NDIS and TDI Hooking, Part II

By: andreas

This is the second and last article on how to hook into the NDIS and TDI layer. The approach we will use will be slightly different from the NDIS case. However, a neat side effect is that this method can be used to hook into any device chain, for example the keyboard to sniff key strokes. It all boils down to getting a pointer to the device object and replace all major functions with our own dispatch function.

To be able to fully control the TDI layer, we need access to the IRP both before and after the original driver has processed it. If we have that, we can choose what the original driver should process and we can also alter results before they are returned to user-space. The "before filtering" can be accomplished in our own, new dispatch function and the "after filtering" can be accomplished in a completion routine.

First, to be able to overwrite and insert our own dispatch function, we need a pointer to the driver object we are going to hook into. An easy way to get this pointer is to call ObReferenceObjectByName with the appropriate driver name. Then we only have to save all old function pointers and overwrite the existing ones with our own. The code to do this would look something like the following:

```
DRIVER_OBJECT RealTDIDriverObject;

NTSTATUS HookTDI(void)
{
    NTSTATUS Status;
    UNICODE_STRING usDriverName;
    PDRIVER_OBJECT DriverObjectToHookPtr;
```

```
    UINT i;



    RtlInitUnicodeString(&usDriverName,L"\\Driver\\Tcpip");

    Status =
ObReferenceObjectByName(&usDriverName,OBJ_CASE_INSENSITIVE,NULL,0,IoDr
iverObjectType,KernelMode,NULL,&DriverObjectToHookPtr);
    if(Status != STATUS_SUCCESS) return Status;

    for(i = 0;i < IRP_MJ_MAXIMUM_FUNCTION;i++) {
        RealTDIDriverObject.MajorFunction[i] =
DriverObjectToHookPtr->MajorFunction[i];
        DriverObjectToHookPtr->MajorFunction[i] = TDIDeviceDispatch;
    }

    return STATUS_SUCCESS;
}
```

RealTDIDriverObject is a DRIVER_OBJECT where we save the original information to both be able to call the old functions and also be able to unhook once we are done. The orignal driver gets all its major functions overwritten with a pointer to our own dispatch function, TDIDeviceDispatch.

We now have control over the IRPs before TDI can process them. But, we still have to make sure we can also control them once TDI is done with it but before it is returned to the IO handler and user-space. We will solve this in our dispatch function with the help of a completion routine. It is not as straight forward as it sounds, since we might be hooking the last entity in the chain, we can't just insert a completion routine with IoSetCompletionRoutine (see the DDK docs), since it in that case never will be called. Completion routines are set in the next IRP stack location, not the current. If we are the last entity, there will be no next stack location in the IRP. Searching through the header files reveal IoSetCompletionRoutine as a macro which only gets the next IRP stack location and sets the CompletionRoutine pointer together with the Control

element. Following the same principcal, we can set our own completion routine to regain control over the IRP with the following dispach function:

```
NTSTATUS TDIDeviceDispatch(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp)
{
    NTSTATUS Status;
    PIO_STACK_LOCATION StackLocationPtr;

    if(Irp == NULL) return STATUS_SUCCESS;

    StackLocationPtr = IoGetCurrentIrpStackLocation(Irp);
    if(StackLocationPtr->CompletionRoutine != NULL) StackLocationPtr-
>Context =
StackLocationPtr->CompletionRoutine;
    else StackLocationPtr->Context = NULL;
    StackLocationPtr->CompletionRoutine =
(PIO_COMPLETION_ROUTINE)TDICompletionRoutine;
    StackLocationPtr->Control = SL_INVOKE_ON_SUCCESS |
SL_INVOKE_ON_ERROR |
SL_INVOKE_ON_CANCEL;

    Status =
RealTDIDriverObject.MajorFunction[StackLocationPtr-
>MajorFunction](DeviceObject,Irp);

    return Status;
}
```

What we actually do is faking a scenario where the layer above set the completion routine for us. We also save a potentially already existing completion routine in the Context element of the IRP. Control is set to invoke the completion routine in all cases. There are 2 potential issues with this code. First, we overwrite whatever is in the Context element. Second, we never save the Control element, so we don't know when to invoke

an already existing completion routine. So far, I have not seen any side-effects from doing this.

The completion routine would look something like:

```
NTSTATUS TDICompletionRoutine(PDEVICE_OBJECT DeviceObject,PIRP
Irp,PVOID
Context)
{
    COMPLETIONROUTINE RealCompletionRoutine =
(COMPLETIONROUTINE)Context;

    if(Context != NULL) return
RealCompletionRoutine(DeviceObject,Irp,NULL);
    else return STATUS_SUCCESS;
}
```

It invokes a potential completion routine as soon as it is done and returns the status from it. Finally, unhooking the driver is just a question of restoring the pointers we overwrote in the hooking function:

```
NTSTATUS ReleaseTDIDevices(void)
{
    NTSTATUS Status;
    UNICODE_STRING usDriverName;
    PDRIVER_OBJECT DriverObjectToHookPtr;
    UINT i;

    RtlInitUnicodeString(&usDriverName,L"\\Driver\\Tcpip");

    Status =
ObReferenceObjectByName(&usDriverName,OBJ_CASE_INSENSITIVE,NULL,0,IoDr
iverObjectType,KernelMode,NULL,&DriverObjectToHookPtr);
    if(Status != STATUS_SUCCESS) return Status;
```

```
    for(i = 0;i < IRP_MJ_MAXIMUM_FUNCTION;i++)
        DriverObjectToHookPtr->MajorFunction[i] =
RealTDIDriverObject.MajorFunction[i];

    return STATUS_SUCCESS;
}
```

There is another way to accomplish the same result which utilizes a more offically supported mode of operation. It is based upon attaching to the device chain with GetDeviceObject and AttachToDevice, which will allow us to process all IRPs before the real device. Once in the dispatch function we contruct a new IRP and add a completion routine to regain control of the IRP before it is returned to the IO system and user-space.

One last important thing to mention; This code is quite untested. It seems to work as intended but it has never been used in any major applications, so use it on your own risk. With that said, hope you have enjoyed this little article series.