

# Designing A Kernel Key Logger

## *A Filter Driver Tutorial*

By Clandestiny

### Introduction:

The following tutorial outlines the design of a simple key logger implementation using a kernel filter driver. Although the key logger itself is only proof-of-concept and lacks the functionality of a useful attack tool, it presents filter drivers as a potentially useful (and underutilized) rootkit hooking technique while demonstrating a few of the basic programming challenges that distinguish kernel design from user land development. The filter is based on the method shown in the ctrl2cap program at sysinternals.com.

If you've written a kernel driver before you already know that the drivers for each of the system's hardware devices are layered into a hierarchal "device stack" and that a hardware device may have one or more drivers associated with it. Filter drivers are a special kind of driver that can be *transparently* inserted on top of or in between existing drivers in the stack. The key word, of course, is "transparently" because the surrounding drivers are unaware of the filter driver's existence. The filter may be used to add functionality to an existing lower level driver, hide or modify data being sent to an upper level driver, or to stealthily intercept data. Clearly, the application of filters to rootkit development falls into this latter category and the possibilities range from simple key loggers to complex network and file system filters.

### Installing The Keyboard Filter:

Installation of the filter is performed in the Driver Entry procedure and just like any other driver, one of the first steps is to fill in the IRP dispatch table. Filter drivers must support the same set of IRP\_MJ\_XXX requests as the underlying driver it is attached to. If we think of a filter driver as a sort of IRP net, this makes a sense. The net must have holes of the same size and position (i.e. same IRP\_MJ\_XXX slots filled in) for the lower driver to properly receive all of the IRP request types it supports. The easiest way to accomplish this is to expose a dispatch routine for every slot in the Major Function table of the filter driver object. If the filter is not interested in some of the IRPs, it simply provides a generic routine that passes them down to the next driver in the stack. A separate routine is provided for the IRPs the filter is interested in inspecting or modifying. In the case of our keyboard filter, we are only interested in inspecting the "read" IRPs. Similarly, the filter driver should set the same flags as the underlying device it is attaching to. After setting up the dispatch table, the device object is created and the device extension (a non paged memory area for the driver to store global variables) is initialized. One primary difference between regular drivers and filter drivers lies in the fact that their device objects are unnamed. We create an unnamed device object by calling IoCreateDevice and passing NULL for the device name. The actual insertion of the filter is performed by calling IoAttachDevice and supplying the filter driver's device object as well as the name of the keyboard device to which we're attaching. We also save the old pointer to the top of the stack so we know where we need to direct IRPs later after we've intercepted them. In the case of a keyboard filter, there are a couple of potential stack insertion points. The first is I8042prt.sys. This is the low level port driver for the keyboard. There is an example in the DDK showing a filter for the port driver. The second point of insertion is kbdclass.sys, Microsoft's upper level class driver for the keyboard. Both this key logger as well as the sysinternals example ctrl2cap attach to the class driver.

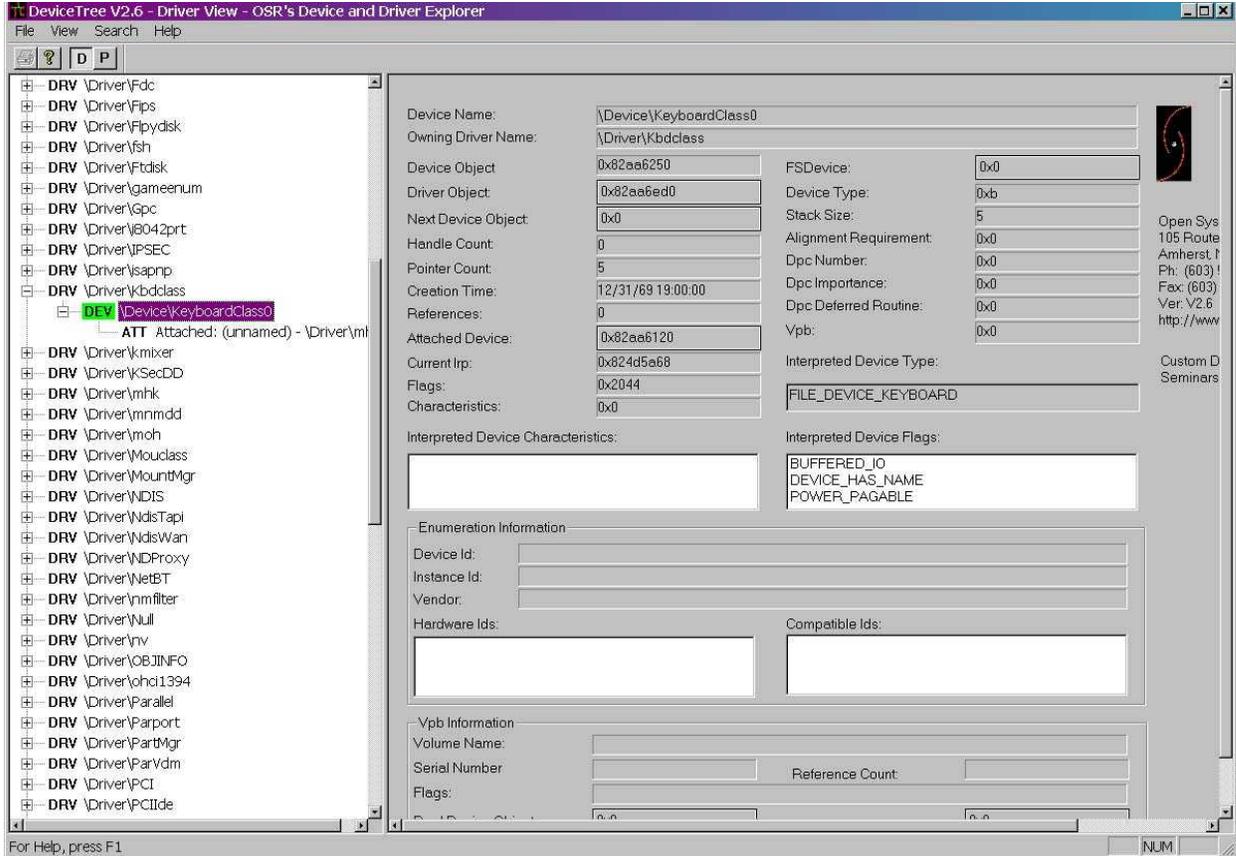
We can summarize the basic steps for attaching a filter driver as:

1. Fill in all entries in the IRP\_MJ\_XXX dispatch table for the filter driver. Unhooked IRPs receive a pointer to a "pass down" routine and hooked IRPs receive a pointer to a special "hook" routine.
2. Create an unnamed keyboard filter device object by calling IoCreateDevice.
3. Initialize the Device Extension

4. Set the flags of the filter device equal to the flags of the underlying target device.
5. Attach the filter device to the target device using IoAttachDevice.

Additional initialization performed by the key logger involves creating a file for logging, starting a system thread for processing File I/O, and setting up a few necessary synchronization objects whose purpose will be discussed later.

On a related note, one can spy out the device stack for system drivers by using the Device Tree tool provided in the DDK. The following illustration shows a filter attached to the keyboard class driver.



## Intercepting & Logging Keyboard Data:

After the filter driver is installed, the implementation can be broken down into two essential functions: intercepting and logging the data. First, we'll discuss intercepting the data. In the case of a key logger, we are primarily interested in sneaking a peek at the data being read from the keyboard by the OS when the user presses a key. In practical terms, this means intercepting the "read" IRP's of the keyboard device. The following diagram and accompanying sequence of steps outlines this process:

1. The Operating System's I/O manager sends an empty IRP packet down the device stack for the keyboard.
2. "Read" IRPs are intercepted by the filter driver's dispatch routine for IRP\_MJ\_READ on their way down the driver stack. While in this routine, the IRPs is "tagged" with a "completion routine". The "completion routine" is basically just a fancy name for a callback function that says "I want to see this IRP again when it comes back up the device stack."

When a “read” IRP is received, the tasks of the filter driver dispatch read routine are as follows:

A. It sets up the IRP stack for the lower driver by setting the next IRP stack location equal to the current IRP stack location. The stack location pointers are obtained by calling `IoGetCurrentIrpStackLocation` and `IoGetNextIrpStackLocation`.

B. It sets a completion routine on the current IRP by calling `IoSetCompletionRoutine`

C. It passes the IRP on to the next driver in the stack with `IoCallDriver`.

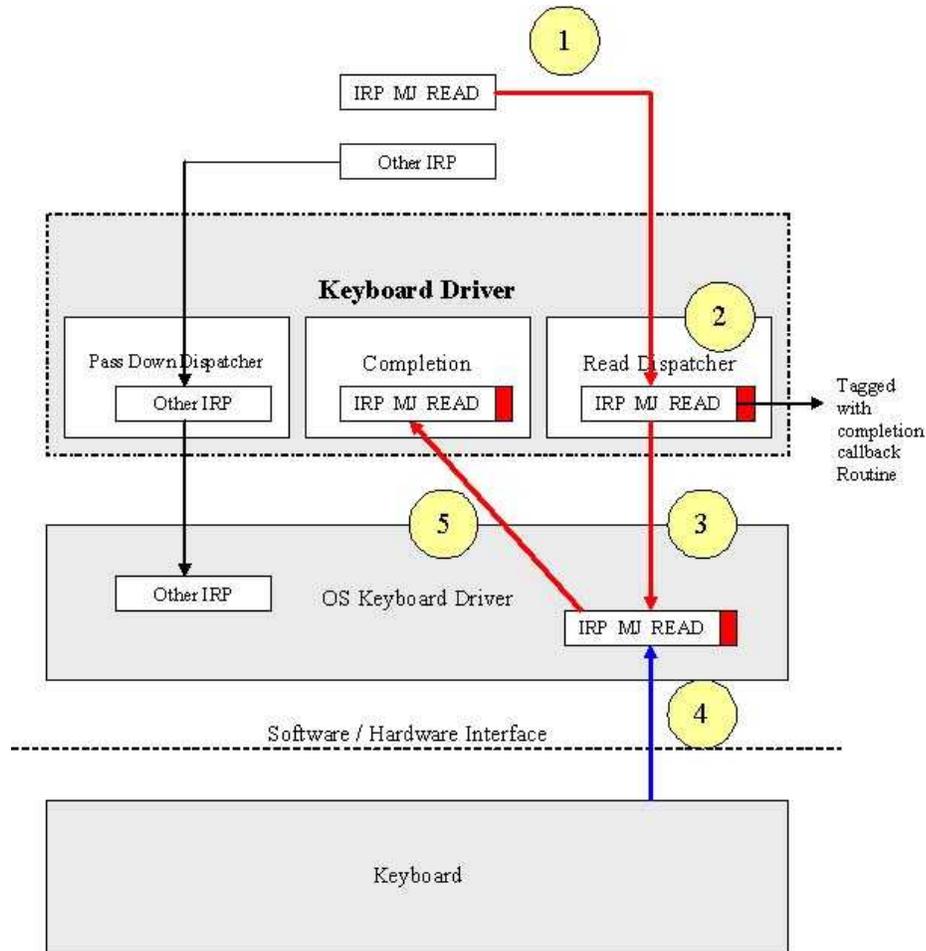
All other received IRPs are directed to a “pass down” dispatch routine which simply passes them down the stack without touching them.

3. When the tagged, empty IRP reaches the bottom of the stack at the hardware / software interface, it waits for a key press.
4. When a key on the keyboard is pressed, it “completes” the IRP. The IRP is filled with the scan code for the pressed key and sent on its way back up the device stack.
5. On its way back up the device stack, the completion routines that the IRP was tagged with on its way down the stack are called and the IRP is passed into them. This gives the filter driver an opportunity to extract the scan code information stored in the packet and add it to a queue for processing by the logging component. It is important to note that the completion routine can be called at `DISPATCH_LEVEL` and, consequently, that we have to be aware of the API and memory allocation restrictions at that level. This affects our design in a practical way. When we enter the completion routine we would like to extract the keyboard scan code and log that scan code to a file. Unfortunately, we can only accomplish the first task due to IRQL restrictions on File I/O. Because file system APIs can only be called at IRQL `PASSIVE_LEVEL`, we have to set up a separate thread running at `PASSIVE_LEVEL` to handle them. Having a separate thread also introduces the need for a queuing mechanism and synchronized access to the queue. We can use an interlocked linked list for this purpose as the `ExInterlockedInsertTailList` and `ExInterlockedInsertHeadList` are guaranteed to be mutually exclusive and this avoids having to deal with a separate mutex or spin lock. A semaphore can also be used to notify the thread doing the file I/O when there are scan codes available for it to log. This is accomplished by incrementing the semaphore count once for every received IRP. By calling `KeWaitForSingleObject` in the thread, the thread will block if the semaphore count is 0 (i.e. there is no work to do). This is more efficient than simply spinning in an infinite loop constantly checking if there is anything in the queue. The last item to be aware of is that memory for the list items must be allocated from the non paged pool since the completion routine may be running at `DISPATCH_LEVEL`.
6. Before writing the scan code out to the log file, it is converted to its ASCII representation. Unfortunately, this conversion is dependent upon the layout of the keyboard and I couldn’t find any kernel support to generically convert scan codes for different keyboard layouts. Therefore, my scan code conversion is currently limited to a manual conversion routine for a US keyboard layout.

#### A Final Caveat:

There is an issue related to unloading a filter driver which needs to be mentioned. As we know, the dispatch routine “tags” keyboard IRPs with a completion routine on their way down the stack and they sit there waiting to be “completed” when the user presses a key on the keyboard. These IRPs are therefore “pending” completion (i.e. until they are filled with data and sent back up the stack). If we attempt to unload the driver while there is an IRP pending completion, the system will crash. This is clearly due to the fact that the memory address for the completion routine will be invalid when the OS tries to call it after unloading the driver. The obvious way to solve this is to simply not unload the

driver. Like a rootkit, there is little need to unload a keylogger so this may be an acceptable solution. As an alternative, we can keep a count of the number of outstanding IRPs and refuse to unload until the count becomes 0. This alternative, however, has the unfortunate side effect that the driver won't unload until the user presses another key (ie. until the IRP is filled with keyboard data and passed back up the stack and handled properly by the completion / logging routines).



### Known Limitations:

1. As its mostly proof of concept, I never bothered to write a proper loader. I will probably add one, but in its present state the driver can be loaded and unloaded using a tool like Numega's Driver Monitor or you can use your own.
2. The scan code conversion is limited to a US layout keyboards. As previously stated, I could not find any kernel functions that would perform the scan code conversions for me so I had to write a basic one manually based on a scan code conversion chart I found. If anyone knows of good way to do generic scan code conversion in a kernel driver, I would be happy to improve this part of the code.
3. The scan code conversion does not detect the state of the caps lock key. This is a minor bug / problem.
4. At present, no effort is made to hide the driver or log file.

5. It logs to the hard coded location C:\klog.txt. When I add a proper loader, I will fix this by allowing the user to select the logging path.

### References:

1. The Windows 2000 Device Driver Book by Art Baker & Jerry Lozano
2. Windows NT File System Internals: A Developer's Guide (has a nice chapter on filter drivers).
3. ctrl2cap from sysinternals.com
4. DDK Documentation

### Contact Me:

Bug reports and suggestions are always welcome. Email me at [clandestiny@despammed.com](mailto:clandestiny@despammed.com).