本文作者是楚狂人，代码来源于开源工程tdifw与DDK的例子，
有问题欢迎与我联系讨论。

mail: mfc_tan_wen@163.com
QQ: 16191935
msn: walled_river@hotmail.com

Windows TDI过滤驱动开发

目 录

(0) TDI概要

最早出现的网络驱动应该是网卡驱动，这是Windows的理所当然的需求，为了进一步分割应用程序的网络数据传输与下层协议直到下层硬件的关系，又出现了协议驱动，后来微软和硬件商联合制定了NDIS标准，作为从硬件到协议的内核驱动程序的调用接口标准，而协议驱动与应用层的API之间，则出现了TDI接口。

最近国内安全软件的开发兴起，网络驱动的开发在其中有不少的应用，如果我们学习TDI接口的话，可能有以下一些目的：

自己要开发协议驱动，向上提供TDI接口，这种可能性存在，但不广泛。
我们想自己调用TDI接口，来进行网络数据传输，意义不大。
我们对TDI进行协议层过滤，开发防火墙或类似安全监控软件，这种应用还是比较多的。

(1) 准备工作

为了开始研究网络驱动开发，建议你做如下的准备：

安装VC.net、VC6.0或者更老的版本也可以编译驱动程序，但是本文提供的工程都是VC.net工程，如果要看范例代码你可能必须安装VC.net。

然后需要安装DDK，另外TDI过滤在DDK中并没有现成的例子（但是提供TDI接口的NDIS网络驱动应该是有的）。直接研究TDI接口是件痛苦的事情，这套接口秉承了微软公司各类接口的特点：臃肿而且复杂。幸运的是有用TDI过滤写成的本地防火墙的开源代码可以研究。

你必须自己学习DDK的编译方法，和一般的应用程序稍有不同。

tdifw是一个基于TDI过滤的开源本地防火墙，有兴趣的读者可以在这里下载代码：

http://tdifw.sourceforge.net/

这个防火墙有很多额外的代码，我把它进行了精简和修改，附在本文的范例代码中提供下载，工作空间为tdi_fw.sln。可以打开直接编译。

此外作为驱动开发者，你还必须准备一些工具：

驱动加载工具，编译好一个驱动你可以动态加载它，(NDIS网络驱动稍微有些不同，以后再详细介绍)。这样你需要一个工具加载它，我推荐installer.exe。遗憾的是网络上有很多软件叫做installer，我只能说这个installer.exe的图标是一个黄色安全帽。

输出查看工具，推荐DbgView.exe，你可以在以下地址下载：

http://www.sysinternals.com/Utilities/DebugView.html

给驱动程序配上界面绝对不是一件简单的工作，所以需要一个输出查看工具，这样至少在程序中可以print一些东西，用这个工具可以看到，在没法安装调试工具的情况下，print能起巨大的作用。我有softice老安装失败最后全部依靠print解决bug的经历。

调试工具，推荐windbg和softice。

windbg可以免费下载，地址为http://www.microsoft.com/whdc/devtools/debugging/default.mspx

windbg和softice的使用都需要专门的章节介绍，本文略过，所以有兴趣的读者请在驱动开发网上寻找相关的介绍。

没有安装并不会使用调试工具的情况下，你也可以继续阅读并试用本文附带的代码。

(2) TDI设备与驱动入手

TDI是一组接口的名字，协议驱动本身都是NDIS协议驱动。但是其上提供TDI接口，Windows的上层网络组件调用这些接口来使用协议驱动。这些接口是可以被过滤的，下面以一个TDI过滤驱动为例。

TDI过滤相比NDIS过滤的一大好处是TDI离应用层比较近，容易得到应用程序的相关信息。比如一个连接建立，你可以获取打开这个连接的PID，也就得知了打开这个连接的应用程序。而不足之处则是安全性，若我想写一个木马来绕过TDI接口，我可以不调用一般网络API来避免调用TDI接口，不过NDIS过滤虽然相对保险，绝对的安全依然是不可能的。

提供TDI接口的Windows协议驱动将在Windows内核中生成所谓的TDI设备。设备是有路径和名字的，比较著名的设备有以下几个: "\\Device\\Tcp"，对应TCP协议。

"\\Device\\Udp"，对应UDP协议。

"\\Device\\RawIp"，对应原始IP包。

如果你安装了TCPIP之外的协议，应该还有更多，既然我们要过滤TDI接口，那么首先就是生成我们自己的设备，来绑定这些设备。

这就是过滤的原理。如果我们用自己的设备绑定了原有的设备，那么Windows将把本来给原设备的请求给我们的新设备。从而可以过滤他们。

驱动开发的老手对生成设备和绑定设备再熟悉不过，但是没有开发过驱动的朋友可能有些困惑，我的建议是没有必要一开始就试图了解太多概念，只要了解代码如何写就可以了。

我们建立.c文件.包含如下文件:

```
#include <ntddk.h>
#include <tdikrnl.h>
```

这两个文件都是ddk的头文件，这就是为何你得安装DDK，然后我们开始写一个函数DriverEntry,

这个函数相当与一般Windows应用程序之WinMain，原型如下：

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
        IN PUNICODE_STRING theRegistryPath)
```

IN是一个无任何内容的宏，代表后面的参数为输入参数，有些用来返回结果的指针前面会带OUT，仅仅起提示作用，最简单的写法为：

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
        IN PUNICODE_STRING theRegistryPath)
{
    return STATUS_UNSUCCESSFUL;
}
```

这个驱动已经可以编译了，只是无法加载，总是返回STATUS_UNSUCCESSFUL错误。

(3) 绑定设备

下面我们在这个过程中，完成绑定主要的TDI设备的任务，下面这个函数来自tdi_fw工程，你可以在tdi_fw.c中找到这些代码。

```
// 这个函数生成一个设备来绑定一个已知名字的设备。
NTSTATUS
c_n_a_device(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT *fltobj, PDEVICE_OBJECT *oldobj,
        wchar_t *devname)
{
    NTSTATUS status;
    UNICODE_STRING str;

    // 生成自己的新设备
    status = IoCreateDevice(DriverObject,
            0,
            NULL,
            FILE_DEVICE_UNKNOWN,
            0,
            TRUE,
            fltobj);
    if (status != STATUS_SUCCESS) {
        KdPrint(("[tdi_fw] c_n_a_device: IoCreateDevice(%S): 0x%x\n", devname, status));
        return status;
    }

    // 设置设备IO方式为直接IO
    (*fltobj)->Flags |= DO_DIRECT_IO;

    // 将要绑定的设备名初始化为一个Unicode字符串
    RtlInitUnicodeString(&str, devname);

    // 绑定这个设备
    status = IoAttachDevice(*fltobj, &str, oldobj);
    if (status != STATUS_SUCCESS) {
        KdPrint(("[tdi_fw] DriverEntry: IoAttachDevice(%S): 0x%x\n", devname, status));
        return status;
    }

    KdPrint(("[tdi_fw] DriverEntry: %S fileobj: 0x%x\n", devname, *fltobj));
    return STATUS_SUCCESS;
```

```
}
```

有这么一些要点要注意的：

1.常常使用Unicode字符串而非一般的字符串，Unicode字符串常常用RtlXxxUnicodeString系列的函数操作，你应该翻阅DDK帮助来熟悉它们。

2.用KdPrint来代替printf输出信息，这些信息可以被你的工具(比如DbgView.exe)看到。

3.IoCreateDevice()和IoAttachDevice()打开一个设备然后对它进行绑定，查阅帮助文档了解它们的用法吧。

(4) 简单的处理请求

如果一个驱动生成了设备，那么则必须设置处理Windows发给这些设备的请求的分发处理函数。作为过滤你可以有很多选择。比如直接调用真实的设备的处理过程或者自己处理。

在前面的DriverEntry中添加：

```
for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
    theDriverObject->MajorFunction[i] = DeviceDispatch;
```

这样一来，所有的请求都发到DeviceDispatch这个函数，这个函数成为唯一的分发函数。

为了让我们的程序能快速的继续开发下去，我们写一个直接调用真实设备的处理过程的DeviceDispatch：

```
NTSTATUS
DeviceDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP irp)
{
    PDEVICE_OBJECT old_devobj = get_original_devobj(devobj, NULL);
    if(old_devobj != NULL)
    {
        // 如果能找到原设备,则发到原设备.
        IoSkipCurrentIrpStackLocation(irp);
        status = IoCallDriver(old_devobj, irp);
    }
    else
    {
        // 如果不能找到,则返回失败.
        status = irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
        IoCompleteRequest (irp, IO_NO_INCREMENT);
    }
```

```
    return status;
}
```

另外一个疑问是如何获得原有真实设备，get_original_devobj的代码很简单：

```
PDEVICE_OBJECT
get_original_devobj(PDEVICE_OBJECT flt_devobj, int *proto)
{
    PDEVICE_OBJECT result;
    int ipproto;

    if (flt_devobj == g_tcpfltobj) {
        result = g_tcpoldobj;
        ipproto = IPPROTO_TCP;
    } else if (flt_devobj == g_udpfltobj) {
        result = g_udpoldobj;
        ipproto = IPPROTO_UDP;
    } else if (flt_devobj == g_ipfltobj) {
        result = g_ipoldobj;
        ipproto = IPPROTO_IP;
    } else {
        KdPrint(("[tdi_fw] get_original_devobj: Unknown DeviceObject 0x%x!\n",
        flt_devobj));
        ipproto = IPPROTO_IP;        // what else?
        result = NULL;
    }

    if (result != NULL && proto != NULL)
        *proto = ipproto;

    return result;
}
```

get_original_devobj之所以能这样做，是因为它已经事先把各个协议的设备以及我们生成的新设备的指针保存在全局变量里了，也就是上面的g_tcpfltobj， g_tcpoldobj， g_udpfltobj， g_udpoldobj 这些变量。简单比较一下就行。

(5) 基础过滤框架

那么我们可以写一个基础的过滤框架了：

```
#include <ntddk.h>
#include <tdikrnl.h>

// 保存设备指针的全局变量
PDEVICE_OBJECT
	g_tcpfltobj = NULL,		// \Device\Tcp
	g_udpfltobj = NULL,		// \Device\Udp
	g_ipfltobj = NULL,		// \Device\RawIp
	g_tcpoldobj = NULL,		// \Device\Tcp
	g_udpoldobj = NULL,		// \Device\Udp
	g_ipoldobj = NULL;		// \Device\RawIp

// 卸载函数.
VOID
OnUnload(IN PDRIVER_OBJECT DriverObject)
{
	// 我对已生成和绑定的设备进行解绑和删除
	if (g_tcpoldobj!= NULL) IoDetachDevice(g_tcpoldobj);
	if (g_tcpfltobj! != NULL) IoDeleteDevice(g_tcpfltobj);
	if (g_udpoldobj!= NULL) IoDetachDevice(g_udpoldobj);
	if (g_udpfltobj! != NULL) IoDeleteDevice(g_udpfltobj);
	if (g_ipoldobj!= NULL) IoDetachDevice(g_ipoldobj);
	if (g_ipfltobj! != NULL) IoDeleteDevice(g_ipfltobj);

}

// 驱动入口.
NTSTATUS
DriverEntry(IN PDRIVER_OBJECT theDriverObject,
		IN PUNICODE_STRING theRegistryPath)
{
	NTSTATUS status = STATUS_SUCCESS;
	int i;

	// 设置分发函数
	for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
		theDriverObject->MajorFunction[i] = DeviceDispatch;
	theDriverObject->DriverUnload = OnUnload;

	// 绑定几个TDI设备
	status = c_n_a_device(theDriverObject, &g_tcpfltobj, &g_tcpoldobj, L"\\Device\\Tcp");
```

```
        if (status != STATUS_SUCCESS) {
            KdPrint(("[tdi_fw] DriverEntry: c_n_a_device: 0x%x\n", status));
            goto done;
        }
        status = c_n_a_device(theDriverObject, &g_udpfltobj, &g_udpoldobj, L"\\Device\\Udp");
        if (status != STATUS_SUCCESS) {
            KdPrint(("[tdi_fw] DriverEntry: c_n_a_device: 0x%x\n", status));
            goto done;
        }
        status = c_n_a_device(theDriverObject, &g_ipfltobj, &g_ipoldobj, L"\\Device\\RawIp");
        if (status != STATUS_SUCCESS) {
            KdPrint(("[tdi_fw] DriverEntry: c_n_a_device: 0x%x\n", status));
            goto done;
        }
    done:
        // 如果失败了
        if (status != STATUS_SUCCESS) {
            OnUnload(theDriverObject);
        }
        return status;
    }
```

里面调用的所有函数上面都提及了，现在你可以编译这个驱动，并动态加载了，这就是我们第一个TDI过滤驱动的框架。

(6) 主要过滤的请求类型

我们已经完成了进行过滤的框架，下面的任务是了解该过滤什么，假设作为一个防火墙或者一个安全监控软件，我希望监控计算机内的软件对网络进行的访问，比如说连接请求，连接ip地址，端口，等等。现在必须进一步了解TDI接口的请求方式。

所有的请求都被封装为IRP并在DeviceDispatch(见前面我们所写的请求处理函数DeviceDispatch)中处理。IRP以主功能号和辅功能号决定它的类别，在DeviceDispatch中，获得主功能号码的代码入下：

```
    NTSTATUS DeviceDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP irp)
    {
        NTSTATUS status;
        PIO_STACK_LOCATION irps;
```

```
    // 获取当前irp栈空间
    irps = IoGetCurrentIrpStackLocation(irp);

    switch (irps->MajorFunction) {

    // 生成请求
    case IRP_MJ_CREATE:
        // ...
        break;

    // 设备控制请求
    case IRP_MJ_DEVICE_CONTROL:
        // ...
        break;

    // 内部设备控制请求
    case IRP_MJ_INTERNAL_DEVICE_CONTROL:
        // ...
        break;

    case IRP_MJ_CLOSE:
        // ...
        break;

    case IRP_MJ_CLEANUP:
        // ...
        break;

    default:

    }

    // ...
    return status;
}
```

次功能号则在irps->MiniorFunction中。根据不同的主功能号有不通的次功能号，不能给出通用的例子，应该逐一分析。同时，有时一些主功能下并没有次功能的区分，这样次功能号并不总是有意义。IRP_MJ_DEVICE_CONTROL和IRP_MJ_INTERNAL_DEVICE_CONTROL的情况，次功能号用得比较多。

对于老手而言只需要了解IRP_MJ_CREATE,IRP_MJ_DEVICE_CONTROL和IRP_MJ_INTERNAL_DEVICE_CONTROL是TDI接口中最重要的三种请求就可以了，新手可能需

要花费一些功夫了解IRP栈空间这样的概念。但是我认为不了解也可以继续阅读，你只需要知道这是获取请求类型的一个必须过程罢了。前面的DeviceDispatch函数中，把请求发送给真实设备的时候，也和栈空间有关，有兴趣的读者可以阅读DDK中关于"请求"IRP的相关介绍。

TDI接口的调用总是遵循着打开->设备控制->关闭的流程.我们将截取这些IRP中的信息,来进行我们的过滤.

(7) CREATE的过滤

对于CREATE的irp的过滤，一般有以下几个步骤：

1.得到当前进程：

一些防火墙喜欢显示出使用一个连接的进程.我们要得到生成连接的进程这就是机会.非常简单:

```
ULONG pid = (ULONG)PsGetCurrentProcessId();
```

在处理Create的irp的进程中调用即可。

2.获取EA数据，打开DDK查看打开设备的核心API ZwCreateFile的文档：

```
NTSTATUS  ZwCreateFile(
    OUT PHANDLE  FileHandle,
    IN ACCESS_MASK  DesiredAccess,
    IN POBJECT_ATTRIBUTES  ObjectAttributes,
    OUT PIO_STATUS_BLOCK  IoStatusBlock,
    IN PLARGE_INTEGER  AllocationSize  OPTIONAL,
    IN ULONG  FileAttributes,
    IN ULONG  ShareAccess,
    IN ULONG  CreateDisposition,
    IN ULONG  CreateOptions,
    IN PVOID  EaBuffer  OPTIONAL,
    IN ULONG  EaLength);
```

这里的EaBuffer我们在平时的开发中极少使用，但是TDI操作却几乎全靠Ea数据来传递信息，微软这样做的理由让人无法得知，现在我们获得Ea数据：

```
FILE_FULL_EA_INFORMATION *ea = (FILE_FULL_EA_INFORMATION *)irp->AssociatedIrp.SystemBuffer;
```

具体的tdi操作请求指令的名称就保存在ea->EaName中。而名字的内容，在CREATE中有以下两个预先定义的宏：

TdiTransportAddress : 表明目前Create的是一个传输层地址。
TdiConnectionContext : 表明目前生成一个连接终端。

而且两个字符串的长度应该分别为：
TDI_TRANSPORT_ADDRESS_LENGTH和TDI_CONNECTION_CONTEXT_LENGTH。
下面是tdi_fw中进行过滤的方法，假设我们在DeviceDispatch中调用tdi_create来进行Create IRP的过滤。

```
int    tdi_create(PIRP irp, PIO_STACK_LOCATION irps, struct completion *completion)
{
        NTSTATUS status;
        FILE_FULL_EA_INFORMATION *ea = (FILE_FULL_EA_INFORMATION *)irp-
>AssociatedIrp.SystemBuffer;
        ULONG pid = (ULONG)PsGetCurrentProcessId();

        if (ea != NULL) {
            if (ea->EaNameLength == TDI_TRANSPORT_ADDRESS_LENGTH &&
               memcmp(ea->EaName, TdiTransportAddress, TDI_TRANSPORT_ADDRESS_LENGTH)
== 0) {
                  // ... 在这里捕获传输层地址生成
            }
        }
        else if (ea->EaNameLength == TDI_CONNECTION_CONTEXT_LENGTH &&
            memcmp(ea->EaName, TdiConnectionContext,
TDI_CONNECTION_CONTEXT_LENGTH) == 0) {

            // ... 在这里捕获连接终端的生成
        }
        // ...
}
```

之后分别对这两种情况进行进一步的信息解析。

(8) 准备解析ip地址与端口

我们的安全软件要监控的是IP地址与端口。这可以通过给CREATE IRP生成传输地址的之后，向该FILE_OBJECT发送一个QUERY IRP来获得。

这件事的困难在于，只有当这个IRP被发到下层，完成之后，才能发送Query请求，来询问IP地址与端口，tdifw的做法是，将一些信息保存到指针completion所指的区域内。不要为不能理解的概念而烦恼，我们看下面的代码：

```
// 生成的处理
int    tdi_create(PIRP irp, PIO_STACK_LOCATION irps, struct completion *completion)
{
    // ...
    if (ea != NULL) {
        if (ea->EaNameLength == TDI_TRANSPORT_ADDRESS_LENGTH &&
            memcmp(ea->EaName, TdiTransportAddress,
TDI_TRANSPORT_ADDRESS_LENGTH) == 0) {

            // 传输层地址生成的处理，我们必须询问被打开的FILE_OBJECT来获得IP地址和端口.询问需要一个IRP。我们调用TdiBuildInternalDeviceControlIrp来分配一个空的IRP。因为完成函数往往不在PASSIVE_LEVEL，所以我们在这里生成它。
            query_irp = TdiBuildInternalDeviceControlIrp(TDI_QUERY_INFORMATION,
                devobj, irps->FileObject, NULL, NULL);
            if (query_irp == NULL) {
                KdPrint(("[tdi_fw] tdi_create: TdiBuildInternalDeviceControlIrp\n"));
                return FILTER_DENY;
            }

            // 指定一个完成函数，这样如果CREATE IRP一完成，就会调用这个函数，我们可以在其中询问IP地址和端口。
            completion->routine = tdi_create_addrobj_complete;

            // 同时把我们已分配的IRP记录下来，在之后使用.
            completion->context = query_irp;

        }
    // ...
}
```

这里涉及到一个中断级别的问题，已知的是IRP的完成函数往往在DISPATCH_LEVEL被调用，这个中断级上不能调用TdiBuildInternalDeviceControlIrp。而分发函数一般都在PASSIVE_LEVEL，调用TdiBuildInternalDeviceControlIrp是合适的，我认为这些概念不理解也并不影响阅读。

回到前面的DeviceDispatch函数，里面是这样调用tdi_create的：

```
// 分发函数
NTSTATUS
DeviceDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP irp)
{

    ... ...
    switch (irps->MajorFunction) {

        case IRP_MJ_CREATE:          /* create fileobject */

            result = tdi_create(irp, irps, &completion);

            status = tdi_dispatch_complete(DeviceObject, irp, result,
                completion.routine, completion.context);

        break;
        ...
    }
    ...
}
```

显然要查询地地址时，IRP将在tdi_dispatch_complete中被完成。完成后tdi_create_addrobj_complete被调用来询问IP地址，我们在下一节里继续。

(9)获取生成的IP地址和端口

上回说到，我们对一个CREAT IRP进行过滤。当其中含有的TDI功能命令为TdiTransportAddress，我们得知一个地址正在生成（实际上是一个连接的本地地址）。只有等这个IRP下发完成之后，我们才可以通过对这个被打开的FILE_OBJECT发送QUERY IRP，得到生成的地址。

那么我们将用到IRP的完成函数。完成函数将在这个IRP被下层完成后调用。你可以如下的设置一个完成函数:

```
IoSetCompletionRoutine(irp, my_complete, context, TRUE, TRUE, TRUE);
```

这里的my_complete就是我们的完成函数。考虑到上回，我们将在地址生成之后进行查询，那么你可以把这个函数指定为tdi_create_addrobj_complete。context是一个上下文。当my_complete被调用的时候，你需要很多参数，比如以前你分配的irp指针，你需要保存的其他信息，都可以通过

这个指针传入。

我们曾经用TdiBuildInternalDeviceControlIrp分配了一个空的irp。那么我们现在应该设置它。指定我们的查询动作。

```
TdiBuildQueryInformation(query_irp, devobj, irps->FileObject,
    tdi_create_addrobj_complete2, context,
    TDI_QUERY_ADDRESS_INFO, mdl);
```

这里query_irp是上面我们分配过的irp指针。devobj这里要指定为真实的设备，irps->FileObject是打开的FILE_OBJECT.这个irps从原来的CREATE IRP中取得。

```
PIO_STACK_LOCATION irps = IoGetCurrentIrpStackLocation(Irp);
```

TDI_QUERY_ADDRESS_INFO是一个查询码。表示我们要查询的是地址信息。tdi_create_addrobj_complete2是我们这个查询irp完成之后所调用的完成函数。而mdl是一块内存区。mdl的分配方法为:

```
TDI_ADDRESS_INFO *tai = (TDI_ADDRESS_INFO *)malloc_np(TDI_ADDRESS_INFO_MAX);
PMDL mdl = IoAllocateMdl(tai, TDI_ADDRESS_INFO_MAX, FALSE, FALSE, NULL);
MmBuildMdlForNonPagedPool(mdl);
```

malloc_np是tdi_fw使用的内存分配函数。如果你不用tdi_fw,可以调用ExAllocatePool。TDI_ADDRESS_INFO是一个传输层地址信息结构。里面可以含有任何传输层协议的地址。然后发出这个irp即可:

```
status = IoCallDriver(devobj, query_irp);
```

那么最后对IP地址和端口的解析发生在tdi_create_addrobj_complete2中:

```
tdi_create_addrobj_complete2(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp, IN PVOID
Context)
{
    if(Irp->MdlAddress)
    {
        // 得到Mdl所指的地址
        TDI_ADDRESS_INFO *tai = (TDI_ADDRESS_INFO *)MmGetSystemAddressForMdl(Irp-
>MdlAddress);
        // 得到一个地址结构
        TA_ADDRESS *addr = tai->Address.Address;
        // 打印取得的信息
        KdPrint(("[tdi_fw] tdi_create_addrobj_complete2: address: %x:%u\n",
            ntohl(((TDI_ADDRESS_IP *)(addr->Address))->in_addr),
            ntohs(((TDI_ADDRESS_IP *)(addr->Address))->sin_port)));
    }
```

```
}
```

当然，还有个问题就是你得自己设法把你取得的ip地址和端口保存起来。

(10) 连接终端的生成与相关信息的保存

前面我们费很多工夫，获得了当一个地址生成时，取得其IP地址和端口的代码。回忆一下前面对CREATE IRP的过滤，有两种生成请求:

```
TdiTransportAddress:   表明目前Create的是一个传输层地址.
TdiConnectionContext:  表明目前生成一个连接终端.
```

我们已经处理了第一种。处理过程颇为费事，那么当连接终端生成时，我们需要做什么呢？
这里要考虑下TDI的建立连接的方式。其过程总是：先生成一个传输层地址。然后生成一个连接终端。之后用一个DeviceIoControl将二者联系起来。然后再继续之后的操作。
连接终端的生成只代表一个连接意图的产生，并没有任何实质性的操作。但是我们有必要将它有关的信息保存下来。否则以后当联系地址与端口的DeviceIoControl发生，我们只能拿到这个连接的FILE_OBJECT，如何去获取进一步的信息？

连接相关的重要参数是一个名为CONNECTION_CONTEXT的结构。这个结构可以从EA数据中取得：

```
CONNECTION_CONTEXT conn_ctx = *(CONNECTION_CONTEXT *)(ea->EaName + ea->EaNameLength + 1);
```

一个连接终端诞生后，一个FILE_OBJECT诞生，一个CONNECTION_CONTEXT也诞生了。因为以后所有的DeviceIoControl截获到，我们能得到的都只是FILE_OBJECT，所以我们应该在内存中保存一张表，把FILE_OBJECT和CONNECTION_CONTEXT对应起来。
实际上地址生成也是一样的，我们应该把FILE_OBJECT和生成的地址对应起来。这样后续的事件发生的时候，我们才知道是什么地址。

```
// 生成的处理
int tdi_create(PIRP irp, PIO_STACK_LOCATION irps, struct completion *completion)
{
    ... ...

    if (ea != NULL) {
```

```
            if (ea->EaNameLength == TDI_TRANSPORT_ADDRESS_LENGTH &&
                memcmp(ea->EaName, TdiTransportAddress,
TDI_TRANSPORT_ADDRESS_LENGTH) == 0) {

                ... ...

                // 创建表单元进行保存
                status = ot_add_fileobj(irps->DeviceObject, irps->FileObject, FILEOBJ_ADDROBJ,
ipproto, NULL);

            }
            else if (ea->EaNameLength == TDI_CONNECTION_CONTEXT_LENGTH &&
                memcmp(ea->EaName, TdiConnectionContext,
TDI_CONNECTION_CONTEXT_LENGTH) == 0) {

                // 获得CONNECTION_CONTEXT
                CONNECTION_CONTEXT conn_ctx = *(CONNECTION_CONTEXT *)
                    (ea->EaName + ea->EaNameLength + 1);

                KdPrint(("[tdi_fw] tdi_create: devobj 0x%x; connobj 0x%x; conn_ctx 0x%x\n",
                    irps->DeviceObject,
                    irps->FileObject,
                    conn_ctx));

                // 创建表单元进行保存
                status = ot_add_fileobj(irps->DeviceObject, irps->FileObject,
                    FILEOBJ_CONNOBJ, ipproto, conn_ctx);

                if (status != STATUS_SUCCESS) {
                    KdPrint(("[tdi_fw] tdi_create: ot_add_fileobj: 0x%x\n", status));
                    return FILTER_DENY;
                }
            }
        ... ...
    }
```

可以使用哈希表，这种算法和驱动没有什么关系。有兴趣的读者可自己尝试下。

(11) TDI_ASSOCIATE_ADDRESS的过滤

有两种IRP都是DEVICE IO CONTROL，他们的主能号为IRP_MJ_DEVICE_CONTROL和IRP_MJ_INTERNAL_DEVICE_CONTROL。在TDI中，两套接口功能完全重复，起相同的作用。但是IRP_MJ_INTERNAL_DEVICE_CONTROL一般用于内核内部发送设备控制命令，而IRP_MJ_DEVICE_CONTROL用于应用层向驱动层发送设备(此为wowocock语,本人未研究)。

那么我们只要研究过滤其中一套就可以了。下面都是以IRP_MJ_INTERNAL_DEVICE_CONTROL为例子，对于这样的irp，你获取次功能号，主要有以下几种：

```
TDI_ASSOCIATE_ADDRESS
TDI_DISASSOCIATE_ADDRESS
...
```

省略的我们以后再说，以使读者耳目清净。

次功能号为TDI_ASSOCIATE_ADDRESS的irp用于把一个传输层地址对象和一个连接对象联系起来，在这里我们要做的是，把我们前面用ot_add_fileobj所保存的两组信息，地址和连接，联系起来，使我们得到一个连接上下文对象的时候，能立刻知道我所使用的本地地址。

```
// 首先找到原来保存过的这个连接的FileObject.
ote_conn = ot_find_fileobj(irps->FileObject, &irql);
if (ote_conn == NULL) {
    ...
}
// 在连接上记录所联系的传输层地址的FileObject.
ote_conn->associated_fileobj = addrobj;
```

当然这有个问题就是我如何从irp中去获得上面传入的传输层地址的FileObject，这个对象的句柄被保存在irp的栈空间参数中：

```
HANDLE addr_handle = ((TDI_REQUEST_KERNEL_ASSOCIATE *)(&irps->Parameters))->AddressHandle;
```

这里拿到的是句柄，并非FILE_OBJECT结构指针，下面获取FileObject：

```
PFILE_OBJECT addrobj = NULL;
status = ObReferenceObjectByHandle(addr_handle, GENERIC_READ, NULL, KernelMode, &addrobj, NULL);
if (status != STATUS_SUCCESS) {
    ...
}
```

这样得到的就是该连接所联系的地址的FileObject指针了。

然后记得所有用ObReferenceObjectByHandle引用的对象指针用完后都必须调用ObDereferenceObject解引用，否则你的系统核心内存肯定越来越少，这种错误很难发觉：

```
if (addrobj != NULL)
    ObDereferenceObject(addrobj);
```

如果抱怨不知道哪里发生了内存泄漏，可不要怪我没有提醒你:)。

此外似乎把CONNECTION_CONTEXT指针和地址FileObject联系起来也很重要，因为可能发生这样的情况，我们仅仅得倒一个CONNECTION_CONTEXT *，我们也要立刻知道本地的地址，这样我们写到另一个表中：

```
status = ot_add_conn_ctx(addrobj, ote_conn->conn_ctx, irps->FileObject);
if (status != STATUS_SUCCESS) {
    ...
}
```

然后TDI_DISASSOCIATE_ADDRESS则完全是以上操作的逆操作，我解除他们之间的关系。

```
// delete connnection object
ote_conn = ot_find_fileobj(irps->FileObject, &irql);
if (ote_conn == NULL) {
    ...
}
// delete link of (addrobj, conn_ctx)->connobj
status = ot_del_conn_ctx(ote_conn->associated_fileobj, ote_conn->conn_ctx);
if (status != STATUS_SUCCESS) {
    ...
}
```

以上就是TDI_ASSOCIATE_ADDRESS要进行的过滤了。

(12) TDI_CONNECT的过滤

当一个irp的主功能号为IRP_MJ_INTERNAL_DEVICE_CONTROL时，子功能号有如下可能:

```
TDI_ASSOCIATE_ADDRESS
TDI_DISASSOCIATE_ADDRESS
```

```
TDI_CONNECT
...
```

前两个在前一节讲述过。省略的我们以后再说，以使读者耳目清净。TDI_CONNECT这个请求仅仅发生与本地试图主动连接外界时。若是外界连接本地，这个请求并不会发生。这个请求如果成功完成，那么连接就已经建立。所以作为监控安全的软件，我们将在这里做主要的安全工作。至少要做以下几点:

1.用户进程使用本地什么地址？（在前面已经解决了这个问题）
2.用户进程试图连接远程什么地址？
3.我们是否允许这个访问的发生?
4.是否要记录关于这次访问的日志？
虽然说问题1已经解决，但我们还是回顾一下，我们得到irp后，首先要得到这个irp的FileObject，这从栈空间指针irps中得到：

```
PIO_STACK_LOCATION irps;
// 获取当前irp当前栈空间
irps = IoGetCurrentIrpStackLocation(irp);

// 判断请求类型
if (irps->MajorFunction == IRP_MJ_INTERNAL_DEVICE_CONTROL &&
    irps->MiniorFunction == TDI_CONNECT) {


    // 然后我们得到以前保存的连接对象:
    ote_conn = ot_find_fileobj(irps->FileObject, &irql);


    ... ...

    // 如果这是一个TCP协议的连接请求,可以进一步得到我们原来保存的传输层地址信息:
    addrobj = ote_conn->associated_fileobj;
}
```

有趣的是这个请求虽然是连接请求，但是没有连接的UDP也会有这个请求，只是这时，irps->FileObject并非一个连接对象的FileObject，而是本地传输层地址的FileObject。
至于远程地址，可以从irps的参数中得到：

```
PTDI_REQUEST_KERNEL_CONNECT param = (PTDI_REQUEST_KERNEL_CONNECT)(&irps->Parameters);
TA_ADDRESS *remote_addr = ((TRANSPORT_ADDRESS *)(param->RequestConnectionInformation->RemoteAddress))->Address;
```

下面的处理，就看你的兴趣了。

(13) TDI_SEND,TDI_RECEIVE,TDI_SEND_DATAGRAM,TDI_RECEIVE_DATAGRAM

当一个irp的主功能号为IRP_MJ_INTERNAL_DEVICE_CONTROL时，子功能号主要的可能如下:

TDI_ASSOCIATE_ADDRESS
TDI_DISASSOCIATE_ADDRESS
TDI_CONNECT
TDI_SEND
TDI_RECEIVE
TDI_SEND_DATAGRAM
TDI_RECEIVE_DATAGRAM
...

将成为网络安全专家的诸位应该已了解两种主要的传输层协议的传输方式：流式与报式。主要的区别在于，流式传输并不关心你每次传输多少，在连接打开到连接中断或者关闭之间，你第一次传输了500字节，第二次传输了1000字节，这和一次传输1500字节是没有区别的。接收者不了解你发了几次，你每次发了多少，只关心这个连接上你发送来1500字节的数据，先发的永远先到。而报式传输则根本没有连接。一次发送一个数据报。后发的包并不保证一定在先发的包之前被接收。这就反过来了，顺序变得不重要，长度和次数变得重要。
对应的两种主要的协议是TCP和UDP,他们使用同样格式的传输层地址，但对应了流式与报式两种不同的传输方式。

TDI_SEND，TDI_RECEIVE用于流方式的发送和接收的过滤，因此UDP收发数据包是不会经过他们的。

作为一个安全软件，我们可以做以下一些事:
1.得到发送这些数据的那个连接。
2.检查一下这些数据是否符合我们的要求，可能我们限制只允许送某类数据。
3.在数据里检索病毒。
4.把数据加密后再发送(如果你对方会解密的话)
5.把数据修改后发送。
6.不允许发送。
7.把发送的数据备份留念。

首先问如何得到连接，这是简单的。irps->FileObject就是连接的文件对象，我们前面保存过这

个连接的相关信息。

```
// 获取前面我们保存的信息
ote_conn = ot_find_fileobj(irps->FileObject, &irql);
```

从ote_conn可以得到连接上下文指针，地址端口等等。

另一个问题是如何得到要发送的或者接受的数据，这更为简单，因为irp->MdlAddress就是含有这些数据的MDL地址。
如果要修改这些数据，你可以直接修改，但是我不知道如何变动发送数据的大小，也许你可以重新分配MDL，并修改TDI_REQUEST_KERNEL_SEND中的参数然后往下传递，返回时修改还原返回给应用层，但是作为流式可靠连接协议，我想这是不可行的。

要禁止发送或者接受是简单的，直接把这个IRP返回错误即可。
TDI_SEND_DATAGRAM,TDI_RECEIVE_DATAGRAM用于报式传输.数据获取方法与上相同.但是获取地址的方式只有一点不同：
报式协议由于没有连接的概念，所以irps->FileObject就不再是连接对象的FileObject了，直接就是传输层地址的FileObject，这个地址对象我们在前面生成的时候就保存在表中，因此查询更简单了。

```
// 获取我们前面保存的地址
ote_addr = ot_find_fileobj(irps->FileObject, &irql);
```

发送与接收就介绍到这里。

(14)设置事件

在TDI网络通信机制中，除去普通的生成地址与连接，以及发送与接收，设置事件也是非常重要的一类请求。设置事件的本质是，客户将指定一个回调函数。当网络上某一事件发生时，下层协议应该调用此函数来通知应用程序该事件的发生。
侦听就是一种典型的事件设置。用户使用一个socket调用listen时，一个类型为TDI_EVENT_CONNECT的设置事件请求将发到下层协议。下层协议将得倒一个回调函数指针。当有外来的主机连接符合这个事件所设置的端口时，该回调函数将被调用。
当一个irp的主功能号为IRP_MJ_INTERNAL_DEVICE_CONTROL时，子功能号主要有如下可能:

TDI_ASSOCIATE_ADDRESS

TDI_DISASSOCIATE_ADDRESS
TDI_CONNECT
TDI_SEND
TDI_RECEIVE
TDI_SEND_DATAGRAM
TDI_RECEIVE_DATAGRAM
TDI_SET_EVENT_HANDLER
...

次功能号为TDI_SET_EVENT_HANDLER时，这是一个事件设置的请求。事件有相当多的种类。我们得到此类请求后，第一件事是获得事件的种类：

```
// 得到请求的参数
PTDI_REQUEST_KERNEL_SET_EVENT r = (PTDI_REQUEST_KERNEL_SET_EVENT)&irps->Parameters;
```

参数格式如下:

```
struct _TDI_REQUEST_KERNEL_SET_EVENT {
    LONG EventType;
    PVOID EventHandler;
    PVOID EventContext;
} TDI_REQUEST_KERNEL_SET_EVENT, *PTDI_REQUEST_KERNEL_SET_EVENT;
```

其中EventType是事件种类。事件种类比较多，DDK文档上提到的有以下几种:

TDI_EVENT_CONNECT
TDI_EVENT_DISCONNECT
TDI_EVENT_RECEIVE
TDI_EVENT_CHAINED_RECEIVE
TDI_EVENT_RECEIVE_EXPEDITED
TDI_EVENT_CHAINED_RECEIVE_EXPEDITED
TDI_EVENT_RECEIVE_DATAGRAM
TDI_EVENT_CHAINED_RECEIVE_DATAGRAM
TDI_EVENT_SEND_POSSIBLE
TDI_EVENT_ERROR
TDI_EVENT_ERROR_EX

此外，EventHandler是一个回调函数。EventHandler也有可能为空。此时则是一个解除事件请求。原来设置的事件（回调函数）将被取消。

EventHandler非空时，对应不同的事件类型，有不同的回调函数。而且这些回调函数原型一般都比较复杂，非常吻合微软的编码恶习。比如当事件类型为TDI_EVENT_CONNECT 时，EventHandler将是如下的一个函数:

```
NTSTATUS tdi_event_connect(
```

```
    IN PVOID TdiEventContext,
    IN LONG RemoteAddressLength,
    IN PVOID RemoteAddress,
    IN LONG UserDataLength,
    IN PVOID UserData,
    IN LONG OptionsLength,
    IN PVOID Options,
    OUT CONNECTION_CONTEXT *ConnectionContext,
    OUT PIRP *AcceptIrp);
```

必要时，你必须过滤此函数。这在下一节详述。

(15)TDI_EVENT_CONNECT类型的设置事件的过滤

前面曾经过滤了TDI_CONNECT请求。并说过该请求仅仅在本地主动连接外部服务器的时候才会发生。那么当对方主动连接我方的时候，我们的安全监控系统将如何得知呢？

对于TCP协议，外部若连接我方，则必有我方的侦听。否则对方是不能连接的。上一节已经说过侦听本身是一个类型为TDI_EVENT_CONNECT设置事件。此时用户设置了一个回调函数，当有外部连接我方端口时，该回调函数将被调用。

我们过滤的方法则是偷梁换柱。当用户试图设置该回调函数时，我们用我们的回调函数取代之，并保存原回调函数的指针。在我们的回调函数调用完毕后，再调用用户设置的回调函数。那么当我们的回调函数被调用时，说明有一个外部主机在试图连接我们侦听的端口。代码如下:

```
PIO_STACK_LOCATION irps;
// 获取当前irp当前栈空间
irps = IoGetCurrentIrpStackLocation(irp);

// 判断请求类型
if (irps->MajorFunction == IRP_MJ_INTERNAL_DEVICE_CONTROL &&
    irps->MiniorFunction == TDI_SET_EVENT_HANDLER) {
    // 得到请求的参数
    PTDI_REQUEST_KERNEL_SET_EVENT r = (PTDI_REQUEST_KERNEL_SET_EVENT)
&irps->Parameters;
    if(r->EventType == TDI_EVENT_CONNECT &&  r->EventHandler != NULL)
    {
        // 在这里保存旧的回调函数并设置为我的回调函数.
        old_handler = r->EventHandler ;
        r->EventHandler = my_handler;       // ... ... (注1)
```

```
        }
    }
```

然后是我们在我们的回调过滤中应该如何做？那么看下我们的回调函数原型：

```
NTSTATUS
mytdi_event_connect(
    IN PVOID TdiEventContext,
    IN LONG RemoteAddressLength,
    IN PVOID RemoteAddress,
    IN LONG UserDataLength,
    IN PVOID UserData,
    IN LONG OptionsLength,
    IN PVOID Options,
    OUT CONNECTION_CONTEXT *ConnectionContext,
    OUT PIRP *AcceptIrp)
```

所有信息都必须从IN参数中去获取。倒是获取对方地址是一件简单的事情，因为数据都已放在参数中了，其他的主要信息，必须通过上下文TdiEventContext传递进来。

我们自行定义了一个上下文信息结构，保存我们需要的信息：

```
typedef struct {
    PFILE_OBJECT    fileobj;            /* address object */
    PVOID        old_handler;    /* old event handler */
    PVOID        old_context;    /* old event handler context */
} TDI_EVENT_CONTEXT;
```

fileobj将帮助我们取得我们保存过得连接信息，地址信息，见前面的章节。这个fileobj将是连接对象的FileObject。

old_handler和old_context将保存原来的回调与原来的上下文，在前面代码的(注1)处，修改为以下代码就可以完成：

```
ctx->fileobj = irps->
ctx->old_handler = r->EventHandler;
ctx->old_context = r->EventContext;
r->EventHandler = my_handler;
r->EventContext = ctx;
```

别忘记在mytdi_event_connect中处理完毕后，要调用旧的回调函数：

```
status = ((PTDI_IND_CONNECT)(ctx->old_handler))
    (ctx->old_context, RemoteAddressLength, RemoteAddress,
    UserDataLength, UserData, OptionsLength, Options, ConnectionContext,
    AcceptIrp);
```

最后如果你对这个回调函数返回STATUS_CONNECTION_REFUSED，这个连接请求将不能建立。

(16) 一个传说中的问题

有一个传说中的问题，那就是netbt这个设备发送TCP数据的时候，无法被我们的TDI过滤驱动过滤到。尽管我们前面已经过滤了TDI_SEND。netbt是Netbios Over Tcp/Ip，是用于计算机名字解析，对于想监控网络邻居的安全系统非常重要。不过更重要的是，如果netbt可以绕过我们的安全监控，那么别的程序也可以，这个漏洞必须弥补。

有人发现netbt将直接获取函数指针TCPSendData进行数据发送，既然不通过发送TDI_SEND请求，那么当然可以绕过我们的监控了。

研究tcpip.sys的代码，以下是tcpip.sys 的分发函数节选，其中有一个IRP_MJ_DEVICE_CONTROL的功能号码为IOCTL_TDI_QUERY_DIRECT_SEND_HANDLER。如果你发出这个请求，tcpip.sys将把TCPSendData这个函数的指针返回给你，你就可以直接用它来发送数据。

```
TDI_STATUS TCPDispatch(PDEVICE_OBJECT pDeviceObject,PIRP Irp)
{
    ...

    irpStack = IoGetCurrentIrpStackLocation(Irp);
    Irp->IoStatus.Information = 0;

    switch (irpStack->MajorFunction)
    {
        ... ...

        case IRP_MJ_DEVICE_CONTROL:
            if (NT_SUCCESS(TdiMapUserRequest(pDeviceObject,Irp,irpStack)))
                return TCPDispatchInternalDeviceControl(pDeviceObject,Irp);
            else
            {
                if (irpStack->Parameters.DeviceIoControl.IoControlCode
                    == IOCTL_TDI_QUERY_DIRECT_SEND_HANDLER)
                {
                    // exists only in W2K/XP
```

```
                    if (Irp->RequestorMode == UserMode)
                        ProbeForWrite (
                        irpStack->Parameters.DeviceIoControl.Type3InputBuffer,
                        sizeof(PULONG),
                        4
                        );
                    irpStack->Parameters.DeviceIoControl.Type3InputBuffer
                        = TCPSendData;
                    Status = STATUS_SUCCESS;
                }
                else
                    return TCPDispatchDeviceControl(Irp,irpStack);
            }
            break;
            ... ...

        }
        ... ...
    }
```

这样一来，我们在IRP_MJ_DEVICE_CONTROL中加以特殊的处理，这个函数返回后，我们得到了TCPSendData这个函数，我们将保存他，并代之以我们自己的过滤函数。

即使是netbt发送tcp数据，也不得不通过我们的过滤函数。这个漏洞就被弥补上了。相关代码如下:

```
    case IRP_MJ_DEVICE_CONTROL:
        if(irps->Parameters.DeviceIoControl.IoControlCode ==
IOCTL_TDI_QUERY_DIRECT_SEND_HANDLER)
        {
            void *buf = irps->Parameters.DeviceIoControl.Type3InputBuffer;

            // send IRP to original driver
            status = tdi_dispatch_complete(DeviceObject, irp, FILTER_ALLOW, NULL, NULL);
            if(buf != NULL && status == STATUS_SUCCESS)
            {
                old_TCPSendData = *(TCPSendData_t **)buf;
                *(TCPSendData_t **)buf = new_TCPSendData;
            }
        }
```

new_TCPSendData是我们自己写的TCP发送数据函数，这其中将过滤发送的数据，需要的话最后调用old_TCPSendData来实际发送数据。

(17)收尾与清理的工作

主功能号的IRP_MJ_CLEANUP的IRP将用于删除你所保留的连接和地址信息。不要与IRP_MJ_CLOSE相混淆，任何应用程序关闭一个FileObject所对应的句柄，都会导致主功能号为IRP_MJ_CLOSE的IRP被发到我们的驱动。但是一个句柄只是对一个FileObject的引用，句柄的关闭并不意味着FileObject的消亡。

当引用计数下降到0时，FileObject将消亡，此时IRP_MJ_CLEANUP将被调用，这时你可以删除从这个FileObject（地址对象或者连接对象）生成以来，所创建和维护的信息表项，以防止内存泄漏。

我们已经遍历的讨论了三种主要的IRP，还剩余我们所说的IRP_MJ_DEVICE_CONTROL。这种IRP功能与IRP_MJ_INTERNAL_DEVICE_CONTROL重合，你可以用如下的一个调用，把一个现有的IRP_MJ_DEVICE_CONTROL，转换为IRP_MJ_INTERNAL_DEVICE_CONTROL。

```
NTSTATUS
TdiMapUserRequest(
    IN PDEVICE_OBJECT  DeviceObject,
    IN PIRP  Irp,
    IN PIO_STACK_LOCATION  IrpSp);
```

若转换失败，则不需要过滤，直接下发。

若转换成功，则作为IRP_MJ_INTERNAL_DEVICE_CONTROL过滤，然后视情况处理，也可以直接下发，起和原来的IRP一样的作用。

具体代码如下：

```
case IRP_MJ_DEVICE_CONTROL:

    // 转换为内部控制irp
    status = TdiMapUserRequest(DeviceObject, irp, irps);

    if (status != STATUS_SUCCESS) {
        // 得到函数指针将返回到的地址
        void *buf = (irps->Parameters.DeviceIoControl.IoControlCode ==
IOCTL_TDI_QUERY_DIRECT_SEND_HANDLER) ?
            irps->Parameters.DeviceIoControl.Type3InputBuffer : NULL;
        // 下发到真实协议设备
        status = tdi_dispatch_complete(DeviceObject, irp, FILTER_ALLOW, NULL, NULL);

        if (buf != NULL && status == STATUS_SUCCESS) {
```

```
                    // 保存我们得倒地函数
                    g_TCPSendData = *(TCPSendData_t **)buf;
                    KdPrint(("[tdi_fw] DeviceDispatch:
IOCTL_TDI_QUERY_DIRECT_SEND_HANDLER: TCPSendData = 0x%x\n",
                        g_TCPSendData));
                    *(TCPSendData_t **)buf = new_TCPSendData;
                    }
                    break;
                }

    // 这里不用调用break,直接作为IRP_MJ_INTERNAL_DEVICE_CONTROL继续处理:
    case IRP_MJ_INTERNAL_DEVICE_CONTROL:

        ... ...

    case IRP_MJ_CLEANUP:

        // 在这里去删除保存过的相关信息
        result = ot_del_fileobj(irps->FileObject, &type);
        status = tdi_dispatch_complete(DeviceObject, irp, result,
            completion.routine, completion.context);
        break;
```

不过你将会发现这里的IRP_MJ_DEVICE_CONTROL的功能号(你可以通过irps->Parameters.DeviceIoControl.IoControlCode取得功能号)为IOCTL_TDI_QUERY_DIRECT_SEND_HANDLER，是一个麻烦的特例，具体内容在上一节中我们已经讨论过了。

有的读者可能注意到还有很多类型的设置事件，因为对我们的安全监控意义不大，所以并没有进行过滤，简单下发即可，兴趣太浓厚的读者可以继续阅读DDK的文档。

TDI驱动相关就介绍到这里了。(完)