

《漏洞利用及渗透测试基础》实验报告

姓名：齐明杰 学号：2113997 班级：信安2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

实验过程：

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。

首先，总结一下我们将要用到的函数：

- 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

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

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。

上述复杂的操作可以用如下简单的代码来实现：

```
int main()
{
    _asm
    {
        mov eax, fs:[0x30] ;PEB 的地址
        mov eax, [eax + 0x0c] ; PEB_LDR_DATA 结构体的地址
        mov esi, [eax + 0x1c] ; 指针 InInitializationOrderModuleList
```

```
        lodsd
        mov eax, [eax + 0x08] ;eax 就是 kernel32.dll 的地址
    }
    return 0;
}
```

(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。

定位 kernel32.dll 导出表及其导出函数名列表的代码如下：

```
mov    ebp, eax           //将 kernel32.dll 基地址赋值给 ebp
mov    eax,[ebp+0x3C]     //dll 的 PE 头的指针（相对地址）
mov    ecx,[ebp+eax+0x78] //导出表的指针（相对地址）
add    ecx,ebp           //ecx=0x78C00000+0x262c 得到导出表的内存地址
mov    ebx,[ecx+0x20]     //导出函数名列表指针
add    ebx,ebp           //导出函数名列表指针的基地址
```

(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。

同时，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。我们使用如下代码计算 API 的 hash 值：

```

#include <stdio.h>
#include <windows.h>
DWORD GetHashCode(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= GetHashCode("MessageBoxA");
    printf("%#x\n",hash);
    getchar();
}

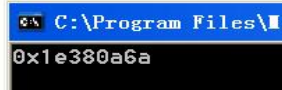
```

对 MessageBoxA 的 hash 计算结果如下图：

```

}
main()
{
    DWORD hash;
    hash= GetHashCode("MessageBoxA");
    printf("%#x\n",hash);
    getchar();
}

```



C:\Program Files\...
0x1e380a6a

同样地可分别计算出 ExitProcess 和 LoadLibrary 的 hash 值。

综上所述，完整 API 函数自搜索代码如下图所示：

```

#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
    }
}

```

```

//=====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

//=====函数名的hash运算

//=====让他做些自己想做的事
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;
}

```

```

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    edi            //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

//=====让他做些自己想做的事

```

运行结果如下图所示:

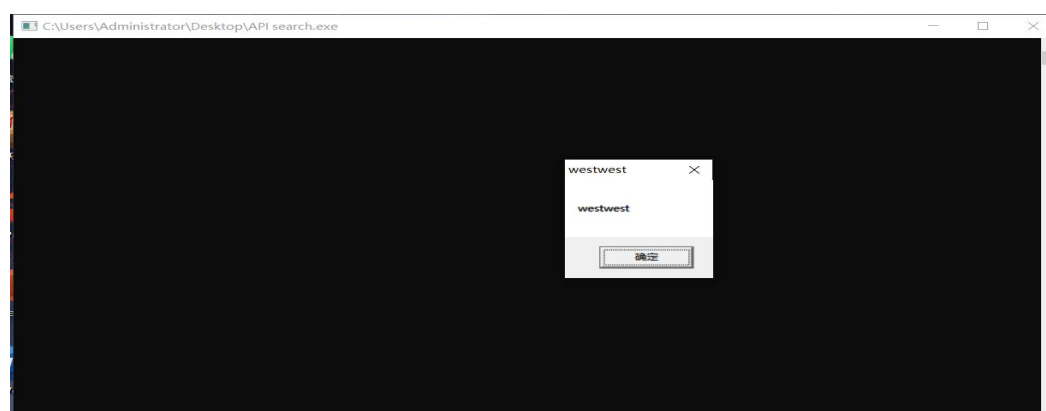
```

//=====让他做些自己想做的事
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(
    push     ebx
    call     [edi-0x08]    //ExitProcess(0);
    nop
    nop

```



将 exe 文件放入 win10 系统中, 运行结果如下图:



在 Windows 10 操作系统上运行生成的 EXE 程序后, 成功显示一个消息框, 内容为 "westwest", 标题也为 "westwest". 在关闭消息框后, 程序使用 ExitProcess 函数正常退出. 这证明了 API 函数自搜索功能在 Windows 10 操作系统上成功执行。

心得体会：

通过本次 API 函数自搜索实验，我学会了以下几点：

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、提高了汇编语言能力：本实验采用了内联汇编，使我们对汇编语言有了更深入的了解。通过对汇编代码的分析和实践，我们学会了如何在 C 语言中嵌入汇编代码，并能够掌握寄存器、堆栈等底层操作。