软件安全实验报告

姓名：郭子涵 学号：2312145 班级：信息安全、法学双学位班

目录:

---

1 实验名称

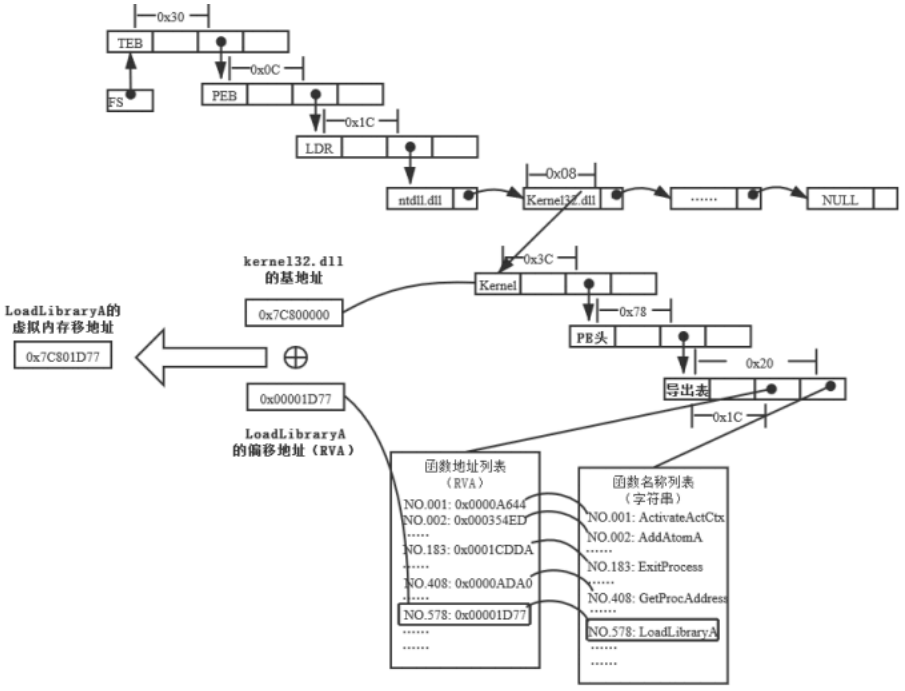
API函数自搜索实验

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windows10操作系统里验证是否成功。

3 实验流程

在上次实验中我们以硬编码的方式调用相应的API函数实现shellcode攻击，本次实验为编写通用shellcode，那么shellcode自身就必须具备动态的自动搜索所需API函数地址的能力，即API自搜索计数。如图所示，大致步骤分为: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。具体流程下文详述。

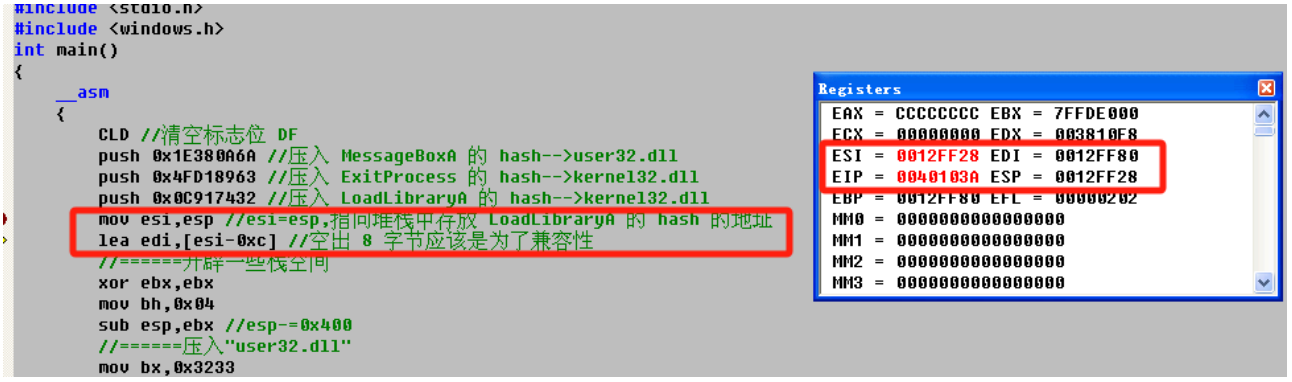


3.1 定位kernel32.dll

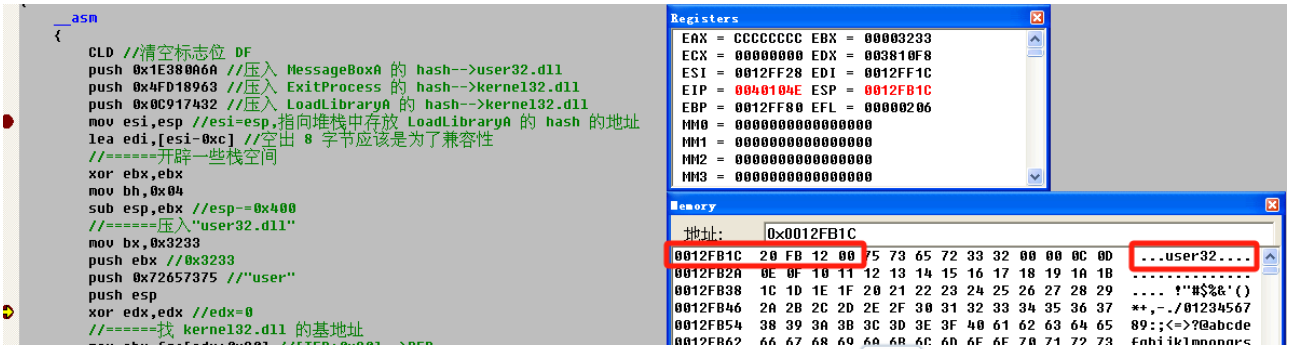
```
__asm
{
    CLD //清空标志位 DF
    push 0x1E380A6A //压入 MessageBoxA 的 hash-->user32.dll
    push 0x4FD18963 //压入 ExitProcess 的 hash-->kernel32.dll
    push 0x0C917432 //压入 LoadLibraryA 的 hash-->kernel32.dll
    mov esi,esp //esi=esp,指向堆栈中存放 LoadLibraryA 的 hash 的地址
    lea edi,[esi-0xc] //空出 8 字节应该是为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

MessageBoxA，ExitProcess，LoadLibraryA三个字符串的哈希值入栈，由函数名的比较变为哈希值的比较，将变址寄存器ESI中存入哈希值的偏移便于之后的定位。

MessageBoxA函数在ueser32的导出表中，LoadLibraryA函数动态加载user.dll模块，才能从中找到MessageBoxA函数的地址并调用它。



将user32.dll的参数值都入栈，将来栈顶正好即为loadlibraryA的参数：



随后，首先通过段选择字 FS 在内存中找到当前的线程环境块 TEB。TSB偏移地址为 0x30 的地址存放着指向进程环境块 PEB 的指针。 PEB 偏移地址为 0x0c 的地方存放着指向 PEB_LDR_DATA 结构体的指针，存放着已经被进程装载的动态链接库的信息。该指针偏移位置为 0x1C 的地址存放着指向模块初始化链表的头指针(InInitializationOrderModuleList)，其中按顺序存放着 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。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。

3.2 定位kernel32.dll的导出表

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

lodsd指令是的eax，esi的值均发生改变，随后比较，若不是MessageBoxA函数即还没找到则继续找，跳转到find_functions函数处，第一轮定位kenel32.dll的导出表（第二轮定位user的导出表找到MessageBoxA函数）：

1.从 kernel32.dll 加载基址算起，偏移 0x3c 的地方就是其 PE 头的指针。

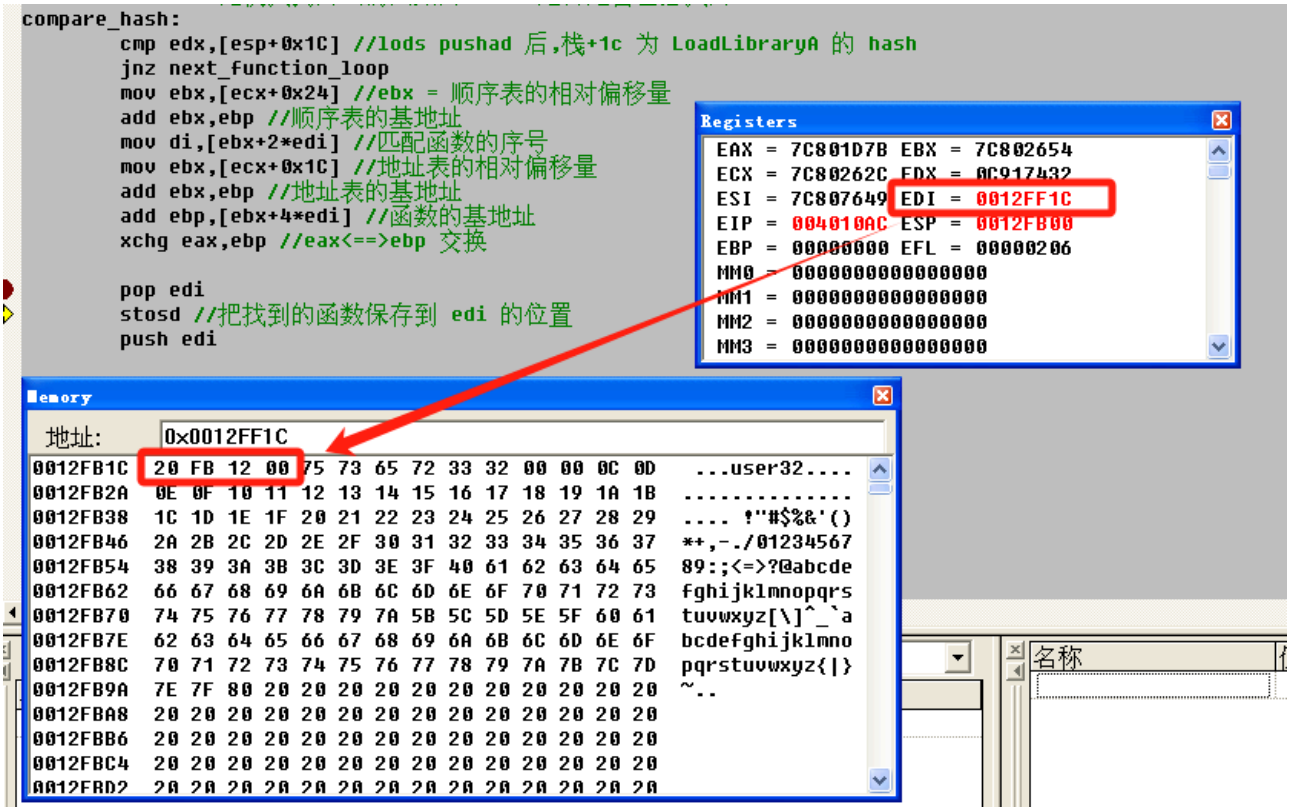
2.PE 头偏移 0x78 的地方存放着指向函数导出表的指针。

3.导出表偏移 0x20 处的指针指向存储导出函数函数名的列表。

3.3 搜索定位目标函数

```
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
 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
```

从导出函数列表中取函数名，随后在hash_loop中计算函数名的hash值，循环多次知道哈希值计算完毕后跳转到compare_hash函数和LoadlibraryA函数名的哈希值进行比较，如果不是则再此跳转到寻找下一个函数名的函数`next-function_loop`继续找知道找到正确的，计算其虚拟地址，把找到的函数地址保存在edi(目标地址寄存器)，查看地址验证：



与messageBox比较不相同，回到find_lib_functions函数此时eax与MessageBoxA的哈希值相等，顺序执行：


```
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
```

调用LoadLibraryA("user32"),继续调试，EBP变为`0x77D10000`即user32.dll的基地址，之前ebp存放的时kenel32.dll的基地址，因此之后查找的即为user32导出表中的函数，最后找到MessageBoxA,在edi中保存三个函数的虚拟地址，进入function_call函数：

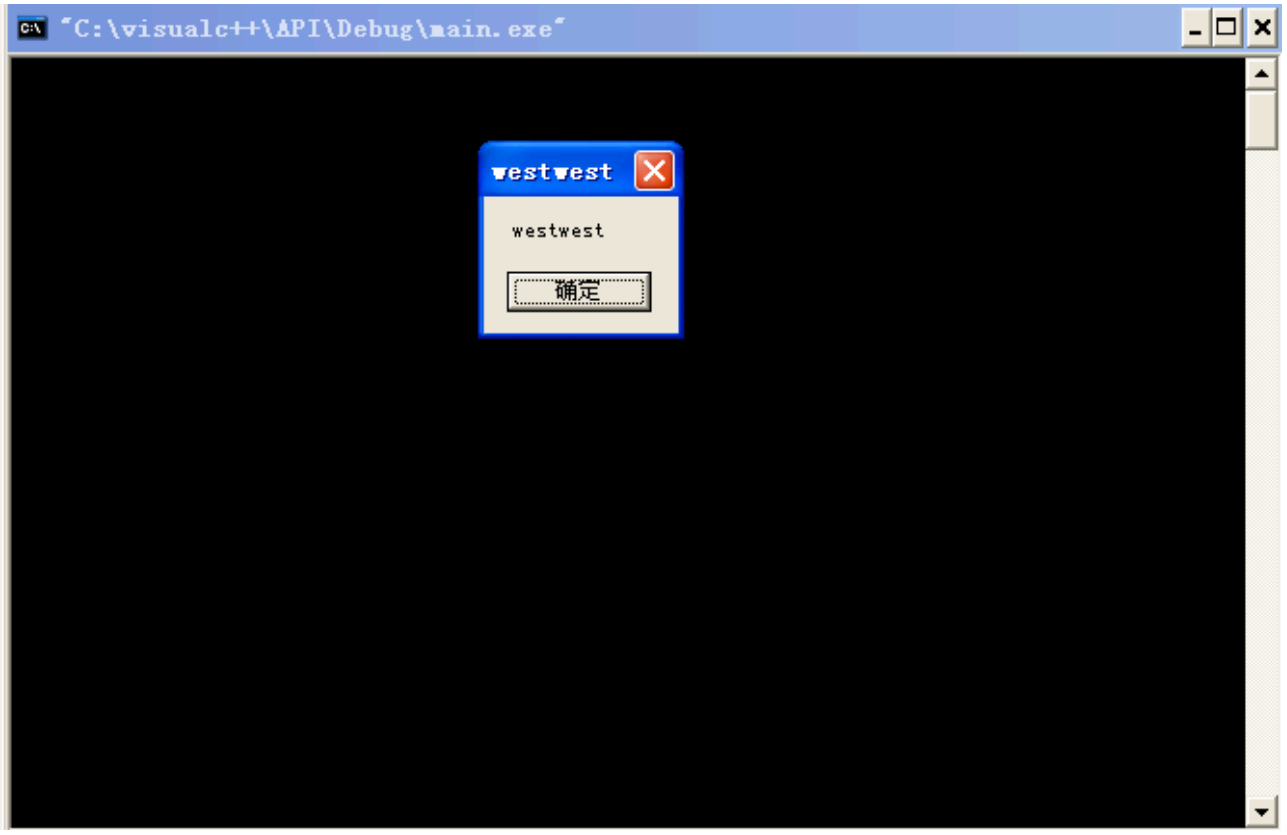
```
find_lib_functions:
        lodsd //即 move eax,[esi], esi+=4, 第一次取 LoadLibraryA 的 hash
        cmp eax,0x1E380A6A //与 MessageBoxA 的 hash 比较
        jne find_functions //如果没有找到 MessageBoxA 函数, 继续找
        xchg eax,ebp //-----------------------------------> |
        call [edi-0x8] //LoadLibraryA("user32") |
        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
```

```
Registers
EAX = 1E380A6A EBX = 7FFDE000
ECX = 7C801BFA EDX = 00140608
ESI = 0012FF34 EDI = 0012FF24
EIP = 00401066 ESP = 0012FB20
EBP = 77D10000 EFL = 00000246
MM0 = 0000000000000000
MM1 = 0000000000000000
MM2 = 0000000000000000
MM3 = 0000000000000000
```

3.4 基于找到的函数地址，完成shellcode的编写

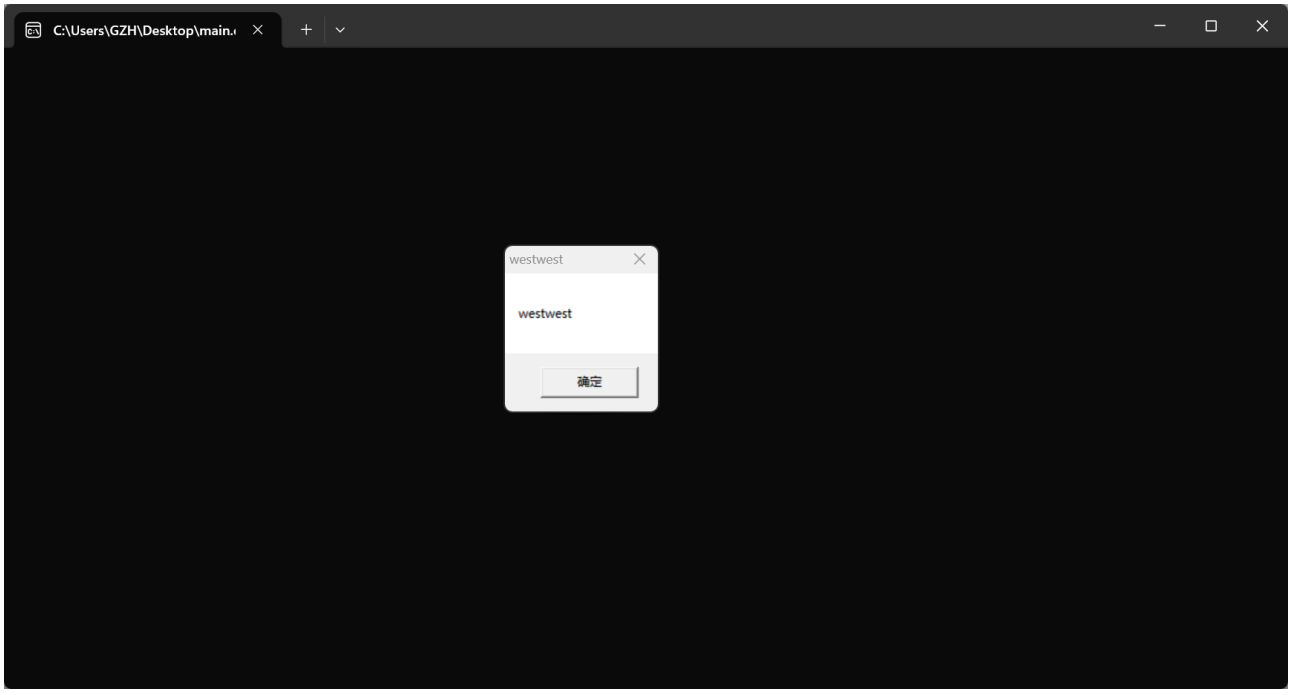
```
function_call:
        xor ebx,ebx
        push ebx
        push 0x74736577
        push 0x74736577 //push "westwest"
        mov eax,esp
        push ebx
        push eax
        push eax
        push ebx
        call [edi-0x04] //MessageBoxA(NULL,"westwest","westwest",NULL)
        push ebx
  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

调用MessageBox函数，弹出“westwest”，最后调用ExitProcess函数退出



3.5 windows10操作系统中执行

将生成的exe程序，复制到window10操作系统中，验证：



仍然成功！

4 心得体会

本次实验代码较长，实现的主要流程图总结如下：

```
APIFunction_Locator
├── 准备阶段
```

```
│   ├── push 函数哈希值（LoadLibraryA / ExitProcess / MessageBoxA）
│   ├── ESI 指向哈希列表，EDI 指向保存地址空间
├── Prepare_user32_string 构造 user32.dll 字符串
├── Get_kernel32_base 获取 kernel32.dll 基地址
├── 定位导出表
├── 主查找循环
│   └── Loop_Over_Hash_List
│       ├── lodsd → 当前目标哈希
│       ├── pushad → 保存现场
│       └── Find_Matching_Function
│           ├── Loop 函数名遍历
│           │   ├── [EBX + EDI*4] → 函数名 RVA → 加基址 → ESI
│           │   └── Compute_Hash
│           │       ├── lodsb → 每字节读取字符
│           │       ├── ror edx, 13 + add edx, eax → 计算哈希
│           │       └── 遇 0 结束 → jz
│           ├── Compare_Hash
│           │   └── cmp edx, [esp+0x1C] → 匹配当前目标哈希
│           ├── If_Match
│           │   ├── 获取函数序号 → [ExportTable+0x24]
│           │   ├── 获取地址偏移 → [ExportTable+0x1C]
│           │   ├── 实际地址 = EBP + offset
│           │   └── xchg eax, ebp → 保存到 EAX
│           └── pop edi / stosd → 保存到目标地址区
│               └── popad / 判断是否最后一个函数 → 否则继续
├── Call_LoadLibrary 加载 user32.dll
├── Call_MessageBox 调用弹窗
└── Call_ExitProcess 退出程序
```

本次实验让我对 Windows 平台下汇编语言的实际应用有了更深入的理解，尤其是在解析 PE 文件结构方面获得了显著提升。通过对 PE 文件头中导出表、函数名表等关键数据结构的手动解析操作，我不仅从理论上掌握了它们在内存中的组织方式，更在实战中体会到了它们如何被程序动态解析和调用。这种“从内核出发”的方式，使我从底层视角重新认识了 API 函数调用背后的本质机制。

实验中最具启发性的部分，是通过访问 PEB（进程环境块）结构中的 LDR 链表，逐步解析模块加载信息，最终获得目标模块的基地址。借助对寄存器操作和栈结构的理解，我成功实现了不依赖导入表而直接解析 kernel32.dll 和 user32.dll 中关键 API 的行为，这不仅提升了我对系统调用机制的掌握，也加深了对 Windows 加载器原理的感性认知。

此外，我也更加清晰地理解了 kernel32.dll 与 user32.dll 模块的职能划分：两者虽然同属 Windows 的核心动态链接库，但 kernel32.dll 更多负责底层系统服务（如内存管理、进程控制、文件操作等），而 user32.dll 则提供了图形用户界面相关的 API（如窗口管理、消息框、键盘鼠标输入等）。在本实验中，正是通过 LoadLibraryA（位于 kernel32.dll）动态加载 user32.dll，才使得我们能够访问 MessageBoxA（位于 user32.dll）函数，形成典型的“系统服务模块→用户界面模块”层次调用流程，这一机制的掌握对后续理解 Windows 应用程序的执行框架有着极大的帮助。

最后，通过构造函数哈希值、在循环中遍历函数名表并与目标哈希进行匹配的过程，我不仅理解了哈希匹配的逻辑与算法，还掌握了如何借助该技术实现“API 隐写”或“脱导入表调用”。与此同时，整个实验过程中我也对内联汇编的语法规范、寄存器与栈帧的交互关系、函数调用的参数传递方式等有了更熟练的掌握，尤其对于pushad/popad、lodsd/stosd等不熟悉的汇编指令指令的具体作用上实现有了更加深入的理解。

```
PUSHAD :将EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI 这8个32位通用寄存器依次压入堆栈,其中SP的值是在此
条件指令未执行之前的值.压入堆栈之后,ESP-32->ESP.
POPAD :依次弹出堆栈中的32位字到 EDI,ESI,EBP,ESP,EBX,EDX,ECX,EAX中,弹出堆栈之后,ESP+32->ESP.
LODSB :将esi指向的地址处的数据取出来赋给AL寄存器， esi=esi+1;
LODSW :则取得是一个字。
LODSD :取得是双字节， 即mov eax， [esi]， esi=esi+4;
STOSB :将AL寄存器的值取出来赋给edi所指向的地址处。mov [edi]， AL; edi=edi+1;
STOSW :取的是一个字。
STOSD :取得是双字节， mov [edi]， eax; edi=edi+4;
```

综上，本次实验不仅是一次汇编技巧的训练，更是一次深入理解操作系统内部机制的探索旅程。它拓宽了我的技术视野，为我后续在系统底层开发、逆向分析与安全研究等方向打下了坚实的基础。