - ✓ RPCRT4!__guard_check_icall_fptr -> ret
 - ✓ RPCRT4!__guard_dispatch_icall_fptr -> jmp rax
- The fptr of the CFG check routine is in IAT that is write protected, but most CFG bypasses can be exploited to make read-only memory writable (calling VirtualProtect)

Calling Arbitrary Function via RPC Mechanism

Disassembly

Offset: 08f67d44

00007ffa`08f67d2b 498bca	mov rcx,r10
00007ffa`08f67d2e e89dffff	call RPCRT4!RpcInvokeCheckICall (00007ffa`08f67cd0)
00007ffa`08f67d33 4c8bd7	mov r10,rdi
00007ffa`08f67d36 488b0c24	mov rcx,qword ptr [rsp]
00007ffa`08f67d3a f30f7e0424	movq xmm0,mmword ptr [rsp]
00007ffa`08f67d3f 488b542408	mov rdx,qword ptr [rsp+8]
00007ffa`08f67d44 f30f7e4c2408	movq xmm1,mmword ptr [rsp+8]
00007ffa`08f67d4a 4c8b442410	mov r8,qword ptr [rsp+10h]
00007ffa`08f67d4f f30f7e542410	movq xmm2,mmword ptr [rsp+10h]
00007ffa`08f67d55 4c8b4c2418	mov r9,qword ptr [rsp+18h]
00007ffa`08f67d5a f30f7e5c2418	movq xmm3,mmword ptr [rsp+18h]
00007ffa`08f67d60 41ffd2	call r10
00007ffa`08f67d63 488b7528	mov rsi,qword ptr [rbp+28h]

Command

```
0:023> u RPCRT4!RpcInvokeCheckICall L2
RPCRT4!RpcInvokeCheckICall:
00007ffa`08f67cd0 4883ec28    sub    rsp,28h
00007ffa`08f67cd4 ff15e6470700 call   qword ptr [RPCRT4!_guard_check_icall_fptr (00007ffa`08fdc4c0)]
0:023> u poi(RPCRT4!_guard_check_icall_fptr) L1
ntdll!LdrQueryModuleInfoLocalLoaderLock32:
00007ffa`096c3c00 c20000    ret    0
0:023> kv
# Child-SP      RetAddr       : Args to Child
00 000000c0`f2c9c278 00007ffa`08f67d63 : 00000000`001fffff 00000000`00000000  f160c000`00000dcc 00000000`00000000 : Call Site
01 000000c0`f2c9c280 00007ffa`08f2450f : 000000c0`f2c9c480 000000c0`f2c9c770 000002a4`18513258 000000c0`f2c9ca18 : KERNELBASE!OpenThread
02 000000c0`f2c9c320 00007ffa`08f2739a : 000000c0`f2c9c9f0 00000000`00000000 000000c0`f2c9ca18 00007ff9`ee6d349a : RPCRT4!Invoke+0x73
03 000000c0`f2c9c9c0 00007ff9`ee6d1393 : 000002ac`6921cb90 00007ff9`00000001 000002a4`21d002b0 00007ff9`ee6d131d : RPCRT4!NdrStubCall2+0x38f
04 000000c0`f2c9c9f0 00007ff9`ee59d873 : 000002a4`18513258 00000000`00000008 00000000`00000000 00007ff9`ee460000 : RPCRT4!NdrServerCall2+0x1a
05 000000c0`f2c9ca40 00007ff9`ee5a0490 : 000002a4`18512048 00007ffa`08f27380 000000c0`f2c9cac0 000002ac`3f48d470 : chakra!amd64_CallFunction+0x93
06 000000c0`f2c9caa0 00007ff9`ee5a4f35 : 000000c0`f2c9cc10 000002ac`693e7ce7 000002ac`21d00310 000000c0`f2c9cc10 : chakra!Js::JavascriptFunction::CallFunction<1>+0x83
07 000000c0`f2c9caf0 00007ff9`ee5a4b07 : 000000c0`f2c9cc10 00000000`00000000 00000000`00000000 00000000`00000000 : chakra!Js::InterpreterStackFrame::OP_CallI<Js::OpLayoutDynamicProfile>+0x315
08 000000c0`f2c9cb80 00007ff9`ee5a8b5e : 000000c0`f2c9cc10 000002ac`693a6be0 000000c0`f2c9cdc0 000000c0`f2c9cf00 : chakra!Js::InterpreterStackFrame::ProcessUnprofiled+0x315
09 000000c0`f2c9cbe0 00007ff9`ee5aa265 : 000002ac`69231440 000000c0`f2c9cf90 000002ac`69410dd2 000000c0`f2c9cf8 : chakra!Js::InterpreterStackFrame::InterpreterHelper+0x48e
0a 000000c0`f2c9cf60 000002ac`69410dd2 : 000000c0`f2c9cfe0 000000c0`f2c9d380 000002ac`69230a20 000002ac`69410000 : chakra!Js::InterpreterStackFrame::InterpreterThunk+0x55
0b 000000c0`f2c9cfb0 00007ff9`ee6d1393 : 000002ac`69231440 00000000`10000002 000002ac`6921cb90 000002a4`28491260 : 0x000002ac`69410dd2
0c 000000c0`f2c9cfe0 00007ff9`ee59d873 : 000002ac`3f48d470 00000000`00000010 00000000`00000080 00007ff9`ee460000 : chakra!amd64_CallFunction+0x93
```

The CFG check on RPCRT4.dll is disabled

Microsoft Made Great Efforts to Improve CFG

Mitigation	In scope	Out of scope
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.	<ul style="list-style-type: none">• Hijacking control flow via return 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• Bypasses that rely on CONTEXT record corruption• Bypasses that rely on race conditions or exception handling• Bypasses that rely on thread suspension• Instances of missing CFG instrumentation prior to an indirect call

CFG Bypasses Still Effective on RS3/RS4

- Attacking mutable read-only memory
 - ✓ Making arbitrary read-only memory writable via loader work mechanism ([HITB 2017](#))
 - ✓ Unlocking .mrdata of ntdll.dll through thread suspension ([KCon 2017](#))
 - ✓ Unlocking .mrdata of ntdll.dll/eShims.dll through fault injection ([KCon 2017](#))
 - ✓ Unlocking .mrdata of eShims.dll through race condition ([TSec 2017](#))
- Corrupting thread's context record
 - ✓ Corrupting thread context in ntdll!LdrInitializeThunk through thread suspension and stack address leaking techniques ([XCon 2017](#))
- Hijacking function pointer
 - ✓ Hijacking AsmJs entryPoint in Js::JavascriptFunction::CallAsmJsFunction<int> through race condition (fixed in RS4)
 - ✓ Hijacking exception/unwind handlers through race condition

A Race Condition Issue in CallAsmJsFunction

```
.text:000000018028B560 __int64 __fastcall Js::JavascriptFunction::CallAsmJsFunction<int>(Js *this)
...
.text:000000018028B58A          mov    rcx, rdx // the AsmJs function entryPoint (FunctionEntryPointInfo->jsMethod)
.text:000000018028B58D          push   rdx
.text:000000018028B58E          push   r8
.text:000000018028B590          push   r9
.text:000000018028B592          call   amd64_CheckICall // check whether the entryPoint is a valid call target
                                //amd64_CheckICall is a wrapper function of "call cs:_guard_check_icall_fptr"
.text:000000018028B597          pop    r9
.text:000000018028B599          pop    r8
.text:000000018028B59B          pop    rdx
.text:000000018028B59C          mov    rax, rcx // rax holds the address of call target
.text:000000018028B59F          mov    rcx, rdi
.text:000000018028B5A2          xor    r10d, r10d
.text:000000018028B5A5          mov    rsi, r9
.text:000000018028B5A8          add    rsi, 8
.text:000000018028B5AC          push   rax // push the call target to stack
.text:000000018028B5AD          push   rcx
.text:000000018028B5AE          sub    rsp, 20h
.text:000000018028B5B2          call   ?GetArgsSizesArray@Js@@YAPEAIPEAVScriptFunction@1@Z
.text:000000018028B5B7          mov    r12, rax
.text:000000018028B5BA          add    rsp, 20h
.text:000000018028B5BE          pop    rcx
.text:000000018028B5BF          pop    rax // pop the call target from stack
.text:000000018028B5C0          push   rax // push it to stack again
.text:000000018028B5C1          push   rcx
.text:000000018028B5C2          sub    rsp, 20h
.text:000000018028B5C6          call   ?GetStackSizeForAsmJsUnboxing@Js@@YAHPEAVScriptFunction@1@Z
.text:000000018028B5CB          mov    r13, rax
.text:000000018028B5CE          add    rsp, 20h
.text:000000018028B5D2          pop    rcx
.text:000000018028B5D3          pop    rax // pop the call target from stack again
.text:000000018028B5D4          sub    rsp, r13
...
.text:000000018028B661          call   rax // ultimately, issue the AsmJs function call
...
.text:000000018028B66F sub_18028B560 endp
```

The time window for the race condition attack

CFG Bypass by Hijacking Function Entry Point in CallAsmJsFunction

Bypass CFG/RFG by Corrupting Fptr on Stack

Bypass CFG/RFG by Corrupting Fptr on Stack

```
Starting PoC 'Bypass CFG/RFG by Corrupting Fptr on Stack' at Sat Jul 15 2017 02:03:13 GMT-0700 (Pacific Daylight Time)
Bypass CFG/RFG by Corrupting Fptr on Stack
ThreadContext @ 0x2aa44683e90
stack address @ 0xffffd1c9edb8
Script engine is executing!
AsmJs stack frame is found!
ByteCode @ 0x2aa5a56bf78
FunctionEntryPointInfo @ 0x2aa71e67dc0
```

Pid 9132 - WinDbg:10.0.14321.1024 AMD64

File Edit View Debug Window Help

Command - Pid 9132 - WinDbg:10.0.14321.1024 AMD64

```
ModLoad: 00007ffe`eb4a0000 00007ffe`eb52a000 C:\WINDOWS\SYSTEM32\firewallapi.dll
ModLoad: 00007ffe`eb440000 00007ffe`eb46b000 C:\WINDOWS\SYSTEM32\fwbase.dll
ModLoad: 00007ffe`e2e20000 00007ffe`e2fa7000 C:\WINDOWS\SYSTEM32\windows.Globalization.dll
(23ac.154c): Break instruction exception - code 80000003 (first chance)
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WINDOWS\SYSTEM32\ntdll.dll -
ntdll!DbgBreakPoint:
00007ffe`f01e57f0 cc          int     3
0:035> bp chakra+508180 "r @r9; eq @r9+0x38 @r9; .echo Overwritten!; g"
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WINDOWS\SYSTEM32\chakra.dll -
0:035> g
onecoreuap\inetcore\urlmon\zones\zoneidentifier.cxx(376)\urlmon.dll!00007FFEE35531A0: (caller: 00007FFEE3552DAD) ReturnHr(4) tid(2284)
r9=000002aa45df8910
Overwritten!
onecoreuap\inetcore\urlmon\zones\zoneidentifier.cxx(376)\urlmon.dll!00007FFEE35531A0: (caller: 00007FFEE3552DAD) ReturnHr(5) tid(2284)
onecoreuap\inetcore\urlmon\zones\zoneidentifier.cxx(376)\urlmon.dll!00007FFEE35531A0: (caller: 00007FFEE3552DAD) ReturnHr(6) tid(2284)
(23ac.e7c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
chakra!JsDisposeRuntime+0x4c91:
00007ffe`cd30b661 ffd0      call    rax {41414141`41414141}
```

Multithread and Thread Communication

- Web worker is a simple means for running scripts in a background thread separate from the main JS thread
 - ✓ `var worker = new Worker('worker.js')`
- The native thread communication:
 - ✓ `Worker.postMessage()`
 - ✓ `SharedArrayBuffer` with `Atomics` (disabled in Edge due to Meltdown/Spectre attack)

Cloning a SharedArrayBuffer to a Web Worker Thread

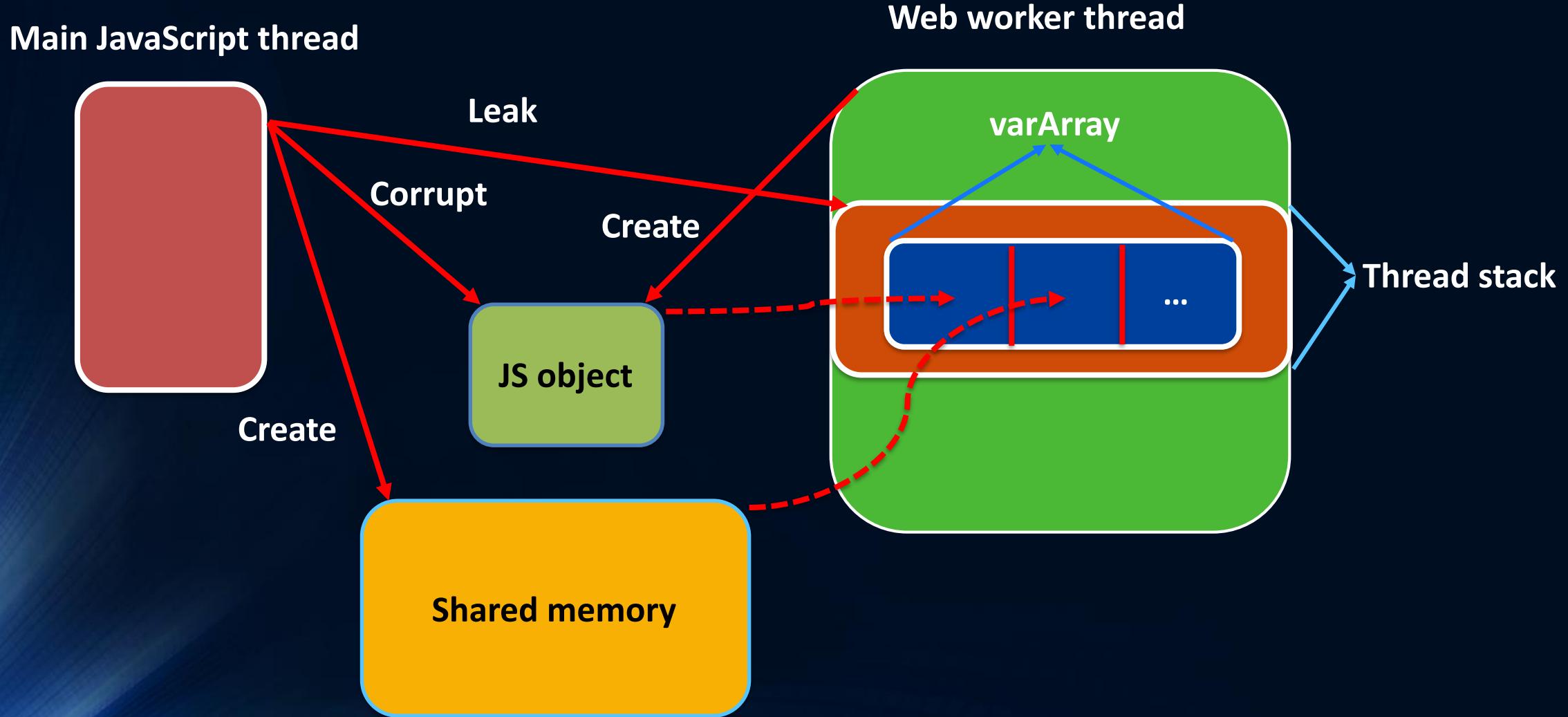
```
0:045> kv
# Child-SP      RetAddr       : Args to Child
00 0000008e`2033e9e8 00007ffc`abf44f9f : 000001d2`7bc2a720 00000000`00000030 00000000`00000000 000001d2`62eed920 : Call Site
01 0000008e`2033e9f0 00007ffc`abcaaab5 : 0000008e`2033ed20 0000008e`00000052 000001d2`00000004 0000008e`2033eb48 : chakra!Js::SharedArrayBuffer
02 0000008e`2033eac0 00007ffc`abcaab5d : 0000008e`2033ec98 0000008e`00000024 0000008e`2033eb48 0000008e`2033ed20 : chakra!Js::DeserializationCloner<Js::StreamReader>::TryCloneObject
03 0000008e`2033eb10 00007ffc`abcaac2e : 0000008e`2033ed20 0000008e`2033eba0 0000008e`2033ed20 0000008e`2033f3f8 : chakra!Js::DeserializationCloner<Js::StreamReader>::ReadObjectProperties
04 0000008e`2033eb70 00007ffc`abcaaaf4 : 0000008e`2033ed20 0000008e`0000001b 0000008e`1361c8a0 000001ca`13611120 : chakra!Js::DeserializationCloner<Js::StreamReader>::CloneProperties
05 0000008e`2033ec20 00007ffc`abcaa7e1 : 0000008e`2033ec98 00000000`00000004 0000008e`2033ec90 000001d2`7bc2a720 : chakra!Js::SCAEngine<unsigned int,void * __ptr64,Js::DeserializationEngine>::Deserialize+0x9a
06 0000008e`2033ec70 00007ffc`abcaa71a : 0000008e`00000004 0000008e`2033ed20 000001d2`62e5f440 00000000`00000000 : chakra!Js::SCAEngine<unsigned int,void * __ptr64,Js::DeserializationEngine>::Deserialize+0x9a
07 0000008e`2033ece0 00007ffc`abcacd2a : 000001d2`7a1848a8 000001ca`62c9faa0 0000008e`2033ee90 000001d2`62e5f440 : chakra!Js::SCAEngine<unsigned int,void * __ptr64,Js::DeserializationEngine>::Deserialize+0x9a
08 0000008e`2033ed70 00007ffc`acbbe746 : 000001d2`62ee1b00 000001ca`62c9faa0 000001d2`7a1d4b40 00000000`00000000 : chakra!ScriptEngineBase::Deserialize+0x1ca
09 0000008e`2033f300 00007ffc`acbbdadb : 000001d2`62ee1b00 00000000`00000000 000001d2`7a1d4b40 0000008e`2033f3b8 : edgehtml!CSCADeserializationContext::CloneVarFromStream+0xbe
0a 0000008e`2033f380 00007ffc`acbbdd7b : 00000000`000000b8 000001d2`7a210c00 000001d2`7a1b0680 00000000`00000000 : edgehtml!CMesssagePort::SetMessageEventData+0x8b
0b 0000008e`2033f3e0 00007ffc`acbbd4c4 : 000001d2`63f0cf50 000001d2`7a1b0680 000001d2`7a1b8800 00000000`00000000 : edgehtml!CMesssagePort::HandlePostMessage+0xa3
0c 0000008e`2033f460 00007ffc`acbbd3dd : 000001d2`63f0cf50 00007ffc`acf5e9d2 00000000`00000000 000001d2`7a18fc0 : edgehtml!CMesssagePort::HandleNotification+0x34
0d 0000008e`2033f4a0 00007ffc`acf4550b : 000001d2`7a1d4c90 0000008e`2033f579 000001d2`7882dec0 00000000`0268f309 : edgehtml!CMesssageDispatcher::ProcessNotification+0x9d
0e 0000008e`2033f4e0 00007ffc`acdbe328 : 00000001a`b14109e8 00000000`002b43e0 00000000`000f4240 00000000`00000000 : edgehtml!GWndAsyncTask::Run+0x1b
0f 0000008e`2033f510 00007ffc`acf27c93 : 00000001a`b14109e8 000001ca`60cffed0 000001d2`63f0cf50 00000000`00000001 : edgehtml!HTML5TaskScheduler::RunReadiedTask+0x208
10 0000008e`2033f5e0 00007ffc`acdfc643 : 000001d2`7a18fcf0 00000000`00000001 000001d2`7a18fd08 00000000`00000000 : edgehtml!HTML5TaskScheduler::RunReadiedTasks+0xe3
11 0000008e`2033f640 00007ffc`acdfc521 : 000001d2`7882de00 00000009`69f55da3 000001ca`62c26400 00000000`00000000 : edgehtml!HTML5EventLoopDriver::DriveLowPriorityTaskExecution+0x11
12 0000008e`2033f680 00007ffc`acdbcf89 : 00000000`00000003 000001ca`62c26400 000001d2`00000000 00000000`00000000 : edgehtml!GlobalWondOnPaintPriorityMethodCall+0x71
13 0000008e`2033f6b0 00007ffc`ce8fb85d : 00000000`001d0624 00000000`00000001 00000000`00000002 00000000`00000000 : edgehtml!GlobalWondProc+0x129
14 0000008e`2033f710 00007ffc`ce8fb54c : 00000000`00000008 00007ffc`acdbce60 00000000`001d0624 00000000`80000000 : user32!UserCallWinProcCheckWow+0x2ad
15 0000008e`2033f880 00007ffc`ce9119c3 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : user32!DispatchClientMessage+0x9c
16 0000008e`2033f8e0 00007ffc`d14c3b14 : 00000000`00000000 00007ffc`acbba380 00007ffc`cb630440 00007ffc`acbbd7bb : user32!fnDWORD+0x33
17 0000008e`2033f940 00007ffc`cd9f1744 : 00007ffc`ce8fb272 000001ca`612d7760 0000008e`2033fa18 00000000`0014092c : ntdll!KiUserCallbackDispatcherContinue (TrapFrame @ 0000008e`2033
18 0000008e`2033f9c8 00007ffc`ce8fb272 : 000001ca`612d7760 0000008e`2033fa18 00000000`0014092c 000001ca`612d7760 : win32u!NtUserDispatchMessage+0x14
19 0000008e`2033f9d0 00007ffc`acd0c682 : 0000008e`2033fa70 000001d2`66cdaab0 000001d2`7a1b8800 00000000`00000000 : user32!DispatchMessageWorker+0x222
1a 0000008e`2033fa50 00007ffc`acd0c60a : 0000008e`2033fb90 00000000`00000000 00000000`00000000 00007ffc`acd13198 : edgehtml!WorkerGlobalScopeThread::RunMessageLoopForSTA+0x36
1b 0000008e`2033fab0 00007ffc`acd0c58b : 000001d2`66cdaab0 000001d2`66cdaab0 00000000`00000000 00000000`00000000 : edgehtml!WorkerGlobalScopeThread::RunMessageLoop+0x2e
1c 0000008e`2033fb10 00007ffc`acd0c4da : 000001d2`7a1b8800 000001d2`66cdaab0 00000000`00000000 00000000`00000000 : edgehtml!WorkerGlobalScopeThread::RunWorkerGlobalScope+0x7b
1d 0000008e`2033fb60 00007ffc`acd0c45e : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : edgehtml!WorkerGlobalScopeThread::RunThread+0x5e
1e 0000008e`2033fbco 00007ffc`ceaa1fe4 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : edgehtml!WorkerGlobalScopeThread::ThreadProc+0xe
1f 0000008e`2033fbf0 00007ffc`d148ef91 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : KERNEL32!BaseThreadInitThunk+0x14
20 0000008e`2033fc20 00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : ntdll!RtlUserThreadStart+0x21
```

- `worker.postMessage({sharedBuffer}, [sharedBuffer]); // transfer`
- `worker.postMessage({sharedBuffer}); // clone`

Sharing Objects Between Multiple JS Threads

- With AAR/AAW, it is possible to directly share certain type of JS objects between multiple JS threads:
 - Web worker thread creates a varArray (JavascriptArray) on its stack and stores one certain JS object (such as an ArrayBuffer) in the varArray
 - Main JS thread creates a shared memory (such as an Uint32Array)
 - Main JS thread leaks the stack of the web worker thread and searches the varArray
 - Main JS thread writes the object of shared memory into the leaked varArray, and reads out the object exposed by web worker
 - Main JS thread helps to corrupt that object to grant the worker thread the ability of AAR/AAW
 - Web worker thread acquires the AAR/AAW and the shared memory to communicate with Main JS thread

Diagram of Sharing Objects Between JS Threads



Callback Mechanism

- Under some scenarios, the ability to allow native code callback into JS is required (API hook, WindowProc etc.)
- Due to the difference in execution mechanism, it is almost impossible to let the native code call a JS function in a natural fashion
 - ✓ Interpreter mode, no true function address
 - ✓ JIT mode, the JITed function requires context info of the JS engine
- It is a good idea to use another thread to help in arguments preparation and context switching
- `amd64_CallFunction` can be used as a pivot to reach the target JS function

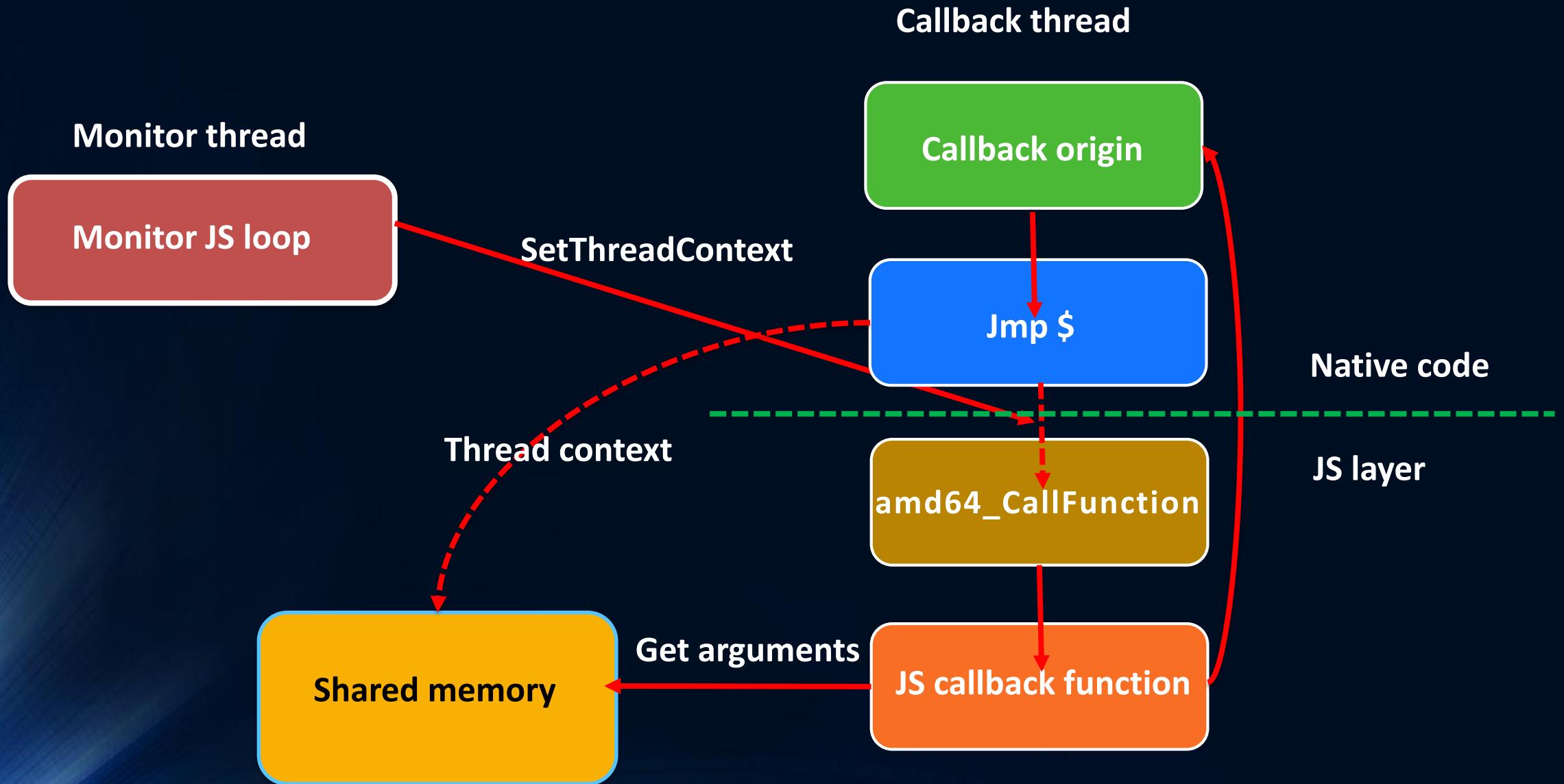
Prototype of chakra!amd64_CallFunction

- var amd64_CallFunction(RecyclableObject *function, JavascriptMethod entryPoint, CallInfo callInfo, uint argc, Var *argv);
 1. function: the JavascriptFunction object of the target JS function
 2. entryPoint: the address of the target JS function ((function + 0x08) -> (Type + 0x18) -> entryPoint)
 3. callInfo: (unused:32 + Flags:8 + Count:24)
 4. argc: the number of arguments
 5. argv: the array of arguments

Calling the JS Function From Native Code

- With the aid of a monitor thread, the control flow can smoothly transition from native code to JS layer:
 - When a user callback needs to be registered, the address of the callback function is set to an instruction of dead loop (`jmp $, \xEB\xFE`)
 - A dedicated monitor thread is created to watch for the occurrence of a user callback (`OpenThread`, `GetThreadContext`)
 - When the user callback occurs (`Context.RIP == jmp $`), the monitor thread saves the current thread context in shared memory, loads the context for the JS layer, and redirects the control flow to `chakra!amd64_CallFunction` (`SetThreadContext`)
 - `amd64_CallFunction` kicks off the usual JS function-calling process in the chakra JS engine
 - The target JS callback function is called and retrieves the original inbound arguments from shared memory, processing according to its logic
 - Finally, `amd64_CallFunction` exits the chakra JS engine and returns to the place where the callback occurred

Diagram of Calling JS Function From Native Code



Restricting SetThreadContext

- A flag in EPROCESS (introduced in RS2) prevents setting the context of a thread of the same process
- On Windows 10 RS2/RS3/RS4, this flag is not enabled for Edge
- Without SetThreadContext, we may explore other approaches (such as ROP) to simulate the context change

EPROCESS.Flags2.RestrictSetThreadContext

The screenshot shows the Immunity Debugger interface with two main panes: the Command pane on the left and the Structures pane on the right.

Command pane:

```
0: kd> dt nt!_EPROCESS fffff800a3cd9c080
+0x000 Pcb : _KPROCESS
+0x2d8 ProcessLock : EX_PUSH_LOCK
+0xe0 UniqueProcessId : 0x00000000`000004ec Void
+0xe8 ActiveProcessLinks : _LIST_ENTRY [ 0xfffff800a`3d4c5868 - 0xfffff800a`3b69186 ]
+0xf8 RundownProtect : EX_RUNDOWN_REF
+0x300 Flags2 : 0xd094
+0x300 JobNotReallyActive : 0y0
+0x300 AccountingFolded : 0y0
+0x300 NewProcessReported : 0y1
+0x300 ExitProcessReported : 0y0
+0x300 ReportCommitChanges : 0y1
+0x300 LastReportMemory : 0y0
+0x300 ForceWakeCharge : 0y0
+0x300 CrossSessionCreate : 0y1
+0x300 NeedsHandleRundown : 0y0
+0x300 RefTraceEnabled : 0y0
+0x300 PicoCreated : 0y0
+0x300 EmptyJobEvaluated : 0y0
+0x300 DefaultPagePriority : 0y101
+0x300 PrimaryTokenFrozen : 0y1
+0x300 ProcessVerifierTarget : 0y0
+0x300 RestrictSetThreadContext : 0y0
+0x300 AffinityPermanent : 0y0
+0x300 AffinityUpdateEnable : 0y0
+0x300 PropagateNode : 0y0
+0x300 ExplicitAffinity : 0y0
+0x300 ProcessExecutionState : 0y00
+0x300 EnableRundownLogging : 0y0
```

Structures pane:

```
mov r8, cs:PsThreadType ; ObjectType
mov edx, 10h ; DesiredAccess
mov [r11-58h], rax
mov sil, [rbx+232h]
mov r9b, sil ; AccessMode
call ObReferenceObjectByHandle
mov edi, eax
test eax, eax
js short loc_14070FBCE
mov rcx, rbx
call IoThreadToProcess
mov rbx, [rsp+78h+Object]
mov rdi, rax
test dword ptr [rax+300h], 20000h
jz short loc_14070FB87
mov rcx, rbx
call IoThreadToProcess
cmp rdi, rax
jnz short loc_14070FB87
mov edi, 0C000060Ah
jmp short loc_14070FBB9
:
; CODE XREF: NtSetContextThread+71↑j
; NtSetContextThread+7E↑j
test dword ptr [rbx+74h], 400h
```

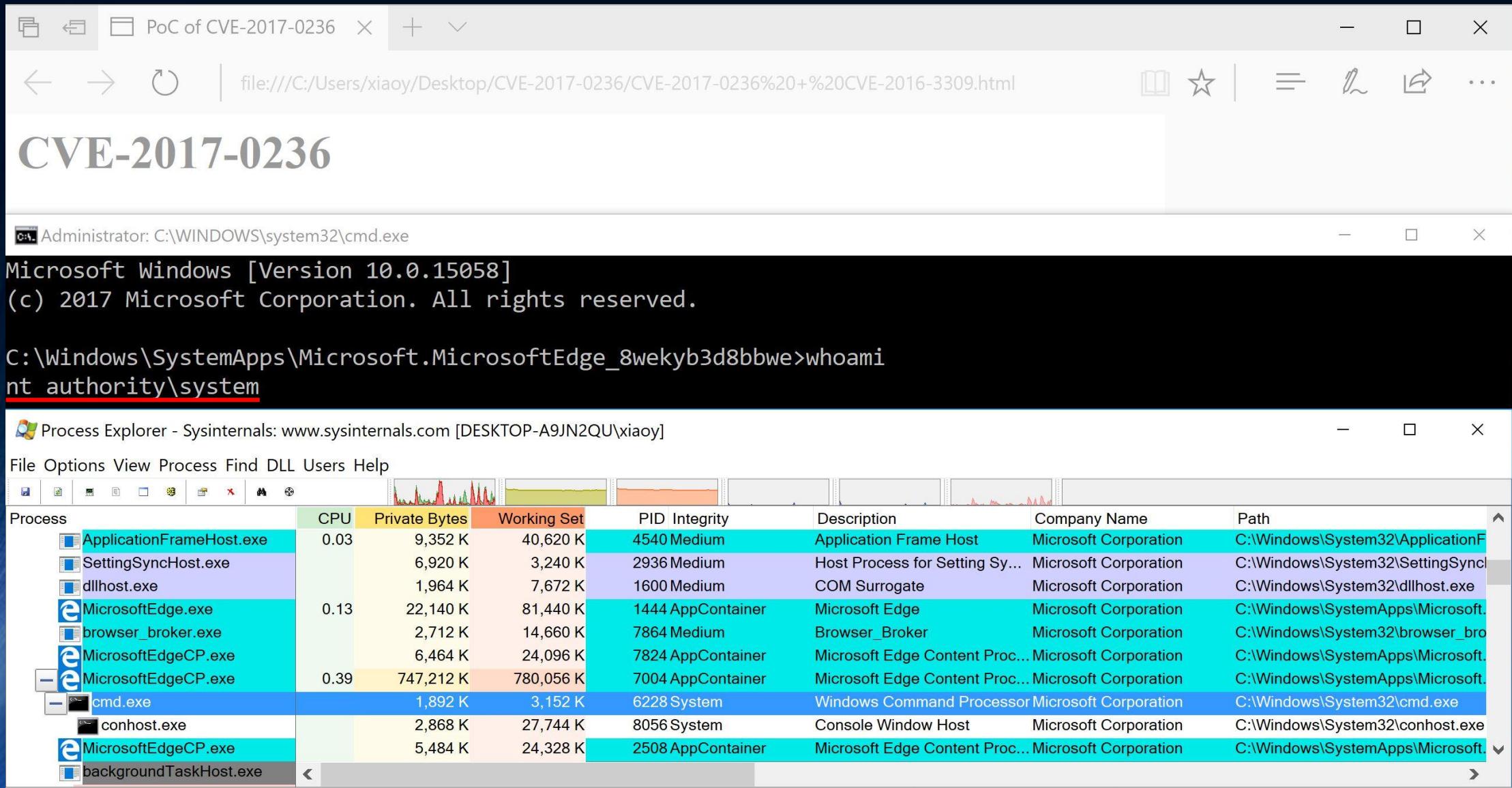
A red arrow points from the assembly code in the Structures pane to the memory dump pane, specifically highlighting the instruction `cmp rdi, rax`. A red annotation on the right side of the Structures pane reads: "Compare the belonging process of the calling thread with that of the target thread".

Demo: WindowProc Callback

The image shows two windows side-by-side. On the left is a Microsoft Edge browser window titled "PoC of CVE-2017-0236". The URL is "file:///C:/Users/xiaoy/Desktop/CVE-2017-0236/CVE-2017-0236". The page content displays "CVE-2017-0236" and a message "This site says... callback 0x60446 0x24 0x0 0x95c94f9ee0". A modal dialog box is partially visible with the word "Ok". On the right is a debugger window titled "Command" showing a call stack. The stack trace includes numerous entries from the Windows API and Edge's internal code, such as ntdll!NtWaitForMultipleObjects, KERNELBASE!WaitForMultipleObjectsEx, USER32!RealMsgWaitForMultipleObjectsEx, EdgeContent!SHProcessMessagesUntilEventsEx, EdgeContent!SpartanCore::ModalUISyncHandler::Wait, EdgeContent!CBrowserTab::CWPCHost::PopupDialogManager::ShowInformationDialog, EdgeContent!CBrowserTab::CWPCHost::OnShowMessage, edgehtml!CWebPlatformTridentHost::OnShowMessage, edgehtml!CDoc::ShowMessageEx, edgehtml!CWindow::AlertHelper, edgehtml!CFastDOM::CWindow::Trampoline_alert, edgehtml!CFastDOM::CWindow::Profiler_alert, chakra!Js::JavascriptExternalFunction::ExternalFunctionThunk, chakra!amd64_CallFunction, chakra!Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js>>, chakra!Js::InterpreterStackFrame::OP_ProfiledCallIWithICIndex<Js::OpLayoutT_Caller>, chakra!Js::InterpreterStackFrame::ProcessProfiled, chakra!Js::InterpreterStackFrame::Process, chakra!Js::InterpreterStackFrame::InterpreterHelper, chakra!Js::InterpreterStackFrame::InterpreterThunk, chakra!Js::ByteCodeWriter::TryWriteElementSlotI2<Js::LayoutSizePolicy<1>>, USER32!UserCallWinProcCheckWow, USER32!DispatchClientMessage, USER32!fnINOUTLPOINT5, ntdll!KiUserCallbackDispatcherContinue (TrapFrame @ 00000095`c94f9d38), win32u!NtUserCreateWindowEx, USER32!VerNtUserCreateWindowEx, USER32!CreateWindowInternal, USER32!CreateWindowExW, RPCRT4!Invoke, RPCRT4!NdrStubCall12, RPCRT4!NdrServerCall12, and chakra!amd64_CallFunction.

```
Call Site
0dc4b0 000001b6`e532e550 : ntdll!NtWaitForMultipleObjects+0x14
000000 00000000`00000654 : KERNELBASE!WaitForMultipleObjectsEx+0xf0
000000 00000000`00000000 : USER32!RealMsgWaitForMultipleObjectsEx+0x160
000001 00000000`ffffffff : EdgeContent!SHProcessMessagesUntilEventsEx+0xc4
0c4000 000001be`e84c4050 : EdgeContent!SpartanCore::ModalUISyncHandler::Wait+0x3e
000002 00000095`c94f8098 : EdgeContent!CBrowserTab::CWPCHost::PopupDialogManager::ShowInformationDialog+0x188
1f8390 000001be`e8499540 : EdgeContent!CBrowserTab::CWPCHost::OnShowMessage+0x188
000000 00000000`0000092e : edgehtml!CWebPlatformTridentHost::OnShowMessage+0x42
000000 00000000`00002026 : edgehtml!CDoc::ShowMessageEx+0x1da
640000 00000095`c94f93b8 : edgehtml!CWindow::AlertHelper+0x17b
3a3033 000001be`00000026 : edgehtml!CFastDOM::CWindow::Trampoline_alert+0x98
1f94c0 000001be`e9e0d5e0 : edgehtml!CFastDOM::CWindow::Profiler_alert+0x25
6c8a0 000001bf`11986fb0 : chakra!Js::JavascriptExternalFunction::ExternalFunctionThunk+0x1d0
1f9540 000001bf`119819c0 : chakra!amd64_CallFunction+0x93
ecd340 00007ff8`00000074 : chakra!Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js>>
000000 000001be`00000006 : chakra!Js::InterpreterStackFrame::OP_ProfiledCallIWithICIndex<Js::OpLayoutT_Caller>
000000 00000000`00000000 : chakra!Js::InterpreterStackFrame::ProcessProfiled+0x131
1f99a0 402e0000`00000001 : chakra!Js::InterpreterStackFrame::Process+0x12c
40d1a 00000095`c94f9b98 : chakra!Js::InterpreterStackFrame::InterpreterHelper+0x4ac
40b440 00007ff8`133a2ba0 : chakra!Js::InterpreterStackFrame::InterpreterThunk+0x51
e10020 00000000`00000001 : 0x0000001be`fa140d1a
65fbcc 00007ff8`334a8876 : chakra!amd64_CallFunction+0x93
55c38 00007ff8`32c1bc50 : chakra!Js::ByteCodeWriter::TryWriteElementSlotI2<Js::LayoutSizePolicy<1>>+0x7
001440 00000000`00000000 : chakra!Js::TaggedInt::Add+0x51
60446 00000000`80000000 : USER32!UserCallWinProcCheckWow+0x280
000000 000001be`e840b440 : USER32!DispatchClientMessage+0x9c
000070 00007ff8`334b90b4 : USER32!fnINOUTLPOINT5+0x32
ef0000 00000000`40000600 : ntdll!KiUserCallbackDispatcherContinue (TrapFrame @ 00000095`c94f9d38)
000600 00007ff8`13260001 : win32u!NtUserCreateWindowEx+0x14
000000 00000000`00000000 : USER32!VerNtUserCreateWindowEx+0x212
1fa9f0 00007ff8`32c26840 : USER32!CreateWindowInternal+0x1b5
e020c0 00000000`00000000 : USER32!CreateWindowExW+0x82
1fac38 00000095`c94fa990 : RPCRT4!Invoke+0x73
1fac38 00007ff8`133a4b8a : RPCRT4!NdrStubCall12+0x38f
2900a0 00007ff8`133a2fb0 : RPCRT4!NdrServerCall12+0x1a
0d630 000001be`00000006 : chakra!amd64_CallFunction+0x93
```

Demo: CVE-2017-0236 + CVE-2016-3309



Conclusion

- ACG+CIG+CFG raise the bar very high for making reliable exploits
- Compared with ACG and CIG, CFG still appears to be weak
- All mitigation features must work collaboratively to be effective
- The interactive runtime provides strong scripting capability that can be leveraged to create a shellcode-free exploit/payload

Q&A and Acknowledgements

- Send questions to bing_sun@mcafee.com, chong_xu@mcafee.com
- Thanks to MSRC for helping to analyze and confirm these issues in a timely manner
- Special thanks to Haifei Li, Dan Sommer, and McAfee IPS Security Research Team