

H1 CS353 Linux内核 Project4 报告

517030910214 刘宏洲

H2 0. 简介

在本Project中，我们将探索有关Linux文件系统方面的内容。

本次Project的实验平台为华为云ECS服务器，它采用双核鲲鹏处理器，系统环境配置如下：

- Ubuntu 18.04.3 LTS (GNU/Linux 5.6.6 aarch64)
- GNU Make 4.1
- gcc 7.4.0

H2 1. 隐藏文件

H3 1.1 实现

首先，我们在romfs/super.c的开头声明参数：

```
1 static char *hided_file_name;
2 static char *encrypted_file_name;
3 static char *exec_file_name;
4 module_param(hided_file_name, charp, 0644);
5 module_param(encrypted_file_name, charp, 0644);
6 module_param(exec_file_name, charp, 0644);
```

在开始阅读romfs文件系统的代码之前，我们先要了解romfs文件系统在磁盘上的inode是什么样的。在linux/romfs_fs.h中，inode定义如下：

```
1 struct romfs_inode {
2     __be32 next; /* low 4 bits see ROMFH_ */
3     __be32 spec;
4     __be32 size;
5     __be32 checksum;
6     char name[0];
7 };
```

我们可以看到，一个inode的头部包含了指向下一个inode的指针。而由于文件系统中的信息是16位对其的，因此这个字段的低4位全为0，但romfs文件系统巧妙地利用了这四位，用来标识inode的权限。spec字段储存了与目录、硬链接、设备文件相关的信息。而size为文件的大小，checksum为校验和，name字段只占了一个字节。

为了隐藏指定名称的文件，我们需要在用户进行与读取文件目录有关的操作（如：ls、find）时跳过指定文件，不显示其信息。在romfs文件系统中，与读取文件目录有关的函数是定义在super.c中的romfs_readdir。在这个函数中，用来遍历文件目录的操作主要在以下循环中：

```
1 /* Not really failsafe, but we are read-only... */
2 for (;;) {
3     if (!offset || offset >= maxoff) {
4         offset = maxoff;
5         ctx->pos = offset;
```

```

6     goto out;
7 }
8     ctx->pos = offset;
9
10    /* Fetch inode info */
11    ret = romfs_dev_read(i->i_sb, offset, &ri, ROMFH_SIZE);
12    if (ret < 0)
13        goto out;
14
15    j = romfs_dev_strnlen(i->i_sb, offset + ROMFH_SIZE,
16        sizeof(fsname) - 1);
17    if (j < 0)
18        goto out;
19
20    ret = romfs_dev_read(i->i_sb, offset + ROMFH_SIZE, fsname, j);
21    if (ret < 0)
22        goto out;
23    fsname[j] = '\0';
24
25    ino = offset;
26    nextfh = be32_to_cpu(ri.next);
27
28    if (strcmp(fsname, hided_file_name) == 0) // can we still use name got
    from dentry? they should be the same
29        goto skip;
30
31    if ((nextfh & ROMFH_TYPE) == ROMFH_HRD)
32        ino = be32_to_cpu(ri.spec);
33    if (!dir_emit(ctx, fsname, j, ino,
34        romfs_dtype_table[nextfh & ROMFH_TYPE]))
35        goto out;
36 skip:
37    offset = nextfh & ROMFH_MASK;
38 }
39 out:
40     return 0;

```

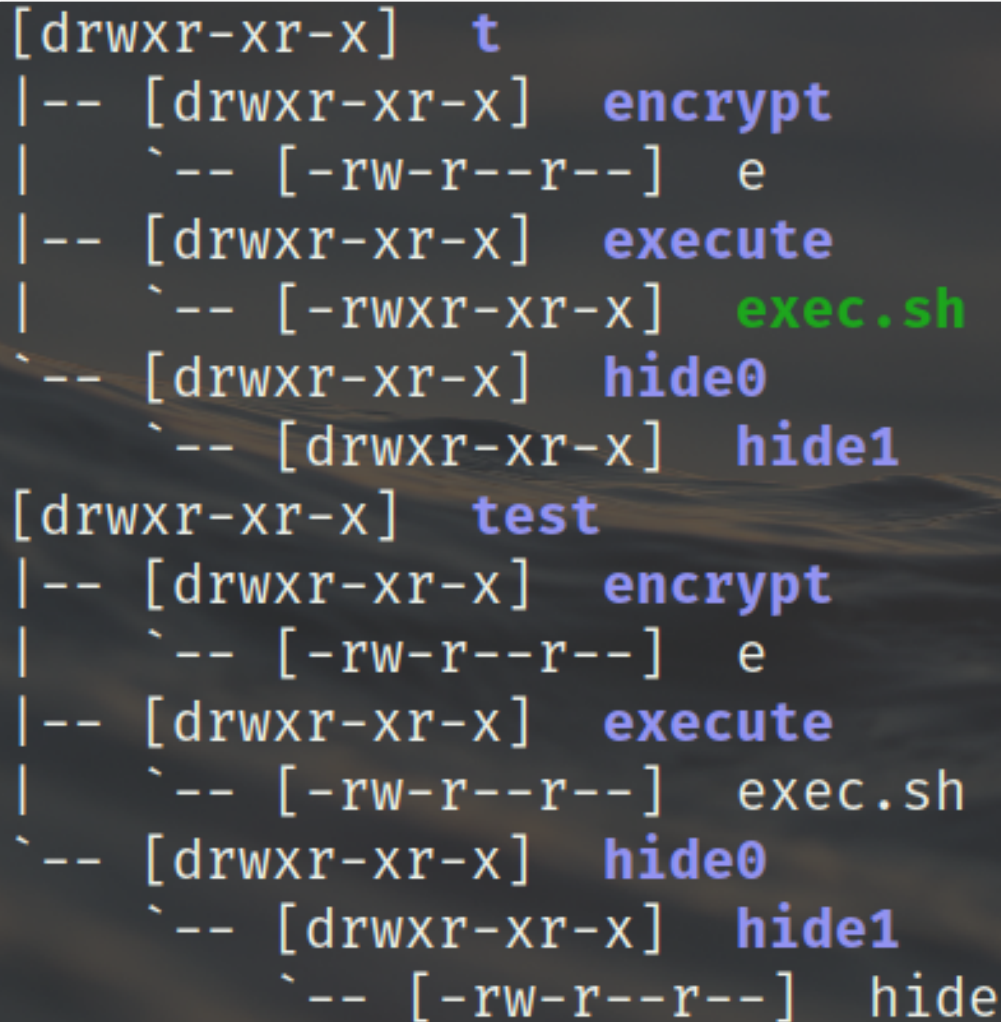
进入循环先判断 `offset` 是否为空，或是已经读到了文件系统的边界。循环前已经根据 `dir_context ctx` 获得了当前文件在超级块中的偏移量，因此我们可以通过 `romfs_dev_read` 函数从超级块上读取inode信息。首先读取当前位置的inode信息存入 `ri`，然后利用 `romfs_dev_strnlen` 从inode信息后计算文件名长度，再根据文件名长度获得文件名 `fsname`。这时候就可以将各种信息通过 `dir_emit` 提供给 `ctx` 了。之后利用 `ri` 的 `next` 字段，开始访问下一个inode。如果要隐藏一个指定文件，那么只需要跳过 `dir_emit` 直接进入对下一个inode的处理即可。因此，我在这个操作之前增加了对文件名的判断，如果满足条件，就会跳转到 `skip` 标号。参见以上代码的28-29及36-37行。

H3 1.2 结果

这里生成romfs镜像并挂载使用了助教提供的命令：

```
1 genromfs -V "vromfs" -f test.img -d test
2 insmod romfs.ko hided_file_name=hide encrypted_file_name=e
  exec_file_name=exec.sh
3 mount -o loop test.img t
```

查看目录使用了 `tree -p` 命令，显示的结果也包括了后续实验完成的内容。我们可以看到，尽管文件 `hide` 被放到了 `hide0/hide1/` 文件夹里面，它仍然被隐藏了。



```
[drwxr-xr-x]  t
|-- [drwxr-xr-x]  encrypt
|   `-- [-rw-r--r--]  e
|-- [drwxr-xr-x]  execute
|   `-- [-rwxr-xr-x]  exec.sh
`-- [drwxr-xr-x]  hide0
    `-- [drwxr-xr-x]  hide1
[drwxr-xr-x]  test
|-- [drwxr-xr-x]  encrypt
|   `-- [-rw-r--r--]  e
|-- [drwxr-xr-x]  execute
|   `-- [-rw-r--r--]  exec.sh
`-- [drwxr-xr-x]  hide0
    `-- [drwxr-xr-x]  hide1
        `-- [-rw-r--r--]  hide
```

图1. 隐藏文件

H2 2. 加密文件

H3 2.1 实现

为了加密一个文件，我们必须读取它的内容。而文件的内容被存放在inode中，因此需要找到romfs文件系统实现中读文件内容的函数。在 `super.c` 开头的 `romfs_readpage` 函数就是一个从文件系统中读取一个页面数据的函数。我们对其进行分析：

```
1 static int romfs_readpage(struct file *file, struct page *page)
2 {
3     int i;
4     struct inode *inode = page->mapping->host;
5     loff_t offset, size;
6     unsigned long fillsize, pos;
7     void *buf;
8     int ret;
9     char *bufp;
```

```

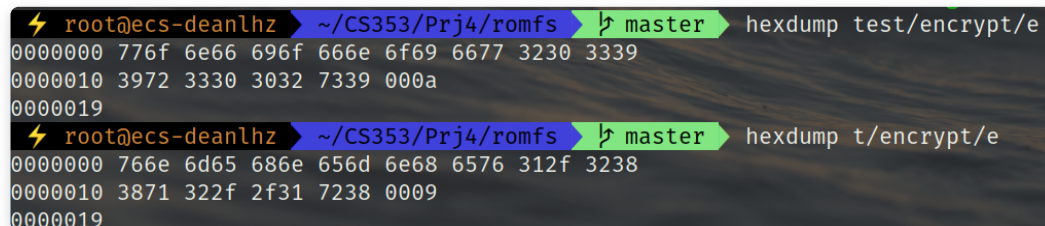
10
11     buf = kmap(page);
12     if (!buf)
13         return -ENOMEM;
14
15     /* 32 bit warning -- but not for us :) */
16     offset = page_offset(page); // current offset in the mapped-virtual memory
17     size = i_size_read(inode); // the file size of this inode
18     fillsize = 0;
19     ret = 0;
20     if (offset < size) { // if the file is not fully mapped into the memory
21         size -= offset; // the rest bytes
22         fillsize = size > PAGE_SIZE ? PAGE_SIZE : size; // read one page, the buffer
size to fill
23
24         pos = ROMFS_I(inode)->i_dataoffset + offset; // the position where not
mapped data starts
25
26         ret = romfs_dev_read(inode->i_sb, pos, buf, fillsize); //read data from
device to the buffer
27
28         if (strcmp(file->f_path.dentry->d_name.name, encrypted_file_name) ==
0)
29         {
30             bufp = (char *)buf;
31             for (i = 0; i < fillsize; ++i)
32             {
33                 *bufp = *bufp - 1;
34                 bufp++;
35             }
36         }
37
38         if (ret < 0) {
39             SetPageError(page);
40             fillsize = 0;
41             ret = -EIO;
42         }
43     }
44
45     if (fillsize < PAGE_SIZE) //set the rest of this buffer as 0
46         memset(buf + fillsize, 0, PAGE_SIZE - fillsize);
47     if (ret == 0)
48         SetPageUptodate(page);
49
50     flush_dcache_page(page);
51     kunmap(page);
52     unlock_page(page);
53     return ret;
54 }

```

首先，函数会将页面映射到buf中，然后获取当前文件对应的页面的偏移量和当前inode的大小。根据页面偏移量和inode大小的关系，我们调整写入页面数据的大小，如果inode没有完全被写入页面（offset < size）那么计算还没有写入页面的数据大小size。由于一次只能写入一个页面，fillsize最多只能为PAGE_SIZE。之后获得inode在超级块上的偏移量，并通过romfs_dev_read从inode中读取数据直接写入buf。之后，如果fillsize的大小不足PAGE_SIZE，还需要将这个页面剩下的空间填满0。而我们对文件的加密，就写在函数将数据从inode读入buf之后。这里使用file->f_path.dentry->d_name.name获取文件名进行对比，与上一个内容中从inode直接获取效果是一样的。如果当前文件是要加密的文件，那么就对buf中的每个字符的ASCII码减去一。具体代码参见以上代码的29-36行。

H3 2.2 结果

利用hexdump以16进制的方法查看文件内容进行对比。如图2，上方是加密前的文件内容，下方是加密后的文件内容。可见，每个字节都较之前减少了1。



```
root@ecs-deanlh2 ~/CS353/Prj4/romfs master hexdump test/encrypt/e
00000000 776f 6e66 696f 666e 6f69 6677 3230 3339
00000010 3972 3330 3032 7339 000a
00000019
root@ecs-deanlh2 ~/CS353/Prj4/romfs master hexdump t/encrypt/e
00000000 766e 6d65 686e 656d 6e68 6576 312f 3238
00000010 3871 322f 2f31 7238 0009
00000019
```

图2. 加密文件

H2 3. 更改文件权限

H3 3.1 实现

我们之前提到了romfs的inode中与权限有关的巧妙设计。在这一部分，我们要修改一个文件的权限为可执行。这就要求我们获得文件的inode，以便进行修改。而我们又知道，查找一个文件时，要通过地址的层层解析获得dentry，然后由dentry找到对应的inode，最终获得数据。romfs中通过romfs_lookup来查找一个dentry，在这个函数中又通过romfs_iget来获取对应的inode。

观察romfs_lookup，我们可以发现，它也是通过一个循环读取磁盘上的inode，比较文件名称，然后获得inode的。具体代码如下：

```
1  for (;;) {
2      if (!offset || offset >= maxoff)
3          break;
4
5      ret = romfs_dev_read(dir->i_sb, offset, &ri, sizeof(ri));
6      if (ret < 0)
7          goto error;
8
9      /* try to match the first 16 bytes of name */
10     ret = romfs_dev_strcmp(dir->i_sb, offset + ROMFH_SIZE, name,
11                             len);
12     if (ret < 0)
13         goto error;
14     if (ret == 1) {
15         /* Hard link handling */
16         if (((be32_to_cpu(ri.next) & ROMFH_TYPE) == ROMFH_HRD)
17             offset = be32_to_cpu(ri.spec) & ROMFH_MASK;
18         inode = romfs_iget(dir->i_sb, offset);
19         break;
20     }
```

```

21
22     /* next entry */
23     offset = be32_to_cpu(ri.next) & ROMFH_MASK;
24 }

```

可以看见这和 `romfs_readdir` 中的循环很相似，不同之处在于，这里找到 `inode` 后就会跳出循环，而不必一直循环到超级块的最后。具体的 `romfs_iget` 代码便不再详细分析。这个函数用于将磁盘上的 `inode` 读取为内核中的 `inode`。获得 `inode` 后，我们就可以对它的权限做更改了。在以上循环之后添加代码：

```

1  if (strcmp(name, exec_file_name) == 0)
2  {
3      inode->i_mode |= S_IXUGO;
4  }

```

这里 `name` 是 `dentry` 的名字，在循环之前就已经获得了。只需要将 `inode` 的 `i_mode` 按位或一个代表可执行权限的宏 `S_IXUGO` 即可。

H3 3.2 结果

这部分的结果就是，原来不能执行的文件变成了可执行文件（对比请参考图1）。而我们直接执行它，就可以得到结果。（文件中包含 `echo hello` 命令）

```

⚡ root@ecs-deanlh2 ~/CS353/Prj4/romfs ➤ master ➤ cd t/execute
⚡ root@ecs-deanlh2 ~/CS353/Prj4/romfs/t/execute ➤ ./exec.sh
hello

```

图3. 更改文件权限

H2 4. 总结

在这个Project中，我对 `romfs` 这个简单的只读文件系统的工作原理有了粗浅的了解，加深了对上课学到的文件系统部分知识的理解，为后续了解更加复杂文件系统的原理打下了基础。