《软件安全》实验报告

姓名：李雅帆　　学号：2213041　　班级：信安班

一、实验名称

API 函数自搜索实验

二、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 Windows10 操作系统里验证是否成功。

三、实验过程

1.在 VC6 中对代码文件进行编译，验证程序是否能够正常运行。

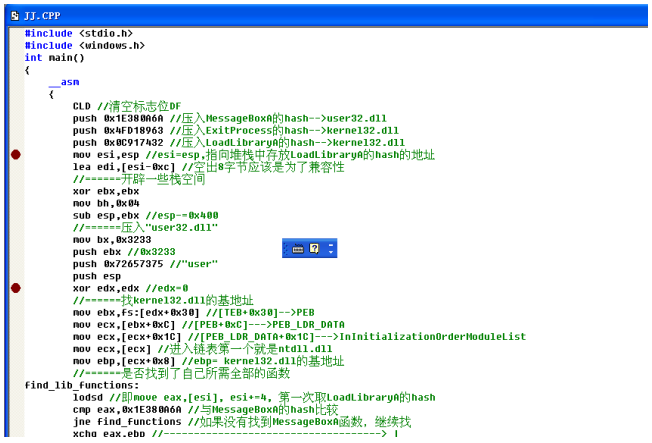
JJ.CPP *

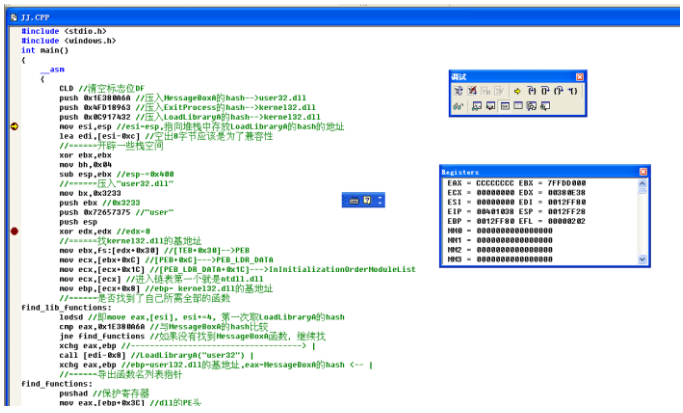
```
#include <stdio.h>
#include <windows.h>
int main()
{
    __asm
    {
        CLD //清空标志位DF
        push 0x1E380A6A //压入MessageBoxA的hash-->user32.dll
        push 0x4FD18963 //压入ExitProcess的hash-->kernel32.dll
        push 0x0C917432 //压入LoadLibraryA的hash-->kernel32.dll
        mov esi,esp //esi=esp,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-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
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
        //------让他做些自己想做的事
function_call:
        xor ebx,ebx
        push ebx
```

```
        jne find_lib_functions
        //======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;
}
```

2.将生成的 exe 文件复制到 win10 操作系统中，验证是否能够正常运行。

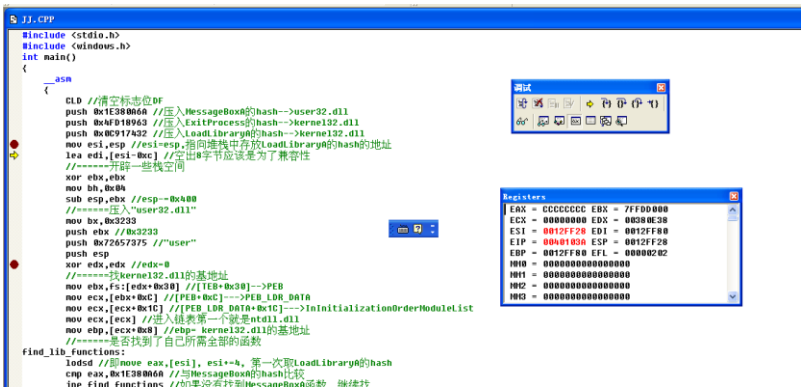
在程序中相应位置加入断点，进入 vc 反汇编模式。



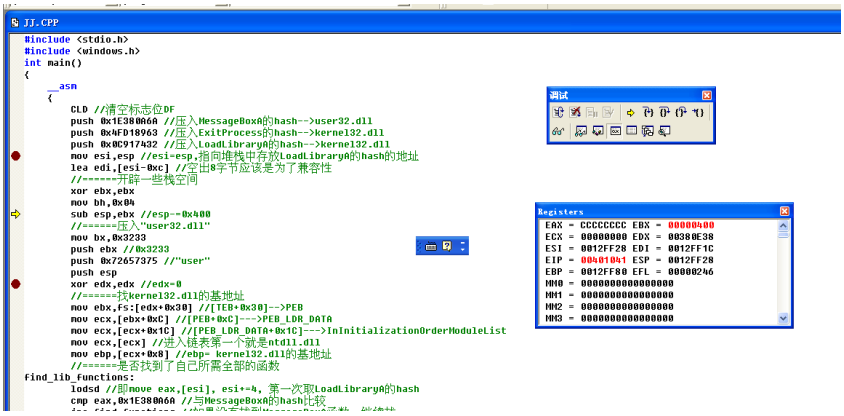


在第一个断点的位置，首先是三个 push 来 push 哈希值，这里的哈希值是通过一个独立的程序算出来的。在后面做函数名比较时，比较的也是这样一个哈希值，而不是字符串。

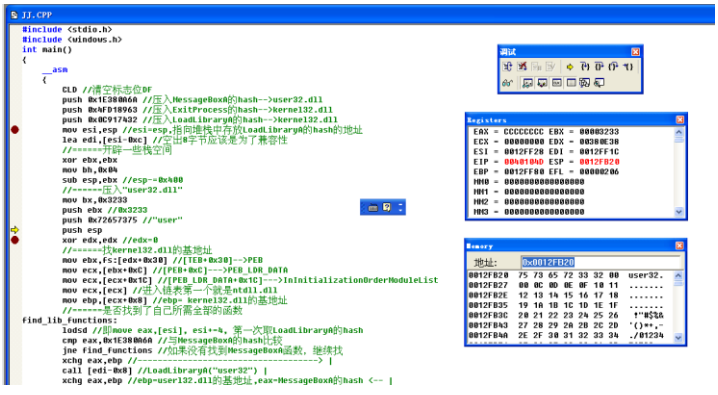
此时进入下一步，esi 寄存器的值此后不发生变化。



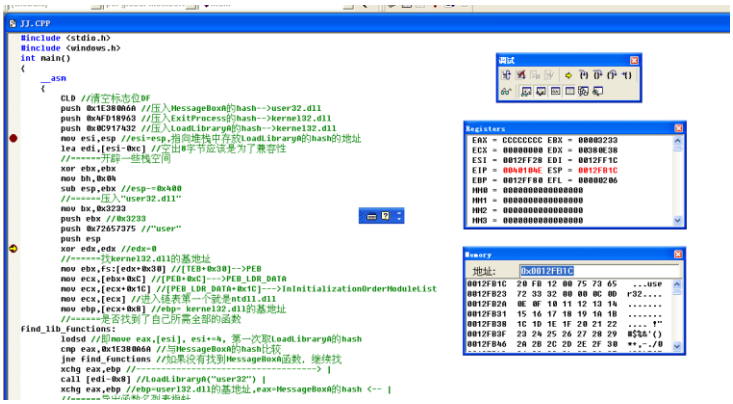
接下来抬高 bh，影响 EBX。



将 esp 抬高后，push 了 user32，这个时候 esp 中储存的是 user32。

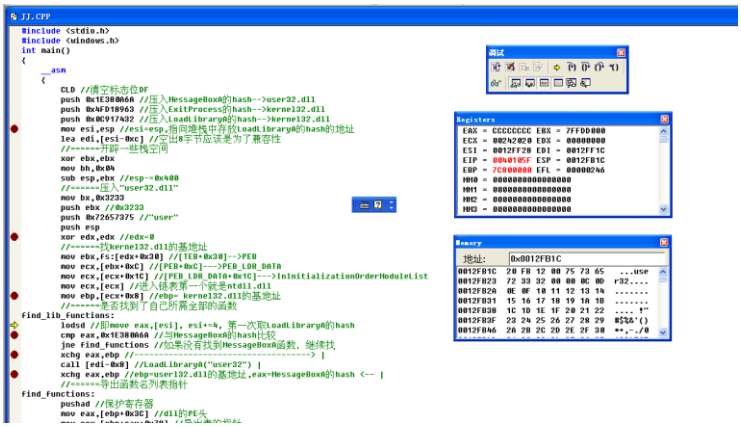


Esp 再次 push 之后，register 中 ESP 的地址就变小了，此时在查看该地址下的存储，可以看出此时存储的是 user32 字符串的地址，我们在此处添加了断点。

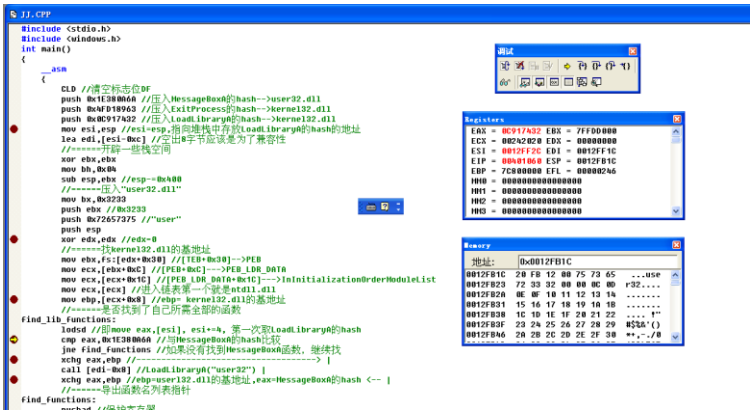


由于我们将来要完成 MessageBox 的调用，所以 shellcode 的逻辑为在找到最后一个 MessageBox 函数的调用时，我们要找到它的地址，调用的时候要先将 user32 参数值入栈，这样将来的时候栈顶刚好是 LoadLibrary 函数对应的参数值。

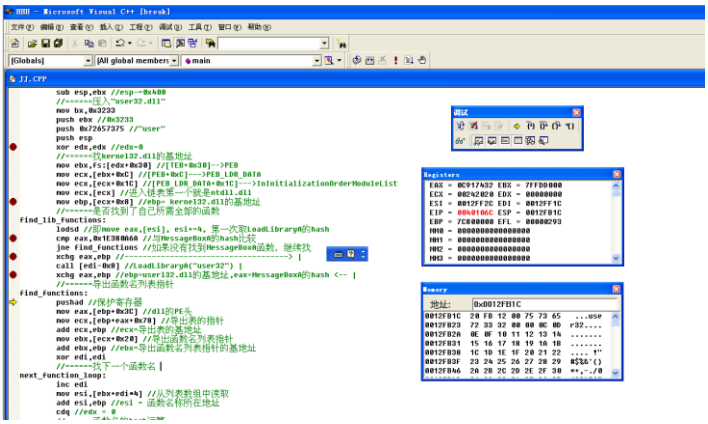
再向下执行，接下来是核心逻辑 lodsd。



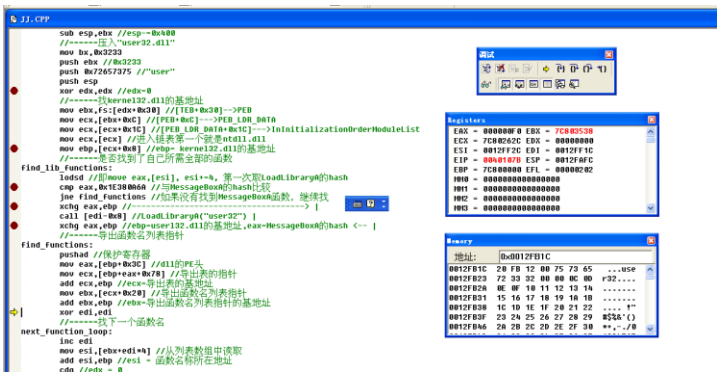
在 lodsd 中，一个是 eax 的值变了，一个是 esi 的值也变了，按下 F10，可以看出此时 eax 里存的是 LoadLibrary 的哈希值。



然后进行了一次 cmp 比较，来比较是不是 MessageBox 的哈希值，即是不是要找的最后一次的哈希值，如果不是，标志位就会告诉我们，就会转到相应位置



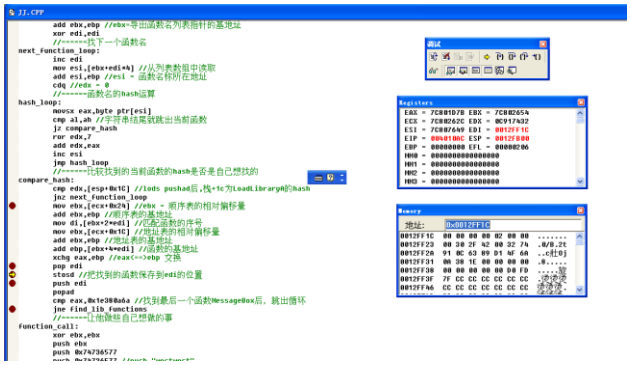
此时会先转导出表。



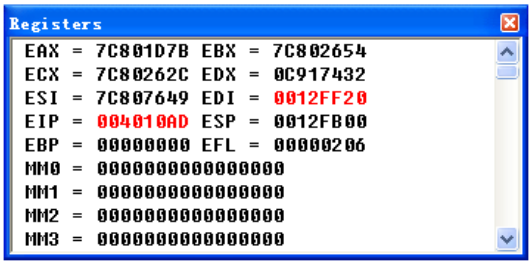
找到函数名列表后，就会开始一个个的取，取到函数名后，会计算它的哈希值，一直到哈希值计算完毕，才会跳转到比较的位置，然后用计算的哈希值和存着的值比较，不是的话就继续找，点击 F5，直到找到了要找到值，即终止。

```
        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
        //======让他做些自己想做的事
function_call:
```

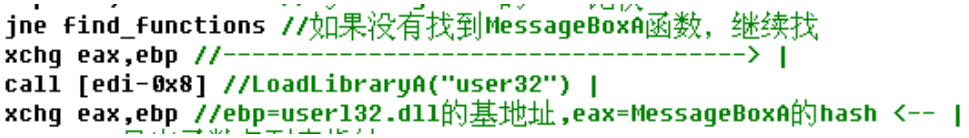
此时我们要计算他的虚拟地址，通过找他的相对偏移量，来做一个加法计算，加上基地址，即可算出虚拟地址，edi 在这是寄存器的目标地址，点击 F10，观察 edi 的值，这里面就存放了我们刚才找到的虚拟地址。



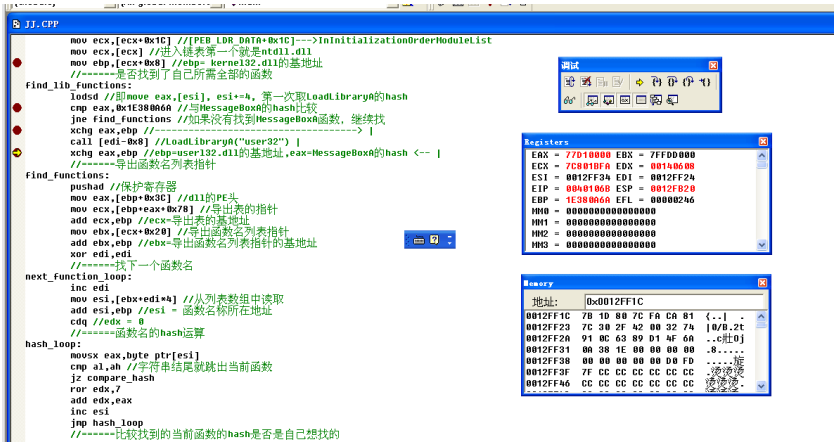
按下 F10，edi 加 4，这个时候我们就成功存储到了 edi 里。



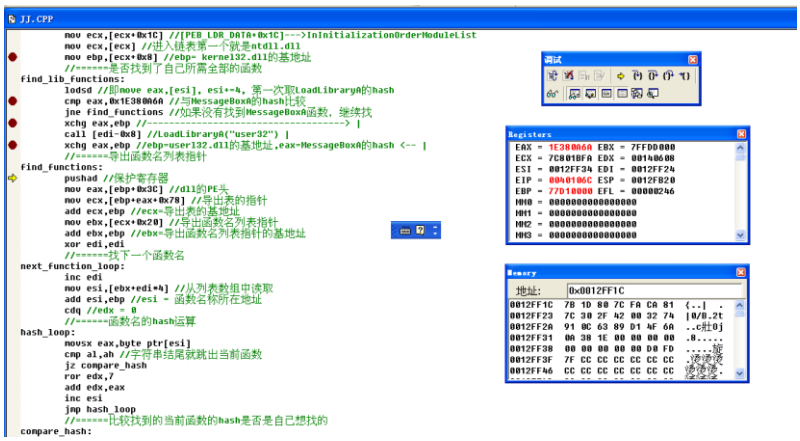
然后保存这个寄存器的状态，再次进行哈希值比较，最终找到的 eax 就是我们要找的最后一个，此时点击 F10，就不会跳转到 Find_function 了，而是执行下面的语句。



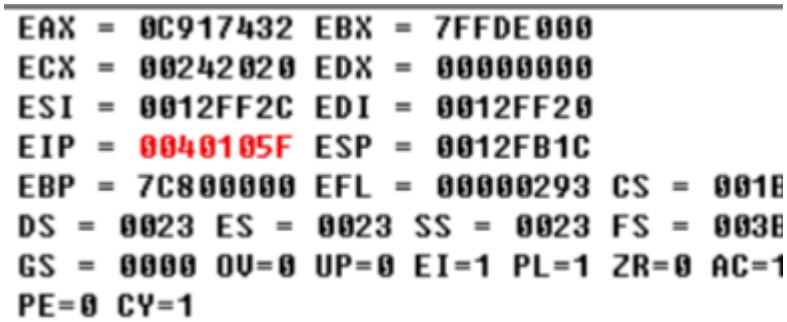
进行完 call 调用是发现，此时的 eip 与之前的 12FB1C 相差不大。



此时点击 F10，ebp 发生变化。



此时 ebp 是 user32 的地址，再往下在循环中导出表就是 user32 的导出表，此后再按 F10，就不会返回了，而是会进入 function_call，此时 edi 中储存的是三个函数的地址，而且 LoadLibrary 函数以及调用了，接下来就会进行 shellcode 的编写，为了程序健壮，最后调用退出程序。



四、心得体会

这次实验让我深入了解了在 Windows 操作系统下进行 API 函数自搜索的过程。通过手动定位函数地址、将函数加载到程序中、入栈相关参数、调用函数等步骤，我深入了解了操作系统底层的一些原理和机制。

我学会了如何通过定位 PEB（Process Environment Block）来获取到 ntdll.dll 的地址，从而进一步获取到其他动态链接库的地址。这个过程需要对操作系统的内存结构有一定的了解，也让我更加熟悉了操作系统的工作原理。

了解到了在获取到动态链接库的基地址后，如何通过 PE 头和导出表来定位到具体函数的虚拟地址。这需要一些计算和 hash 匹配的操作，让我对二进制文件的解析有了更深入的了解。

最后，我学会了如何在程序中调用这些函数，并通过消息框的方式输出相关信息。这个过程虽然涉及到一些汇编级别的操作，但通过实践和理解，我对程序的执行流程和调用栈有了更加清晰的认识。