### API函数自搜索技术

前面我们讲的Shellcode的编写，都采用硬编址的方式来调用相应API函数。首先，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。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。

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。

**1. 通用型Shellcode的编写逻辑**

仍然以MessageBoxA函数的调用的shellcode为例，来解释通用型shellcode的编写逻辑。

首先，总结一下我们将要用到的函数：

·**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。

进而，介绍通用型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。

（4）第四步：基于找到的函数地址，完成Shellcode的编写。

难点在于第一步到第三步，即如何实现API函数自搜索。

**2. API函数自搜索技术**

**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;  } |

这个代码可以在最后一句设置断点，自行进行演示验证。比如，在Windows XP系统里利用Depends工具查看Kernel32.dll的加载基址与程序运行的结果。本文这里省略。

**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，可以用下图5-13来概括。

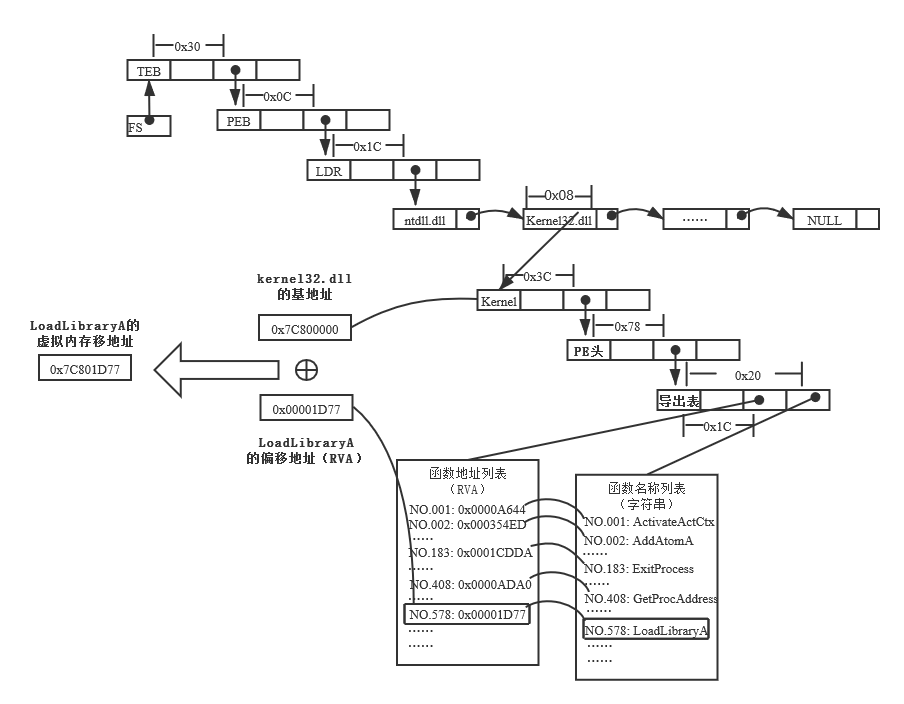


图5-13 搜索定位目标函数步骤图

**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”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。使用的hash算法如示例5-10所示。

示例5-10（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。

**示例5-11（完整API函数自搜索代码）：**

|  |
| --- |
| #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;  } |

**实验七：基于示例5-11，完成上述实验，将生成的exe程序，复制到windows 10操作系统里验证是否成功。**