《软件安全》实验报告

姓名：王昱　　学号：2212046　班级：信息安全班

一、实验名称：

API 函数自搜索实验

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、实验原理：

前面的 Shellcode 的编写，都采用硬编址的方式来调用相应 API 函数。首先，获取所要使用函数的地址，然后将该地址写入 ShellCode，从而实现调用。如果系统的版本变了，很多函数的地址往往都会发生变化，那么调用肯定就会失败了。

在实际中为了编写通用 shellcode，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 函数自搜索技术。

四、实验过程：

一、步骤流程

总结将要用到的函数：

- MessageBoxA 位于 user32.dll 中，用于弹出消息框。
- ExitProcess 位于 kernel32.dll 中，用于正常退出程序。所有的 Win32 程序都会自动加载 ntdll.dll 以及 kernel32.dll 这两个最基础的动态链接库。
- LoadLibraryA 位于 kernel32.dll 中，并不是所有的程序都会装载 user32.dll，所以在调用 MessageBoxA 之前，应该先使用 LoadLibrary(“user32.dll”)装载 user32.dll。

进而我们可以按照如下逻辑编写 API 自搜索代码：

（一）定位 kernel32.dll

如果想要在 Win32 平台下定位 kernel32.dll 中的 API 地址，可以使用如下方法：

（1）首先通过段选择字 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。

（二）定位 kernel32.dll 的导出表

找到了 kernel32.dll，由于它也是属于 PE 文件，那么我们可以根据 PE 文件的结构特征，定位其导出表，进而定位导出函数列表信息，然后进行解析、遍历搜索，找到我们所需要的 API 函数。

定位导出表及函数名列表的步骤如下：

（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。

（三）搜索定位目标函数

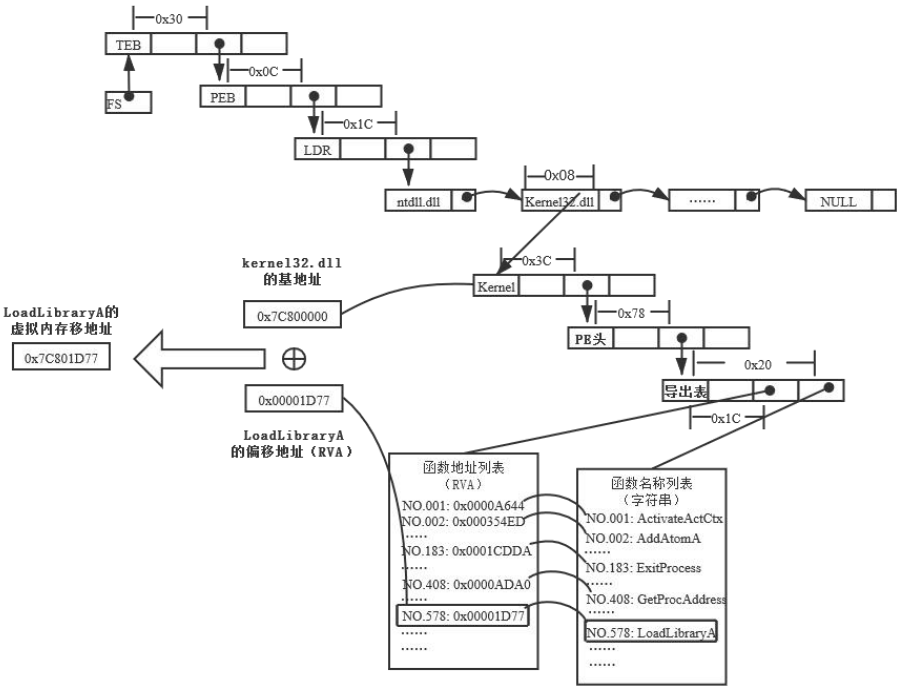
至此，可以通过遍历两个函数相关列表，算出所需函数的入口地址：

（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）获得 RVA 后，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。

按照这个方法，就可以获得 kernel32.dll 中的任意函数。

上述完整的几个步骤，可以用下图来概括。



二.汇编代码解析：

1.函数名入栈

```
__asm
{
    CLD
    push    0x1E380A6A
    push    0x4FD18963
    push    0x0C917432
    mov esi,esp
    lea  edi,[esi-0xc]
```

- 首先清空标志位 DF，三个 push 依次把 MessageBoxA、ExitProcess、LoadLibraryA 函数名字符串的哈希值入栈，利用哈希将后面函数名的比较变为哈希值的比较。
- mov 语句将 esp 赋值给 esi,esi 存放此时的栈顶位置，即 LoadLibraryA 的 hash 所在位置，之后 esi 不再变化，esi 专门用来找这三个函数名的哈希值。

2.开辟栈空间，压入”user32.dll”

```
//======开辟一些栈空间
xor     ebx,ebx
mov     bh,0x04
sub     esp,ebx
//======压入"user32.dll"
mov     bx,0x3233
push        ebx
push        0x72657375
push        esp
xor     edx,edx
```

- ebx 与自己做异或，相当于 ebx 置 0。bh 是 ebx 的低 16 位的高8 位，因此 bh 设为 0x04 后，ebx 变为 0x00000400，所以 sub 语句即为将 esp 减 0x400，为栈开辟新空间。
- 0x72657375 是字符串 “user” 的 ASCII 值，也将其入栈。
- 将此时的栈顶指针 esp入栈，其实就是将 “user32“ 字符串所在的地址入栈，再将 edx 置 0。

作用：

- 因为程序要完成对 MessageBoxA 函数的调用，而MessageBoxA 位于 user32.dll 中，但并不是所有的程序都会装载 user32.dll。LoadLibraryA 位于 kernel32.dll 中，所有的 Win32 程序都会自动加载 ntdll.dll 以及 kernel32.dll 这两个最基础的动态链接库，所以在调用 MessageBoxA 之前，应该先使用LoadLibrary(“user32”)装载 user32.dll。
- 当 Shellcode 调用MessageBoxA 函数时，需要先执行 LoadLibrary(“user32”)。而我们现在先把它的参数 ”user32“ 先入栈，使得当执行到此部分的时候，栈顶正好是 LoadLibrary 函数的参数区域。

3.找 kernel32.dll 的基地址

```
//======找kernel32.dll的基地址
mov     ebx,fs:[edx+0x30]   //[
mov     ecx,[ebx+0xC]       //[PE
mov     ecx,[ecx+0x1C]        //
mov     ecx,[ecx]           //进入
mov     ebp,[ecx+0x8]         //e
```

- 将 kernel32.dll 的基地址存入 ebx

4.是否找到了自己所需全部的函数

```
        //======[illegible]
find_lib_functions:
        lodsd     //即move eax,[
        cmp       eax,0x1E380A6A
        jne       find_functions
        xchg eax,ebp
        call      [edi-0x8]      /
        xchg      eax,ebp        //e
```

- loadsd：即 mov eax,[esi] 与 esi+=4。例如，在循环中第一次调用时，eax 得到的是 LoadLibraryA 的 hash。然后 esi 再往高地址增长4，即往栈底找下一个函数名的 hash。
- cmp：将 eax 遍历得到的函数名哈希值与 MessageBoxA 的 hash 比较。

若此时 eax 存的不是 MessageBoxA 的 hash，则跳转到 find_functions。当经历三次 hash 值的读取之后，eax = MessageBoxA 的 hash：

- 此时 call [edi-0x8]，会调用 LoadLibraryA("user32") 函数，呼应了上文提到的参数 ”user32“ 入栈。
- 此后 ebp 不再是 kernel32.dll 的基地址，而是 user32.dll 的基地址。下面的 find_functions、
- next_function_loop 等循环就变成了在 user32 的导出表里找 MessageBoxA 函数。

```
        //------导出函数名列表指针
find_functions:
        pushad
        mov       eax,[ebp+0x3C]
        mov       ecx,[ebp+eax+0x78]
        add       ecx,ebp
        mov       ebx,[ecx+0x20]
        add       ebx,ebp
        xor       edi,edi
```

第一次跳转到这里的时候，ebp 存的是 kernel32.dll 的基地址，这段代码定位 kernel32.dll 的 PE 头、导出 表、导出函数名列表等。此时，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]
        add       esi,ebp
        cdq

        //------函数名的hash运算
hash_loop:
        movsx     eax,byte ptr[esi]
        cmp       al,ah
        jz        compare_hash
        ror       edx,7
        add       edx,eax
        inc       esi
        jmp       hash_loop
```

由 find_function 找到函数名列表（ebx：导出函数名列表指针的基地址）之后，开始遍历函数名列表，分别对遍历到的函数名计算哈希值（hash_loop 循环部分是用于计算函数名

hash 的）。当计算出此函数名的 hash 之后，会由 jz compare_hash 跳转到下面的 compare_hash 部分。此时该函数的 hash 在 edx 中保存。

```
        //------比较找到的当前函数的
compare_hash:
        cmp     edx,[esp+0x1C]     /
        jnz     next_function_loop
        mov     ebx,[ecx+0x24]     /
        add     ebx,ebp            /
        mov     di,[ebx+2*edi]     /
        mov     ebx,[ecx+0x1C]     /
        add     ebx,ebp            /
        add     ebp,[ebx+4*edi]    /
        xchg    eax,ebp            /

        pop     edi
        stosd                      /
        push    edi

        popad
        cmp     eax,0x1e380a6a     /
        jne     find_lib_functions
```

- 第一个 cmp 语句比较 edx 与 [esp+0x1C]，即比较当前函数的 hash 是否是自己想找的。
- 根据 jnz 语句，若当前函数不是想找的函数，则跳转到上面的 next_function_loop，继续遍历、计算、比较下一个函数，直到在该 dll 的导出函数名列表中找到了想找的函数。

当我们在 dll 的导出函数名列表中找到所需函数后，由顺序表、地址表的相对偏移量、基地址等计算它的虚拟地址。

- popad 与 pushad 对应，整个保存所有寄存器状态。
- cmp 比较当前找到的函数名 hash 是不是 MessageBoxA 的 hash。
- jne 语句说明，若当前找到的不是 MessageBoxA 函数，则跳回 find_lib_functions。
- 若 find_lib_functions 部分的 eax 读取到了 MessageBoxA 的 hash，并在 user32.dll 中找到了函数 MessageBox 后，跳出循环

5. 执行Shellcode：

```
        //------比较找到的当前函数的
compare_hash:
        cmp     edx,[esp+0x1C]     /
        jnz     next_function_loop
        mov     ebx,[ecx+0x24]     /
        add     ebx,ebp            /
        mov     di,[ebx+2*edi]     /
        mov     ebx,[ecx+0x1C]     /
        add     ebx,ebp            /
        add     ebp,[ebx+4*edi]    /
        xchg    eax,ebp            /

        pop     edi
        stosd                      /
        push    edi

        popad
        cmp     eax,0x1e380a6a     /
        jne     find_lib_functions
```

- xor ebx, ebx 和 push ebx 这两行将寄存器 EBX 清零，并将其值（0）推入堆栈。在 Windows API 中，这个值通常用作函数参数，表示空指针或者无特定窗口。
- push 0x74736577 将字符串"west"的 ASCII 码值以十六进制形式推入堆栈两次，这样堆栈

上就有了"westwest"这个字符串。

- mov eax, esp 将当前堆栈指针（即"westwest"字符串的地址）移动到 EAX 寄存器中。
- push eax 将 EAX 寄存器的值（也就是"westwest"字符串的地址）推入堆栈两次，一次用作 MessageBox 的文本内容（lpText），一次用作标题栏内容（lpCaption）。
- call [edi-0x04] 调用 MessageBoxA 函数，显示消息框。其中 edi-0x04 应该是指向 MessageBoxA 函数的指针。
- push ebx 和 call [edi-0x08] 这两行调用 ExitProcess 函数，结束程序运行。其中 edi-0x08 应该是指向 ExitProcess 函数的指针。

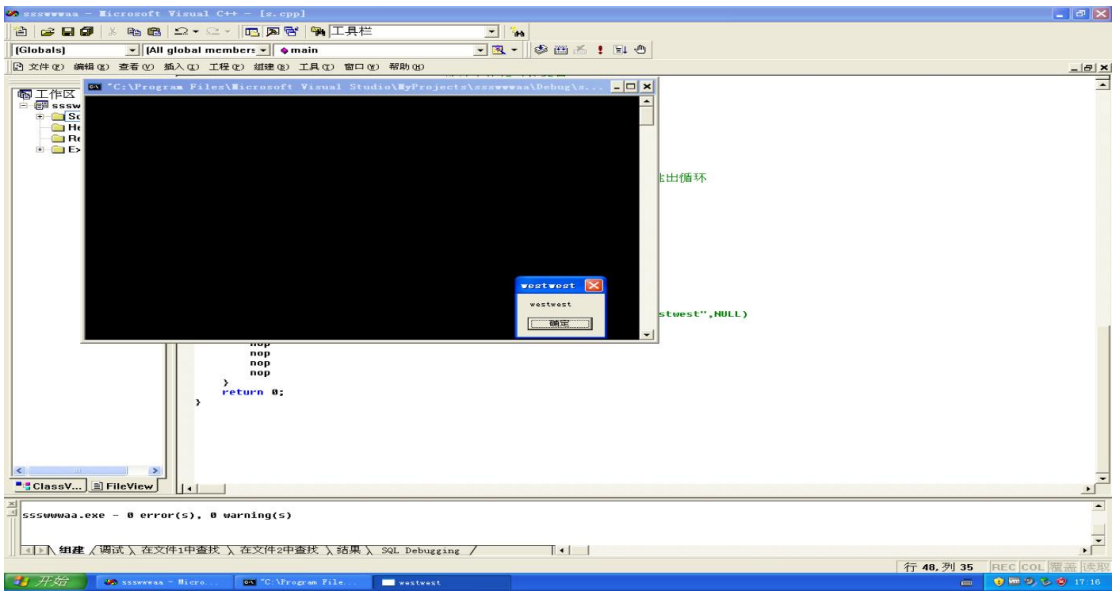
6.总结分析：

整个实验的流程就是通过手动定位函数的地址，将函数装载到程序中，然后将相关的参数入栈，最后通过 call 指令调用函数，将我们想要输出的字符串通过消息框的方式输出。

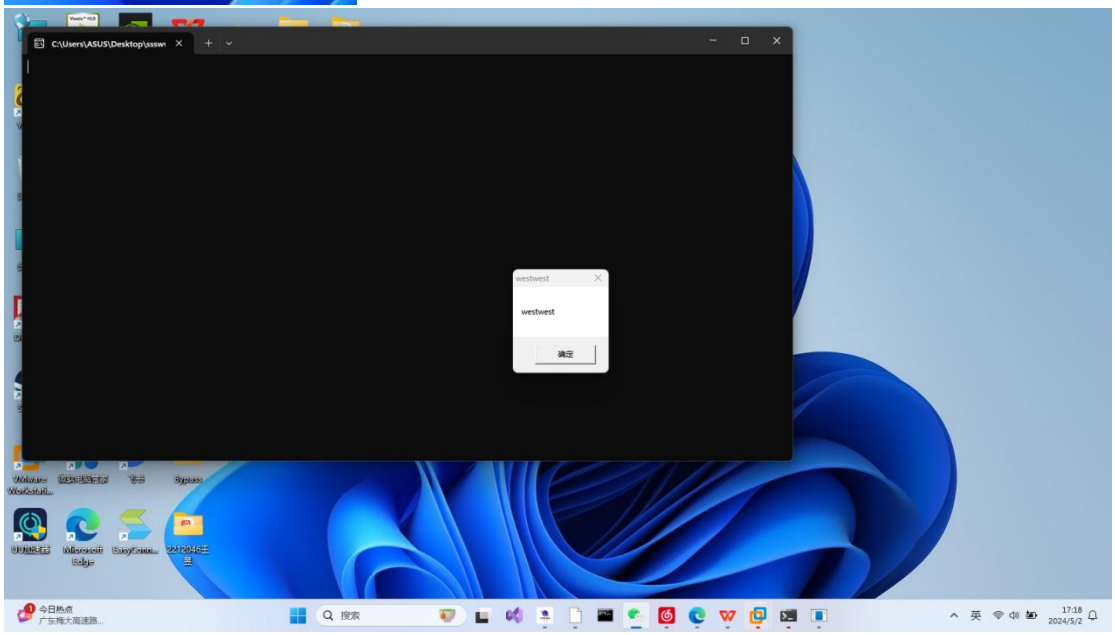
我们首先需要定位到 MessageBoxA、ExitProcess 和 LoadLibraryA 这三个函数的地址。可以通过 TEB 找到 PEB 的位置，通过 PEB 中定位 ntdll.dll 的地址，在 ntdll 中去寻找我们引入的动态链接库的地址。以 Kernel32.dll 为例，在定位到 Kernel32.dll 库的基地址之后，要先保存其基地址，方便后续根据函数的偏移地址得到函数的虚拟地址。由于.DLL 文件也属于 PE 文件结构，因此可以通过 Kernel32 的 PE 头定位到其导出表，通过导出表根据函数名的 hash 去匹配函数，再到函数地址列表中找到对应的函数相对虚拟地址。根据这个相对虚拟地址和之前得到的 DLL 文件的基地址通过运算可以得到函数的虚拟地址。装载 MessageBoxA、ExitProcess 和 LoadLibraryA 这三个函数的思路和方法基本相同。

至此，自搜索找到了当前系统下三个函数的入口地址。退出循环，执行想要做的事。此时，edi 所保存的地址区域，是按照前面依次入栈的顺序，即 MessageBox、ExitProcess 和 Loadlibrary 的入口地址，在这里调用[edi-0x04] 和 [edi-0x08] 即可。

7..在 XP 系统下验证程序是否能够正常运行：



8.把在虚拟机上生成的 exe 程序复制到 Windows 10 下运行：



五、心得体会：

1.深入理解 PEB（进程环境块）：PEB 是一个数据结构，包含了进程相关的信息。在本 实验中，我们学会了如何利用 PEB 来查找加载到进程中的 DLL 模块列表和获取内核基址。这有助于我们更好地理解 Windows 操作系统的底层机制。

2.掌握了 API 函数自搜索技术：在不使用导入表的情况下，我们学会了如何查找并调 用指定的 API 函数。这种技术可以应用于动态加载库等场景。同时，了解这种技术也有助于 我们识别和分析恶意软件、病毒和渗透测试工具中的类似方法。

3.在这次实验中，我了解了如何动态的去自动定位函数 API，了解了 API 函数自搜索技术，结合之前所学的 PE 文件结构对 API 函数子搜索技术有了更深一步的了解。通过 API 函数自搜索的手段，可以在一定程度上提高我们编写的 shellcode 的跨平台性和可以执行。

4.提高了汇编语言能力。