

《软件安全》实验报告

姓名：邢清画 学号： 2211999 班级： 1023（物联网）

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. 复现第五章实验七

通过加断点在合适的位置，便于了解 API 函数自搜索的机制。

断点 mov esi,esp 前的 3 个 push，用来 push MessageBoxA，ExitProcess 和 LoadLibraryA 的哈希值（通过独立的程序算出），在后面做函数名的比较时，比较的是哈希值而不是字符串。

Esi 的值 0012FF28 在整个程序里面始终不会改变，始终指向在栈中的三个哈希值。



```
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
```

Registers

EAX	=	CCCCCCCC	EBX	=	7FFDF000
ECX	=	00000000	EDX	=	004300B0
ESI	=	0012FF28	EDI	=	0012FF80
EIP	=	0040103A	ESP	=	0012FF28
EBP	=	0012FF80	EFL	=	00000202
CS	=	1	DS	=	0023
ES	=	0023	SS	=	0023
FS	=	1	GS	=	0000
OV	=	0	UP	=	0
EI	=	1	PL	=	0
ZR	=	0	PE	=	0
CV	=	0			

mov bh, 0x04 抬高 bh, 影响的是 EBP 的值。

sub esp, ebx 从栈顶指针 esp 中减去 ebx 的值，把 esp 抬高（值变小）。



```
int main()
{
    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
    ...
}
```

Registers

EAX	=	CCCCCCCC	EBX	=	00000400
ECX	=	00000000	EDX	=	004300B0
ESI	=	0012FF28	EDI	=	0012FF1C
EIP	=	00401041	ESP	=	0012FF28
EBP	=	0012FF80	EFL	=	00000246
CS	=	1	DS	=	0023
ES	=	0023	SS	=	0023
FS	=	1	GS	=	0000
OV	=	0	UP	=	0
EI	=	1	PL	=	0
ZR	=	1	PE	=	1
CV	=	0	ST0	=	+0.0000000000000000e+0000

之后压入"user32.dll"，两个 push 用来处理字符串"user"和"32"，



```
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	=	004300B0
ESI	=	0012FF28	EDI	=	0012FF1C
EIP	=	00401048	ESP	=	0012FF28
EBP	=	0012FF80	EFL	=	00000206
CS	=	0023	DS	=	0023
ES	=	0023	SS	=	0023
FS	=	0	GS	=	0000
OV	=	0	UP	=	0
EI	=	1	PL	=	0
ZR	=	0	PE	=	1
CV	=	0	ST0	=	+0.0000000000000000e+0000

地址压栈，用于后续的 LoadLibraryA 调用，0x72657375 是“user”的 ASCII 码。

```
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]
```

Registers

EAX = CCCCCC	EBX = 00003233
ECX = 00000000	EDX = 00430000
ESI = 0012FF28	EDI = 0012FF1C
EIP = 00401040	ESP = 0012FB20
EBP = 0012FF80	EFL = 00000206
CS = 001B	DS = 0023
ES = 0023	SS = 0023
FS = 003B	GS = 0000
OV=0	UP=0
EI=1	PL=0
ZR=0	AC=0
PE=1	CV=0
ST0 = +0.0000000000000000e+0000	

Address: 0x0012FB20

75	73	65	72	33	32	00	00	0C	0D	0E	0F	10	11	12	13	14	15	16	17	user32.....									
18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B !"\$%&'()*+,-./0123456789:;<=>?									
2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F !"\$%&'()*+,-./0123456789:;<=>?									

之后 push esp，此时 esp 中存储的值变小，现在的 0x0012FB1C 存储的是之前的 0x0012FB20 (user32 字符串的地址)

```
//=====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]
```

Registers

EAX = CCCCCC	EBX = 00003233
ECX = 00000000	EDX = 00430000
ESI = 0012FF28	EDI = 0012FF1C
EIP = 0040104E	ESP = 0012FB1C
EBP = 0012FF80	EFL = 00000206
CS = 001B	DS = 0023
ES = 0023	SS = 0023
FS = 003B	GS = 0000
OV=0	UP=0
EI=1	PL=0
ZR=0	AC=0
PE=1	CV=0
ST0 = +0.0000000000000000e+0000	

Address: 0x0012FB1C

20	FB	12	00	75	73	65	72	33	32	00	00	0C	0D	0E	0F	10	11	12	13	...user32.....									
14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27 !"\$%&'()*+,-./0123456789:;<=>?									
28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B !"\$%&'()*+,-./0123456789:;<=>?									
3C	3D	3E	3F	40	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F !"\$%&'()*+,-./0123456789:;<=>?									
70	71	72	73	74	75	76	77	78	79	7A	5B	5C	5D	5E	5F	60	61	62	63	pqrstuvwxyz[\]^_`abc									

然后获取 PEB 的地址，访问 PEB_LDR_DATA 结构来遍历加载的模块列表，动态解析函数地址，最终找到 kernel32.dll 的基地址 (EBP 存储的 7C800000)。

```
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]
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:
```

Registers

EAX = CCCCCC	EBX = 00003233
ECX = 00242020	EDX = 00000000
ESI = 0012FF28	EDI = 0012FF1C
EIP = 0040105F	ESP = 0012FB1C
EBP = 7C800000	EFL = 00000206
CS = 0023	DS = 0023
ES = 0023	SS = 0023
FS = 003B	GS = 0000
OV=0	UP=0
EI=1	PL=0
ZR=0	AC=0
PE=1	CV=0
ST0 = +0.0000000000000000e+0000	

Address: 0x0012FB1C

20	FB	12	00	75	73	65	72	33	32	00	00	0C	0D	0E	0F	10	11	12	13	...user32.....									
14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27 !"\$%&'()*+,-./0123456789:;<=>?									
28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B !"\$%&'()*+,-./0123456789:;<=>?									
3C	3D	3E	3F	40	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F !"\$%&'()*+,-./0123456789:;<=>?									
70	71	72	73	74	75	76	77	78	79	7A	5B	5C	5D	5E	5F	60	61	62	63	pqrstuvwxyz[\]^_`abc									

Lodsd 从 esi 指向的地址加载一个值到 eax 并将 esi 增加 4，用来逐个检查压栈的哈希值。如 0C917432 就是 LoadLibraryA 的 hash。

```
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=user32.dll的基地址,eax=MessageBoxA的hash <-- |
```

Registers

EAX = 0C917432	EBX = 7FFDF1
ECX = 00242020	EDX = 00000000
ESI = 0012FF2C	EDI = 0012FF1C
EIP = 00401060	ESP = 0012FB1C
EBP = 7C800000	EFL = 00000206
DS = 0023	ES = 0023
SS = 001B	GS = 0000
OV=0	UP=0
EI=1	PL=0
ZR=0	AC=0
PE=1	CV=0
ST0 = +0.0000000000000000e+0000	

Address: 0x0012FB1C

20	FB	12	00	75	73	65	72	33	32	00	00	0C	0D	0E	0F	10	11	12	13	...user32.....									
14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27 !"\$%&'()*+,-./0123456789:;<=>?									
28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B !"\$%&'()*+,-./0123456789:;<=>?									
3C	3D	3E	3F	40	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F !"\$%&'()*+,-./0123456789:;<=>?									
70	71	72	73	74	75	76	77	78	79	7A	5B	5C	5D	5E	5F	60	61	62	63	pqrstuvwxyz[\]^_`abc									

Cmp 作比较，判断是否是最后一个需要查找的哈希值，如果不是，jne 跳转到 find_functions 中，通过访问 DLL 的 PE 头和导出表，可以找到函数名和它们对应的实际地址。在导出表中逐一检查每个函数名。且 ebp 会被更新为目标函数的地址。不断寻找，直到找到后开始计算虚拟地址，最后把找到的函数保存到 edi 中。

```

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

    //=====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

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            +4
    stosd
    push    edi
    popad
    cmp     eax,0x1e380a6a //找到最后一个函数MessageBox后,跳出
    jne     find_lib_functions //如果不是继续循环

```

Registers	
EAX = 00000041	EBX = 7C803538
ECX = 7C80262C	EDX = 00000000
ESI = 7C8040AA	EDI = 00000001
EIP = 00401087	ESP = 0012FAFC
EBP = 7C800000	EFL = 00000206
DS = 0023	ES = 0023
SS = 0023	FS = 0000
OU=0	UP=0
EI=1	PL=0
ZR=0	PE=1
CV=0	ST0 = +0.0000000000000000e+0000


```

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            +4
    stosd
    push    edi
    popad
    cmp     eax,0x1e380a6a //找到最后一个函数MessageBox后,跳出
    jne     find_lib_functions //如果不是继续循环

```

Registers	
EAX = 00000000	EBX = 7C802654
ECX = 7C80262C	EDX = 0C917432
ESI = 7C807649	EDI = 00000244
EIP = 004010AA	ESP = 0012FAFC
EBP = 7C801D7B	EFL = 00000206
DS = 0023	ES = 0023
SS = 0023	FS = 0000
OU=0	UP=0
EI=1	PL=0
ZR=0	PE=1
CV=0	ST0 = +0.0000000000000000e+0000

不断重复上述过程，直到找到最后一个函数的 hash，跳出循环，执行 call 语句，观察到 esp 的值是 0x0012FB20，即 user32 字符串的地址，完成了 LoadLibraryA("user32")的调用。

```

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=user32.dll的基地址,eax=MessageBoxA的
    //=====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

```

Registers	
EAX = 77D10000	EBX = 7FFDF000
ECX = 7C801BFA	EDX = 00140608
ESI = 0012FF34	EDI = 0012FF24
EIP = 0040106B	ESP = 0012FB20
EBP = 1E380A6A	EFL = 00000246
DS = 0023	ES = 0023
SS = 0023	FS = 0000
OU=0	UP=0
EI=1	PL=0
ZR=1	PE=1
CV=0	ST0 = +0.0000000000000000e+0000

继续执行观察到 ebp 的值发生变化，77D10000 其实是 user32.dll 的基地址。

```

10050 //move eax,[esi], esi=4, 第一个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的基地址

//=====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

```

Registers	
EAX = 1E380A6A	EBX = 7FFDF000
ECX = 7C801BFA	EDX = 00140608
ESI = 0012FF34	EDI = 0012FF24
EIP = 0040106C	ESP = 0012FB20
EBP = 77D10000	EFL = 00000246
CS = 01	DS = 0023
ES = 0023	SS = 0023
FS = 01	GS = 0000
OU=0	UP=0
EI=1	PL=0
ZR=1	AC=0
PE=1	CY=0
ST0 = +0.0000000000000000e+0000	

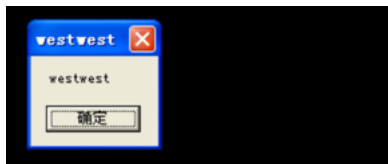
构造调用 MessageBoxA 的参数，并调用 ExitProcess 函数以结束进程，使用 NOP 指令作为填充。

```

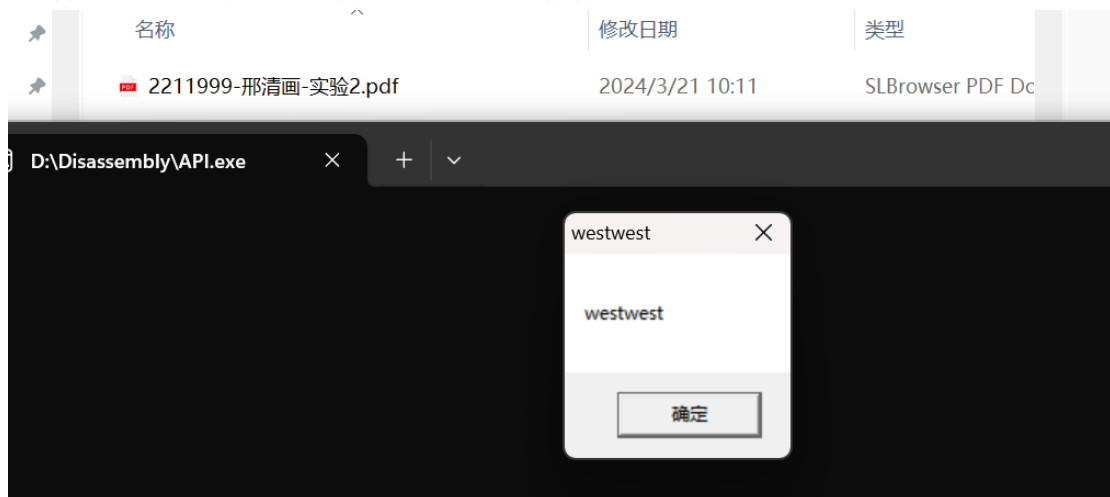
function_call:
xor     ebx,ebx
push    ebx
push    0x74736577 //push "westwest"
mov     eax,esp
push    ebx
push    eax
push    ebx
push    ebx
call    [edi-0x04] //MessageBoxA(NULL,"westwest","westwest",NULL)
push    ebx
call    [edi-0x08] //ExitProcess(0);
nop
nop
nop

```

在 VC6 中运行显示 westwest 对话框：



2. 将生成的 exe 程序，复制到 windows 10 操作系统里验证成功。



心得体会：

学会了动态定位函数 API 的方法，更加了解 PE 文件结构，掌握了如何提高 Shellcode 的跨平台性和适应性，加深了我对汇编语言的理解，提高了汇编语言能力。