# Kernel

## 编译

下载内核

curl -O -L https://mirrors.tuna.tsinghua.edu.cn/kernel/v5.x/linux-5.4.98.tar.xz

下载公钥

gpg --locate-keys torvalds@kernel.org [gregkh@kernel.org](mailto:gregkh@kernel.org)

解压内核

tar -xf linux-5.4.98.tar

编译配置

make menuconfig

编译内核

make -j3 bzImage

关键文件

bzImage：arch/x86/boot/bzImage

vmlinux：源码所在的根目录下。

编译驱动（待补充）

下载编译 busybox

❯ wget https://busybox.net/downloads/busybox-1.32.1.tar.bz2

❯ tar -jxf busybox-1.32.1.tar.bz2

❯ make menuconfig

make -j3

配置系统

make install

mkdir -p proc sys dev etc/init.d

编写启动脚本init

#!/bin/sh

echo "INIT SCRIPT"

mkdir /tmp

mount -t proc none /proc

mount -t sysfs none /sys

mount -t devtmpfs none /dev

mount -t debugfs none /sys/kernel/debug

mount -t tmpfs none /tmp

insmod /ko\_test.ko

echo -e "Boot took $(cut -d' ' -f1 /proc/uptime) seconds"

setsid /bin/cttyhack setuidgid 1000 /bin/sh

poweroff -f

打包文件系统

find . | cpio -o --format=newc > ../rootfs.img

将打包文件与内核文件放到同一目录

编写启动文件

#!/bin/sh

qemu-system-x86\_64 \

-m 64M \

-nographic \

-kernel ./bzImage \

-initrd ./rootfs.img \

-append "root=/dev/ram rw console=ttyS0 oops=panic panic=1 kaslr" \

-smp cores=2,threads=1 \

-cpu kvm64

### 内核文件介绍

* **bzImage**：目前主流的 kernel 镜像格式，即 big zImage（即 bz 不是指 bzip2），适用于较大的（大于 512 KB） Kernel。这个镜像会被加载到内存的高地址，高于 1MB。bzImage 是用 gzip 压缩的，文件的开头部分有 gzip 解压缩的代码，所以我们不能用 gunzip 来解压缩。
* **zImage**：比较老的 kernel 镜像格式，适用于较小的（不大于 512KB） Kernel。启动时，这个镜像会被加载到内存的低地址，即内存的前 640 KB。zImage 也不能用 gunzip 来解压缩。
* **vmlinuz**：vmlinuz 不仅包含了压缩后的 vmlinux，还包含了 gzip 解压缩的代码。实际上就是 zImage 或者 bzImage 文件。该文件是 bootable 的。 bootable 是指它能够把内核加载到内存中。对于 Linux 系统而言，该文件位于 /boot 目录下。该目录包含了启动系统时所需要的文件。
* **vmlinux**：静态链接的 Linux kernel，以可执行文件的形式存在，尚未经过压缩。该文件往往是在生成 vmlinuz 的过程中产生的。该文件适合于调试。但是该文件不是 bootable 的。
* **vmlinux.bin**：也是静态链接的 Linux kernel，只是以一个可启动的 (bootable) 二进制文件存在。所有的符号信息和重定位信息都被删除了。生成命令为：objcopy -O binary vmlinux vmlinux.bin。
* **uImage**：uImage 是 U-boot 专用的镜像文件，它是在 zImage 之前加上了一个长度为 0x40 的 tag 而构成的。这个 tag 说明了这个镜像文件的类型、加载位置、生成时间、大小等信息。

### 其它

解包文件系统

cpio -idmv < rootfs.img

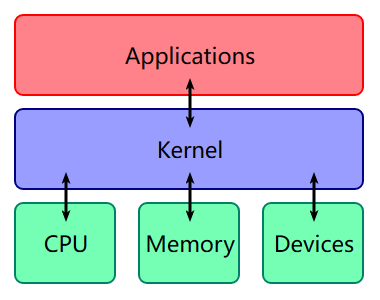
启动时

-monitor /dev/null -m 64M --nographic \

可以通过ctrl+c终止

## 基本

kernel 也是一个程序，用来管理软件发出的数据 I/O 要求，将这些要求转义为指令，交给 CPU 和计算机中的其他组件处理，kernel 是现代操作系统最基本的部分。



kernel 最主要的功能有两点：

1. 控制并与硬件进行交互
2. 提供 application 能运行的环境

包括 I/O，权限控制，系统调用，进程管理，内存管理等多项功能都可以归结到上边两点中。

需要注意的是，**kernel 的 crash 通常会引起重启**。

### 模块相关指令

* **insmod**: 讲指定模块加载到内核中
* **rmmod**: 从内核中卸载指定模块
* **lsmod**: 列出已经加载的模块
* **modprobe**: 添加或删除模块，modprobe 在加载模块时会查找依赖关系

### 内核态函数

相比用户态库函数，内核态的函数有了一些变化

* printf() -> **printk()**，但需要注意的是 printk() 不一定会把内容显示到终端上，但一定在内核缓冲区里，可以通过 dmesg 查看效果
* memcpy() -> **copy\_from\_user()/copy\_to\_user()**
  + copy\_from\_user() 实现了将用户空间的数据传送到内核空间
  + copy\_to\_user() 实现了将内核空间的数据传送到用户空间
* malloc() -> **kmalloc()**，内核态的内存分配函数，和 malloc() 相似，但使用的是 slab/slub 分配器
* free() -> **kfree()**，同 kmalloc()

另外要注意的是，kernel 管理进程，因此 kernel 也记录了进程的权限。kernel 中有两个可以方便的改变权限的函数：

* **int commit\_creds(struct cred \*new)**
* **struct cred\* prepare\_kernel\_cred(struct task\_struct\* daemon)**

从函数名也可以看出，执行 commit\_creds(prepare\_kernel\_cred(0)) 即可获得 root 权限，0 表示 以 0 号进程作为参考准备新的 credentials。

#### 设备文件

**设备号**

对于**所有**设备文件来说，一共分为三种，分别是：

* 字符设备（ char device），例如控制台
* 块设备（block device），例如文件系统
* 网络设备（network device），例如网卡

设备文件可以通过设备文件名来访问，通常位于 /dev 目录下。ls -a 出来的第一个字符即说明了当前设备文件的类型：

|  |  |
| --- | --- |
|  |  |

我们可以在**设备文件条目**中最后一次修改日期之前看到两个数字(用逗号分隔)，例如上面的 5, 0（这个位置通常显示的是普通文件的**文件长度**），对于**设备文件条目**的信息中，形如5,0这样的一对数字，分别是特定设备的**主设备号**和**副设备号**。

在传统意义上，**主设备号**标识与设备相关的**驱动程序**。例如，/dev/null 和 /dev/zero 都是由驱动1管理的。而多个串行终端（即 ttyX, ttySX）是由驱动4管理的。现代的Linux内核已经**支持多个驱动程序共享主设备号**，但是我们仍然可以看到，目前大多数设备仍然是按照**一个主设备号对应一个驱动程序**的方式来组织的。

内核**使用副设备号来确定引用的是哪个设备**，但副设备号的作用仅限于此，内核不会知道更多关于某个特定副设备号的信息。

主设备号和副设备号可同时保存与类型 dev\_t 中，而该类型实际上是一个 u32；其中的12位用于保存主设备号，20位用于保存副设备号。

typedef u32 \_\_kernel\_dev\_t;

typedef \_\_kernel\_dev\_t dev\_t;

在编写驱动程序需要使用主副设备号时，最好不要直接进行位运算操作，而是使用 <linux/kdev\_t.h> 头文件中的宏定义操作：

#define MAJOR(dev) ((dev)>>8) // 获取主设备号

#define MINOR(dev) ((dev) & 0xff) // 获取副设备号

#define MKDEV(ma,mi) ((ma)<<8 | (mi)) // 从主副设备号中生成一个 dev\_t 类型的变量

#### 函数

##### alloc\_chrdev\_region(&babydev\_no, 0, 1, "babydev");

用于分配一个字符设备号

1. **alloc\_chrdev\_region**是一个系统调用，用于在内核中分配一个字符设备号。它的原型如下：

int alloc\_chrdev\_region(dev\_t \*dev, unsigned int firstminor, unsigned int count, const char \*name);

* + **dev**是一个指向**dev\_t**类型的指针，用于存储分配的设备号。在这里，设备号将被保存在**babydev\_no**变量中。
  + **firstminor**是第一个次设备号，通常为0。
  + **count**是要分配的设备号数量，这里为1。
  + **name**是设备的名称字符串，这里为"babydev"。

1. 函数返回一个整数值，表示分配设备号的结果。如果分配成功，返回值为0或正数；否则返回一个负数错误代码。

##### cdev\_init(&cdev\_0, &fops);

1. **cdev\_init**是一个内核函数，用于对字符设备结构体进行初始化，以便后续的字符设备注册和操作。
2. 函数原型如下：

void cdev\_init(struct cdev \*cdev, const struct file\_operations \*fops);

* + **cdev**是一个指向要初始化的字符设备结构体的指针。在这里，指针**&cdev\_0**表示对名为**cdev\_0**的字符设备结构体进行初始化。
  + **fops**是一个指向定义字符设备操作的结构体**file\_operations**的指针。这个结构体定义了字符设备支持的各种操作函数，例如**open**、**read**、**write**等。在这里，指针**&fops**表示使用名为**fops**的操作函数集合。

1. 该函数没有返回值，只是对传入的字符设备结构体进行了初始化。

#### Cred

struct cred {

atomic\_t usage;

#ifdef CONFIG\_DEBUG\_CREDENTIALS

atomic\_t subscribers; /\* number of processes subscribed \*/

void \*put\_addr;

unsigned magic;

#define CRED\_MAGIC 0x43736564

#define CRED\_MAGIC\_DEAD 0x44656144

#endif

kuid\_t uid; /\* real UID of the task \*/

kgid\_t gid; /\* real GID of the task \*/

kuid\_t suid; /\* saved UID of the task \*/

kgid\_t sgid; /\* saved GID of the task \*/

kuid\_t euid; /\* effective UID of the task \*/

kgid\_t egid; /\* effective GID of the task \*/

kuid\_t fsuid; /\* UID for VFS ops \*/

kgid\_t fsgid; /\* GID for VFS ops \*/

...

}

内核会通过进程的 task\_struct 结构体中的 cred 指针来索引 cred 结构体，然后根据 cred 的内容来判断一个进程拥有的权限，如果 cred 结构体成员中的 uid-fsgid 都为 0，那一般就会认为进程具有 root 权限。

## 一般流程

### 解压

mkdir babydriver

tar -xf babydriver.tar -C babydriver

### 启动

cd babydriver

./boot.sh

查看一下根目录的 /init 文件

### 解包

先创建rootfs文件夹然后cd,复制文件进去

$ file rootfs.cpio

rootfs.cpio: gzip compressed data, last modified: Tue Jul 4 08:39:15 2017, max compression, from Unix, original size modulo 2^32 2844672

mv rootfs.cpio rootfs.cpio.gz

gunzip rootfs.cpio.gz

$ file rootfs.cpio

rootfs.cpio: ASCII cpio archive (SVR4 with no CRC)

cpio -idmv < rootfs.cpio

在/lib/modules/4.4.72/babydriver.ko下找到目标驱动。

**checksec babydriver.ko**

### 解答

#!/bin/bash

# 判断当前权限是否为 root，需要高权限以执行 gef-remote --qemu-mode

user=$(env | grep "^USER" | cut -d "=" -f 2)

if [ "$user" != "root" ]

then

echo "请使用 root 权限执行"

exit

fi

# 静态编译 exp

gcc exp.c -static -o rootfs/exp

# rootfs 打包

pushd rootfs

find . | cpio -o --format=newc > ../rootfs.cpio

popd

# 启动 gdb

gnome-terminal -e 'gdb -x mygdbinit'

# 启动 qemu

qemu-system-x86\_64 \

-initrd rootfs.cpio \

-kernel bzImage \

-append 'console=ttyS0 root=/dev/ram oops=panic panic=1' \

-enable-kvm \

-monitor /dev/null \

-m 64M \

--nographic \

-smp cores=1,threads=1 \

-cpu kvm64,+smep \

-s

### 调试

vmlinux-to-elf bzImage vmlinux

解压出来的 vmlinux 就是带符号的，可以正常被 gdb 读取和下断点

gdb 中使用 add-symbol-file 加载 ko 的符号

set architecture i386:x86-64

add-symbol-file vmlinux

gef-remote --qemu-mode localhost:1234

c

mygdbinit

# 先 continue， 在 insmod 之后手动 Ctrl+C 再设置断点，免得断点处于 pending 状态

add-symbol-file babydriver.ko 0xffffffffc0000000

b babyread

b babywrite

b babyioctl

b babyopen

b babyrelease

c