

# Linux内核 Lab04 - 文件系统

## 实验内容

以Linux内核中/fs/romfs作为文件系统源码修改的基础，实现以下功能：

romfs.ko 模块接受三种参数：hided\_file\_name, encrypted\_file\_name, 和 exec\_file\_name。

- hided\_file\_name=xxx，隐藏名字为xxx的文件/路径。
- encrypted\_file\_name=xxx，加密名字为xxx的文件中的内容。
- exec\_file\_name，修改名字为xxx的文件的权限为可执行。

上述功能通过生成并挂载一个格式为romfs的镜像文件test.img来检查，镜像文件可通过genromfs来生成。

## 实验思路与实现过程

### 0. romfs 文件系统

#### 1. 文件系统结构

在linux官方文档 [1] 中较详细介绍了romfs文件系统模块(源代码位于/fs/romfs中)的结构：

The layout of the filesystem is the following:

offset	content
0	-   r   o   m   \
+----+----+----+----+ The ASCII representation of those bytes	
4	1   f   s   -   / (i.e. "-rom1fs-")
+----+----+----+----+	
8	full size   The number of accessible bytes in this fs.
+----+----+----+----+	
12	checksum   The checksum of the FIRST 512 BYTES.
+----+----+----+----+	
16	volume name   The zero terminated name of the volume,
:	: padded to 16 byte boundary.
+----+----+----+----+	
xx	file
:	: headers :

The following bytes are now part of the file system; each file header must begin on a 16 byte boundary.

offset	content
+----+----+----+----+	

0	next filehdr x	The offset of the next file header (zero if no more files)
4	spec.info	Info for directories/hard links/devices
8	size	The size of this file in bytes
12	checksum	Covering the meta data, including the file name, and padding
16	file name	The zero terminated name of the file, : padded to 16 byte boundary
xx	file data	

## 2. 文件系统实现源码

在romfs源代码文件中一共有3个核心文件 internal.h super.c, storage.c;

```
root@ecs-youngster /u/s/1/f/romfs# ls
internal.h  mmap-nommu.c  romfs.ko      romfs.mod.o  storage.o
kconfig    modules.order  romfs.mod     romfs.o      super.c
Makefile    Module.symvers  romfs.mod.c   storage.c     super.o
```

其中internal.h是头文件 storage.c中是对读取文件系统内容接口的实现, 包括内容读取, 内容长度获取和内容比较, super.c是这个模块的核心文件, 其实现了romfs文件系统对应linux内核中文件系统描述的若干接口, 需要修改的函数都在这里面;

另外在/include/uapi/linux文件夹下还有一个头文件 romfs\_fs.h, 定义了一些参数的值(包括文件头的字节大小)

## 1. 实验思路以及实现过程

### 0. 单独编译内核模块

如果每次都编译整个操作系统内核, 则效率非常低下; 下述命令可以实现单独编译romfs这个内核模块:

```
sudo make CONFIG_FUSE_FS=m -C /xxx/linux-5.x.x M=/xxx/linux-5.x.x/fs/romfs
modules
```

(两个路径中, 前者为内核的目录, 后者为romfs模块的目录)

### 1. 隐藏文件

romfs会在挂载的时候读取文件系统中所有文件的信息, 核心步骤位于以下函数的一个for循环中

```
static int romfs_readdir(struct file *file, struct dir_context *ctx)
{
    ...
    for (;;) {
        ...
    }
}
```

我们只需要在这个循环中加入对文件名的判断

```
if (hide_file_name != NULL)
    ret = romfs_dev_strcmp(i->i_sb, offset + ROMFH_SIZE, hide_file_name, j);
```

然后在文件名匹配到了想要隐藏的文件名时，跳过将其加载即可：

```
if (strcmp(fsname, hide_file_name) != 0)
{
    if (!dir_emit(ctx, fsname, j, ino,
        romfs_dtype_table[nextfh & ROMFH_TYPE]))
        goto out;
}
```

以下是实验效果(要隐藏的文件命名为file\_hide):

genromfs 命令指定文件夹(图中为./test)作为根目录连同内容一起生成一个romfs文件系统(图中为test.img)

挂载到linux的文件系统中，通过ls命令可以发现，挂载的文件系统中的 "file\_hide"文件不见了

```

root@ecs-youngster:/usr/src/linux-5.6.7# fish
Welcome to fish, the friendly interactive shell
root@ecs-youngster /u/s/linux-5.6.7# cd /home/sjtu-project/lib04/
root@ecs-youngster /h/s/lib04# ls
a.out* test/ test.img
root@ecs-youngster /h/s/lib04# rm a.out
root@ecs-youngster /h/s/lib04# genromfs -V "vromfs" -f test.img -d ./test
root@ecs-youngster /h/s/lib04# ls ./test
file encrypted file exec* file hide file unexec hello world.cpp
root@ecs-youngster /h/s/lib04# mount test.img /mnt -o loop -o noexec
root@ecs-youngster /h/s/lib04# ls /mnt/
file encrypted file exec* file unexec hello world.cpp
root@ecs-youngster /h/s/lib04#

```

```

root@ecs-youngster:/usr/src/linux-5.6.7# fish
Welcome to fish, the friendly interactive shell
root@ecs-youngster /u/s/linux-5.6.7# cd /usr/src/linux-5.6.7/fs/romfs/
root@ecs-youngster /u/s/l/f/romfs#
insmod romfs.ko hide_file_name="file_hide" encrypted_file_name="file_encrypted" exec_file_name="file_exec"
root@ecs-youngster /u/s/l/f/romfs#

```

## 2. 加密指定文件名的文件

linux在读取某个文件时，会将文件系统的内容拷贝到内存页中，相应函数接口为：

```

static int romfs_readpage(struct file *file, struct page *page)
{
    ...
}

```

在读取完成之后，我们通过比较文件名并加密匹配的文件来判断是否加密；

但在这一步与上一个功能稍有不同的地方在于，读取文件的offset参数变量需要自己计算，

源代码中已经获得了管理索引节点inode数据结构的指针，而romfs在 "internal.h"文件中提供了接口ROMFS\_I(...)来使得我们能够获取到管理romfs节点信息的数据结构 romfs\_inode\_info：

```

struct romfs_inode_info {
    struct inode    vfs_inode;
    unsigned long   i_metasize; /* size of non-data area */
    unsigned long   i_dataoffset; /* from the start of fs */
};

```

而 "romfs\_fs.h" 中的宏定义ROMFH\_SIZE 则可以使我们计算出头文件头在文件系统的位置并借此读取其中的文件名，在拿到文件名后，判断是否对文件内容加密，相关代码如下：

```

static int romfs_readpage(struct file *file, struct page *page)
{
    ...
    /* 获取文件名长度 j */
    j = romfs_dev_strnlen(inode->i_sb,
        ROMFS_I(inode)->i_dataoffset - ROMFS_I(inode)->i_metasize +
ROMFH_SIZE,
        ROMFS_MAXFN);

    if (j < 0)
    {
        printk("problem in romfs_dev_strnlen\n");
        return 0;
    }

    /* 获取文件名 到fsname 中 */
    ret_1 = romfs_dev_read(inode->i_sb,
        ROMFS_I(inode)->i_dataoffset - ROMFS_I(inode)->i_metasize +
ROMFH_SIZE,
        fsname, j);
    if (ret_1 < 0)
    {
        printk("problem in romfs_dev_read\n");
        return 0;
    }
    fsname[j] = '\0';
    printk("fsname = %s\n", fsname);
    printk("fsname_encrypted = %s\n", encrypted_file_name);

    /* 如果文件名相同，则对文件内容进行加密，fillsize 为之前计算出的文件内容长度 */
    printk("%d\n", ret_1);
    if(strcmp(fsname, encrypted_file_name) == 0)
    {
        printk("file is encrypted, length is %d\n", j);
        encrypt_file(buf, fillsize);
    }
    ...
}

```

其中加密函数为简单地加一:

```

static void encrypt_file(char* buf, unsigned long size)
{
    int i;
    for (i = 0; i < size; ++i)
    {
        buf[i] += 1;
    }
}

```

以下是实验效果:

```
PROBLEMS  TERMINAL  ...  1: fish, fish
root@ecs-youngster:/usr/src/linux-5.6.7# fish
Welcome to fish, the friendly interactive shell
root@ecs-youngster /u/s/linux-5.6.7# cd /home/sjtu-project/lib04/
root@ecs-youngster /h/s/lib04# ls
a.out* test/ test.img
root@ecs-youngster /h/s/lib04# rm a.out
root@ecs-youngster /h/s/lib04# genromfs -v "vromfs" -f test.img -d ./test
root@ecs-youngster /h/s/lib04# ls ./test
file_encrypted file_exec* file_hide file_unexec hello_world.cpp
root@ecs-youngster /h/s/lib04# mount test.img /mnt -o loop -o noexec
root@ecs-youngster /h/s/lib04# ls /mnt/
file encrypted file_exec* file_unexec hello_world.cpp
root@ecs-youngster /h/s/lib04# cat ./test/file_encrypted
This file should be encrypted!
root@ecs-youngster /h/s/lib04# cat /mnt/file_encrypted
Uijt!gjmfl!tipvmelcf!fodszqufe"
root@ecs-youngster /h/s/lib04#

root@ecs-youngster:/usr/src/linux-5.6.7# fish
Welcome to fish, the friendly interactive shell
root@ecs-youngster /u/s/linux-5.6.7# cd /usr/src/linux-5.6.7/fs/romfs/
root@ecs-youngster /u/s/l/f/romfs#
insmod romfs.ko hide_file_name="file_hide" encrypted_file_name="file_encrypted" exec_f
ile_name="file_exec"
root@ecs-youngster /u/s/l/f/romfs#
```

### 3. 修改可执行权限

当有访问文件的相关操作执行的时候，会先根据路径找到这个文件并获取相应的inode数据结构，核心函数为romfs\_lookup:

```
static struct dentry *romfs_lookup(struct inode *dir, struct dentry *dentry,
                                   unsigned int flags)
{
    ...
    /* 获取文件对应的inode数据结构 */
    inode = romfs_iget(dir->i_sb, offset);
}
```

inode数据结构定义在 /include/linux/fs.h 中:

```

/*
 * keep mostly read-only and often accessed (especially for
 * the RCU path lookup and 'stat' data) fields at the beginning
 * of the 'struct inode'
 */
struct inode {
    umode_t      i_mode;
    unsigned short i_opflags;
    kuid_t      i_uid;
    kgid_t      i_gid;
    unsigned int  i_flags;
    ...
}

```

在另一个官方文档[2]中列出了各个成员的含义，其中i\_mode中包含我们要修改的权限：

The `i_mode` value is a combination of the following flags:

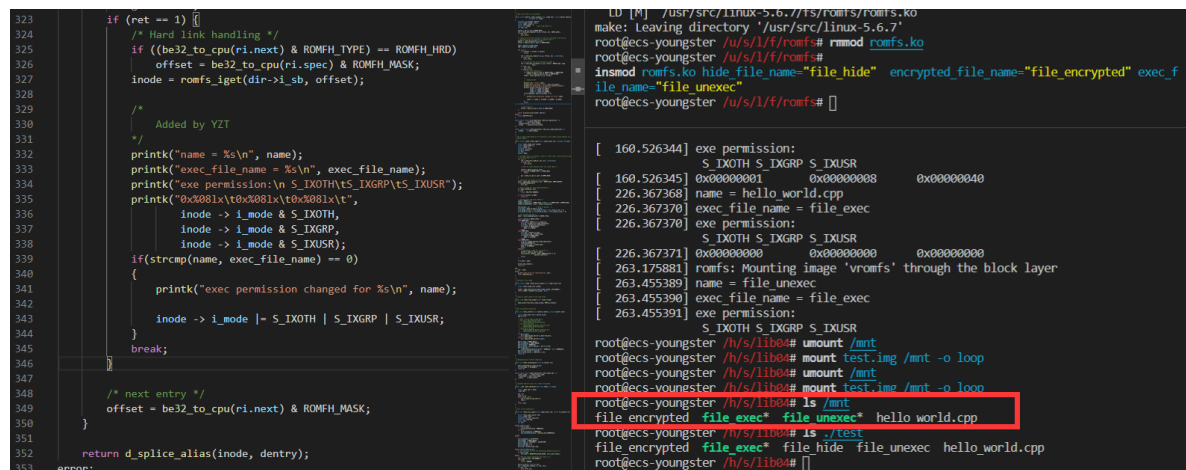
Value	Description
<b>0x1</b>	<b>S_IXOTH (Others may execute)</b>
0x2	S_IWOTH (Others may write)
0x4	S_IROTH (Others may read)
<b>0x8</b>	<b>S_IXGRP (Group members may execute)</b>
0x10	S_IWGRP (Group members may write)
0x20	S_IRGRP (Group members may read)
<b>0x40</b>	<b>S_IXUSR (Owner may execute)</b>
0x80	S_IWUSR (Owner may write)
0x100	S_IRUSR (Owner may read)
0x200	S_ISVTX (Sticky bit)
0x400	S_ISGID (Set GID)
0x800	S_ISUID (Set UID)
	These are mutually-exclusive file types:
0x1000	S_IFIFO (FIFO)
0x2000	S_IFCHR (Character device)
0x4000	S_IFDIR (Directory)
0x6000	S_IFBLK (Block device)
0x8000	S_IFREG (Regular file)
0xA000	S_IFLNK (Symbolic link)
0xC000	S_IFSOCK (Socket)

所有我们只要通过位或运算修改inode里面修改可执行权限的内容即可:

```
if(strcmp(name, exec_file_name) == 0)
{
    inode -> i_mode |= S_IXOTH | S_IXGRP | S_IXUSR;
}
```

实验效果如下:

设置exec\_file\_name="file\_unexec"一个不可执行的文件file\_unexec在被载入后变为了可执行 (fish shell 将可执行文件渲染成绿色)



```
LD [M] /usr/src/linux-5.6.7/fs/romfs/romfs.ko
make: Leaving directory '/usr/src/linux-5.6.7'
root@ecs-youngster /u/s/l/t/romfs# rmmod romfs.ko
root@ecs-youngster /u/s/l/t/romfs# insmod romfs.ko hide file_name="file_hide" encrypted_file_name="file_encrypted" exec_file_name="file_unexec"
root@ecs-youngster /u/s/l/t/romfs# [

[ 160.526344] exe permission:
[ 160.526345] S_IXOTH S_IXGRP S_IXUSR
[ 226.367368] 0x00000001 0x00000008 0x00000040
[ 226.367370] name = hello_world.cpp
[ 226.367371] exec file_name = file_exec
[ 226.367372] exe permission:
[ 226.367373] S_IXOTH S_IXGRP S_IXUSR
[ 226.367374] 0x00000000 0x00000000 0x00000000
[ 263.175881] romfs: Mounting image 'vromfs' through the block layer
[ 263.455389] name = file_unexec
[ 263.455390] exec file_name = file_exec
[ 263.455391] exe permission:
[ 263.455392] S_IXOTH S_IXGRP S_IXUSR
root@ecs-youngster /h/s/lib04# umount /mnt
root@ecs-youngster /h/s/lib04# mount test.img /mnt -o loop
root@ecs-youngster /h/s/lib04# umount /mnt
root@ecs-youngster /h/s/lib04# mount test.img /mnt -o loop
root@ecs-youngster /h/s/lib04# ls /mnt
file encrypted file exec* file unexec* hello_world.cpp
root@ecs-youngster /h/s/lib04# ls ./test
file encrypted file exec* file hide file_unexec hello_world.cpp
root@ecs-youngster /h/s/lib04# [
```

## 实验体会

linux 虚拟文件系统在抽象层定义了统一的接口, 而具体如何实现则由具体的文件系统自己去设计, 这是比较精妙的设计; 在这个实验中, 较为具体地体会到了这一点

## 参考资料

[1] romfs文件系统

<https://www.kernel.org/doc/Documentation/filesystems/romfs.txt>

[2] inode 数据结构 (包括读写、可执行权限等bit位的具体位置)

<https://www.kernel.org/doc/html/latest/filesystems/ext4/dynamic.html#i-mode>



