《软件安全》实验报告

姓名：禹相祐　　学号：2312900　　班级：计算机科学与技术

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。

实验过程：

本次实验需要编写能在不同系统中通用的 shellcode 代码，让其不仅能在 x86 同时也要在 windows 11 下运行。要想实现这个功能，就要求我们的 shellcode 代码能有动态 API 函数地址自搜索的功能。

1. 梳理全程的思路

a. MessageBoxA 位于 user32.dll 中，能拿来弹出对话框；
b. ExitProcess 位于 kernel32.dll 中，能拿来退出程序。需要注意的是，所有程序都会自动加载 kernel32.dll 这个库，所以也就成为了我们实现定位的切入点。
c. 而 LoadLibraryA 位于 kernel32.dll 中。所以我们需要先通过 kernel32.dll 实现定位，再定位 LoadLibraryA 等函数，最终实现 shellcode 编写。

2. 定位 kernel32.dll

代码如下：

```
01 // 压入 user32.dll
02 mov bx,0x3233
03 push ebx
04 push 0x72657375
05 push esp
06 xor edx,edx
07 // 找 kernel32.dll 的基址
08 mov ebx,fs:[edx+0x30]  //[TED+0x30]-->PEB
09 mov ecx,[ebx+0xC]      //[PEB+0xC]--->PEB_LDR_DATA
```

```
mov ecx,[ecx+0x1C]
//[PEB_LDR_DATA+0x1C]--->InInitializationOrderMoudleList
mov ecx,[ecx]          // 进入链表第一个就是 ntdll.dll
mov ebp,[ecx+0x8]      // ebp= kernel32.dll 的基址
```

理解：

首先将 user32.dll 的地址压入栈中，并且通过 xor 将 edx 设置为 0，然后通过 fs 段寄存器定位到当前的线程块 TEB，加上 0x30 的偏移量得到 PEB 并保存在 ebx 内，然后再偏移 0xC 到 PEB_LDR_DATA，然后再是 0x1C 的偏移量到 InInitializationOrderMoudleList。直到找到这个链表后，第一个就是节点 ntdll.dll，再偏移 8 位就是 kernel32.dll，至此实现定位。

3. 定位 kernel32.dll 的导出表

代码如下：

```
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
```

理解：

首先将 ebp 地址偏移 0x3C 指向 PE 头指针，而 PE 头指针偏移 0x78 处存放着导出表的指针，所以将 ebp+eax+0x78 就得到了导出表的基地址，然后再偏移 0x20 指向函数名的指针，最后再加上 ebp 去获得函数名列表的基本地址。后续只需要一个个对比 hash 值就能找到我们所需要的特定函数。

4. 定位 LoadLibrary 等特定目标函数

代码如下：

```
#include <stdio.h>
#include <windows.h>
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;
}
main()
{
    DWORD hash;
    Hash = GetHash("MessageBoxA");
    printf("%#x\n",hash);
}
```

然后将三个函数名称的 hash 值入栈：

```
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] //为了兼容性空出 8 字节
```

然后通过三个函数 find_lib_functions & find_functions & next_function_loop 进行循环从而找到我们需要的三个函数的地址。

```
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

//======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

//======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
```

理解：

其实这些函数就如名字一样：Find_lib_functions 通过调用 find_functions 实现寻找函数，next_function_loop 即如果不符合要求，那就一直往后寻找，找到就跳出循环。

Hash 循环和 Hash 值对比的函数如下：

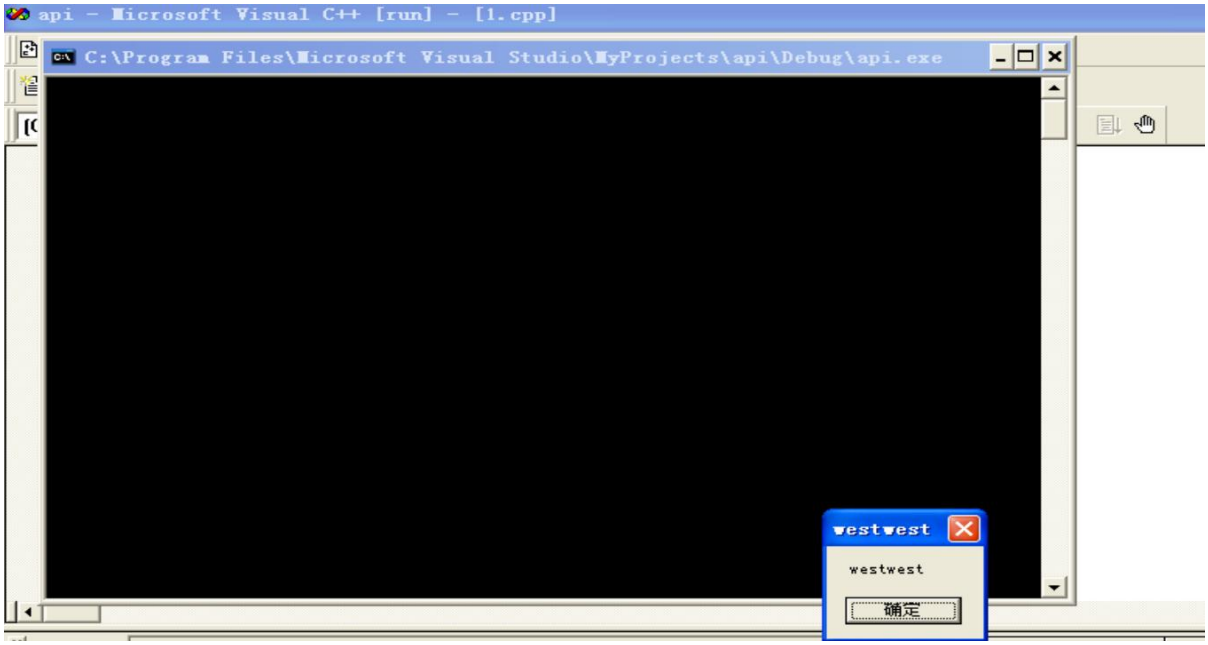
```
hash_loop:
    movsx eax,byte ptr[esi]
    cmp al,ah //字符串结尾就跳出当前函数
    jz compare_hash
    ror edx,7
    add edx,eax
    inc esi
    jmp hash_loop

//======比较当前函数的 hash 是否是自己想找的 hash
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 // 保存到 edi 的位置
    push edi
    popad
    cmp eax,0x1e380a6a //找到 MessageBox 后，跳出循环
    jne find_lib_functions
```

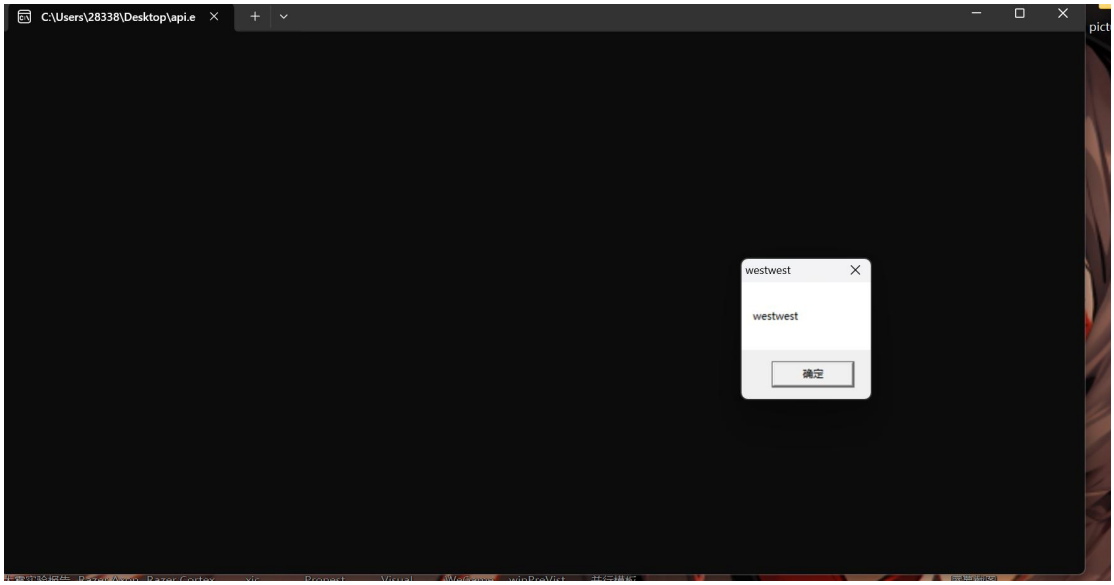
5. 完成 shellcode 代码的编写：

```
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;
}
```

运行如图：



在 windows11 上运行：



心得体会：

此次实验让我掌握了 API 函数的子搜索技术，学会了通过 TEB、PEB 等逐步通过加偏移量实现定位，并通过比较 hash 值最终确定需要的函数；另外，通过此次实验，也让我对汇编指令有了更深的理解。