《漏洞利用及渗透测试基础》实验报告

姓名：管昀玫 学号：2013750 班级：计科一班

**实验名称：**

API函数自搜索实验

**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操作系统里验证是否成功。

**实验过程：**

1. 思路分析：

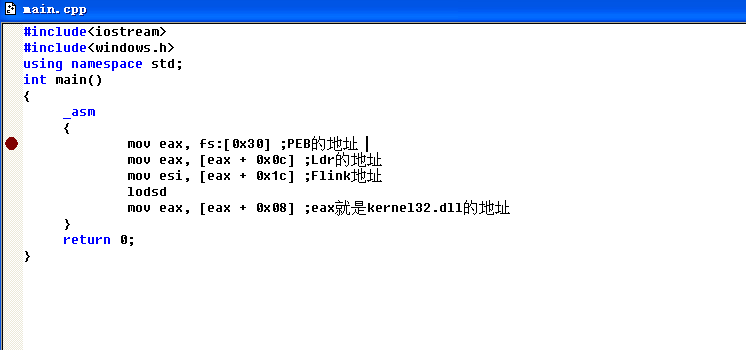
由于系统装载机制的不同，动态链接库装载的baseaddress是不同的，所以之前编写的直接使用地址进行调用在不同的系统上是不太可行的，所以需要自搜索。在实际中为了编写通用shellcode，shellcode自身就必须具备动态的自动搜索所需API函数地址的能力，即API函数自搜索技术。

需要调出、关闭消息框->需要调用user32.dll中的MassageBoxA和kernel32.dll中的ExitProcess。又由于不是所有进程都会调用user32.dll，但是都会调用kernel32.dll->需要调用kernel32.dll中的LoadLibrary来装载user32.dll。综上，实验的步骤为：

* + 第一步：定位kernel32.dll。
  + 第二步：定位kernel32.dll的导出表。
  + 第三步：搜索定位LoadLibrary等目标函数。
  + 第四步：基于找到的函数地址，完成Shellcode的编写。

1. 实验过程
   1. 定位kernel32.dll

根据书本编写如下代码



简单的代码中包含以下步骤：

[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。

可以看出通过上述代码能够得到kernel32.dll的地址，并且把kernel32.dll的地址存放到了eax中

除此之外，还需要定位kernel32.dll导出表及其导出函数名列表：

[7] 从kernel32.dll加载基址算起，偏移0x3c的地方就是其PE头的指针。

[8] PE头偏移0x78的地方存放着指向函数导出表的指针。

[9] 获得导出函数偏移地址（RVA）列表、导出函数名列表：

• 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。

• 导出表偏移0x20处的指针指向存储导出函数函数名的列表。

具体代码如下：

mov ebp, eax //将kernel32.dll基地址赋值给ebp

mov eax,[ebp+0x3C] //dll的PE头的指针（相对地址）

mov ecx,[ebp+eax+0x78] //导出表的指针（相对地址）

add ecx,ebp //ecx=0x78C00000+0x262c 得到导出表的内存地址

mov ebx,[ecx+0x20] //导出函数名列表指针

add ebx,ebp //导出函数名列表指针的基地址

* 1. 实现功能

在得到了kernel32.dll的地址后，需要找到kernel.dll的导出表位置（从kernel32.dll的装载地址算起，偏移0x3c + 0x78的位置就是导出表的指针），之后遍历导出表，依次对比寻找需要的API函数（通过对比函数名称，寻找到他装载的指针）

* + 1. 编写Hash函数

通常想法是通过对比函数名的字符串去确定函数，根据书上的说法，ShellCode应当是尽量的简短，所以采用计算函数名的hash值，通过对比数值的方式也是可以的，同时也能起到一个压缩的效果

得到函数Hash的函数如下：

DWORD GetHash(char \*fun\_name)

{

DWORD digest=0;

while(\*fun\_name)

{

digest=((digest<<25)|(digest>>7)); //循环右移7位

/\*

movsx eax,byte ptr[esi]

cmp al,ah

jz compare\_hash

ror edx, 7 ; ((循环))右移,不是单纯的 >>7

add edx,eax

inc esi

jmp hash\_loop

\*/

digest+= \*fun\_name ; //累加

fun\_name++;

}

return digest;

}

通过调用上述的函数，得到”MassageBoxA”的Hash值为：506989162（0x1E38 0A6A）

同理得到”ExitProcess”：1339132259（0x4FD1 8963）

“LoadLibraryA”：210859058（0x0C917432）

* + 1. 定位API的地址并进行相应的调用

参照课本，完整代码如下：

#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

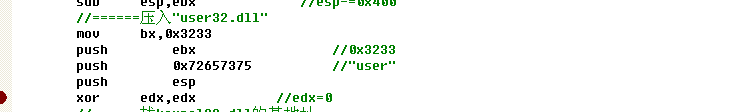
}

return 0;

}



此处三个push的都是哈希值，且后面做的比较都是哈希的比较，而非字符串的比较。而ESI用来指向哈希值。



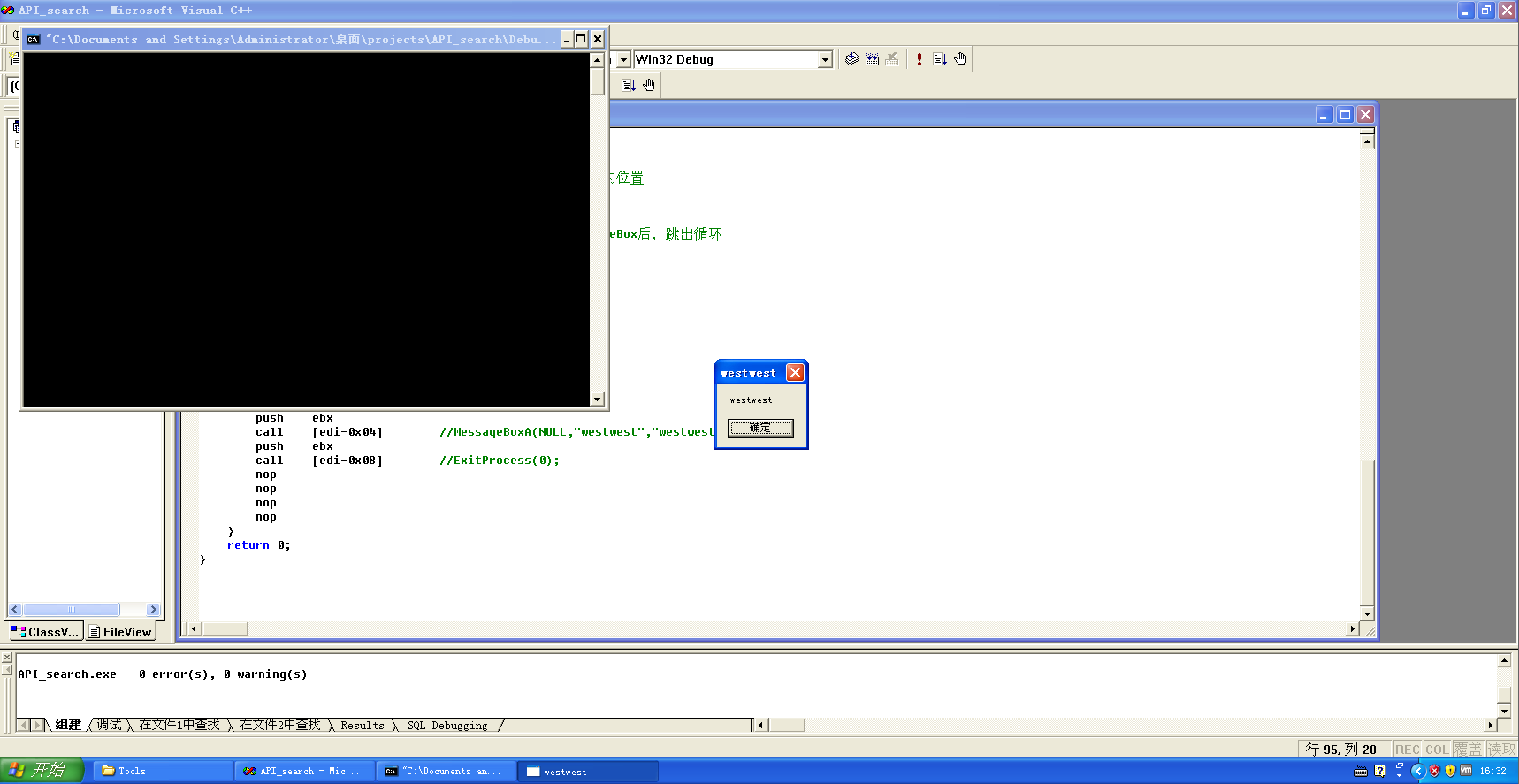
此处用于push user32，而ESP中储存的是user32的字符串的地址。整体来说是user32的参数值入栈。



此处，jne语句会比较是否找到MessageBoxA函数的地址，如果不是，则会跳转到Find\_Functions函数。多次跳转之后，最后将能找到函数地址，跳出循环。

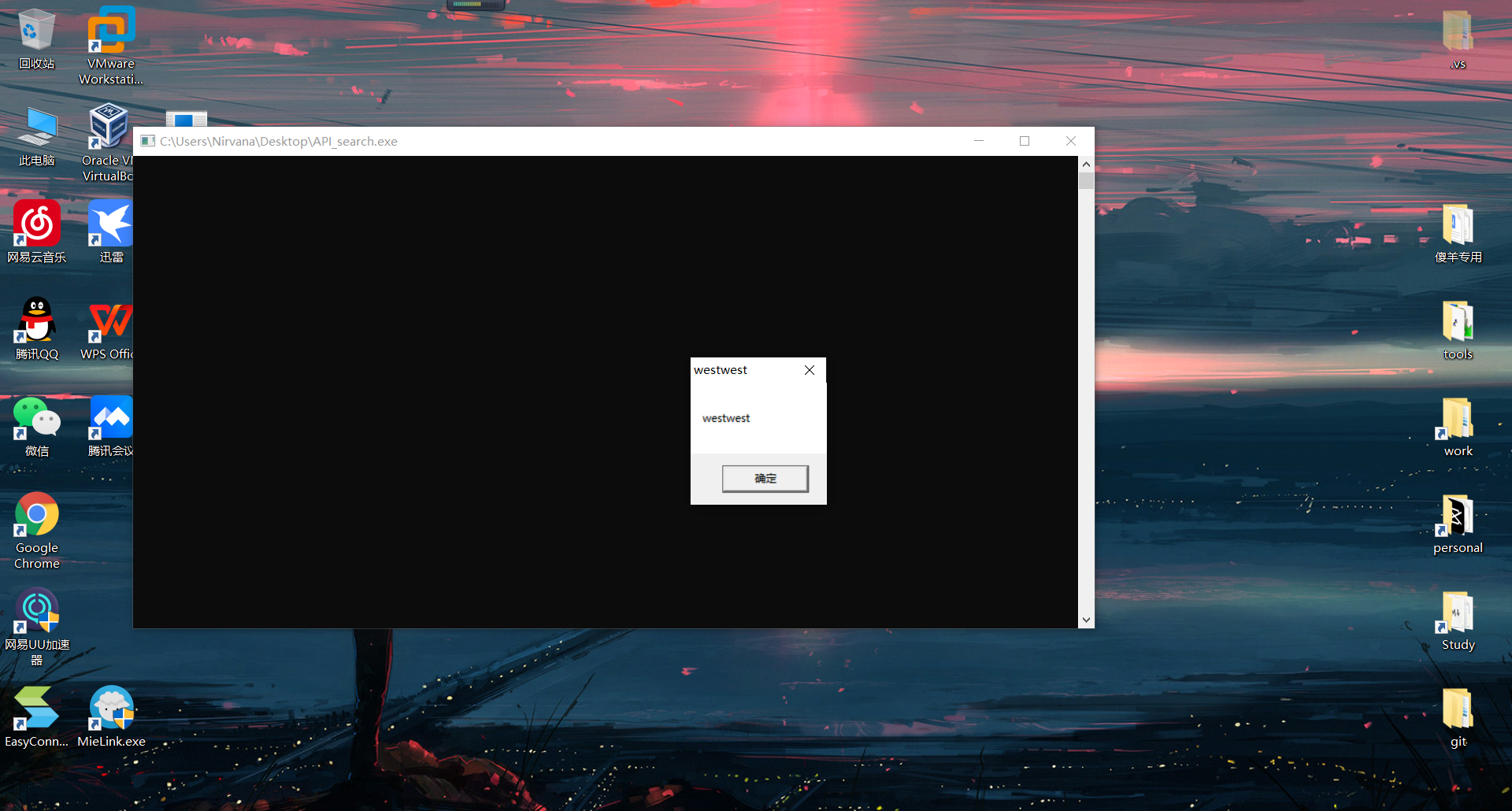
之后的逻辑就与注释相同。

在xp系统上运行得到的结果:



得到想要的结果westwest

由于本次实验利用的是自搜索技术，地址是动态计算的，因此将程序放在win10上运行也没问题。将exe放在win10上运行的结果：



得到结果

综上：实验复现成功

**心得体会：**

通过本次的实验，看汇编代码比以前更加顺畅一些，对动态链接库的调用过程也更加了解了。对于软件安全这门课来说，学会、了解汇编代码还是很重要的，今后也要加强这方面的练习。

API自搜索的方法主要是利用的PE文件头等的相关知识，实现在程序加载时对相对位置的查找、定位，更加适用于现在复杂的应用环境中，同时也能够相应的绕开一些简单的防护机制，达到自己的目的。