РЕАЛИЗАЦИЯ ПРОГРАММНОГО СРЕДСТВА ШИФРОВАНИЯ ДАННЫХ ЖЁСТКОГО ДИСКА В СИСТЕМЕ GNU/LINUX

А.Э. ТИМОФЕЕВ

Пермский государственный национальный исследовательский университет, 614990, Пермь, Букирева, 15

Информация - одна из самых ценных вещей в современной жизни. И с развитием мобильной техники угроза кражи и потери носителя информации стала более актуальной.

В GNU/Linux системе прозрачное шифрование легко реализуется с помощью модуля ядра блочного устройства.

Модуль ядра, загружаемый модуль ядра (англ. loadable kernel module, LKM) — объект, содержащий код, который расширяет функциональность запущенного или т. н. базового ядра ОС.[1]

Для создания модуля необходимы заголовочные файлы ядра обычно находятся в директории /usr/src/linux или можно скачать[2].

Модуль выводящий сообщения при загрузке и выгрузке:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE( "GPL" );
static int __init md_init( void )
{
  printk( "+ module md start!\n" );
  return 0;
}
static void __exit md_exit( void )
{
  printk( "+ module md unloaded!\n" );
}
module_init( md_init );
module_exit( md_exit );
```

Типовой сценарий сборки:

```
CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)
TARGET1 = md1
TARGET2 = md2
TARGET3 = md3
obj-m   := $(TARGET1).o $(TARGET2).o $(TARGET3).o
default:
```

```
	$(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
	@rm -f *.o .*.cmd .*.flags *.mod.c *.order
	@rm -f .*.*.cmd *~ *.*~ TODO.*
	@rm -fR .tmp*
	@rm -rf .tmp_versions
disclean: clean
	@rm *.ko *.symvers
```

После компиляции можно проверять модуль командами insmod, rmmod.

Первым шагом, выполняемым большинством блочных драйверов является регистрация себя в ядре. Функцией для выполнения этой задачи является register_blkdev (которая объявлена в <linux/fs.h>):

int register_blkdev(unsigned int major, const char *name);

Аргументами являются старший номер, который будет использовать ваше устройство, и связанное с ним имя (которое ядро будет показывать в /proc/devices). Если major передаётся как 0, ядро выделяет новый старший номер и возвращает его вызвавшему.

Соответствующая функция для отмены регистрации блочного драйвера:
int unregister_blkdev(unsigned int major, const char *name);

Здесь аргументы должны совпадать с переданными в register_blkdev или эта функция вернёт -EINVAL и ничего не отменит.

Структура описывающая действия с блочным устройством - struct block_device_operations, которая объявлена в <linux/fs.h>. Которая содержит операции обслуживания устройства:

```
int (*open)(struct inode *inode, struct file *filp);
int (*release)(struct inode *inode, struct file *filp);
int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd, un-
signed long arg);
int (*media_changed) (struct gendisk *gd);
int (*revalidate_disk) (struct gendisk *gd);
struct module *owner;
```

struct gendisk (объявленная в <linux/genhd.h>) является представлением в ядре отдельного дискового устройства.. В struct gendisk есть несколько полей, которые должны быть проинициализированы блочным драйвером:

- int major;
- int first_minor;
- int minors - поле, которые описывает номер устройства, используемого диском. Как минимум, накопитель должен использовать по крайней мере один младший номер.

- char disk_name[32] - поле, которое должно содержать имя дискового устройства. Оно появляется в /proc/partitions и sysfs.
- struct block_device_operations *fops;
- struct request_queue *queue;
- Структура, используемая ядром для управления запросами ввода/вывода для этого устройства;
- int flags - (Редко используемый) набор флагов, описывающий состояние привода.
- sector_t capacity - ёмкость диска, в секторах по 512 байт. Тип sector_t может быть размером в 64 бит. Драйвер не должен устанавливать это поле напрямую; вместо этого число секторов передаётся в set_capacity.
- void *private_data - Блочные драйверы могут использовать это поле для указания на свои собственные внутренние данные.

struct gendisk представляет собой динамически создаваемую структуру, которая требует специальной манипуляции в ядре для инициализации; драйверы не могут создавать эту структуру сами. Вместо этого, вы должны вызвать: struct gendisk *alloc_disk(int minors);

Аргумент minors должен быть числом младших номеров, используемых для этого диска.

Когда диск больше не нужен, он должен быть освобождён с помощью: void del_gendisk(struct gendisk *gd);

Выделенная структура gendisk не делает диск доступным системе. Для инициализации структуры необходимо вызвать add_disk: void add_disk(struct gendisk *gd);

Имейте в виду одну важную вещь: как только вызывается add_disk, диск становится "живым" и его методы могут быть вызваны в любое время. Фактически, первые такие вызовы произойдут, вероятно, ещё до того, как произойдёт возврат из add_disk; ядро будет читать первые несколько блоков в попытке найти таблицу разделов.

Пример инициализация устройства:

```
static int __init SDevice_init(void)
{
	SDevice_major = register_blkdev(SDevice_major, Device_name);
	if (SDevice_major <= 0) {
		return -EBUSY;
	}
	Devices = kmalloc(sizeof(struct SDevice), GFP_ATOMIC);
	memset (Devices, 0, sizeof (struct SDevice));
```

```
        Devices->Queue = blk_alloc_queue(GFP_ATOMIC);
        blk_queue_make_request(Devices->Queue, SDevice_make_request);
        blk_queue_logical_block_size(Devices->Queue, SECTOR_SIZE);
        Devices->Queue->queuedata = Devices;
        Devices->gd = alloc_disk(SDEVICE_MINORS);
        Devices->gd->major = SDevice_major;
        Devices->gd->first_minor = (Devices-
>Device_number)*SDEVICE_MINORS;
        Devices->gd->fops = &SDevice_ops;
        Devices->gd->queue = Devices->Queue;
        Devices->gd->private_data = Devices;
        snprintf (Devices->gd->disk_name, 32, Device_name);
        set_capacity(Devices->gd, 1);
        add_disk(Devices->gd);
}
```

Функция обработки I/O запросов

```
Static SDevice_make_request (struct request_queue *q, struct bio *bio)
{
        struct SDevice _dev *dev = q->queuedata;
        int status;
        status = SDevice _xfer_bio(dev, bio);
        bio_endio(bio, status);
        return 0;
}
static int SDevice _xfer_bio(struct SDevice _dev *dev, struct bio *bio)
{
        int i;
        struct bio_vec *bvec;
        sector_t sector = bio->bi_sector;
        /* Do each segment independently. */
        bio_for_each_segment(bvec, bio, i) {
                char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
                SDevice _transfer(dev, sector, bio_cur_sectors(bio),
                buffer, bio_data_dir(bio) == WRITE);
                sector += bio_cur_sectors(bio);
        __bio_kunmap_atomic(bio, KM_USER0);
        }
        return 0; /* Always "succeed" */
}

static void SDevice _transfer(struct SDevice _dev *dev, unsigned long sector,
 unsigned long nsect, char *buffer, int write)
```

```
{
unsigned long offset = sector*KERNEL_SECTOR_SIZE;
unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;

if (write)
        memcpy(dev->data + offset, buffer, nbytes);
else
       memcpy(buffer, dev->data + offset, nbytes);
}
```

Для шифрования достаточно заменить копирование массивов на криптографическое преобразование.

Приведённый пример работает только для RAM устройств, для реальных устройств необходима ещё одна очередь для работы с реальным устройством.

СПИСОК ЛИТЕРАТУРЫ

1. http://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C_%D1%8F%D0%B4%D1%80%D0%B0
2. https://www.kernel.org/
3. http://www.ibm.com/developerworks/ru/library/l-linux_kernel_01
4. Third Edition of *Linux Device Drivers*, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman
5. Роберт Лав, "Linux. Системное программирование"