UNIX Device Drivers

Nirav Sheth

December 07, 2016

Abstract

Modern era technology landscape has seen a major growth in various hardware devices such as smartphones, video cameras, 3D glasses, smartwatches, wearable technology, etc. Device drivers are software that let these hardware devices connect to the operating system. The growth in hardware devices has increased the demand for device driver developers. However, device drivers are often confused with user applications, which manage the device, or other components of the hardware software interaction chain. This report sheds light on what exactly is a device driver, how the driver fits into the hardware software chain, and how the driver securely becomes part of a UNIX operating system. We find that device driver is a software interface for the kernel to control a hardware device. We also find that device drivers, being part of the kernel, have the potential of being a major source of security vulnerabilities. The intent of this report is to provide the reader with a working knowledge of UNIX device drivers.

Table of Contents

List of Figures

Introduction

Each physical device that is plugged into a computer needs a device driver. A device driver is a programming interface used to control that device. To understand device drivers and how they fit into the UNIX kernel one must think in layers (Figure 1). At the very top of the layer is the hardware controller for each physical device. A hardware controller has its own control and status registers (CSR)s which are specific to the device. The CSRs main function is to control the device and provide device status information. The next layer is the device driver, which are modules that are part of or plugged into the kernel. The device driver provides a software interface for the kernel in order to control the device's hardware controller. Device driver maps standard kernel calls such as memory allocation, interrupts, and input/output operations for the device. The kernel in turn provides a standard interface for an application running in user space in order to control the device (Digital Equipment Corporation, 1996).

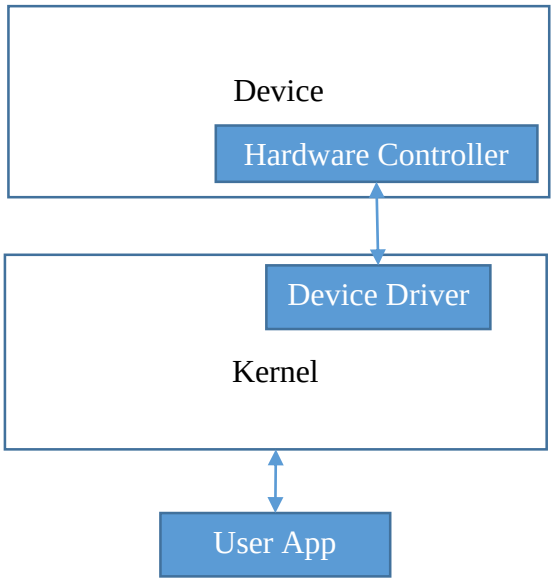

Figure 1: How Device Drivers fit into UNIX OS

Device drivers for standard devices such as hard disks, memory, CPU, network interfaces, etc. are usually complied as part of the kernel. However, device drivers for pluggable devices are built separately from the kernel as modules and plugged in at runtime when needed.

Lastly, efficient and problem free device drivers should strictly contain the mechanism and not the policy nor the intelligence to control the device. Adding policy or intelligence creates the potential for security vulnerability as well as restricts the user's control of the device. In short, device drivers should contain code to interface with the hardware but should not lock down device functionality or restrict user functionality by containing policy or intelligence.

Learning about Device Drivers

Currently, we live in a period of time known as the information age and a key part of the information age is a concept known as Internet of Things (IOT). IOT is a concept where information is collected from numerous physical devices which are interconnected and omnipresent in our daily lives. Physical devices are embedded into various objects such as buildings, clothing, accessories, etc. New devices are introduced into the market and are becoming obsolete at a high pace and so there is great demand for software developers who know how to write device drivers. Second, device drivers are the key pieces of the hardware software interaction chain and thus provide an opportunity to avoid security issues and improve hardware and software performance. Lastly, since device drivers are part of kernel code and map standard kernel calls they provide a great entry point to learning about the UNIX kernel (Corbet, Rubini, & Kroah-Hartman, 2005).

UNIX Kernel Modules

The UNIX kernel is responsible for five main tasks: process management, memory management, file system management, device control, and networking. The device control sub-system of the kernel is responsible for managing devices that are not related to processor, memory, and motherboard entities.

A key feature of the UNIX kernel is that you can add or remove functionality to the kernel at runtime via modules. Modules are object file that contain code which can be dynamically linked to and unlinked from the running kernel. Modules provide the opportunity to extend the kernel without re-compiling or restarting the kernel. Thus modules provide the perfect vehicle for implementing device drivers. Kernel modules can also be compiled with the kernel. However, having to compile modules as part of the kernel would significantly increase the size and reduce the extensibility of the kernel (Corbet, Rubini, & Kroah-Hartman, 2005).

Types of Device Drivers

There are three general types of device drivers: char device, block device, and network device. The char device provides a stream for performing input and output operation. The char device driver implements functions such as open, close, read, and write. Char device driver instantaneously and directly transfer data between the device and the user process. They are usually implemented by devices for functionality such as video capture, audio capture, serial communication, etc. Examples of a char device are the console, video cards, and sound cards.

The block device provides a buffered mechanism for random access operations. The block device driver implements functions such as buffered read, buffered write, and seek. Rather than exchanging information one byte at a time, as the char devices do, the block device exchanges information in blocks of data. Block devices are typically used for performance reasons since they provide a buffering functionality. Where the char device may run into lag issues when processing information, the block devices offers caching that can be used to increase performance. Common examples of block devices are hard disks and tape drives.

The network device is responsible for interacting with network interfaces. The network device registers itself to a kernel data structure and is responsible for exchanging network packets. Rather than reading and writing data, as the char and block devices do, the network device exchanges packets asynchronously with other network systems and their devices. The network device driver also implements network specific operations such as setting IP address, managing network communication parameters, and monitoring network traffic.

Both the char and the block device must register with the kernel and are represented as a file in the UNIX system located under the /dev directory. The device file represents the device driver using major and minor device numbers. All devices controlled by the same device driver have the same major device number. The device driver then uses the minor device number to distinguish between the different devices it manages. Software calls are made to the device referencing the device file using its major device number and the kernel maps the device file to the device in kernel tables. The network device must also register with the kernel but is not

represented as a file; rather the network device implements kernel calls to request exchange of packets. During registration the network driver registers with a kernel data structure and populates it with a list of each detected network interface (Rusling, 1999).

Kernel Module Example

The following is a simple 'hello world' example (Figure 2) of a kernel module that maps two basic kernel calls which initialize and unload the module code into the kernel.

```
#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

MODULE_LICENSE("GPL");

status = register_chrdev(major_num, "hello",
&hello_fops);

static int hello_init(void) {

   printk(KERN_ALERT, "Hello, world\n");

   return 0;

}

static void hello_exit(void) {

   printk(KERN_ALERT, "Exit\n");

}

module_init(hello_init);

module_exit(hello_exit);
```

Figure 2: hello_mod.c; a simple module

'MOUDLE_LICENSE' function lets the kernel know that this module is developed under an open source licensing. If a proprietary licensing is used, then the kernel will mark the module as "tainted" and will display a warning during module initialization. The 'register_chrdev' function links the device using the major number to a file structure representing a file in the /dev directory. The 'module_init' and 'module_exit' functions map the two procedures which are called during module initialization and exit. Lastly, file open, close, read, and write functions specified in the device file's structure, '&hello_fops' in this case, must also be implemented in this module's source (Molloy, 2015).

Before the module can be loaded into the kernel, the kernel headers must be installed in the operating system. The module source file must then be compiled into a kernel object file and loaded into the kernel using 'insmod' command. The module can also be unloaded from the kernel using the 'rmmod' command. Once the module is properly loaded into the kernel the device driver can be accessed by user applications by performing file IO operations on the device file located in the /dev directory (Calbet, 2006).

Security Considerations

Device drivers run as part of the kernel and have privileged access to the kernel space and the user space. Therefore, security is vital to writing good device drivers. As the first layer of defense, the kernel performs a security check during module initialization in order to verify that the user is authorized to load the module. However, once the device driver is loaded into the kernel any security vulnerability in the module will be a vulnerability in the kernel.

The following best practices should be considered for device driver security. First, the driver has privileged access to system resources. Therefore, the calling user application's privileged access must also be verified in the module. Second, proper memory management techniques should be implemented in order to prevent buffer overflows. Third, memory is provided by the kernel and so it cannot be assumed to be empty and could contain privileged information. Memory provided by the kernel should be properly initialized so as to prevent user calls from accessing any information that may have previously been on it. Fourth, validation checks should be done on all user data in to and from the device driver. Lastly, the UNIX system provides a nuclear option where the kernel can be compiled without module support and thus closing all module related security holes (Corbet, Rubini, & Kroah-Hartman, 2005).

Conclusion

To summarize, device drivers are a key part of the UNIX operating system as they allow various hardware devices to be plugged into the system and thus extend the functionality of the system. Modularized device drivers offer ad-hoc extensibility and prevent having to compile the UNIX kernel with device drivers, for all the hardware devices it may ever need. The three general types char, block, and network device drivers provide comprehensive and efficient methods of software/hardware interaction. Lastly, writing device drivers with security best practices in mind will ensure a secure and resilient UNIX system.

Bibliography

Calbet, X. (2006, April 26). *Writing device drivers in Linux: A brief tutorial.* Retrieved from Free Software Magazine: http://freesoftwaremagazine.com/articles/drivers_linux/

Corbet, J., Rubini, A., & Kroah-Hartman, G. (2005). *Linux Device Drivers, Third Edition.* Sebastopol: O'Reilly Media, Inc.

Digital Equipment Corporation. (1996, March). *Introduction to Device Drivers.* Retrieved from The i-acoma group at UIUC: http://iacoma.cs.uiuc.edu/~nakano/dd/drivertut3.html

Molloy, D. (2015, April 14). *Writing a Linux Kernel Module — Part 1: Introduction.* Retrieved from derekmolloy.ie: http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/

Rusling, D. (1999). *The Linux Kernel.* Wokingham: Rusling, David.