《软件安全》实验报告

姓名：蒋枘言　学号：2313546　班级：信息安全班

一、实验名称

API 函数自搜索

二、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。

三、实验过程

接下来我们要编写通用型 Shellcode，动态搜索并调用 API 函数（如 MessageBoxA、LoadLibraryA、ExitProcess），避免硬编码函数地址，提升跨系统版本的兼容性。

（1）定位 kernel32.dll 基地址

原理：通过进程环境块（PEB）遍历模块链表，找到 kernel32.dll 的基地址。
实现步骤：

①通过段寄存器 FS:[0x30]获取 PEB 地址。
②从 PEB 偏移 0x0C 处找到 PEB_LDR_DATA 结构体。
③从 PEB_LDR_DATA 偏移 0x1C 处获取模块初始化链表 InInitializationOrderModuleList。
④遍历链表，第二个节点（首个为 ntdll.dll）即为 kernel32.dll，其偏移 0x08 处为基地址。

（2）解析 kernel32.dll 导出表

原理：基于 PE 文件格式，定位导出表以获取函数地址列表和名称列表。
实现步骤：

①从 kernel32.dll 基地址偏移 0x3C 处找到 PE 头。
②PE 头偏移 0x78 处为导出表地址（RVA），需转换为实际地址（基地址 + RVA）。
③导出表偏移 0x20 处为函数名称列表的 RVA，偏移 0x1C 处为函数地址列表的 RVA。

（3）哈希算法优化函数名匹配

目的：缩短 Shellcode 长度，避免直接存储函数名字符串。
实现方法：

①对函数名逐字符循环右移 7 位并累加，生成唯一哈希值。
②遍历导出函数名列表，计算每个名称的哈希值并与目标哈希比较。

关键哈希值：

MessageBoxA：0x1E380A6A
LoadLibraryA：0x0C917432

ExitProcess：0x4FD18963

（4）动态加载 user32.dll 并定位 MessageBoxA

步骤：

①调用 LoadLibraryA("user32.dll")加载 DLL。

②重复步骤（1）~（3），解析 user32.dll 的导出表，搜索 MessageBoxA 的地址。

（5）整合 Shellcode 并调用 API

功能实现：

①弹窗显示消息：调用 MessageBoxA(NULL, "westwest", "westwest", 0)。

②正常退出程序：调用 ExitProcess(0)。

根据以上步骤，得到完整的 API 函数自搜索代码如下：

```
#include <stdio.h>
#include <windows.h>

int main() {
    __asm {
        // ====== 初始化部分 ======
        CLD                 // 清空方向标志位 DF，确保字符串操作（如 lodsd）向高地址增长

        // == 压入目标 API 函数的哈希值（逆序压入，栈顶为 LoadLibraryA 的哈希） ==
        push    0x1E380A6A              // MessageBoxA 的哈希（user32.dll）
        push    0x4FD18963              // ExitProcess 的哈希（kernel32.dll）
        push    0x0C917432              // LoadLibraryA 的哈希（kernel32.dll）

        mov esi, esp                    // esi 指向栈顶（当前栈中存放哈希值的起始地址）
        lea edi, [esi - 0xc]        // edi 指向栈中预留的 12 字节空间（用于后续存储函数地址）

        // ====== 开辟栈空间（0x400 字节）用于临时操作 ======
        xor ebx, ebx                    // ebx 清零
        mov bh, 0x04                    // bh=0x04 → ebx=0x0400
        sub esp, ebx                    // esp -= 0x400（开辟栈空间）

        // ====== 压入"user32.dll"字符串到栈中 ======
        mov bx, 0x3233                  // ebx 低 16 位=0x3233（对应字符"32"）
        push ebx                        // 压入"32"（栈顶为"32"）
        push 0x72657375                 // 压入"user"（栈顶为"user32"）
        push esp                    // 压入字符串起始地址（esp 此时指向"user32"的地址）

        xor edx, edx                    // edx 清零（用于后续偏移计算）

        // ====== 定位 kernel32.dll 基地址 ======
        mov ebx, fs: [edx + 0x30]//从 FS 段寄存器获取 TEB 地址，TEB+0x30 处为 PEB 地址
        mov ecx, [ebx + 0xC]            // PEB+0xC → PEB_LDR_DATA 结构体地址
        mov ecx, [ecx + 0x1C]           // PEB_LDR_DATA+0x1C →
                                        // InInitializationOrderModuleList（模块链表头）
        mov ecx, [ecx]                  // 第一个节点是 ntdll.dll
        mov ebp, [ecx + 0x8] //第二个节点是 kernel32.dll，其基地址存放在节点偏移 0x8 处

    // ====== 主循环：遍历哈希列表，动态加载 DLL 并搜索函数 ======
```

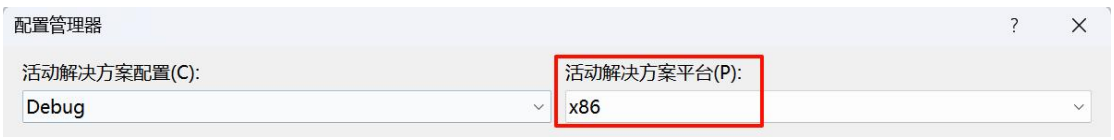
```
find_lib_functions :
    lodsd           // 从 esi 读取哈希值到 eax，esi+=4（首次读取 LoadLibraryA 的哈希）
    cmp eax, 0x1E380A6A             // 检查是否为 MessageBoxA 的哈希
                                    //（判断是否需要加载 user32.dll）
    jne find_functions              // 如果不是，直接搜索 kernel32.dll 中的函数

    // ======= 加载 user32.dll =======
    xchg eax, ebp               // 交换 eax（MessageBoxA 哈希）和 ebp（当前 DLL 基地址）
    call[edi - 0x8]             // 调用 LoadLibraryA（参数为栈中的"user32.dll"字符串地址）
    xchg eax, ebp               // ebp = user32.dll 基地址，eax 恢复为 MessageBoxA 哈希

// ======= 解析 DLL 导出表，搜索目标函数 =======
find_functions :
        pushad                          // 保存所有通用寄存器（用于后续恢复）

        // --- 定位导出表 ---
        mov eax, [ebp + 0x3C]       // PE 头偏移（NT 头 RVA）
        mov ecx, [ebp + eax + 0x78] // 导出表 RVA（PE 头+0x78 →
                                    // Export Directory RVA）
        add ecx, ebp                // 导出表实际地址（基地址 + RVA）
        mov ebx, [ecx + 0x20]       // 导出函数名列表 RVA（导出表+0x20 →
                                    // AddressOfNames）
        add ebx, ebp                // 函数名列表实际地址

        xor edi, edi                // edi 清零（用于函数名索引）

    // ======= 遍历函数名列表，计算哈希并匹配 =======
    next_function_loop :
        inc edi                     // 索引递增（从 1 开始）
        mov esi, [ebx + edi * 4]    // 获取第 edi 个函数名的 RVA（AddressOfNames 数组）
        add esi, ebp                // 转换为实际地址（基地址 + RVA）
        cdq                         // edx 清零（用于哈希计算）

        // --- 计算函数名哈希 ---
    hash_loop :
        movsx eax, byte ptr[esi]    // 读取函数名字符（符号扩展为 32 位）
        cmp al, ah                  // 检查字符是否为 0（字符串结尾）
        jz compare_hash             // 如果是，跳转到哈希比较
        ror edx, 7                  // 循环右移 7 位（哈希算法核心）
        add edx, eax                // 累加字符 ASCII 值
        inc esi                     // 指向下一个字符
        jmp hash_loop               // 继续循环

    // --- 比较哈希值 ---
    compare_hash :
        cmp edx, [esp + 0x1C]       // 比较计算出的哈希与目标哈希
                                    //（栈中保存的 pushad 前的 esi 值）
        jnz next_function_loop      // 不匹配则继续遍历

        // --- 找到函数后获取地址 ---
        mov ebx, [ecx + 0x24]       // 导出序号列表 RVA（AddressOfNameOrdinals）
        add ebx, ebp                // 序号列表实际地址
        mov di, [ebx + 2 * edi]     // 获取函数序号（序号 = 索引-1）
        mov ebx, [ecx + 0x1C]       // 导出地址表 RVA（AddressOfFunctions）
        add ebx, ebp                // 地址表实际地址
        add ebp, [ebx + 4 * edi]    // 获取函数 RVA 并转换为实际地址（基地址 + RVA）
        xchg eax, ebp               // eax = 函数实际地址，ebp 恢复为 DLL 基地址
```

```
            pop edi                         // 恢复 edi（指向函数地址存储位置）
            stosd                       // 将 eax（函数地址）存入 edi 指向的内存，edi += 4
            push edi                        // 保存 edi（更新存储位置）
            popad                           // 恢复所有通用寄存器

            // --- 检查是否完成所有函数搜索 ---
            cmp eax, 0x1e380a6a             // 是否已处理 MessageBoxA（最后一个哈希）
            jne find_lib_functions      // 未完成则继续循环

        // ====== 调用 API 函数 ======
        function_call :
            xor ebx, ebx                // ebx 清零（用于参数传递）
            push ebx                        // 压入 MessageBox 的 hWnd 参数（NULL）
            push 0x74736577                 // 压入字符串"west"（小端存储为"west"）
            push 0x74736577                 // 再次压入"west"（构成"westwest"）
            mov eax, esp                // eax 指向字符串地址
            push ebx                        // 压入 MessageBox 的 uType 参数（0）
            push eax                        // 压入标题字符串地址（"westwest"）
            push eax                        // 压入内容字符串地址（"westwest"）
            push ebx                        // 压入 hWnd 参数（NULL）
            call[edi - 0x04]            // 调用 MessageBoxA（地址存储在 edi-4）
            push ebx                        // 压入 ExitProcess 的退出码（0）
            call[edi - 0x08]            // 调用 ExitProcess（地址存储在 edi-8）

            // ====== 填充 nop 指令（用于对齐或调试） ======
            nop
            nop
            nop
            nop
    }
    return 0;
}
```

在 VC6 里运行，发现成功弹出了显示“west”的对话框。

将生成的 exe 文件复制到 Windows 10/11 系统里运行，发现也能成功弹出显示“west”的对话框。

注意：如果将以上代码直接复制到 Visual Studio 2022 里执行，需要先将“生成->配置管理器”里面的活动解决方案平台改为 x86，代码才能成功运行。这是由于 x86 架构（32 位）支持使用__asm 关键字进行内联汇编，而 x64 架构（64 位）不支持内联汇编。尝试使用__asm 会导致编译错误。



四、心得体会

1.动态定位 API 地址的核心逻辑

通过此次实验掌握了通过 PEB（进程环境块）遍历模块链表的底层方法，理解了 Windows 系统加载 DLL 的机制。

再次熟悉了 PE 文件格式，尤其是导出表（IMAGE_EXPORT_DIRECTORY）的结构与解析流程，包括函数名列表（AddressOfNames）、函数地址列表（AddressOfFunctions）的定位方法。

实践了哈希算法在 Shellcode 中的应用，通过计算函数名的哈希值替代硬编码字符串，大幅缩短代码体积，提升兼容性。

2.Shellcode 编写的通用性设计

学会了如何通过 LoadLibraryA 动态加载非基础 DLL（如 user32.dll），避免依赖目标程序预先加载特定模块。

理解了 kernel32.dll 和 ntdll.dll 在 Windows 程序中的基础性地位，即所有进程默认加载，是 Shellcode 的可靠切入点。