Ring3 注入总结及编程实现

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。

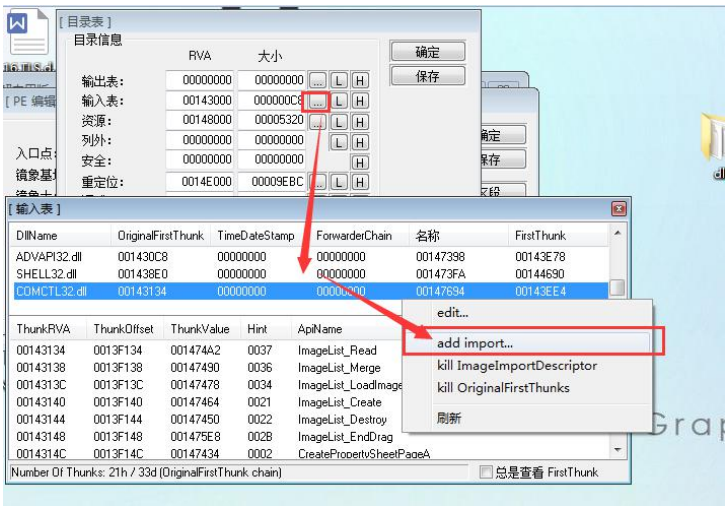
注入的 DLL 源码:

```
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;
}
```

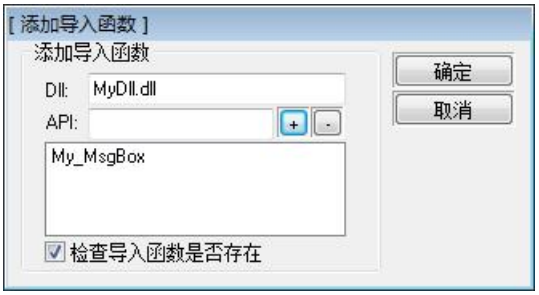
```
EXTERN_C EXPORT My_MsgBox()
{
	//在这里添加需要执行的代码，不要破坏堆栈平衡

	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

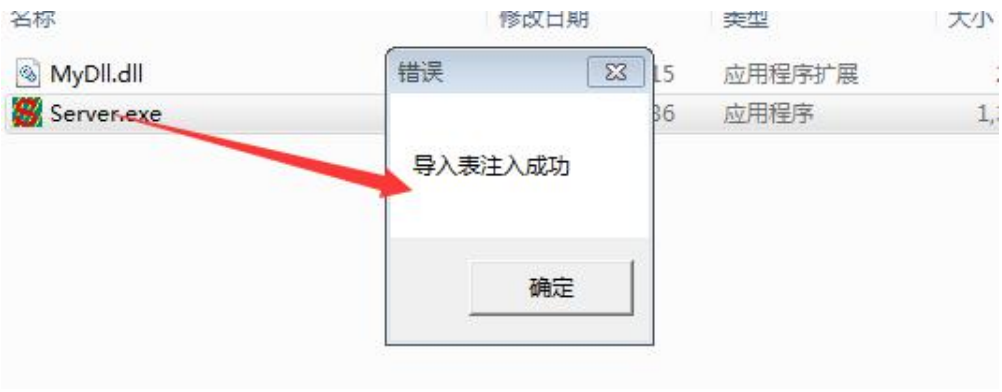
打开 LordPe 的 PE 编辑功能。如下如点击。



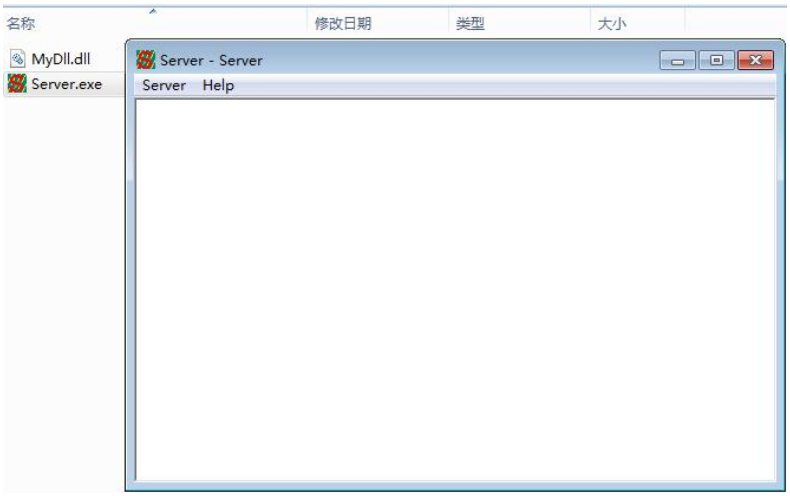
添加 DLL 名和导出的函数，点击确定完成修改



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

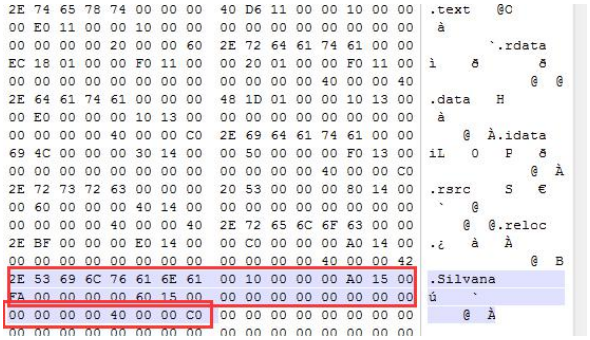


点击确定，弹出了正常的的服务器程序，说明导入表注入成功。



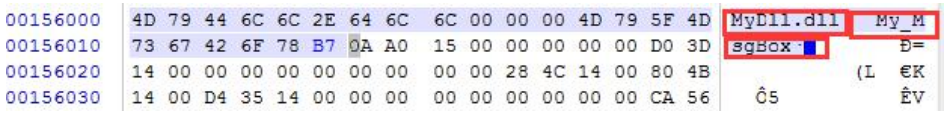
用 Winhex 观察 Pe 文件，对比看看 LordPe 对 Pe 文件做了啥。

①首先填加一个节，如图

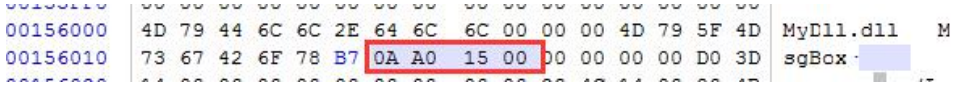


②在节中写入 DLL 名和函数名（这里其实写入的是 IMAGE_IMPORT_BY_NAME 结构体）

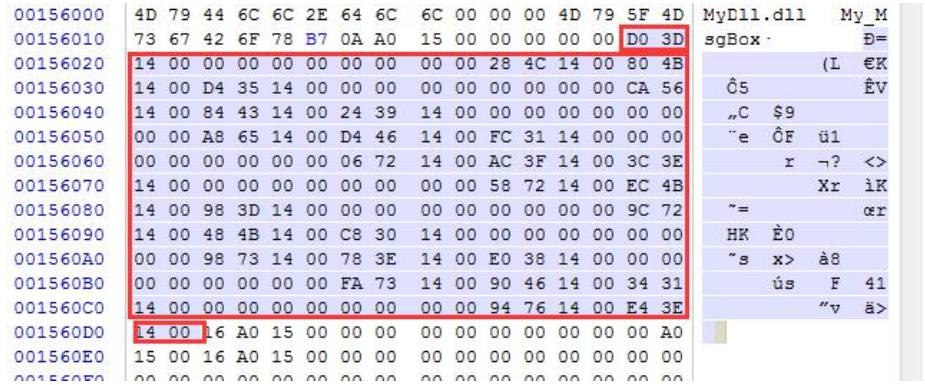
如图



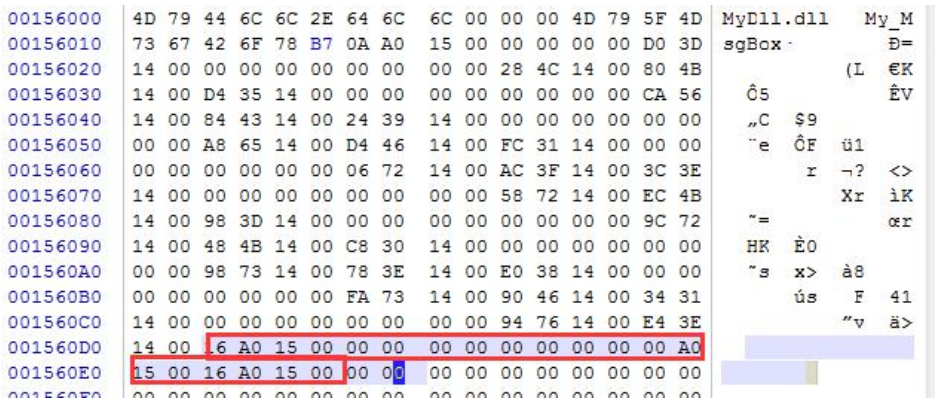
③紧接其后的是写入的 IMAGE_THUNK_DATA 结构体，如下图



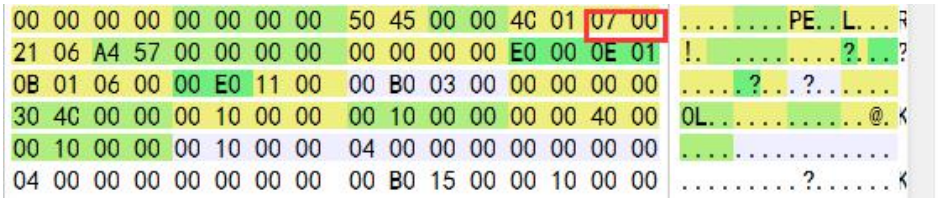
④把原来的导入表拷贝到新的节中，如下图



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

①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);
```

②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);

}
```

③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_H
EADER));
        fseek(fp,(DWORD)lpFirstSectionHeader-(DWORD)theApp.m_stMapFile.Ima
```

```
geBase, 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

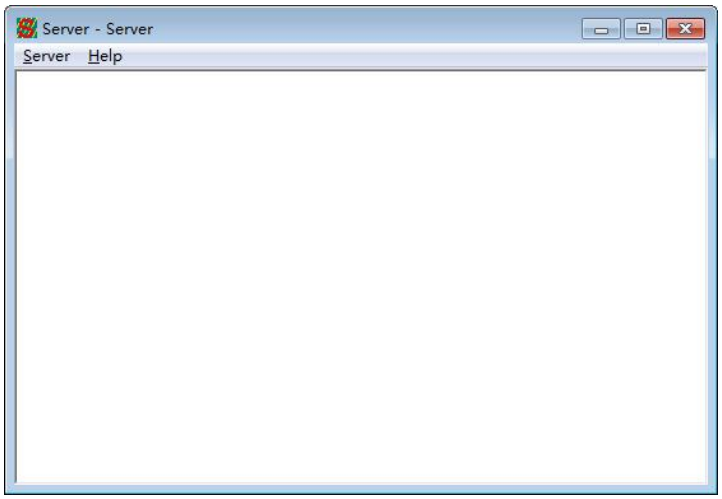
```
    //6.更新 NT 头数据
        memcpy(lpNewNtHeader, lpNtHeader, sizeof(IMAGE_NT_HEADERS));
        int nNewImageSize = lpNtHeader->OptionalHeader.SizeOfImage +
ClacAlignment(dwNewOffset, nSectionAlignment);
        lpNewNtHeader->OptionalHeader.SizeOfImage = nNewImageSize;
        lpNewNtHeader->OptionalHeader.DataDirectory[11].Size = 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_ENT
RY_IMPORT].VirtualAddress = dwNewImportRva;
        lpNewNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENT
RY_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++

①ShellCode 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

```
begin_label:
        push 12345678h                          ;push到栈中占坑用，存放ret后额的返回地址
        pushfd                                  ;保存标志位
        pushad                                  ;保存寄存器
        call $+5  ;重定位                       ;重定位代码，获取目标进程中的地址
FIXADDR:
        pop ebp
        sub ebp, FIXADDR                        ;求目标进程与注入中虚拟地址的差值，用于计算变量实际地址

        lea eax, [ebp + offset g_szDllPath]     ;DLL的路径
        push eax

        mov eax, [ebp + offset g_pfnLoadLibrary]
        call eax                                ;调用API
        popad                                   ;恢复寄存器
        popfd                                   ;恢复标志位
        ret
        g_pfnLoadLibrary DWORD 0
        g_szDllPath db 256 dup(0)
end_label:
```

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。

(2)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。

②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;
    }
```

③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;
}
```

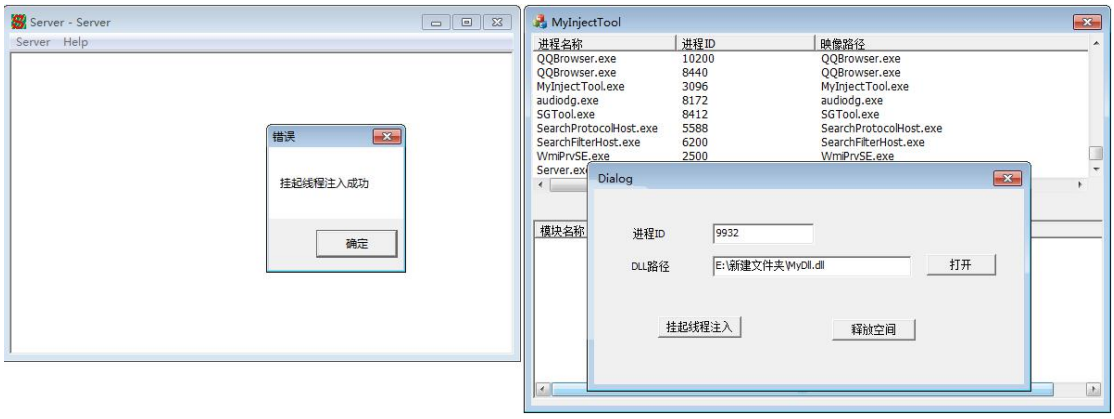
⑥扫尾工作，单独设了个函数清除目标进程中申请的空间，注意这个操作务必等待我们的 ShellCode 执行完再执行，否则会导致目标程序崩溃

```
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

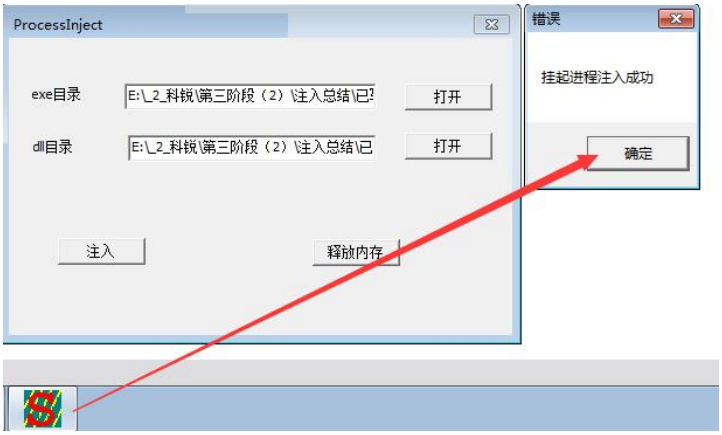
由于这种方式和挂起线程注入方式的注入手段大同小异，所以只看区别部分

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

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,                // name of executable module
  LPTSTR lpCommandLine,                     // command line string
  LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD
  LPSECURITY_ATTRIBUTES lpThreadAttributes,  // SD
  BOOL bInheritHandles,                     // handle inheritance option
  DWORD dwCreationFlags,                    // creation flags
  LPVOID lpEnvironment,                     // new environment block
  LPCTSTR lpCurrentDirectory,               // current directory name
  LPSTARTUPINFO lpStartupInfo,              // startup information
  LPPROCESS_INFORMATION lpProcessInformation // process information
);
```

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.

4.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。

①ShellCode 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()
```

②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);
```

③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

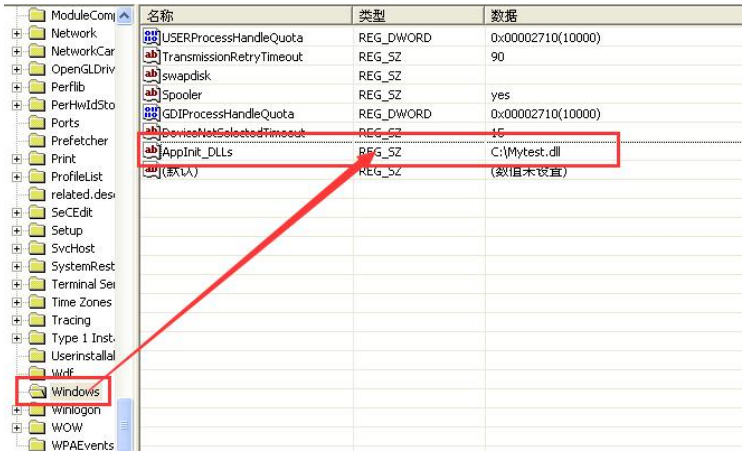
注册表注入是比较简单易懂的一种注入方式，这种方法比较暴力，本人在 win7 64 位系

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

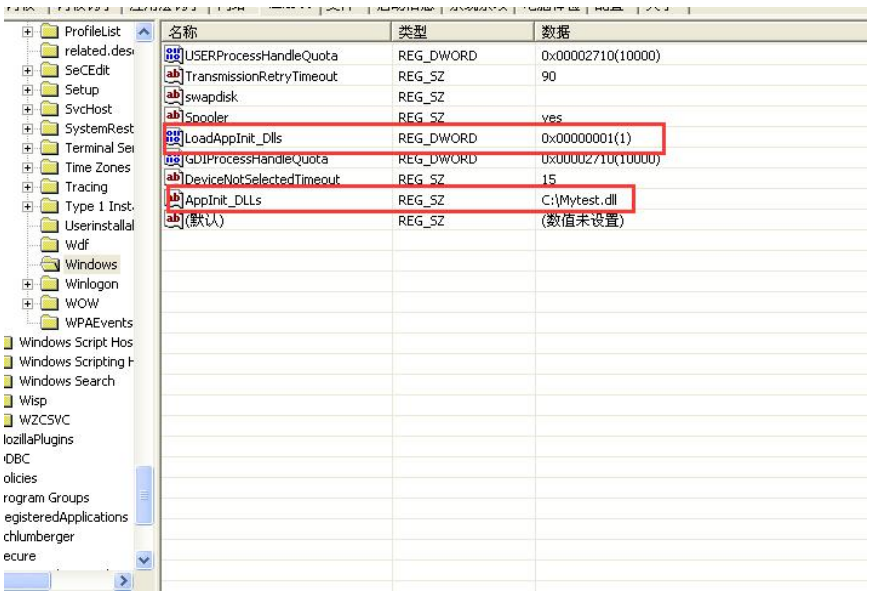
HKEY_LOCAL_MACHINE/Software/Microsoft/WindowsNT/CurrentVersion/Windows/AppInit_DLLs 的键值为我们 DLL 的路径，只要使用了 user32.dll 的程序都会加载这个目录下的 DLL。

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

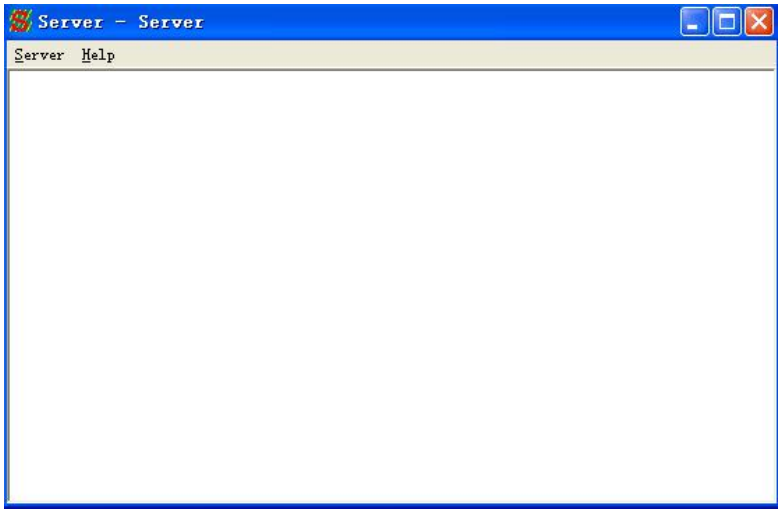
其实没必要用代码搞，手工几下就搞好了，但是其他的都用代码实现了，这个不实现心里不舒服。

```
//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, // hook type
                        HOOKPROC lpfn, // hook procedure
                        HINSTANCE hMod, // handle to application instance
                        DWORD dwThreadId // thread identifier);
```

参数介绍:

①idHook: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。

②lpfn:　挂钩回调函数。

③hMod：这个句柄指向包含有钩子回调函数的 DLL 的句柄。

④dwThreadId: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 // 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

①注入程序：一个挂钩一个解钩

②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;
        }
    }
```

③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

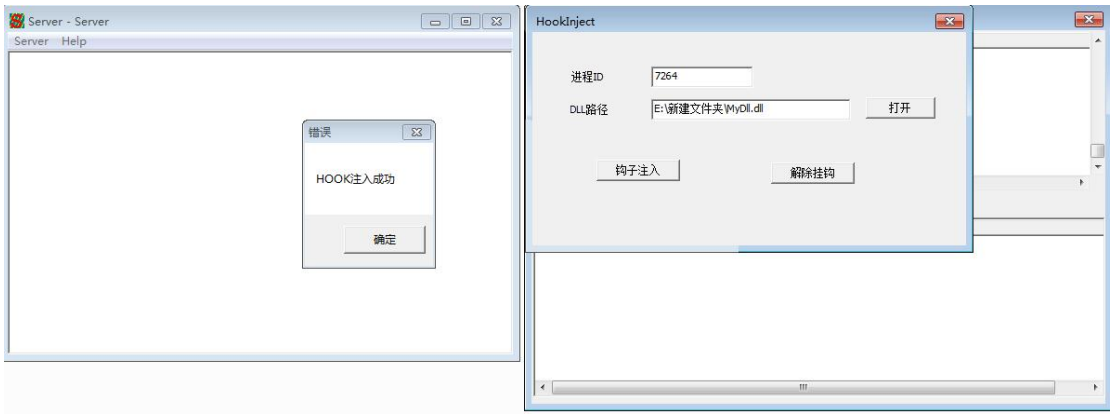
```
    //钩子回调函数
    LRESULT CALLBACK HookProc(
                                int code,           // hook code
                                WPARAM wParam,    // virtual-key code
                                LPARAM lParam      // keystroke-message information
                                )
```

```
    {
        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, // APC function
                    HANDLE hThread, // handle to thread
                    ULONG_PTR dwData // 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

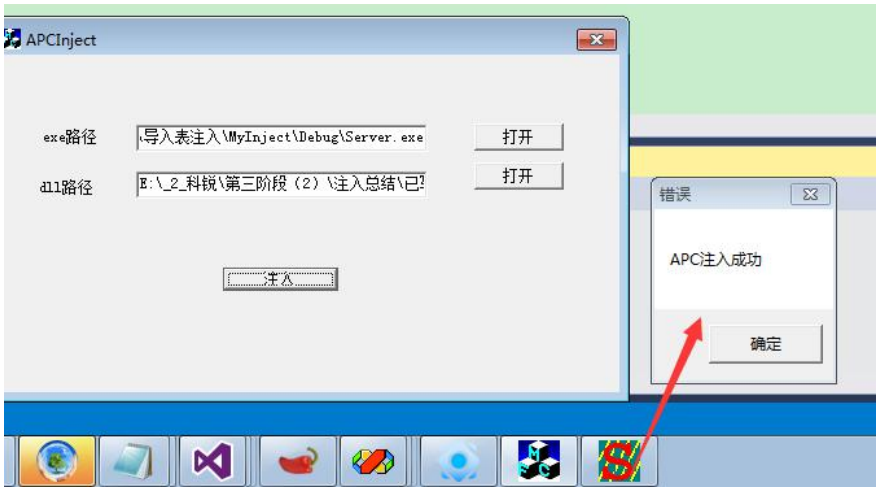
```
void CAPCInjectDlg::OnInject()
{
```

```
	// 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 结构的指针，该结指定了线程的安全属性

参数 3：线程栈初始大小,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

①打开远程进程

```
hProcess = OpenProcess(PROCESS_ALL_ACCESS,FALSE,dwProcessId);

if (hProcess == NULL)
{
    MessageBox("打开进程失败!!!!");
    return;
}
```

②VirtualAllocEx 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");
    }
```

③获取远程进程中 LaodLibry 的地址

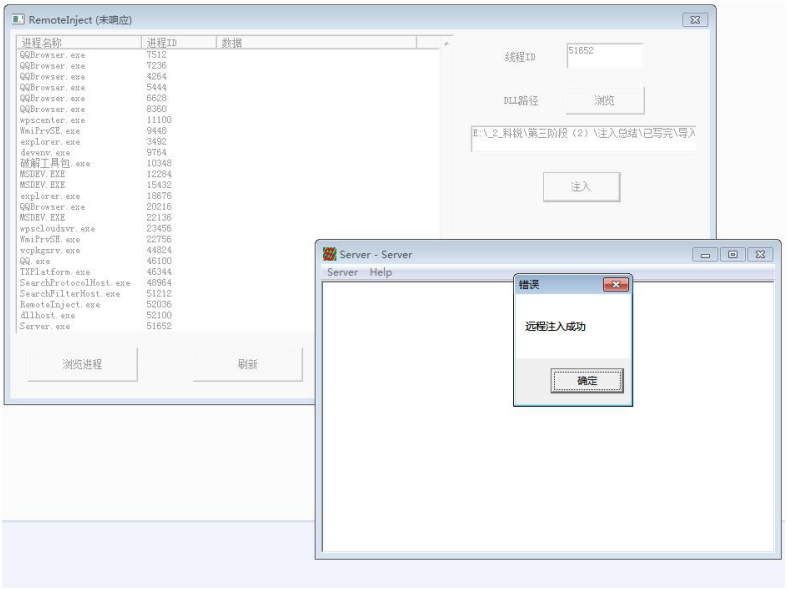
```
 //3.获取远程进程中 LaodLibry 的地址,这里你用的巧合是每个程序中的 kernel32 的地址的都一样，远程中也一样在
    HMODULE hMouDle = GetModuleHandle("Kernel32");
    PTHREAD_START_ROUTINE pfnLoadLibrary = (PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle("Kernel32"),"LoadLibraryA");

    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:

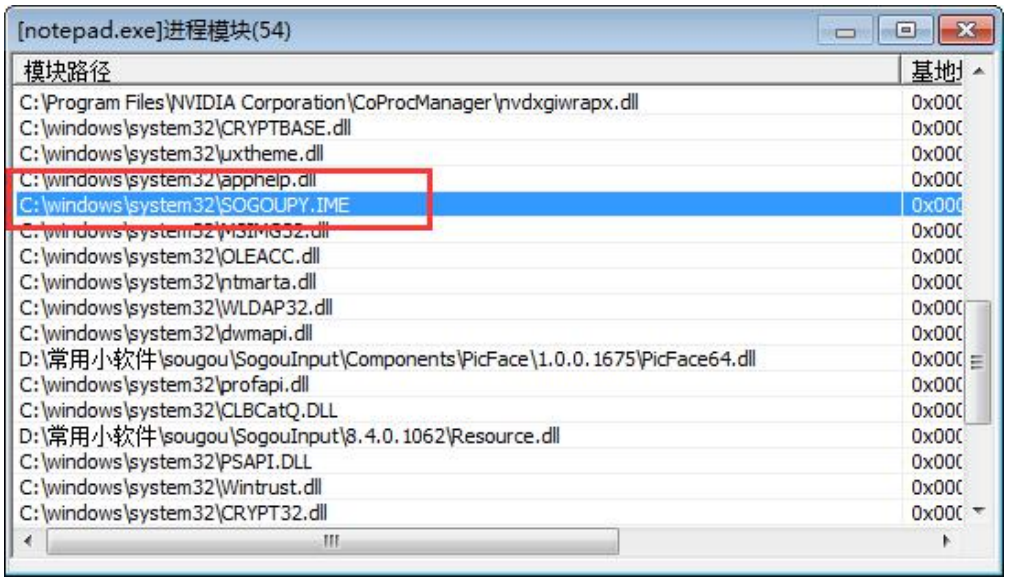


九、输入法注入

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:

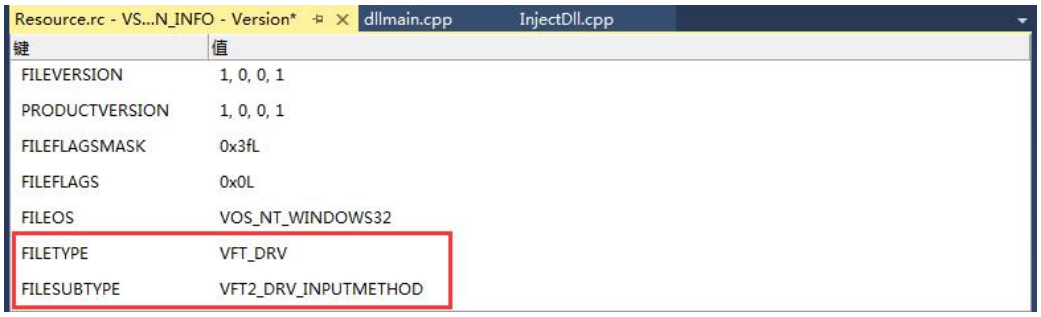
http://www.cnblogs.com/freedomshe/archive/2012/11/30/ime_learning.html
http://blog.csdn.net/pkfish/article/details/7339909

0x01 具体实现思路:

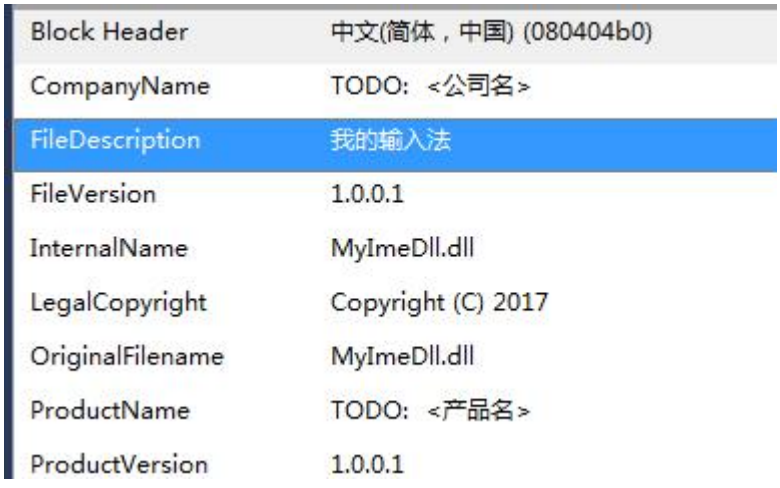
①编写输入法 Ime 文件，其实就是 DLL

（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



②SystemParametersInfo 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);
```

③CopyFile 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:

名称	类型	数据
Layout Text	REG_SZ	我的输入法
Layout File	REG_SZ	kbdus.dll
Ime File	REG_SZ	MYIMEDLL.IME
(默认)	REG_SZ	(数值未设置)

⑩清除注册表项，删除输入法文件

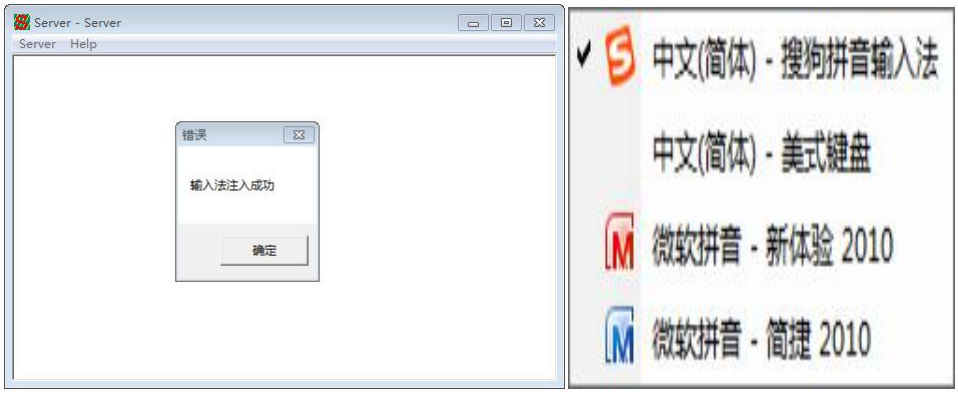
```
//12.删除注册表项，清理痕迹

    //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");
```

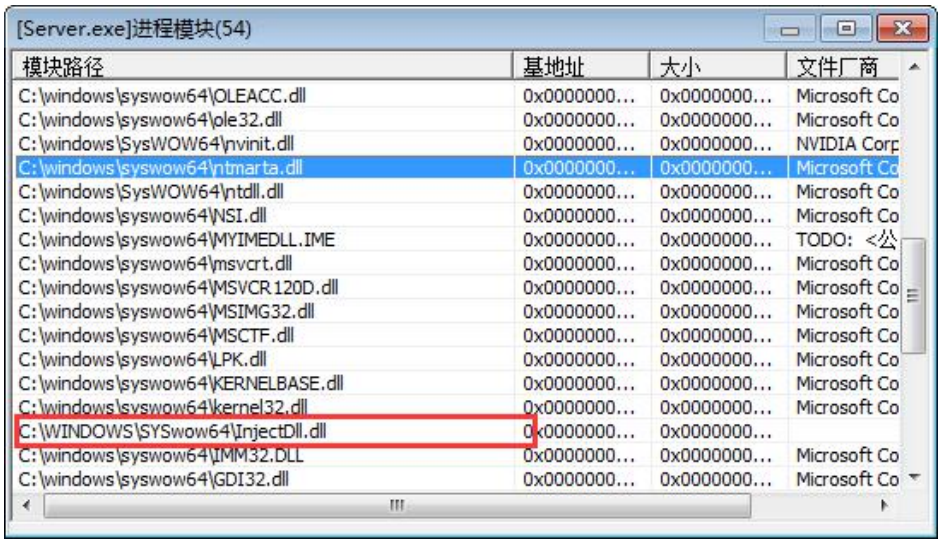
0x02 实验效果

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

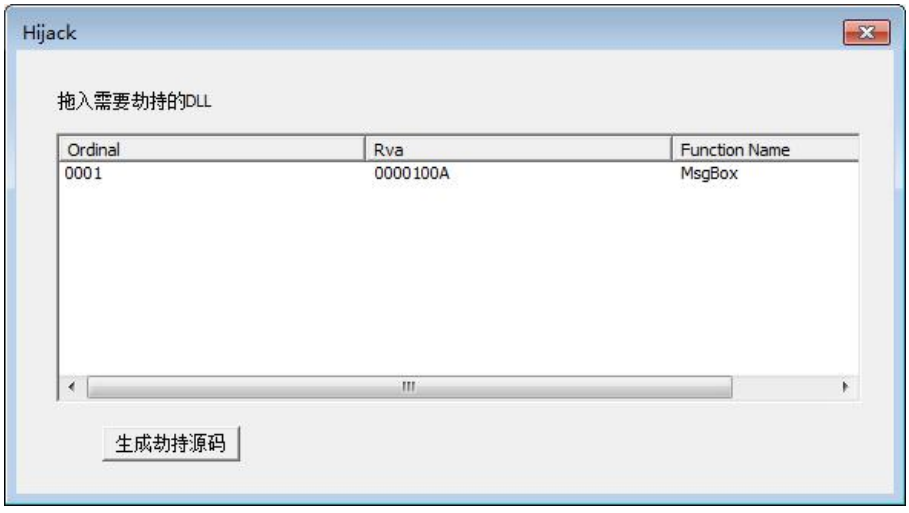
①我们需要劫持的 DLL 为 MyDll.dll,功能很简单，只有一个导出函数，功能为弹出一个 MessageBox，源码如下:

```
#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);
}
```

②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]
}
```

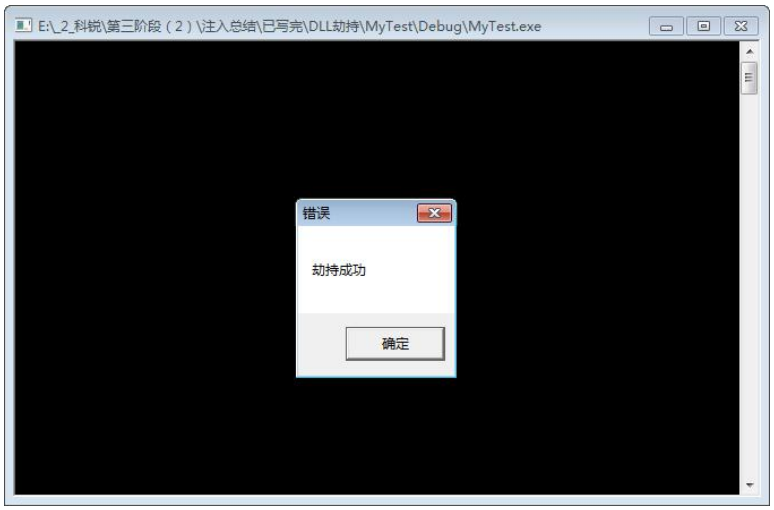
③编写测试程序

```
#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;
```

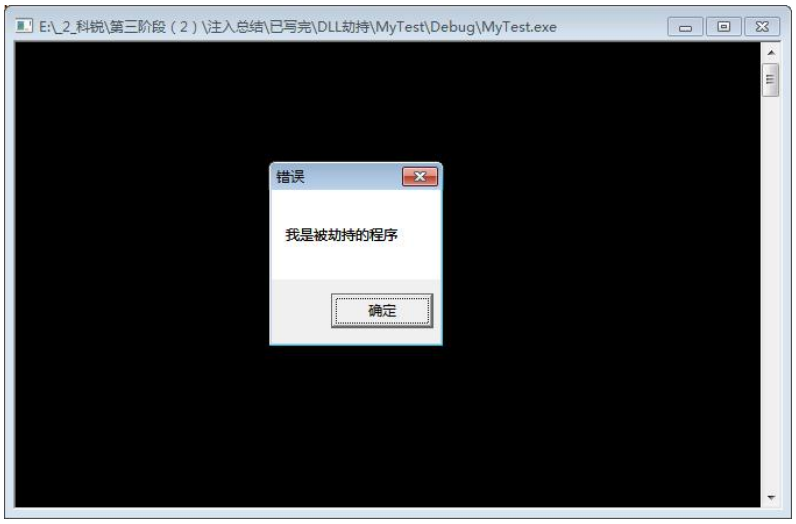
```
}
```

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

①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。

②挂起线程注入

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

③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。

①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。

②钩子注入

(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);
    }
```

(2)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

③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

《简单总结下常用的注入姿势》

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