南開大學
Nankai University

计算机学院

软件安全实验报告

实验六：API 函数自搜索

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 4 月 20 日

目录

1 实验名称

API 函数自搜索

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 windows 10 操作系统里验证是否成功。

```
首先，总结一下我们将要用到的函数：
· 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，介绍通用型 shellcode 编写的步骤：
（1）第一步：定位 kernel32.dll。
（2）第二步：定位 kernel32.dll 的导出表。
（3）第三步：搜索定位 LoadLibrary 等目标函数。
（4）第四步：基于找到的函数地址，完成 Shellcode 的编写。
```

3 实验过程

3.1 定位 kernel32.dll

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

这段代码的含义是，首先通过段选择字 FS 在内存中找到当前的线程环境块 TEB，通过对其进行偏移 0x30，从而找到了指向进程环境块 PEB 的指针，并将其保存在 ebx 中，再对其进行 0xC 的偏移，找到了指向 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，再偏移 0x1C，找到了存放着指向模块初始化链表的头指针 (InInitializationOrderModuleList)，而在这个链表中，第一个就是 ntdll.dll，第二个就是 kernel32.dll，对其偏移 8 位即可得到 kernel32.dll 的基地址，并将其保存在 ebp 寄存器中。

3.2 定位 kernel32.dll 的导出表

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

这段代码的含义是，由于我们之前把 kernel32.dll 的基地址保存在 ebp 寄存器中，ebp+0x3C 指向的就是 kernel32.dll 偏移 0x3C 字节的地址，也就是 PE 头的指针，而 PE 头偏移 0x78 处，存放着导出表的指针，将相对偏移地址保存在 ecx 寄存器中，通过将 kernel32.dll 的基地址与相对偏移地址相加，即可得到导出表的基地址，导出表偏移 0x20 处的指针指向存储导出函数函数名的列表，ebx 与 ebp 相加，即可得到导出函数名列表指针的基地址，并保存在 ebx 寄存器中。从而完成了对导出表的定位。

3.3 搜索定位目标函数

在这一过程中，我们的目的是通过比较函数名对应的 hash 值，来获取我们需要的函数。

```
#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);
}
```

这段程序帮助我们获取了 MessageboxA、ExitProcess、LoadLibraryA 这三个函数名的 hash 值。

```
    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
```

以上代码把三个函数的 hash 值依次压入栈。

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

这一段把 user32.dll 的地址压入栈中。在之后需要使用 LoadLibraryA 将其加载到程序中，从而获得 MessageBoxA 函数。

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

这一段我们在上面说明过，就不再赘述。

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

```
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
```

这段代码的目的是遍历函数列表来找到我们需要的函数地址。eax 寄存器保存了我们需要的函数的 hash 值，如果不是 MessageBoxA，那么我们就利用 kernel32.dll，依次去找到 LoadLibraryA 和 ExitProcess；如果是 MessageBoxA，那么就会先 LoadLibraryA("user32")，并利用 user32.dll 去找到 MessageBoxA。find_lib_functions 会调用 find_functions，找到 kernel32.dll 或 user32.dll 的导出表的函数列表地址，next_function_loop 会找到下一个函数名。再之后通过比较 hash 值，判断取出的函数名是不是我们需要的函数，比较 hash 值的代码如下：

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

hash_loop 获得函数列表中函数的 hash 值，如果获得了一个函数名的 hash 值就调用 compare_hash 函数，比较从函数名列表中取出的函数 hash 值是否与我们需要的函数的 hash 值相等，如果相等的话，就把其地址保存在 edi 寄存器中。如果找到了最后一个函数 MessageBoxA 就跳出循环。至此，我们找到了 MessageboxA、ExitProcess、LoadLibraryA 三个函数的地址。

3.4 通用型 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
```

[edi-0x04], 指向的就是 MessageBoxA 函数的起始地址，通过上述代码，即可完成通用型 Shellcode 的编写，测试结果如下：

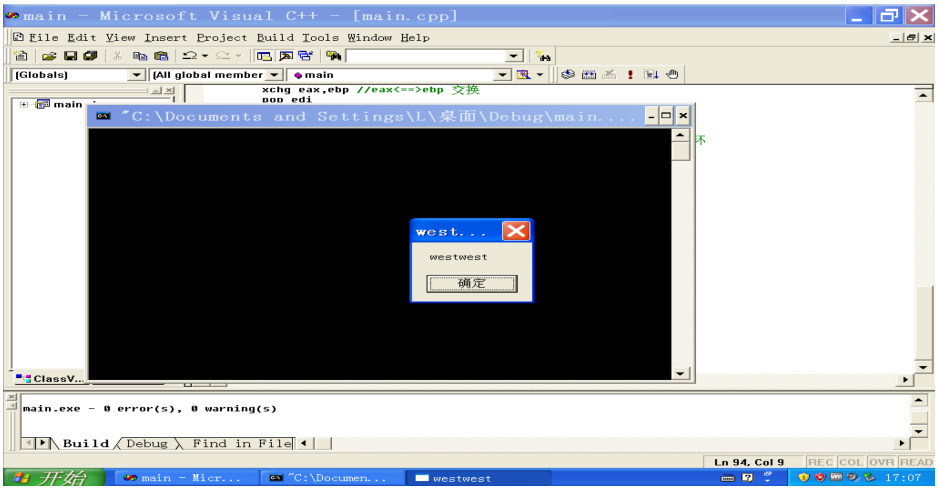

图 3.1: 测试结果

可以看到成功弹出窗口 westwest。

3.5 win11 验证

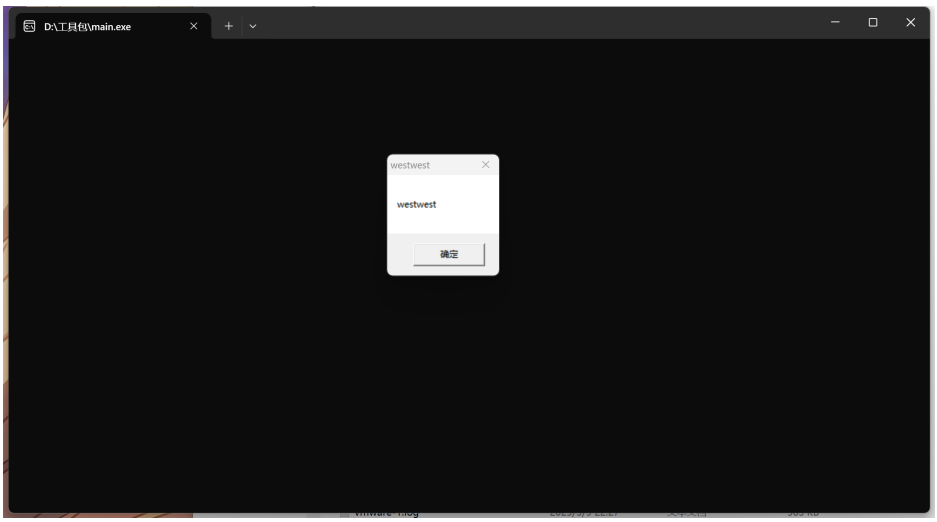

图 3.2: win11 验证

可以看到成功弹出窗口 westwest，说明了我们编写的 shellcode 的通用性。

4 心得体会

1. 通过此实验，掌握了 API 函数的自搜索技术，学会了 dll、导出表、函数列表的定位。

2. 对 PE 文件的结构有了更好的理解。

3. 加强了汇编代码的读写能力。

4. 学会了如何编写通用型 shellcode，重点在于如何获取 MessageboxA、ExitProcess、LoadLibraryA 这三个函数的地址。