

软件安全实验6

姓名：何叶 学号：2313487 班级：范玲玲班

实验名称

API函数自搜索

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。

实验原理

前面所介绍的shellcode编写，都采用硬编码的方式来调用相应API函数。首先，获取所要使用函数的地址，然后将该地址写入shellcode，从而实现调用。如果系统的版本变了，很多函数的地址往往都会发生变化，那么调用就会失败。

在实际中为了编写通用型shellcode，shellcode自身必须具备动态的自搜索所需API函数地址的能力，即API函数自搜索技术。

实验步骤

一、通用型shellcode的编写逻辑

首先总结将要调用到的函数。

(1) MessageBoxA位于user32.dll中，用于弹出消息框。

(2) ExitProcess位于kernel32.dll中，用于正常退出程序。所有的Win32程序都会自动加载ntdll.dll以及kernel32.dll这两个最基础的动态链接库。

(3) LoadLibraryA位于kernel32.dll中，并不是所有的程序都会装载user32.dll，所以在调用MessageBoxA之前，应该先使用LoadLibrary("user32.dll")装载user32.dll。

然后介绍通用型shellcode编写的步骤。

(1) 定位kernel32.dll。

```
//=====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
```

(2) 定位kernel32.dll的导出表。

```
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
```

(3) 搜索定位LoadLibrary等目标函数。

```
#include <stdio.h>
#include <windows.h>
DWORD GetHashCode(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;
}

main()
{
    DWORD hash;
    hash= GetHashCode("MessageBoxA");
    printf("%#x\n",hash);
}
```

(4) 基于找到的函数地址，完成shellcode的编写。

```
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;
}

```

难点在于第一步到第三步，即如何实现API函数自搜索。

二、API函数自搜索技术

1. 定位kernel32.dll

```

int main()
{
    __asm
    {
        mov eax, fs:[0x30]      ;PEB的地址
        mov eax, [eax+0x0c]     ;PEB_LDR_DATA结构体的地址
        mov esi, [eax+0x1c]     ;指针InitializationOrderModuleList
        lodsd
        mov eax, [eax+0x08]     ;eax是kernel32.dll的地址
    }
    return 0;
}

```

如果想要在Win32平台下定位kernel32.dll中的API地址，可以使用如下方法。

- (1) 首先通过段寄存器FS在内存中找到当前的线程环境块。
- (2) 线程环境块中偏移地址为0x30的地方存放着指向进程环境块的指针。
- (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结构体中偏移地址为0x1c的地方存放着指向模块初始化链表的头指针InitializationOrderModuleList。
- (5) 模块初始化链表InitializationOrderModuleList中按顺序存放着PE头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在内存中的加载基址。

2. 定位kernel32.dll的导出表

```

mov     ebp, eax      ;将kernel32.dll基址赋值给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+0x262c2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址

```

找到kernel32.dll，由于它也属于PE文件，因此可以根据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kernel32.dll导出表及其导出函数名列表的步骤如下。

- (1) 从kernel32.dll加载基址算起，偏移地址为0x3c的地方就是其PE头的指针。
- (2) PE头中偏移地址为0x78的地方存放着指向函数导出表的指针。
- (3) 获得导出函数地址为偏移地址(RVA)列表，导出函数名列表：
 - ① 导出表中偏移地址为0x1c处的指针指向存储导出函数偏移地址(RVA)的列表。
 - ② 导出表中偏移地址为0x20处的指针指向存储导出函数名的列表。

定位kernel32.dll导出表及其导出函数名列表的代码如下：

3. 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和名字按照顺序存放在上述两个列表中，我们可以在函数名的列表中定位到所需的函数是第几个，然后在函数偏移地址的列表中找到对应的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

三、完整API函数自搜索代码

1.hash运算

为了让shellcode更加通用，能被大多数缓冲区容纳，所以总是希望shellcode尽可能短。因此，一般情况下并不会用MessageBoxA等这么长的字符串进行直接比较。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。

2.完整API函数自搜索代码

```
#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字节应该也是为了兼容性
        //=====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=user132.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

```

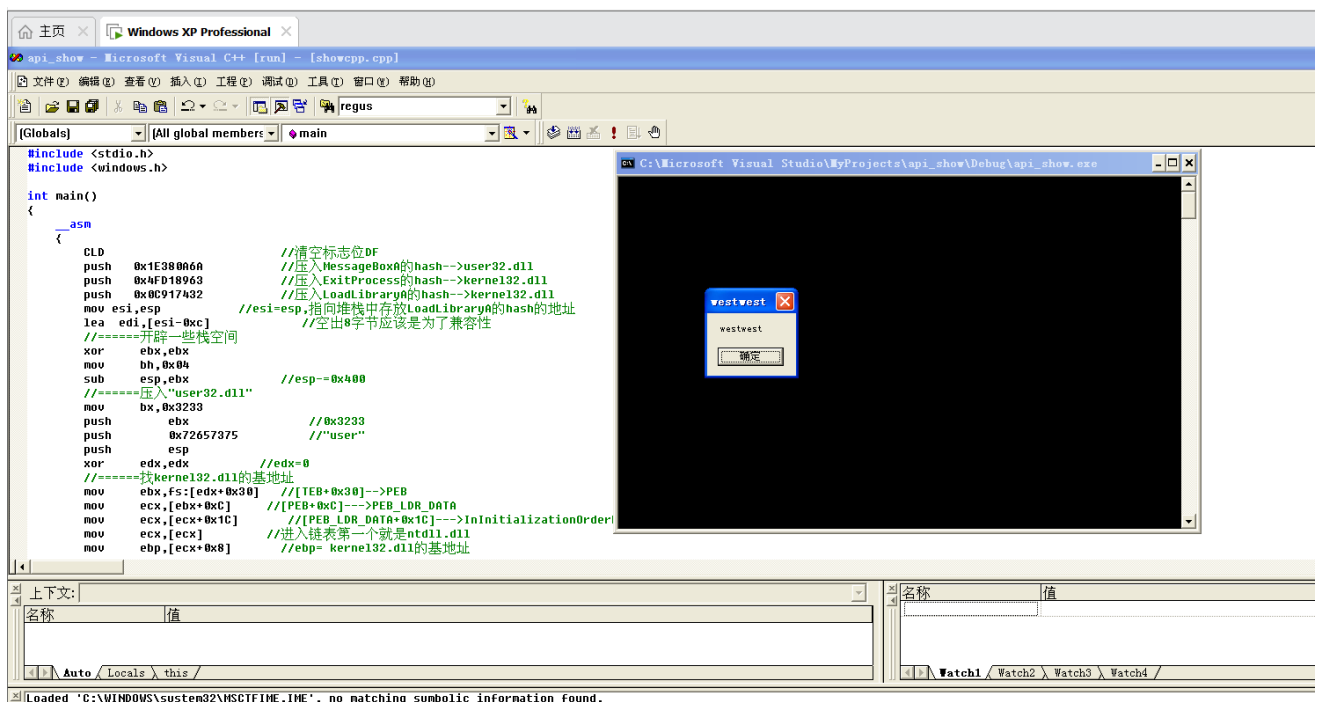
```

call    [edi-0x04]           //MessageBoxA(NULL,"westwest","westwest",NULL)
push    ebx
call    [edi-0x08]           //ExitProcess(0);
nop
nop
nop
nop
}
return 0;
}

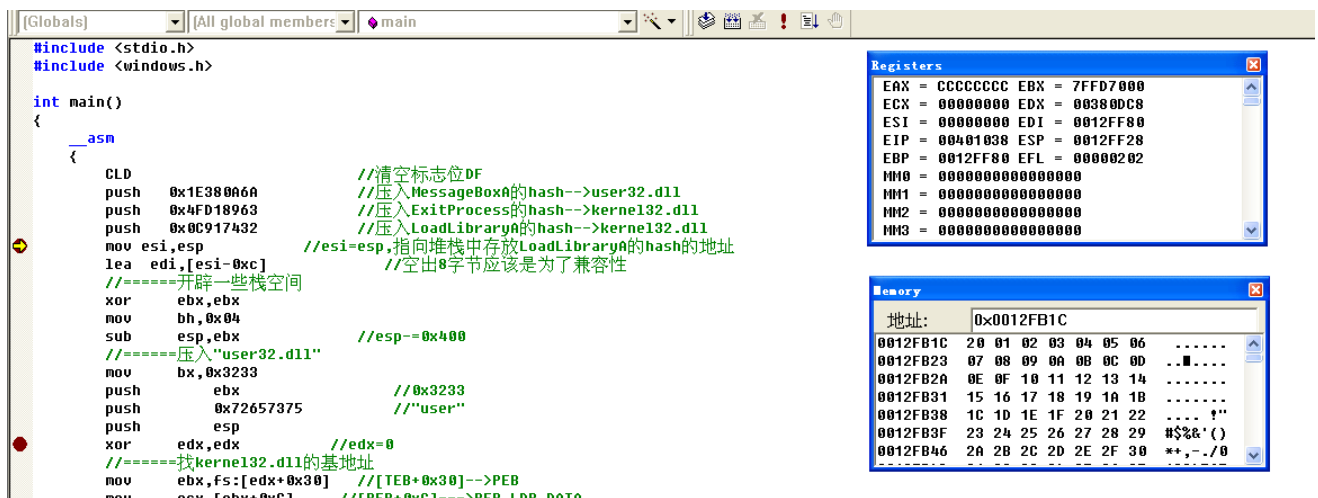
```

四、通过反汇编观察代码运行原理

1.运行成功



2.3个push入之前求出的hash值



3.esi用于找到hash值

```
#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字节应该也是为了兼容性
        //=====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
    }
```

Registers

EAX	=	CCCCCCCC	EBX	=	7FFD7000
ECX	=	00000000	EDX	=	00380DC8
ESI	=	0012FF28	EDI	=	0012FF80
EIP	=	0040103A	ESP	=	0012FF28
EBP	=	0012FF80	EFL	=	00000202
MM0	=	00000000	MM1	=	00000000
MM2	=	00000000	MM3	=	00000000

Memory

地址:	0x0012FB1C
0012FB1C	20 01 02 03 04 05 06
0012FB23	07 08 09 0A 0B 0C 0D
0012FB2A	0E 0F 10 11 12 13 14

4.抬高esp

```
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字节应该也是为了兼容性
        //=====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
    }
```

Registers

EAX	=	CCCCCCCC	EBX	=	00000400
ECX	=	00000000	EDX	=	00380DC8
ESI	=	0012FF28	EDI	=	0012FF1C
EIP	=	00401043	ESP	=	0012FB28
EBP	=	0012FF80	EFL	=	00000206
MM0	=	00000000	MM1	=	00000000
MM2	=	00000000	MM3	=	00000000

Memory

地址:	0x0012FB1C
0012FB1C	20 01 02 03 04 05 06
0012FB23	07 08 09 0A 0B 0C 0D
0012FB2A	0E 0F 10 11 12 13 14
0012FB31	15 16 17 18 19 1A 1B
0012FB38	1C 1D 1E 1F 20 21 22
0012FB3F	23 24 25 26 27 28 29
0012FB46	2A 2B 2C 2D 2E 2F 30

5.esp储存user32

```
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字节应该也是为了兼容性
        //=====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
    }
```

Registers

EAX	=	CCCCCCCC	EBX	=	00003233
ECX	=	00000000	EDX	=	00380DC8
ESI	=	0012FF28	EDI	=	0012FF1C
EIP	=	0040104D	ESP	=	0012FB20
EBP	=	0012FF80	EFL	=	00000206
MM0	=	00000000	MM1	=	00000000
MM2	=	00000000	MM3	=	00000000

Memory

地址:	0x0012FB20
0012FB20	75 73 65 72 33 32 00 user32.
0012FB27	00 0C 0D 0E 0F 10 11
0012FB2E	12 13 14 15 16 17 18
0012FB35	19 1A 1B 1C 1D 1E 1F
0012FB3C	20 21 22 23 24 25 26
0012FB43	27 28 29 2A 2B 2C 2D
0012FB4A	2E 2F 30 31 32 33 34

6.ebp储存 kernel32.dll的基地址

```
push ecx
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++, 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=user32.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
```

Registers

EAX	=	CCCCCCCC	EBX	=	7FFD7000
ECX	=	00242020	EDX	=	00000000
ESI	=	0012FF28	EDI	=	0012FF1C
EIP	=	0040105F	ESP	=	0012FB1C
EBP	=	7C800000	EFL	=	00000246
MM0	=	00000000	MM1	=	00000000
MM2	=	00000000	MM3	=	00000000

Memory

地址:	0x7C800000
7C800000	4D 5A 90 00 03 00 00 M2.....
7C800007	00 04 00 00 00 FF FF
7C80000E	00 00 88 00 00 00 00
7C800015	00 00 00 40 00 00 00
7C80001C	00 00 00 00 00 00 00
7C800023	00 00 00 00 00 00 00
7C80002A	00 00 00 00 00 00 00

7.比较hash

```
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=user132.dll的基地址,eax=MessageBoxA的hash <-- |
```

8.找到函数名, 计算hash值

```
//-----找到函数名
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
```

9.循环多次找到值

```
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
```

Registers	
EAX =	00000000
ECX =	7C80262C
EDX =	74E0245E
ESI =	7C8048B2
EDI =	00000001
EIP =	00401093
ESP =	0012FAFC
EBP =	7C800000
EFL =	00000246
MM0 =	00000000
MM1 =	00000000
MM2 =	00000000
MM3 =	00000000

Memory	
地址:	0x7C800000
7C800000	4D 5A 90 00 03 00 00 M2.....
7C800007	00 04 00 00 00 FF FF
7C80000E	00 00 B8 00 00 00 00
7C800015	00 00 00 40 00 00 00 ...@...
7C80001C	00 00 00 00 00 00 00
7C800023	00 00 00 00 00 00 00
7C80002A	00 00 00 00 00 00 00

10.算出基地址

```
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            //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
```

Registers	
EAX =	0000006F
ECX =	7C80262C
EDX =	C9920843
ESI =	7C8048C0
EDI =	00000003
EIP =	004010A8
ESP =	0012FAFC
EBP =	7C800000
EFL =	00000206
MM0 =	00000000
MM1 =	00000000
MM2 =	00000000
MM3 =	00000000

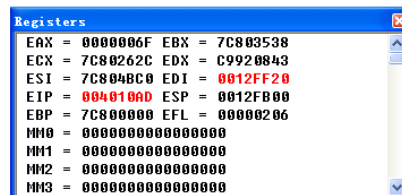
Memory	
地址:	0x7C800000
7C800000	4D 5A 90 00 03 00 00 M2.....
7C800007	00 04 00 00 00 FF FF
7C80000E	00 00 B8 00 00 00 00
7C800015	00 00 00 40 00 00 00 ...@...
7C80001C	00 00 00 00 00 00 00
7C800023	00 00 00 00 00 00 00

11.存到edi

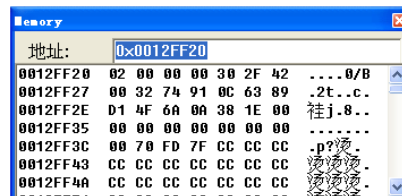
```
mov     ebx,[ecx+0x1C] //地址表的相对偏移量
add     ebx,ebp        //地址表的基地址
add     ebp,[ebx+4*edi] //函数的基地址
xchg    eax,ebp        //eax<=>ebp 交换

pop     edi            //把找到的函数保存到edi的位置
stosd   edi
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 "westwest"
mov     eax,esp
push    ebx
push    eax
push    ebx
call    [edi-0x04]      //MessageBox(NULL,"westwest","westwest",NULL)
push    ebx
```



Register	Value
EAX	0000006F
EBX	7C803538
ECX	7C80262C
EDX	C9920843
ESI	7C8048C0
EDI	0012FF20
EIP	004010AD
ESP	0012FB00
EBP	7C800000
EFL	00000206
MM0	00000000
MM1	00000000
MM2	00000000
MM3	00000000



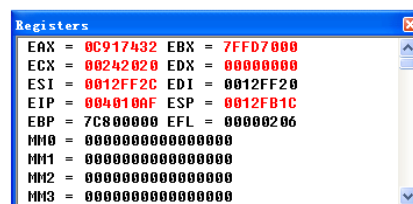
Address	Value
0012FF20	02 00 00 00 30 2F 42 ... 0/B
0012FF27	00 32 74 91 0C 63 89 ... 2t.c.
0012FF2E	D1 4F 6A 0A 38 1E 00 ... 挂j.8..
0012FF35	00 00 00 00 00 00 00
0012FF3C	00 70 FD 7F CC CC CC ... p?汤
0012FF43	CC CC CC CC CC CC CC ... 汤汤汤
0012FF4A	CC CC CC CC CC CC CC ... 汤汤汤

12.比较当前找到的是不是

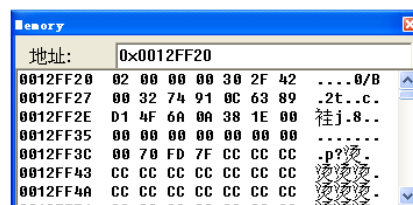
```
mov     ebx,[ecx+0x1C] //地址表的相对偏移量
add     ebx,ebp        //地址表的基地址
add     ebp,[ebx+4*edi] //函数的基地址
xchg    eax,ebp        //eax<=>ebp 交换

pop     edi            //把找到的函数保存到edi的位置
stosd   edi
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 "westwest"
mov     eax,esp
push    ebx
push    eax
push    ebx
call    [edi-0x04]      //MessageBox(NULL,"westwest","westwest",NULL)
push    ebx
```



Register	Value
EAX	0C917432
EBX	7FFD7000
ECX	00242020
EDX	00000000
ESI	0012FF2C
EDI	0012FF20
EIP	004010AF
ESP	0012FB1C
EBP	7C800000
EFL	00000206
MM0	00000000
MM1	00000000
MM2	00000000
MM3	00000000



Address	Value
0012FF20	02 00 00 00 30 2F 42 ... 0/B
0012FF27	00 32 74 91 0C 63 89 ... 2t.c.
0012FF2E	D1 4F 6A 0A 38 1E 00 ... 挂j.8..
0012FF35	00 00 00 00 00 00 00
0012FF3C	00 70 FD 7F CC CC CC ... p?汤
0012FF43	CC CC CC CC CC CC CC ... 汤汤汤
0012FF4A	CC CC CC CC CC CC CC ... 汤汤汤

13.如果不是, 跳转

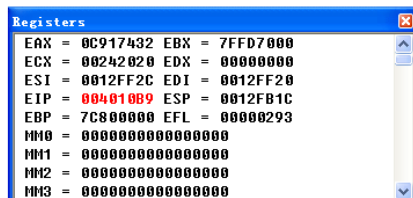
```
//=====是否找到了自己所需全部的函数
find_lib_functions:
lodsd   //即move eax,[esi], esi+=4, 第一次取LoadLibraryA的hash
cmp     eax,0x1E380A6A //与MessageBoxA的hash比较
jne     find_functions //如果没有找到MessageBoxA函数, 继续找
xchg    eax,ebp        //-----> |
call    [edi-0x08]      //LoadLibraryA("user32") |
xchg    eax,ebp        //ebp=user32.dll的基地址,eax=MessageBoxA的hash <-- |

//=====导出函数名列表指针
```

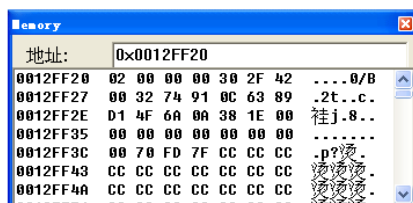
14.自己shellcode编写

```
pop     eax,0x1e380a6a //找到最后一个函数MessageBox后,跳出循环
cmp     eax,0x1e380a6a
jne     find_lib_functions

//=====让他做些自己想做的事
function_call:
xor     ebx,ebx
push    ebx
push    0x74736577      //push "westwest"
push    0x74736577      //push "westwest"
mov     eax,esp
push    ebx
push    eax
push    ebx
call    [edi-0x04]      //MessageBox(NULL,"westwest","westwest",NULL)
push    ebx
call    [edi-0x08]      //ExitProcess(0);
nop
nop
nop
nop
return 0;
```

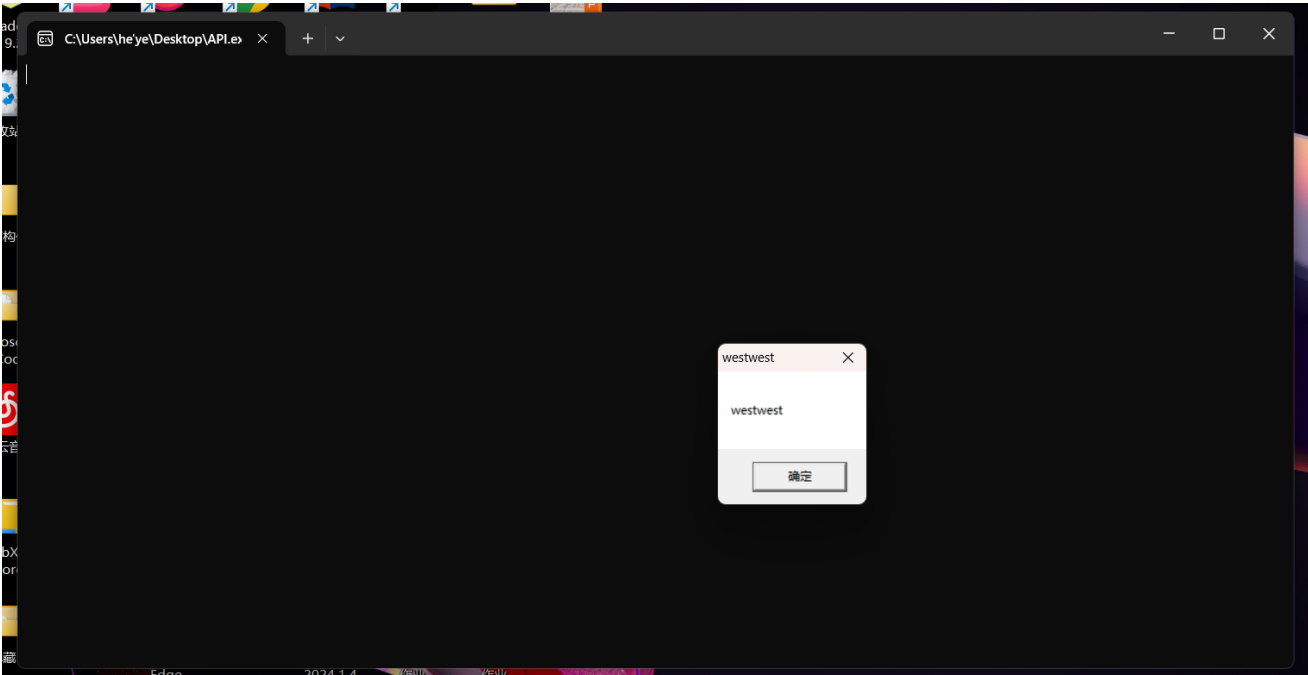
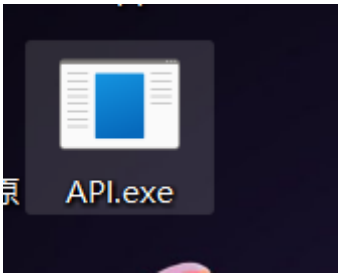


Register	Value
EAX	0C917432
EBX	7FFD7000
ECX	00242020
EDX	00000000
ESI	0012FF2C
EDI	0012FF20
EIP	004010B9
ESP	0012FB1C
EBP	7C800000
EFL	00000293
MM0	00000000
MM1	00000000
MM2	00000000
MM3	00000000



Address	Value
0012FF20	02 00 00 00 30 2F 42 ... 0/B
0012FF27	00 32 74 91 0C 63 89 ... 2t.c.
0012FF2E	D1 4F 6A 0A 38 1E 00 ... 挂j.8..
0012FF35	00 00 00 00 00 00 00
0012FF3C	00 70 FD 7F CC CC CC ... p?汤
0012FF43	CC CC CC CC CC CC CC ... 汤汤汤
0012FF4A	CC CC CC CC CC CC CC ... 汤汤汤

五、复制到windows11系统运行



六、心得体会

这次API函数自搜索实验让我对动态调用API有了更深刻的理解，意识到在shellcode中动态搜索API地址对于编写通用代码的重要性。通过实际操作，我加深了对PE文件结构的认识，尤其是如何定位DLL基地址和导出表。编写和调试汇编代码的过程也提升了我的编程技能，让我对底层系统有了更深入的理解。此外，实验还增强了我的安全意识，让我认识到自搜索技术在安全领域的应用及其潜在的安全威胁。整个实验过程不仅提高了我的技术能力，也激发了我对软件安全领域的兴趣。