**向其它进程植入代码**

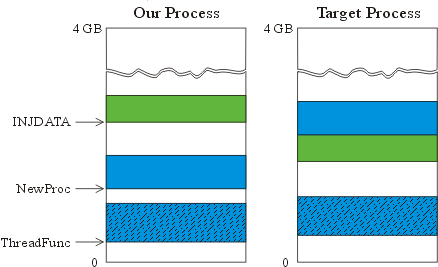
如何用该技术子类（subclass）一个远程控件  
示例程序：InjectEx

    让我们来讨论一个更复杂的问题：如何子类属于其他进程的一个控件？

    首先，要完成这个任务，你必须复制两个函数到远程进程：  
    1． ThreadFunc，这个函数通过调用SetWindowLong API来子类远程进程中的控件，  
    2． NewProc， 那个控件的新窗口过程（Window Procedure）。

    然而，最主要的问题是如何传递数据到远程的NewProc。因为NewProc是一个回调（callback）函数，它必须符合特定的要求（译者注：这里指的主要是参数个数和类型），我们不能再简单地传递一个INJDATA的指针作为它的参数。幸运的我已经找到解决这个问题的方法，而且是两个，但是都要借助于汇编语言。我一直都努力避免使用汇编，但是这一次，我们逃不掉了，没有汇编不行的。

解决方案1  
看下面的图片：



     不知道你是否注意到了，INJDATA紧挨着NewProc放在NewProc的前面？这样的话在编译期间NewProc就可以知道INJDATA的内存地址。更精确地说，它知道INJDATA相对于它自身地址的相对偏移，但是这并不是我们真正想要的。现在，NewProc看起来是这个样子：

|  |
| --- |
| static LRESULT CALLBACK NewProc(   HWND hwnd,       // handle to window   UINT uMsg,       // message identifier   WPARAM wParam,   // first message parameter   LPARAM lParam )  // second message parameter {     INJDATA\* pData = (INJDATA\*) NewProc;  // pData 指向                                           // NewProc;     pData--;              // 现在pData指向INJDATA;                           // 记住，INJDATA 在远程进程中刚好位于                            // NewProc的紧前面;      //-----------------------------     // 子类代码     // ........     //-----------------------------      //调用用来的的窗口过程;     // fnOldProc (由SetWindowLong返回) 是被ThreadFunc（远程进程中的）初始化     // 并且存储在远程进程中的INJDATA里的;     return pData->fnCallWindowProc( pData->fnOldProc,                                      hwnd,uMsg,wParam,lParam ); } |

    然而，还有一个问题，看第一行：

|  |
| --- |
| INJDATA\* pData = (INJDATA\*) NewProc; |

    pData被硬编码为我们进程中NewProc的地址，但这是不对的。因为NewProc会被复制到远程进程，那样的话，这个地址就错了。

    用C/C++没有办法解决这个问题，可以用内联的汇编来解决。看修改后的NewProc：

|  |
| --- |
| static LRESULT CALLBACK NewProc(   HWND hwnd,      // handle to window   UINT uMsg,      // message identifier   WPARAM wParam,  // first message parameter   LPARAM lParam ) // second message parameter {     // 计算INJDATA 的地址;     // 在远程进程中，INJDATA刚好在     //NewProc的前面;     INJDATA\* pData;     \_asm {         call    dummy dummy:         pop     ecx         // <- ECX 中存放当前的EIP         sub     ecx, 9      // <- ECX 中存放NewProc的地址         mov     pData, ecx     }     pData--;      //-----------------------------     // 子类代码     // ........     //-----------------------------      // 调用原来的窗口过程     return pData->fnCallWindowProc( pData->fnOldProc,                                      hwnd,uMsg,wParam,lParam ); } |

    这是什么意思？每个进程都有一个特殊的寄存器，这个寄存器指向下一条要执行的指令的内存地址，即32位Intel和AMD处理器上所谓的EIP寄存器。因为EIP是个特殊的寄存器，所以你不能像访问通用寄存器（EAX，EBX等）那样来访问它。换句话说，你找不到一个可以用来寻址EIP并且对它进行读写的操作码（OpCode）。然而，EIP同样可以被JMP，CALL，RET等指令隐含地改变（事实上它一直都在改变）。让我们举例说明32位的Intel和AMD处理器上CALL/RET是如何工作的吧：

    当我们用CALL调用一个子程序时，这个子程序的地址被加载进EIP。同时，在EIP被改变之前，它以前的值会被自动压栈（在后来被用作返回指令指针[return instruction-pointer]）。在子程序的最后RET指令自动把这个值从栈中弹出到EIP。

    现在我们知道了如何通过CALL和RET来修改EIP的值了，但是如何得到他的当前值？  
还记得CALL把EIP的值压栈了吗？所以为了得到EIP的值我们调用了一个“假（dummy）函数”然后弹出栈顶值。看一下编译过的NewProc：

|  |
| --- |
| Address   OpCode/Params   Decoded instruction -------------------------------------------------- :00401000  55         push ebp            ; entry point of                                                ; NewProc :00401001  8BEC            mov ebp, esp :00401003  51              push ecx :00401004  E800000000      call 00401009       ; \*a\*    call dummy :00401009  59         pop ecx            ; \*b\* :0040100A  83E909          sub ecx, 00000009   ; \*c\* :0040100D  894DFC          mov [ebp-04], ecx   ; mov pData, ECX :00401010  8B45FC          mov eax, [ebp-04] :00401013  83E814          sub eax, 00000014   ; pData--; ..... ..... :0040102D  8BE5            mov esp, ebp :0040102F  5D              pop ebp :00401030  C21000          ret 0010 |

    a． 一个假的函数调用；仅仅跳到下一条指令并且（译者注：更重要的是）把EIP压栈。  
    b． 弹出栈顶值到ECX。ECX就保存的EIP的值；这也就是那条“pop ECX”指令的地址。  
    c． 注意从NewProc的入口点到&ldquo;pop ECX”指令的“距离”为9字节；因此把ECX减去9就得到的NewProc的地址了。

    这样一来，不管被复制到什么地方，NewProc总能正确计算自身的地址了！然而，要注意从NewProc的入口点到“pop ECX”的距离可能会因为你的编译器/链接选项的不同而不同，而且在Release和Degub版本中也是不一样的。但是，不管怎样，你仍然可以在编译期知道这个距离的具体值。  
    1． 首先，编译你的函数。  
    2． 在反汇编器（disassembler）中查出正确的距离值。  
    3． 最后，使用正确的距离值重新编译你的程序。

    这也是InjectEx中使用的解决方案。InjectEx和HookInjEx类似，交换开始按钮上的鼠标左右键点击事件。

解决方案2

    在远程进程中把INJDATA放在NewProc的前面并不是唯一的解决方案。看一下下面的NewProc：

|  |
| --- |
| static LRESULT CALLBACK NewProc(   HWND hwnd,      // handle to window   UINT uMsg,      // message identifier   WPARAM wParam,  // first message parameter   LPARAM lParam ) // second message parameter {     INJDATA\* pData = 0xA0B0C0D0;    // 一个假值      //-----------------------------     // 子类代码     // ........     //-----------------------------      // 调用以前的窗口过程     return pData->fnCallWindowProc( pData->fnOldProc,                                      hwnd,uMsg,wParam,lParam ); } |

    这里，0XA0B0C0D0仅仅是INJDATA在远程进程中的地址的占位符（placeholder）。你无法在编译期得到这个值，然而你在调用VirtualAllocEx（为INJDATA分配内存时）后确实知道INJDATA的地址！（译者注：就是VirtualAllocEx的返回值）

    我们的NewProc编译后大概是这个样子：

|  |
| --- |
| Address   OpCode/Params     Decoded instruction -------------------------------------------------- :00401000  55                push ebp :00401001  8BEC              mov ebp, esp :00401003  C745FCD0C0B0A0    mov [ebp-04], A0B0C0D0 :0040100A  ... .... .... :0040102D  8BE5              mov esp, ebp :0040102F  5D                pop ebp :00401030  C21000            ret 0010 |

    编译后的机器码应该为：558BECC745FCD0C0B0A0......8BE55DC21000。

    现在，你这么做：  
    1． 把INJDATA，ThreadFunc和NewFunc复制到目的进程。  
    2． 改变NewPoc的机器码，让pData指向INJDATA的真实地址。  
    比如，假设INJDATA的的真实地址（VirtualAllocEx的返回值）为0x008a0000，你把NewProc的机器码改为：

|  |
| --- |
| 558BECC745FCD0C0B0A0......8BE55DC21000  <- 修改前的 NewProc 1     558BECC745FC00008A00......8BE55DC21000  <- 修改后的 NewProc |

    也就是说，你把假值 A0B0C0D0改为INJDATA的真实地址2  
    3． 开始指向远程的ThreadFunc，它子类了远程进程中的控件。

    ¹ 你可能会问，为什么A0B0C0D0和008a0000在编译后的机器码中为逆序的。这时因为Intel和AMD处理器使用littl-endian标记法（little-endian notation）来表示它们的（多字节）数据。换句话说：一个数的低字节（low-order byte）在内存中被存放在最低位，高字节（high-order byte）存放在最高位。  
想像一下，存放在四个字节中的单词“UNIX”，在big-endia系统中被存储为“UNIX”，在little-endian系统中被存储为“XINU”。

    ² 一些蹩脚的破解者用类似的方法来修改可执行文件的机器码，但是一个程序一旦载入内存，就不能再更改自身的机器码（一个可执行文件的.text段是写保护的）。我们能修改远程进程中的NewProc是因为它所处的那块内存在分配时给予了PAGE\_EXECUTE\_READWRITE属性。

    何时使用CreateRemoteThread和WriteProcessMemory技术

   通过CreateRemoteThread和WriteProcessMemory来注入代码的技术，和其他两种方法相比，不需要一个额外的DLL文件，因此更灵活，但也更复杂更危险。一旦你的ThreadFunc中有错误，远程线程会立即崩溃（看附录F）。调试一个远程的ThreadFunc也是场恶梦，所以你应该在仅仅注入若干条指令时才使用这个方法。要注入大量的代码还是使用另外两种方法吧。

    再说一次，你可以在文章的开头部分[下载](http://bbs.gdwg.net/" \t "_blank)到WinSpy，InjectEx和它们的源代码。

    写在最后的话

    最后，我们总结一些目前还没有提到的东西：  
   
    方法 适用的操作系统 可操作的进程进程      
    I. [Windows](http://bbs.gdwg.net/)钩子 Win9x 和WinNT 仅限链接了USER32.DLL的进程1      
    II. CreateRemoteThread & LoadLibrary 仅WinNT2 所有进程3，包括系统服务4      
    III. CreateRemoteThread & WriteProcessMemory 近WinNT 所有进程，包括系统服务

    1． 很明显，你不能给一个没有消息队列的线程挂钩。同样SetWindowsHookEx也对系统服务不起作用（就算它们连接了USER32）。  
    2． 在Win9x下没有CreateRemoteThread和VirtualAllocEx（事实上可以在9x上模拟它们，但是到目前为止还只是个神话）  
   3． 所有进程 ＝ 所有的Win32进程 ＋ csrss.exe  
    本地程序（native application）比如smss.exe, os2ss.exe, autochk.exe，不使用Win32 APIs，也没有连接到kernel32.dll。唯一的例外是csrss.exe，win32子系统自身。它是一个本地程序，但是它的一些库（比如winsrv.dll）需要Win32 DLL包括kernel32.dll.  
    4.如果你向注入代码到系统服务或csrss.exe，在打开远程进程的句柄（OpenProcess）之前把你的进程的优先级调整为“SeDebugprovilege”(AdjustTokenPrivileges)。

    大概就这些了吧。还有一点你需要牢记在心：你注入的代码（特别是存在错误时）很容易就会把目的进程拖垮。记住：责任随权利而来（Power comes with responsibility）！