[一、导入表注入 2](#_Toc5387)

[二、挂起线程注入 12](#_Toc28497)

[三、挂起进程注入 19](#_Toc17483)

[四、调试器注入 21](#_Toc18923)

[五、注册表注入 28](#_Toc23442)

[六、钩子注入 32](#_Toc28285)

[七、APC注入 37](#_Toc4042)

[八、远程线程注入 41](#_Toc18600)

[九、输入法注入 45](#_Toc1773)

[十、DLL劫持 52](#_Toc5108)

[Ring3注入总结 58](#_Toc19981)

[关于Ring3下的反注入思路 60](#_Toc4651)

**Ring3注入总结及编程实现**

这段时间时间学习了一些Ring3下的注入方式，这些方式大多都被玩烂了，但是为了练习一下开发所以就边学注入边实验，每学一种注入都用MFC或者汇编开发一个小工具加强印象，感觉可能一些新手朋友可能会用到，水平有限，如有错误，望大家多多赐教。

# 一、导入表注入

**0x00 导入表注入原理**

导入表注入简单了来说就是给导入表添加导入表项，当系统的PE加载器完成PE加载的时候加载我们的DLL以完成注入，很多PE工具都有这个功能，比如说LordPe。我们来借鉴一下，这里以LordPe为例子模仿LordPe编写一个导入表注入器。

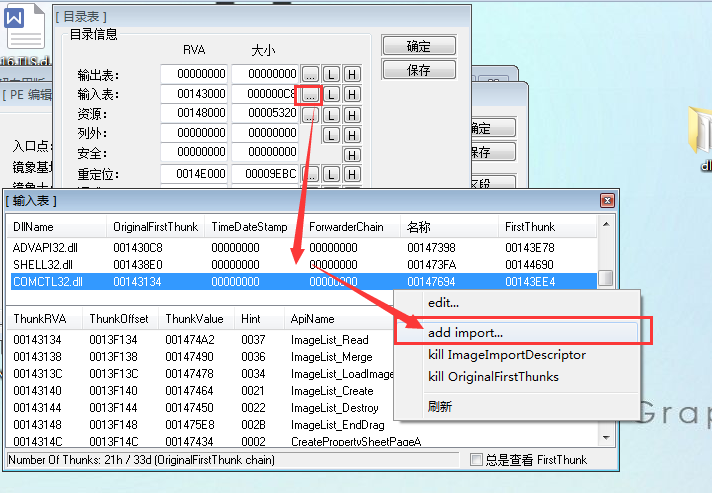
我们准备了一个被注入程序Server.exe，和一个MyDll.dll，DLL导出一个My\_MsgBox的导出函数，功能就是加载成功是弹个MessageBox，以后的的注入中都会用这个DLL文件做演示。

注入的DLL源码:

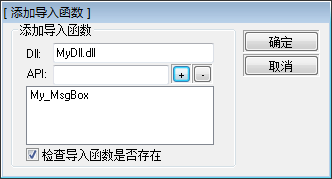
|  |
| --- |
| BOOL APIENTRY DllMain( HANDLE hModule,  DWORD ul\_reason\_for\_call,  LPVOID lpReserved  )  {  switch (ul\_reason\_for\_call)  {  case DLL\_PROCESS\_ATTACH:  MessageBox(NULL,"导入表注入成功",NULL,MB\_OK);  break;  case DLL\_THREAD\_ATTACH:  case DLL\_THREAD\_DETACH:  case DLL\_PROCESS\_DETACH:  break;  }  return TRUE;  }  EXTERN\_C EXPORT My\_MsgBox()  {  //在这里添加需要执行的代码，不要破坏堆栈平衡  MessageBox(NULL,"导入表注入成功",NULL,MB\_OK);  } |

我们先围观一下LoadPe的导入表注入功能

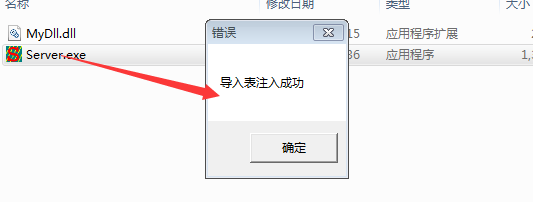
打开LordPe的PE编辑功能。如下如点击。



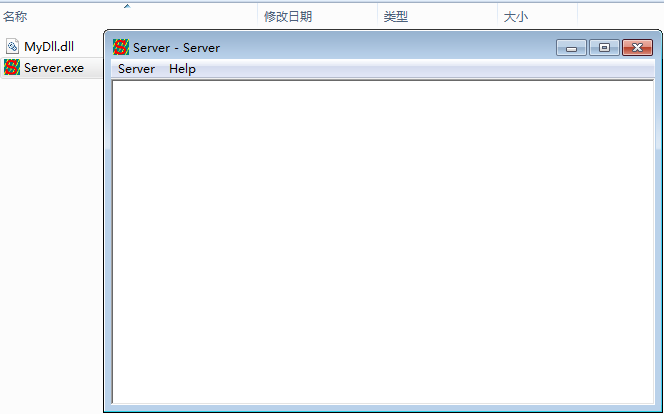
添加DLL名和导出的函数，点击确定完成修改



再次打开我们的目标程序Server.exe，可以看到先弹出了MyDll.dll中的导入表注入成功的对话框

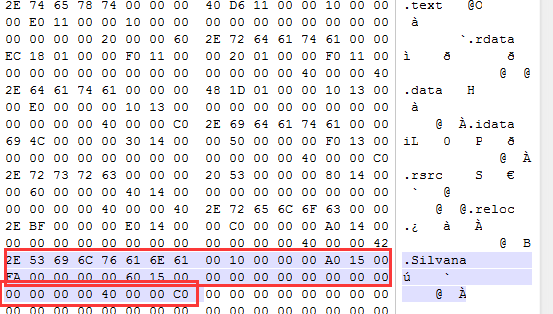


点击确定，弹出了正常的的服务器程序，说明导入表注入成功。

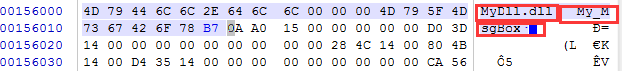


用Winhex观察Pe文件，对比看看LordPe对Pe文件做了啥。

➀首先填加一个节，如图



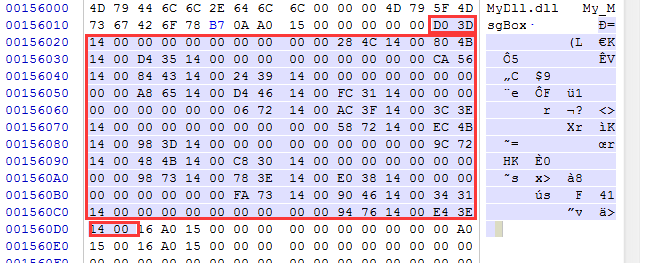
➁在节中写入DLL名和函数名（这里其实写入的是IMAGE\_IMPORT\_BY\_NAME结构体）如图



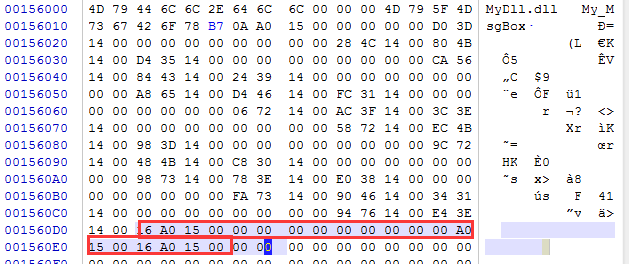
➂紧接其后的是写入的IMAGE\_THUNK\_DATA结构体，如下图



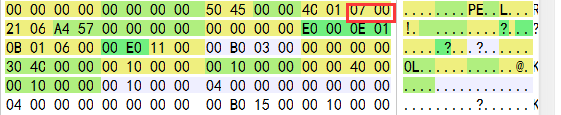
④把原来的导入表拷贝到新的节中，如下图



⑤将新的导入表项写在原来的导入表项后面，如下图



⑥最后更新FILE HEADER头中的NumberOfSection



总体思路就是添加一个新节，写入我们的DLL名和导出函数完成注入

**0x01 导入表注入编程实现**

➀我修改的方法比较简单，为了省事儿，并且不破坏原程序，复制一份原程序名为Temp.exe，读取原来的Server.exe的数据，然后对Temp.exe进行修改

|  |
| --- |
| //复制一份文件用于修改，源文件保留。  bRet = ::CopyFile(m\_strFile.GetBuffer(0),m\_strTempPath.GetBuffer(0),FALSE); |

➁在新的节写入DLL名,紧随其后写入所有的的IMAGE\_IMPORT\_BY\_NAME结构,也就是写入所有函数名

|  |
| --- |
| //1.在复制的文件中写入DLL名  fseek(fp, dwNewFA, SEEK\_SET);  dwNewOffset = m\_strDll.GetLength() + 1;  fwrite(m\_strDll.GetBuffer(0), dwNewOffset, 1, fp);  DWORD \*arrINTRva = new DWORD[nFunNum + 1];  memset(arrINTRva, 0, sizeof(DWORD)\*(nFunNum + 1));  //2.写入所有的的IMAGE\_IMPORT\_BY\_NAME结构,也就是写入所有DLL名  for (int i = 0; i < nFunNum; i++)  {  DWORD dwTempRva = 0;  static int nFunLen = 0;  PIMAGE\_IMPORT\_BY\_NAME pImportFun = new IMAGE\_IMPORT\_BY\_NAME;  pImportFun->Hint = i;  CString strFunName = m\_strFunList.GetItemText(i,0);  fseek(fp, dwNewFA + dwNewOffset, SEEK\_SET);  //计算IMAGE\_IMPORT\_BY\_NAME的RVA存入数组  dwTempRva = dwNewSectionRVA + dwNewOffset;  arrINTRva[i] = dwTempRva;  dwNewOffset = dwNewOffset + strFunName.GetLength() + 1 + sizeof(WORD);  memcpy(pImportFun->Name, strFunName.GetBuffer(0), strFunName.GetLength() + 1);  fwrite(pImportFun, strFunName.GetLength() + 1 + sizeof(WORD), 1, fp);  } |

➂写入所有的的INT结构

|  |
| --- |
| //3.写入所有的的INT结构  for (int i = 0; i < nFunNum + 1; i++)  {  fseek(fp, dwNewFA + dwNewOffset, SEEK\_SET);  dwNewOffset += sizeof(DWORD);  //末尾填充0结构体  fwrite(&arrINTRva[i], sizeof(DWORD), 1, fp);  } |

④在新节中存入原有的IID和新的IID结构

|  |
| --- |
| //4.申请新空间存放旧的的IID和新的IID  lpNewImport = (PIMAGE\_IMPORT\_DESCRIPTOR)malloc(dwNewImportSize);  memset(lpNewImport, 0, dwNewImportSize);  memcpy(lpNewImport, lpImport, dwImportSize);  int i = 0;  while (1)  {  if (lpNewImport[i].OriginalFirstThunk == 0  && lpNewImport[i].TimeDateStamp == 0  &&lpNewImport[i].ForwarderChain == 0  && lpNewImport[i].Name == 0  && lpNewImport[i].FirstThunk == 0)  {  lpNewImport[i].Name = dwNewSectionRVA;  lpNewImport[i].TimeDateStamp = 0;  lpNewImport[i].ForwarderChain = 0;  lpNewImport[i].FirstThunk = dwINTRVA;  lpNewImport[i].OriginalFirstThunk = dwINTRVA;  break;  }  else i++;  }  //计算新的导入表RVA  dwNewImportRva = dwNewSectionRVA + dwNewOffset;  //写入所有的导入表项  fseek(fp, dwNewFA + dwNewOffset, SEEK\_SET);  fwrite(lpNewImport, dwNewImportSize, 1, fp); |

⑤添加一个新的节表头项

|  |
| --- |
| //5.添加一个新节表头项  memset(&ImgNewSection, 0, sizeof(IMAGE\_SECTION\_HEADER));  //添加名为.newsec的新节  strcpy((char\*)ImgNewSection.Name, ".newsec");  ImgNewSection.VirtualAddress = dwNewSectionRVA;  ImgNewSection.PointerToRawData = dwNewFA;  ImgNewSection.Misc.VirtualSize=ClacAlignment(dwNewOffset, nSectionAlignment);  ImgNewSection.SizeOfRawData=ClacAlignment(dwNewOffset, nFileAlignment);  ImgNewSection.Characteristics = 0xC0000040;  //计算新节头的文件偏移  dwNewSectionOffset = (DWORD)lpFirstSectionHeader -  (DWORD)theApp.m\_stMapFile.ImageBase+ sizeof(IMAGE\_SECTION\_HEADER)\*nSectionNum;  fseek(fp, dwNewSectionOffset, 0);  //写入节表头  fwrite(&ImgNewSection, sizeof(IMAGE\_SECTION\_HEADER), 1, fp);  memcpy(&ImgNewSection,lpFirstSectionHeader,sizeof(IMAGE\_SECTION\_HEADER));  fseek(fp,(DWORD)lpFirstSectionHeader-(DWORD)theApp.m\_stMapFile.ImageBase, SEEK\_SET);  fwrite(&ImgNewSection, sizeof(IMAGE\_SECTION\_HEADER), 1, fp); |

⑥更新NT头数据，更新新的ImageSize，NumberOfSections，导入表数据目录的

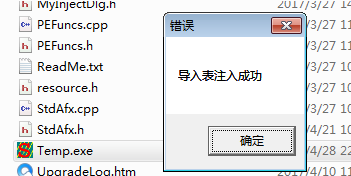
VirtualAddress和Size

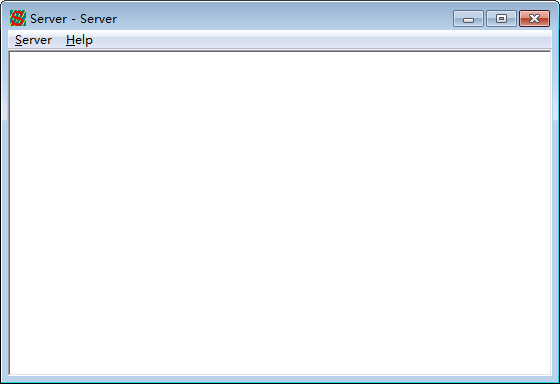
|  |
| --- |
| //6.更新NT头数据  memcpy(lpNewNtHeader, lpNtHeader, sizeof(IMAGE\_NT\_HEADERS));  int nNewImageSize = lpNtHeader->OptionalHeader.SizeOfImage + ClacAlignment(dwNewOffset, nSectionAlignment);  lpNewNtHeader->OptionalHeader.SizeOfImage = nNewImageSize;  lpNewNtHeader->OptionalHeader.DataDirectory[11].Size = 0;  lpNewNtHeader->OptionalHeader.DataDirectory[11].VirtualAddress = 0;  lpNewNtHeader->OptionalHeader.DataDirectory[12].Size = 0;  lpNewNtHeader->OptionalHeader.DataDirectory[12].VirtualAddress = 0;  lpNewNtHeader->FileHeader.NumberOfSections = nSectionNum + 1;  lpNewNtHeader->OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_IMPORT].VirtualAddress = dwNewImportRva;  lpNewNtHeader->OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_IMPORT].Size = dwNewImportSize;  //写入新的NT头  fseek(fp, (DWORD)(lpNtHeader)-(DWORD)theApp.m\_stMapFile.ImageBase, SEEK\_SET);  fwrite(lpNewNtHeader, sizeof(IMAGE\_NT\_HEADERS), 1, fp); |

进行完以上操作，导入表注入基本完成，细节比较多，详细代码请看源码。

**0x02 实验效果**

打开修改后的程序Temp.exe,首先弹出导入表注入成功，接着正常运行，注入成功





# 二、挂起线程注入

**0x00挂起线程注入原理**

这种注入方式的思路：首先先向目标程序中写入我们的ShellCode,比如说写入LoadLibry加载我们的DLL。然后把目标进程中当主线程挂起，然后获取线程上下文，修改线程上下文中的EIP到我们写入的ShellCode处执行完代码，然后再设置为线程原来的上下文·继续执行，执行完成过后在目标中释放申请的空间。

具体编程实现大致实现思路：

1. 构造ShellCode
2. VirtualAllocEx在目标进程中申请空间，WriteProcessMemory写入ShellCode
3. 通过线程快照获取目标主线程
4. OpenThread打开线程，SuspendThread挂起线程
5. GetThreadContext获取目标主线程线程上下文
6. SetThreadContext修改目标主线程上下文到我们写入的ShellCode处执行
7. ResumeThread恢复线程让ShellCode执行
8. VirtualFreeEx扫尾释放空间

其实挂起线程注入的思路其实也是挺简单的，而且这种方式相对于常规的DLL注入，隐蔽性更高,一些病毒也比较喜欢用这种方式。

下面介绍详细的实现步骤。

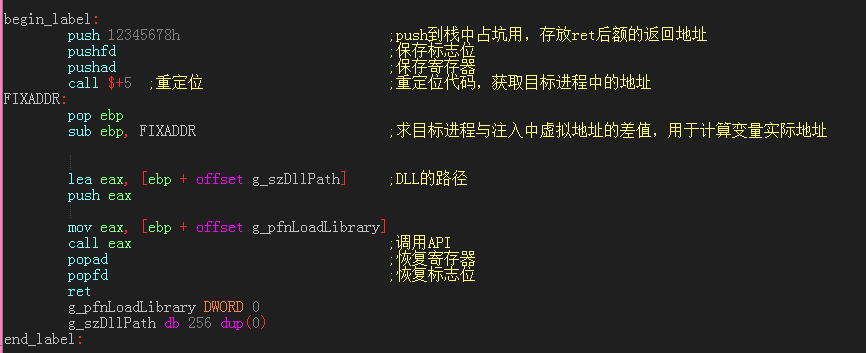
**0x01 挂起线程注入详细的编程实现**

**由于这种注入方式要注入ShellCode，用C++实现比较麻烦一点因为要扣二进制，我也用汇编一个版本，在附件中，这里还是用C++**

**➀ShellCode的构造，LoudLibrary加载DLL**

|  |
| --- |
| //结构必须字节对齐1  #pragma pack(1)  typedef struct \_INJECT\_CODE  {  BYTE byPUSH;  DWORD dwPUSH\_VALUE;  BYTE byPUSHFD;  BYTE byPUSHAD;  BYTE byMOV\_EAX; //mov eax, addr szDllpath  DWORD dwMOV\_EAX\_VALUE;  BYTE byPUSH\_EAX; //push eax  BYTE byMOV\_ECX; //mov ecx, LoadLibrary  DWORD dwMOV\_ECX\_VALUE;  WORD wCALL\_ECX; //call ecx  BYTE byPOPAD;  BYTE byPOPFD;  BYTE byRETN;  CHAR szDllPath[MAX\_PATH];  }INJECT\_CODE, \*PINJECT\_CODE;  #pragma pack() |

**利用C++注入不用考虑重定位的问题，因为C++中提供offsetof宏可以求出变量偏移，但是在汇编实现中就要考虑求变量的偏移了，如下图**



其实在这注入方式中，最有学习意义的就是ShellCode的构造，我总结了一些我在构造ShellCode中学习到的知识点。

1. 代码重定位

在目标进程中，要想实现API调用的难点其实就是传参数，因为我们的注入程序和目标进程中基址是不一样的，这样就注定了写入参数的地址不一样，这就涉及到重定位问题。

经典的重定位代码。

|  |
| --- |
| Call $+5 ;将下一行的地址入栈，获得目标进程下一行的地址  FIXADDR： ;这个是注入进程中地址Lable，用于求差值  Pop ebp ;弹出入栈的目标进程的地址  Sub ebp,FIXADDR ;求得目标进程与注入进程中的地址差值 |

求出了目标进程和注入进程之间的地址差，就可以访问我们写入变量的地址了。

1. 释放问题

一定要等所有的ShellCode执行完成再释放空间，否则会有同步问题，导致目标程序崩溃。

**➁打开进程，写入ShellCode**

|  |
| --- |
| //打开进程  g\_hProcess = OpenProcess(PROCESS\_ALL\_ACCESS, FALSE, m\_dwPid);    if (!g\_hProcess)  {  MessageBox("OpenProcess 失败");  return;  }  g\_lpBuffer=VirtualAllocEx(g\_hProcess,NULL,0x1000,MEM\_COMMIT,PAGE\_EXECUTE\_READWRITE);  if (!g\_lpBuffer)  {  MessageBox("VirtualAllocEx 失败");  return;  }  //给ShellCode结构体赋值  ic.byPUSH = 0x68;  ic.dwPUSH\_VALUE = 0x12345678;  ic.byPUSHFD = 0x9C;  ic.byPUSHAD = 0x60;  ic.byMOV\_EAX = 0xB8;  ic.dwMOV\_EAX\_VALUE = (DWORD)g\_lpBuffer + offsetof(INJECT\_CODE, szDllPath);  ic.byPUSH\_EAX = 0x50;  ic.byMOV\_ECX = 0xB9;  ic.dwMOV\_ECX\_VALUE = (DWORD)&LoadLibrary;  ic.wCALL\_ECX = 0xD1FF;  ic.byPOPAD = 0x61;  ic.byPOPFD = 0x9D;  ic.byRETN = 0xC3;  memcpy(ic.szDllPath, m\_strDllPath.GetBuffer(0), m\_strDllPath.GetLength());  //写入ShellCode  bRet = WriteProcessMemory(g\_hProcess, g\_lpBuffer, &ic, sizeof(ic), NULL);  if (!bRet)  {  MessageBox("写入内存失败");  return;  } |

**➂创建线程快照查找目标程序主线程**

|  |
| --- |
| //创建线程快照查找目标程序主线程  te32.dwSize = sizeof(te32);  hThreadSnap = CreateToolhelp32Snapshot(TH32CS\_SNAPTHREAD, 0);  if (hThreadSnap == INVALID\_HANDLE\_VALUE)  {  MessageBox("CreateToolhelp32Snapshot 失败");  return;  }  //遍历查询目标程序主线程ID  if (Thread32First(hThreadSnap, &te32))  {  do  {  if (m\_dwPid == te32.th32OwnerProcessID)  {  dwThreadId = te32.th32ThreadID;  break;  }  } while (Thread32Next(hThreadSnap, &te32));  } |

**④打开并且挂起目标主线程，获取线程上下文，修改Eip为ShellCode处的地址**

|  |
| --- |
| //挂起目标主线程  bRet = SuspendThread(hThread);  if (bRet == -1)  {  MessageBox("SuspendThread 失败");  return;  }  oldContext.ContextFlags = CONTEXT\_FULL;  bRet = GetThreadContext(hThread, &oldContext);  if (!bRet)  {  MessageBox("GetThreadContext 失败");  return;  }  newContext = oldContext;  newContext.Eip = (DWORD)g\_lpBuffer;  //;将指针指向ShellCode第一句push 12345678h中的地址,写入返回地址  bRet = WriteProcessMemory(g\_hProcess, ((char\*)g\_lpBuffer) + 1, &oldContext.Eip, sizeof(DWORD), NULL);  if (!bRet)  {  MessageBox("写入内存失败");  return;  } |

**⑤设置上下文，恢复线程跑起来**

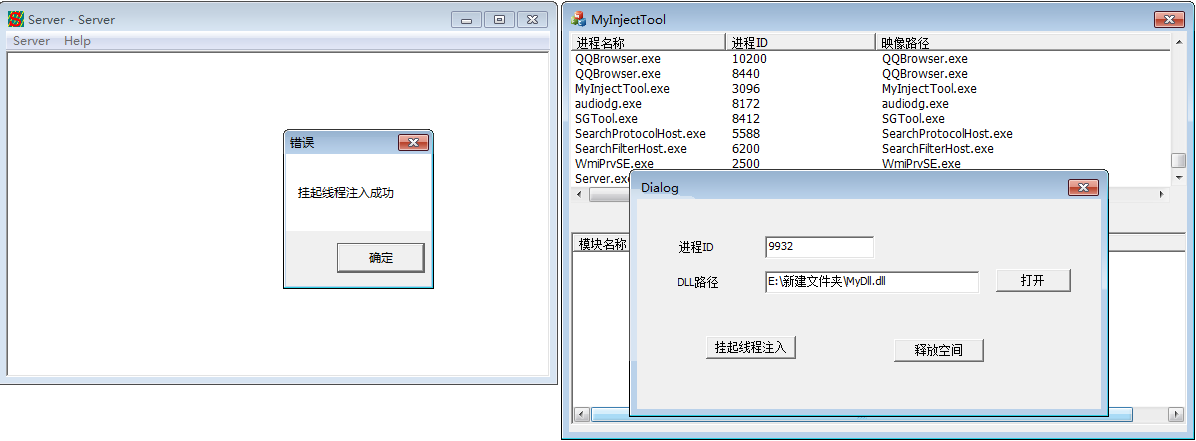
|  |
| --- |
| bRet = SetThreadContext(hThread, &newContext);  if (!bRet)  {  MessageBox("SetThreadContext 失败");  return;  }  //然后把主线程跑起来  bRet = ResumeThread(hThread);    if (bRet == -1)  {  MessageBox("ResumeThread 失败");  return;  } |

**⑥扫尾工作，单独设了个函数清除目标进程中申请的空间，注意这个操作务必等待我们的ShellCode执行完再执行，否则会导致目标程序崩溃**

|  |
| --- |
| if (!VirtualFreeEx(g\_hProcess, g\_lpBuffer, 0, MEM\_RELEASE))  {  MessageBox("VirtualFreeEx 失败");  return;  }  MessageBox("释放对方空间成功"); |

详细源码见附件

实验效果如下，Dll注入成功



# 三、挂起进程注入

**0x00 挂起进程注入原理**

挂起进程注入其实和挂起线程注入其实是一个道理，利用了CreateProcess这个Api中的CREATE\_SUSPENDED以挂起的方式打开进程，趁机进行注入。区别在于挂起进程在程序开始时候，挂起线程在程序运行中，本质上都是挂起主线程，执行写入的ShellCode，这里就不做赘述了，具体原理看上面的挂起线程注入

实现步骤:

1. 构造ShellCode
2. CreateProcess以挂起的方式启动目标进程

3.OpenThread打开线程，SuspendThread挂起线程

4.GetThreadContext获取目标主线程线程上下文

5.SetThreadContext修改目标主线程上下文到我们写入的ShellCode处执行

6.ResumeThread恢复线程让ShellCode执行

7.VirtualFreeEx扫尾释放空间

这种方式和挂起线程的方式代码编写思路几乎一样，并且更加简单方便，因为CreateProcess可以拿到目标进程的进程句柄和主线程句柄，省去了自己去拿。

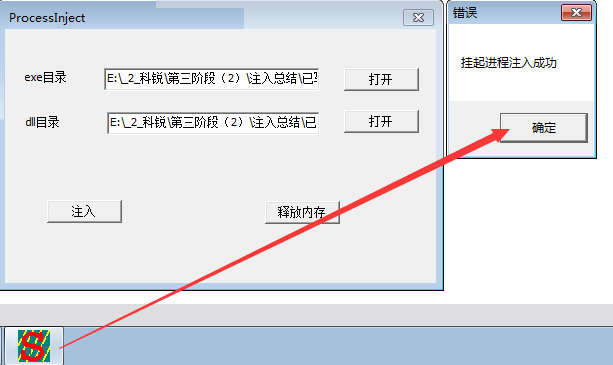
**0x01 挂起进程注入具体编程实现步骤**

由于这种方式和挂起线程注入方式的注入手段大同小异，所以只看区别部分

区别就在于使用的CreateProcess的API一口气实现了挂起主线程，拿到进程句柄，主线程句柄的操作，其他并无区别，详细请看附件代码

|  |
| --- |
| //以挂起的方式创建进程  bRet = CreateProcess(m\_strExePath.GetBuffer(0), NULL, NULL, NULL, FALSE, CREATE\_SUSPENDED,  NULL, NULL, &si, &pi);  if (!bRet)  {  MessageBox("CreateProcess 失败");  return;  }  g\_hProcess1 = pi.hProcess;  hThread = pi.hThread; |

实验效果:



在程序启动前，弹出我们自定义的MessgeBox，注入成功

# 四、调试器注入

**0x00 调试器注入的原理**

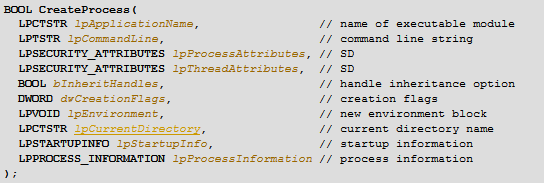
调试器注入个人理解是利用CreateProcess注入的另一种玩法，探究其本质也是挂起主线程注入，只不过跟调试器相关，CreateProcess可以与调试方式打开程序，在调试器开始以调试方式打开程序时第一个来到的调试事件是CREATE\_PROCESS\_DEBUG\_EVENT调试事件，在CREATE\_PROCESS\_DEBUG\_EVENT处理函数中，向目标程序中写入我们的ShellCode完成相应功能，例如LoadLibrary加载一个DLL，并且我们的ShellCode中写入以CC断点，代码指令时触发EXCEPTION\_DEBUG\_EVENT事件，我们在EXCEPTION\_DEBUG\_EVENT的处理函数中回到原来的执行流程。

写过调试器的同学应该很快就能明白，如果对调试器不理解的同学可以先去看雪上看看调试器的实现原理,这里就不再科普.

调试器原理相关文章

<http://bbs.pediy.com/thread-206292.htm>

参看MSDN，CreateProcess



CreateProcess的第六个参数使用了DEBUG\_ONLY\_THIS\_PROCESS，这意味着调用CreateProcess的进程成为了调试器，而它启动的子进程成了被调试的进程。除了DEBUG\_ONLY\_THIS\_PROCESS之外，还可以使用DEBUG\_PROCESS，两者的不同在于：DEBUG\_PROCESS会调试被调试进程以及它的所有子进程，而DEBUG\_ONLY\_THIS\_PROCESS只调试被调试进程，不调试它的子进程。一般情况下我们只想调试一个进程，所以应使用后者。

**实现思路:**

1. 构造ShellCode，ShellCode的最后一条指令为CC断点，用于触发EXCEPTION\_DEBUG\_EVENT
2. CreateProcess调试打开一个被注入进程，WaitForDebugEvent等待调试事件
3. CREATE\_PROCESS\_DEBUG\_EVENT是调试器收到的第一个调试事件，我们在这时候

在被注入进程中VirtualAllocEx申请空间，并且WriteProcessMemory写入代码，GetThreadContext获取上下文保存起来，SetThreadContext设置Eip为我们写入的ShellCode地址，执行ShellCode.

1. 因为ShellCode写入了CC断点，所以必然触发EXCEPTION\_DEBUG\_EVENT事件（注意先屏蔽系统断点），在这里我们利用SetThreadContext还原原来的Eip，调试器进程退出，让被注入程序正常跑起来。

**0x01 具体编程实现**

**用C++构造ShellCode略微要麻烦一点，因为要定义结构体，还要扣二进制，最好的办法就是用汇编，没关系，C++也是可以玩的。**

**➀ShellCode的构造用结构体构造，结构体对齐值设为1.**

|  |
| --- |
| //结构必须字节对齐!  #pragma pack(1)  typedef struct \_INJECT\_CODE  {  BYTE byMOV\_EAX; //mov eax, addr szDllpath  DWORD dwMOV\_EAX\_VALUE;  BYTE byPUSH\_EAX; //push eax  BYTE byMOV\_ECX; //mov ecx, LoadLibrary  DWORD dwMOV\_ECX\_VALUE;  WORD wCALL\_ECX; //call ecx  BYTE byINT3; //int 3  CHAR szDllPath[MAX\_PATH];  }INJECT\_CODE, \*PINJECT\_CODE;  #pragma pack() |

**➁以调试方式打开线程**

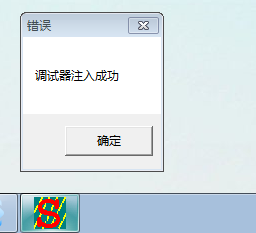
|  |
| --- |
| **bRet = CreateProcess(NULL,**  **m\_strExePath.GetBuffer(0),**  **NULL,**  **NULL,**  **FALSE,**  **DEBUG\_ONLY\_THIS\_PROCESS,**  **NULL,**  **NULL,**  **&si,**  **&pi);** |

**➂编写调试循环，我们只用到了两个事件，在CREATE\_PROCESS\_DEBUG\_EVENT事件处理函数中，申请内存空间，给ShellCode赋值,写入ShellCode，修改Eip执行ShellCode，在EXCEPTION\_DEBUG\_EVENT事件中恢复原来的Eip（注意屏蔽系统断点）**

|  |
| --- |
| while (WaitForDebugEvent(&dbgEvent, INFINITE))  {  switch(dbgEvent.dwDebugEventCode)  {  case CREATE\_PROCESS\_DEBUG\_EVENT:  hProcess = dbgEvent.u.CreateProcessInfo.hProcess;  hThread = dbgEvent.u.CreateProcessInfo.hThread;    //分配内存,填充注入指令  lpBaseAddress = VirtualAllocEx(hProcess,  NULL,  sizeof(INJECT\_CODE),  MEM\_COMMIT | MEM\_RESERVE,  PAGE\_EXECUTE\_READWRITE);  if (NULL == lpBaseAddress)  {  MessageBox("申请内存失败");  return;  }  //给ShellCode结构体赋值  ic.byMOV\_EAX = 0xB8;  ic.dwMOV\_EAX\_VALUE = (DWORD)lpBaseAddress +  offsetof(INJECT\_CODE, szDllPath);  ic.byPUSH\_EAX = 0x50;  ic.byMOV\_ECX = 0xB9;  ic.dwMOV\_ECX\_VALUE = (DWORD)&LoadLibrary;  ic.wCALL\_ECX = 0xD1FF;  ic.byINT3 = 0xCC;  memcpy(ic.szDllPath, m\_strDllPath.GetBuffer(0),m\_strDllPath.GetLength());  //写入ShellCode  bRet = WriteProcessMemory(hProcess, lpBaseAddress, &ic, sizeof(ic), NULL);  if (!bRet)  {  MessageBox("写入内存失败");  return;  }  //获取当前线程上下文  bRet = GetThreadContext(hThread, &ctxOld);  if (!bRet)  {  MessageBox("获取线程上下文失败");  return;  }  ctxNew = ctxOld;  ctxNew.Eip = (DWORD)lpBaseAddress;  bRet = SetThreadContext(hThread,&ctxNew);  if (!bRet)  {  MessageBox("设置线程上下文失败");  return;  }  break;  case EXCEPTION\_DEBUG\_EVENT:  if(dbgEvent.u.Exception.ExceptionRecord.ExceptionCode==EXCEPTION\_BREAKPOINT)  {  //屏蔽掉系统断点  if (bIsSystemBp)  {  bIsSystemBp = FALSE;  break;  }  //释放内存  bRet = VirtualFreeEx(hProcess,  lpBaseAddress,  0,  MEM\_RELEASE  );  if (!bRet)  {  MessageBox("获取线程上下文失败");  return;  }  //恢复到程序创建时的EIP  bRet = SetThreadContext(hThread, &ctxOld);  if (!bRet)  {  MessageBox("获取线程上下文失败");  return;  }  bRet = ContinueDebugEvent(dbgEvent.dwProcessId,dbgEvent.dwThreadId,DBG\_CONTINUE);  if (!bRet)  {  MessageBox("继续线程事件失败!!");  return;  }  //退出本进程，让被调试程序跑起来  ExitProcess(0);  return;    }  break;  }  bRet=ContinueDebugEvent(dbgEvent.dwProcessId,dbgEvent.dwThreadId,DBG\_EXCEPTION\_NOT\_HANDLED);  if (!bRet)  {  MessageBox("继续线程事件失败!!");  return;  }  } |

测试效果:

程序启动之前已弹出我们的注入对话框，注入成功。



# 五、注册表注入

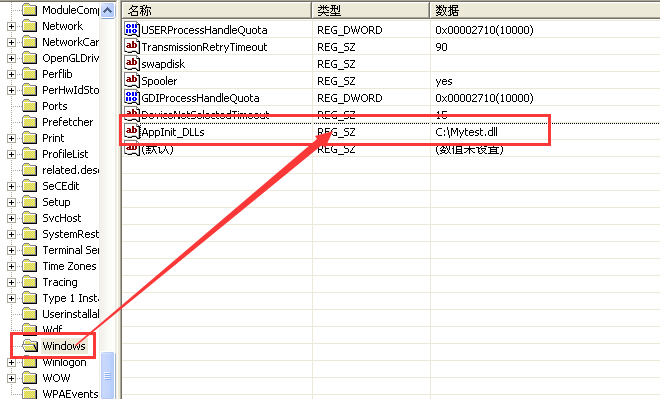
**0x00 注入思路**

注册表注入是比较简单易懂的一种注入方式，这种方法比较暴力，本人在win7 64位系统下测试的时候很多进程都能够注入，相当于全局的一个注入，这种方法比较其实就是修改了

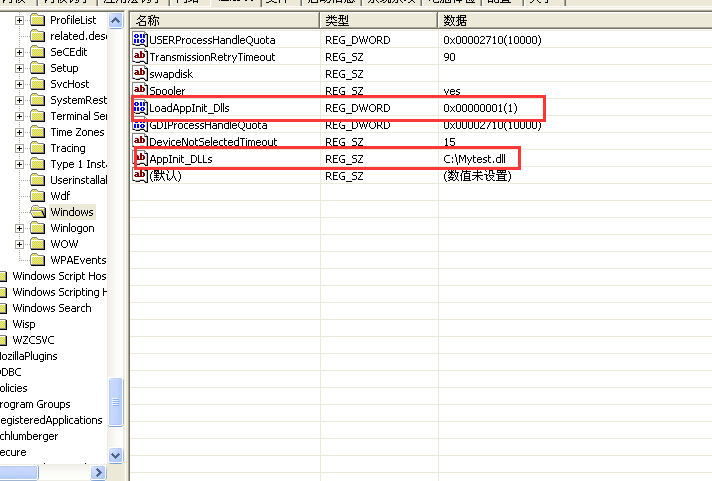
HKEY\_LOCAL\_MACHINE/Software/Microsoft/WindowsNT/CurrentVersion/Windows/AppInit\_DLLs的键值为我们DLL的路径，只要使用了user32.dll的程序都会加载这个目录下的DLL。

通过修改注册表

HKEY\_LOCAL\_MACHINE/Software/Microsoft/WindowsNT/CurrentVersion/Windows/AppInit\_DLLs的键值，添加我们的DLL路径如图



原理其实也比较简单，就是因为在进程中加载User32.dll的时候，这个键值下面的所有DLL都会被LoadLibrary函数加载到都会被加载到进程空间中来，这样久完成了DLL注入的效果



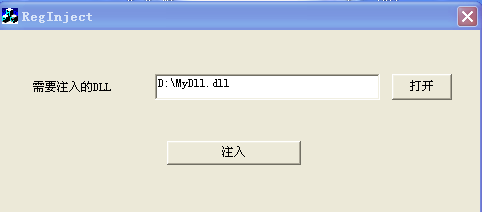
**0x01 代码实现**

**其实没必要用代码搞，手工几下就搞好了，但是其他的都用代码实现了，这个不实现心里不舒服。**

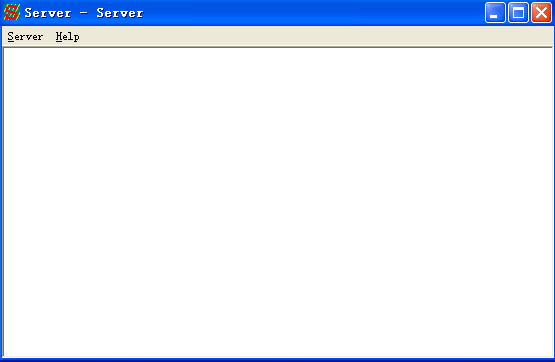
|  |
| --- |
| //打开HKEY\_LOCAL\_MACHINE/Software/Microsoft/WindowsNT/CurrentVersion/Windows  nReg = RegOpenKeyEx(HKEY\_LOCAL\_MACHINE,  "SOFTWARE\\Microsoft\\WindowsNT\\CurrentVersion\\Windows",  0,  KEY\_ALL\_ACCESS,  &hKey);  if (nReg != ERROR\_SUCCESS)  {  MessageBox("打开注册表失败");  RegCloseKey(hKey);  return;  }  //设置AppInit\_DLLs的键值为我们的Dll  nReg = RegSetValueEx(hKey,  "AppInit\_DLLs",  0,  REG\_SZ,  (byte \*)m\_strDllPath.GetBuffer(0),  m\_strDllPath.GetLength()  );  if(nReg != ERROR\_SUCCESS)  {  MessageBox("设置注册表失败！");  RegCloseKey(hKey);  return;  } |

实现效果

XP下打开注入工具，选择我们的测试DLL注入



打开测试测序



用PCHunter查看



可以看到我们的DLL成功注入

# 六、钩子注入

**0x00 钩子注入原理**

钩子注入又叫消息注入，就是利用一个Ring3API函数SetWindowsHookEx对消息进行拦截，在拦截消息进行处理的时候，需要在DLL中完成消息回调函数，SetWindowsHookEx会加载DLL，利用这个特性我们也就完成了DLL的注入。

**API介绍**

**HHOOK SetWindowsHookEx( int** *[idHook](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\winui.chm::/hh/winbase/)***,** // hook type

**HOOKPROC** *[lpfn](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\winui.chm::/hh/winbase/)***,** // hook procedure

**HINSTANCE** *[hMod](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\winui.chm::/hh/winbase/)***,** // handle to application instance

**DWORD** *[dwThreadId](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\winui.chm::/hh/winbase/)* // thread identifier**);**

参数介绍:

➀[idHook](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\winui.chm::/hh/winbase/):安装钩子的类型。我们这里使用WH\_CALLWNDPROC表示拦截发送到窗口过程函数的消息。

➁[lpfn](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\winui.chm::/hh/winbase/): 挂钩回调函数。

➂[hMod](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\winui.chm::/hh/winbase/)：这个句柄指向包含有钩子回调函数的DLL的句柄。

④[dwThreadId](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\winui.chm::/hh/winbase/):要注入的线程ID。

注意:消息钩子又分为局部消息钩子和全局消息钩子，全局消息钩子只用把SetWindowsHookEx的第四个参数设置为NULL就可以了，表示所有的程序都下钩子。这里不做实验，感兴趣的可以自己测试。

**解钩子函数**

BOOL UnhookWindowsHookEx( HHOOK [hhk](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\winui.chm::/hh/winbase/) // handle to hook procedure);

钩子注入的方法比较简单，下面我们就来进行编程实验。

**0x01 编程实现钩子注入**

在主程序中调用SetWindowsHookEx为目标程序安装一个键盘钩子，在注入的DLL中编写回调函数，在回调函数中加载想要注入的DLL

**➀注入程序：一个挂钩一个解钩**



**➁安装钩子函数，调用DLL中的SetHook函数**

|  |
| --- |
| void CHookInjectDlg::OnHook()  {  // TODO: Add your control notification handler code here  //加载待注入的DLL  g\_hDll = LoadLibrary("InjectDll.dll");  if (g\_hDll != NULL)  {  g\_pfnSetHook = (LPFUN2)GetProcAddress(g\_hDll,"SetHook");  g\_pfnUnHook = (LPFUN)GetProcAddress(g\_hDll,"UnHook");  }  else  {  MessageBox("加载DLL失败！");  return;  }  GetDlgItemText(IDC\_EDIT1,m\_strExeName);  //安装钩子函数  if (g\_pfnSetHook != NULL)  {  g\_pfnSetHook(m\_strExeName.GetBuffer(0),m\_strDllPath.GetBuffer(0));  }  else  {  MessageBox("安装钩子失败！");  return;  }  } |

**➂卸载钩子函数，调用注入DLL中的OnUnHook**

|  |
| --- |
| void CHookInjectDlg::OnUnHook()  {  // TODO: Add your control notification handler code here  if (g\_hDll != NULL)  {  //卸载钩子函数  g\_pfnUnHook();  //抹掉DLL  FreeLibrary(g\_hDll);  }  } |

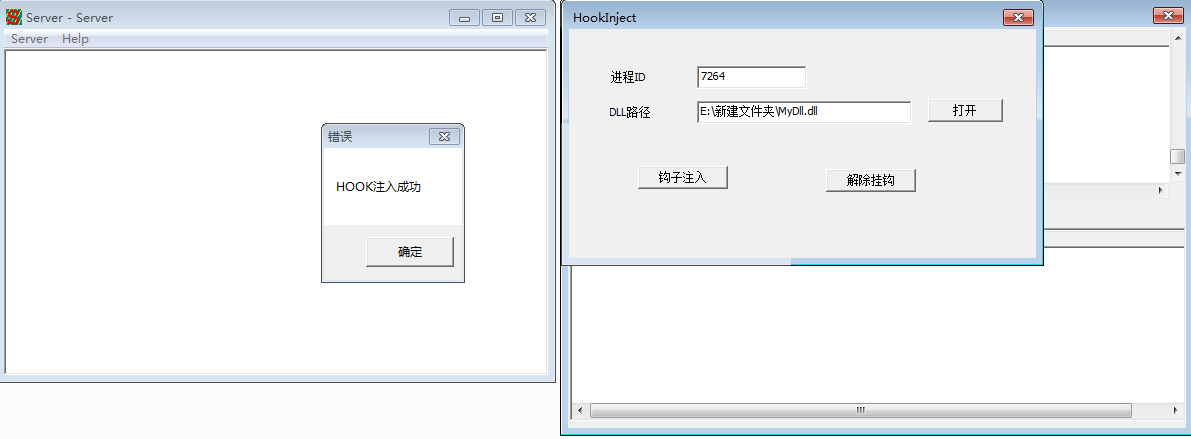
**④DLL中函数的编写，SetHook完成找到计算器安装钩子的功能，UnHook完成了UnhookWindowsHookEx的功能**

|  |
| --- |
| //钩子回调函数  LRESULT CALLBACK HookProc(  int code, // hook code  WPARAM wParam, // virtual-key code  LPARAM lParam // keystroke-message information  )  {  MessageBox(NULL,"KEY PRESS","钩子注入成功",MB\_OK);  return 1;  }  EXPORTFUN void SetHook()  {  DWORD dwTid = -1;  //获取计算器窗口句柄  g\_hCalc = FindWindowEx(NULL,NULL,NULL,"计算器");  if (g\_hCalc == 0)  {  return;  }    //获取目标主线程  dwTid = GetWindowThreadProcessId(g\_hCalc,NULL);    if (dwTid != -1)  {  //安装键盘钩子  g\_hHook= SetWindowsHookExA(WH\_KEYBOARD,HookProc,GetModuleHandle("InjectDll.dll"),dwTid);  }  }  EXPORTFUN void UnHook()  {  UnhookWindowsHookEx(g\_hHook);  } |

详细见程序源码

实验效果:

按下任意键弹出MessageBox，注入成功



# 七、APC注入

本人也是刚刚学习APC相关知识，所以做一个详细记录。（以下内容摘抄自百度）

**0x01 APC对象及其原理**

APC(Asynchronous procedure call)异步程序调用，

在NT中，有两种类型的APCs：用户模式和内核模式。用户APCs运行在用户模式下目标线程当前上下文中，并且需要从目标线程得到许可来运行。特别是，用户模式的APCs需要目标线程处在alertable等待状态才能被成功的调度执行。通过调用下面任意一个函数，都可以让线程进入这种状态。这些函数是：KeWaitForSingleObject, KeWaitForMultipleObjects, KeWaitForMutexObject, KeDelayExecutionThread。  
对于用户模式下，可以调用函数SleepEx, SignalObjectAndWait, WaitForSingleObjectEx, WaitForMultipleObjectsEx,MsgWaitForMultipleObjectsEx 都可以使目标线程处于alertable等待状态，从而让用户模式APCs执行,原因是这些函数最终都是调用了内核中的KeWaitForSingleObject,KeWaitForMultipleObjects,KeWaitForMutexObject, KeDelayExecutionThread等函数。另外通过调用一个未公开的alert-test服务KeTestAlertThread，用户线程可以使用户模式APCs执行。

当一个用户模式APC被投递到一个线程，调用上面的等待函数，如果返回等待状态STATUS\_USER\_APC，在返回用户模式时，内核转去控制APC例程，当APC例程完成后，再继续线程的执行.

上面一大堆内容的意思就是，当进程某个线程调用函数SleepEx, SignalObjectAndWait, WaitForSingleObjectEx, WaitForMultipleObjectsEx,MsgWaitForMultipleObjectsEx 这些函数时候，会让执行的线程中断，我们需要利用QueueUserAPC()在线程中断的时间内向APC中插入一个函数指针，当线程苏醒的时候APC队列里面这个函数指针就会被执行，我们在APC队列中插入LoadLibrary函数就可以完成DLL注入的工作。

**编程实现思路:**

1. 我们用CreateProcess以挂起的方式打开目标进程。
2. WriteProcessMemory向目标进程中申请空间，写入DLL名称。
3. 使用QueueUserAPC()这个API向队列中插入Loadlibrary()的函数指针，加载我们的DLL

**API介绍**

|  |
| --- |
| DWORD QueueUserAPC( PAPCFUNC *[pfnAPC](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\dllproc.chm::/hh/winbase/)*, // APC function  HANDLE *[hThread](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\dllproc.chm::/hh/winbase/)*, // handle to thread  ULONG\_PTR *[dwData](mk:@MSITStore:D:\\_1_各种软件安装包\\_1_编程相关\\MSDN\\2001OCT\\1033\\dllproc.chm::/hh/winbase/)* // APC function parameter); |

参数1：APC回调函数地址;

参数2：线程句柄

参数3：回调函数的参数

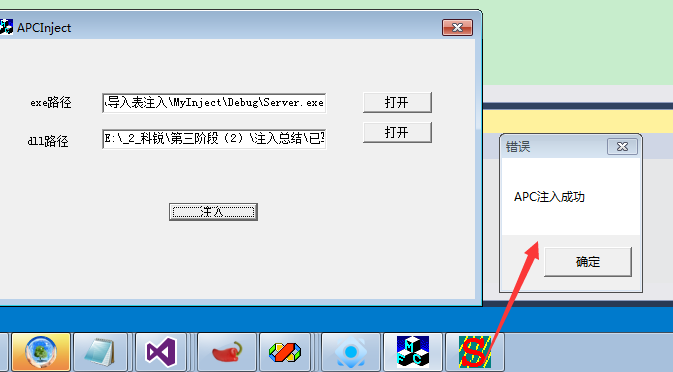
**0x01 编程实现**

**思路上面已经将清了**

|  |
| --- |
| void CAPCInjectDlg::OnInject()  {  // TODO: Add your control notification handler code here  DWORD dwRet = 0;  PROCESS\_INFORMATION pi;  STARTUPINFO si;  ZeroMemory(&pi,sizeof(pi));  ZeroMemory(&si,sizeof(si));  si.cb = sizeof(STARTUPINFO);    //以挂起的方式创建进程  dwRet = CreateProcess(m\_strExePath.GetBuffer(0),  NULL,  NULL,  NULL,  FALSE,  CREATE\_SUSPENDED,  NULL,  NULL,  &si,  &pi);    if (!dwRet)  {  MessageBox("CreateProcess失败！！");  return;  }  PVOID lpDllName = VirtualAllocEx(pi.hProcess,  NULL,  m\_strDllPath.GetLength(),  MEM\_COMMIT,  PAGE\_READWRITE);    if (lpDllName)  {  //将DLL路径写入目标进程空间  if(WriteProcessMemory(pi.hProcess, lpDllName, m\_strDllPath.GetBuffer(0),m\_strDllPath.GetLength(), NULL))  {  LPVOID nLoadLibrary=(LPVOID)GetProcAddress(GetModuleHandle("kernel32.dll"),"LoadLibraryA");  //向远程APC队列插入LoadLibraryA  if(!QueueUserAPC((PAPCFUNC)nLoadLibrary,pi.hThread,(DWORD)lpDllName))  {  MessageBox("QueueUserAPC失败！！");  return;  }  }  else  {  MessageBox("WriteProcessMemory失败！！");  return;  }  }    //恢复主线程  ResumeThread(pi.hThread);  MessageBox("APC注入成功");  } |

**实验效果:**

目标程序启动之前DLL注入成功



# 八、远程线程注入

**0x00 远程线程注入原理**

这是一种玩的比较多的注入方式，网上的又很多源码，很多例子，固定的套路。

主要是用到CreateRemoteThread这个API，通过它可以打开目标进程中的远程线程，然后跑我们的代码完成注入。

**API介绍**

|  |
| --- |
| HANDLE WINAPI CreateRemoteThread(       HANDLE hProcess,       LPSECURITY\_ATTRIBUTES lpThreadAttributes,       SIZE\_T dwStackSize,       LPTHREAD\_START\_ROUTINE lpStartAddress,       LPVOID lpParameter,       DWORD dwCreationFlags,       LPDWORD lpThreadId       ); |

参数1：目标进程的句柄

参数2：一个指向 [SECURITY\_ATTRIBUTES](http://baike.baidu.com/item/SECURITY_ATTRIBUTES" \t "http://baike.baidu.com/_blank) 结构的指针, 该结指定了[线程](http://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/103101" \t "http://baike.baidu.com/_blank)的安全属性

参数3：[线程](http://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/103101" \t "http://baike.baidu.com/_blank)栈初始大小,以字节为单位,如果该值设为0,那么使用系统默认大小.

参数4：远程线程的线程函数的执行的起始地址.

参数5：传给线程函数的参数.

参数6：线程的创建标志.

参数7：传出参数，传出线程ID

具体实现思路:

1.OpenProcess打开远程线程

2.VirtualAllocEx在远程中申请空间中申请空间，WriteProcessMemory写入要注入的DLL的路径

3.GetProcAddress获取目标进程中的LoadLibraryA()的函数地址（这里利用的是Kernel32.dll在每个进程中地址一样的巧合）

4.创建远程线程CreateRemoteThread，用目标进程中的LoadLibraryA()加载需要注入的DLL

5.WaitForSingleObject等待信号量

6.VirtualFreeEx释放远程线程中申请的空间

**0x01 远程线程注入编程实现**

**➀打开远程进程**

|  |
| --- |
| hProcess = OpenProcess(PROCESS\_ALL\_ACCESS,FALSE,dwProcessId);  if (hProcess == NULL)  {  MessageBox("打开进程失败!!!!");  return;  } |

**➁VirtualAllocEx在远程中申请空间中申请空间，WriteProcessMemory写入要注入的DLL的路径**

|  |
| --- |
| //1.在远程进程中分配内存  pszRemoteBuffer=(char\*)VirtualAllocEx(hProcess,NULL,nLibFileLen,MEM\_COMMIT,PAGE\_READWRITE);  if (pszRemoteBuffer == NULL)  {  MessageBox("申请远程空间失败");  return;  }  //2.在远程申请的地址当中写入DLL的路径  char \* szDLLPath = m\_filePath.GetBuffer(nLibFileLen);  SIZE\_T dwWriten;  if (!WriteProcessMemory(hProcess,pszRemoteBuffer,(LPVOID)szDLLPath,nLibFileLen,&dwWriten))  {  MessageBox("写入内存失败");  } |

➂获取远程进程中LaodLibry的地址

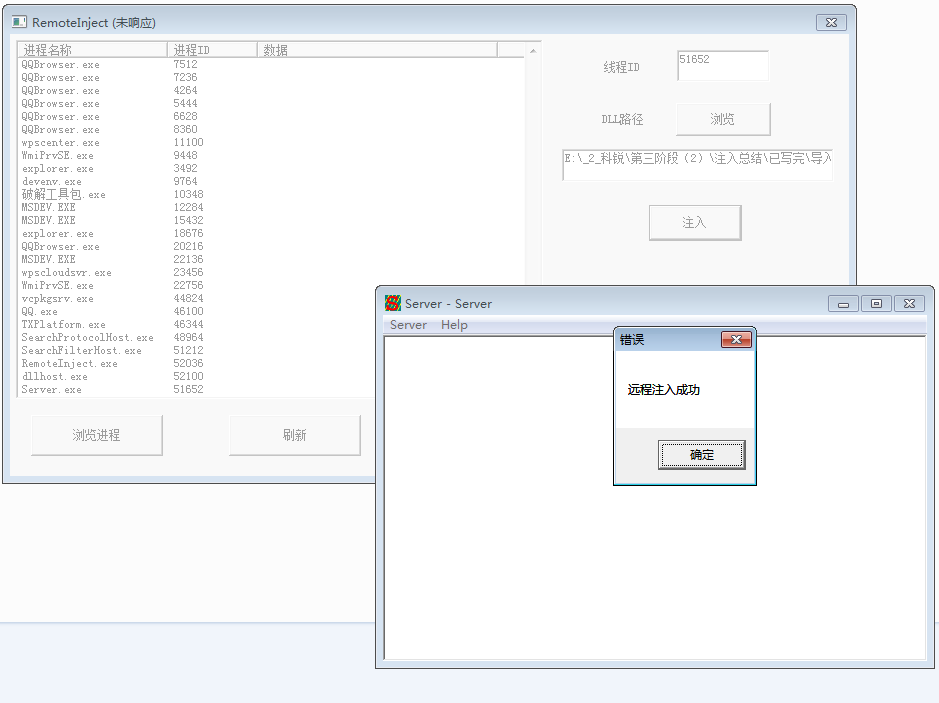
|  |
| --- |
| //3.获取远程进程中LaodLibry的地址,这里你用的巧合是每个程序中的kernel32的地址的都一样，远程中也一样在  HMODULE hMouDle = GetModuleHandle("Kernel32");  PTHREAD\_START\_ROUTINE pfnLoadLibrary = (PTHREAD\_START\_ROUTINE)GetProcAddress(GetModuleHandle("Kernel32"),"LoadLibraryA");  if (pfnLoadLibrary == NULL)  {  MessageBox("获取LoadLibrary地址失败！！！");  return;  } |

④创建远程线程，等待远程线程执行完

|  |
| --- |
| //4.创建远程线程  hThread = CreateRemoteThread(hProcess,NULL,0,pfnLoadLibrary,pszRemoteBuffer,0,NULL);  DWORD dwErrCode = GetLastError();  if (hThread == NULL)  {  MessageBox("创建远程线程失败");  return;  }    WaitForSingleObject(hThread,INFINITE); |

其余还有一些遍历进程的代码详细见源代码

**实验效果:**

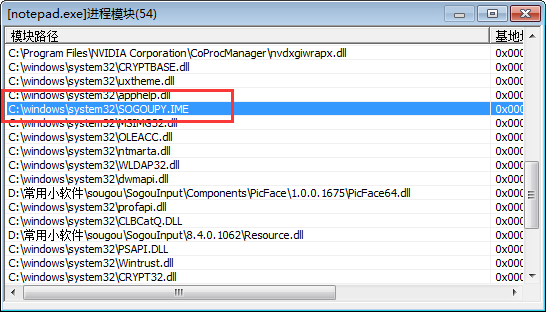


# 

# 九、输入法注入

输入法注入算是一种比较麻烦的注入方式吧，就是一堆API加上固定的套路完成说的注入方式。首先输入法的原理就是向进程中注入一个ime的输入法文件，这个ime文件本质上就是一个DLL，这种机制和DLL劫持有几分相似，都是利用构造系统信任的DLL执行我们DLL或者我们的代码。

我们打开记事本用PCHunter观察它的所有模块



可以看到这里搜狗输入法也是以Ime的形式存在的。

我们注入的思路:

自己编写Ime，然后在Ime中加载我们要注入的Dll完成注入。

这就涉及到输入法编程了，这里就不多提了，我在网上找了一些资料学习，附上链接:

http://www.cnblogs.com/freedomshe/archive/2012/11/30/ime\_learning.html

http://blog.csdn.net/pkfish/article/details/7339909

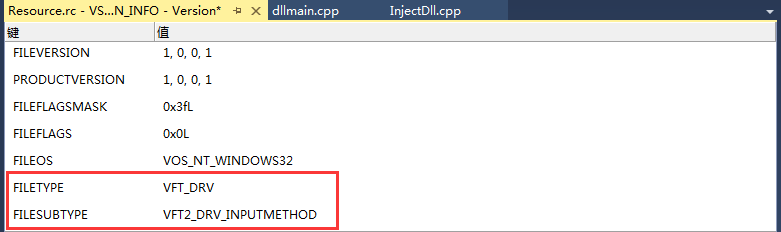
**0x01 具体实现思路:**

**➀编写输入法Ime文件，其实就是DLL**

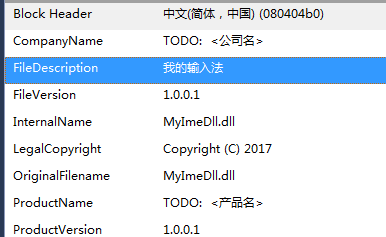
（1）创建一个DLL工程，给DLL添加一个.RC资源文件，再添加一个Version资源如图。



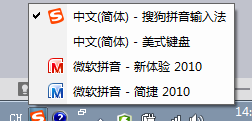
修改FILETYPE，FILESUBTYPE为如图的值



修改我们输入法的在布局中的名字FileDescription



**➁SystemParametersInfo保存原始的键盘布局，方便后面还原布局，布局如下图**



|  |
| --- |
| //1.保存原始的键盘布局，方便后面还原  SystemParametersInfo(SPI\_GETDEFAULTINPUTLANG, 0, &HklOldInput,0); |

**➂CopyFile复制Ime文件和需要注入的DLL到C:\\WINDOWS\\SYSTEM32\\目录下**

**④ImmInstallIME安装输入法**

|  |
| --- |
| //3.加载输入法  HklNewInput = ImmInstallIME("C:\\WINDOWS\\SYSTEM32\\MyImeDll.ime", "我的输入法"); |

**⑤激活输入法键盘布局,获取被激活的输入法注册表项用于后面清除**

|  |
| --- |
| ActivateKeyboardLayout(HklNewInput, 0);  GetKeyboardLayoutName(ImeSymbol); |

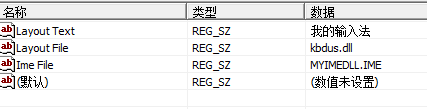
**⑥根据窗口名激活指定窗口的输入法**

|  |
| --- |
| **//7.获取串口句柄**  **hWnd = GetHwndByProcessId(m\_dwPID);**  **//8.激活指定窗口的输入法**  **if (HklNewInput != NULL)**  **{**  **::PostMessage(hWnd, WM\_INPUTLANGCHANGEREQUEST, 0x1, (LPARAM)HklNewInput);**  **}** |

**⑦还原键盘布局，还原顶层窗口的键盘布局，卸载输入法**

|  |
| --- |
| //9.还原键盘布局  SystemParametersInfo(SPI\_SETDEFAULTINPUTLANG,0,&HklOldInput,SPIF\_SENDWININICHANGE);  //10.顶层窗口换回去，不换回去后面卸载不到，因为ime文件占用  do  {  hTopWnd = ::FindWindowEx(NULL, hTopWnd, NULL, NULL);  if (hTopWnd != NULL)  {  ::PostMessage(hTopWnd,WM\_INPUTLANGCHANGEREQUEST,0x1,(LPARAM)&HklOldInput);  }  } while (hTopWnd == NULL);  //11.卸载输入法  if (HklNewInput != NULL)  {  UnloadKeyboardLayout(HklNewInput);  } |

**⑨因为编写输入法注入的时候会在HKEY\_LOCAL\_MACHINE\\System\\CurrentControlSet\\Control\\Keyboard Layouts目录下写入注册表项如图:**

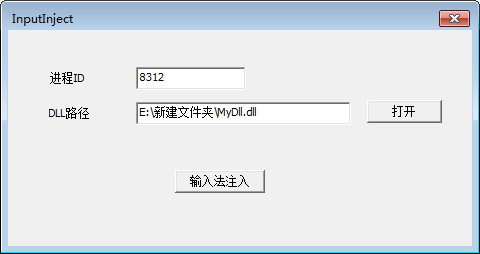


**⑩清除注册表项，删除输入法文件**

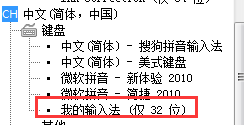
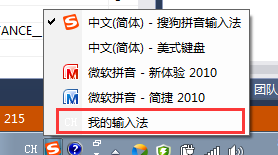
|  |
| --- |
| //12.删除注册表项，清理痕迹  //打开注册表项目，获取句柄  RegOpenKey(HKEY\_CURRENT\_USER, "Keyboard Layout\\Preload", &phkResult);  //枚举注册表项目  while (RegEnumValue(phkResult, i, ValueName, &lenValue, NULL, NULL, (LPBYTE)lpData, &lenData) != ERROR\_NO\_MORE\_ITEMS)  {  if (lenData != 0)  {  if (strcmp(ImeSymbol, lpData) == 0)  {  //删除项目数值  RegDeleteValue(phkResult, ValueName);  break;  }  }  memset(lpData, 0, MAX\_PATH);  memset(ValueName, 0, MAX\_PATH);  lenValue = MAX\_PATH;  lenData = MAX\_PATH;  i++;  }  **// 删除输入法文件**  **DeleteFile("C:\\WINDOWS\\SYSTEM32\\MyImeDll.ime");** |

**0x02 实验效果**

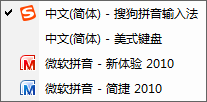
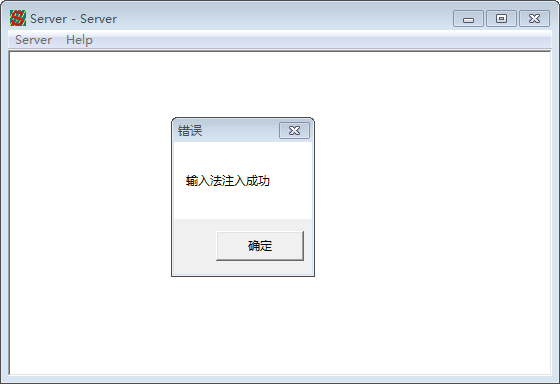
填入我们要注入的DLL和要被注入的进程ID

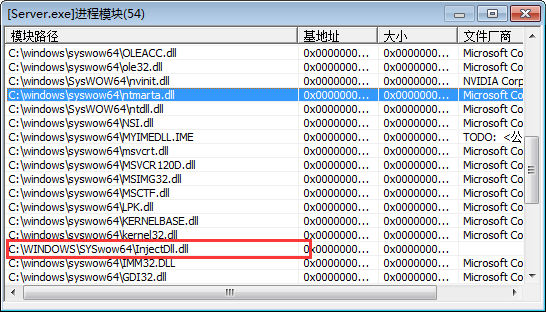


当运行完ImmInstallIME时候，我们的输入法就安装成功了



**运行完毕后注入成功，我们的输入法也自动清除**





# 十、DLL劫持

**0x00 DLL劫持的原理简介**

在一个应用程序加载DLL时，Windows会按照一定的顺序去系统中搜索指定的DLL，这个顺序称之为DLL的搜索顺序。

**标准的DLL搜索顺序：**

1.应用程序的加载目录。

2.当前目录（默认为程序加载目录，可以通过SetCurrentDirectory修改，通过GetCurrentDirectory获取）

3.系统目录（32位系统下通常是，C:\Windows\System32，可以通过GetSystemDirectory获取）

4.Windows目录（通常是，C:\Windows，可以通过GetWindowsDirectory获取）

5.PATH环境变量中列出的所有路径

利用系统的这个特性，就可以使应用程序强制加载我们指定的DLL做一些特殊的工作。

**实现过程:**

1.构造一个与要劫持DLL相同的导出表

2.在DLL中加载原DLL

3.在加载过程中执行我们的代码

4.转发导出函数到原来的DLL上，调用被劫持DLL的导出函数。

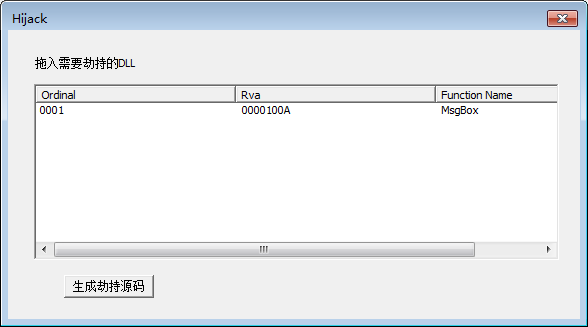
0x01 编程实现

**➀我们需要劫持的DLL为MyDll.dll,功能很简单，只有一个导出函数，功能为弹出一个MessageBox，源码如下:**

|  |
| --- |
| #include "stdafx.h"  #define EXPORT \_\_declspec(dllexport)  BOOL APIENTRY DllMain( HANDLE hModule,  DWORD ul\_reason\_for\_call,  LPVOID lpReserved  )  {  switch (ul\_reason\_for\_call)  {  case DLL\_PROCESS\_ATTACH:  break;  }  return TRUE;  }  extern "C" EXPORT void MsgBox()  {  MessageBox(NULL,"我是被劫持的程序",NULL,MB\_OK);  } |

**➁劫持DLL的编写，以下DLL劫持函数是通过DLL劫持工具生成的，稍加修改就可以使用**

**拖入DLL生成劫持代码，生成劫持DLL**



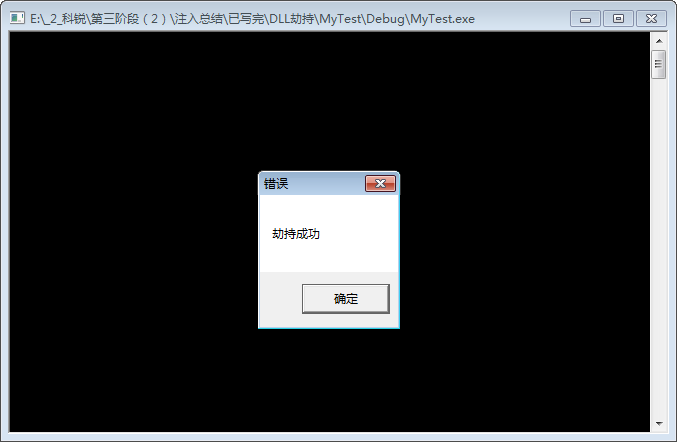
|  |
| --- |
| // 导出函数  #pragma comment(linker, "/EXPORT:MsgBox=\_My\_MsgBox,@1")  // 宏定义  #define NAKED \_\_declspec(naked)  #define EXPORT \_\_declspec(dllexport)  #define MYEXPROT EXPORT NAKED  //全局变量  HMODULE hDll = NULL;  DWORD dwRetaddress[2]; //存放返回地址  // 内部函数 获取真实函数地址  FARPROC WINAPI GetAddress(PCSTR pwcProcName)  {  FARPROC fpAddress;  WCHAR wcTemp[MAX\_PATH] = { 0 };  fpAddress = GetProcAddress(hDll, pwcProcName);  if (fpAddress == NULL)  {  sprintf\_s(wcTemp, MAX\_PATH, "无法找到函数 :%s 的地址 ", pwcProcName);  MessageBoxA(NULL, wcTemp, "错误", MB\_OK);  ExitProcess(-2);  }  //返回真实地址  return fpAddress;  }  // DLL MAIN  BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved)  {  switch (fdwReason)  {  case DLL\_PROCESS\_ATTACH:  hDll = LoadLibrary("MyDll.dll");  if (!hDll)  {  MessageBox(NULL,"要劫持的DLL不存在",NULL,NULL);  return FALSE;  }  MessageBox(NULL,"劫持成功",NULL,NULL);  break;  case DLL\_PROCESS\_DETACH:  if (hDll != NULL)  FreeLibrary(hDll);  break;  }  return TRUE;  }  //导出函数 1 不要在里边定义变量  MYEXPROT My\_MsgBox()  {  GetAddress("MsgBox");  //栈顶为保存函数的返回地址  \_\_asm pop dwRetaddress[1]  //Call原来的函数  \_\_asm call eax  //跳回原来的返回地地址  \_\_asm jmp dword ptr dwRetaddress[1]  } |

**➂编写测试程序**

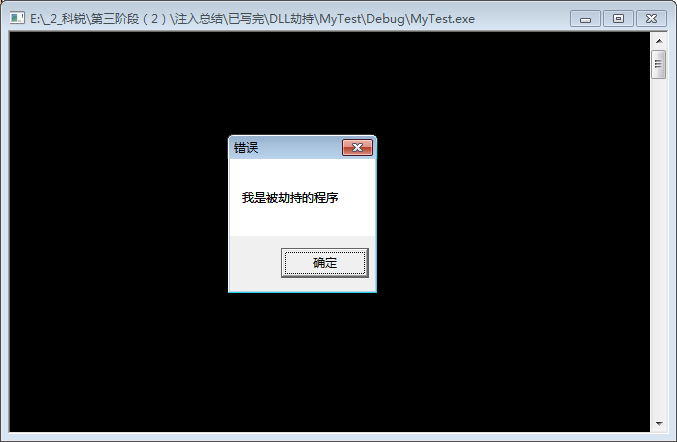
|  |
| --- |
| #include <windows.h>  //注意这里是\_\_stdcall  typedef FARPROC (\_\_stdcall \*PFUN)();  PFUN g\_pMyMsg = NULL;  int main(int argc, char\* argv[])  {  HMODULE hMod = LoadLibrary("MyDll.DLL");    g\_pMyMsg= (PFUN)GetProcAddress(hMod,"MsgBox");    (\*g\_pMyMsg)();    return 0;  } |

**实验效果:**

在弹出正常对话框之前首先弹出劫持DLL中的对话框



接着才是被劫持DLL中的对话框，完成注入



# Ring3注入总结

**➀导入表注入**

静态注入的方法。修改PE文件，添加一个新节，修改导入表添加一个新的DLL实现注入。

**➁挂起线程注入**

OpenThread-->SuspendThread-->GetThreadContext-->获取EIP-->修改EIP-->SetThreadContext-->ResumeThread

**➂挂起进程注入**

CreateProcess注入方法之一，CREATE\_SUSPENDED以挂起的方式打开进程，写入ShellCode注入。

**④调试器注入**

CreateProcess注入方法之二，DEBUG\_ONLY\_THIS\_PROCESS以调试的方法打开进程，利用CREATE\_PROCESS\_DEBUG\_EVENT，向目标程序中写入我们的ShellCode完成相应功能，并且我们的ShellCode中写入以CC断点，代码指令时触发EXCEPTION\_DEBUG\_EVENT事件，在EXCEPTION\_DEBUG\_EVENT的处理函数中回到原来的执行流程。

**⑤注册表注入**

全局的注入方式：

修改：

HKEY\_LOCAL\_MACHINE/Software/Microsoft/WindowsNT/CurrentVersion/Windows/AppInit\_DLLs的键值为我们DLL的路径，只要使用了user32.dll的程序都会加载这个目录下的DLL。

**⑥钩子注入**

利用SetWindowsHookEx拦截消息进行注入。

**⑦APC注入**

APC注入的原理是利用当线程被唤醒时APC中的注册函数会被执行的机制，并以此去执行我们的DLL加载代码，进而完成DLL注入的目的。利用QueueUserAPC()可以向APC队列投入Loadlibrary函数指针完成注入，其实这种方法配合CreateProcess使用注入最为简单，先挂起打开线程，再QueueUserAPC()，再恢复线程，完成注入。

**⑧远程线程注入**

老的套路，主要是用到CreateRemoteThread这个API，通过它可以打开目标进程中的远程线程，然后跑我们的代码完成注入。

**⑨输入法注入**

利用输入法在工作时需要向进程中加载Ime文件（其实就是个Dll）,我们构造自己的Ime文件，在Ime文件注入对方进程的时候加载我们自己的DLL完成注入

**⑩DLL劫持**

简单来说就是自己实现应用程序的某些DLL，完成导出函数，替换DLL实现注入。

# 关于Ring3下的反注入思路

反注入的方法大牛应该都是在Ring0下面玩，我只了解一些三环下的反注入思路，这里只聊3环下的反注入思路。

➀关于导入表注入，毕竟是静态修改文件的方法注入DLL，可以对自身PE文件求校验值判断是否被修改。

➁钩子注入

(1)HOOK自身进程的LoadLibraryExW这个函数，判断调用是否来自user32.dll,因为钩子注入时LoadLibraryExW的调用者为user32.dll，HOOK关键代码如下

|  |
| --- |
| HMODULE WINAPI newLoadLibraryExW(LPCWSTR lpLibFileName,HANDLE hFile,DWORD dwFlags)  {  //get the return address  DWORD dwCaller;  //ebp+4返回上层调用者的地址  \_\_asm push dword ptr [ebp+4]  \_\_asm pop dword ptr [dwCaller]  if(dwCaller > m\_dwUser32Start && dwCaller < m\_dwUser32End)  {  return FALSE;  }  return rawLoadLibraryExW(lpLibFileName,hFile,dwFlags);  } |

1. 安装WH\_DEBUG消息钩子，在WH\_DEBUG钩子的消息回到中屏蔽消息钩子，回调

|  |
| --- |
| LRESULT CALLBACK DebugProc(     int nCode,       WPARAM wParam,       LPARAM lParam  ) |

其中第三个参数 lParam  指向DEBUGHOOKINFO结构体如下:

|  |
| --- |
| typedef struct  {       DWORD idThread;  //安装WH\_DEBUG钩子的线程ID。       DWORD idThreadInstaller;  //当前即将被调用的钩子所在的线程ID。       LPARAM lParam;       WPARAM wParam;       int code;  } DEBUGHOOKINFO, \*PDEBUGHOOKINFO; |

判断这两个idThread与idThreadInstaller是否相等即可判断是自己进程的钩子

**➂远程线程注入**

当程序被远程线程被注入时候，线程的入口点一定为LoadLibraryA或者LoadLibraryW，我们用判断线程的入口点是不是LoadLibraryA或者LoadLibraryW，是就干掉。拿到线程入口点的方法是通过为公开的API --GetThreadStartAddress

关键代码如下:

|  |
| --- |
| typedef enum \_THREADINFOCLASS {  ThreadBasicInformation,  ThreadTimes,  ThreadPriority,  ThreadBasePriority,  ThreadAffinityMask,  ThreadImpersonationToken,  ThreadDescriptorTableEntry,  ThreadEnableAlignmentFaultFixup,  ThreadEventPair\_Reusable,  ThreadQuerySetWin32StartAddress,  ThreadZeroTlsCell,  ThreadPerformanceCount,  ThreadAmILastThread,  ThreadIdealProcessor,  ThreadPriorityBoost,  ThreadSetTlsArrayAddress,  ThreadIsIoPending,  ThreadHideFromDebugger,  ThreadBreakOnTermination,  MaxThreadInfoClass  } THREADINFOCLASS;  typedef LONG (WINAPI \*NtQueryInformationThreadProc)(  \_In\_ HANDLE ThreadHandle,  \_In\_ THREADINFOCLASS ThreadInformationClass,  \_Inout\_ PVOID ThreadInformation,  \_In\_ ULONG ThreadInformationLength,  \_Out\_opt\_ PULONG ReturnLength  );  NtQueryInformationThreadProc NtQueryInformationThread = NULL;  hNtdll = GetModuleHandleW(L"ntdll.dll");  NtQueryInformationThread = (NtQueryInformationThreadProc)GetProcAddress(hNtdll, "NtQueryInformationThread");  HANDLE hThread = NULL;  PVOID pvStart = NULL;  hThread = OpenThread(THREAD\_QUERY\_INFORMATION | THREAD\_TERMINATE, FALSE, te32.th32ThreadID);  NtQueryInformationThread(hThread, ThreadQuerySetWin32StartAddress, &pvStart, sizeof(pvStart), NULL); |

**④DLL劫持**

DLL的防御姿势我能想到就是对自己进程每个要加载的DLL求一个校验值，程序运行过程中比对。

学习注入参考：

《Windows核心编程》

《加密与解密》

《[Ring3/Ring0下注入方法](http://blog.csdn.net/sr0ad/article/details/8184586)》

<http://blog.csdn.net/sr0ad/article/details/8184586>

《[简单总结下常用的注入姿势](http://blog.csdn.net/u013761036/article/details/53967920)》

http://blog.csdn.net/u013761036/article/details/53967920

《应用层反外挂技术研究》

http://bbs.pediy.com/thread-173897.htm