[](https://bbs.pediy.com/)

窗体顶端



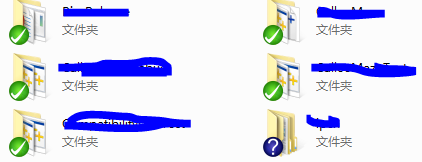
窗体底端

1. [看雪论坛](https://bbs.pediy.com/)
2. [『编程技术』](https://bbs.pediy.com/forum-41.htm)

[发新帖](https://bbs.pediy.com/thread-create-41.htm)

[原创]深入探究Windows平台客户端安全问题-进程地址空间入侵和白加黑高阶利用

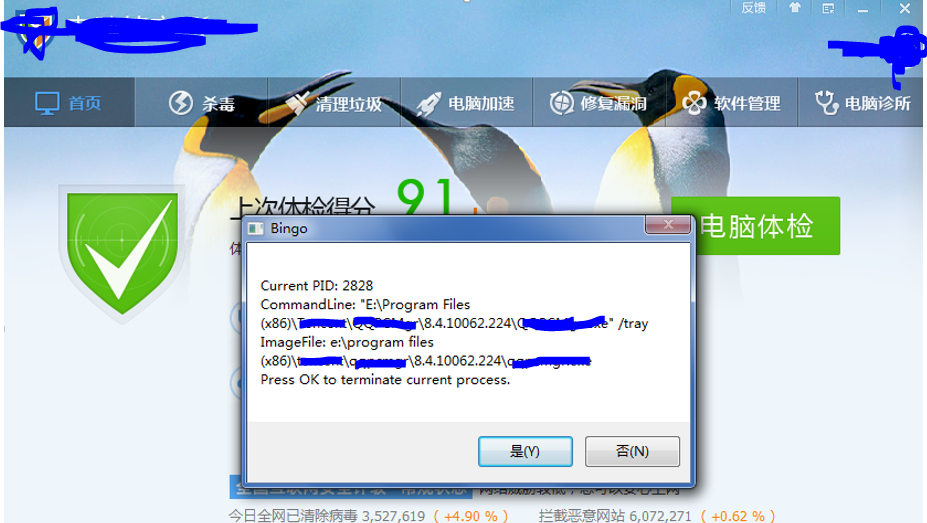
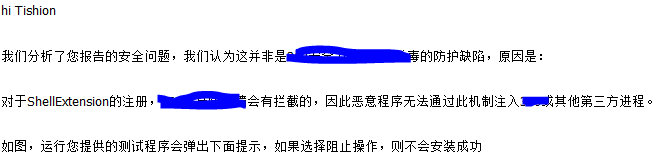
[tishion](https://bbs.pediy.com/user-271122.htm) 2014-9-8 00:03  34635

**前言**  
为了避免被读者骂“标题党”，笔者在文章开头先澄清一下这个高大尚的“进程地址空间入侵”的可替代词语—注入。  
  
看完第一句还能看到这里的读者一般有两种：1.初学者，实在是不懂所以需要学习的同学 2.大牛，只是想看看笔者打算怎么来炒“注入”这盘冷饭。  
  
好吧，如很多人都知道的一样，Windows平台下的远程模块”注入“是盘冷饭，但是却很少有人对这个技术做一个全面，系统，深入的讲解。因为笔者现在是在职的开发人员，不像学生时代一个月能写五六篇文章那样高产，现在的状态也就是每年两篇看雪精华帖，所以笔者力求自己现在写的每篇文章都能推陈出新，浅显易懂，并且对问题的本质，利用方法，解决方法等多方面都进行介绍。  
  
进程地址空间入侵[Process Address Space Intrusion](注入)  
注入就是使一个非目标进程所拥有的Dll模块通过某种手段被目标进程加载到自己的地址空间，并且使得该Dll模块中的代码得到执行权限。说到这里可能大部分人就会想到通过CreateRemoteThread传入LoadLibraryA/W和目标模块路径的字符串来让一个目标进程主动加载目标模块这种方法。确实这个方法都被许多人讲烂了，但是读者有没有仔细思考过这种行为的本质是什么？Windows平台下的所有进程的虚拟地址空间都是独立的，这样就保证了每个进程的先对稳定性和安全行，但是Windows也提供了对非自身进程的其他进程的地址空间进行访问的能力，比如Reader/WriteProcessMemory函数，这样就可以提供了一种向远程进程写入一段可执行的代码，并且让这段代码得到执行权限/机会的能力，发散一下思维，注入一整个模块其实也是通过某种手段在目标进程的地址空间开辟出一个能容纳目标模块的内存Section，然后让这个模块里面所含的代码得到执行权限。所以上面所说的“注入”其实本质上就是一种进程地址空间入侵。  
  
由于Windows系统在设计上的灵活性和完整性，系统本身就已经提供了非常多的进程空间入侵手段，这里总结一下：  
1. 通过CreateRemoteThread创建远程线程调用LoadLibrary  
2. 通过QueueUserAPC挂载APC调用LoadLibrary  
3. 向远程进程中写入一段简单的Shell Code，在Shell Code调用LoadLibrary  
4. 修改远程线程的Context，在其Context中构造调用LoadLibray的堆栈状态  
5. Lsp注入  
6. 输入法注入  
7. Explorer Shell Extension注入  
8. 通过SetWindowsHookEx或者SetWinEventHook函数设置全局钩子  
9. Dll模块劫持(LPK.dll,ComRes.dll, 以及开发者犯错构成的模块劫持)  
  
以上是笔者自己总结的现有的所有的进程地址空间入侵的方法，如果读者觉得有遗漏，还请不吝指出。有一点需要明确，以上所有方法都需要LoadLibrary函数的介入，因为要把一个模块加载到一个进程空间就涉及到PE Loader相关的操作了，所以直接使用Windows自身的加载方法可以省却很多麻烦。上述方法中的1~6都是比较有名的方法，而且各大安全软件厂商也都有比较成熟的防御方法，所以本文就不做深入的讲解了，下面笔者将对7，8，9这三种方法的原理和利用进行详细深入的探讨，至于防御方法还是需要更多人去研究了。  
  
Explorer Shell Extension注入法  
原理  
这种注入方法知道的人比较少，也因为该种注入方法有不少限制，所以一般适用性不太大，但是正由于其不引起人们的注意，所以某些安全软件都会没有对其做相应的防护。先说一下该方法的大致思路，因为Windows的Explorer Shell提供了非常高的可扩展性，所以这也提供了一种进程去加载一个指定DLL模块的途径。比如Explorer中的右键菜单，当你在桌面空白或者某个文件对象上点击右键的时候Explorer会把当前系统中注册的需要显示的右键菜单所在的Dll模块加载到Explorer进程中，然后调用响应的接口显示出菜单项目。看到了么，轻而易举的将一个进程注入到了Explorer进程中，但是一般选择Explorer作为目标进程没有太大实际意义，所以这种方法就没有用处了么？不是这样的，应该这样说，凡是使用了Windows Shell函数的进程都遵循这一规则，比如一个进程使用了GetOpenFileName这个函数，这个函数的作用就是弹出一个常见的文件选择对话框，如果用户在弹出的对话框中单击鼠标右键，那么当前进程还是要遵照Shell Extension的规则把当前系统中当前用户注册的所有右键菜单的模块都加载到该进程中。可以看出此种注入方法存在的限制  
1. 进程使用了Explorer Shell Extension函数  
2. 用户点击了右键  
限制确实是多了点，但是我们可以再减少一点，选择右键菜单的Shell 扩展并不是最明智的做法，更简单的方法是选择ShellIconOverlayIdentifiers这个扩展点来进行注入，这个扩展点作用是控制Explorer中的文件对象的图标，比如我们常用的SVN和GIT软件，他们都会注册这个扩展点来实现对文件和文件夹图标的添加额外显示元素，如下图所示：  
  
<P1>  
使用这个扩展点的好处是，只要Explorer的对话框显示出来，这些扩展点注册的模块就会立即被加载起来，无需用户去点击右键菜单，在这个扩展点加载模块时候的堆栈如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | ModLoad: 00000000`73e00000 00000000`73e0d000   E:\Users\xxxxx\Desktop\spookshellext\Release\SpookShlExt.dll  ntdll!NtMapViewOfSection+0xa:  00000000`77b7153a c3              ret  0:000> .effmach x86  Effective machine: x86 compatible (x86)  0:000:x86> kvn 5000   # ChildEBP RetAddr  Args to Child  00 002dbd10 77d3bf70 0000027c ffffffff 002dbe3c ntdll32!NtMapViewOfSection+0x12 (FPO: [10,0,0])  01 002dbd64 77d3c5fb 0000027c 00000000 00000000 ntdll32!LdrpMapViewOfSection+0xc7 (FPO: [Non-Fpo])  02 002dbe58 77d3c42c 002dbea4 012dc004 00000000 ntdll32!LdrpFindOrMapDll+0x333 (FPO: [Non-Fpo])  03 002dbfd8 77d3c558 002dc03c 002dc004 00000000 ntdll32!LdrpLoadDll+0x2b2 (FPO: [Non-Fpo])  04 002dc010 75962c95 002dc004 002dc054 002dc03c ntdll32!LdrLoadDll+0xaa (FPO: [Non-Fpo])  05 002dc04c 764a9d43 00000000 00000000 00556614 KERNELBASE!LoadLibraryExW+0x1f1 (FPO: [Non-Fpo])  06 002dc068 764a9cc7 00000000 002dc0e4 00000008 ole32!LoadLibraryWithLogging+0x16 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\common\loadfree.cxx @ 157]  07 002dc08c 764a9bb6 002dc0e4 002dc0b0 002dc0b4 ole32!CClassCache::CDllPathEntry::LoadDll+0xa9 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\dllcache.cxx @ 1925]  08 002dc0bc 764a90be 002dc0e4 002dc3cc 002dc0dc ole32!CClassCache::CDllPathEntry::Create\_rl+0x37 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\dllcache.cxx @ 1783]  09 002dc308 764a8f93 00000001 002dc3cc 002dc338 ole32!CClassCache::CClassEntry::CreateDllClassEntry\_rl+0xd4 (FPO: [Non-Fpo]) (CONV: thiscall) [d:\w7rtm\com\ole32\com\objact\dllcache.cxx @ 886]  0a 002dc350 764a8e99 00000001 00525464 002dc37c ole32!CClassCache::GetClassObjectActivator+0x224 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\dllcache.cxx @ 4795]  0b 002dc388 764a8c57 002dc3cc 00000000 002dc9d4 ole32!CClassCache::GetClassObject+0x30 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\dllcache.cxx @ 4574]  0c 002dc404 764c3170 765c6444 00000000 002dc9d4 ole32!CServerContextActivator::CreateInstance+0x110 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\actvator.cxx @ 974]  0d 002dc444 764a8dca 002dc9d4 00000000 002dcf38 ole32!ActivationPropertiesIn::DelegateCreateInstance+0x108 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\actprops\actprops.cxx @ 1917]  0e 002dc498 764a8d3f 765c646c 00000000 002dc9d4 ole32!CApartmentActivator::CreateInstance+0x112 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\actvator.cxx @ 2268]  0f 002dc4b8 764a8ac2 765c6494 00000001 00000000 ole32!CProcessActivator::CCICallback+0x6d (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\actvator.cxx @ 1737]  10 002dc4d8 764a8a73 765c6494 002dc830 00000000 ole32!CProcessActivator::AttemptActivation+0x2c (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\actvator.cxx @ 1630]  11 002dc514 764a8e2d 765c6494 002dc830 00000000 ole32!CProcessActivator::ActivateByContext+0x4f (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\actvator.cxx @ 1487]  12 002dc53c 764c3170 765c6494 00000000 002dc9d4 ole32!CProcessActivator::CreateInstance+0x49 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\actvator.cxx @ 1377]  13 002dc57c 764c2ef4 002dc9d4 00000000 002dcf38 ole32!ActivationPropertiesIn::DelegateCreateInstance+0x108 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\actprops\actprops.cxx @ 1917]  14 002dc7dc 764c3170 765c6448 00000000 002dc9d4 ole32!CClientContextActivator::CreateInstance+0xb0 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\actvator.cxx @ 685]  15 002dc81c 764c3098 002dc9d4 00000000 002dcf38 ole32!ActivationPropertiesIn::DelegateCreateInstance+0x108 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\actprops\actprops.cxx @ 1917]  16 002dcfec 764c9e25 00545700 00000000 00000401 ole32!ICoCreateInstanceEx+0x404 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\objact.cxx @ 1334]  17 002dd04c 764c9d86 00545700 00000000 00000401 ole32!CComActivator::DoCreateInstance+0xd9 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\immact.hxx @ 343]  18 002dd070 764c9d3f 00545700 00000000 00000401 ole32!CoCreateInstanceEx+0x38 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\actapi.cxx @ 157]  19 002dd0a0 76ae82c0 00545700 00000000 00000401 ole32!CoCreateInstance+0x37 (FPO: [Non-Fpo]) (CONV: stdcall) [d:\w7rtm\com\ole32\com\objact\actapi.cxx @ 110]  1a 002dd338 76ae83bd 00545700 00000000 00000401 SHELL32!\_SHCoCreateInstance+0x1ac (FPO: [Non-Fpo])  1b 002dd35c 76aa9a6f 00545700 00000000 00000401 SHELL32!SHExtCoCreateInstance+0x1e (FPO: [Non-Fpo])  1c 002dd37c 76a8354e 00545660 00000007 00000002 SHELL32!DCA\_SHExtCoCreateInstance+0x32 (FPO: [Non-Fpo])  1d 002dd8f0 76a83a49 00000000 00545610 76a83a19 SHELL32!CFSIconOverlayManager::\_s\_LoadIconOverlayIdentifiers+0x12b (FPO: [Non-Fpo])  1e 002dd8fc 76a83a19 002dd98c 80004005 00002000 SHELL32!CFSIconOverlayManager::\_InitializeHdsaIconOverlays+0xb (FPO: [0,0,0])  1f 002dd910 76a83991 00000000 76a839b8 002dd924 SHELL32!CFSIconOverlayManager::CreateInstance+0x4e (FPO: [Non-Fpo])  20 002dd92c 76a92a84 00000000 00000001 002dd994 SHELL32!IconOverlayManagerInit+0x2a (FPO: [Non-Fpo])  21 002dd93c 76a83974 002dd98c 76889679 002dda00 SHELL32!GetIconOverlayManager+0x17 (FPO: [Non-Fpo])  22 002dd994 76b8210a 00000000 00542058 002ddc10 SHELL32!FileIconInit+0x218 (FPO: [Non-Fpo])  23 002dd9a4 63db4285 00000003 63d45538 002dda00 SHELL32!SHGetImageList+0x1b (FPO: [Non-Fpo])  24 002ddc10 63d543c6 00542058 0001090a 63db4084 explorerframe!CBreadcrumbBar::InitBreadcrumbBar+0x1d9 (FPO: [Non-Fpo])  25 002ddc28 63db4066 00534e80 00541078 00000001 explorerframe!CAddressBand::\_EnsureBreadcrumbBar+0x6f (FPO: [0,0,4])  26 002ddc68 63d558ac 00534e80 00534e80 80004005 explorerframe!CAddressBand::\_CreateAddressBand+0xf5 (FPO: [Non-Fpo])  27 002ddc80 758f34b6 00541078 00000000 00534e70 explorerframe!CAddressBand::SetSite+0x62 (FPO: [Non-Fpo])  28 002ddcac 63d4acdb 00541080 00534e80 00534e80 SHLWAPI!IUnknown\_SetSite+0x44 (FPO: [Non-Fpo])  29 002ddcc8 63d4ad7d 00541080 00000001 00534d78 explorerframe!CBandSite::\_AddBandByID+0x91 (FPO: [Non-Fpo])  2a 002ddcdc 63d74082 00534e80 00541080 76889679 explorerframe!CBandSite::AddBand+0x17 (FPO: [Non-Fpo])  2b 002ddd60 63d73e12 00534d78 00534dc0 00000000 explorerframe!CNavBar::\_CreateBands+0x11a (FPO: [Non-Fpo])  2c 002dde58 63d74231 00526c48 002dde84 759a8a45 explorerframe!CNavBar::\_CreateBar+0x137 (FPO: [Non-Fpo])  2d 002dde64 759a8a45 00534d78 00000001 00000000 explorerframe!CNavBar::ShowDW+0x10 (FPO: [Non-Fpo])  2e 002dde84 759a85bb 00526c48 00000000 000208da COMDLG32!CFileOpenSave::\_CreateNavigationBar+0xd8 (FPO: [Non-Fpo])  2f 002de110 759a7f97 000500f4 00000001 759a3a45 COMDLG32!CFileOpenSave::\_InitOpenSaveDialog+0x68a (FPO: [Non-Fpo])  30 002de37c 768862fa 000500f4 00000110 000208da COMDLG32!CFileOpenSave::s\_OpenSaveDlgProc+0x10b (FPO: [Non-Fpo])  31 002de3a8 768af9df 759a3a45 000500f4 00000110 USER32!InternalCallWinProc+0x23  32 002de424 768af784 0052719c 759a3a45 000500f4 USER32!UserCallDlgProcCheckWow+0xd7 (FPO: [Non-Fpo])  33 002de474 768af889 0131aaa0 00000000 00000110 USER32!DefDlgProcWorker+0xb7 (FPO: [Non-Fpo])  34 002de494 768862fa 000500f4 00000110 000208da USER32!DefDlgProcW+0x29 (FPO: [Non-Fpo])  35 002de4c0 76886d3a 768af860 000500f4 00000110 USER32!InternalCallWinProc+0x23  36 002de538 7688965e 0052719c 77b446b4 000500f4 USER32!UserCallWinProcCheckWow+0x109 (FPO: [Non-Fpo])  37 002de57c 768b206f 0131aaa0 00000000 77b446b4 USER32!SendMessageWorker+0x581 (FPO: [Non-Fpo])  38 002de650 768acf4b 759a0000 00000007 00000000 USER32!InternalCreateDialog+0xb9f (FPO: [Non-Fpo])  39 002de688 768ace8a 759a0000 005272e0 00050810 USER32!InternalDialogBox+0xc1 (FPO: [Non-Fpo])  3a 002de6a8 768acc0e 759a0000 005272e0 00050810 USER32!DialogBoxIndirectParamAorW+0x37 (FPO: [Non-Fpo])  3b 002de6c8 759a597b 759a0000 005272e0 00050810 USER32!DialogBoxIndirectParamW+0x1b (FPO: [Non-Fpo])  3c 002de714 759e1b50 005272e0 00050810 00000000 COMDLG32!CFileOpenSave::Show+0x181 (FPO: [Non-Fpo])  3d 002de740 759e29d0 00526c4c 00000611 00050810 COMDLG32!\_InvokeNewFileOpenSave+0xab (FPO: [Non-Fpo])  3e 002de76c 759e2b15 00000611 00050810 00526c4c COMDLG32!\_CreateNewFileOpenSaveInProc+0xae (FPO: [Non-Fpo])  3f 002de7a8 759e2b71 00000611 00000000 002de7e4 COMDLG32!NewGetFileName+0x121 (FPO: [Non-Fpo])  40 002de7b8 759d9a9b 002de7f8 00050810 012245c0 COMDLG32!NewGetOpenFileName+0xf (FPO: [Non-Fpo])  41 002de7e4 759da33f 002de7f8 759d9603 012215d0 COMDLG32!GetFileName+0xcd (FPO: [Non-Fpo])  \*\*\* ERROR: Module load completed but symbols could not be loaded for MYInjector.exe  42 002df864 0122189f 012245c0 00000000 012215fc COMDLG32!GetOpenFileNameW+0x6a (FPO: [Non-Fpo]) |

实例  
给出这种注入方法的两个实现实例，相关模块源码可以在这里Clone：

|  |  |
| --- | --- |
| 1 | git clone git://git.code.sf.net/p/spoonshlext/code spoonshlext-code |

  
<P2>  
  
<P3>  
  
防御  
由于该种注入方法需要先向系统注册扩展模块，必须先写注册表项目，所以安全软件可以把相关的注册表路径添加到自己的HIPS防御规则中，例如以下某安全软件的做法。  
  
<P4>  
但是笔者在这里想说，作为安全软件，不应该仅仅提供这个最初的防御方法，更应该加强自身的安全性，去年给三家安全软件提交这个漏洞，只有一家做了写注册表提示，没有对自身进程加载的模块进行过滤，而其他两家连写注册表的提示都没有，所以很容易就入侵到了安全软件自身的进程地址空间。时隔一年现在再来看这个漏洞，那两家没有提示写注册表的厂商依然没有修正这个问题，而之前那家做了写注册表提示的厂家现在把自己的所有进程做的更加坚固了，直接过滤了除微软和自己签名的所有模块，这才是针对模块注入的终极防护，因为作为一款安全软件，如果随意让第三方模块进入自己的进程空间，那基本所有的防护就形同虚设了。不过在这里笔者要吐槽一下那家做的不错的厂商，因为当时笔者提交漏洞的时候给他提的解决方案如下：  
  
<P5>  
但是厂商给笔者的回复是  
<P6>  
但是现在看看，厂商还是注意到了笔者提出的第二点解决方案，并且在后续版本中实现了这一机制。由于此种注入方法利用度不高，所以介绍的相对浅显。  
  
SetWinodwsHookEx/SetWinEvetnHook全局钩子注入法  
原理  
通过Windows基于消息机制的Hook或者基于无障碍化event机制的Hook设置全局钩子，也是一种比较常见的注入方法，这种注入方法稳定高效，所以被用在很多地方，既然有名，当然也被各种安全软件封杀的比较厉害了。没见过有人分析过SetWindowsHookEx这个函数实现注入的原理，所以笔者带领各位读者跟随Windows内核源码来领略一下这种注入方法的本质吧。首先看一下SetWindowsHookExA/W的的函数原型

|  |  |
| --- | --- |
| 1  2  3  4  5 | HHOOK WINAPI SetWindowsHookEx(    \_In\_  int idHook,    \_In\_  HOOKPROC lpfn,    \_In\_  HINSTANCE hMod,    \_In\_  DWORD dwThreadId); |

然后存在如下函数调用链：  
SetWindowsHookExA/W→SetWindowsHookExAW →\_SetWindowsHookEx →NtUserSetWindowsHookEx →zzzSetWindowsHookEx  
以上调用关系链中最后两个函数已经进入到Windows内核中，而前面的所有函数调用无外乎都是进行参数转换以及合法性检查，其中参数转换就是把模块句柄转hMod转换成模块所在的全路径，相关代码位于SetWindowsHookExAW中;

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | HHOOK SetWindowsHookExAW(      int idHook,      HOOKPROC lpfn,      HINSTANCE hmod,      DWORD dwThreadID,      DWORD dwFlags)  {      WCHAR pwszLibFileName[MAX\_PATH];        /\*       \* If we're passing an hmod, we need to grab the file name of the       \* module while we're still on the client since module handles       \* are NOT global.       \*/  if (hmod != NULL) {      // 在这里获取传入的hMod的模块所在的全路径          if (GetModuleFileNameW(hmod, pwszLibFileName, sizeof(pwszLibFileName)/sizeof(TCHAR)) == 0) {                /\*               \* hmod is bogus - return NULL.               \*/              return NULL;          }      ………………………………  ………………………………      return \_SetWindowsHookEx(hmod,              (hmod == NULL) ? NULL : pwszLibFileName,              dwThreadID, idHook, (PROC)lpfn, dwFlags);  } |

然后，最终调用进入zzzSetWindowsHookEx，这就是设置钩子的核心函数了，由于这个函数篇幅比较长，所以这里只截取片段分析：

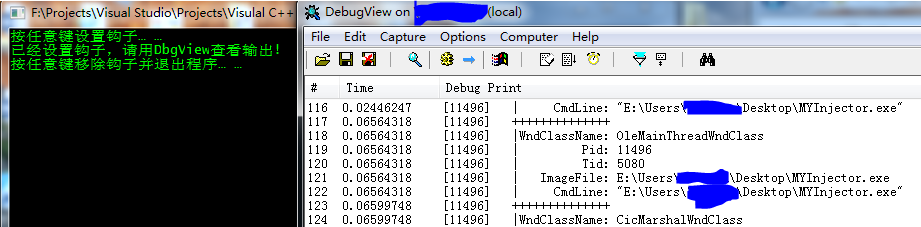
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121 | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\  \* zzzSetWindowsHookEx  \*  \* SetWindowsHookEx() is the updated version of SetWindowsHook().  It allows  \* applications to set hooks on specific threads or throughout the entire  \* system.  The function returns a hook handle to the application if  \* successful and NULL if a failure occured.  \*  \* History:  \* 28-Jan-1991 DavidPe      Created.  \* 15-May-1991 ScottLu      Changed to work client/server.  \* 30-Jan-1992 IanJa        Added bAnsi parameter  \\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/    PHOOK zzzSetWindowsHookEx(      HANDLE hmod,      PUNICODE\_STRING pstrLib,      PTHREADINFO ptiThread,      int nFilterType,      PROC pfnFilterProc,      DWORD dwFlags)  {      ACCESS\_MASK amDesired;      PHOOK       phkNew;      TL          tlphkNew;      PHOOK       \*pphkStart;      PTHREADINFO ptiCurrent;           ………..  ……篇幅原因 略去部分代码……  ………..        /\*       \* Allocate the new HOOK structure.    // 分配一个HOOK结构       \*/      phkNew = (PHOOK)HMAllocObject(ptiCurrent, ptiCurrent->rpdesk,              TYPE\_HOOK, sizeof(HOOK));      if (phkNew == NULL) {          return NULL;      }           ………..  ……篇幅原因 略去部分代码……  ………..          // 在模块路径添加到ATOM表中，并且返回Atom值          phkNew->ihmod = GetHmodTableIndex(pstrLib);            if (phkNew->ihmod == -1) {              RIPERR0(ERROR\_MOD\_NOT\_FOUND, RIP\_VERBOSE, "");              HMFreeObject((PVOID)phkNew);              return NULL;          }            /\*           \* Add a dependency on this module - meaning, increment a count           \* that simply counts the number of hooks set into this module.           \*/          if (phkNew->ihmod >= 0) {              AddHmodDependency(phkNew->ihmod);          }      }        /\*       \* Depending on whether we're setting a global or local hook,       \* get the start of the appropriate linked-list of HOOKs.  Also       \* set the HF\_GLOBAL flag if it's a global hook.       \*/      // 如果是本地Hook，则获取目标线程的Hool链表头      if (ptiThread != NULL) {          pphkStart = &ptiThread->aphkStart[nFilterType + 1];        ………..  ……篇幅原因 略去部分代码……  ………..        } else {          // 如果是全局Hook，则获取当前所在Desktop的全局Hook链表头          pphkStart = &ptiCurrent->pDeskInfo->aphkStart[nFilterType + 1];          phkNew->flags |= HF\_GLOBAL;            /\*           \* Set the WHF\_\* in the SERVERINFO so we know it's hooked.           \*/          ptiCurrent->pDeskInfo->fsHooks |= WHF\_FROM\_WH(nFilterType);            phkNew->ptiHooked = NULL;      }           ………..  ……篇幅原因 略去部分代码……  ………..      /\*       \* Initialize the HOOK structure.  Unreferenced parameters are assumed       \* to be initialized to zero by LocalAlloc().       \*/      // 记录Hook类型      phkNew->iHook = nFilterType;        /\*       \* Libraries are loaded at different linear addresses in different       \* process contexts.  For this reason, we need to convert the filter       \* proc address into an offset while setting the hook, and then convert       \* it back to a real per-process function pointer when calling a       \* hook.  Do this by subtracting the 'hmod' (which is a pointer to the       \* linear and contiguous .exe header) from the function index.       \*/      // 记录Hook处理函数在模块中的偏移      phkNew->offPfn = ((ULONG\_PTR)pfnFilterProc) - ((ULONG\_PTR)hmod);      /\*       \* Link this hook into the front of the hook-list.       \*/      // 将新的Hook结构添加到上述获取到的Hook链表头      phkNew->phkNext = \*pphkStart;      \*pphkStart = phkNew;           ………..  ……篇幅原因 略去部分代码……  ………..        return phkNew;  } |

从上面的代码中可以看出，当我们设置一个全局钩子的时候，系统只是在一个全局的钩子链表中记录了一个新的Hook结构，该结构中包括Hook处理函数所在的模块路径，以及Hook处理函数在该模块中的偏移，Hook类型等，那么系统在特定的Hook目标事件发生时候是怎样调用已经设置的Hook函数的呢？这里我们可以选择一个特定的钩子类型，例如WH\_GETMESSAGE，因为每次系统调用的GetMessage函数的时候都会调用这种类型的Hook处理函数，所以我们可以从GetMessage函数入手，首先看一下GetMessage函数从RING3到Ring0的调用链：User32!GetMessageA/W→ User32!NtUserGetMessage→Win32k!NtUserGetMessage→Win32k!xxxInternalGetMessage，关键代码就在Win32k!xxxInternalGetMessage这个函数里面了，在这个函数里面调用了一个特别的函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\*   \* If we're here then we have input for this queue. Call the   \* GetMessage() hook with this input.   \*/  if (IsHooked(ptiCurrent, WHF\_GETMESSAGE))      xxxCallHook(HC\_ACTION, flags, (LPARAM)lpMsg, WH\_GETMESSAGE); |

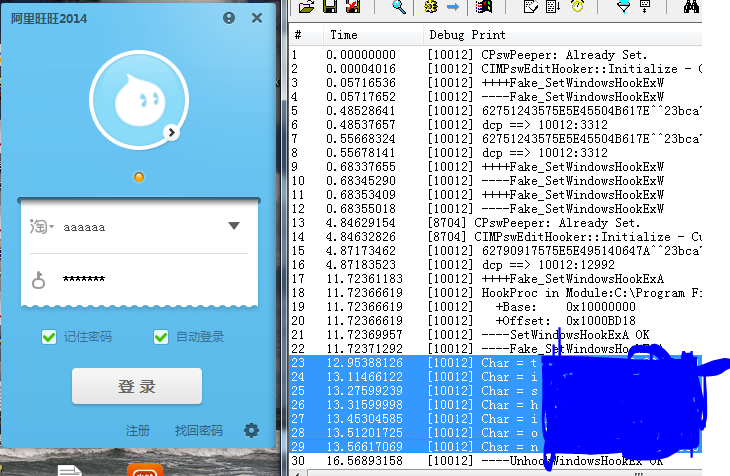
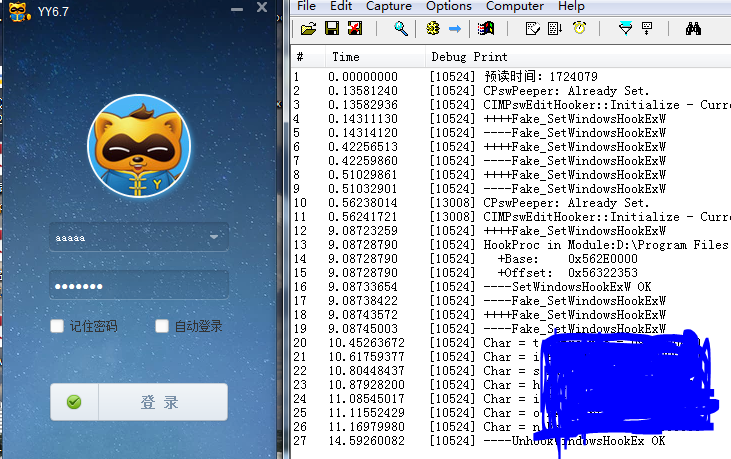
可以看到这段代码先是判断了当前线程是否被设置了WH\_GETMESSAGE类型的钩子，如果是的话就会调用xxxCallHook，那继续分析xxxCallHook，xxxCallHook只是对xxxCallHook2函数进行了一层包裹调用，由于xxxCallHook2函数篇幅实在过大，所以走只在这里概括介绍一下该函数的功能，该函数的内部会遍历当前进程中的本地Hook处理函数链，并且一次调用每一个Hook处理函数，然后会遍历当前Desktop下的全局Hook处理函数链，在处理每一个Hook结构的时候会检测该Hook处理函数所在的模块是否在当前进程空间内，如果不再则通过函数xxxLoadHmodIndex去把目标Dll模块加载到当前进程内，而xxxLoadHmodIndex函数内部首先根据传入的ATOM去查找之前保存的目标Dll的路径，然后通过ClientLoadLibrary这个函数去加载目标Dll模块，ClientLoadLibrary这个函数有经验的读者应该会很熟悉了，这个函数就是通过KeUserModeCallback函数从Ring0调用Ring3的回调函数LoadLibrary，这样就实现了目标进程主动加载一个Dll的机制，而这一点恰恰可以被当作一种稳定的注入方法。而对于SetWinEventHook实现的注入，原理与SetWindowsHookEx是相同的，所以这里不再结合代码做介绍。  
实例  
这里给出一个自己实现的WndSpy程序，注册一个全局的CBT钩子，然后在Hook处理函数中把当前创建的窗口信息以及当前进程信息都打印出来，相关源码可以在这里得到：

|  |  |
| --- | --- |
| 1 | git clone git://git.code.sf.net/p/wndspy/code wndspy-code |

  
<P7>  
防御  
目前所有的安全软件对这种注入的防御方法都是对SetWindowsHookE这个函数的动作进行监控，一旦发现有可疑程序设置全局钩子就会对用户进行提示，但是这要依赖对内核函数进行Patch，所以这种防御方法在X64系统下面就无法实现了，你会发现在x64系统下可以任意设置全局钩子，无论是32位进程还是64位进程。  
对于独立的应用软件来说，如果需要防止其他程序通过此种方法注入到自身进程，还是要在加载模块的源头LoadLibary这个函数调用链上下功夫，比如自己Hook住LdrpLoadDll这个函数，然后对所有加载的模块进行黑白校验。

Dll模块劫持注入方法  
原理  
Dll劫持严格意义来说不算一种注入方法，只能算是开发者在开发过程中犯的错误而已。Dll劫持的原理就是利用了Windows在加载模块时候有一个按照优先级排列的搜索路径，如果LoadLibrary或者通过倒入表导入的模块不能被直接找到，那么系统会按照优先级去使用搜索路径里面的每一项去构造一个模块路径，然后去尝试加载，例如系统预设的搜索路径有A,B,C,D四个，而目标模块实际位置是D\Mod.dll，那么如果该模块在被导入或者被加载的时候系统就会按照顺序去构造：  
A\mod.dll  
B\Mod.dll  
C\Mod.dll  
D\Mod.dll  
如果人为的在前三个路径上A\mod.dll，B\Mod.dll或者C\Mod.dll构造一个假的FakeMod.dll，那么系统就会直接加载最先构造的并且存在的模块，当然这只是Dll劫持的简单化解释，实际上在Windows系统中搜索顺序比较复杂，并且伴随着注册表以及LoadLibrary参数的影响，很多时候会把开发者搞的摸不着头脑。有一点需要声明一下，Dll劫持后必须是不改变原始模块的位置的一种欺骗行为，目前看到很多写外挂或者插件的开发者直接替换原始程序模块，然后把原始程序模块移动到其他目录，这种做法跟DLL劫持没有任何关系，而且这种做法并不是一种漏洞利用的手法，而是直接修改原软件作品的静态二进制代码的不高明手法。关于Dll劫持的详细介绍也是非常真多，可以参考笔者几年写的这篇文章：<http://blog.csdn.net/otishiono/article/details/7084079>  
  
另外需要补充说明的是LPK注入，这种方法在Windows XP时代很好用，但是在Windows 7（也可能是Vista，因为笔者没有用过vista，所以没有去查看这个）时代已经被微软终结了，因为微软也发现这个模块被太多人利用来劫持了，所以干脆直接再加入KnowsDll吧。  
实例  
这个实例比较多，比如Lpk.dll，ComRes.dll，还有以上介绍Dll劫持的文章中的msimg32.dll。  
防御  
首先当然要提高开发者的安全意识，让他们知道存在这一种导致安全漏洞的问题，至少在调用LoadLibrary函数的时候明确的给出全路径，至于显示链接产生的模块依赖，这个还是需要一个专门的审计系统去做的，不过一般规模的工程做起来都是非常容易的。当然有写有能力的公司会有一套很好的底层，比如封装一下LoadLibrary函数，改变默认搜索顺序。但是作为底层代码来说，如果提供给应用层的开发者使用的接口不能做到完全的考虑那么就必须花时间和人力成本去给应从层开发者介绍在使用这个底层接口时候需要注意什么，该怎么用等等。比如我现在所知的某个第一品牌的程序，就存在这样一个弄巧成拙的Dll劫持漏洞。

综合利用—白加黑的无敌杀伤力  
Ok，本文介绍到这里也是着实够枯燥无味的是，但是笔者认为还没有把自己想要表达的东西表达出来，现在笔者要说，请不要轻视这些注入方法带来的危害，尤其是Dll劫持这种问题。  
  
众所周知，现在的安全软件对Windows的挖掘已经非常非常的彻底了，几乎把Windows的骨架全部安装上了枷锁，所以系统变得“安全”（但是没有那个厂商可以可以否认拖慢了系统的运行速度）。诸如远程线程，APC，SetWindowsHookEx等等这些注入方法全部都被列入了各种HIPS规则的Notify或者Block列表中，而这一切都离不开一个至关重要的因素——数字签名。  
为什么微软的程序就可以随便SetWindowsHookEx，而你的程序却什么都不能做？因为你没有一个被洗白的签名……  
为什么QQ程序可以任意的写开机自动运行的注册表，而你的程序却会被安全软件拦截提示？因为你没有一个被洗白的签名……  
为什么搜狗的输入法就可以被任意进程自由加载，而你写的输入法注入模块就被直接查杀？因为你没有一个被洗白的签名……  
  
这说明，安全软件对签名的依赖是多么重要，安全软件在拦截到一个需要判断的操作之前，会首先获取发起该操作的进程链，然后通过对进程链中的所有进程进行黑白校验来判断该操作是否放行或者阻止。然而对单一进程的黑白校验仅仅是验证进程主模块文件的签名，但是我们知道一个进程发起一个操作，可以是exe模块发起的，当然也可以是任意一个dll模块发起的，说道这里读者应该明白什么是白加黑了。  
  
所谓白加黑就是利用一个具有合法并且被安全软件列入白名单的签名的进程，通过各种方法，将一个非法的模块注入到器地址空间，然后在该合法进程的庇护下实施一些恶意操作，从而绕过安全软件的提示和拦截。前段时间爆出来的“食猫鼠”，就是利用了某个安全厂商的一个exe进程，去加载一个恶意的Dll达到了很广的破坏力，当然他使用的方法不属于注入，而是仅仅把Exe当作一个合法的Loader，然后去加载他的恶意的模块而已。  
  
对于白加黑，目前来说，安全厂商并无十分好的对策，如果把判断黑白的粒度缩小到模块程度，那么就要在每个可疑操作时去判断当前模块的合法性，实现方法可以是栈回溯，也可以是应用层Context.Eip检验，但是这些方法势必会带来严重的性能影响，所以不适用，也正因为这样，白加黑目前来说在Windows平台下算是破坏力蛮大的。  
  
白加黑的初衷是要把一个非法模块加载到一个合法的进程中去，这就可以利用到注入技术了，但是在黑没有把白加黑利用起来之前，除了Dll模块劫持这种注入方法，其他的注入方法都会被安全软件拦截，所以Dll模块劫持几乎成了绕过安全软件的一个黄金钥匙，试想如果一个大公司的软件产品，用户量非常之大，然后这个产品又具有较高的启动权限，比如开机自启动，然后这个产品又存在Dll劫持漏洞……读者不要笑……然后读者自己脑补一下吧。安全软件最想说的话：不怕神一样的对手，就怕猪一样的队友。

如果读者脑补不出来，笔者在这里添加若干实例吧：  
TaobaoProtect.exe LoadLibraryA ( "OLEACC.dll" )  
TaobaoProtect.exe LoadLibraryA ( "RASAPI32.dll" )  
TaobaoProtect.exe LoadLibraryA ( "VERSION.dll" )  
TaobaoProtect.exe LoadLibraryA ( "WINHTTP.dll" )  
aliwssv.exe LoadLibraryA ( "NETAPI32.DLL" )  
这是笔者自己审计出来的几个Dll劫持的实例，还需要脑补利用？好吧，笔者利用上述的alwssv.exe的劫持漏洞实现了一个密码记录的模块（毫不夸张的说国内所有主流IM的密码框在笔者这里都是摆设，笔者都有无需任何硬编码的稳定记录方法）。  
总体思路是：  
1. 实现一个恶意模块名字叫做NetApi32.dll，放在aliwssv.exe所在目录下，一旦aliwssv.exe在开机时候通过服务管理器被启动，那么我们的这个NetApi32.dll就被加载到一个具有BAT签名的进程中了  
2. 我们的假的NetApi32.dll加载原始的NetApi32.dll，然后继续加载我们写的另一个恶意模块比如xxx.dll  
3. xxx.dll模块所做的事情主要是首先设置一个全局钩子，因为当前进程合法，所以安全软件不会有任何动作，全局钩子一旦设置，我们就相当于设置了一个全局的Monitor，只要在这个Mnotior中检测当前进程是感兴趣的进程，那就可以进行任何事情了，比如记录一下密码……  
  
<P8>  
  
<P9>  
  
还有一家，就不上图了。

好了本文大概是笔者憋了一年多的一篇文章，一直没有整理，这几天没有网络，闲来无事整理一下发出来来，也是为了完成自己每年两篇看雪精华帖的目标把，谢谢观赏。上传的附件：

©2000- 2017 看雪学院 | Based on [Xiuno BBS](http://bbs.xiuno.com/) | 域名 [加速乐](https://www.yunaq.com/) 保护 | SSL证书 [又拍云](https://www.upyun.com/) 提供 | 微信公众号：ikanxue

Time: **0.025**, SQL: **14** / 京ICP备10040895号-17