Layered Driver Rootkit Detection on Microsoft Windows PCs

Christopher Wood
Faculty Mentor: Dr. Rajendra K. Raj, Department of Computer Science

Description

A rootkit is a "kit" of small programs used by attackers to gain permanent, undetectable access to the root of the target system. Rootkits are typically deployed in the kernel of the host operating system (OS) where they can directly modify the kernel memory and objects or host OS to achieve stealth and provide future functionality for the attacker.

Problem

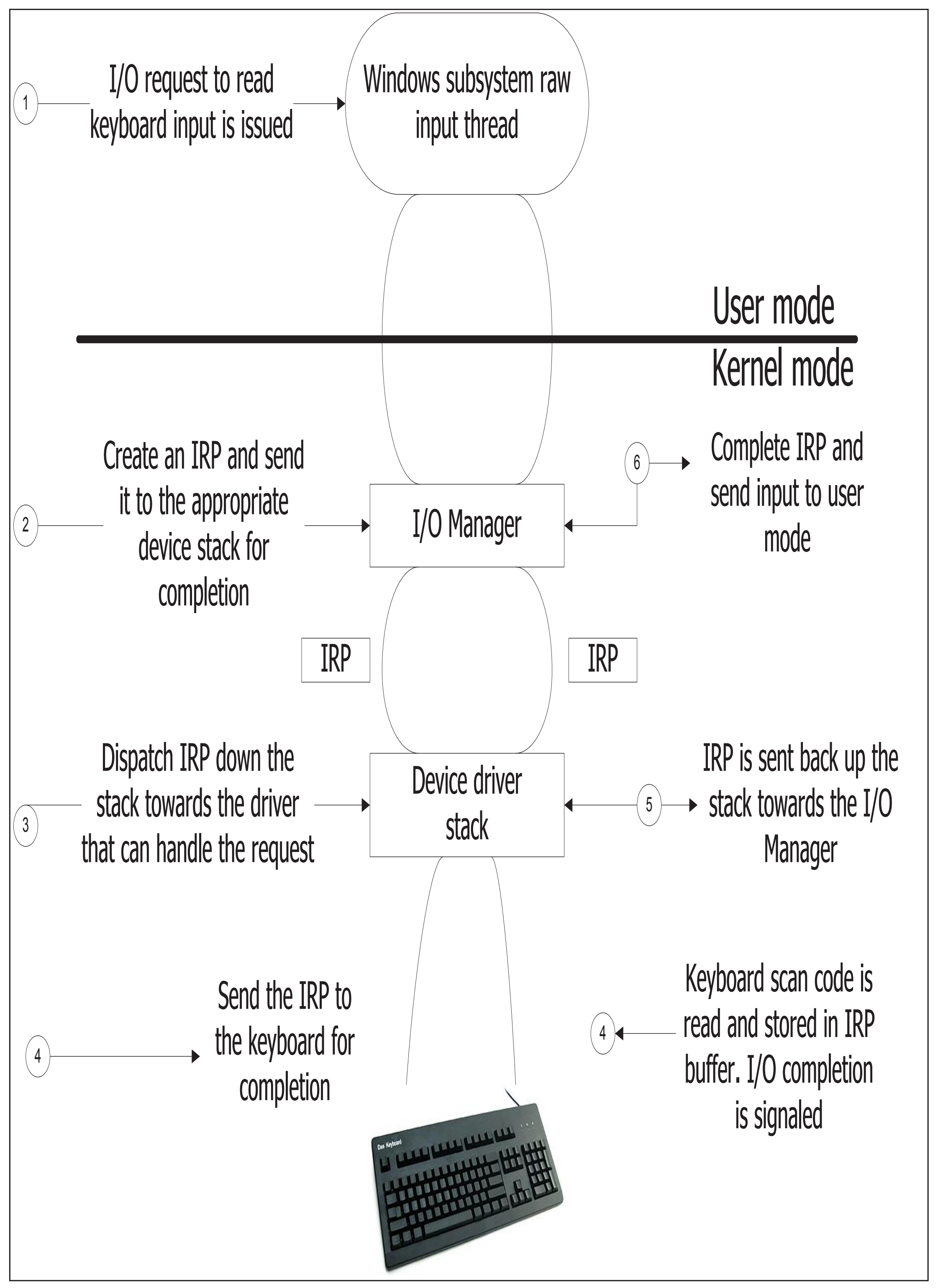
Perhaps the most useful, robust, and reliable way for an attacker to implement a rootkit on Microsoft Windows machines is as a layered driver for a human interface device (HID). With the ability to intercept, modify, and analyze I/O requests between the hardware device and the software sending the request, layered HID drivers can do whatever they want with the most vital source of information for a computer - the user. Rootkits that are implemented as layered HID drivers typically intercept information from the user, such as input from the keyboard, and transmit it elsewhere for malicious use. These types of rootkits pose an enormous threat to the safety and security of the infected machine.

Solution

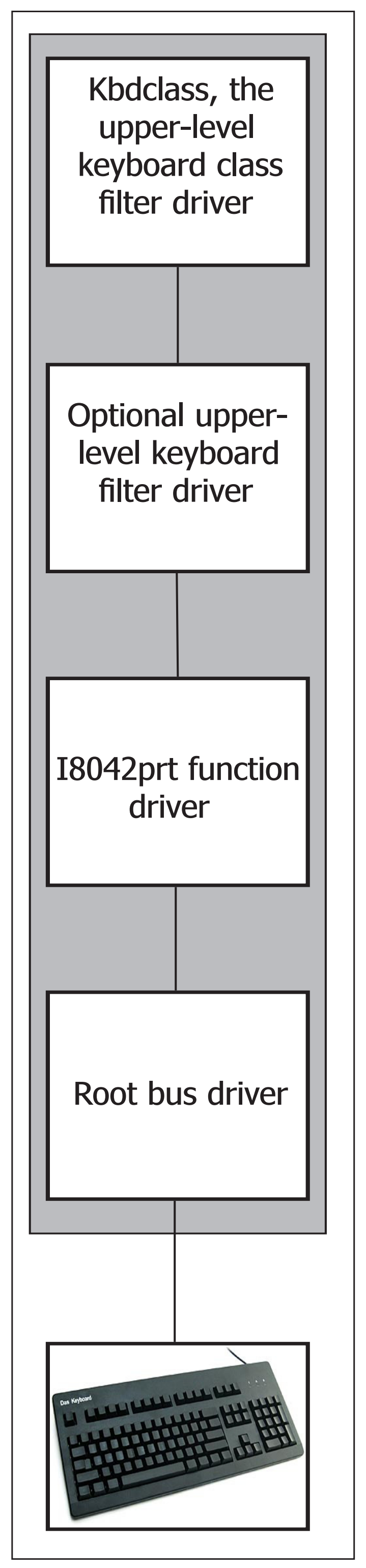
Placing a custom filter driver on top of the keyboard driver stack to determine the presence of any malicious third-party drivers on the same stack using the following techniques:

1.) Hooking the System Service Dispatch Table (SSDT) to monitor key driver support routines used by layered HID driver rootkits to record user input to the hard disk.
2.) I/O request packet (IRP) analyzing.

Typical Keyboard Read Request

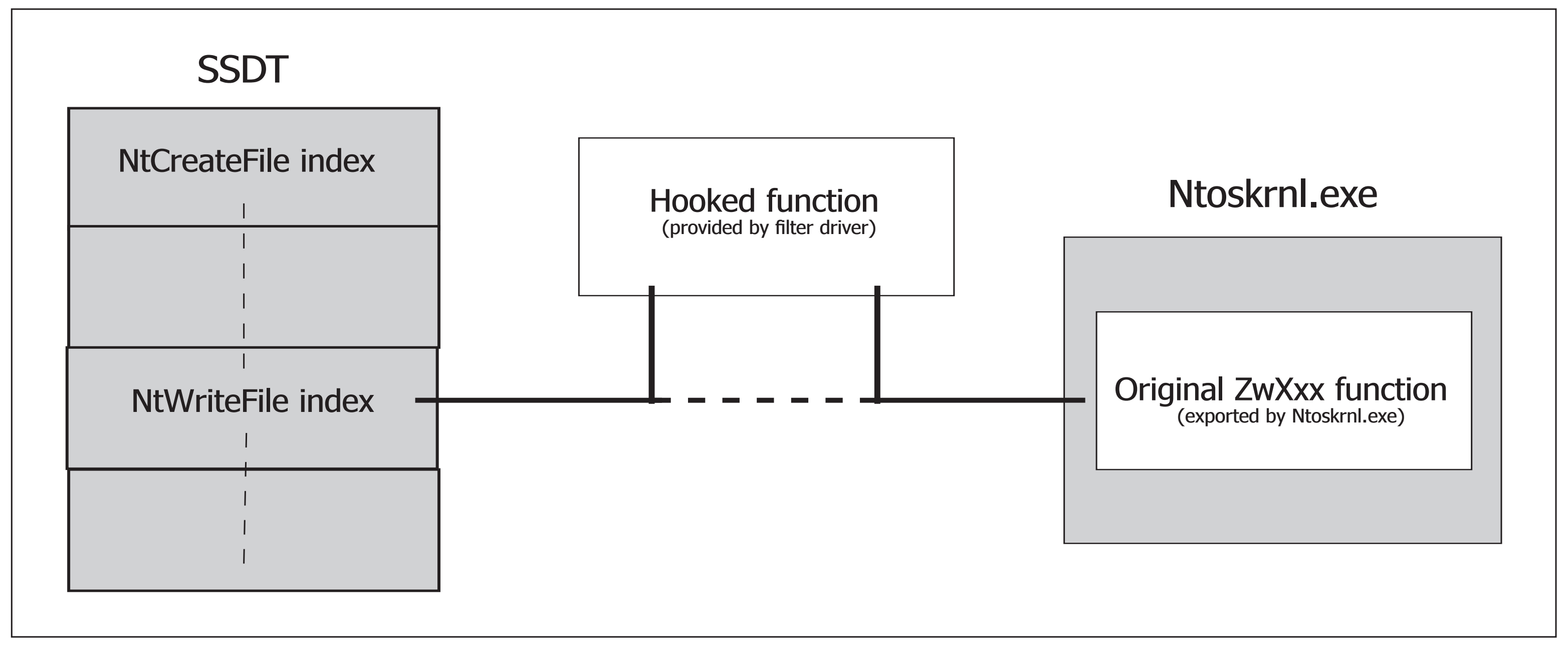

1
I/O request to read keyboard input is issued
Windows subsystem raw input thread
User mode
Kernel mode
2
Create an IRP and send it to the appropriate device stack for completion
I/O Manager
6
Complete IRP and send input to user mode
IRP
IRP
3
Dispatch IRP down the stack towards the driver that can handle the request
Device driver stack
5
IRP is sent back up the stack towards the I/O Manager
4
Send the IRP to the keyboard for completion
4
Keyboard scan code is read and stored in IRP buffer. I/O completion is signaled

Keyboard Stack


Kbdclass, the upper-level keyboard class filter driver
Optional upper-level keyboard filter driver
I8042prt function driver
Root bus driver

System Service Dispatch Table (SSDT) Hook Technique

The SSDT is a kernel structure that contains the locations in memory of the Windows native system service functions and routines. An SSDT hook changes the index of the target function in the SSDT to point from the original function to another function as illustrated below.



Hooking the SSDT can be done with the use of four powerful macros shown below as implemented in the HideProcessHookMDL rootkit [2]. SYSTEMSERVICE takes the address of a ZwXxx function and returns the address of the corresponding NtXxx function in the SSDT. The SYSCALL_INDEX macro takes the address of a ZwXxx function and returns the index of the corresponding NtXxx function in the SSDT. Lastly, the HOOK_SYSCALL and UNHOOK_SYSCALL exchange the addresses of the functions in the SSDT as needed.

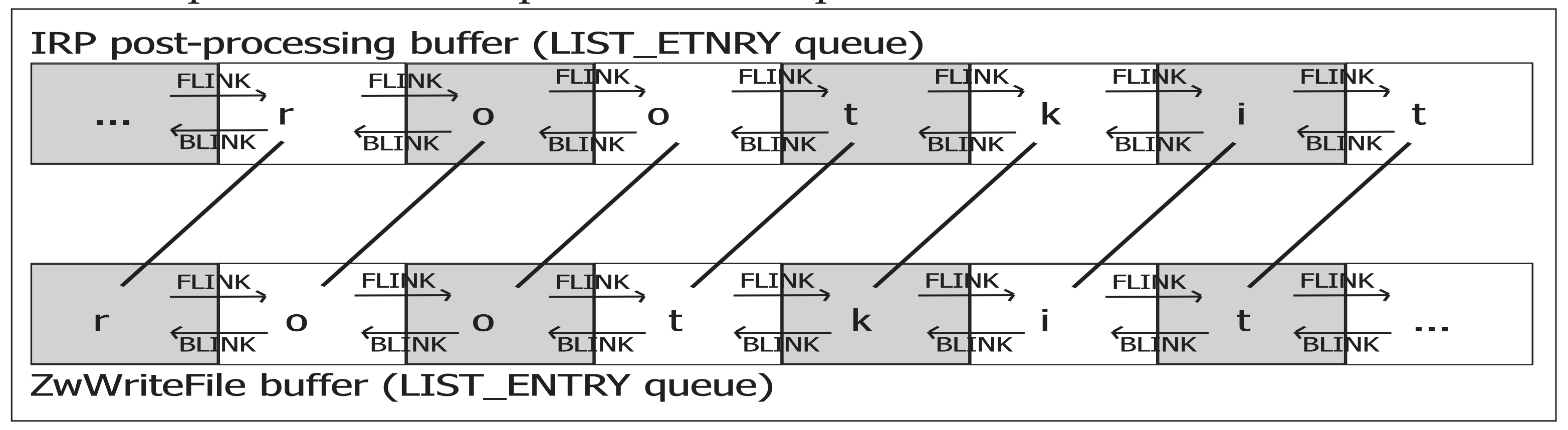
```
#define SYSTEMSERVICE(_Function)  \
   KeServiceDescriptorTable.ServiceTableBase[*(PULONG) \
   ((PUCHAR)_Function+1)]

#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)

#define HOOK_SYSCALL(_Function, _Hook, _Orig )  \
   _Orig = (PVOID) InterlockedExchange( (PLONG) \
   &MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)

#define UNHOOK_SYSCALL(_Function, _Hook, _Orig )  \
   InterlockedExchange( (PLONG) \
   &MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)
```

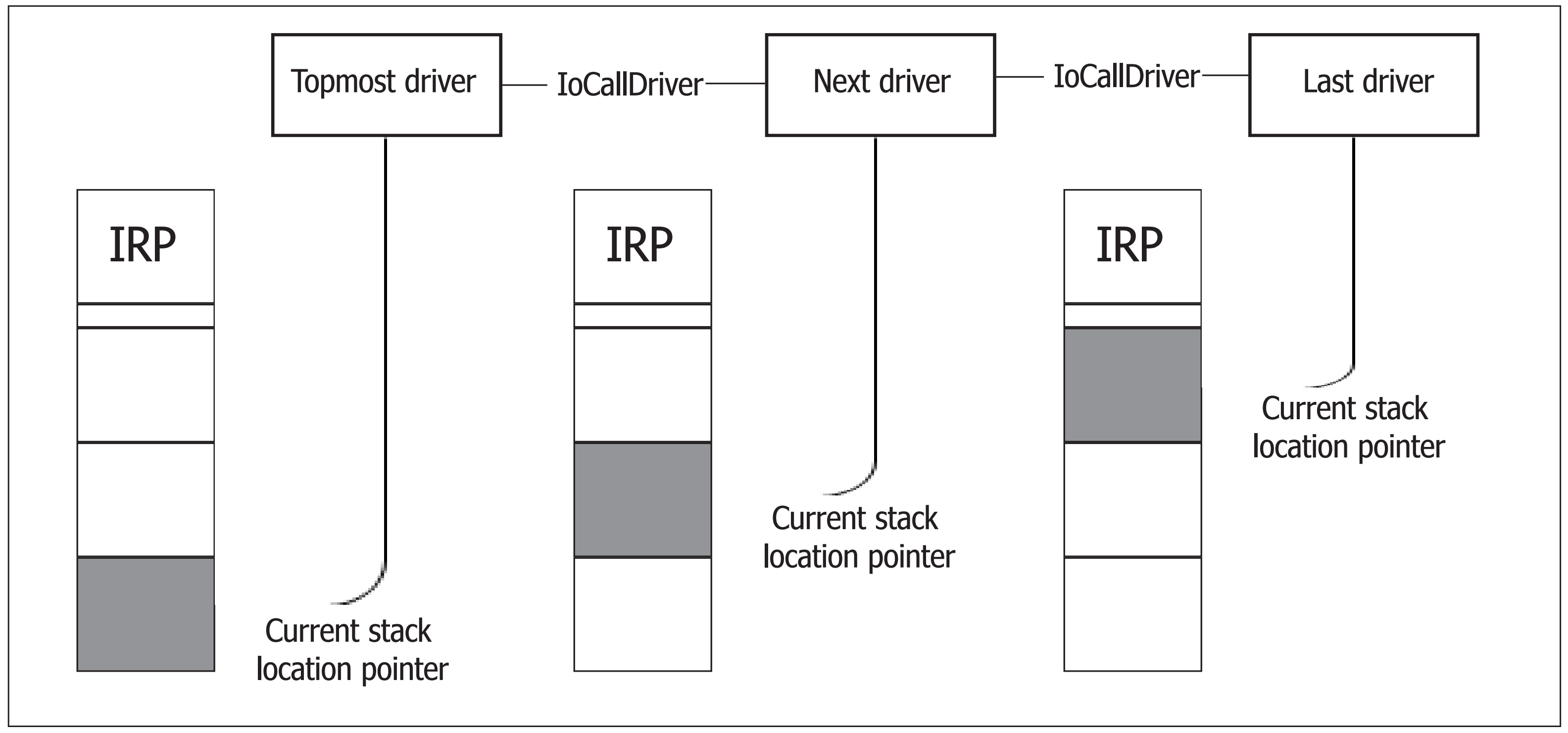
A rootkit that wants to write keystrokes to a file on the local hard disk will use the ZwCreateFile routine to create the file to log keystrokes to and the ZwWriteFile to actually write the keystrokes. By hooking these two functions in the SSDT table and creating a detour between the SSDT index and the original function locations, the parameters, target file, and buffers involved in the creating and writing processes can be analyzed while normal operation continues. One part of this technique is that it analyzes the parameters for suspicious content, such as a target file with the keyword "log" or "report" in it. Another part of this technique involves IRP post processing and buffer comparison. Upon completion of a read request by the keyboard, the scan code in the IRP's buffer is changed to the correct character and temporarily stored in a doubly linked LIST_ENTRY queue. When ZwWriteFile is called, the contents in the buffer that are being written to a file are then placed into another doubly linked LIST_ENTRY queue. Then the queues are compared as illustrated below.


IRP post-processing buffer (LIST_ETNRY queue)
...
FLINK
BLINK
r
o
o
t
k
i
t
ZwWriteFile buffer (LIST_ENTRY queue)

If common strings of characters begin to emerge during the comparison, then that is an indication that the data being read in is being written else where on the hard disk.

I/O Request Packet (IRP) Analyzing Technique

An IRP is composed of two parts; a header and an array of sub-routines that are entries in the I/O stack associated with the IRP. The header contains information needed by the I/O manager and the various drivers that handle the IRP. Each sub-routine in the IRP is meant for one and only one driver on the target driver stack as illustrated below (one-to-one correspondence), so the length of the array of sub-routines can be used to find the driver stack size.


Topmost driver
IoCallDriver
Next driver
IoCallDriver
Last driver
IRP
IRP
IRP
Current stack location pointer
Current stack location pointer
Current stack location pointer

The size of the device stack can be used to check for drivers that don't belong on the stack. For consistency, this is done in two ways. Firstly, the stack size member of the IRP structure is compared with the stack size of the Kbdclass driver plus 1. Secondly, the size of a new IRP with a stack size of the Kbdclass driver plus 1 is compared to the size of the given IRP. If there are discrepancies, then there is a driver above the Kbdclass and below the filter driver.

Results

The filter driver has moderate success dealing with the Klog [3] rootkit. Both techniques work but there are several bugs that need to be taken care of before further use. Memory allocation and queue management problems are causing unwanted bug checks. I will fix these as I continue my work with rootkits.

Acknowledgements

I am grateful for the guidance from Dr. Rajendra K. Raj. I would also like to thank the Computer Science Department and ITS for the lab facilities and the Honors Program for the Undergraduate Summer Research Award grant.

Future Work and Further Information

I plan to focus my attention on detecting suspicious TCP/IP network activity resulting from a layered HID driver rootkit using a special Network Driver Interface Specification (NDIS) filter driver. I also plan to investigate the design and implementation of hardware and firmware rootkits. This will involve the knowledge and use of reverse engineering and binary analysis techniques, as well as a thorough knowledge of the assembly language of the target system.

For further information about this project, contact me at caw4567@rit.edu.

References

[1] Butler, James and Greg Hoglund. *Rootkits: Subverting the Windows Kernel.* New Jersey. Addison Wesley, 2006. http://www.rootkit.com
[2] Fuzen_op. (2005, March 8) *HideProcessHookMdl.zip.* http://www.rootkit.com/vault/fuzen_op/HideProcessHookMDL.zip
[3] Clandestiny. (2004, August 27) *Klog 1.0.zip.* http://www.rootkit.com/vault/Clandestiny/Klog%201.0.zip