tdi_fw贴码析(TDI开源网络防火墙分析)

转载

whatday 于 2014-07-22 10:58:03 发布 2975 收藏 1

tdi_fw是一个基于TDI的网络防火墙，继承自tdifw，完全采用AttachDevice的方式来实现功能，目标是成为一个高效轻巧的架构，并稳定运行于xp，win7的32位与64位版本。

memtrack模块

memtrack.h
memtrack.c

跟踪内存分配与释放，避免内存泄露

#if DBG 时，自定义的内存分配函数malloc_np会在分配内存的同时建立一个链表记录使用的内存，在驱动卸载时会检查该链表以找出未释放内存。

note:
DriverEntry中一开始需要memtrack_init()
DriverUnload的最后需要memtrack_free()，tdi_fw并不支持动态卸载，动态卸载仅在测试有无内存泄露时使用，在进行任意网络操作后，确定关闭了所有应用程序，并稍侯片刻，才可以动态卸载，如有未完成的网络操作进入已被卸载的tdi_fw流程就会蓝屏。
所有内存分配需使用malloc_np，释放需使用free

sock模块

sock.h 定义一些网络结构
sock.c 提供两个函数，用来把小尾序转为网络序

tdi_fw模块

tdi_fw.h 所有依赖的头文件，注意包含顺序
tdi_fw.c
DriverEntry里首先初始化几个必要的组件，后面详解，然后主要是创建三个过滤设备并绑定到：

```
status |= c_n_a_device(DriverObject, &g_tcpfltobj, &g_tcpoldobj, L"\\Device\\Tcp");
status |= c_n_a_device(DriverObject, &g_udpfltobj, &g_udpoldobj, L"\\Device\\Udp");
status |= c_n_a_device(DriverObject, &g_ipfltobj,  &g_ipoldobj,  L"\\Device\\RawIp");
```

几乎所有应用程序的网络操作都要通过这三个设备，AttachDevice后正确处理传来的请求，就能实现防火墙的功能。

创建两个设备，第一个用于本驱动的功能控制，第二个用来实时向应用层传出网络信息。

```
RtlInitUnicodeString(&us_DevName, L"\\Device\\tdifw");
RtlInitUnicodeString(&us_DevName, L"\\Device\\tdifw_nfo");
```

创建过滤设备后，就需要处理发来的IRP请求了。
我们主要关心 IRP_MJ_CREATE，IRP_MJ_DEVICE_CONTROL，IRP_MJ_INTERNAL_DEVICE_CONTROL，IRP_MJ_CLEANUP，其余的都使用：

```
IoSkipCurrentIrpStackLocation(irp);
status = IoCallDriver(old_devobj, irp);
```

来处理。

当有udp网络操作时，上层会对L"\\Device\\Udp" 传来IRP_MJ_CREATE，tcpip.sys会处理该irp，并分配这次操作用到的本地IP地址和端口。对应的irps->FileObject在逻辑上被称为地址对象。实际上这个请求完成以后，只会确定本地端口号。

网络操作为udp时，只需要创建地址对象，该端口就可以接收信息，或者说此端口为listen状态。如果用此irps->FileObject（地址对象）做参数，调用TDI_SEND_DATAGRAM，就是udp发包。

实际代码里，由tdi_create函数来处理IRP_MJ_CREATE，判断irp的FileObject在逻辑上是地址对象后，先创建向下层查询地址的IRP，（因为IRP_MJ_CREATE的完成函数里IRQL太高，不能创建IRP，所以在这里先创建查询用的IRP。当IRP_MJ_CREATE完成，在IRP_MJ_CREATE的完成函数里iocalldriver这个irp。） 然后设置IRP_MJ_CREATE的完成函数，当IRP_MJ_CREATE完成以后，由IRP_MJ_CREATE的完成函数发送这个IRP。这里请注意，发送包的函数的参数只有irp，我们能得到FileObject，它其实就是IRP_MJ_CREATE中的FileObject，要知道发送者使用的到底是本地地址，明显在IRP_MJ_CREATE我们已经获取到了，那么在IRP_MJ_CREATE做个结构记录下来，发送的时候一查便知，这就是贯穿tdi_fw的_addr_entry结构的真相。

当有tcp网络操作时，上层依然会对L"\\Device\\Tcp" 传来IRP_MJ_CREATE创建地址对象，然后会再次调用IRP_MJ_CREATE，此时对应的irps->FileObject在逻辑上被称为连接对象，发包的时候TDI_SEND使用连接对象作为参数。创建连接对象后，上层还会调用TDI_ASSOCIATE_ADDRESS把连接对象和地址对象关联起来，我们则需要记录该连接对象和它与地址对象的对应关系，这就是_conn_entry结构的真相。

IRP_MJ_CLEANUP顾名思义是清理的作用，在udp操作结束以后，会调用IRP_MJ_CLEANUP去clean掉创建的地址对象，我们也需要从自己的结构里把对应的_addr_entry抹掉。实验发现，对于tcp操作，结束时有且仅有对地址对象的IRP_MJ_CLEANUP，前面创建的连接对象，应该在TDI_DISASSOCIATE_ADDRESS里抹掉。

接下来是IRP_MJ_DEVICE_CONTROL，其实里面我们只用了TdiMapUserRequest(DeviceObject, irp, irpsp)简单的把IRP_MJ_DEVICE_CONTROL请求转化为IRP_MJ_INTERNAL_DEVICE_CONTROL，然后在IRP_MJ_INTERNAL_DEVICE_CONTROL里统一处理；那么case IRP_MJ_DEVICE_CONTROL后面必然不能有break，以便走入IRP_MJ_INTERNAL_DEVICE_CONTROL里继续处理。

IRP_MJ_INTERNAL_DEVICE_CONTROL里需要处理的irpsp->MinorFunction有这些：

```
#define TDI_ASSOCIATE_ADDRESS    (0x01)
#define TDI_DISASSOCIATE_ADDRESS (0x02)
#define TDI_CONNECT              (0x03)
#define TDI_LISTEN               (0x04)
#define TDI_ACCEPT               (0x05)
#define TDI_DISCONNECT           (0x06)
#define TDI_SEND                 (0x07)
#define TDI_RECEIVE              (0x08)
#define TDI_SEND_DATAGRAM        (0x09)
#define TDI_RECEIVE_DATAGRAM     (0x0A)
#define TDI_SET_EVENT_HANDLER    (0x0B)
```

tdi_fw用一个表来对应：

```
PVOID g_tdi_ioctls[] = {
    NULL,
    /*TDI_ASSOCIATE_ADDRESS,*/      tdi_associate_address,
    /*TDI_DISASSOCIATE_ADDRESS,*/   tdi_disassociate_address,
    /*TDI_CONNECT,*/                tdi_connect,
    /*TDI_LISTEN,*/                 tdi_deny_stub,  // for now only deny stubs for security reasons
    /*TDI_ACCEPT,*/                 tdi_deny_stub,  // for now only deny stubs for security reasons
    /*TDI_DISCONNECT,*/             tdi_disconnect,
    /*TDI_SEND,*/                   tdi_send,
    /*TDI_RECEIVE,*/                tdi_receive,
    /*TDI_SEND_DATAGRAM,*/          tdi_send_datagram,
    /*TDI_RECEIVE_DATAGRAM,*/       tdi_receive_datagram,
    /*TDI_SET_EVENT_HANDLER,*/      tdi_set_event_handler
};
```

显然g_tdi_ioctls[irpsp->MinorFunction] 就是对应irpsp->MinorFunction的处理函数。

大多数的分发函数，会进行一些过滤，如果需要“完成后”处理，就通过结构：

```
struct _completion {
    PIO_COMPLETION_ROUTINE  routine;
    PVOID      context;
};
```

传出需要设置的完成函数，然后由统一的收尾tdi_dispatch_complete来处理，该设置完成函数就设置完成函数，该下传IRP的继续下传，被阻止的行为就直接完成IRP，不往下传了。

注意TDI_SET_EVENT_HANDLER，此调用其实会设置一个处理函数，当下层到来网络操作时，会直接调用TDI_SET_EVENT_HANDLER设置的函数，通过函数参数来传递信息，就不需要创建irp了，那么自然避免了irp的效率损失。

所有的事件类型如下：

```
#define TDI_EVENT_CONNECT            ((USHORT)0) // TDI_IND_CONNECT event handler.
#define TDI_EVENT_DISCONNECT         ((USHORT)1) // TDI_IND_DISCONNECT event handler.
#define TDI_EVENT_ERROR              ((USHORT)2) // TDI_IND_ERROR event handler.
#define TDI_EVENT_RECEIVE            ((USHORT)3) // TDI_IND_RECEIVE event handler.
#define TDI_EVENT_RECEIVE_DATAGRAM   ((USHORT)4) // TDI_IND_RECEIVE_DATAGRAM event handler.
#define TDI_EVENT_RECEIVE_EXPEDITED ((USHORT)5) // TDI_IND_RECEIVE_EXPEDITED event handler.
#define TDI_EVENT_SEND_POSSIBLE      ((USHORT)6) // TDI_IND_SEND_POSSIBLE event handler
#define TDI_EVENT_CHAINED_RECEIVE    ((USHORT)7) // TDI_IND_CHAINED_RECEIVE event handler.
#define TDI_EVENT_CHAINED_RECEIVE_DATAGRAM  ((USHORT)8) // TDI_IND_CHAINED_RECEIVE_DATAGRAM event handler.
#define TDI_EVENT_CHAINED_RECEIVE_EXPEDITED ((USHORT)9) // TDI_IND_CHAINED_RECEIVE_EXPEDITED event handler.
#define TDI_EVENT_ERROR_EX       ((USHORT)10) // TDI_IND_UNREACH_ERROR event handler.
```

tdi_fw用一个表来对应我们关心的内容：

```
PVOID tdi_event_handler[] = {
    /*TDI_EVENT_CONNECT,*/       tdi_event_connect,
    /*TDI_EVENT_DISCONNECT,*/      tdi_event_disconnect,
    /*TDI_EVENT_ERROR,*/                          NULL,
    /*TDI_EVENT_RECEIVE,*/           tdi_event_receive,
    /*TDI_EVENT_RECEIVE_DATAGRAM,*/          tdi_event_receive_datagram,
    /*TDI_EVENT_RECEIVE_EXPEDITED,*/    tdi_event_receive,
    /*TDI_EVENT_SEND_POSSIBLE,*/                  NULL,
    /*TDI_EVENT_CHAINED_RECEIVE,*/    tdi_event_chained_receive,
    /*TDI_EVENT_CHAINED_RECEIVE_DATAGRAM,*/  NULL,
    /*TDI_EVENT_CHAINED_RECEIVE_EXPEDITED,*/  tdi_event_chained_receive,
    /*TDI_EVENT_ERROR_EX,*/                       NULL
};
```

实验：请先check build tdi_fw，在虚拟的干净xp里加载tdi_fw，运行download.exe（源码见download.cpp），windbg里可以得到一份、tdi_fw输出的最简单的网络通讯的信息，笔者截获的内容如下：

```
*******************************************************************************
*
* This is the string you add to your checkin description
* Driver Verifier: Enabled for t********************tdi_fw********************
[tdi_fw] &g_request_event: f87d6320
[tdi_fw] c_n_a_device: \Device\Tcp   fltdevobj: 0x814e8838
[tdi_fw] c_n_a_device: \Device\Udp   fltdevobj: 0x814de648
[tdi_fw] c_n_a_device: \Device\RawIp fltdevobj: 0x814de3f8
*******************************************************************************
[tdi_fw]
[tdi_fw] tdi_create: [addrobj]       fltdevobj: 0x814de648; FileObj 0x814dee78
[tdi_fw] tdi_dispatch_complete[ALLOW]fltdevobj: 0x814de648; FileObj 0x814dee78 : major 0x0, minor 0x0.
[tdi_fw] tdi_create_addrobj_complete:fltdevobj: 0x814de648; FileObj 0x814dee78
[tdi_fw] tdi_query_addr_complete: address: 7f000001:1029, proto: 17
[tdi_fw] tdi_set_event_handler[(+)]: fltdevobj: 0x814de648; FileObj 0x814dee78; EventType: 4
[tdi_fw]  -tdi_set_event_handler: old_handler 0xf71818a0; old_context 0x814f31d0
[tdi_fw] tdi_connect: (pid:1664/1664) FileObj 0x814dee78: 7f000001:1029 -> 7f000001:1029 (ipproto = 17)
[tdi_fw]
[tdi_fw] tdi_create: [addrobj]       fltdevobj: 0x814de648; FileObj 0x815e09c0
[tdi_fw] tdi_dispatch_complete[ALLOW]fltdevobj: 0x814de648; FileObj 0x815e09c0 : major 0x0, minor 0x0.
[tdi_fw] tdi_create_addrobj_complete:fltdevobj: 0x814de648; FileObj 0x815e09c0
[tdi_fw] tdi_query_addr_complete: address: 0:59765, proto: 17
[tdi_fw] tdi_set_event_handler[(+)]: fltdevobj: 0x814de648; FileObj 0x815e09c0; EventType: 4
[tdi_fw]  -tdi_set_event_handler: old_handler 0xf71818a0; old_context 0x814f1630
[tdi_fw] tdi_send_datagram(pid:1664/1664): addrobj 0x815e09c0 (size: 31) 0:59765 -> 8080808:53
[tdi_fw] tdi_event_receive_datagram(pid:1664) addrobj 0x815e09c0: 8080808:53 -> 0:59765
[tdi_fw]
[tdi_fw] tdi_create: [addrobj]       fltdevobj: 0x814e8838; FileObj 0x814eb038
[tdi_fw] tdi_dispatch_complete[ALLOW]fltdevobj: 0x814e8838; FileObj 0x814eb038 : major 0x0, minor 0x0.
```

```
[tdi_fw] tdi_create_addrobj_complete:fltdevobj: 0x814e8838; FileObj 0x814eb038
[tdi_fw] tdi_query_addr_complete: address: 0:1030, proto: 6
[tdi_fw] tdi_set_event_handler[(+)]: fltdevobj: 0x814e8838; FileObj 0x814eb038; EventType: 1
[tdi_fw]  -tdi_set_event_handler: old_handler 0xf7182961; old_context 0x814dec88
[tdi_fw] tdi_set_event_handler[(+)]: fltdevobj: 0x814e8838; FileObj 0x814eb038; EventType: 3
[tdi_fw]  -tdi_set_event_handler: old_handler 0xf7183e16; old_context 0x814dec88
[tdi_fw] tdi_set_event_handler[(+)]: fltdevobj: 0x814e8838; FileObj 0x814eb038; EventType: 5
[tdi_fw]  -tdi_set_event_handler: old_handler 0xf718faba; old_context 0x814dec88
[tdi_fw] tdi_set_event_handler[(+)]: fltdevobj: 0x814e8838; FileObj 0x814eb038; EventType: 7
[tdi_fw]  -tdi_set_event_handler: old_handler 0xf7182ef9; old_context 0x814dec88
[tdi_fw]
[tdi_fw] tdi_create: [connobj]       fltdevobj: 0x814e8838; connobj 0x814ebcd8; conn_ctx 0x815774c0
[tdi_fw] tdi_associate_address:      fltdevobj: 0x814e8838; connobj 0x814ebcd8 ---> addrobj = 0x814eb038
[tdi_fw] tdi_connect: (pid:1664/1664) FileObj 0x814ebcd8: 0:1030 -> dcb56f94:80 (ipproto = 6)
[tdi_fw] tdi_dispatch_complete[ALLOW]fltdevobj: 0x814e8838; FileObj 0x814ebcd8 : major 0xf, minor 0x3.
[tdi_fw] tdi_send: FileObj: 0x814dee78; SendLength: 1; SendFlags: 0x0
[tdi_fw] tdi_connect_complete: connobj 0x814ebcd8
[tdi_fw] set_tcp_conn_localaddr: got CONNECT LOCAL f02000a:1030
[tdi_fw] tdi_event_receive_datagram(pid:1664) addrobj 0x814dee78: 7f000001:1029 -> 7f000001:1029
[tdi_fw] tdi_send: FileObj: 0x814ebcd8; SendLength: 253; SendFlags: 0x0
[tdi_fw] tdi_send: FileObj: 0x814dee78; SendLength: 1; SendFlags: 0x0
[tdi_fw] tdi_event_chained_receive: connobj 0x0;  312; flags: 0xe20; status 0x103
[tdi_fw] tdi_event_chained_receive: connobj 0x0; 1420; flags: 0xa20; status 0x103
[tdi_fw] tdi_event_chained_receive: connobj 0x0;   20; flags: 0xe20; status 0x103
[tdi_fw] tdi_event_chained_receive: connobj 0x0;  190; flags: 0xe20; status 0x103
[tdi_fw] tdi_event_receive_datagram(pid:1664) addrobj 0x814dee78: 7f000001:1029 -> 7f000001:1029
[tdi_fw] tdi_disconnect: connobj 0x814ebcd8 (flags: 0x2)
[tdi_fw] tdi_disassociate_address: connobj 0x814ebcd8
memtrack: Total memory leakage: 0 bytes (0 blocks)
********************************************************************************
```