《API 函数自搜索定位》实验报告

姓名：汤清云 学号：2013536 班级： 1075

实验名称：

API 函数自搜索定位技术

实验要求：

1. 定位 kernal32.dll：
   （1） 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。
2. 找到 kernal32.dll 的导出表：
   （1） 从 kernel32.dll 加载基址算起，偏移 0x3c 的地方就是其 PE 头的指针。
   （2） PE 头偏移 0x78 的地方存放着指向函数导出表的指针。
   （3） 获得导出函数偏移地址（RVA）列表、导出函数名列表：

   ①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。

   ②导出表偏移 0x20 处的指针指向存储导出函数函数名的列表。
3. 搜索定位目标函数：
   （1） 函数的 RVA 地址和名字按照顺序存放在上述两个列表中，我们可以在名称列表中定位到所需的函数是第几个，然后在地址列表中找到对应的 RVA。
   （2） RVA 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，这个地址就是最终在 ShellCode 中调用时需要的地址。

实验过程：

```
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
```

将 MessageBoxA/ExitProcess/LoadLibraryA 函数的字符串转为哈希值后先后压入栈内。在之后作函数名比较时也是使用字符串的哈希值进行比较。

```
mov esi,esp                 //esi=esp,指向堆栈中存放LoadLibraryA的地址
```

将此时 ESP 的值赋值给 ESI，使得 ESI 的值为 0012FF28，用以标注三个函数名哈希值存放处。

```
lea  edi,[esi-0xc]                    //空出8字节应该是为了兼容性
```

EDI 指向未压入三个函数名字哈希值之前的栈顶位置，即 0012FF1C

```
xor     ebx,ebx
mov     bh,0x04
sub     esp,ebx                       //esp-=0x400
```

将 ebx 清零后赋值为 00000400（bh 为 bx 的高八位），将栈顶抬高 0x400，增加栈空间。

```
//======压入"user32.dll"
mov     bx,0x3233
push    ebx                   //0x3233
push    0x72657375            //"user"
push    esp
xor     edx,edx               //edx=0
```

将‘32’存放在 ebx 中，bx 是 ebx 的低十六位，故 ebx 值为 00003233，再将“user”的字符串转哈希值压入栈，此时 esp 处则存下了 user.32 的哈希值，再将 esp 压入栈，即将字符串“user32”字符串哈希值地址压入栈中。最后将 edx 归零。

```
//======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]      //[PEC_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
```

由实验原理 1 所得出代码，此时 ebp 存放了 kernel32 的基地址，为 7C800000

```
lodsd   //即mov eax,[esi],esi+=4, 第一次取LoadLibraryA的hash
```

此时 eax 值为：0C917432，为 LoadLibrary 的哈希值，ESI 的值变为 0012FF2C

```
cmp eax,0x1E380A6A          //与MessageBoxA的hash比较
jne find_functions  //如果没有找到最后一个函数，继续找
xchg     eax,ebp
call    [edi-0x8]     //LoadLibraryA("user32") |
xchg    eax,ebp
```

与 MessageBox 的哈希值比较，查看是否为最后一个需要找的函数的哈希值，如果不是的话需要继续寻找，跳转到 find_functions 函数处完成后续操作。如果是的话则将 eax 的值与 ebp 互换，调用 LoadLibrary 的 user32.dll，并返回后再次交换 eax 与 ebp 的值，则此时 ebp 存放了 user32 的基地址，eax 存放了 messagebox 的哈希值。

```
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
```

函数逻辑为：定义到导出表，再定位到导出函数名列表。

```
//======找下一个函数名
next_function_loop:
        inc edi
        mov     esi,[ebx+edi*4]       //从列表数组中读取函数名
        add esi,ebp    //esi = 函数名称所在地址
        cdq                           //edx = 0
```

依次取出下一个未访问过的函数。

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

使用 hash_loop 计算出函数名字符串对应的哈希值。

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

使用 compare_hash 来进行判别，比较当前找出的函数的哈希值是否是自己想要的，如果不是则跳转回 next_function_loop 寻找下一个函数名，继续进行上述操作，直到匹配成功。

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

找到之后同实验原理 3，计算出其虚拟地址。

```
        pop edi
        stosd                   //把找到的函数保存到edi的位置
        push    edi
        popad                 //一次性完成多个寄存器状态保存和恢复
        cmp eax,0x1e380a6a  //找到函数MessageBox后，跳出循环
        jne find_lib_functions
```

EDI 由 00000244 变为 0012FF1C(存放着刚才找到的虚拟地址)，再将其+4 后压入栈中。

```
        popad                   //一次性完成多个寄存器状态保存和恢复
```

保存多个寄存器的状态。

```
        cmp eax,0x1e380a6a  //找到函数MessageBox后，跳出循环
        jne find_lib_functions
```

判断是否为我们需要找的最后一个函数 MessageBox 的哈希值，若不是则继续寻找，是则跳出循环。

```
        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
```

找到 messagebox 函数后，通过以上语句计算出其地址。

```
        pop edi
        stosd                           //把找到的函数保存到edi的位置
        push    edi
        popad                         //一次性完成多个寄存器状态保存和恢复
        cmp eax,0x1e380a6a  //找到函数MessageBox后，跳出循环
        jne find_lib_functions
```

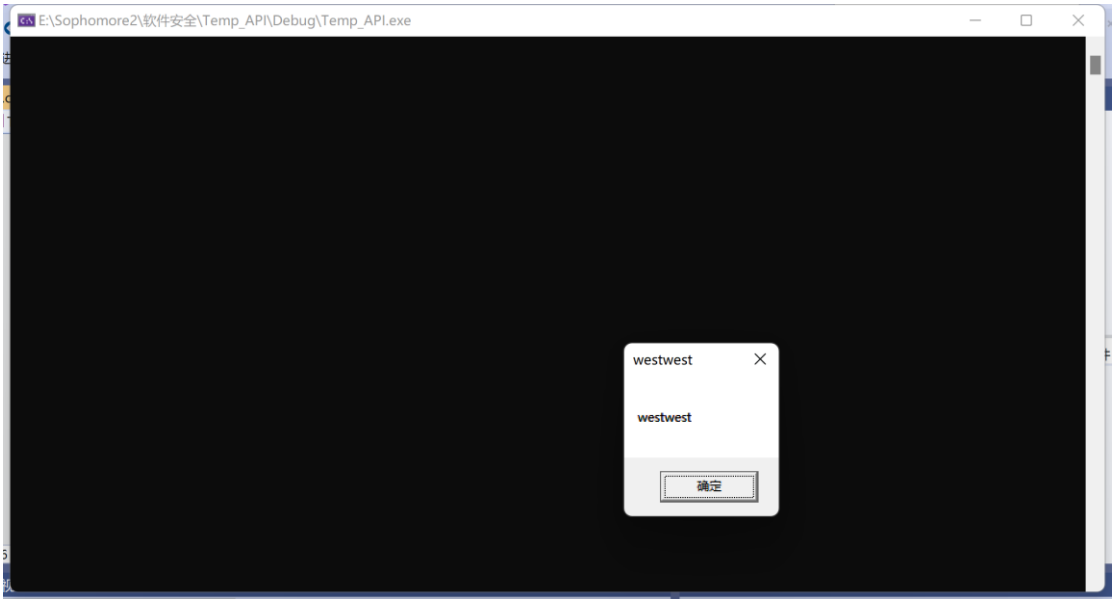
将计算出的地址放入 edi 中，再次比较是否为 messagebox 函数。此时 edi 存放了三个函数的虚拟地址。

```
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
    }
```

此处进行 shellcode 编写，将 ebx 清零后压入栈中。将“westwest”的 ascii 码压入栈中，esp（westwest 字符串的地址）赋值给 eax，压入 ebx，eax，eax，ebx，调用 messagebox 函数，进行调用时会自动调取栈中前四个函数，故对应了 null,westwest,westwest,null

调用结果见上。之后再次压入 ebx（0），调用 exitprocess 函数，退出程序。

在 windows11 VS2019 版本运行结果为：



总结心得：

学习了如何进行 API 函数的自搜索，了解了 kernel32 以及 user32 的 dll 的查询。