进程注入的研究与实现

    为了对内存中的某个进程进行操作，并且获得该进程地址空间里的数据，或者修改进程的私有数据结构，必须将自己的代码放在目标进程的地址空间里运行，这时就避免不了使用进程注入方法了。

进程注入的方法分类如下：

    带DLL的注入

        利用注册表注入

        利用Windows Hooks注入

        利用远程线程注入

        利用特洛伊DLL注入

    不带DLL的注入

        直接将代码写入目标进程，并启动远程线程

1. 利用注册表注入

   在Windows NT/2000/XP/2003中，有一个注册表键值HKEY\_LOCAL\_MACHINE\Software\Microsoft\WindowsHKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit\_DLLs。当某个进程加载User32.dll时，这里面列出的所有的DLL都将User32.dll利用LoadLibrary函数加载到该进程空间中。我们可以把自己的代码放在一个DLL中，并加入该键值，这样就可以注入到所有使用User32.dll的进程中了。

   当DLL以LoadLibrary的方式加载时，DllMain会被以DLL\_PROCESS\_ATTACH为原因调用，实际上我们也只需要关心DLL\_PROCESS\_ATTACH

**Cpp代码**

1. **BOOL** APIENTRY DllMain( **HMODULE** hModule, **DWORD**  ul\_reason\_for\_call, **LPVOID** lpReserved )
2. {
3. **if** (ul\_reason\_for\_call == DLL\_PROCESS\_ATTACH)
4. {
5. **HANDLE** f = CreateFile(L"D:\\InjectSuccess.txt", FILE\_ADD\_FILE, FILE\_SHARE\_WRITE,
6. NULL, CREATE\_NEW, FILE\_ATTRIBUTE\_NORMAL, NULL);
7. CloseHandle(f);
8. }
9. **return** TRUE;
10. }

2. 利用Windows Hooks注入

   Windows系统给我们提供了一些挂钩函数，使得被挂钩的进程可以在自己处理接收到的消息之前，先执行我们的消息处理函数，而这个消息处理函数一般会放在DLL中，来让目标进程加载，这实际上已经达到了注入代码的效果。

   一般情况下，我们把挂钩函数和消息处理函数都放在dll中：

**Cpp代码**

1. **HHOOK** g\_hHook = NULL;
2. **HINSTANCE** hInst;
3. **BOOL** APIENTRY DllMain( **HMODULE** hModule, **DWORD**  ul\_reason\_for\_call, **LPVOID** lpReserved)
4. {
5. hInst = (**HINSTANCE**)hModule;
6. **return** TRUE;
7. }
8. **LRESULT** myKeyBrdFuncAd (**int** code, **WPARAM** wParam, **LPARAM** lParam)
9. {
10. // To be nice, your rootkit should call the next-lower
11. // hook, but you never know what this hook may be.
12. //::MessageBoxA(NULL, "Hello Injection", "Injection", MB\_OK);
13. **return** CallNextHookEx(g\_hHook, code, wParam, lParam);
14. }
15. #ifdef \_\_cplusplus    // If used by C++ code,
16. **extern** "C" {          // we need to export the C interface
17. #endif
18. **\_\_declspec**(**dllexport**) **bool** SetHook(**DWORD** dwThreadId)
19. {
20. g\_hHook = SetWindowsHookEx(WH\_KEYBOARD, (HOOKPROC)myKeyBrdFuncAd, hInst, dwThreadId);
21. **if** (g\_hHook == NULL)
22. {
23. **return** **false**;
24. }
25. **return** **true**;
26. }
27. #ifdef \_\_cplusplus
28. }
29. #endif

   注入进程要加载这个DLL，然后执行里面的SetHook函数，传入的参数为被注入线程的ID

**Cpp代码**

1. **typedef** **bool** (\*MYPROC) (**DWORD** dwThreadId);
2. ......
3. **HANDLE** hLib = LoadLibraryA("D:\\source\\rootkits\\injecting\\InjectDll\\Debug\\InjectDll.dll");
4. **if** (hLib == NULL)
5. {
6. printf("LoadLibrary Error!\n");
7. }
8. MYPROC myProc = (MYPROC)GetProcAddress((**HMODULE**)hLib,"SetHook");
9. **if** (myProc != NULL)
10. {
11. **if** ((\*myProc)((**DWORD**)4860) == **false**)
12. {
13. printf("loose: %d", GetLastError());
14. }
15. }
16. **else**
17. {
18. printf("GetProcAddress error: %d", GetLastError());
19. }

3. 利用远程线程注入DLL

    １）、取得远程进程的进程ID；

　　２）、在远程进程空间中分配一段内存用来存放要注入的DLL完整路径；

　　３）、将要注入的DLL的路径写到刚才分配的远程进程空间；

    4 ）、从Kernel32.dll中取得LoadLibray的地址；

　　５）、调用CreateRemoteThread函数以从Kernel32.dll中取得的LoadLibrary函数的地址为线程函数的地址，以我们要注入的DLL文件名为参数，创建远程线程；

　　在第二三步中，为什么要把我们要注入的DLL的文件名写到远程进程的地址空间进行操作，《WINDOWS核心编程》中是这样描述的：

“（要注入的DLL文件名）字符串是在调用进程的地址空间中。该字符串的地址已经被赋予新创建的远程线程，该线程将它传递给L o a d L i b r a r y A。但是，当L o a d L i b r a r y A取消对内存地址的引用时， D L L路径名字符串将不再存在，远程进程的线程就可能引发访问违规”；

　　至于第四步中为什么不直接对LoadLibrary进行调用，《WINDOWS核心编程》中是这样描述的：

“如果在对C r e a t e R e m o t e T h r e a d的调用中使用一个对L o a d L i b r a r y A的直接引用，这将在你的模块的输入节中转换成L o a d L i b r a r y A的形式替换程序的地址。将形实替换程序的地址作为远程线程的起始地址来传递，会导致远程线程开始执行一些令人莫名其妙的东西。其结果很可能造成访问违规。”

   我在DLL中只是做一些简单的处理以显示注入成功。

**Cpp代码**

1. **BOOL** APIENTRY DllMain( **HMODULE** hModule, **DWORD**  ul\_reason\_for\_call, **LPVOID** lpReserved)
2. {
3. hInst = (**HINSTANCE**)hModule;
4. **if** (ul\_reason\_for\_call == DLL\_PROCESS\_ATTACH)
5. {
6. **HANDLE** f = CreateFile(L"D:\\InjectSuccess.txt", FILE\_ADD\_FILE, FILE\_SHARE\_WRITE,
7. NULL, CREATE\_NEW, FILE\_ATTRIBUTE\_NORMAL, NULL);
8. CloseHandle(f);
9. hInst = (**HINSTANCE**)hModule;
10. }
11. **return** TRUE;
12. }

   按照上面的4个步骤，注入进程的代码如下：

**Cpp代码**

1. **LPWSTR** lpszLibName = L"D:\\source\\rootkits\\injecting\\InjectDll\\Debug\\InjectDll.dll";
2. **HANDLE** hProcess = OpenProcess(PROCESS\_CREATE\_THREAD | PROCESS\_VM\_OPERATION | PROCESS\_VM\_WRITE,FALSE, 3256);
3. **LPWSTR** lpszRemoteFile = (**LPWSTR**)VirtualAllocEx(hProcess, NULL, **sizeof**(**WCHAR**) \* lstrlenW(lpszLibName) + 1, MEM\_COMMIT, PAGE\_READWRITE);
4. WriteProcessMemory(hProcess, lpszRemoteFile,(**PVOID**)lpszLibName, **sizeof**(**WCHAR**) \* lstrlenW(lpszLibName) + 1, NULL);
5. PTHREAD\_START\_ROUTINE pfnThreadRtn = (PTHREAD\_START\_ROUTINE)GetProcAddress(GetModuleHandle(L"Kernel32.dll"), "LoadLibraryW");
6. **HANDLE** hThread = CreateRemoteThread(hProcess, NULL, 0, pfnThreadRtn, // LoadLibrary地址
7. lpszRemoteFile, // 要加载的DLL名
8. 0, NULL);

    在上面的过程中，实际上我们做了一个假设，就是所有进程中的kernel32.dll和user32.dll都被映射到了同一段虚拟地址上。

4. 利用特洛伊DLL进行注入

    这种方法的原理就是由自己写一个与原有进程调用的DLL具有相同接口函数的DLL，再用我们的DLL替换原有的DLL。在替换的过程中，由我们自己编写感兴趣的函数替换原有函数，而对其它不感兴趣的函数，则以函数转发的形式调用原有DLL中的函数。这里面有个前提，就是你在编写DLL时你必须知道原有DLL中的函数都有哪些，以免导至其它进程调用DLL时找不到相应的API函数，特别是在替换系统DLL文件时更要小心。

5. 无DLL注入

   在第三中方法中，我们启动远程线程时，线程函数是我们从Kernel32.dll中取得的LoadLibrary函数的地址为线程函数的地址，其实我们可以直接将线程函数体和函数参数写入目标进程的地址空间，然后创建远程线程。

   使用这个方法时，需要注意以下几个问题：

   (1) 远程线程函数体不得使用kernel32.dll，user32.dll以外的函数。因为这个两个模块在各个进程的相对地址是一样的，如果一定要使用其它函数，则必须将函数体写入目标进程空间。

   (2) 不能使用任何字符串常量，因为字符串常量是存放在PE文件里.data这个段里面的，函数里保存的只是相对地址。

   (3) 去掉编译器的/GZ编译选项，这个选项是用来Enable Stack Frame Run-Time Error Checking。当这个选项打开时，编译器会在每个函数中加入一些代码，用来检验ESP在函数体中是否被改变，但是这些检验函数的地址在不同PE文件中有可能是不一样的。

   (4) 不得使用增量链接（incremental linking）。增量链接是编译器为了减少链接时间做的处理，把函数体用一个JMP指令代替，这样就可以随意改变函数的内容，而不用修改CALL指令。

   (5) 不要在函数体内使用超过4kb的局部变量。局部变量是存放在栈中的，例如下面这个函数

 void Dummy(void) {

     BYTE var[256];

     var[0] = 0;

     var[1] = 1;

     var[255] = 255;

 }

 在分配局部变量空间时是这样的

 :00401000   push ebp

 :00401001   mov  ebp, esp

 :00401003   sub  esp, 00000100           ; change ESP as storage for

                                          ; local variables is needed

 :00401006   mov  byte ptr [esp], 00      ; var[0] = 0;

 :0040100A   mov  byte ptr [esp+01], 01   ; var[1] = 1;

 :0040100F   mov  byte ptr [esp+FF], FF   ; var[255] = 255;

 :00401017   mov  esp, ebp                ; restore stack pointer

 :00401019   pop  ebp

 :0040101A   ret

 但是当局部变量的大小超过4kb时，栈指针并不直接改版，而是调用另一个函数来分配内存，这个函数有可能在不同进程中的地址不一样。

   (6) 函数体内switch语句中的case不要超过3个，否则编译器会在PE文件中使用跳转表，而这个跳转表有可能在目标进程中并不存在。

   下面是一个无DLL注入的例子：

//参数结构 ;

**Cpp代码**

1. **typedef** **struct** \_RemotePara{
2. **PVOID** dwMessageBox;
3. **wchar\_t** strMessageBox[12];
4. }RemotePara;
5. // 远程线程执行体
6. **DWORD** \_\_stdcall ThreadProc(RemotePara \*Para)
7. {
8. **typedef** **int** (/\*\_\_stdcall\*/ \*PMessageBox) ( **HWND** , **LPCTSTR** , **LPCTSTR** , **UINT** );
9. PMessageBox MessageBoxFunc = (PMessageBox)Para->dwMessageBox;
10. MessageBoxFunc(NULL, Para->strMessageBox, Para->strMessageBox, MB\_OK);
11. **return** 0 ;
12. }
13. **DWORD** THREADSIZE=1024;
14. **DWORD** pID = 4688;
15. **DWORD** byte\_write;
16. **HANDLE** hRemoteProcess,hThread;
17. RemotePara myRemotePara,\*pRemotePara;
18. **void** \*pRemoteThread;
19. **HINSTANCE** hUser32 ;
20. hRemoteProcess = OpenProcess(PROCESS\_ALL\_ACCESS,FALSE,pID);
21. **if**(!hRemoteProcess)**return** 0;
22. // 在远程进程地址空间分配虚拟内存
23. pRemoteThread = VirtualAllocEx(hRemoteProcess, 0, THREADSIZE, MEM\_COMMIT | MEM\_RESERVE,PAGE\_EXECUTE\_READWRITE);
24. **if**(!pRemoteThread)**return** 0;
25. // 将线程执行体ThreadProc写入远程进程
26. **if**(!WriteProcessMemory(hRemoteProcess, pRemoteThread, &ThreadProc, THREADSIZE,0))**return** 0;
27. ZeroMemory(&myRemotePara,**sizeof**(RemotePara));
28. hUser32 = LoadLibrary(L"user32.dll");
29. myRemotePara.dwMessageBox = (**PVOID**)GetProcAddress(hUser32, "MessageBoxW");
30. wcscat(myRemotePara.strMessageBox,L"Hello!"); //复制MessageBox函数的参数
31. //写进目标进程
32. pRemotePara =(RemotePara \*)VirtualAllocEx (hRemoteProcess ,0,**sizeof**(RemotePara),MEM\_COMMIT,PAGE\_READWRITE);
33. **if**(!pRemotePara)**return** 0;
34. **if**(!WriteProcessMemory (hRemoteProcess ,pRemotePara,&myRemotePara,**sizeof** myRemotePara,0))**return** 0;
35. // 启动线程
36. hThread = CreateRemoteThread(hRemoteProcess ,0,0,(LPTHREAD\_START\_ROUTINE)pRemoteThread ,pRemotePara,0,&byte\_write);
37. FreeLibrary(hUser32);
38. CloseHandle(hRemoteProcess);

参考文献：

http://www.codeproject.com/threads/winspy.asp?df=100&forumid=16291&select=1025152&msg=1025152#section\_2

http://hi.baidu.com/sunshineboys/blog/item/828f4136c46534dda2cc2b35.html