《软件安全》实验报告

姓名：许洋　学号：2313721 班级：1070

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：

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，即让我们的shellcode代码具有动态API函数地址自搜索的能力。

(1)编写逻辑

在实现通用Shellcode的过程中，我们需要确保代码能够在不同系统中运行，并具备动态搜索API函数地址的能力，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，并非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。

基于上述分析，我们的实现步骤分为以下四个部分：

定位kernel32.dll。

定位kernel32.dll 的导出表。

搜索并定位目标函数（如LoadLibraryA）。

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

以下是定位kernel32.dll的代码：

```
//======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，获取指向进程环境块（PEB）的指针，将其存储在ebx寄存器中，PEB再偏移0x0C，获取指向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

接下来，我们需要定位kernel32.dll的导出表，以获取其导出函数列表。代码如下：

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

由于kernel32.dll是一个 PE 文件，我们可以通过其结构特征定位导出表，并进一步获取导出函数列表信息，从而遍历搜索所需的 API 函数。

首先，从寄存器ebp中存储的基地址出发，偏移0x3C位置可找到 PE 文件头指针。在 PE 文件头偏移0x78处存放了导出表的指针，将该指针与ebp基地址相加即可得到导出表的实际基地址。接着，从导出表基地址偏移0x20的位置可以获取导出函数名列表的指针，再将其与ebp基地址相加，即可获得函数名列表的实际基地址。最后，通过逐一比对函数名的哈希值，即可定位到所需的 API 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，为了找到目标函数，我们使用哈希值进行匹配，而不是直接比较函数名。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);
}
```

通过上述代码，我们可以计算出MessageBoxA、ExitProcess和LoadLibraryA的哈希值，并将其压入栈中：

```
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，通过循环遍历导出函数名列表，计算每个函数名的哈希值并与目标哈希值进行比较，找到匹配的函数地址。

```
//======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-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：

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

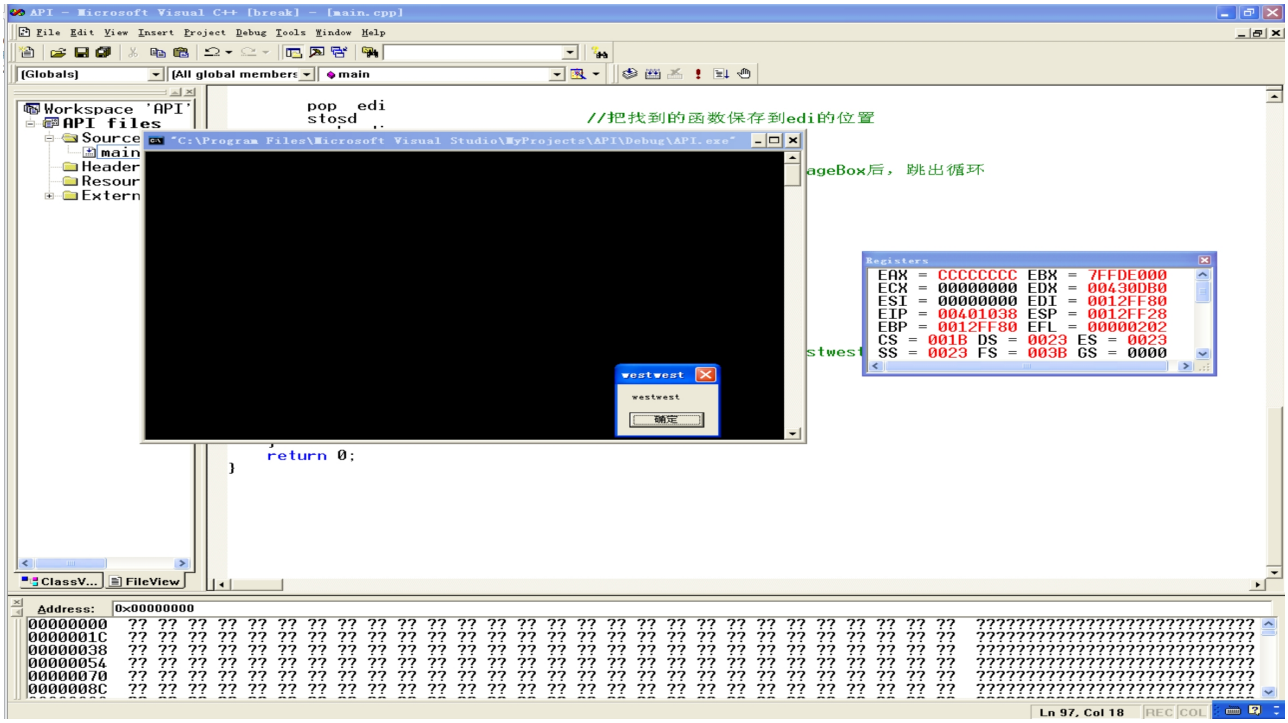
hash_loop函数完成了对函数hash值查找的循环；而compare_hash则完成了对于函数hash值的比较。最后，通过以上的这些函数，我们成功找到了三个函数的地址，通过edi保存。之后，我们就可以用edi寄存器来进行访问了。

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
```

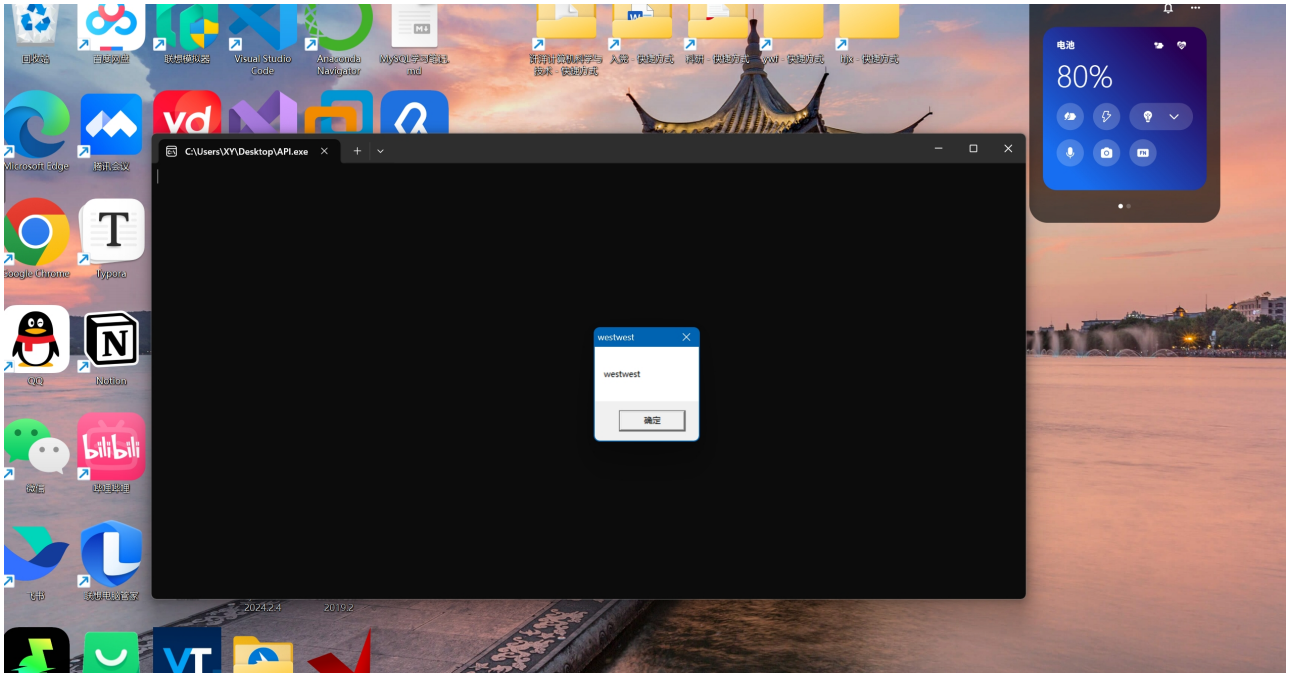
我们综合上面的所有代码，然后在VC6.0运行，得到结果如下所示：



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

(3)在win10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，我对 Shellcode 的编写以及动态 API 函数地址自搜索的实现有了更深刻的理解。以下是我的几点心得体会：

1. 深入理解 PE 文件结构的重要性
   在实验中，定位kernel32.dll的基地址和解析其导出表是关键步骤。这要求我们对 PE 文件的结构有清晰的认识，包括 TEB、PEB、PE 头、导出表等核心概念。通过实际操作，我更加熟悉了这些结构在内存中的布局及其作用。
2. 哈希值匹配的优势
   使用哈希值匹配函数名的方式比直接比较字符串更为高效和简洁。这种方式不仅减少了代码量，还提高了程序运行的效率。同时，我也学会了如何通过循环右移和累加计算函数名的哈希值，并将其应用于函数地址的查找。
3. Shellcode 的通用性设计
   实验的核心目标是编写能够在不同系统中运行的通用 Shellcode。通过动态搜索 API 函数地址的方式，避免了硬编码函数地址带来的兼容性问题。这种方法让我认识到，在实际开发中，尤其是涉及底层编程时，灵活性和通用性是非常重要的设计原则。

希望在今后的学习中，能够将这些知识应用到更多的实际场景中，为软件安全领域的发展贡献自己的力量。