Lab 4 : File System

实 验 报 告

刘子涵　518021910690

【1】实验要求

以 Linux 内核中的 `/fs/romfs` 作为文件系统源码修改的基础，实现功能：`romfs.ko` 模块接受 3 种参数：

- `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 的文件内容
- `exec_file_name=xxx`：修改名字为 xxx 的文件权限为可执行

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。

【2】实验环境

- 实验平台：华为云弹性云服务器（2vCPUs | 4GiB | kc1.large.2 | Ubuntu 18.04 server 64bit with ARM）
- Linux 内核版本：5.5.11
- GCC 版本：7.4.0

【3】实验思路及具体实现

本次实验需要修改 `linux-5.5.11/fs/romfs` 下的文件系统源码 `super.c`，修改后重新编译生成 `romfs.ko` 模块并加载。实现本实验的功能需要设置 3 个字符串参数，在 `super.c` 文件开始处添加如下代码：

```
// 模块参数声明
#include <linux/moduleparam.h>

static char *hided_file_name;
static char *encrypted_file_name;
static char *exec_file_name;

module_param(hided_file_name, charp, 0644);
module_param(encrypted_file_name, charp, 0644);
module_param(exec_file_name, charp, 0644);
```

1. 隐藏文件

阅读 `super.c`，可以发现 romfs 在 `romfs_readdir()` 函数中读入一个目录的文件到内核空间，如果想要隐藏某个文件，只需要在读入相应目录时跳过该文件。在该函数中，对目录下的每个文件，会使用 `romfs_dev_strnlen()` 读入文件名长度，使用 `romfs_dev_read()` 读入文件名 `fsname`。在此之后，该函数会调用 `dir_emit()` 函数读入文件到内核，需要在此之前检查文件名 `fsname` 是否为 `hided_file_name`，如果是则直接跳过 `dir_emit()`，继续读下一个文件。具体实现如下：

```
        fsname[j] = '\0';  // 得到当前文件名 fsname

        ino = offset;
        nextfh = be32_to_cpu(ri.next);

        // 如果当前文件名 fsname 和 hided_file_name 一致，则跳过 dir_emit() 函数（跳转到 skip ）
        if (hided_file_name && !strcmp(hided_file_name, fsname))
            goto skip;

        if ((nextfh & ROMFH_TYPE) == ROMFH_HRD)
            ino = be32_to_cpu(ri.spec);
        if (!dir_emit(ctx, fsname, j, ino,
                romfs_dtype_table[nextfh & ROMFH_TYPE]))
            goto out;
// 跳转到此处
skip:
        offset = nextfh & ROMFH_MASK;
    }
```

2. 加密文件

阅读 `super.c`，可以发现 romfs 在 `romfs_readpage()` 函数中读取文件内容。首先需要获取当前文件名，然后才能对指定文件存放内容的缓冲区进行修改。该函数首先得到指向索引节点的指针 `struct inode *inode`，之后调用 `romfs_dev_read()` 函数将文件内容读到缓冲区 `buf` 里。在此之后，便可以检查当前文件名并对文件缓冲区进行修改。我添加了两个函数以实现上述功能：

- **`is_encrypted_file()`：判断当前文件名是否为 `encrypted_file_name`**

  获取当前文件名的实现可以参考 `romfs_readdir()` 函数，难点是得到文件名的偏移量 `offset`。阅读 `linux-5.5.11/fs/romfs/internal.h` 中定义的数据结构 `romfs_inode_info` 发现，可以通过函数 `ROMFS_I` 将 `inode` 指针转换为 `romfs_inode_info` 指针，然后偏移量即为 `i_dataoffset` 减去 `i_metasize`。

  ```
  // linux-5.5.11/fs/romfs/internal.h

  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 */
  };

  static inline struct romfs_inode_info *ROMFS_I(struct inode *inode)
  {
      return container_of(inode, struct romfs_inode_info, vfs_inode);
  }
  ```

由此，可以得到函数 `is_encrypted_file()` 实现如下：

```
/*
 * is_encrypted_file -- whether the file is to be encrypted
 */
static bool is_encrypted_file(struct inode *i) {
    int j, ret;
    unsigned long offset;
    struct romfs_inode_info *inode;
    char fsname[ROMFS_MAXFN];

    inode = ROMFS_I(i);
    offset = (inode->i_dataoffset) - (inode->i_metasize);

    j = romfs_dev_strnlen(i->i_sb, offset + ROMFH_SIZE, sizeof(fsname) - 1);
    if (j < 0)
        return false;

    ret = romfs_dev_read(i->i_sb, offset + ROMFH_SIZE, fsname, j);
    if (ret < 0)
        return false;

    fsname[j] = '\0';

    if (encrypted_file_name && !strcmp(encrypted_file_name, fsname))
        return true;
    else
        return false;
}
```

- `encrypt()`：**对缓冲区进行加密（对每个字符 + 1），需要传入缓冲区指针 `buf` 和缓冲区大小 `fillsize`**

```
/*
 * encrypt -- encrypt the buffer by add 1 to each character
 */
static void encrypt(char *buf, int fillsize) {
    int i;
    for (i = 0; i < fillsize; i ++)
        buf[i] += 1;
}
```

基于这两个函数，可以在 `romfs_readpage()` 函数中实现加密：

```
    // 判断当前索引节点对应的文件名是否为 encrypted_file_name ，记录在 bool 变量 flag 中，是则为 true
    flag = is_encrypted_file(inode);

    if (offset < size) {
        size -= offset;
        fillsize = size > PAGE_SIZE ? PAGE_SIZE : size;

        pos = ROMFS_I(inode)->i_dataoffset + offset;
```

```
        ret = romfs_dev_read(inode->i_sb, pos, buf, fillsize);
        if (ret < 0) {
            SetPageError(page);
            fillsize = 0;
            ret = -EIO;
        }

        // 如果当前文件名是 encrypted_file_name ，且没有发生 page error 导致 fillsize
为 0
        if (flag && fillsize > 0)
            encrypt((char*)buf, fillsize);
```

3. 修改文件权限

阅读 `super.c` ，可以发现 romfs 在 `romfs_lookup()` 函数中调用了函数 `romfs_iget()` 用于获取 inode 对象（根据其在映像中的位置），在此之后可以修改 inode 对象的 `i_mode` 字段修改文件权限。在函数 `romfs_lookup()` 函数的前面已经获取了文件名：`name = dentry->d_name.name;` ，所以不需要另外自己实现。只需要在调用 `romfs_iget()` 函数后，比较 `name` 和 `exec_file_name` 是否相同，如果相同则在 S_IXUGO 对应位设置为 1（user、group、other 用户权限均设为可执行）。实现如下：

```
        if (ret == 1) {
            /* Hard link handling */
            if ((be32_to_cpu(ri.next) & ROMFH_TYPE) == ROMFH_HRD)
                offset = be32_to_cpu(ri.spec) & ROMFH_MASK;
            inode = romfs_iget(dir->i_sb, offset);

            // 添加执行权限
            if (exec_file_name && !strcmp(exec_file_name, name))
                inode->i_mode |= S_IXUGO;

            break;
        }
```

【4】实验测试及效果截图

1. 模块编译

基于 `linux-5.5.11/fs/romfs/Makefile` ，我实现了自己的 `Makefile` 用于编译修改后的模块。

```
obj-$(CONFIG_ROMFS_FS) += romfs.o
romfs-y := storage.o super.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

ifneq ($(CONFIG_MMU),y)
romfs-$(CONFIG_ROMFS_ON_MTD) += mmap-nommu.o
endif

all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

2. 测试过程与分析

- 在代码目录修改好 `super.c` 后，使用 `make` 命令进行编译。
- 在文件夹 `src` 准备测试文件，可以看到，文件 `aa`，`bb`，`ft` 的权限均为 644。

```
root@ecs-lzh:~/lab4# ls
internal.h  Makefile      ref  storage.c  test.sh
Kconfig     mmap-nommu.c  src  super.c    tgt
root@ecs-lzh:~/lab4#
root@ecs-lzh:~/lab4# make
make -C /lib/modules/5.5.11/build M=/root/lab4 modules
make[1]: Entering directory '/usr/src/linux-5.5.11'
  CC [M]  /root/lab4/storage.o
  CC [M]  /root/lab4/super.o
  LD [M]  /root/lab4/romfs.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /root/lab4/romfs.mod.o
  LD [M]  /root/lab4/romfs.ko
make[1]: Leaving directory '/usr/src/linux-5.5.11'
root@ecs-lzh:~/lab4#
root@ecs-lzh:~/lab4# ls -l src
total 12
-rw-r--r-- 1 root root 6 May 27 23:56 aa
-rw-r--r-- 1 root root 6 May 27 23:56 bb
-rw-r--r-- 1 root root 6 May 27 23:56 ft
```

- 使用 genromfs 生成格式为 romfs 的镜像文件 `lab4.img`。
- 插入编译好的模块 `romfs.ko`，传入参数为 `src` 下的文件 `aa`，`bb`，`ft`，分别实现被隐藏、被加密（字符+1）、权限变为可执行。
- 挂载镜像文件 `lab4.img` 到目录 `tgt` 下。可以看到，目录 `tgt` 下的 `aa` 文件已经被隐藏，`bb` 文件中的字符都加了 1，`ft` 文件增加了可执行权限。

```
root@ecs-lzh:~/lab4# genromfs -V "vromfs" -f lab4.img -d src
root@ecs-lzh:~/lab4# insmod romfs.ko hided_file_name=aa encrypted_file_name=bb
exec_file_name=ft
root@ecs-lzh:~/lab4# mount -o loop lab4.img tgt
root@ecs-lzh:~/lab4#
root@ecs-lzh:~/lab4# ls -l tgt
total 0
-rw-r--r-- 1 root root 6 Jan  1  1970 bb
-rwxr-xr-x 1 root root 6 Jan  1  1970 ft
root@ecs-lzh:~/lab4#
```

```
root@ecs-lzh:~/lab4# cat src/bb
abc123root@ecs-lzh:~/lab4#
root@ecs-lzh:~/lab4#
root@ecs-lzh:~/lab4# cat tgt/bb
bcd234root@ecs-lzh:~/lab4#
```

【5】实验心得

本次实验基于 Linux 内核虚拟文件系统的理论课程，通过实践熟悉了 Linux 中各种用于文件管理的数据结构和操作，提高了我阅读分析大规模系统软件源码的能力和调试能力。本次实验在实验手册的帮助下，相对比较容易，但是也需要阅读较多的源码，分析各种容易混淆的数据结构的含义。总之，本次实验收获不少！感谢老师和助教的指导！