Hooking into NDIS and TDI, part 1

By: andreas

This is the fist part in a series of 2 articles on how to hook into the NDIS and TDI layer. In this first one, we will discuss where and how to hook in to the NDIS layer.

In the second, we will do the same for TDI.

First, lets take a quick look at a quite simplicit view of the network stack in kernel space:
TDI
NDIS protocol layer
NDIS Intermediate layer
Miniport layer
Hardware

To be able to control data flow in NDIS, we have 3 potential points where we can either add a device / driver or hook into existing. First, we have the miniport layer, these are the drivers controlling the NIC hardware, which is a bit to low of a level for what we want at this time. Next, we have the intermediate layer. This layer would be perfect for this purpose, since it would allow us to contol the dataflow to all NDIS protocol drivers. But, it has a major drawback: To be able to add a driver to this layer, it has to be signed on a default install.

Depending on what system and under what circumstances we are installing this code, it might not be possible to get past this problem in an easy manner. Last, we have the protocol layer. Adding a driver to this layer would be easy, software such as WinPcap does that. However, in that case we will not be able to control what a user would see through for software relaying on these types of drivers, such as Ethereal. So, is there any way we can get around the driver signing issue in the intermediate layer and at the same time control data in the protocol layer and above? Yes! We can virually add a layer in between intermediate and protocol by hooking all NDIS protocol drivers and their protocol functions.

When an NDIS protocol is registered, it is recorded in a linked list. Each element in this linked list carries a pointer to a structure named NDIS_OPEN_BLOCK. This structure carries the pointers for all registered function pointers for the protocol. The linked list elements are made out of a structure looking something like the following:

```
typedef struct _NDIS_LINKED_LIST {
  PNDIS_OPEN_BLOCK pOpenBlock;
  PVOID p;
  REFERENCE ref;
  struct _NDIS_LINKED_LIST *Next;
} NDIS_LINKED_LIST,*PNDIS_LINKED_LIST;
```

It was a year or so since I played with this code, so I can actually not remember the real name of this struct. Interested persons can find it through google. This will also reflect in the source code later on, since it is relying on absolute offsets instead of the typedef'ed struct.

To be able to hook into all registered NDIS protocols, we need to find the first element in this linked list. This is actually returned by NdisRegisterProtocol as the NDIS_HANDLE. So, what we have to do is to register a bogus NDIS protocol, save the pointer and then remove the protocol. This will give us the ability to walk through the list of registered NDIS protocols and exchange existing function points to functions we control.

First, we register the bogus protocol to get the pointer. To make sure the registration does not fail, the protocol we register needs to have a ReceiveHandler:

```
NDIS_STATUS DummyNDISProtocolReceive(
    IN NDIS_HANDLE ProtocolBindingContext,
    IN NDIS_HANDLE MacReceiveContext,
    IN PVOID HeaderBuffer,
    IN UINT HeaderBufferSize,
    IN PVOID LookAheadBuffer,
    IN UINT LookAheadBufferSize,
    IN UINT PacketSize)
```

```
{
    return NDIS_STATUS_NOT_ACCEPTED;
}

NDIS_HANDLE RegisterBogusNDISProtocol(void)
{
    NTSTATUS Status = STATUS_SUCCESS;
    NDIS_HANDLE hBogusProtocol = NULL;
    NDIS_PROTOCOL_CHARACTERISTICS BogusProtocol;
    NDIS_STRING ProtocolName;

    NdisZeroMemory(&BogusProtocol,sizeof(NDIS_PROTOCOL_CHARACTERISTICS
));
    BogusProtocol.MajorNdisVersion = 0x04;
    BogusProtocol.MinorNdisVersion = 0x0;

    NdisInitUnicodeString(&ProtocolName,L"BogusProtocol");
    BogusProtocol.Name = ProtocolName;
    BogusProtocol.ReceiveHandler = DummyNDISProtocolReceive;

    NdisRegisterProtocol(&Status,&hBogusProtocol,&BogusProtocol,
        sizeof(NDIS_PROTOCOL_CHARACTERISTICS));

    if(Status == STATUS_SUCCESS) return hBogusProtocol;
    else return NULL;
}
```

Once we have the pointer, we can deregister the protocol again:

```
void DeregisterBogusNDISProtocol(NDIS_HANDLE hBogusProtocol)
{
    NTSTATUS Status;

    NdisDeregisterProtocol(&Status,hBogusProtocol);
```

```
}
```

Once we start walking the linked list and overwriting function pointers, we need to save the old pointers to be able to call them from our functions. There are atleast 2 ways of doing this:

1. Create a linked list of "hooked instances", holding the old pointers for each protocol. When our NDIS functions are called, the linked list has to be searched for the right element.

2. Allocate one instance of our functions for each protocol we hook and write the old pointer directly into the code of the function. This is slighly more work during hooking, but should be faster during run-time than searching through a linked list for every packet.

When this code was written, I never thought of option number 2, but that is probably the option I would use today. So, enjoy option 1, it works well and I haven't seen any major performance hits from it.

For every element in the NDIS registered protocol linked list, I allocate one element in my own list and save all important pointers together with 2 context handles. The handles values are later used to find the right element for the current protocol. Relevant pointers are then overwritten to point to my versions of the send and receive functions. We also save a pointer to the NDIS_OPEN_BLOCK itself to make unhooking easy. The code walking the list and hooking into the protocol would look something like this:

```
NTSTATUS HookExistingNDISProtocols(void)
{
    UINT *ProtocolPtr;
    NDIS_HANDLE hBogusProtocol = NULL;
    PNDIS_OPEN_BLOCK OpenBlockPtr = NULL;
    PNDIS_PROTOCOL_HOOK pNode;

    hBogusProtocol = RegisterBogusNDISProtocol();
    if(hBogusProtocol == NULL) return STATUS_UNSUCCESSFUL;

    ProtocolPtr = (UINT*)hBogusProtocol;
    ProtocolPtr = (UINT*)((PBYTE)ProtocolPtr + sizeof(REFERENCE) + 8);
    ProtocolPtr = (UINT*)(*ProtocolPtr);
```

```
    while(ProtocolPtr != NULL) {
        OpenBlockPtr = (PNDIS_OPEN_BLOCK)(*ProtocolPtr);
        if(OpenBlockPtr != NULL) {
            pNode = NewNDISNode();
            if(pNode != NULL) {
                pNode->ProtocolBindingContext = OpenBlockPtr-
>ProtocolBindingContext;
                pNode->MacBindingContext = OpenBlockPtr-
>MacBindingHandle;
                pNode->OpenBlockPtr = OpenBlockPtr;
                pNode->RealSendHandler = OpenBlockPtr->SendHandler;
                //How about WanSendHandler?
                pNode->RealPostNt31ReceiveHandler = OpenBlockPtr-
>PostNt31ReceiveHandler;

                InsertNDISNode(pNode);

                OpenBlockPtr->SendHandler = NDISSendHandler;
                //How about WanSendHandler?
                OpenBlockPtr->PostNt31ReceiveHandler =
NDISPostNt31ReceiveHandler;
            }
        }

        ProtocolPtr = (UINT*)((PBYTE)ProtocolPtr + sizeof(REFERENCE) +
8);
        ProtocolPtr = (UINT*)(*ProtocolPtr);
    }

    DeregisterBogusNDISProtocol(hBogusProtocol);

    return STATUS_SUCCESS;
}
```

There are more functions in the NDIS_OPEN_BLOCK that might be of interest to hook, but if you only want to control network traffic, send and receive are enough. Another thing worth mentioning is that the NDIS_OPEN_BLOCK changes with OS versions. It looks different in Win2K compared to XP, mostly due to member names changing.

The next thing to do now is to implement send and recieve functions which searches through the linked list to find the original function pointers and then calls them if the traffic is to be passed on. If the traffic is to be altered, that is performed before calling the real protocol function. If the traffic is supposed to be dropped, we can just skip calling the real function and return with the appropriate status:

```
NDIS_STATUS NDISSendHandler(
    IN NDIS_HANDLE MacBindingHandle,
    IN PNDIS_PACKET Packet)
{
    PNDIS_PROTOCOL_HOOK Node;

    Node = FindNDISNode(MacBindingHandle,2);
    if(Node == NULL) return NDIS_STATUS_SUCCESS;

    return Node->RealSendHandler(MacBindingHandle,Packet);
}

NDIS_STATUS NDISPostNt31ReceiveHandler(
    IN NDIS_HANDLE ProtocolBindingContext,
    IN NDIS_HANDLE MacReceiveContext,
    IN PVOID HeaderBuffer,
    IN UINT HeaderBufferSize,
    IN PVOID LookAheadBuffer,
    IN UINT LookAheadBufferSize,
    IN UINT PacketSize)
{
    PNDIS_PROTOCOL_HOOK Node;

    Node = FindNDISNode(ProtocolBindingContext,1);
    if(Node == NULL) return NDIS_STATUS_SUCCESS;
```

```
    return Node-
>RealPostNt31ReceiveHandler(ProtocolBindingContext,MacReceiveContext,
        HeaderBuffer,HeaderBufferSize,LookAheadBuffer,LookAheadBufferS
ize,PacketSize);
}
```

Now, there is only one thing left, unhooking. We do this by walking our linked list of "hooked instances" and replace all pointers:

```
NTSTATUS ReleaseExistingNDISProtocols(void)
{
    PNDIS_PROTOCOL_HOOK CurrentNode;
    PNDIS_OPEN_BLOCK OpenBlockPtr = NULL;

    CurrentNode = GetFirstNDISNode();
    if(CurrentNode == NULL) return STATUS_UNSUCCESSFUL;

    while(CurrentNode != NULL) {
        OpenBlockPtr = CurrentNode->OpenBlockPtr;
        if(OpenBlockPtr != NULL) {
            OpenBlockPtr->SendHandler = CurrentNode->RealSendHandler;
            OpenBlockPtr->PostNt31ReceiveHandler = CurrentNode-
>RealPostNt31ReceiveHandler;
        }
        CurrentNode = GetNextNDISNode(CurrentNode);
    }

    return STATUS_SUCCESS;
}
```

What is left to be done? The code does not hook into NDIS protocol being registred after NDIS is hooked into. This is left up to the reader to figure out, one way to do it can be found in the win32 version of sebek.

Does the code work? Sure, I use it in a win32 version of knockd called sesame that can be found at .http://www.toolcrypt.org/.