DOI:10.16184/j.cnki.comprg.2008.03.005

进 程 管 理 器

王国松

摘 要 开发的进程管理器，以进程管理功能为主，兼具模块(DLL)管理、服务管理、启动项管理、驱动查询、端口扫描、窗口扫描等实用功能，适合用于查杀木马病毒。介绍了实例程序主要功能的实现思路和方法，并给出了关键源代码。

关键词 进程，模块，服务，端口，窗口，启动项

一、引言

进程管理器在系统维护过程中起着十分重要的作用。当发现某个程序停止响应，或是怀疑有病毒程序在运行时，自然就会想到利用进程管理器查看内存中运行着的程序及进程列表，强制结束无响应的程序或病毒程序。虽然 Windows 任务管理器的功能较多，但缺少诸如枚举进程引用模块、获取进程的路径或命令行、进程端口扫描、驱动及服务列表等重要功能，于是网友们开发出现了不少好的进程管理软件，以弥补 Windows 任务管理器功能的不足。笔者自己也动手开发了一个进程管理器，实用效果较好，现奉献给广大读者。

二、进程管理器功能

笔者自编的进程管理器可运行在 WindowsXP/2000 环境下，以进程管理功能为主，兼具模块(DLL)管理、服务管理、启动项管理、驱动查询、端口扫描、窗口扫描等实用功能，最适合用于查杀木马病毒。实例程序的各功能简介如下。

1. 进程管理

枚举进程、向远程进程空间注入 DLL、结束进程、设置进程优先级、设置进程运行权限、隐藏进程、复制进程路径或命令行、查询进程相关信息(进程 ID、用户及账号、路径或命令行、内存占用、优先级等)，如图 1 所示。

2. 模块管理

枚举进程引用的 DLL、从远程进程空间卸载 DLL、注册/注销 DLL、删除 DLL 文件、打开 DLL 所在文件夹、复制 DLL 文件名、查询 DLL 相关信息(入口地址、文件版本、宿主进程等)，如图 2 所示。

3. 服务管理

枚举服务（包括 Win32 应用服务、系统内核服务）、设置服务启动方式、启动服务、停止服务、删除服务、删除服务设备、复制服务设备名，如图 3 所示。

图 1 进程管理界面

图 2 模块管理界面

图 3　Win32 应用服务管理界面

4. 设备查询

查询内核驱动设备，界面如图 4 所示。

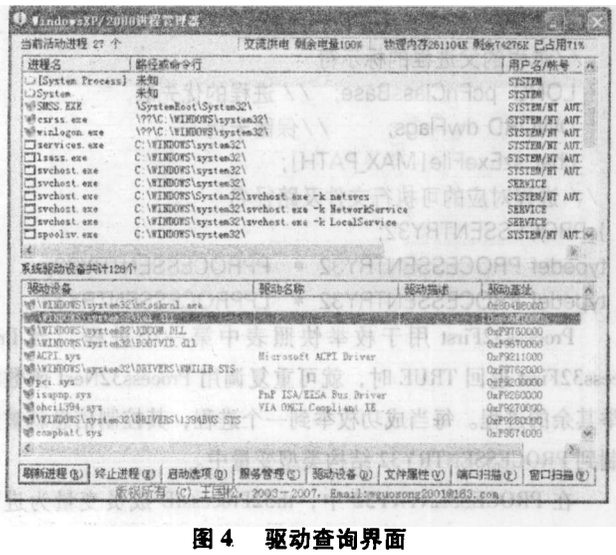

图 4　驱动查询界面

5. 启动选项

可查询、修改、添加、删除注册表中以下启动项设置：Run、RunOnce、RunOnceEX、RunServices、RunServicesOnce、Explorer Run、Windows、Winlogon、Session Manager、Command Processor、ShellExcuteHooks、Browser Help Objects，还可启用/禁用注册表编辑器，运行界面如图 5 所示。

6. 窗口管理

扫描窗口、鼠标捕捉窗口、显示/隐藏窗口、启用/禁用窗口、关闭窗口、获取密码、向窗口发送消息，如图 6 所示。

7. 端口扫描

扫描进程使用的端口及 IP 地址，只适用 WindowsXP 系统，运行界面如图 7 所示。

8. 其他功能

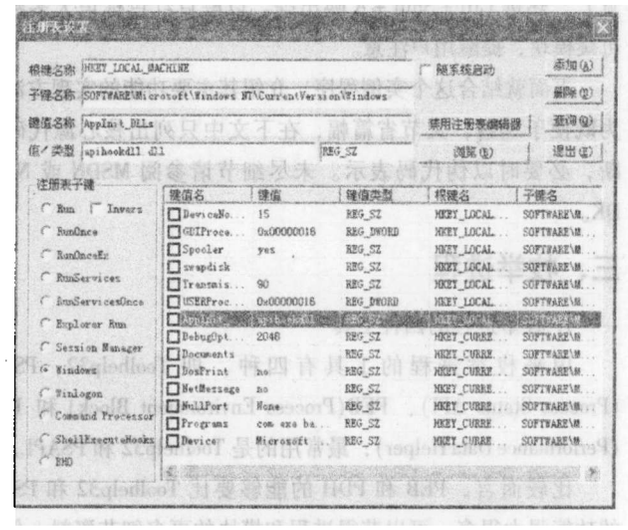

图 5　启动项管理界面

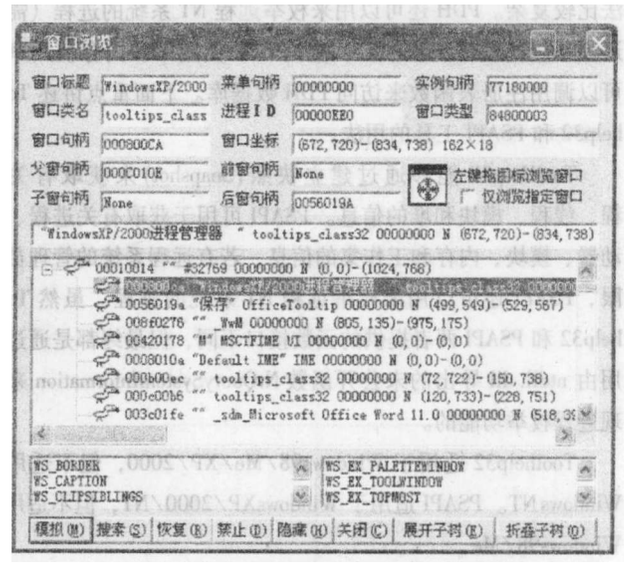

图 6　窗口管理界面

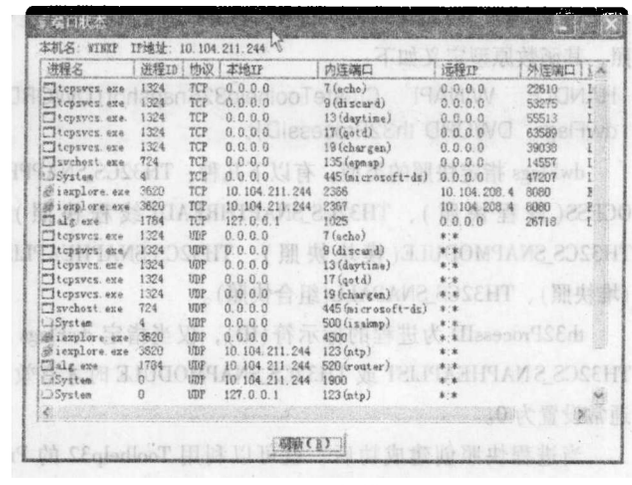

图 7　端口扫描界面

支持系统托盘、右键快捷菜单、文件拖放、列表框排序、列表框单元格颜色设置、列表框宽度设置（自动设置、各列等

宽）、热键 Ctrl + Shift + A 唤出等。以醒目红色标识大多数的可疑模块，提醒用户注意。

下面就结合这个实例程序，介绍其主要功能的实现方法。为既便于理解，又节省篇幅，在下文中只列出核心源代码片断，必要时以伪代码表示。未尽细节请参阅 MSDN 或 NTDDK。

三、枚举进程

1. 枚举进程的四种工具

用来枚举进程的工具有四种，即 Toolhelp32、PSAPI (Process Status API)、PEB(Process Environment Block) 和 PDH (Performance Data Helper)，最常用的是 Toolhelp32 和 PSAPI。

比较而言，PEB 和 PDH 的能够要比 Toolhelp32 和 PSAPI 的功能强大得多，可以获得进程和模块的更多细节资料，但用法比较复杂。PDH 还可以用来枚举远程 NT 系统的进程（需有远程系统的管理员权限），查询 CPU 使用率和内存使用率。可以调用注册表函数来访问 PDH 数据库。下面重点讲述 Toolhelp32 和 PSAPI 工具的用法。

ToolHelp32 能够通过建立快照 (Snapshot) 来获取有关进程、线程、模块和堆的信息。PSAPI 可用于获取有关进程、驱动器、模块、内存和工作集的信息，若有远程系统的管理员权限，PSAPI 还可以用来枚举远程 NT 系统的进程。虽然 Toolhelp32 和 PSAPI 两者提供的函数接口不同，但最终都是通过调用由 ntdll.dll 导出的未公开函数 NtQuerySystemInformation 来实现进程枚举功能的。

Toolhelp32 适用于 Windows98/Me/XP/2000，但不适用于 Windows NT。PSAPI 适用于 WindowsXP/2000/NT，但不适用于 Windows98/Me。

2. 使用 Toolhelp32 枚举进程

Toolhelp32 的 CreateToolhelp32Snapshot 函数用于创建快照，其函数原型定义如下：

```
HANDLE WINAPI CreateToolhelp32Snapshot(DWORD dwFlags, DWORD th32ProcessID);
```

dwFlags 指定快照的类型，有以下几种：TH32CS_SNAPPROCESS(进程快照)、TH32CS_SNAPTHREAD(线程快照)、TH32CS_SNAPMODULE(模块快照)、TH32CS_SNAPHEAPLIST(堆快照)、TH32CS_SNAPALL(组合快照)。

th32ProcessID 为进程的标示符(ID)，仅当指定 dwFlags 为 TH32CS_SNAPHEAPLIST 或 TH32CS_SNAPMODULE 时才有效，通常设置为 0。

当进程快照创建成功后，就可以利用 Toolhelp32 的 Process32First 及 Process32Next 函数枚举所有的进程。

```
PROCESSENTRY32 p32;
p32.dwSize = sizeof(PROCESSENTRY32);
HANDLE hSnapshot = CreateToolhelp32Snapshot
(TH32CS_SNAPPROCESS, 0);
if(Process32First(handle, & p32))
{
.....
while(Process32Next(handle, & p32))
.....
}
```

hSnapshot 为 CreateToolhelp32Snapshot 返回的进程快照表的句柄，lppe 为指向进程控制块的 LPPROCESSENTRY32 结构类型变量的指针，用于存储进程控制信息。LPPROCESSENTRY32 结构的定义如下：

```
typedef struct tagPROCESSENTRY32 {
    DWORD dwSize;              // 结构的大小
    DWORD cntUsage;            // 进程的引用计数
    DWORD th32ProcessID;       // 进程的标示符
    DWORD th32DefaultHeapID;  //进程默认堆的标示符
    DWORD th32ModuleID;        // 进程模块的标示符
    DWORD cntThreads;          // 进程所含的线程数
    DWORD th32ParentProcessID;
   //进程的父进程的标示符
    LONG pcPriClassBase;  // 进程的优先级
    DWORD dwFlags;        //保留未用
    char szExeFile[MAX_PATH];
  // 进程对应的可执行文件及路径名
} PROCESSENTRY32;
typedef PROCESSENTRY32 *  PPROCESSENTRY32;
typedef PROCESSENTRY32 *  LPPROCESSENTRY32;
```

Process32First 用于枚举快照表中第一个进程。当 Process32First 返回 TRUE 时，就可重复调用 Process32Next 函数枚举其余的进程。每当成功枚举到一个进程，其控制信息就被复制到 PROCESSENTRY32 结构类型变量中。

在 PROCESSENTRY32 中，th32ProcessID 成员变量为进程的标识符(ID)，它可以被传给 OpenProcess 以获得该进程的句柄，pcPriClassBase 为进程的优先级。它分为以下几个级别：0 - 暂缺，4 - 低，5 至 7 - 低于标准，8 - 标准，9 至 12 - 高于标准，13 - 高，24 - 实时。系统允许用户改变进程的优先级。th32ParentProcessID 为父进程的标识符，据此可以构造进程树。

3. 使用 PSAPI 枚举进程

WindowsXP/2000/NT 环境下，创建进程快照表可使用 PSAPI 提供的一组函数，这些函数由 psapi.dll 导出。编译时需要 psapi.h 和 psapi.lib 这两个文件，可在 Platform SDK 中可以找到，也可以从网上单独下载。

使用 PSAPI 创建进程列表的第一步是调用 EnumProcesses，其函数原型定义：

```
BOOL EnumProcesses(DWORD * lpidProcess, DWORD
```

cbSize, DWORD * cbNeeded);

lpidProcess 为指向存放返回进程列表的数组指针，cbSize 为预先定义的该数组的大小，cbNeeded 用于存放实际返回的数组大小，使用算式 cbNeeded/sizeof(DWORD) 可以得到返回的进程数目。

由于 EnumProcesses 不会在 cbNeeded 中返回一个大于 cbSize 参数传递值，为确保 EnumProcesses 函数调用成功，若出现返回的 cbNeeded = cbSize 值的现象，就必须增大 cbSize 的值，并重复调用 EnumProcesses 一直到 cbNeeded < cbSize 为止，才能枚举到所有的进程。为简单起见，可为 cbSize 预置一个足够大的值如 1024，绝大多数情况下都能够保证 EnumProcesses 调用成功。

四、关联信息

虽然进程控制块以提供了较多的信息，但缺少一些诸如进程路径名、命令行、用户名、账号、可视窗口等，需要通过其他途径来获取。

1. 路径名

许多病毒制造者为迷惑用户，有意将病毒文件名伪装成系统文件名，常见的如 smss.exe、crss.exe、winlogong.exe、sertvices.exe、lsass.exe、svchost.exe 等，使一般用户难以辨别出哪些是系统进程，哪些是病毒进程。若能获取进程对应的可执行文件名及路径名，就容易区分出病毒进程，因为同一目下的每个文件名都具有唯一性，与系统文件同名的病毒文件是不可能与系统文件同时存放在同一目录下的。

当使用 Toolhelp32 枚举进程时，Process32First 及 Process32Next 返回的进程信息存放 PROCESSENTRY32 结构类型变量中。在 Windows9X/Me 环境下，此结构的 szExeFile 成员变量存放的是进程对应的可执行文件名及其路径全名，而在 WindowsXP/2000/NT 环境下，szExeFile 成员变量存放的仅是进程对应的可执行文件名，但不包括可执行文件所在路径名，此时要想获得路径名就必须调用 PSAPI 的 GetModuleFileNameEx 函数，代码如下：

```
char szProcessName[MAX_PATH] = {0};
HMODULE hMod;
DWORD cbNeeded;
HANDLE hProcess = OpenProcess
(PROCESS_QUERY_INFORMATION |PROCESS_VM_READ,
FALSE, ProcessID);
if (EnumProcessModules(hProcess, & hMod, sizeof
(hMod), & cbNeeded))
GetModuleFileNameEx(hProcess, hMod, szProcessName,
sizeof(szProcessName));
CloseHandle(hProcess);
```

ProcessID 为目标进程的标识符。EnumProcessModules 将返回目标进程引用第一个模块（即进程自身）的句柄，并将其保存到变量 hMod 中，GetModuleFileNameEx 返回的进程可执行文件名及路径名。

如果将 hMod 定义为一个足够大的数组，那么 EnumProcessModules 将返回目标进程引用的所有模块。如果用 GetModuleBaseName 来替代 GetModuleFileNameEx，则仅能够获得进程的可执行文件名。

有个别系统进程的路径名比较特殊，类似于“\SystemRoot\System32\”或“\??\C: WINDOWS\system32\”，这是 kernel32.dll 导出函数 Bug 造成的，其实都表示系统目录。

2. 命令行

枚举到进程并且知道了进程对应的可执行文件及路径后，如能获取启动进程的命令行将是十分有用的，由此可以了解进程启动时附加了哪些参数，可以帮助有经验的用户快速查找病毒进程。

在 WindowsXP/2000/NT 环境下，许多进程(特别是在后台运行服务类的进程)是通过 svchost.exe 来加载的。而 Windows 自带的任务管理器，包括网上流行的多个进程管理器，因不具备进程命令行参数显示功能，都只会显示多个 svchost.exe 进程，但无法帮助用户区分每个 svchost.exe 进程所加载的模块，对于 rundll32.exe 进程也有类似的情况。

如何获得进程的命令行呢？这个问题对于 DOS 程序的编写者来说比较简单，但对 MFC 应用程序编写者来说就没有那么简单了。虽然有现成的 GetCommandLine 函数可供调用，但它只能返回当前进程的命令行，对其他进程无效，而要获取所有进程的命令行，这就需要调用另外一个未公开的函数 NtQueryInformationProcess，设法从进程环境块 PEB 中读取命令行。

第一步，调用 OpenProcess 打开进程，注意其 dwDesiredAccess 参数必须指定为 PROCESS_QUERY_INFORMATION | PROCESS_VM_READ；

第二步，调用 NtQueryInformationProcess 用于得到目标进程的进程环境块 PEB，其函数原型定义如下：

```
NTSTATUS NtQueryInformationProcess(IN HANDLE ProcessHandle,
    IN PROCESSINFOCLASS ProcessInformationClass,
    OUT PVOID ProcessInformation, IN ULONG ProcessInformationLength,
    OUT PULONG ReturnLength OPTIONAL);
```

NtQueryInformationProcess 函数由 ntdll.dll 导出，需调用 GetProcAddress 获得其地址。ProcessInformationClass 是一个可变的枚举类型，在此需要取值为 0，即可复制进程基本信息块到自定义的 PROCESS_BASIC_INFORMATION 结构类型变量中，其成员变量 PebBaseAddress 指明了进程环境块 PEB 的首地址；

第三步，调用 ReadProcessMemory 从 PebBaseAddress 指明

的首地址开始复制进程环境块信息到 PEB 结构类型变量中，其成员变量 ProcessParameters 指明了进程命令行首地址；

第四步，调用 ReadProcessMemory 从 ProcessParameters 指明的首址开始复制进程命令行到 PROCESS_PARAMETERS 结构类型变量中，其成员变量 CommandLine. Buffer 和 CommandLine. Length 分别指明了进程命令行首址和长度；

第五步，调用 ReadProcessMemory 从 CommandLine. Buffer 指明的首址开始复制命令行到一个 LPWSTR(Unicode 字符)类型指针指向的缓冲区中；

第六步，调用 WideCharToMultiByte 将返回到缓冲区的命令行字符串由 Unicode 格式转换为 ANSI 格式。

```
PEB Peb;
PROCESS_PARAMETERS ProcParam;
PROCESS_BASIC_INFORMATION pbi;
char cmdline[MAX_PATH] = {0}, buf[MAX_PATH] = {0};
LPVOID lpBuf;
DWORD dwRead, dwLen;
typedef LONG (WINAPI * PROCNTQIP) (HANDLE, UINT,
PVOID, ULONG, PULONG);
PROCNTQIP lpNtQIP;
lpNtQIP = (PROCNTQIP)GetProcAddress(
GetModuleHandle( " ntdll"), " NtQueryInformationPro-
cess");
HANDLE hProcess = OpenProcess
(PROCESS_QUERY_INFORMATION|
PROCESS_VM_READ, FALSE, ProcessID);
LONG status = lpNtQIP( hProcess, 0, (PVOID)& pbi,
sizeof(PROCESS_BASIC_INFORMATION), NULL);
ReadProcessMemory( hProcess, pbi. PebBaseAddress, &
Peb, sizeof(PEB), & dwRead);
ReadProcessMemory( hProcess, Peb. ProcessParameters,
& ProcParam,
sizeof(PROCESS_PARAMETERS), & dwRead);
lpBuf = ProcParam. CommandLine. Buffer;
dwLen = ProcParam. CommandLine. Length;
ReadProcessMemory( hProcess, lpBuf, (LPWSTR) buf,
dwLen, & dwRead );
WideCharToMultiByte( CP_ACP, 0, (LPWSTR)cmdline, -
1, buf, sizeof( buf ), NULL, NULL );
CloseHandle (hProcess);
```

buf 中存放的是 Unicode 格式命令行字符串，cmdline 中存放的是 ANSI 格式命令行字符串。cmdline 和 buf 两个缓冲区的大小至少要等于 dwLen。

这里说明一下 Unicode 格式字符串集转和 ANSI 格式的字符串的区别。假设有一个 ANSI 格式字符串“ABC”，其对应的 Unicode 格式字符串则为‘A’，0，‘B’，0，‘C’，0。由此可见，后者是将前者的每个字节扩展为双字节，且双字节的高位一律为 0。

上面用到的 PROCESS_BASIC_INFORMATION、PEB 和 PROCESS_PARAMETERS 三个自定义结构是由 NTDDK 提供的。

3. 用户名及账号

作为系统管理员，有时可能需要了解每个进程的拥有者，即其用户名、账号和密码。

要想获取进程的用户名和账号，应首先调用 OpenProcessToken 打开进程的令牌(Token)，打开方式设为 TOKEN_QUERY，然后将 TOKEN_INFORMATION_CLASS 设为 TokenUser 调用 GetTokenInformation 枚举进程的令牌信息块 SID，通常称为安全标识(Security Identifier)，最后调用 LookupAccountSid 从 SID 中检索用户名和账号。

```
char buf[256], szUser[128], szComputer[128];
HANDLE hProcess, hToken;
DWORD cbUser, cbComputer;
SID_NAME_USE snu;
if(ProcessID < = 0x0c)
    显示用户名为"SYSTEM";
Else
    显示用户名为"SERVICE ";
hProcess =OpenProcess
(PROCESS_QUERY_INFORMATION, 0, ProcessID);
OpenProcessToken(hProcess, TOKEN_QUERY, & hTo-
ken);
GetTokenInformation(hToken, TokenUser, & buf, 256, &
cbUser);
cbComputer =sizeof(szComputer);
LookupAccountSid(NULL, (DWORD *)( * (DWORD *)
buf), szUser,
 & cbUser, szComputer, & cbComputer, & snu);
CloseHandle(hToken);
CloseHandle(hProcess);
```

代码中 ProcessID 为目标进程的标识符。标识符小于 0x0c 的进程为系统核心进程，在 Windows 任务管理器中显示其用户名为“SYSTEM”。GetTokenInformation 的第二个参数指明要枚举的令牌信息块类型，在这里将其指定为 TokenUser，buf 缓冲区用于接收返回的令牌信息块 SID。SID 的首个双字为一指针，此指针指向一个地址，此地址内又存放着块内用户名及账号字符串的地址。需要注意的是，在调用 LookupAccountSid 前，必须为变量 cbComputer 赋值，其大小为预先分配的 szComputer 缓冲区的大小，并且以 (DWORD *) (* (DWORD *) buf) 方式来指定作为输入参数的令牌信息块 SID 内用户名及账号字符串的首址。返回的用户名和账号分别存放到变量 szUser 和 szComputer 中。

在 WindowsXP/2000/NT 中，所有用户的登录密码都存放在 Winlogon. exe 进程的地址空间内，可以通过调用 NtQuerySystemInformation 枚举系统信息块获得。考虑到安全性，本文对如

何获得用户密码的细节不作介绍。

4. 可视窗口

查找进程的可视窗口，查看窗口标题，有助于了解进程的功能。

既然进程是一个正在运行着的可执行程序的实例，那么它就可能就会有可视窗口，也可能没有可视窗口(如多数后台服务类进程)。一个进程若有可视窗口的话，则其可视窗口可能是一个，也可能是多个，典型的如 Explorer. exe 和 Iexplore. exe 两个进程，各自都有多个可视窗口。

给定一个目标进程，从桌面窗口开始枚举所有顶层窗口，逐个判别是否具有 WS_VISIBLE 特性，若有的话再调用 GetWindowThreadProcessId，判别其返回的窗口的进程标识符是否与给定的目标进程标识符相同，如相同则找到了目标进程的一个可视窗口，接着再判别其余的顶层窗口是否为目标进程的可视窗口。重复上述步骤，即可找到所有目标进程的可视窗口。

```
DWORD pID;
hWndNext =:: GetWindow( :: GetDesktopWindow( ),
GW_CHILD);
while(hWndNext)
{
  if(GetWindowLong(hWnd, GWL_STYLE) & WS_VI
SIBLE)
      GetWindowThreadProcessId(hWnd, & pID);
  if(pID = =ProcessID & & :: GetParent(hWnd) = =
NULL)
//记录找到的目标进程的一个可视窗口
}
hWndNext =: : GetWindow(hWndNext,
 GW_HWNDNEXT);
```

如果去掉匹配目标进程标识符 ProcessID 这一限制条件，用上述方法可以找到系统所有可视窗口，即在 Windows 任务管理器“应用程序”栏内看到的内容。

5. 打开的文档

通过调用 DeviceIoControl 可以很容易地查询到 Windows98/ME 进程打开的文档，但不能查询 WindowsXP/2000 进程打开的文档，只能查询其磁盘设备。

要想查询 WindowsXP/2000 进程打开的文档，可靠的方法是创建一个管道专门用来接收应用程序输出，通过重定向输出得到进程打开的所有文档列表。由于具体实现方法十分复杂，笔者在这里只提供思路，供有兴趣的读者自己研究。

五、枚举进程引用的模块

与枚举进程一样，枚举进程引用模块所使用的工具也是 Toolhelp32、PSAPI、PEB 和 PDH，常用的也是 Toolhelp32 和 PSAPI。

1. Toolhelp32 枚举模块

在前面“使用 Toolhelp32 枚举进程”一节已经提到 CreateToolhelp32Snapshot 也可以创建进程引用的模块快照，只需将第一个参数 dwFlags 设为 TH32CS_SNAPMODULE，将第二个参数 th32ProcessID 设为目标进程的标识符即可，实现的核心代码如下：

```
MODULEENTRY32 m32;
m32. dwSize = sizeof(MODULEENTRY32);
HANDLE hSnapshot = CreateToolhelp32Snapshot
(TH32CS_SNAPMODULE, ProcessID);
if(Module32First (handle, & m32))
{
……
While(Module32Next(handle, & m32))
……
}
```

当模块快照创建成功后，就可以利用的 Module32First 及 Module32Next 函数枚举目标进程引用的所有模块。

Module32First 用于枚举快照表中第一个进程。当 Module32First 返回 TRUE 时，就可重复调用 Module32Next 函数枚举其余的进程。每当成功枚举到一个进程，其控制信息就被复制到 MODULEENTRY32 结构类型变量中。MODULEENTRY32 结构定义如下：

```
typedef struct tagMODULEENTRY32 {
   DWORD   dwSize;
   DWORD   th32ModuleID; // 模块标识符
   DWORD   th32ProcessID; // 引用此模块进程的标识符
   DWORD   GlblcntUsage;  // 模块的引用计数(全局)
   DWORD   ProccntUsage;  // 模块的引用计数(局部)
   BYTE  * modBaseAddr;   // 模块基地址
   DWORD   modBaseSize;   // 模块大小
   HMODULE hModule;       // 模块句柄
   char    szModule[MAX_MODULE_NAME32 + 1];
   // 模块名
   char    szExePath[MAX_PATH];   // 模块路径
} MODULEENTRY32;
typedef MODULEENTRY32 *  PMODULEENTRY32;
typedef MODULEENTRY32 *  LPMODULEENTRY32;
```

hModule 是一个十分重要的成员变量，有些专杀注入型木马病毒的软件正是由此获得病毒模块句柄的。

2. PSAPI 枚举模块

在前面“获取进程的路径名”一节里，已经提到使用 EnumProcessModules 来枚举目标进程引用的模块，其第二个参数 lphModule 为指向存放返回模块句柄列表的数组指针，cbSize 为预先定义的该数组的大小，lpcbNeeded 用于存放实际返回的数组大小，使用算式 * lpcbNeeded/sizeof(HMODULE)可以得到返回的模块句柄数目，代码如下：

```
HANDLE hProcess;
    DWORD cbNeeded, cbSize = 1024;
    HMODULE hMods[cbSize];
    MODULEINFO mf;
    char szModName[MAX_PATH] = {" 无 "};
    unsigned int i;
    hProcess = OpenProcess
(PROCESS_QUERY_INFORMATION |
PROCESS_VM_READ,
FALSE, ProcessID);
    if(EnumProcessModules(hProcess, hMods, sizeof
(hMods), & cbNeeded))
        for (i = 0; i<(cbNeeded/sizeof(HMODULE));
i++)
        {
            GetModuleInformation(hProcess, hMods[i],
& mf, sizeof(MODULEINFO));
            GetModuleFileNameEx(hProcess, hMods[i],
szModName, sizeof(szModName));
        }
    CloseHandle(hProcess);
```

ProcessID 为目标进程标识符，返回的模块信息存放在 MODULEINFO 结构类型变量中，GetModuleFileNameEx 返回的模块文件名及路径名存储在 szModName 中。

与调用 EnumProcesses 的情形很类似，可为 cbSize 预置一个足够大的值如 1024，绝大多数情况下都能够保证 EnumProcessModules 调用成功。

MODULEINFO 结构定义如下：

```
typedef struct _MODULEINFO {
    LPVOID lpBaseOfDll;
    DWORD SizeOfImage;
    LPVOID EntryPoint;
} MODULEINFO, *LPMODULEINFO;
```

成员变量 lpBaseOfDll 为模块的句柄，SizeOfImage 为模块的大小，EntryPoint 为模块代码段在目标进程地址空间内的起始地址，并不是 DllMain 函数入口地址。

EnumProcessModules 返回的第一个模块总是进程自身。

六、模块的关联信息

1. 版本信息

通过查看模块的版本信息，这些对于了解模块的来源和用途十分有用，在编写诸如程序安装、进程管理、病毒检测等软件时，通常需要先检测和判断版本信息。

当在 Windows 资源管理器里右击一个文件时，会弹出一个标准的文件属性对话框，可以看到文件的多个属性项描述，其实现代码如下。

```
SHELLEXECUTEINFO sei;
ZeroMemory(& sei, sizeof(sei));
sei.cbSize = sizeof(sei);
sei.lpFile = szFileName;
sei.lpVerb = _T("properties");
sei.fMask = SEE_MASK_INVOKEIDLIST;
ShellExecuteEx(& sei);
```

注意 SHELLEXECUTEINFO 结构的成员变量 lpVerb 的赋值，必须指定为“properties”。

为获取模块版本信息的更多细节，本文使用了 version.dll 的导出函数 VerQueryValue，它能够查询更多的文件版本信息。

在调用 VerQueryValue 之前，要先调用 GetFileVersionInfoSize 返回目标文件版本信息块的大小，如果存在版本信息块，则返回的值不为 0，就可以据此分配缓存区。然后，再调用 GetFileVersionInfo 将目标文件版本信息块复制到缓存区，之后就可以调用 VerQueryValue 函数从缓存区检索操作系统信息，以及描述文件版本信息分类项目描述字串。下面的示例代码中，szFilename 为包含路径的目标模块名。

```
char pBuf[MAX_PATH] ={0}, char szLanguageName
[MAX_PATH] ={0};
long *pValue = NULL, LangCodePage = 0;
DWORD m_dwHandle = 0, m_cbUser = 0;
LPVOID m_lpBuffer = NULL, m_lpData = NULL;
HANDLE hMemBuffer, hMemBufferData;
UINT    m_uiDataSize = MAX_PATH;
VS_FIXEDFILEINFO vsinfo;
m_cbUser = GetFileVersionInfoSize(szFilename, &
m_dwHandle);
if(m_cbUser = = 0)
    return;
else
{
    hMemBuffer = GlobalAlloc(GMEM_MOVEABLE,
 m_cbUser);
    m_lpBuffer = GlobalLock(hMemBuffer);
    hMemBufferData = GlobalAlloc(GMEM_MOVEABLE,
m_uiDataSize);
    m_lpData = GlobalLock(hMemBufferData);
    if (hMemBuffer)
    {
// 获取版本信息块
        GetFileVersionInfo(szFilename, 0, m_cbUser,
m_lpBuffer);
// 从版本信息块中获取操作系统信息
        VerQueryValue(m_lpBuffer, TEXT("\\"), &
m_lpData, & m_uiDataSize);
        MoveMemory((void *) & vsinfo, m_lpData,
m_uiDataSize);
//根据 vsinfo.dwFileType 判断模块类型
//根据 vsinfo.dwFileOS 判断运行平台
```

```
// 获取系统语言和页代码
        VerQueryValue(m_lpBuffer, TEXT("\\ VarFileInfo\\Translation"),
                        & m_lpData, & m_uiDataSize);
        pValue = (long *)m_lpData;
        LangCodePage =LOWORD(pValue[0]) < <16 | HIWORD(pValue[0]);
// 获取系统语言名称
        VerLanguageName(LOWORD(pValue[0]), szLanguageName, MAX_PATH);
// 格式化文件版本信息"描述"项检索请求标记
          sprintf(pBuf, "%s%08x%s%s", "\\StringFileInfo\\",
LangCodePage, "\\", "FileDescription" );
        VerQueryValue(m_lpBuffer, pBuf, & m_lpData, & m_uiDataSize);
//记录(char *)m_lpData 存放的文件版本信息"描述"项字
//符串;将 sprintf 语句内最后一个参数分别替换为"Compa
//nyName"、" LegalCopyright"、"OriginalFilename"、"Pro-
//ductName"、"ProductVersion"、"Comments"、"LegalT-
//rademarks"、"PrivateBuild"、"SpecialBuild"、"InternalN-
//ame"、"FileVersion",则得到版本信息各分类项目的描
//述字串
    }
    GlobalUnlock(hMemBufferData);
    GlobalFree(hMemBufferData);
    GlobalUnlock(hMemBuffer);
    GlobalFree(hMemBuffer);
}
```

上面用到的 VerQueryValue 原型定义如下：

```
BOOL VerQueryValue(const LPVOID pBlock, LPTSTR lpSubBlock,
LPVOID *lplpBuffer, PUINT puLen);
```

参数 pBlock 为指向存放文件版本信息块缓存区的指针，lpSubBlock 为要检索的文件版本信息分类项目描述字串检索请求标志，分以下几种：

"\" - 检索文件类型和操作系统信息的标志。需要将返回的信息块复制到 VS_FIXEDFILEINFO 结构中。VS_FIXEDFILEINFO 结构定义如下：

```
typedef struct VS_FIXEDFILEINFO {
    DWORD dwSignature;
    DWORD dwStrucVersion;
    DWORD dwFileVersionMS;
    DWORD dwFileVersionLS;
    DWORD dwProductVersionMS;
    DWORD dwProductVersionLS;
    DWORD dwFileFlagsMask;
    DWORD dwFileFlags;
    DWORD dwFileOS;
    DWORD dwFileType;
    DWORD dwFileSubtype;
    DWORD dwFileDateMS;
    DWORD dwFileDateLS;
} VS_FIXEDFILEINFO;
VS_FIXEDFILEINFO vsinfo, *pvs;
```

"\VarFileInfo\Translation" - 检索文件所用语言代码及名称的标志。

"\StringFileInfo\lang - codepage\string - name" - 检索版本信息分类项目描述字符串的标志。其中，lang - codepage 为语言和页代码，string - name 为下列描述文件版本信息分类项目的字符串常量之一（注意区分大小写）：

CompanyName、FileDescription、LegalCopyright、OriginalFilename、ProductName、ProductVersion、Comments、LegalTrademarks、PrivateBuild、SpecialBuild、InternalName、FileVersion

参数 lplpBuffer 为指向要获取的文件版本信息分类项目缓存区的指针，参数 puLen 为文件版本信息分类项目缓存区的大小。

在检索操作系统及文件版本之前的信息，必须先锁定预分配的全局内存，检索完后解锁。

2. 宿主进程

许多用户都曾遇到过这样的情况，杀毒软件提示发现某个病毒文件，却提示文件正在使用无法删除。Unlocker 是个很好工具软件，它可以找到调用病毒文件的宿主进程，断开病毒文件与宿主进程的关联。笔者借鉴 Unlocker 的方法，采用二重循环的方式，外层循环逐个枚举进程，内层循环逐个枚举进程引用的模块列表。若在模块列表中发现给定的带路径模块名，则当前枚举到的那个进程就是给定模块的一个宿主进程。

七、进程和模块管理

1. 设置进程优先级和运行权限

通过调用 SetPriorityClass 函数，其第二个参数可设为 IDLE_PRIORITY_CLASS（低）、NORMAL_PRIORITY_CLASS（标准）、HIGH_PRIORITY_CLASS（高）和 REALTIME_PRIORITY_CLASS（实时）四个级别，而不是 MSDN 所述的十三个级别。

运行本文实例程序需要具有管理员权限，否则部分功能受到限制。下面语句设置实例进程的权限为"调试"级别：

```
SetTokenPrivilege(::GetCurrentProcess(), "SeDebugPrivilege");
```

2. 结束进程

结束进程很简单，先打开进程，获得退出代码，最后调用 TerminateProcess 强制进程退出。

```
HANDLE handle;
DWORD ExitCode;
handle = OpenProcess (PROCESS_TERMINATE, TRUE,
```

```
ProcessID);
GetExitCodeProcess(handle, & ExitCode);
TerminateProcess(handle, ExitCode) ;
```

3. 将模块注入到远程进程空间内

将模块(DLL)注入到远程进程空间内，能够实现捆绑而又十分隐蔽，这是许多木马病毒惯用的手法，但许多有名的杀毒软件也采用了此法，如图 1 下部红色标注的 apihookdll. dll，即为木马客星注入 Explorer. exe 进程空间内的模块。

下面是注入模块的核心代码。在此特别提醒读者朋友，切勿将其用于非法目的。

```
   WCHAR pszLibFilename[MAX_PATH] ={0};
   DWORD dwID;
//szDLLFilenam 为模块名,dwRemoteProcessId 为被注入
//进程的 ID
MultiByteToWideChar(CP_ACP,
MB_ERR_INVALID_CHARS, szDLLFilename,
       strlen(szDLLFilename), pszLibFilename,
MAX_PATH);
   HANDLE hRemoteProcess = OpenProcess
(PROCESS_CREATE_THREAD |
       PROCESS_VM_OPERATION |
PROCESS_VM_WRITE,
       FALSE, dwRemoteProcessId );
//允许创建远程线程和远程虚内存写操作
   int cbSize = (lstrlenW(pszLibFilename) +1) * sizeof
(WCHAR);
   PWSTR pszLibFileRemote = (PWSTR) VirtualAllocEx(
hRemoteProcess,
NULL, cbSize, MEM_COMMIT, PAGE_READWRITE);
//在远程进程空间为模块名分配空间
   int iReturnCode = WriteProcessMemory
(hRemoteProcess,
       pszLibFileRemote, (PVOID) pszLibFilename, cb-
Size, NULL);
//将模块名复制到远程进程空间
   PTHREAD_START_ROUTINE pfnStartAddr =
(PTHREAD_START_ROUTINE)
       GetProcAddress(GetModuleHandle(TEXT( " Ker-
nel32")), "LoadLibraryW");
   HANDLE hRemoteThread = CreateRemoteThread( hRe-
moteProcess, NULL, 0,
       pfnStartAddr, pszLibFileRemote, 0, NULL);
//创建远程线程,调用注入模块
   WaitForSingleObject(hRemoteThread, INFINITE);
//清理现场
   if (pszLibFileRemote ! = NULL)
       :: VirtualFreeEx(hRemoteProcess, pszLibFileRe-
mote, 0, MEM_RELEASE);
   if (hRemoteThread ! = NULL)
   {
       ::GetExitCodeThread(hRemoteThread, & dwID);
       ::TerminateThread(hRemoteThread, dwID);
       ::CloseHandle(hRemoteThread);
   }
   if (hRemoteProcess ! = NULL)
   {
       ::GetExitCodeProcess(hRemoteProcess, & dwID);
       ::TerminateProcess(hRemoteProcess, dwID);
       ::CloseHandle(hRemoteProcess);
   }
```

实例程序可向单个或多个远程进程空间内注入模块，实际上是将模块名和模块代码段的起始地址写到远程进程空间内，而模块代码段是各个远程进程共享的。

在采用上述方法时，如果 DLL 模块本身不稳固，在向系统核心进程 winlogon. exe 和 csrss. exe 注入的过程中（smss. exe 进程拒绝注入），通常会因导致系统重启，这就是为什么有些病毒入侵后，会反复导致系统重启的主要原因。考虑到这一点，实例程序在注入模块时，特意跳过 winlogon. exe 和 csrss. exe 进程。

有时会出现无法向目标进程空间注入模块的情形，一般是因为目标进程受到了保护，如木马克星就会采用拦截 API 的方式、来保护 Explorer. exe 进程不会被注入非法模块。

4. 从远程进程空间内卸载模块

知道了将模块注入远程地址空间的过程，也就不难将模块从远程进程空间内卸载掉，有名的 IceSword 和 Unlocker 都具有此功能。下面是卸载模块的核心代码。

```
//ProcessID 为远程进程 ID, hRemoetMoudle 为模块代码
//段在目标进程空间的起始地址
HANDLE hThread;
HANDLE hProcess =OpenProcess(
PROCESS_CREATE_THREAD|PROCESS_VM_OPERATION|
PROCESS_VM_WRITE, FALSE, ProcessID);
if(!hProcess) return;
hThread =CreateRemoteThread(hProcess, NULL, 0,
    (LPTHREAD_START_ROUTINE)FreeLibrary, (LPVOID)
hRemoetMoudle, 0, NULL);
//创建卸载操作的远程线程
WaitForSingleObject(hThread, INFINITE);
VirtualFreeEx(hProcess, & hRemoetMoudle, 0,
MEM_RELEASE);
//卸载模块
::CloseHandle(& hRemoetMoudle);
//清理现场
if (hThread ! = NULL)
{
   :: GetExitCodeThread(hThread, (DWORD * ) & Pro-
cessID);
   ::TerminateThread(hThread, ProcessID);
```

```
        ::CloseHandle(hThread);
    }
    if (hProcess != NULL)
    {
        ::GetExitCodeProcess(hProcess, (DWORD *)&
ProcessID);
        ::TerminateProcess(hProcess, ProcessID);
        ::CloseHandle(hProcess);
    }
```

与注入过程类似，在卸载系统核心进程 winlogon.exe 和 scrss.exe 空间内的模块时，有可能会导致系统重启；在卸载 Exploer.exe 进程空间内的模块时，通常会导致 Exploer.exe 关闭后重新运行。

实例程序可从单个或多个远程进程空间内卸载模块。有时会出现无法卸载的情形，一般是因为目标模块受到了保护，如木马克星就会采用拦截 API 的方式，来保护自身注入 Explorer.exe 进程空间的模块不会被卸载。

5. 注册/注销模块

Windows 提供的 Regsvr32.exe 用来注册/注销模块，它是通过调用模块内部的 DllRegisterServer/DllUnregisterServer 接口来实现的，实现的关键代码如下：

```
// pszDllName 为要注册/注销的模块名,regFunction 为函
//数名称 DllRegisterServer 或 DllUnregisterServer
    HINSTANCE hLib = LoadLibrary(pszDllName);
    if (hLib < (HINSTANCE)HINSTANCE_ERROR)
        return;
    FARPROC lpDllEntryPoint = GetProcAddress(hLib, reg-
Function);
    if(lpDllEntryPoint != NULL)
        (*lpDllEntryPoint)();
    FreeLibrary(hLib);
```

八、枚举驱动及服务

WindowsXP/2000 的服务分两类：一类是 Win32 应用服务(一般用无窗口的 .EXE 类程序启动)，另一类是内核服务(一般用 .SYS 类程序驱动，也就是通常所说的驱动程序)。

可能是考虑到安全性，WindowsXP/2000 自带的服务管理器只提供 Win32 应用服务管理功能，这让不少采用内核服务(底层驱动)方式的木马病毒钻了空子，用户采用常规手段根本无法清除这些病毒。

本文实例程序不仅能管理 Win32 应用服务，也能管理内核服务，比 WindowsXP/2000 自带的服务管理器功能强大，可以有效地查杀采用底层驱动的木马病毒。

1. 使用 PSAPI 枚举驱动设备

PSAPI 提供 EnumDeviceDrivers 用于枚举系统内核驱动设备，与枚举进程和模块的过程类似，但获取得信息较少，下面的是实现的代码：

```
PVOID aDrivers[1024];
DWORD cbNeeded;
char szBaseName[MAX_PATH] = {0};
char szDriverFileName[MAX_PATH] = {0};
unsigned i;
if ( !EnumDeviceDrivers( aDrivers, sizeof(aDrivers), & cb-
Needed))
    return;
DWORD cDrivers = cbNeeded / sizeof(aDrivers[0]);
for (i = 0; i < cDrivers; i++)
{
if (GetDeviceDriverBaseName( aDrivers[i],
szBaseName, sizeof(szBaseName)))
    {
        GetDeviceDriverFileName( aDrivers[i],
szDriverFileName, sizeof(szDriverFileName));
.....
    }
}
```

2. 使用注册表枚举服务

WindowsXP/2000 的服务设置存储在注册表中，通过读取注册表中的 HKEY_LOCAL_MACHINESYSTEM\SYSTEM\CurrentControlSet\Services 分支即可获得所有服务的以下各项参数：

DisplayName、Description、ImagePath、ObjectName、DependOnGroup、Start、Group、DependOnService、Type。

以上各参数字面意思不难理解，在此只对 Start 和 Type 参数含义作些解释。Start 指服务的启动方式，有自动、手工、禁用和系统 I/O(仅限于内核服务) 四种。Type 指服务的类型，有独立进程服务和共享进程服务两种。

对于采用类似\%SystemRoot%\system32\svchost.exe -k netsvcs 方式通过 svchost.exe 启动的 Win32 应用服务，还要在注册表中查找对应的服务模块，例如，要查找 AudioSrv 的服务模块，就需要读注册表中的以下分支：

HKEY_LOCAL_MACHINESYSTEMSYSTEM\CurrentControlSet\Services\AudioSrv\Parameters

从注册表中获得的内核服务项目和数量，与利用 PSAPI 的 EnumDeviceDrivers 函数获得的内核服务项目和数量会略有差异，几个最核心的内核服务在上述注册表分支中是找不到的。

九、服务管理

服务管理内容包括服务的安装、启动、停止、删除、启动方式设置等，在此只介绍后四项功能的实现方法。

1. 启动服务

Windows 提供了服务管理器接口，具有启动服务、停止服务、删除服务、设置服务启动方式的功能。下面的代码演示了启动 AudioSrv 服务的过程：先调用 OpenSCManager 函数打开服务管理器，再调用 OpenService 函数打开指定服务的句柄，最后

调用 StartService 函数启动 AudioSrv 服务。启动服务后，要及时释放服务管理器的句柄。

OpenService 函数的第三个参数可设为 SERVICE_START 或 SERVICE_STOP，分别指定启动服务或停止服务的操作。

```
SC_HANDLE hSCManager = OpenSCManager(NULL,
SERVICES_ACTIVE_DATABASE,
SC_MANAGER_ALL_ACCESS);
if(hSCManager ! = NULL )
{
    SC_HANDLE hService = OpenService( hSCManag-
er, "AudioSrv", SERVICE_START);
    if(hService)
        if(!StartService(hService, 0, NULL))
            ……
    CloseServiceHandle(hService);
    CloseServiceHandle(hSCManager);
}
```

2. 停止服务

与启动服务的代码类似，不过要调用 ControlService 函数来停止服务。下面的代码演示了停止 AudioSrv 服务的过程。

```
SC_HANDLE hSCManager = OpenSCManager(NULL,
SERVICES_ACTIVE_DATABASE,
SC_MANAGER_ALL_ACCESS);
if(hSCManager ! = NULL )
{
    SERVICE_STATUS svrstatus;
    SC_HANDLE hService = OpenService( hSCManag-
er, "AudioSrv", SERVICE_STOP);
    if(hService)
        if(!ControlService(hService, SERVICE_
CONTROL_STOP, & svrstatus))
            ……
    CloseServiceHandle(hService);
    CloseServiceHandle(hSCManager);
}
```

3. 删除服务

也与启动服务的代码类似，直接调用 DeleteService 函数即可删除服务。下面的代码演示了删除 AudioSrv 服务的过程。

```
SC_HANDLE hSCManager = OpenSCManager(NULL,
SERVICES_ACTIVE_DATABASE,
SC_MANAGER_ALL_ACCESS);
if(hSCManager ! = NULL )
{
    SC_HANDLE hService = OpenService( hSCManag-
er, "AudioSrv", DELETE); if(hService)
    if(!DeleteService(hService))
……
    CloseServiceHandle(hService);
    CloseServiceHandle(hSCManager);
}
```

4. 设置服务启动方式

设置服务的启动方式需要修改注册表中服务的 Start 键值(DWORD 类型)，值 1、2、3、4 分别对应系统 I/O(仅限于内核服务)、自动、手工和禁用四种启动方式。下面代码演示了将 AudioSrv 服务设置为自动启动方式的过程。

```
HKEY hRootKey =HKEY_LOCAL_MACHINE;
HKEY hKey;
char szKeyName[MAX_PATH] ={"SYSTEM\\CurrentCon-
trolSet\\Services\\AudioSrv"};
BYTE * pVal =(BYTE *)& dwModel;
DWORD i = RegOpenKey(hRootKey, szKeyName, &
hKey);
if(i = = ERROR_SUCCESS)
    RegSetValueEx(hKey, " Start", 1, REG_DWORD,
pVal, sizeof(DWORD));
RegCloseKey(hKey);"
```

十、端口扫描

实例程序的端口扫描功能与 Windows 自带的 Netstat 功能类似，只是存在一个问题，那就是在 WindowsXP 下扫描正常，但在 Windows2000 下扫描却无效，笔者至今也没有找到有效的解决办法。

端口扫描功能的实现，最主要的是利用了 iphlpapi.dll 的两个导出函数 AllocateAndGetTcpExTableFromStack 和 AllocateAndGetUdpExTableFromStack，两者分别用于获取 TCP 和 UDP 端口的连接信息。

十一、窗口扫描

相信许多读者都遇到过这样的情况，IE 浏览器的主页设置按钮被禁用无法使用，这一般是恶意网站所为。

本文实例程序可以恢复 IE 浏览器的主页设置按钮，例如，恢复设置空白页的过程如下：

运行 IE 浏览器→点击“工具”→“Internet 选项”→运行实例程序→点击“窗口扫描”→在窗口列表中依次展开“Internet 选项”→“常规”→选定“使用空白页”→点击“恢复”即可。

1. 使用递归法扫描窗口

图 6 所示的窗口列表通过递归法枚举扫描获得，它与文件目录类似，呈树状结构。下面列出实例程序中递归法扫描窗口函数 BTSearchWindow 的代码，参数 hWnd 为扫描起始的一个根节点即父窗口，szClass 为窗口的类名，szName 为窗口的标题。

```
BOOL CSearchWindow: : BTSearchWindow(HWND hWnd,
CString, CString)
{
HWND hWndChild, hWndNext;
```

```
    if( !hWnd )
        return FALSE;
    this -> InsertTreeItem(hWnd);
//插入父窗口
    if(IsFound(hWnd, szClass, szName))
//判断是否指定窗口
    {
        hChild =hWnd;
        return TRUE;
    }
    hWndChild =::GetWindow(hWnd, GW_CHILD);
//扫描子窗口
    branch =TRUE;
    BTSearchWindow(hWndChild, szClass, szName);
    hWndNext =::GetWindow(hWnd, GW_HWNDNEXT);
//扫描兄弟窗口
    branch =FALSE;
    hTreeItem =m_WndTree.GetParentItem(hTreeItem);
    BTSearchWindow(hWndNext, szClass, szName);
    return TRUE;
}
```

在调用 BTSearchWindow 时，hWnd、szClass 和 szName 可选。一般宜调用 GetDesktopWindow 获得桌面窗口的句柄作为扫描起始根节点。如指定 szClass 和 szName，则查找到匹配窗口后停止。

2. 使用鼠标捕捉窗口

SPY + + 具有鼠标捕捉窗口功能，其原理很简单，先进行屏幕坐标转换，然后调用 WindowFromPoint 函数，跟踪鼠标光标并判断其在哪个窗口内。

```
    ClientToScreen(& point);
    ::GetCursorPos(& point);
    CWnd * pWnd =WindowFromPoint(point);
    if(pWnd)
    {
        if(GetWindowThreadProcessId(pWnd ->
m_hWnd, NULL) ! =
            GetWindowThreadProcessId(this ->
m_hWnd, NULL))
//判断是否捕捉程序自身的窗口
}
```

实例程序可列出注册表中 12 个分支的启动项设置。在执行删除前，勾选“Invers”可备份删除的键值，需要时可用添加命令恢复。添加操作只支持 REG_SZ、REG_MULTI_SZ 和 REG_DWORD 三种类型键值设置。

十二、用实例程序查杀木马病毒

实例程序运行在 WindowsXP/2000 环境下，用户需具有管理员权限。实例程序界面为对话框，除使用按钮外，更多功能需在列表框上按右键使用快捷菜单调出。

查杀木马病毒时，先观察有无可疑进程，再查看进程引用的模块。在查看模块时，要特别要留心红色标记的模块，凡是来历不明的大都是注入进程内部的病毒模块。当出现进程无法终止或模块无法卸载的现象时，有可能是对应的服务处于活动状态，可停止服务后再尝试终止进程或卸载模块，必要时在安全模式下运行。查看启动项设置、扫描端口和窗口，有助于发现一些隐藏的进程或注入的模块。

在使用文件删除功能时，要特别小心，文件一旦被删除便无法恢复，除非使用 Recover4All 之类的专用工具软件。

（收稿日期：2007 年 12 月 27 日）

江民成立北京 2008 奥运网络安全应急响应小组

近日，国内领先的计算机反病毒软件厂商江民科技宣布，正式成立 2008 北京奥运会网络安全应急响应小组，响应奥组委以及国家网络安全主管部门的号召，为 2008 北京奥运会成功举办在网络安全方面保驾护航。

江民北京 2008 奥运会网络安全应急响应小组由奥组委特聘网络与信息安全专家王江民担任总指挥，江民科技研发部 30 余名技术骨干和反病毒专家担任组员，应急小组将利用江民科技近 20 年在计算机反病毒及网络安全方面积累的技术优势，为奥运会提供信息安全保障服务，特别是为直接与奥运会相关的网站以及各种信息资源提供信息安全保卫工作。

江民奥运会网络安全应急小组主要工作包括日常事务以及应急服务。奥运期间，应急小组将每日汇总上报江民科技反病毒预警监测平台获得的各种有价值线索和信息；对奥运相关网站进行网络安全保卫工作。对在奥运期间出现的网络安全或者病毒事件等建立“事故登记制度”，分析事故发生原因并积极提供事故解决方案。应急小组特别注重的是奥运期间的网络安全应急响应服务，要求小组成员 24 小时待命，第一时间到奥运现场处理网络和信息安全事件，包括病毒事件、网页挂马事件等。

江民科技董事长、北京奥运会特聘网络与信息安全专家王江民认为，作为国内网络安全和计算机反病毒重要技术力量，保卫 2008 北京奥运会网络安全江民科技责无旁贷。江民科技拥有多次保卫国家重大活动和体育赛事的网络安全保卫经验，曾多次为全国“两会”、“全国城市运动会”等提供技术支持，江民科技完全有能力为保卫北京奥运会网络安全做出自己的贡献。相信在奥组委、国家信息与网络安全主管部门的领导下，在全国人民的共同努力下，2008 北京奥运会必将成为全球奥运史上成功的一次体育盛事。