Advanced Exploitation of Internet Explorer Heap Overflow Vulnerabilities (MS12-004)

题注：由仙果翻译，其中肯定有错误之处，欢迎大家指正，感激不尽。
http://hi.baidu.com/zhanglinguo11
原始链接为：
http://www.vupen.com/blog/20120117.Advanced_Exploitation_of_Windows_MS12-004_CVE-2012-0003.php

嗨，大家好！

2012 年刚刚开始 CVE 上就爆出了很多有趣的漏洞。其中一个就是 CVE-2012-0003,一个危急级别的漏洞影响到了微软多媒体库并且与 MIDI 文件处理流程有关。在上个星期已经作为 MS12-004 公告的一部分进行修补。
鉴于此漏洞的危急程度，我们建议尽快打上补丁。

从维基百科上，我们能知道：“MIDI 是一个 1982 年制定的行业标准协议”，基本上在它制定 30 年后，这种格式仍然触发像微软这样的软件厂商的漏洞。

单独来说，这种漏洞非常常见但是漏洞利用并不容易，像 Windows Media Player 或者 Internet Explorer 这样的应用程序解析一个 MIDI 文件时，分配一个静态的堆缓冲区但写入却是超过 0x440 个字节（漏洞触发原因）。为了处理如此大的分配大小，Internet Explorer 下通用的利用技巧并不适用这个漏洞，这才是真正有挑战性的事情。

此博客中，我们将演示说明此漏洞的危急程度，通过 Internet Explorer 9/8/7/6 绕过 ASLR/DEP 实现代码执行。

1.漏洞分析技巧

这是漏洞的关键点。一个 MIDI 文件只要包含 2 种类型结构块，一个名称是 MThd，另一个是 MTrk。下列表格是 2 个部分的结构：

Offset	Field	Size
0	Type = 'MThd'	4
4	Length = 6	2
6	format	2

8	tracks	2
10	division	2

Offset	Field	Size
0	Type = 'MTrk'	4
4	Length	var
var	delta_time	var
var	event	var

处理文件之前，Windows Media 在”winmm.dll”中使用”mseOpen”函数分配 2 个缓冲区：

```
.text:76B5CDB1 mov edi, [ebp+arg_4]
.text:76B5CDB4 mov eax, [edi+10h]
.text:76B5CDB7 lea eax, ds:94h[eax*8]
.text:76B5CDBE cmp eax, 10000h
.text:76B5CDC3 mov [ebp+var_4], 7
.text:76B5CDCA jnb loc_76B5CED7
.text:76B5CDD0 push ebx
.text:76B5CDD1 push esi
.text:76B5CDD2 push eax
.text:76B5CDD3 call winmmAlloc(x)            // 分配一个缓冲区
.text:76B5CDD8 mov esi, eax
.text:76B5CDDA xor ebx, ebx
.text:76B5CDDC cmp esi, ebx
.text:76B5CDDE jz loc_76B5CED5
.text:76B5CDE4 push 400h
.text:76B5CDE9 call winmmAlloc(x)          // 分配第二个大小为 0x400 的缓冲区
```

第二个缓冲区在接下来的处理中标记为”b1”。这个特殊的漏洞的触发原因在于对”MTrk”数据块特定事件的解析上。这些事件首先通过 *quartz.dll* 定义的函数 "*smfReadEvents()*"进行读取。

```
.text:74903483 loc_74903483:
.text:74903483 push [ebp+arg_C]
.text:74903486 lea eax, [ebp+var_14]
.text:74903489 push eax
.text:7490348A push esi
.text:7490348B call smfGetNextEvent(x,x,x) // 读取一个事件并放在 var_8（变量 8）
74903490 test eax, eax
.text:74903492 jnz loc_749035B1
.text:74903498 mov ecx, [ebp+var_8]
.text:7490349B cmp cl, 0F0h
```

```
.text:7490349E jnb short loc_749034EC
```

它的首字节作为事件的 ID，并标示为 e1 e2 e3,因此 ECX=0x00e3e2e1.只有当事件 e1 < 0xF0 时才是有趣的。接下来的代码段为，在之前分配的数组中偏移 8 的位置写入事件。

```
.text:749034B4 loc_749034B4:
.text:749034B4
.text:749034B4 mov eax, [esi+10h]
.text:749034B7 add eax, [ebp+var_14]
.text:749034BA movzx ecx, cl
.text:749034BD mov [edi], eax                  // 写入第一个 dword
.text:749034BF and dword ptr [esi+10h], 0
.text:749034C3 add edi, 4
.text:749034C6 and dword ptr [edi], 0          //写入 0
.text:749034C9 movzx eax, byte ptr [ebp+var_8+2]
.text:749034CD movzx edx, byte ptr [ebp+var_8+1]
.text:749034D1 shl eax, 8
.text:749034D4 or eax, edx
.text:749034D6 shl eax, 8
.text:749034D9 add edi, 4
.text:749034DC or eax, ecx
.text:749034DE
.text:749034DE loc_749034DE:
.text:749034DE mov [edi], eax                  //写入事件
.text:749034E0 add edi, 4
.text:749034E3 add dword ptr [ebx+8], 0Ch //  增加计数器
.text:749034E7 jmp loc_749035A2
```

这个数组会在 *winmm.dll 中的*"*midiOutPlayNextPolyEvent()*" 函数中进行处理：

```
.text:76B5D0B2 mov eax, [ebp+wParam]
.text:76B5D0B5 mov ecx, [ebx+eax]          // 读取一个事件存放在 ECX 中
.text:76B5D0B8 add ebx, 4
.text:76B5D0BB mov eax, ecx
.text:76B5D0BD mov [esi+24h], ebx
.text:76B5D0C0 shr eax, 18h
.text:76B5D0C3 and ecx, 0FFFFFFh           // ecx = e3 e2 e1
```

接下来，程序对是否 e1>7Fh 进行判断：

```
.text:76B5D1B6 loc_76B5D1B6:
```

```
.text:76B5D1B6 cmp [ebp+hmo], ebx
.text:76B5D1B9 mov esi, [edi+84h]
.text:76B5D1BF jz loc_76B5D276
.text:76B5D1C5 test cl, cl                    // cl = e1
.text:76B5D1C7 mov al, cl                     // al = e1
.text:76B5D1C9 mov ebx, ecx
.text:76B5D1CB js short loc_76B5D1E3     // jump if 80h <= e1 <= FFh
[...]
.text:76B5D1E3 loc_76B5D1E3:
.text:76B5D1E3 mov edx, ecx
.text:76B5D1E5 shr edx, 8                     // dl = e2
.text:76B5D1E8 mov [edi+54h], cl
.text:76B5D1EB mov byte ptr [ebp+wParam+3], dl
.text:76B5D1EE shr ebx, 10h                   // bl = e3
```

解下来我们可以看到，e1&F0h=80h 或者 90h 时 Windows Media 特有的处理事件流程：

```
.text:76B5D1F1 loc_76B5D1F1:
.text:76B5D1F1 mov dl, al                  // dl = e1
.text:76B5D1F3 and dl, 0F0h
.text:76B5D1F6 cmp dl, 90h
.text:76B5D1F9 mov [ebp+var_1], dl
.text:76B5D1FC jz short loc_76B5D203
.text:76B5D1FE cmp dl, 80h
.text:76B5D201 jnz short loc_76B5D25F
```

这种情况下，在上面分配的缓冲区b1 的某个偏移写入数据，而这个偏移是通过e1 和e2 计算得来的：

```
.text:76B5D203 loc_76B5D203:
.text:76B5D203 movzx edx, byte ptr [ebp+wParam+3] // edx = e2
.text:76B5D207 and eax, 0Fh                              // eax = e1 & 0Fh
.text:76B5D20A shl eax, 7
.text:76B5D20D add eax, edx
.text:76B5D20F cdq
.text:76B5D210 sub eax, edx
.text:76B5D212 sar eax, 1
.text:76B5D214 cmp [ebp+var_1], 80h
.text:76B5D218 jz short loc_76B5D244
```

计算结果为：EAX = ((*e1* & 0Fh) * 2^7 + e2) / 2。

这是 b1 缓冲区要被修改的数据。如果 e1 和 e2 为特定的值，那么就有可能得到 EAX>400h。例如，e1 = 9Fh => 0fh*2^7 = 780h，然后如果 e2>7Fh,e1+e2>800h 就使得 EAX => 400h,程序写入的数据就会越过分配的缓冲区的界限。

依据 e1 & F0h = 90h 或 80h 或者 e3 = 1，有可能增加或减少一个任意字节。例如 e1 & F0h = 90h:

```
.text:76B5D21E add esi, eax
.text:76B5D220 test byte ptr [ebp+wParam+3], 1
.text:76B5D224 mov al, [esi]                    // 从 b1 中读一个字节
.text:76B5D23E inc al
.text:76B5D240 mov [esi], al                    // 加 1
```

如果 e1 & F0h = 80h:

```
.text:76B5D248 lea edx, [eax+esi]
.text:76B5D24B mov al, [edx]                    //从 b1 中读一个字节
[...]
.text:76B5D25B dec al
.text:76B5D25D mov [edx], al                    // 减 1
```

因为 b1 的长度是 0x400 个字节，当 e1 = 8Fh 或 9Fh 且 e2=7Fh 时就触发一个堆溢出。实际上，它使得破坏 b1 缓冲区之后的 0x40 字节，以足够实现任意代码执行成为可能。

2. 绕过ASLR/DEP的高级利用。

在 Internet Explorer 中可以载入漏洞模块，有可能做到稳定的缺陷利用。

我们知道 winmm.dll 的"DllProcessAttach()"和 mshtml.dll 的"_DllMainStartup()"库函数在他们的分配中使用了同一个堆：

In "DllProcessAttach()":

```
.text:76B43F8F mov eax, large fs:18h             // GetProcessHeap inlined
.text:76B43F95 mov eax, [eax+30h]
.text:76B43F98 mov eax, [eax+18h]
.text:76B43F9B mov _hHeap, eax
```

In "*_DllMainStartup()*":

```
.text:3CEAC930 call GetProcessHeap()
.text:3CEAC936 push eax
```

```
.text:3CEAC937 mov _g_hProcessHeap, eax
```

结果就是，它有可能利用漏洞破坏一个 Internet Explorer 的对象。

一般情况下，利用这类型漏洞的方法是找到大小等于漏洞缓冲区大小的一个对象。在此例中，可以使用以下方法来达到利用的目的：使用一个字符串和一个对象连续分配漏洞缓冲区，一旦完成覆写字符串的长度就可以泄露出 vTable（虚函数表） 并且推导出 mshtml.dll 的基址：

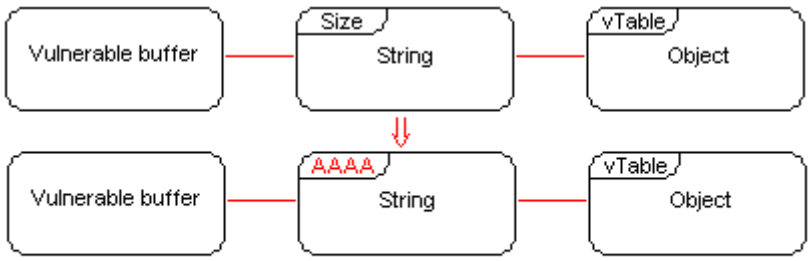


Figure 1 – Heap layout required to disclose a vTable thanks to a fake string's length

然后分配一个新的缓冲区，触发漏洞第二次覆写对象的 vTable(虚函数表)：

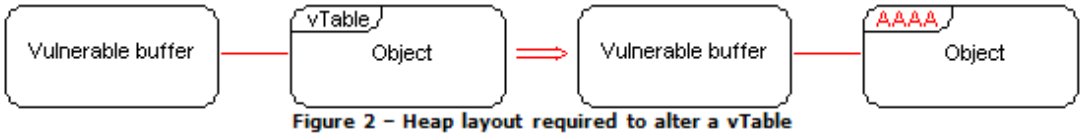


Figure 2 – Heap layout required to alter a vTable

另外，使用 mshtml.dll 的动态 ROP 代码进行堆喷射，从被修改的对象中调用一个函数就可以直接控制程序执行流程。

（不）幸运的是，这个方法需要漏洞触发两次而且只有它可能根据给定的大小去分配一个对象时才有效。给定的 sizes < 0x100 字节，IE 为这种方法实现提供了足够的对象，但是 MS12-004 漏洞特定的情形下，缓冲区的长度是 0x400 字节，在 mshtml.dll 中并没有分配任意一个有趣的对象，并且在 3FCh 或者 400h 出引用了外部数据（意思可以理解为，因为缓冲区的长度为 0x400,并且引用了外部数据，mshtml.dll 并没有分配可以利用的对象）。

创建一个特定的 CImplAry 对象则可以稳定的利用此漏洞。这个方法以创建的大小为 0x400 字节的数组为主要部分，精确地修改它的内容以泄露一个任意的 vTable (虚函数表)并且重定向执行流程。

当克隆一个元素时，就会创建一个数组作为实例。我们知道，"CElement::Clone()" 函数调用 "CElement::CloneAttributes()"，并最终调用

"CAttrArray::Clone()"来克隆属性。以下代码属于"CAttrArray::Clone()"函数：

```
.text:3D06A356 call CAttrArray::operator new(uint) // 分配一个新的 CAttrArray 对象
.text:3D06A35B cmp eax, edi
.text:3D06A35D jz short loc_3D06A368
.text:3D06A35F mov ecx, eax
.text:3D06A361 call CAttrArray::CAttrArray(void)    //初始化对象
.text:3D06A366 mov edi, eax
.text:3D06A368
.text:3D06A368 loc_3D06A368:
.text:3D06A368 test edi, edi
.text:3D06A36A mov esi, [ebp+arg_4]
.text:3D06A36D mov [esi], edi
.text:3D06A36F jz loc_3D0F5DE5
```

同时，程序校验原先的元素是否包含一个属性：

```
.text:3D06A375 mov eax, [ebx+4]
.text:3D06A378 shr eax, 2                              // eax 表示与原来元素相关联的属性数目
.text:3D06A37B js loc_3D053663
.text:3D06A381 cmp eax, [edi+8]
.text:3D06A384 jbe loc_3D0F5DF1
.text:3D06A38A push 10h
.text:3D06A38C call CImplAry::EnsureSizeWorker(uint,long)
```

如果这个数目不为空，IE 将*在"CImplAry::EnsureSizeWorker()"*函数中分配 10h*#*attributes*（*元素属性的数目*）*的数组。因此，如果原始元素定义了 0x40 个属性，IE 就会精确地为新数组分配 0x400 个字节。这些属性接下里使用*"CAttrValue::Copy()"*进行拷贝：

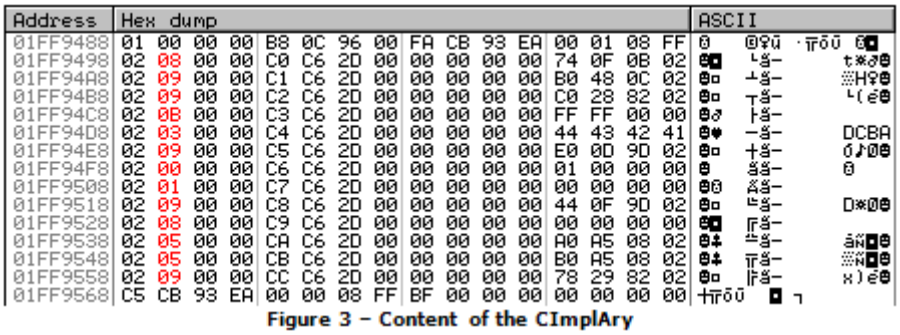
```
.text:3D06A3D9 mov byte ptr [esi+1], 0
.text:3D06A3DD call CAttrValue::Copy
```

现在我们设想下列脚本，包含创建一个新的“Select”元素并关联不同的属性和克隆代码：

```
var test = document.createElement("select")
test.obj0 = "AAAAAAAAAAAAAAAAAAAA"
test. obj1 = this
[...]
test.obj8 = alert
[...]
test.obj12 = new Date()
```

```
var cl0ne = test.cloneNode(true)
```

当克隆后，*CImplAry 数组如下所示：*

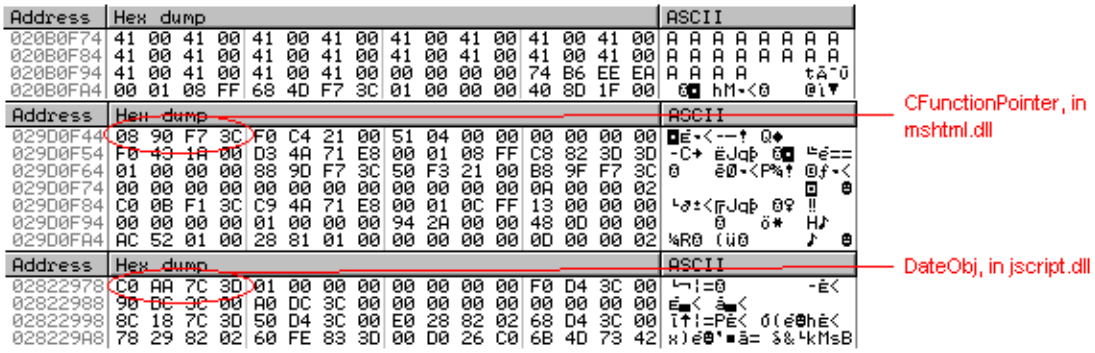


Address	Hex dump	ASCII
01FF9488	01 00 00 00 B8 0C 96 00 FA CB 93 EA 00 01 08 FF	
01FF9498	02 08 00 00 C0 C6 2D 00 00 00 00 00 74 0F 0B 02	
01FF94A8	02 09 00 00 C1 C6 2D 00 00 00 00 00 B0 48 0C 02	
01FF94B8	02 09 00 00 C2 C6 2D 00 00 00 00 00 C0 28 82 02	
01FF94C8	02 0B 00 00 C3 C6 2D 00 00 00 00 00 FF FF 00 00	
01FF94D8	02 03 00 00 C4 C6 2D 00 00 00 00 00 44 43 42 41	DCBA
01FF94E8	02 09 00 00 C5 C6 2D 00 00 00 00 00 E0 0D 9D 02	
01FF94F8	02 00 00 00 C6 C6 2D 00 00 00 00 00 01 00 00 00	
01FF9508	02 01 00 00 C7 C6 2D 00 00 00 00 00 00 00 00 00	
01FF9518	02 09 00 00 C8 C6 2D 00 00 00 00 00 44 0F 9D 02	D
01FF9528	02 08 00 00 C9 C6 2D 00 00 00 00 00 00 00 00 00	
01FF9538	02 05 00 00 CA C6 2D 00 00 00 00 00 A0 A5 08 02	
01FF9548	02 05 00 00 CB C6 2D 00 00 00 00 00 B0 A5 08 02	
01FF9558	02 09 00 00 CC C6 2D 00 00 00 00 00 78 29 82 02	x)
01FF9568	C5 CB 93 EA 00 00 08 FF BF 00 00 00 00 00 00 00	

Figure 3 – Content of the CImplAry

实际上一个属性定义包含 3 个部分，在内存中使用 0x10 个字节。注意观察图 3 中的标注红色的字节。它们表示在 MSDN 中定义的不同类型：0x08 表示一个字符串，0x09 表示一个对象，0x03 为一个整型等等。

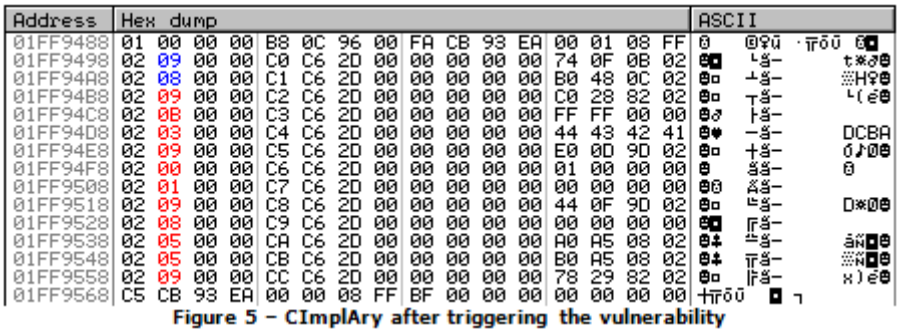
下图为 obj0, obj8 和 obj12 在内存中的表示：



Address	Hex dump	ASCII
020B0F74	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A A A A A A A A
020B0F84	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A A A A A A A A
020B0F94	41 00 41 00 41 00 41 00 00 00 00 00 74 B6 EE EA	A A A A tA^
020B0FA4	00 01 08 FF 68 4D F7 3C 01 00 00 00 40 8D 1F 00	hM-< @
Address	**Hex dump**	**ASCII**
029D0F44	08 90 F7 3C F0 C4 21 00 51 04 00 00 00 00 00 00	
029D0F54	F0 43 1A 00 D3 4A 71 E8 00 01 08 FF C8 82 3D 3D	
029D0F64	01 00 00 00 88 9D F7 3C 50 F3 21 00 B8 9F F7 3C	
029D0F74	00 00 00 00 00 00 00 00 00 00 00 00 0A 00 00 02	
029D0F84	C0 0B F1 3C C9 4A 71 E8 00 01 0C FF 13 00 00 00	
029D0F94	00 00 00 00 01 00 00 00 94 2A 00 00 48 0D 00 00	
029D0FA4	AC 52 01 00 28 81 01 00 00 00 00 00 0D 00 00 02	R(
Address	**Hex dump**	**ASCII**
02822978	C0 AA 7C 3D 01 00 00 00 00 00 00 00 F0 D4 3C 00	
02822988	90 DC 3C 00 A0 DC 3C 00 00 00 00 00 00 00 00 00	
02822998	8C 18 7C 3D 50 D4 3C 00 E0 28 82 02 68 D4 3C 00	P(h
028229A8	78 29 82 02 60 FE 83 3D 00 D0 26 C0 6B 4D 73 42	x) `= &kMsB

Figure 4 – Representations of obj0, obj8, and obj12 in the CImplAry

因为程序确切地依赖上述不同的类型来判断属性的类型（即通过不同的字节如 0x3,0x8 或 0x9 来确定属性的类型），它就有可能使用一个堆溢出来破坏这个数组且强制浏览器混淆类型。图 5 表示增加和减少两个字节之后的数组：



Address	Hex dump	ASCII
01FF9488	01 00 00 00 B8 0C 96 00 FA CB 93 EA 00 01 08 FF	
01FF9498	02 09 00 00 C0 C6 2D 00 00 00 00 00 74 0F 0B 02	
01FF94A8	02 08 00 00 C1 C6 2D 00 00 00 00 00 B0 48 0C 02	
01FF94B8	02 09 00 00 C2 C6 2D 00 00 00 00 00 C0 28 82 02	
01FF94C8	02 0B 00 00 C3 C6 2D 00 00 00 00 00 FF FF 00 00	
01FF94D8	02 03 00 00 C4 C6 2D 00 00 00 00 00 44 43 42 41	DCBA
01FF94E8	02 09 00 00 C5 C6 2D 00 00 00 00 00 E0 0D 9D 02	
01FF94F8	02 00 00 00 C6 C6 2D 00 00 00 00 00 01 00 00 00	
01FF9508	02 01 00 00 C7 C6 2D 00 00 00 00 00 00 00 00 00	
01FF9518	02 09 00 00 C8 C6 2D 00 00 00 00 00 44 0F 9D 02	D
01FF9528	02 08 00 00 C9 C6 2D 00 00 00 00 00 00 00 00 00	
01FF9538	02 05 00 00 CA C6 2D 00 00 00 00 00 A0 A5 08 02	
01FF9548	02 05 00 00 CB C6 2D 00 00 00 00 00 B0 A5 08 02	
01FF9558	02 09 00 00 CC C6 2D 00 00 00 00 00 78 29 82 02	x)
01FF9568	C5 CB 93 EA 00 00 08 FF BF 00 00 00 00 00 00 00	

Figure 5 – CImplAry after triggering the vulnerability

图 5 中，obj0(对象 0)为一个对象而 obj1 为一个字符串。通过使用 JavaScript 脚本泄露对象的 vTalbe(虚函数表)来绕过 ASLR/DEP 就成为可能。如下所示：

Figure 6 – vTable disclosure of the *this* object

因此这个恶意的字符串在函数"*CAttrValue::GetIntoVariant()*"使用并触发漏洞并到一个 CALL 指令执行任意代码而不用管 ASLR/DEP:

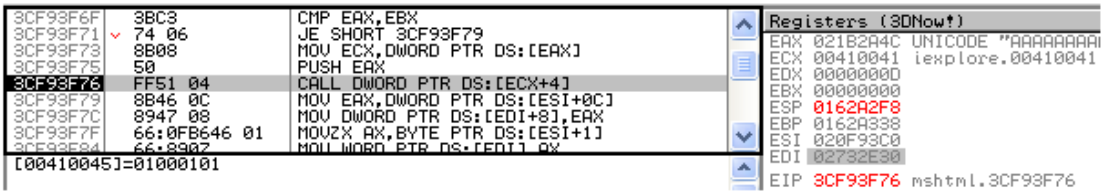


Figure 7 – Crash after using the corrupt string w0

使用这种方法需要修改 2 个字节。如果你只修改了一个字节，请一定通知我！需要注意的是这种方法通用于所有 Internet Explorer 版本，包括 9/8/7 甚至 IE6，使用 Sizes > 0x230 进行堆分配(heapAlloc)。

就像博客中演示的一样，这个漏洞确实是危急程度，因此我们强烈建议尽快打上补丁。