《软件安全》实验报告

姓名： 2313211 学号：2313211

实验名称：

API函数自搜索

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。

实验过程：

本次实验，我们需要编写通用的shellcode代码，使其能够在不同的系统中都能运行，让我们的shellcode代码具有动态API函数地址自搜索的能力。本次实验，我们通过复现书本上的例子，输出弹窗“westwest”，来介绍如何一步一步实现API函数自搜索的具体步骤。

(1)编写逻辑

我们首先需要完成shellcode通用代码的编写，理清其中的逻辑。

1.`MessageBoxA` 位于 `user32.dll` 中，用于弹出消息框。

2.`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。

3.`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` 。

因此，我们的总体步骤分为以下四个：

第一步：定位`kernel32.dll`。

第二步：定位 `kernel32.dll` 的导出表。

第三步：搜索定位`LoadLibrary` 等目标函数。

第四步：基于找到的函数地址，完成 `Shellcode` 的编写。

(2)具体流程

1.定位kernel32.dll

代码如下所示：

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

首先，我们将 `user32.dll` 的地址压入栈，将 `edx` 的值赋值为0，然后再去寻找 `kernel32.dll` 的基 地址。通过 `fs` 段寄存器定位到当前的线程块 `TEB` ，通过对其偏移 `0x30` ，指向进行环境块 `PEB` 的指针， 保存在`ebx`寄存器中， `PEB` 再偏移`0x0C` ，地址处存放了 `PEB_LDR_DATA` 的结构体指针；`PEB_LDR_DATA`结构体偏移 `0x1C` 的地址处存放了模块初始化链表头指针（`InInitializationOrderModuleList`），进 入这个链表，第一个结点就是我们的 `ntdll.dll`，再偏移8位，链表中的第二个位置就是我们要找的 `kernel32.dll`，上面就是我们的定位 `kernel32.dll`的过程。

2.定位kernel32.dll的导出表

代码如下所示：

```
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
```

`kernel32.dll` 是一个PE文件，所以我们可以通过其结构特征去定位它的导出表，进而定位导出的函数列表信息，遍历搜索出我们需要的API函数。

首先，我们将 `ebp` 寄存器的地址偏移 `0x3C` 位，所指向的地方就是PE头指针；PE头偏移 `0x78` 处，存 放着导出表的指针，将导出表的指针地址加上 `ebp` 寄存器，就获得了导出表的基地址。我们将导出表的 基地址偏移 `0x20` ，指向导出函数名的列表指针，最后还是通过加上 `ebp` 基地址，去获得函数名列表的基 地址。后续我们只需要通过一一比对，就可以通过 `hash` 值来找到我们需要的函数。

3.搜索定位LoadLibrary等目标函数

在得到函数名列表的基地址后，为了找到我们所需要的具体函数，我们还需要通过 hash 值的搜索， 去找到我们所需要的函数，因为通过函数名去对比不是特别方便，所以我们通过函数的 hash 值去寻找。 代码如下所示：

```
#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);
}
```

这部分代码实现了对函数名的 `hash` 值的计算，帮助我们获取到了 `MessageboxA`、`ExitProcess`、 `LoadLibraryA` 的函数 `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
```

然后再程序的开头，我们首先将这三个函数的 `hash` 值入栈，然后我们通过前面提到的 `find_lib_functions`、`find_functions` 、`next_function_loop` 这三个函数来进行循环，从而找到我们所需要的函数地址。

```
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
```

可以看到，第一个函数`find_lib_functions`调用了后面第二个函数`find_functions`来完成寻找 函数的功能；第三个函数的作用是，如果不符合`hash`值的要求，那么就继续往后遍历来进行寻找。我们 通过比较`hash`值来判断是否需要跳出循环，找到一样的`hash`值后，我们就跳出循环， `hash`循环和`hash`比较的代码如下所示：

```
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`函数完成了对函数`hash`值查找的循环；而`compare_hash`则完成了对于函数`hash`值的 比较。最后，通过以上的这些函数，我们成功找到了三个函数的地址，通过`edi`保存。之后，我们就可 以用`edi`寄存器来进行访问了。

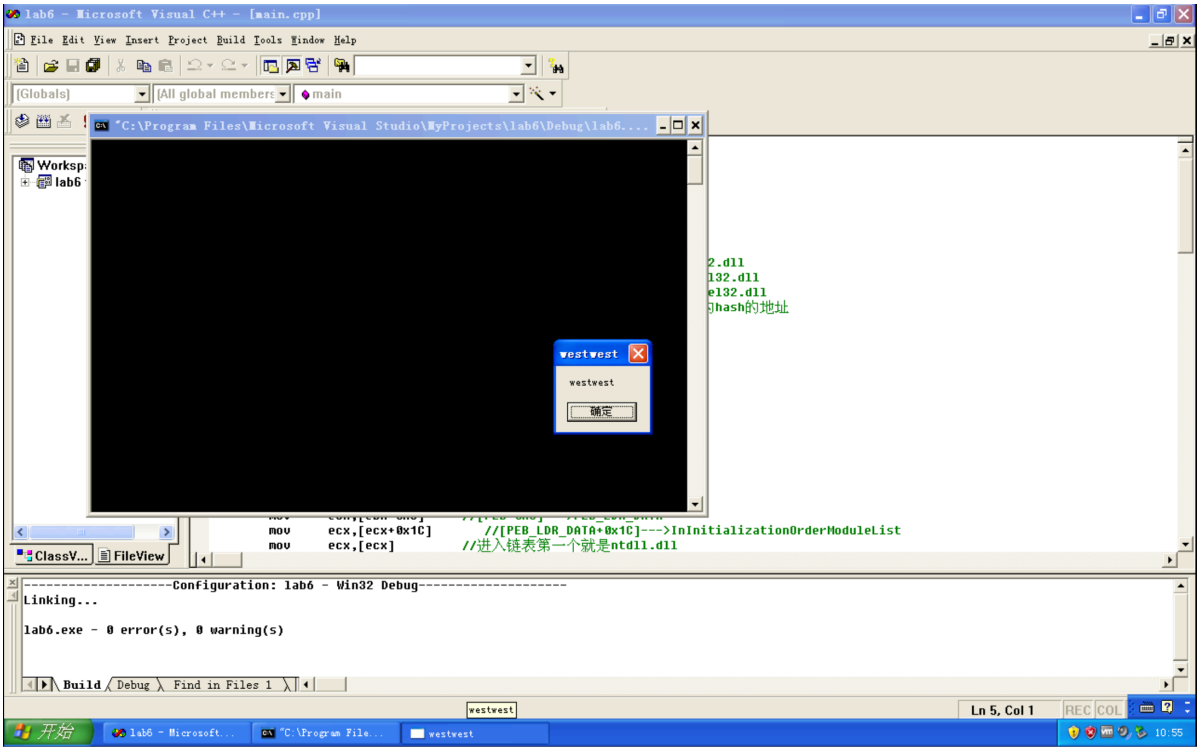
4.基于找到的函数地址，完成shellcode代码的编写

根据源代码，我们本次需要输出的就是“westwest”，因此，我们编写以下的shellcode代码：

```
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;
}
```

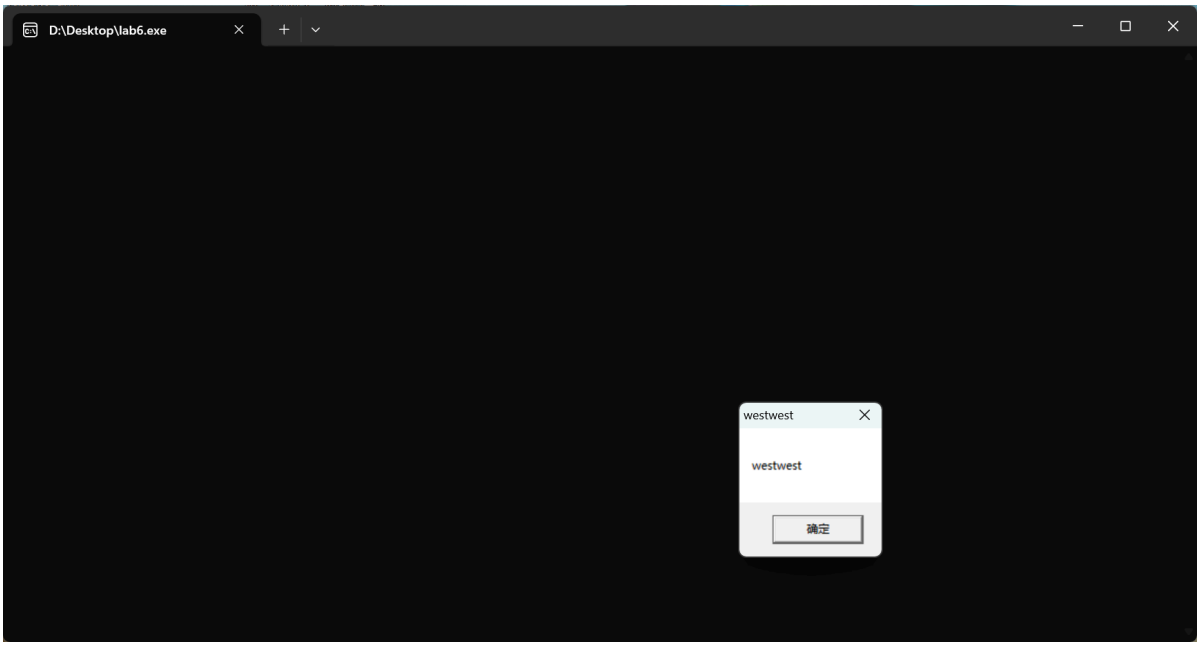
我们综合上面的所有代码，然后在VC6.0运行，得到结果如下所示：



说明我们成功找到了 `Messagebox` 函数并且运行了shellcode代码。

(3)在win11系统运行，验证API自搜索性

为了验证API函数的可移植性，我们将vmware中生成的 exe 文件移植到自己的win11系统下（电脑是win11）查看是否能够运行，结果如下：



我们发现，在`win11`系统下， `exe` 文件仍然可以运行，证明了我们编写的`shellcode`代码是通用的， 在不同的系统上都能实现API函数的自搜索。

心得体会：

通过本次实验，我掌握了API函数的子搜索技术，学会了在不适用导入表的情况下，根据 `TEB` 、`PEB` 等逐步通过偏移定位到导出表，然后再利用哈希值对比找到我们需要的API函数。

提高了自己对于汇编语言代码的理解能力。本次实验采用了内联汇编，在C语言代码中嵌入了很多 `asm` 汇编代码，在实验过程中，我对一些寄存器的使用了解更加深了，可以自己调试汇编语言代 码，完成我们需要完成的任务

我了解到了shellcode的通用性要求我们不能依赖固定的API地址，而是要通过遍历PEB、LDR等结构，动态查找所需DLL和API函数的地址。这一过程加深了我对Windows底层结构的理解，也让我认识到操作系统兼容性和安全机制对代码设计的影响。

实验中通过hash函数名的方式定位API，避免了直接存储字符串，提高了shellcode的隐蔽性和移植性。这种技巧在实际攻防和安全研究中非常实用。整个实验锻炼了我的汇编编程能力和调试技巧，也让我体会到shellcode开发的复杂性和挑战性。只有深入理解系统底层原理，才能写出高效、稳定且通用的shellcode。