《漏洞利用及渗透测试基础》实验报告

姓名： 艾明旭 学号：2111033 班级：信息安全一班

**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windows10操作系统里验证是否成功

**实验过程：**

之前进行的实验所编写的Shellcode是采用硬编址的方式来调用相应API函数，即获取所要使用函数的地址，然后将该地址写入ShellCode，从而实现调用。但如果系统的版本变了，很多函数的地址往往都会发生变化，那么调用肯定就会失败了。所以在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。本实验就是编写程序完成API函数自搜索，从而使得Shellcode可以通用。

我们以MessageBoxA函数的调用的shellcode为例：

首先要用到的函数是：

MessageBoxA，位于user32.dll中，用于弹出消息框。

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。

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。

通用型shellcode编写的步骤：

（1）第一步：定位kernel32.dll。

（2）第二步：定位kernel32.dll的导出表。

（3）第三步：搜索定位LoadLibrary等目标函数。（API函数自搜索）

（4）第四步：基于找到的函数地址，完成Shellcode的编写。

API自搜索技术：

**1）定位kernel32.dll**

如果想要在 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;  } |

**2）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。

* 1. 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 |

**3）搜索定位目标函数**

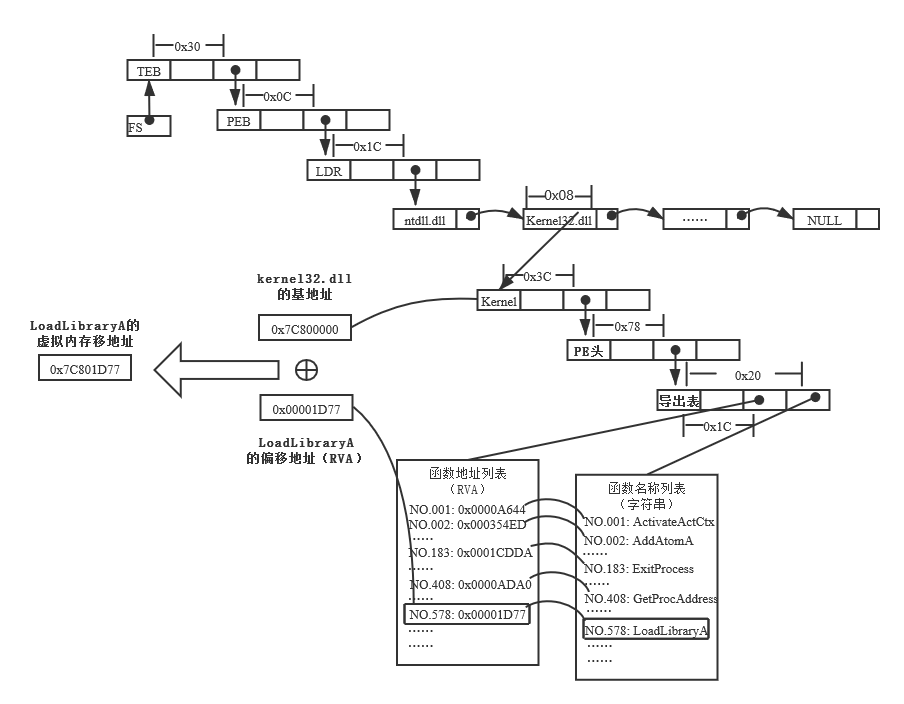
至此，可以通过遍历两个函数相关列表，算出所需函数的入口地址：

（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。

上述完整的几个步骤如下图：



**3. 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”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。

压缩函数名的hash算法：

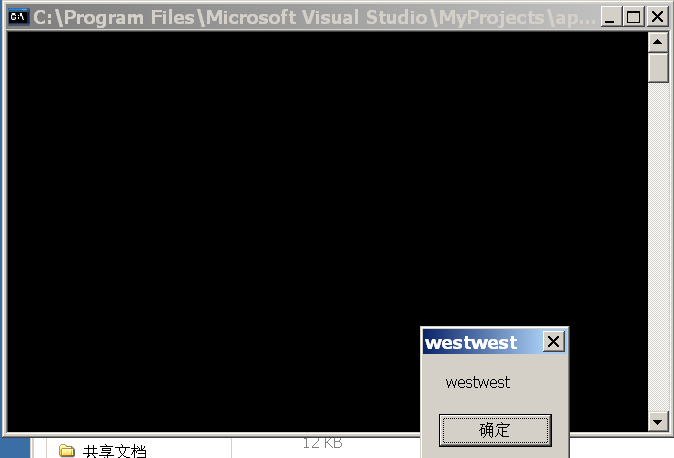
|  |
| --- |
| #include <stdio.h>  #include <windows.h>  DWORD GetHash(char \*fun\_name)  {  DWORD digest=0;  while(\*fun\_name)  {  digest=((digest<<25)|(digest>>7)); //循环右移7位  /\* movsx eax,byte ptr[esi]  cmp al,ah  jz compare\_hash  ror edx, 7 ; ((循环))右移,不是单纯的 >>7  add edx,eax  inc esi  jmp hash\_loop  \*/  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，我们可以获得MessageboxA的hash值。接下来，我们可以在shellcode中通过压栈的方式将这个hash值压入栈中，再通过比较得到动态链接库中的API地址。

**完整API函数自搜索代码**。首先，基于上述流程找到函数的入口地址；之后，可以编写自己的shellcode，如下面完整代码中的function\_call

|  |
| --- |
| #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=userl32.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  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;  } |

运行结果：

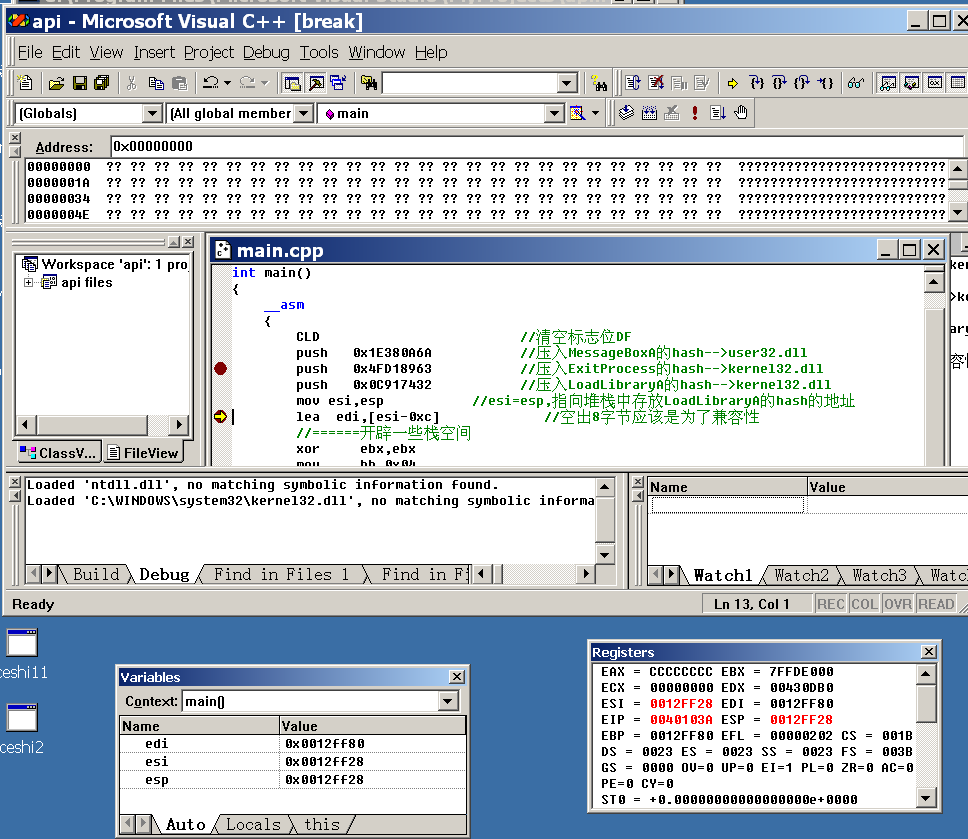


接下来我们通过打断点进行逐步解析：

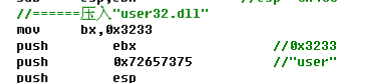
Push进三个hash值 后续函数值的比较变为hash值的比较



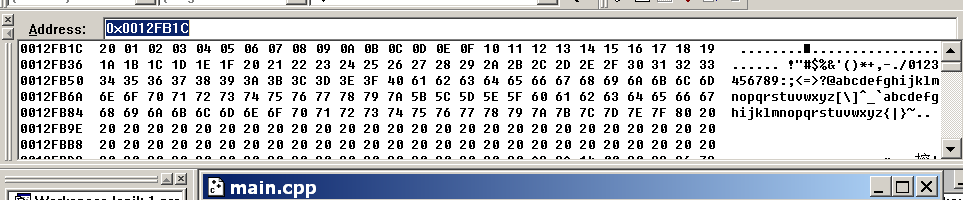
ESI地址 后面通过ESI寄存器寻找hash值



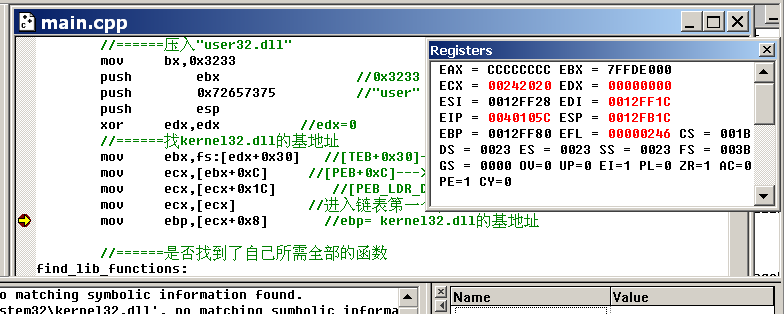
Push进user32



ESP里存了user32的地址

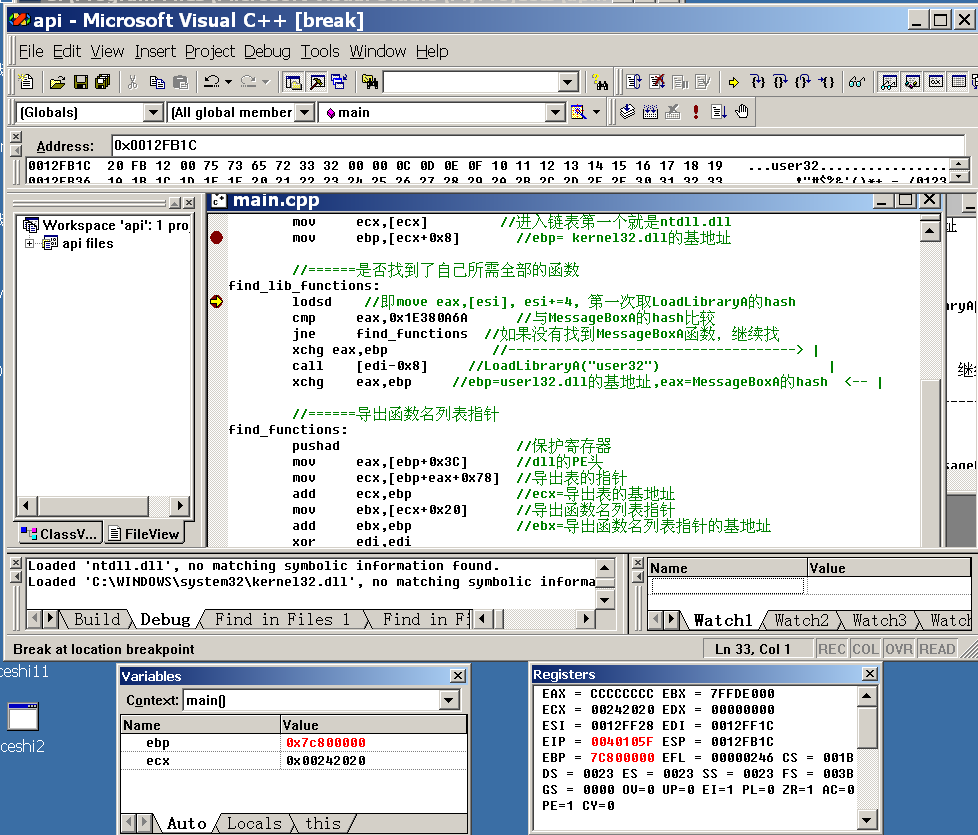


执行到 mov ebp,[ecx+0x8] EBP存的是kernel32.dll的基地址



执行lodsd，即mov eax,[esi], esi+=4，可以看到EAX ESI的值发生了变化

此时EAX中存的是LoadLibraryA的hash

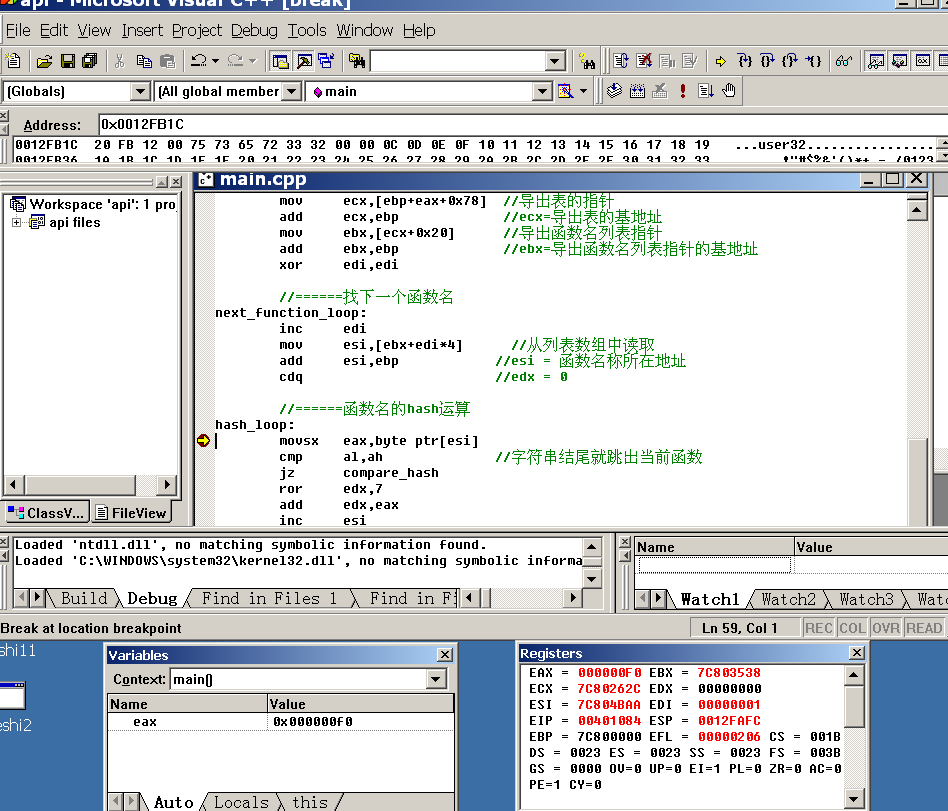


将字符串进行对比，若不相等，则跳转到find\_functions

find\_functions的作用是找到函数名表

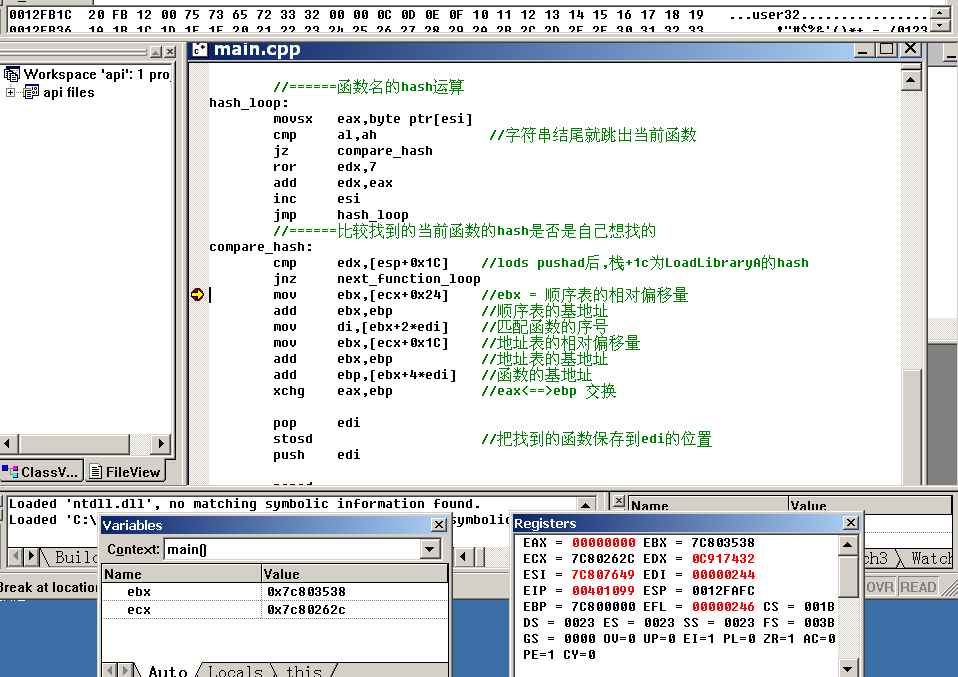
找到函数名后计算函数的hash值

这是一个循环，hash计算完毕后会跳转到比较的位置，如果比较结果不相同则跳到next\_Function\_loop接着寻找



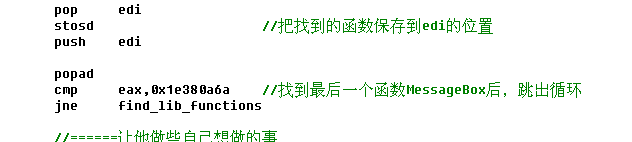
找到后计算虚拟地址：基地址+相对偏移量=所求地址

Pop edi stosd push edi把所求得的地址存到edi



可以看到edi里存着找到的虚拟地址

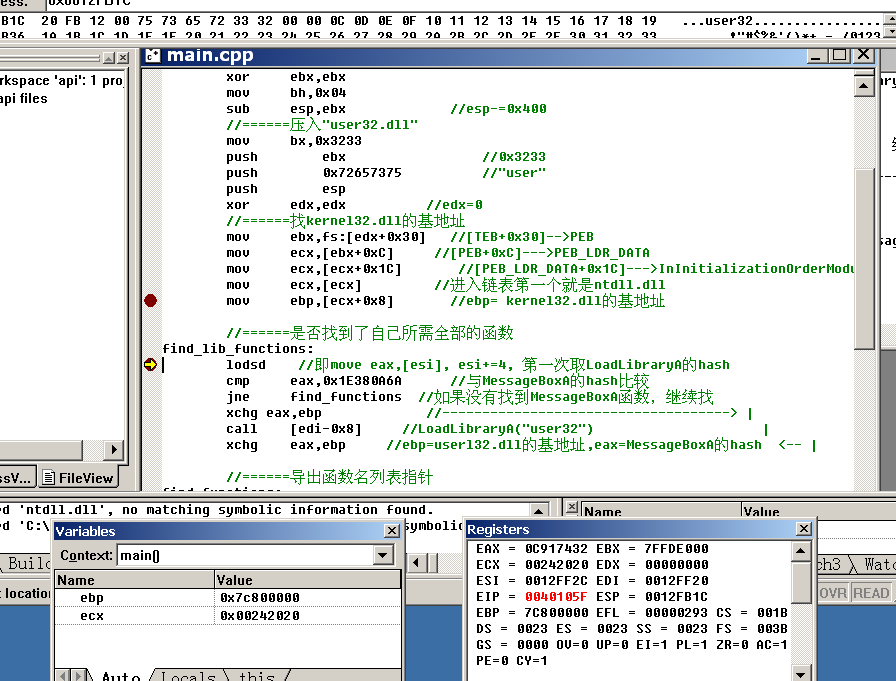
进行cmp操作，若找到的不是最后一个函数，进入循环继续寻找



最后一次循环 此时ebp=user32的基地址 eax=MessageBox的hash值

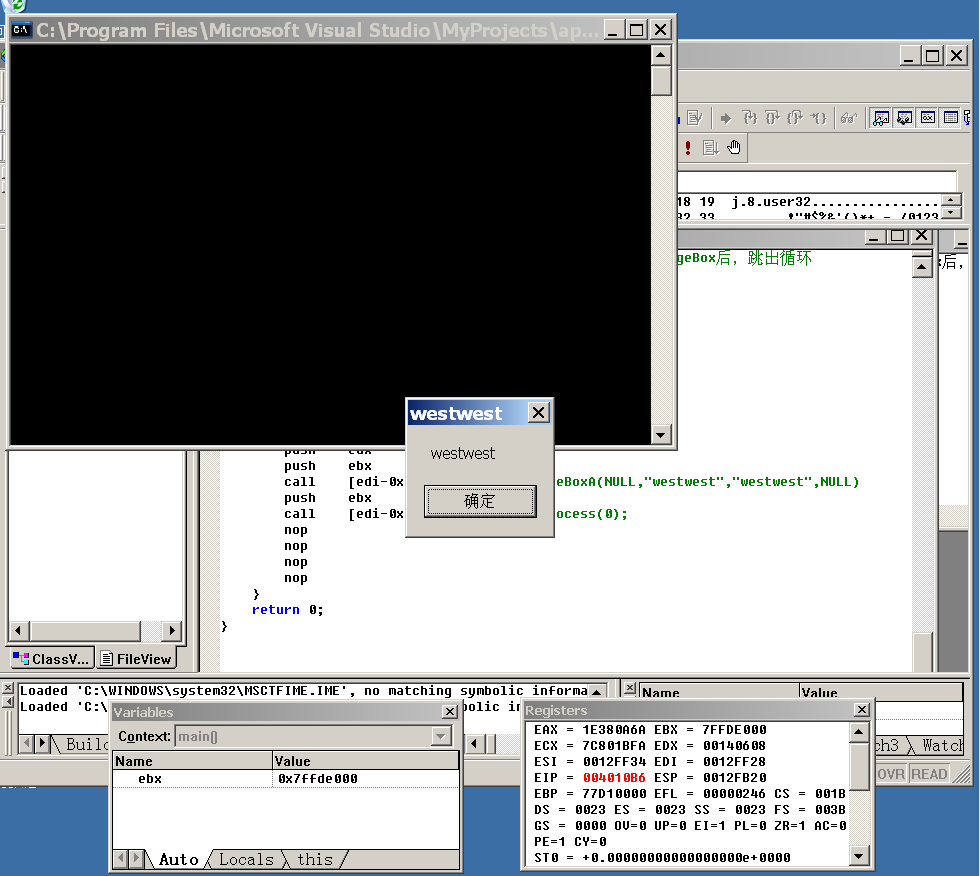
找到最后一个函数MessageBox后，跳出循环，此时EDI里存了三个函数的hash值

最后在function\_call中编写Shellcode代码即可



将生成的exe程序，复制到windows 10操作系统里验证是否成功

弹出对话框 Shellcode攻击成功



**心得体会：**

初步了解并掌握实现了API函数自搜索技术，并且对相应的函数在进行自搜索过程当中所运用到的栈帧的变化以及寄存器的变化有了一定的了解。对相对地址偏移量和其中的基址地址等变化有了一定的掌握。

在此基础上，我还明白了通用Shellcode的编写原理，和过去的实验进行对比，我发现了shellcode编写过程当中存在的一些规律和原理。