Linux device drivers tutorial for beginners pdf

This is a Series on Linux Device Drivers. The purpose of this series is to provide an easy and practical example so that everyone can understand the concept in a simple way. So let's get into Linux Device Driver Part 1 – Introduction. Before we start with programming, it's always better to know some basic things about Linux and its drivers. This is the introduction of Linux.Linux Device Driver Part 1 - IntroductionLinux - IntroductionLinux is a free open-source operating system (OS) based on UNIX created in 1991 by Linus Torvalds. Users can modify and create variations of source code, known as distributions, for computers and other devices. Linux ArchitectureLinux is mainly divided into User Space & Kernel Space. These two components interact through the System Call Interface – which is a predetermined interface and is embedded into the Linux Kernel for Userspace applications. The image below will give you a basic understanding. SpaceKernel Kernel Space is where the kernel (i.e., the core of the operating system) executes (that is, runs) and provides its services. User SpaceUser Space is where the user's application is run. The Linux KernelKernel module module is a piece of code that can be loaded and unloaded into the kernel on demand. They extend kernel functionality without the need to reboot the system. Custom code can be added to the Linux kernel via two methods. The basic way is to add code to the kernel source tree and reprocess the kernel. A more efficient way is to add code to the kernel as it runs. This process is called loading a module, where the module refers to the code we want to add to the kernel. Since we load this code at processing time and they are not part of the official Linux kernel, this is called a loadable kernel module (MFI), which is different from the basic kernel. The basic kernel is located in the /boot directory and is always loaded when we boot our machine whereas the MFI is loaded after the base kernel is already loaded. Nonetheless, these MFIs are very much part of our kernel and they communicate with the basic kernel to complete its functionality. MFIs can perform a variety of tasks, but basically they are under three main categories, Device DriverFilesystem driverSystem callsDevice device driver device drivers designed for specific hardware. The kernel uses it to communicate with that hardware without having to know the details of how the hardware works. Filesystem driver The file system driver interprets the contents of the file system (which is usually the contents of a disk drive) as files and directories and the like. There are many different ways to store files and directories and such on disk drives, on network servers, and in other ways. For each way, you file system drivers. For example, there is a file system driver for ext2 ext2 the type used is almost universal on Linux disk drives. There is one for the MS-DOS file system as well, and one for NFS. The SystemUserspace call program uses system calls to obtain services from the kernel. For example, there is a system call to read files, to create a new process, and to shut down the system. Most system calls are an integral part of the system and are very standard, so it is always built into the basic kernel (no MFI option). But you can find your own system calls and install them as MFIs. Or you can decide you don't like the way Linux does things and overwrite existing system calls with your own MFIs. The advantage of MFIOne the main advantage they have is that we don't have to keep rebuilding the kernel every time we add a new device or if we upgrade an old device. This saves time and also helps keep our basic kernel error-free. MFIs are very flexible, in the sense that they can be loaded and unloaded with a single command line. This helps in storing memory when we load MFIs only when we need them. The difference between the Kernel Module and the Modulkernel User Program has a separate address space. The module runs in kernel space. The application runs in the user's space. System software is protected from user programs. Kernel space and user space have their own memory address space. Kernel modules have higher execution rights. Code that runs in kernel space has greater privileges than code that runs in the userspace. Kernel modules are not executed sequentially. User programs typically run sequentially and perform one task from start to finish. Kernel modules do not run sequentially. The kernel module registers itself to serve future requests. Kernel modules use different header files. Kernel modules require a different set of header files than the user program requires. The difference between kernel drivers and kernel modules in Kernel ModulesA is that little compiled code can be included in the kernel at run-time, such as withinsmod or modprobe. The driver is a bit of code that runs in the kernel to talk to some hardware. It powers the hardware. Most of the hardware on your computer has an associated driver. Device DriverA device driver is a specific form of application software designed to allow interaction with hardware devices. Without the required device drivers, the appropriate hardware fails to work. Device drivers typically communicate with the hardware through the communication subsystem or bus of the computer on which the hardware is connected. Curated device driver operating system and relies on hardware. The device driver acts as a translator between the hardware and the program or operating system that is using it. TypeIn the traditional classification, there are three types of devices: Device charactersBlock devicesNetwork devicesIn Linux, all are files. I mean Linux Linux everything as a file even hardware. A Char DeviceA character file is a hardware file that reads/writes data in characters according to character mode. Some classic examples are keyboards, mice, serial printers. If a user uses a char file to write data, no other user can use the same char file to write data that blocks access to other users. Character files use Synchronize Technic to write data. From you observe char files are used for communication purposes and they cannot be installed. DeviceA block file block is a hardware file that reads/writes data in blocks, not characters by character. This file type is very useful when we want to write/read data in bulk. All our disks such as HDD, USB, and CDROM are block devices. This is the reason when we format we consider the block size. Data writing is done in an asynchronous way and it is CPU intensive activity. These device files are used to store data on real and installable hardware so that we can access the data we have written. Network DeviceA devices are, as far as the Linux network subsystem is concerned, the entity that sends and receives data packets. This is usually a physical device such as an ethernet card. Some network devices though only software such as loopback devices are used to send data to yourself. It's all about the basics of linux and device drivers. We'll move on to Linux Device Driver Programming in the next tutorial. Author: Hcamael@Knownsec 404 TeamChinese Version: when I learned IoT, due to the lack of devices, the simulation of running firmware will often lack / dev / xxx, so I began to wonder if I could write my own drivers to make the firmware run. No matter how difficult it is and whether it can achieve my original intentions or not, it pays off a lot if you learn how to develop Linux drivers. Introduction The series I wrote is mainy about exercise, which doesn't talk much about theory. I learned how to develop drivers from the Linux Device Drivers book, and there is code for the example described in this book on GitHub [1]. As for the basic concept, Linux systems are divided into kernel mode and user mode. Hardware can only be accessed in kernel mode, and drivers can be considered apIs provided in kernel mode to let user mode code access hardware. With the basic concept in mind, I have come up with a series of problems, which inspired me to learn about the development of the driver. All code learning starts with Hello World, so how do I write the Hello World program? How generate device files under /dev? How do drivers access the actual hardware? How do I get system-based code? Or can reverse the driver without a code? Where are the binaries that store drivers? In the future, there may be opportunities to try to learn security of the drive. Everything Starts from Hello WorldMy Hello World code is as follows [2]:D linuxriver developed by means of C Language, which is different normal form that we use. What we often use is the Libc library, which is not in the kernel. While the driver is a program running in the kernel, we use library functions in the kernel. For example, printk is analogous to printf in Libc, an output function defined in the kernel. But I think it's more like a logger function in Python, because printk output is printed in kernel logs, which can be seen through the dmesg command. There is only one entry point and one exit point in the driver code. Loading the driver into the kernel will run the function specified by the module_init function, which in the code above is a high-hello_init function. When a driver is unloaded from the kernel, the function defined by the function module_exit called, which in the above code is the same hello_exit function. The above code makes it clear that when the driver is loaded, it prints Hello World and when the driver is unloaded, it prints Goodbye World.PS: MODULE_LICENSE and MODULE_AUTHOR don't really matter. I'm not a professional development driver, so there's no need to pay attention to them. PSS: There must be adding a new line to the printk output, otherwise the buffer will not be flushed. The compile of drivers needs to be compiled by the create command, and the Makefile is shown below:In general, the kernel source code is in the source directory /usr/src/linux-headers-$(shell uname -r)/directory, like:And all we need is a compiled source directory, i.e. /usr/src/linux-headers-4.4.0-135-generic/. Driver code header files need to be searched from this directory. The M=$(PWD) parameter indicates that the driver compilation output is in the current directory. Finally, via the command obj-m := hello.o, which means compiling hello.o to hello.ko, and the ko file is a kernel module file. Load Driver into the KernelSome system command that needs to be used:Lsmod: See which kernel modules are currently loaded. Insmod: Loads kernel modules and requires root permissions. Rmmod: Remove module. For example: The old kernel used the above method to load and remove the kernel, but the new version of the Linux kernel added module verification. The current actual situation is as follows:From a security perspective, the kernel currently assumes that the module is untrustworthy and needs to be signed with a trusted certificate to load the module. Two solutions: Enter the BIOS and disable Secure Boot UEFI. Add a self-signed certificate to the kernel and use it to sign the driver module (you can refer to [3]). See ResultsAdd Device Files under /devOnce again, first we provide the code, and then describe the code sample [4]. Knowledge Point 1 — Classification DriverDriver is divided into three categories: character devices, block devices, and network interfaces. The code above is an example of a character device, and the other two will be discussed later. As shown above, brw-rw-- -- the permission bar, device blocks start with b and device characters begin with c. Knowledge Point 2 — The Major and Minor NumbersThe large number is used to distinguish drivers. In general, the same main number indicates that it is controlled by the same driver. Multiple devices can be created in a single drive, distinguished by a minor number. The primary and minor numbers define the driver device together (as shown above). The main number of tools sda and sda1 is 8, and one minor number is 0 and the other minor number is 1.Knowledge Point 3 - How Drivers Provide APIIn my mind, the interface provided by the driver is / dev / xxx, and under Linux, everything is about files, so the operation of the driver device is actually the operation of the file and the driver is used to define / open / read / write ... what /dev/xxx will happen. The API drivers you can think of are all about file operations. What file operations are there? Everything is defined in the file_operations structure of the kernel header file <linux s.h=>[5]. In the code I illustrated above:I declare the structure and define it. Except for the owner, the value of the other member is the function pointer. Then I used cdev_add to register the file operating structure with each driver in this scull_setup_cdev. For example, if I perform an open operation on a driver device, I will execute a scull_openfunction, which is equivalent to associating an open function in a system call. Knowledge Point 4 — Generate the Appropriate Device under /devCompile code above, get scull.ko, then sign it, and finally insert it into the kernel via insmod. Check if it loaded successfully:Although the driver loaded successfully, it does not create device files in the /dev directory. We need to manually use mknod for device linking:SummaryIn this example, there is no operation on the actual physical device, just use kmalloc to file a block of memory in the kernel space. There are no more details about the code, which can be found by searching for header files or Google.Here I want to share how I learned driver development: read the book to understand the basic concept first, and then look for details when you need to use it. For example, I don't need to know what APIs a driver can provide, and all I need to know is that the APIs provided by drivers are all about file operations. As for file operations, at the moment I just need to open, close, read, and write. I'll be looking for many file operations when necessary. ReferenceBeijing Knownsec Information Technology Co., Ltd was founded by a group of high-profile </linux> </linux> security experts. It has over a hundred border security talents nationally as a core security research team to provide advanced long-term international network security solutions to governments and companies. Knownsec's specialties include network attacks and integrated defense and R&D technologies products in new situations. It provides visualization solutions that meet world-class security technology standards and enhance customer network security, alarm, and defense monitoring capabilities with industry-leading capabilities in cloud computing and big data processing. The company's technical strength is highly recognized by the Ministry of State Public Security, Central Government Procurement Center, Ministry of Industry and Information Technology (MIIT), China National Information Security Vulnerability Database (CNNVD), Central Bank, Hong Kong Jockey Club, Microsoft, Zhejiang Satellite TV and other well-known clients.404 Team, Knownsec's core security team, is dedicated to security and offensive vulnerability research, IoT, industrial control, blockchain, etc. Team 404 has sent vulnerability research to many well-known vendors such as Microsoft, Apple, Adobe, Tencent, Alibaba, Baidu, etc. And it has earned a high reputation in the industry. Knownsec 404's most famous team divisions include: KCon Hacking Conference, Seebug Vulnerability Database, and ZoomEye Cyber Search Engine. Machine.