Microsoft Internet Explorer - Use-After-Free (CVE-2020-0674) -- Micky Thongam (deadbeefcafeh)

Background: A use-after-free bug in Internet Explorer was discovered and found to be exploited in the wild by Qihoo 360 back in 2020. The vulnerability exists in a legacy (still there for compatibility reasons) scripting engine on Internet Explorer, "Jscript.dll". Affected systems and versions include: IE 8 to 11 and Windows update patch "KB4534251" and below.

Project Description: You will find exploits and write-ups for this vulnerability on exploit-db by **maxpl0it** and **Forrest Orr** (refer link below) so, the purpose of this project was to recreate and learn more about the bug, the exploitation techniques used and hopefully port it to Windows8.1 and Windows10 systems.

Vulnerability:

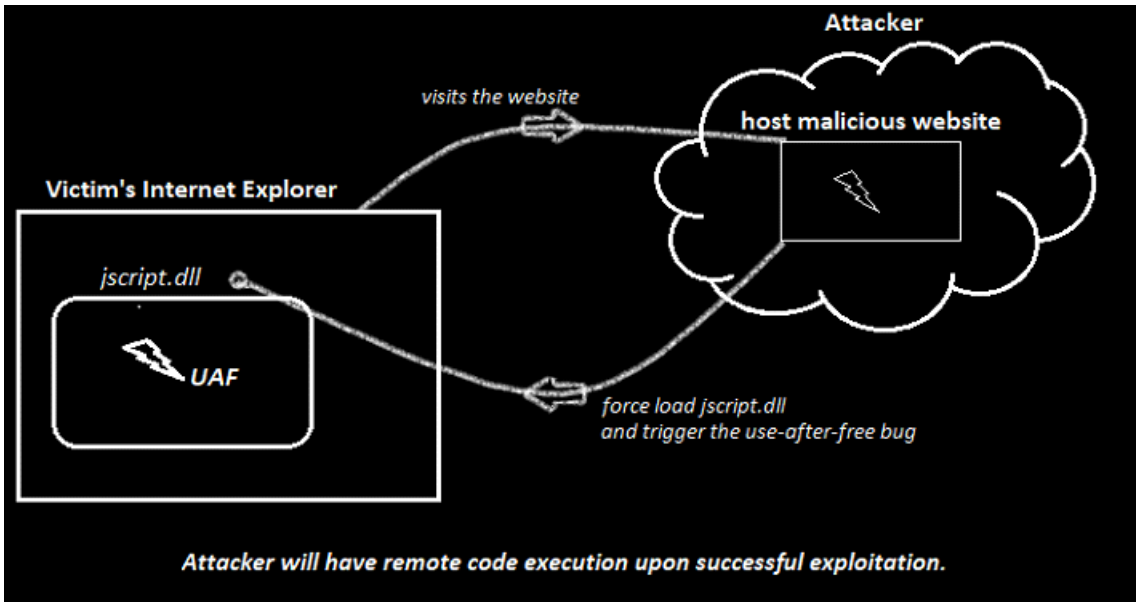

figure a: attack scenario

Proof-Of-Concept (UAF trigger code):

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="x-ua-compatible" content="IE=EmulateIE8"/>
        <script language="JScript.Compact">   //necessary for invoking the
jscript.dll

              function trig_uaf(untrack1, untrack2){    //0
                     var spray = new Array();
                     for(var i = 0; i < 20000; i++) spray[i] = new Object();  //1
                     untrack1 = spray[7777];   //2
                     spray = new Array();      //3
                     CollectGarbage();         //4
                     alert(untrack1);          //5
                     return 0;
              }
              [0,0].sort(trig_uaf);            //6
              </script>
       </head>
</html>
```

Technical Details: The vulnerability is a USE-AFTER-FREE bug in jscript.dll, in an Array object's sort function when a callback function is **executed**(*6*).
The **arguments**(*0*) in this callback function is not tracked by Garbage collector and thus can be used to cause a use-after-free.
The array "spray" is deallocated when, **freed**(*3*) and **collect garbage**(*4*) even though "untrack1" still holds a **reference**(*2*).

Exploits: I've crafted two distinct exploits, one designed for Windows 8.1 and the other for Windows 10. While they share substantial similarities, the key distinctions lie in the system call numbers and specific offsets.

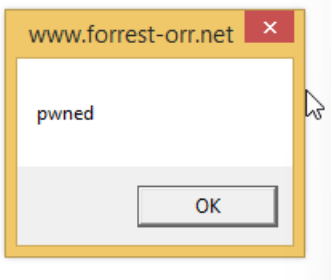

figure b: messagebox after successful code execution

Exploit code available here at my github page.

win8_1.html -> Successfully tested and verified on the Windows 8.1 platform with the '*KB4534251*' update, specifically targeting *32-bit processes*.
The rationale behind this choice is rooted in the default behavior of Internet Explorer, where, despite the parent IE process operating in a 64-bit environment, its child processes (tabs) default to a 32-bit architecture. This condition persists unless explicitly configured to employ the '64-bit process mode' of 'Enhanced Protected Mode' (figure c).

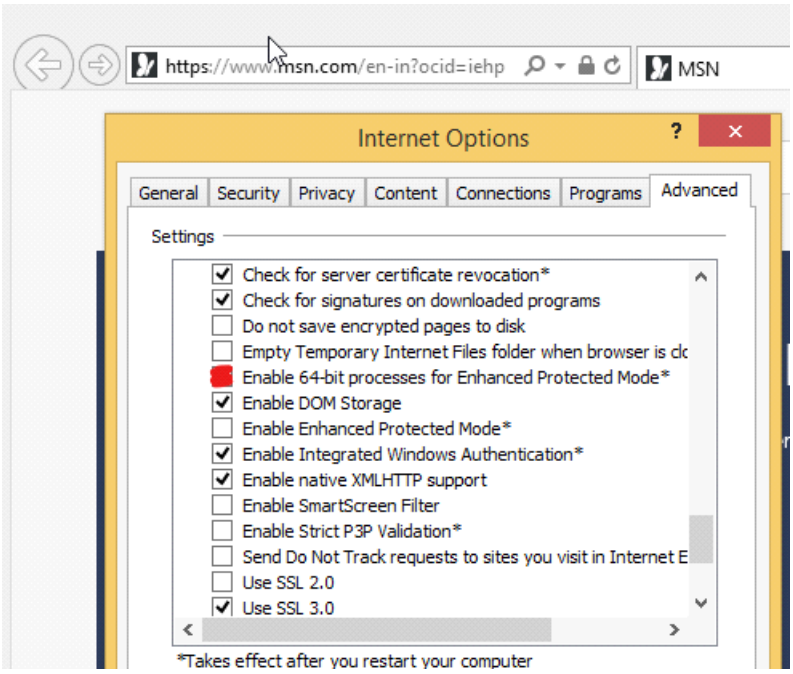

figure c: configure 64-bit process mode

win10.html -> Successfully tested and verified on the Windows 10 platform (*CFG disabled*) with updates below '*KB4534251*' targeting *32-bit processes*.
The unique obstacle encountered on Windows 10 revolves around the supplementary Control Flow Guard (CFG) protection (figure d). I am presently engaged in developing a resolution for this specific issue.

```
(bc.8d4): Security check failure or stack buffer overrun - code c0000409 (!!! second chance !!!)
Subcode: 0xa FAST_FAIL_GUARD_ICALL_CHECK_FAILURE
ntdll_775d0000!RtlFailFast2:
7764d5a0 cd29            int     29h
0:009:x86> k
 # ChildEBP RetAddr
00 07dbc6e4 776a54e1     ntdll_775d0000!RtlFailFast2
01 07dbc710 7764c9d8     ntdll_775d0000!RtlpHandleInvalidUserCallTarget+0x73
02 07dbc798 70604c72     ntdll_775d0000!LdrpValidateUserCallTargetBitMapRet+0x3b
03 07dbc7c0 705ee40e     JSCRIPT!CScriptRuntime::TypeOf+0x130
04 07dbcb7c 705f0850     JSCRIPT!CScriptRuntime::Run+0x197e
05 07dbcc6c 705f094b     JSCRIPT!ScrFncObj::CallWithFrameOnStack+0xa0
06 07dbccc4 705ebaca     JSCRIPT!ScrFncObj::Call+0x7b
```

figure d: control-flow-guard check

Adapting Exploits for x64 Architecture?

To facilitate the transition of these exploits to x64 architecture, it is imperative to carefully adjust the sizes and offsets of Windows internals and JavaScript structures.

Exploitation Techniques: This is just a summary on the techniques used for successful '**code execution**' or 'shellcode execution'. For better understanding this whole process, I strongly suggest this great write-up by "Max Van Amerongen" >

- The initial Use-After-Free (UAF) exploit is crafted to exploit a type-confusion bug. Subsequently, this type-confusion vulnerability is leveraged once more to attain an arbitrary read primitive, paving the way for code execution.
  - Transitioning from a Use-After-Free (UAF) state to a Type-Confusion scenario involves spraying the heap with thousands of Objects (Object == VAR struct, and GcBlock struct holds VARs),
    freeing them after we got an untracked pointer to one of the Objects, now this untracked pointer will point to a freed memory which originally was pointing to a VAR inside of a GcBlock,
    now what we can do is spray the heap with *VVAL* (another stucture), why VVAL?
    We can control the size of allocation on the heap with VVAL struct, this is important because we will spray the heap with many VVALs with allocation size similar to the size of a GcBlock,
    hoping one of the VVALs we sprayed fall in place of the previously freed GcBlock. And thus we have a type confusion. VVAL acting as GcBlock.
  - Transitioning from Type-Confusion to an Information Leak becomes imperative in the presence of Address Space Layout Randomization (ASLR). Techniques employed for information leakage include:

Reading residual pointers from the preceding GcBlock;
Manipulating the hash value within the VVAL structure, this method is to disclose a pointer to the *next VVAL*, ultimately serving as a means to leak addresses.

- Advancing from Information Leak to a Read Primitive involves a careful strategy utilizing the previously acquired leaked address. The techniques employed include crafting a fake VAR (a string object) and pointing it to the address we aim to leak. Since string objects are stored as BSTR, a reliable method for reading the pointer involves using a trick to extract bytes from the '*length*' property, as illustrated well here

- Leaking the base addresses of DLLs involves a multistep process. Initially, we create an object and read its Vtable. By effectively walking back the address and identifying the DOS stub, we ascertain the base address of JSCRIPT. Subsequently, we traverse the Import Address Table (IAT) of JSCRIPT, retrieve a random function address from the desired DLL, and employ the aforementioned technique again to deduce its base address.

- To attain code execution exclusively through a read primitive, we will create a fake object and trigger a call to its 'Vtable,' which remains under the control of the attacker. Employed techniques include stack pivoting via Return-Oriented Programming (ROP), utilizing NtProtectVirtualMemory to circumvent Data Execution Prevention (DEP), and executing shellcode, among others.

Patch: The identified bug has been addressed in the Windows update "KB4537767." (*simplified*) Invoking a call to "**IScavengerBase::LinkToGc**" establishes a link between the arguments of the callback function and the Garbage Collection mechanism, effectively resolving and patching the bug, as detailed in the aforementioned blog.

Reference links:

Microsoft Internet Explorer 11 32-bit - Use-After-Free > by deadlock (Forrest Orr)

Microsoft Internet Explorer 11 - Use-After-Free 64-bit > by Max Van Amerongen (maxpl0it)

Internet Exploiter: Understanding vulnerabilities in Internet Explorer > by Max Van Amerongen (maxpl0it)

DoubleStar > by deadlock (Forrest Orr)

Actions Speak Browser Than Words (Exploiting n-days for fun and profit) > by Max Van Amerongen (maxpl0it)

By: Micky Thongam (deadbeefcafeh)

Github

Linkedin (micky thongam)