

《漏洞利用及渗透测试基础》实验报告

姓名：张洋 学号：2111460 班级：信安二班

实验名称：

API 函数自搜索实验

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。

具体操作流程如下：

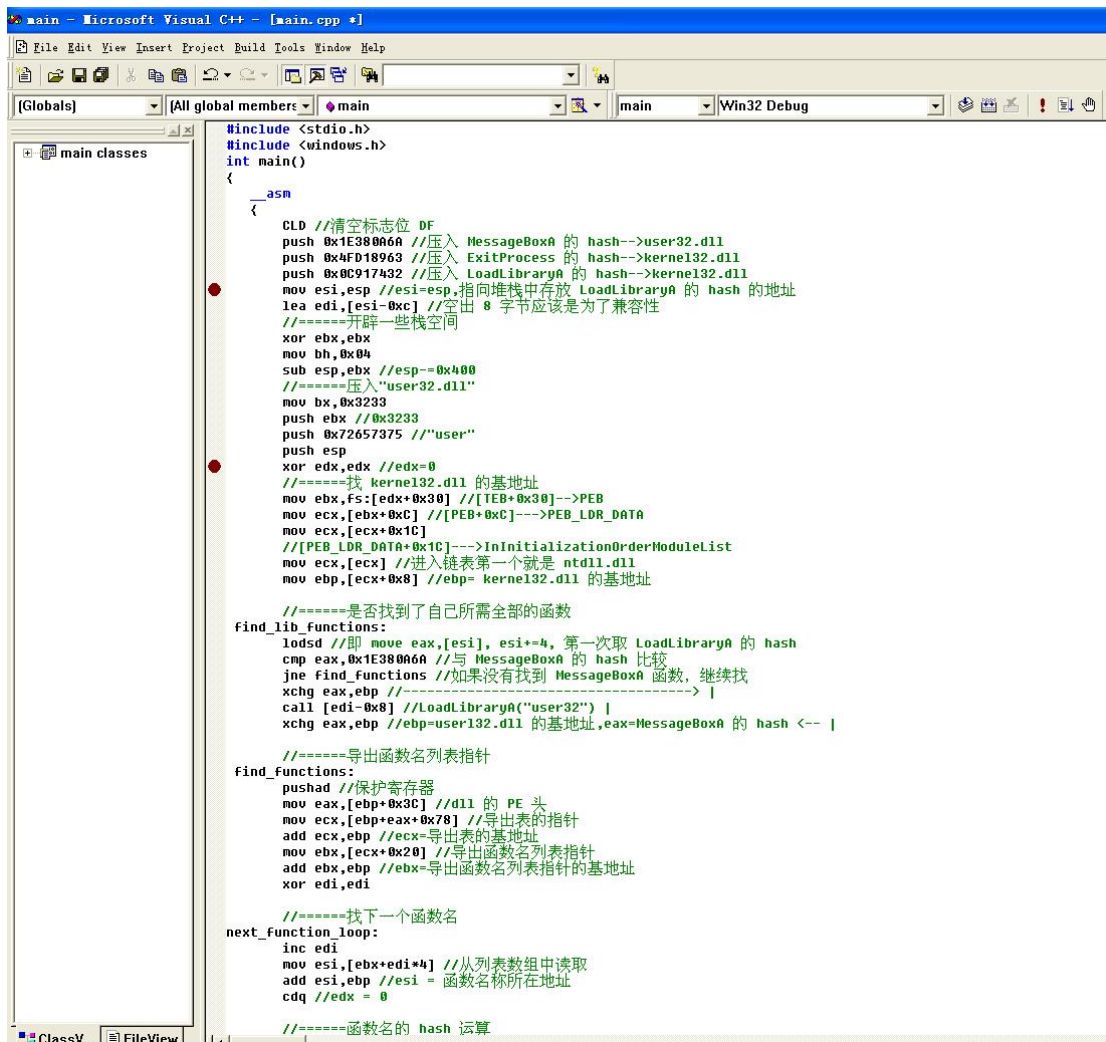
1. 定位 kernel32.dll:
 - (1) 通过段选择字 FS 在内存中找到当前的线程环境块 TEB。
 - (2) 线程环境块偏移地址为 0x30 的地址存放着指向进程环境块 PEB 的指针。
 - (3) 进程环境块中偏移地址为 0x0c 的地方存放着指向 PEB_LDR_DATA 结构体的指针，其中，存放着已经被进程装载的动态链接库的信息。
 - (4) PEB_LDR_DATA 结构体偏移位置为 0x1c 的地址存放着指向模块初始化链表的头指针 InInitializationOrderModuleList。
 - (5) 模块初始化链表 InInitializationOrderModuleList 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 kernel32.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。
4. 完整 API 函数自搜索代码

为了让 shellcode 更加通用，能被大多数缓冲区容纳，总是希望 shellcode 尽可能短。因此，一般情况下并不会“MessageBoxA”等这么长

的字符串去进行直接比较。所以会对所需的 API 函数名进行 hash 运算，这样只要比较 hash 所得的摘要就能判定是不是我们所需的 API 了。

实验过程：

1. 打开 VC6，写入代码



```
#include <stdio.h>
#include <windows.h>
int main()
{
    _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 字节应该也是为了兼容性
        //=====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

        //=====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-0xc] //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

        //=====函数名的 hash 运算
```

2. 逐行解释 API 函数自搜索代码：

```
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
```

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。

EAX = CCCCCCCC	EBX = 7FFD8000
ECX = 00000000	EDX = 00430DB0
ESI = 0012FF28	EDI = 0012FF80
EIP = 0040103A	ESP = 0012FF28

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 字节应该也是为了兼容性
```

EDI 指向未压入三个函数名字哈希值之前的栈顶位置，即 0012FF1C

```
//=====开辟一些栈空间
xor ebx,ebx
mov bh,0x04
sub esp,ebx //esp-=0x400
```

xor ebx, ebx 实现 ebx 清零，清零后给 ebx 赋值为 00000400，再将栈顶抬高 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”的哈希值存放在 ebx 中，bx 是 ebx 的低十六位，故 ebx 值为 0x00003233，再将“user”的字符串转哈希值压入栈。

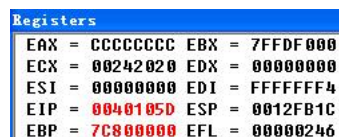


Address:	0012FB20
0012FB20	75 73 65 72 33 32 00 00

此时 esp 处则存下了 user32 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]
//[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
```

把 kernel32 的基地址存入 ebp



Registers			
EAX	=	CCCCCC	EBX = 7FFDF000
ECX	=	00242020	EDX = 00000000
ESI	=	00000000	EDI = FFFFFFFF
EIP	=	0040105D	ESP = 0012FB1C
EBP	=	7C800000	EFL = 00000246

EBP 的值为 7C800000。

进入 find_lib_functions 函数：

```
lodsd //即 move 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 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

```

eax 与 MessageBox 的哈希值比较，如果没有找到 MessageBoxA 函数则需要继续寻找，跳转到 find_functions 函数处完成后续操作。如果找到的话则将 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。

```

//=====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

```

eax 存储 dll 的 PE 头。定位到导出表，ecx 存储导出表的基地址。再定位到导出函数名列表，ebx 存储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] //从列表数组中读取
    add esi, ebp //esi = 函数名称所在地址
    cdq //edx = 0

```

从列表中读取函数名，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

```

hash_loop 函数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] //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

```

Compare_hash 函数用于比较当前函数的 hash 是否是想要找的，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。

如果找到了想要的函数名，计算出虚拟地址，虚拟地址等于相对偏移量加基地址。

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。

一次性完成多个寄存器状态保存和恢复，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。

找到 MessageBox 函数后，通过上述语句计算出虚拟地址，将地址存放入 edi 中，此时 edi 中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
```

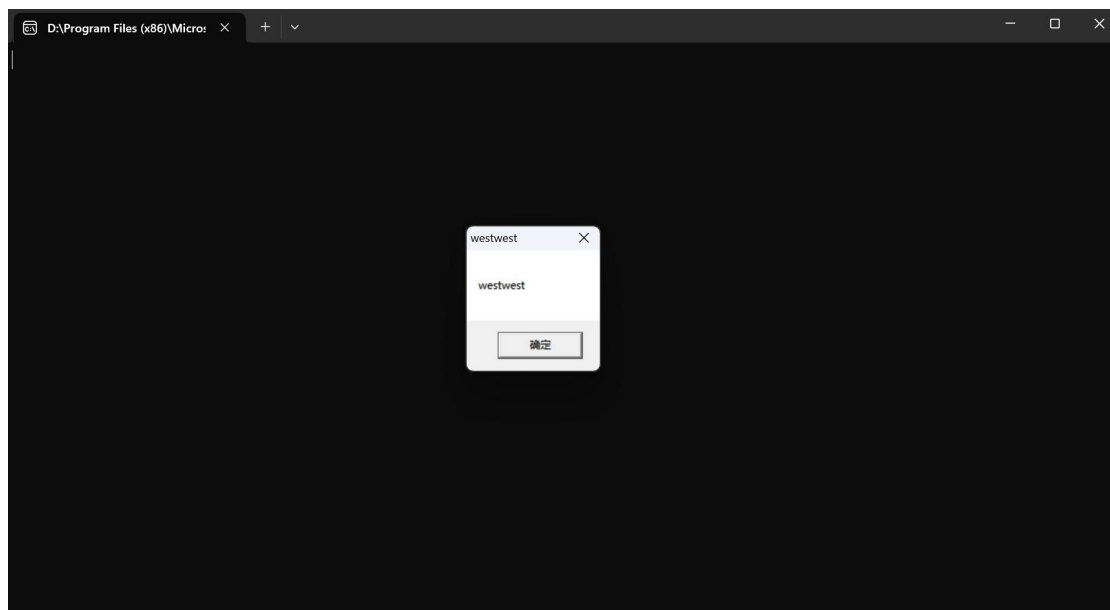
在 `Function_call` 函数中进行 shellcode 编写，将 `ebx` 清零后压入栈。将“westwest”对应的 `ascii` 码压入栈中，`esp` (`westwest` 字符串的地址) 赋值给 `eax`，压入 `ebx`, `eax`, `eax`, `ebx`，调用 `MessageBoxA` 函数，进行调用时会自动调取栈中前四个函数，故对应了 `null`, `westwest`, `westwest`, `null`。

最后之后再次压入 `ebx` (0)，调用 `exitprocess` 函数，退出程序。

调用后结果为：



Windows 11 调用后结果为：



心得体会：

此次实验通过调用 `messagebox` 函数实现弹出对话框，可以帮助我们更好地理解和掌握 Windows API 的使用方法。在实验过程中，我不仅学会了如何调用 API 函数，还学会了如何根据实际需求进行参数的自定义和调整。这些知识可以在日常工作中帮助我更好地完成相关任务和项目。具体的操作流程可以做如下总结：定位到动态链接库，找到动态链接库的导出表，定位到目标函数，完成实验内容。

总的来说，通过完成 API 函数的自搜索实验，我深入了解了 Windows API 的使用方法，同时也加深了对计算机系统和操作系统的理解。希望未来能够继续探索 API 函数的更多应用和技巧。