在同一个进程空间运行两个程序（进程隐藏）

在同一个进程空间运行两个程序 运行环境：Windows NT4.0 / Windows 2000 关键字：进程隐藏，API 截获，映像加载

众所周知，bo2k 可以在一个指定的进程空间（比如 explorer.exe 进程）做为一个线程运行。 本文试图找出一种方法，使得任意 exe 都可以在其他进程中以线程运行（当然，这里说的 "任意" 是有条件的，下面会讲到）。为行文简单起见，我把先加载的 exe 称为宿主，后加载的 exe 称为客户。对于上面的例子，explorer.exe 为宿主，bo2k.exe 为客户。

基本知识
每一个 exe 都有一个缺省加载基址，一般都是 0x400000。 如果实际加载基址和缺省基址相同，程序中的重定位表就不需要修正（fixup），否则，就必须修正重定位表； 如果一个程序没有重定位表，而且如果程序不能在缺省基址处加载，那么程序将不能运行。 举个例子，Windows95 的最低加载基址是 0x400000，你在 Windows NT 上开发了一个 exe，指定其加载基址为 0x10000，如果连接时让连接器剥离重定位表，那么他将无法在 Windows95 下运行。

bo2k 为了避免和普通程序冲突，选了一个极其特殊的基址：0x03140000，这个地址一般不会有程序用到。这样 bo2k 启动后，用 WriteProcessMemory 将自身复制到宿主进程的 0x03140000 地址处， 再用 CreateRemoteThread 远程启动一个线程，从入口点开始执行。

bo2k 能够在其他进程空间正常运行，关键有两点：

实际加载基址和缺省基址相同，这样就无需修正重定位表。
与 bo2k 隐性联接 (implicitly link) 的动态联接库在目标进程中的加载基址和 bo2k 启动时的加载基址一致，这样就无需修改导入函数表。 除非只用到 ntdll.dll 和 kernel32.dll 两个 dll，否则这点很难保证。
bo2k 的解决办法是，远程运行的代码不用隐性调用，所有用到 API 都在远程代码运行后再动态确定（用 LoadLibrary 和 GetProcAddress）。

我的目标是让 "所有 " 的程序都能在其他进程空间跑。在这里，"所有 " 的含义是所有那些 "重定位表没有被剥离" 的 32 位 pe 格式的可执行程序。 对于 Visual C++，这包括所有 Debug 版程序和以 "/FIXED:NO" 选项链接的 Release 版程序。 对于一般的程序，上面两点都很难满足：

绝大多数程序的加载基址都是 0x400000，这样，客户 exe 就很难保证加载到其缺省基址。解决办法只能是修正重定位表。 如果，很不幸，这个 exe 的重定位表被剥离，这个 exe 就没法在其他进程空间跑。 对于 Visual C++，剥离重定位表是 Release 版 exe 的缺省设置。可以在工程文件的连接选项中加入 "/FIXED:NO" 来防止连接器剥离重定位表。
很多程序都用隐性联接调用 Windows API，而只用到 kernel32.dll 导出 API 的程序很少，因此这一点也很难保证。 解决办法是重填导入表 (import table)。
另外，对于有界面的程序，光修正重定位表和导入表还不够。因为他们都会直接或间接用到 GetModuleHandle 和 LoadResource 这些函数。 GetModuleHandle 有个特点，如果传递给他的 ModuleName 为 NULL，则返回宿主 exe 的模块句柄。LoadResource 也类似，如果传递给他的模块句柄为 NULL，则认为是宿主 exe 模块，类似的 API 还有一些，不一一列举。 客户 exe 调用这些 API 显然会得到错误的结果。因此必须截获这些 API 做特殊处理。

综合上面分析，要让两个程序共享一份进程空间，要做的工作有：

打开进程边界：用 WriteProcessMemory 向宿主进程注入代码，用 CreateRemoteThread 启动远程代码；
在远程代码中，加载客户 exe，必要时修正重定位表和填充 dll 导入表。
获 GetModuleHandle，LoadResource 等 API，在客户 exe 以缺省参数调用时返回客户 exe 的模块句柄，而不是宿主句柄。
根据以上思路，我写了 remote.dll，导出三个函数：RemoteRunA，RemoteRunW，和 RemoteCall。 原型分别为：

```
1 BOOL WINAPI RemoteRunA( DWORD processId, LPCSTR lpszAppPath, LPCSTR lpszCmdLine, int
  nCmdShow );
2 BOOL WINAPI RemoteRunW( DWORD processId, LPCWSTR lpszAppPath, LPCWSTR lpszCmdLine, int
  nCmdShow );
3 BOOL WINAPI RemoteCall( DWORD processId, PVOID pfnAddr, PVOID pParam, DWORD cbParamSize,
  BOOL fSyncronize );
```

RemoteRunA 用于在宿主进程中加载执行客户 exe；
RemoteRunW 是 RemoteRunA 的 unicode 版本；
RemoteCall 实现远程注入并运行代码。
调用例子： 假如宿主 exe 为 Depends.exe（我经常使用的宿主进程），pid 为 136。客户 exe 为 "C:\WINNT\system32\CALC.EXE",

```
RemoteRunA( 136, "C:\\WINNT\\system32\\CALC.EXE", NULL, SW_SHOW );
// 或
RemoteRunW( 136, L"C:\\WINNT\\system32\\CALC.EXE", NULL, SW_SHOW );
```

RemoteCall 是一个很 cool 的副产品，可以在任意宿主进程运行一系列你自己精心准备的代码。 远程代码无需特殊处理，就像在本地调用一样。RemoteCall 支持很多特性：

可以对 Windows API 进行隐性调用（无需用 LoadLibrary 和 GetProcAddress 动态确定）
可以使用全局 / 静态变量（除了不能动态初始化）；
可以使用编译时数据，特别是字符串常量；
支持异常处理；
支持源码级调试；
支持同步、异步调用；
对于同步调用，可以取得返回结果和错误号；
对远程代码做了异常保护，代码执行错误不会使宿主进程崩溃。
RemoteCall 的唯一缺点是效率不高（当然，还有一个缺点，你的 exe 必须是可重定位的）。

调用例子：

在 Windows 2000 中，对有密码保护风格的 Edit control 调用 SendMessage(hwnd, WM_GETTEXT, ...) 试图得到密码内容时， 系统会检查调用 SendMessage 的进程和 Edit control 所在的进程是否相同，不同则返回空字符串，调用失败。 解决办法显然应该是在目标进程中调用 SendMessage。 利用 RemoteCall，可以很容易地实现：

```
typedef struct _tagGETPASS {
   HWND hwndPassword; // in
   char szPassText[1024]; // out
} GETPASS;

static int *_p = NULL;
BOOL NullFunction() {
    // 可以用静态变量和异常保护。
    __try {
        *_p = 0;
    }__except(EXCEPTION_EXECUTE_HANDLER){}
    return TRUE;
}

// 准备在远程运行的代码
BOOL WINAPI RemoteGetPasswordText( GETPASS* pgp ) {
    // 可以使用相对调用 (near call)，没什么用，演示一下
    NullFunction();

    // 隐性调用 Windows API
    if ( SendMessageA( pgp->hwndPassword, WM_GETTEXT, sizeof(pgp->szPassText)-1,
(LPARAM)pgp->szPassText ) ) ) {
        MessageBoxA( NULL,
            pgp->szPassText,
            "Great!!", // 可以使用字符串常量
            MB_OK );
        return TRUE;
    }
    return FALSE;
}

void GetPasswordText( HWND hwnd ) {
    GETPASS gp;
    gp.hwndPassword = hwnd;
    DWORD processId;
    GetWindowThreadProcessId( hwnd, &processId );

    HMODULE hLib = ::LoadLibrary( "remote.dll" );
```

```
    if ( hLib != NULL ) {
        typedef BOOL (WINAPI *PFN_RemoteCall)( DWORD processId, PVOID pfnAddr, PVOID pParam, DWORD cbParamSize, BOOL fSyncronize );
        PFN_RemoteCall fnRemoteCall = (PFN_RemoteCall)::GetProcAddress( hLib, "RemoteCall" );
        if ( fnRemoteCall != NULL ) {
            if ( fnRemoteCall( processId, RemoteGetPasswordText, &gp, sizeof(gp), TRUE ) )
                MessageBoxA( NULL, gp.szPassText, "we get the password!!", MB_OK );
        }

        ::FreeLibrary( hLib );
    }
}
```

RemoteRun 的调用例子：

```
void PrintUsage() {
    printf( "\tUsage: rmExe <target process id> <Exe file path>\n" );
}

int main(int argc, char* argv[]) {
    if ( argc <= 2) {
        PrintUsage();
        return -1;
    }

    int pid = atoi( argv[1] );
    if ( pid != 0 ) {
        HMODULE hRemote = ::LoadLibrary( "remote.dll" );
        if ( hRemote != NULL ) {
            typedef DWORD (WINAPI *PFN_RemoteRun)( DWORD processId, LPCSTR lpszAppPath, LPSTR lpszCmdLine, int nCmdShow);
            PFN_RemoteRun fnRemoteRun = (PFN_RemoteRun)::GetProcAddress( hRemote, "RemoteRunA" );
            if ( fnRemoteRun != NULL )
                fnRemoteRun( pid, argv[2], NULL, SW_SHOW );
                FreeLibrary( hRemote );
        }
    }
    return 0;
}
```

应该注意的问题：

最困难的部分是加载客户 exe，简单的调用 LoadLibrary 根本不能解决问题，他不会替你修改重定位表和导入表。 另外对于 .tls section（用于支持线程本地存储）和 .bss section（用于为初始化数据），我目前还不是很清楚如何处理； 希望有人和我一起探讨；

目前 remote.dll 还不能支持在一个进程空间运行三个或更多程序。问题出在我在 remote.dll 中维护着一个客户 exe 的 thread 列表， 用于判断谁调用了 GetModuleHandle 等 API，目前只能处理一个客户 exe。这个问题不难解决；

有一些工具可以查看进程中加载的模块列表，如果想做进程彻底隐藏，不想让这些工具检测到我们的模块，在我看来，至少有两种解决办法：

不用 LoadLibrary，自己写 LoadDLL，这看起来似乎很困难，幸运的是，在 bo2k 的源代码中提供了一套这样的工具（在 dll_load.cpp 中实现）。 remote.dll 中修改重定位表和导入表基本上用的都是 dll_load.cpp 里的代码。值得注意的是，dll_load.cpp 原来的实现中有一点 bug，他不能正确处理有 Borland 的 tlink32 生成的 exe。 具体原因请仔细阅读 Matt pietrek 的 "Windows 95 system programming secrets"， 或 msdn 文章："Peering Inside the PE: A Tour of the Win32 Portable Executable File Format"，里面讲到了 ms linker 和 borland linker 的区别。

我自己实现了一种模块剥离技术，可以让进程脱离 .exe 文件和 .dll 文件运行。其思想是先对要剥离的 exe 或 dll 模块的所有数据做好备份，然后用 FreeLibrary 或者 UnmapViewOfFile 卸掉模块， 再把备份的模块数据恢复回来。我以前在 csdn 上贴过代码的，自己找吧。

截获 API 用的是 MS Detours Package 1.3。我不打算附上它的源代码，自己去下载吧：http://research.microsoft.com/sn/detours

在截获 API 时必须挂起其他线程。我用了两个未公开的接口：NtQuerySystemInformation 用于枚举线程；NtOpenThread 用于得到线程句柄。 推荐一本工具书："Windows NT Native API reference"（中文译名为 "Windows NT 本机 API 参考 "），书名大致如此，不必深究。气人的是居然把 Navtive 翻为本机，I 服了 you。书中列出了很多 Native API 的原型及其用到的数据结构。虽然翻译巨糙无比，但独此一家，别无选择，买一本参考参考还是值得的，如果你想研究 "本机" API 的话，:)。

运行成功的例子：

在 Depends.exe 进程中运行 Calc.exe；
在 Depends.exe 进程中运行 Acrobat 5.0；
在 Depends.exe 进程中运行 Microsoft Visio 2000；
在 Depends.exe 进程中运行 Process Hacker（我自己写的一个进程查看工具），用了很多低层接口；
在 Process Hacker 进程中运行 Acrobat 5.0。
唯一失败的例子是以客户身份运行 matlab 5.1。这个可执行文件很特殊，有多个 code section 和 data section，还有 .tls section 和 .bss section。 失败原因不是很清楚（主要是没有足够的时间研究），可能是 .tls 和 .bss section 在加载时没有处理好； 也可能是某个应该做特殊处理的 API 没有拦截处理。