

软件安全实验报告

姓名：辛浩然 学号：2112514 班级：信息安全、法学

1 实验名称

API 函数自搜索

2 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。

3 实验过程

3.1 API 函数自搜索步骤

1. 定位 kernel32.dll。
2. 定位 kernel32.dll 的导出表。
3. 搜索定位 LoadLibrary 等目标函数。
4. 基于找到的函数地址，完成 Shellcode 的编写。

3.2 API 函数自搜索代码及实现

```
1 #include <stdio.h>
2 #include <windows.h>
3 int main()
4 {
5     __asm
6     {
7         CLD
8         push    0x1E380A6A
9         push    0x4FD18963
10        push    0x0C917432
11        mov esi,esp
12        lea edi,[esi-0xc]
13        //=====开辟一些栈空间
14        xor     ebx,ebx
15        mov     bh,0x04
```

```

16      sub     esp,ebx
17      //=====压入"user32.dll"
18      mov     bx,0x3233
19      push    ebx
20      push    0x72657375
21      push    esp
22      xor     edx,edx
23      //=====找kernel32.dll的基地址
24      mov     ebx,fs:[edx+0x30]
25      mov     ecx,[ebx+0xC]
26      mov     ecx,[ecx+0x1C]
27      mov     ecx,[ecx]
28      mov     ebp,[ecx+0x8]
29      //=====是否找到了自己所需全部的函数
30      find_lib_functions:
31      lodsd
32      cmp     eax,0x1E380A6A
33      jne     find_functions
34      xchg    eax,ebp
35      call    [edi-0x8]
36      xchg    eax,ebp
37      //=====导出函数名列表指针
38      find_functions:
39      pushad
40      mov     eax,[ebp+0x3C]
41      mov     ecx,[ebp+eax+0x78]
42      add     ecx,ebp
43      mov     ebx,[ecx+0x20]
44      add     ebx,ebp
45      xor     edi,edi
46      //=====找下一个函数名
47      next_function_loop:
48      inc     edi
49      mov     esi,[ebx+edi*4]
50      add     esi,ebp
51      cdq
52      //=====函数名的hash运算
53      hash_loop:
54      movsx   eax,byte ptr[esi]
55      cmp     al,ah
56      jz      compare_hash
57      ror     edx,7

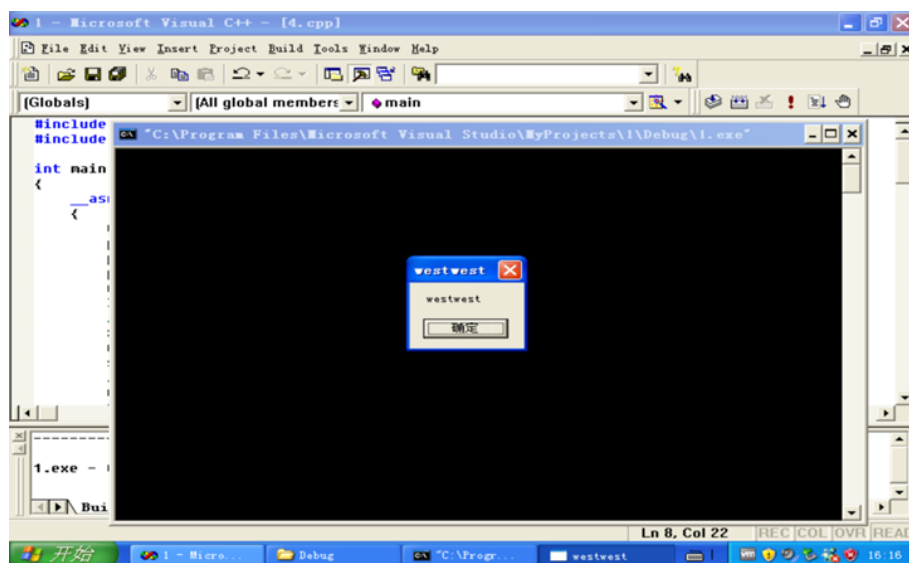
```

```

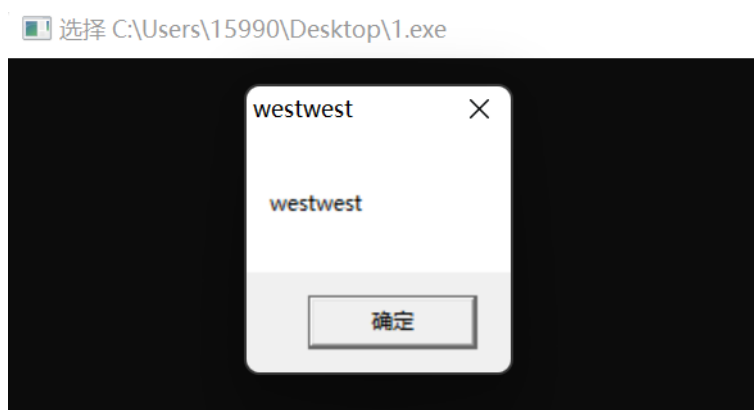
58     add     edx, eax
59     inc     esi
60     jmp     hash_loop
61     //===== 比较找到的当前函数的hash是否是自己想找的
62     compare_hash:
63     cmp     edx, [esp+0x1C]
64     jnz     next_function_loop
65     mov     ebx, [ecx+0x24]
66     add     ebx, ebp
67     mov     di, [ebx+2*edi]
68     mov     ebx, [ecx+0x1C]
69     add     ebx, ebp
70     add     ebp, [ebx+4*edi]    //函数的基地
71     xchg    eax, ebp
72     pop     edi
73     stosd
74     push    edi
75     popad
76     cmp     eax, 0x1e380a6a
77     //找到最后一个函数MessageBox后, 跳出循环
78     jne     find_lib_functions
79     //===== 让他做些自己想做的事
80     function_call:
81     xor     ebx, ebx
82     push    ebx
83     push    0x74736577
84     push    0x74736577
85     mov     eax, esp
86     push    ebx
87     push    eax
88     push    eax
89     push    ebx
90     call    [edi-0x04]
91     push    ebx
92     call    [edi-0x08]
93     nop
94     nop
95     nop
96     nop
97 }
98 return 0;
99 }

```

运行该程序：



将生成的 exe 程序，复制到 windows10 操作系统：



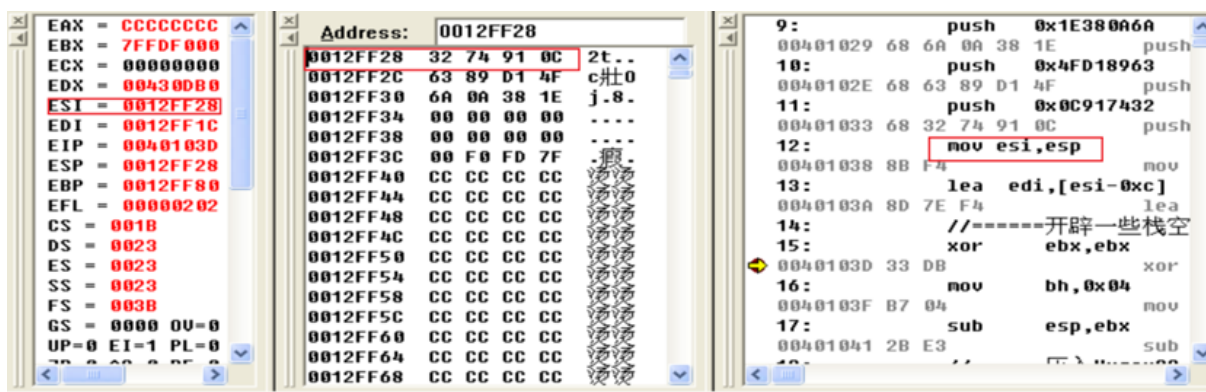
操作成功。

3.3 代码分析

3.3.1

```
1 CLD //清空标志位DF
2 push 0x1E380A6A //压入MessageBoxA的hash-->user32.dll
3 push 0x4FD18963 //压入ExitProcess的hash-->kernel32.dll
4 push 0x0C917432 //压入LoadLibraryA的hash-->kernel32.dll
5 mov esi,esp //esi=esp,指向堆栈中存放LoadLibraryA的hash的地址
6 lea edi,[esi-0xc] //空出8字节应该是为了兼容性
```

程序一开始先清空标志位 DF，然后分别压入 MessageBoxA、ExitProcess、LoadLibraryA 的 hash 的地址。然后将 esp 赋值给 esi，使 esi 指向堆栈中存放 LoadLibraryA 的 hash 的地址。进行调试分析，如图所示，可以发现确实如此。

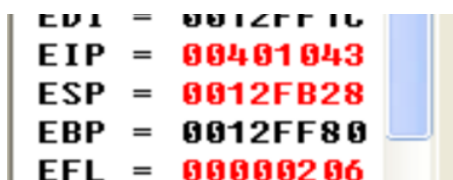


```

1  xor ebx,ebx
2  mov bh,0x04
3  sub esp,ebx //esp-=0x400

```

随后将 esp 抬高 0x400，开辟栈空间。

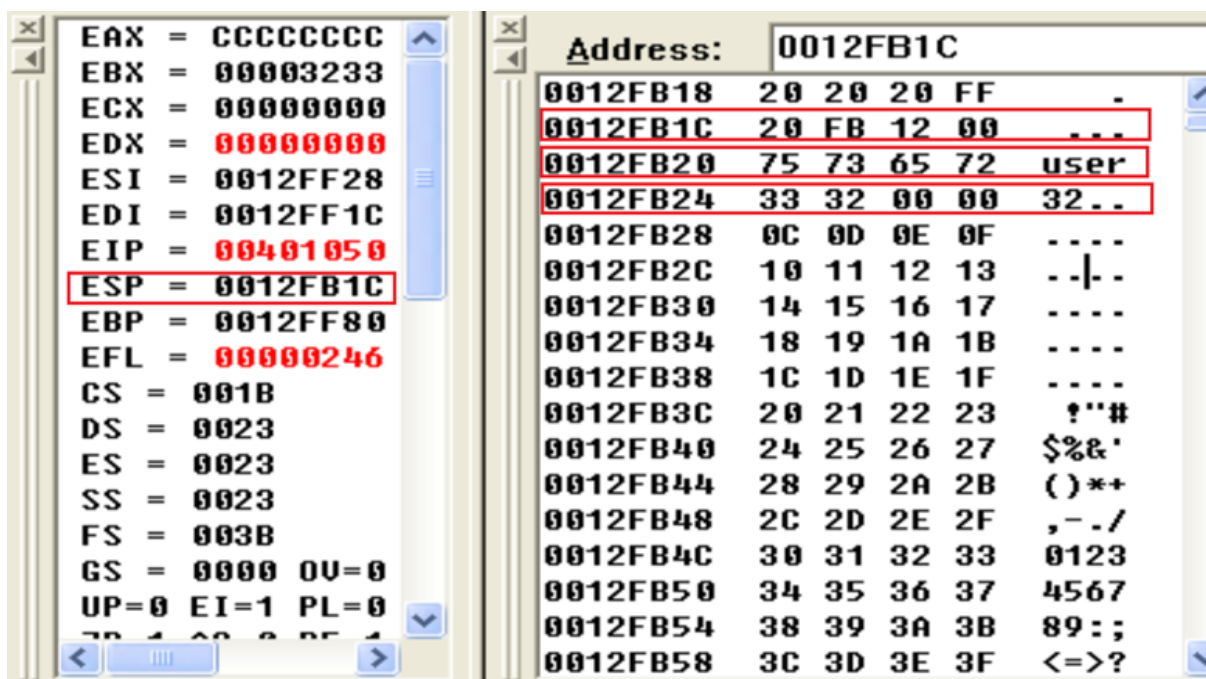


```

1  mov bx,0x3233
2  push ebx //0x3233
3  push 0x72657375 //"user"
4  push esp
5  xor edx,edx //edx=0

```

压入 user32



3.3.2 定位 kernel32.dll

```
1  mov ebx,fs:[edx+0x30] //[TEB+0x30]-->PEB
2  mov ecx,[ebx+0xC] //[PEB+0xC]--->PEB_LDR_DATA
3  mov ecx,[ecx+0x1C]
4  //[PEB_LDR_DATA+0x1C]--->InInitializationOrderModuleList
5  mov ecx,[ecx] //进入链表第一个就是 ntdll.dll
6  mov ebp,[ecx+0x8]//ebp=kernel32.dll的基地址
```

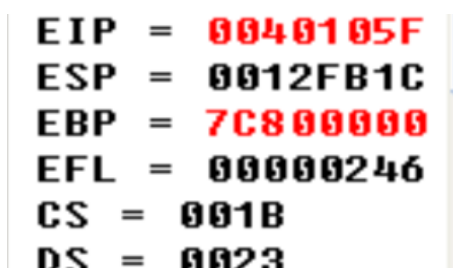
找 kernel32.dll 的基地址:

先通过段选择字 FS 在内存中找到当前的线程环境块 TEB。

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。偏移 0x30 找到指向的 PEB 指针。

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。

进入链表第一个就是 ntdll.dll,再偏移 0x08 就是 kernel32.dll 的地址,存入 ebp 中。如图,ebp 值为 0x7c800000。



EIP = 0040105F
ESP = 0012FB1C
EBP = 7C800000
EFL = 00000246
CS = 001B
DS = 0023

利用 depends 工具查看 kernel32.dll 的基地址确实为 0x7c800000。

Module	Time Stamp	Size	Attributes	Machine	Subsystem	Debug	Base
i. EXE	04/04/23 4:15p	151,634	A	Intel x86	Win32 console	Yes	0x00400000
kernel32.dll	04/14/08 8:00p	1,150,464	A	Intel x86	Win32 console	Yes	0x7C800000
NTDLL.DLL	04/14/08 8:00p	589,312	A	Intel x86	Win32 console	Yes	0x7C920000

3.3.3

```
1  lodsd //即 move eax,[esi], esi+=4, 第一次取 LoadLibraryA 的 hash
2  cmp eax,0x1E380A6A //与 MessageBoxA 的 hash 比较
3  jne find_functions //如果没有找到 MessageBoxA 函数,继续找
```

判断是否找到了自己所需全部的函数:

第一次取 LoadLibraryA 的 hash,与 MessageBoxA 的 hash 比较,如果没有找到 MessageBoxA 函数,利用 find_fuctions 函数继续找。

3.3.4 定位 kernel32.dll 的导出表

```
1  find_functions:
2  pushad //保护寄存器
3  mov eax,[ebp+0x3C] //dll 的 PE 头
```

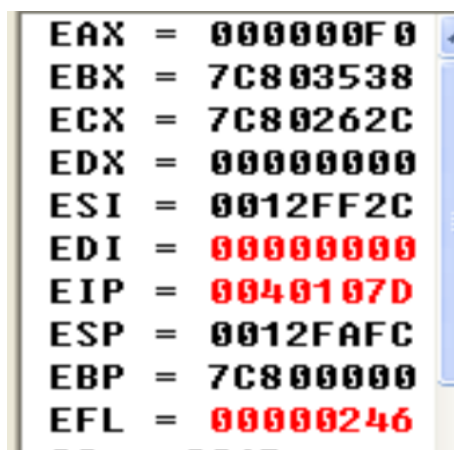
```

4  mov ecx,[ebp+eax+0x78] //导出表的指针
5  add ecx,ebp //ecx=导出表的基地址
6  mov ebx,[ecx+0x20] //导出函数名列表指针
7  add ebx,ebp //ebx=导出函数名列表指针的基地址
8  xor edi,edi

```

定位 kernel32.dll 的导出表:

ebp 的值为 kernel32.dll 的基地址。利用 ebp 是将 dll 的 PE 头存入 eax，将导出表的基地址存入 ecx，将导出函数名列表指针的基地址存入 ebx。



EAX	=	000000F0
EBX	=	7C803538
ECX	=	7C80262C
EDX	=	00000000
ESI	=	0012FF2C
EDI	=	00000000
EIP	=	0040107D
ESP	=	0012FAFC
EBP	=	7C800000
EFL	=	00000246

3.3.5 搜索定位 LoadLibrary 等目标函数

```

1  next_function_loop:
2      inc edi
3      mov esi,[ebx+edi*4] //从导出函数名列表数组中读取
4      add esi,ebp //esi = 函数名称所在地址
5      cdq //edx = 0

```

从导出函数名列表数组中读取函数名，esi 赋值为函数名称所在地址。

```

1  hash_loop:
2      movsx eax,byte ptr[esi]
3      cmp al,ah //字符串结尾就跳出当前函数
4      jz compare_hash
5      ror edx,7
6      add edx,eax
7      inc esi
8      jmp hash_loop
9  compare_hash:
10     cmp edx,[esp+0x1C] //lods pushad 后,栈+1c 为 LoadLibraryA 的 hash
11     jnz next_function_loop
12     mov ebx,[ecx+0x24] //ebx = 顺序表的相对偏移量
13     add ebx,ebp //顺序表的基地址
14     mov di,[ebx+2*edi] //匹配函数的序号

```

```

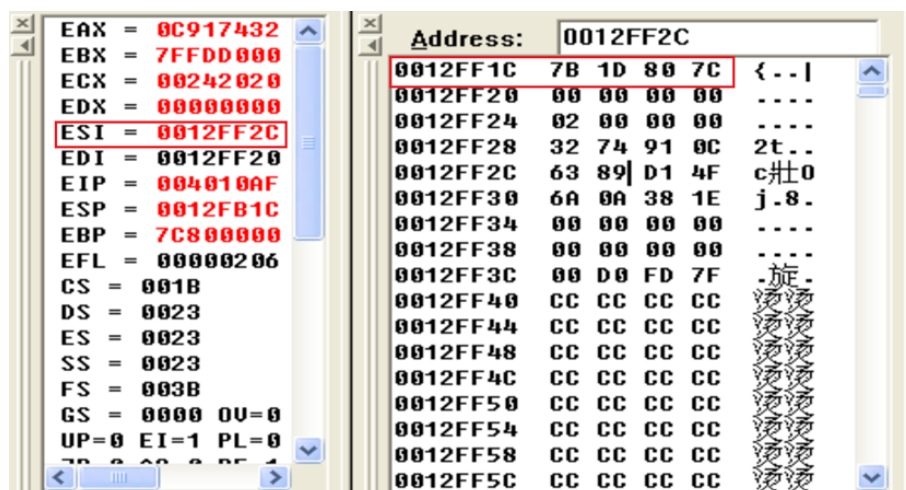
15  mov ebx,[ecx+0x1C] //地址表的相对偏移量
16  add ebx,ebp //地址表的基地址
17  add ebp,[ebx+4*edi] //函数的基地址
18  xchg eax,ebp //eax<=>ebp 交换
19  pop edi
20  stosd //把找到的函数保存到 edi 的位置
21  push edi
22  popad
23  cmp eax,0x1e380a6a //找到最后一个函数 MessageBox 后,跳出循环
24  jne find_lib_functions

```

从列表数组中依次取值,进入 hash_loop 计算其哈希值,然后跳转函数compare_hash 进行比较,判断当前 hash 是否目的函数的,如果不是,跳回 next_function_loop 比较下一个函数,直到成功找到。

成功找到后:把找到的函数存到 edi 的位置。

加断点调试,可以发现 edi 为 0x0012ff2c,跳转该地址处,该地址处存放 0x7c801d7b.



利用 depends 工具查看 LoadLibraryA 的偏移地址为 0x00001d7b,加上基地址 0x7c800000,地址就是 0x7c801d7b.

l	~	Hint	Function	Entry Point
(0x0241)	576	(0x0240)	LZRead	0x00066219
(0x0242)	577	(0x0241)	LZSeek	0x0006618E
(0x0243)	578	(0x0242)	LZStart	0x0008012E
(0x0244)	579	(0x0243)	LeaveCriticalSection	0x000091AD
(0x0245)	580	(0x0244)	LoadLibraryA	0x00001D7B
(0x0246)	581	(0x0245)	LoadLibraryExA	0x00001D53
(0x0247)	582	(0x0246)	LoadLibraryExW	0x00001AF5
(0x0248)	583	(0x0247)	LoadLibraryW	0x0000AEDB

不断重复上述过程,直到找到最后一个函数 MessageBox.

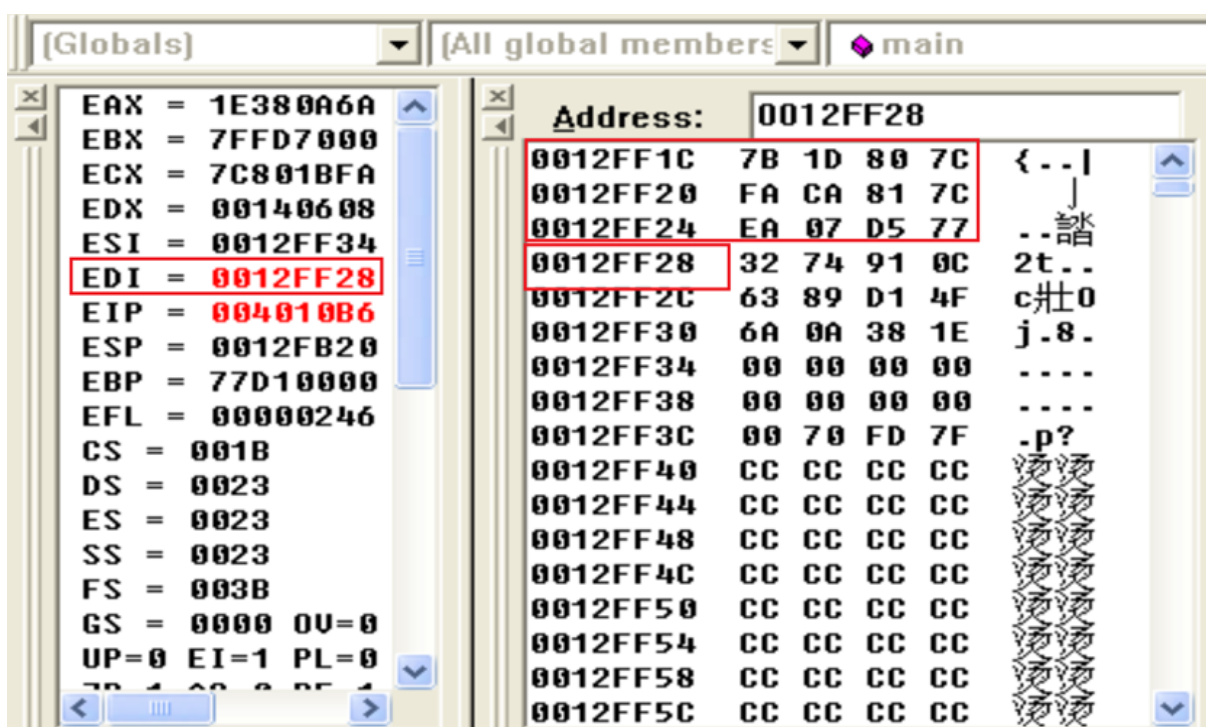
```

1  xchg eax,ebp
2  call [edi-0x8] //LoadLibraryA("user32") |
3  xchg eax,ebp //ebp=user32.dll 的基地址,eax=MessageBoxA 的 hash

```

找到 MessageBox 函数后,调用 LoadLibraryA("user32"),上述代码执行后 ebp 为 user32.dll

的基地址，eax 为 MessageBoxA 的 hash。



上述代码执行完后,edi-0x12 为 0x7c801d7b,为 LoadLibraryA 的地址(基址 0x7c80000+0x00001d7b)。

l	^	Hint	Function	Entry Point
(0x0241)	576	(0x0240)	LZRead	0x00066219
(0x0242)	577	(0x0241)	LZSeek	0x0006618E
(0x0243)	578	(0x0242)	LZStart	0x0008012E
(0x0244)	579	(0x0243)	LeaveCriticalSection	0x000091AD
(0x0245)	580	(0x0244)	LoadLibraryA	0x00001D7B
(0x0246)	581	(0x0245)	LoadLibraryExA	0x00001D53
(0x0247)	582	(0x0246)	LoadLibraryExW	0x00001AF5
(0x0248)	583	(0x0247)	LoadLibraryW	0x0000AEDB

edi-0x8 为 0x7c81cafa, 为 ExitProcess 的地址(基址 0x7c80000+0x0001cafa)。

l	^	Hint	Function	Entry Point
(0x00B5)	180	(0x00B4)	EraseTape	0x0006C0FB
(0x00B6)	181	(0x00B5)	EscapeCommFunction	0x00066771
(0x00B7)	182	(0x00B6)	ExitProcess	0x0001CAFA
(0x00B8)	183	(0x00B7)	ExitThread	0x0000C0E8
(0x00B9)	184	(0x00B8)	ExitVDM	0x00068675
(0x00BA)	185	(0x00B9)	ExpandEnvironmentStringsA	0x000329F1
(0x00BB)	186	(0x00BA)	ExpandEnvironmentStringsW	0x000305E6
(0x00BC)	187	(0x00BB)	ExpungeConsoleCommandHistoryA	0x00071647

edi-0x4 为 0x77d507ea, 为 MessageBoxA 的地址(基址 0x77d10000+0x000407ea)。

Module	Time Stamp	Size	Attributes	Machine	Subsystem	Debug	Base	File Ver	Produ
GDI32.DLL	04/14/08 8:00p	285,184	A	Intel x86	Win32 console	Yes	0x77EF0000	5.1.2600.5512	5.1.2
KERNEL32.DLL	04/14/08 8:00p	1,150,464	A	Intel x86	Win32 console	Yes	0x7C800000	5.1.2600.5512	5.1.2
NTDLL.DLL	04/14/08 8:00p	589,312	A	Intel x86	Win32 console	Yes	0x7C920000	5.1.2600.5512	5.1.2
USER32.DLL	04/14/08 8:00p	574,976	A	Intel x86	Win32 GUI	Yes	0x77D10000	5.1.2600.5512	5.1.2

Ordinal	Hint	Function	Entry Point
473 (0x01D9)	472 (0x01D8)	MenuItemFromPoint	0x0005CD70
474 (0x01DA)	473 (0x01D9)	MenuWindowProcA	0x00046873
475 (0x01DB)	474 (0x01DA)	MenuWindowProcW	0x0004682E
476 (0x01DC)	475 (0x01DB)	MessageBeep	0x00021F7B
477 (0x01DD)	476 (0x01DC)	MessageBoxA	0x000407EA
478 (0x01DE)	477 (0x01DD)	MessageBoxExA	0x0004085C
479 (0x01DF)	478 (0x01DE)	MessageBoxExW	0x00040838
480 (0x01E0)	479 (0x01DF)	MessageBoxIndirectA	0x0002A082
481 (0x01E1)	480 (0x01E0)	MessageBoxIndirectW	0x000564D5

3.3.6 编写 shellcode

```

1 function_call:
2     xor ebx,ebx
3     push ebx
4     push 0x74736577
5     push 0x74736577 //push "westwest"
6     mov eax,esp
7     push ebx
8     push eax
9     push eax
10    push ebx
11    call [edi-0x04] //MessageBoxA(NULL,"westwest","westwest",NULL)
12    push ebx
13    call [edi-0x08] //ExitProcess(0);
14

```

最后执行 shellcode. 调用 messagebox 函数, 弹出窗口, 最调用用 ExitProcess 函数, 退出程序。

4 心得体会

通过本次实验, 加深了对 API 函数自搜索技术原理的理解, 掌握了 API 函数自搜索步骤及实现方法。