LAB4

吴非 519021910924

内核环境：5.5.11-050511-generic

首先根据提示复制 super.c 到实验目录，第一步要做的事首先根据传参设置三个参数：

```
83  static char *hided_file_name;
84  static char *encrypted_file_name;
85  static char *exec_file_name;
86
87  module_param(hided_file_name, charp, 0644);
88  module_param(encrypted_file_name, charp, 0644);
89  module_param(exec_file_name, charp, 0644);
```

实验一使用过此函数：
module_param (name, type, perm)是一个宏，表示向当前模块传入参数
charp/*一个字符指针值. 内存为用户提供的字串分配，指针因此设置.*/
下面根据三个功能依次阐述：

1．隐藏文件：

romfs_readdir : read the entries from a directory
.iterate_shared = romfs_readdir,
iterate_shared: called when the VFS needs to read the directory contents
when filesystem supports concurrent dir iterators

可见这个函数的作用是读取文件夹下文件时（ls -l /mnt）遍历(iterate)给定目录下的所有文件，这一点也可以从函数内的 for 循环得知。
而我需要做的工作就是在 for 循环中找到每个文件名 fsname 并和我们传入的文件名称参数比较，得到文件名后我们可以发现下面有个显眼的函数 dir_emit（查找其用法，读目录项，然后调用 dir_emit 函数填充至用户空间，也就是显示在终端） 那么我们要做的工作就是如果文件名匹配跳过这个函数的执行，这也决定了我插入的位置就在这个函数之前，即实现了隐藏，代码如下。

```
233         /* modify*/
234         if (hided_file_name && !strcmp(hided_file_name, fsname))
235             goto skip;
236     /* modify*/
237
238         if (!dir_emit(ctx, fsname, j, ino,
239                 romfs_dtype_table[nextfh & ROMFH_TYPE]))
240             goto out;
241
242     skip://设置跳过到的位置，因为最后这句无论如何应该都要执行的
243         offset = nextfh & ROMFH_MASK;
244     }
245 out:
246     return 0;
247 }
```

2 文件加密:

romfs_readpage read a page worth of data from the image
.readpage = romfs_readpage
readpage: called by the VM to read a page from backing store.
虽然描述有些模糊，但是可以推测出应该是 cat 的时候被调用了（此时已经存在于 backing store 中了）
由注释可以大致知道这个函数是读取文件内容的，根据实验展示我们需要判断是否是加密文件，如果是，我们把文件的内容用一个简单的加密算法计算后然后再输出给用户。
其中第一步，根据提示我们需要根据 inode 找到文件名，我留意到 romfs_readdir 函数中有获取文件名的函数，主要通过 romfs_dev_strnlen 和 romfs_dev_read 函数实现，观察到缺少 offset 变量，但是这个变量经尝试没找到怎么获得。转变思路从 file 结构体入手，此结构体中没有对应的文件名，查看其结构体中的第二个成员 f_path，在其结构体中看到了熟悉的 dentry 成员，继续搜索相关 dentry 结构体信息，In the path /mnt/cdrom/foo, the components /, mnt, cdrom, and foo are all dentry objects.，以及 struct qstr d_name; /* dentry name */ 可以看到这个成员就是我们要找的文件名，之后通过 fillsize = size > PAGE_SIZE ? PAGE_SIZE : size;和下面的读取函数 romfs_dev_read 可以得到 fillsize 是最后读出的数据量，由于我们的数据是 char 占一个字节，所以直接用 fillsize 为循环终止数即可。最后实际的操作我直接把所有的 char 加 1 视作加密了，插入的位置根据提示，直接加在读出数据到 buf 的函数后面即可，代码如图:

```
152     ret = romfs_dev_read(inode->i_sb, pos, buf, fillsize);
153     if (ret < 0) {
154         SetPageError(page);
155         fillsize = 0;
156         ret = -EIO;
157     }
158     /*********modify********/
159 if (strcmp(file->f_path.dentry->d_name.name, encrypted_file_name) == 0){
160     if (fillsize > 0)
161         encrypt((char*)buf, fillsize);
162 }
163     }
164     /*********modify********/
```

```
//add 1
static void encrypt(char *buf, int fillsize) {
    int i;
    for (i = 0; i < fillsize; i ++)
        buf[i] += 1;
}
```

3. 修改权限:

.lookup = romfs_lookup,
lookup: called when the VFS needs to look up an inode in a parent directory.
我暂且认为这个函数在我挂载镜像的时候就执行了。
和上两个功能一样，需要找到文件名，可以看到源代码中直接给出了，因此我们只需要修改权限即可。而本函数的所有变量中只有两个可能与文件权限有关，inode，dentry。搜索

inode 结构体，第一个成员就是叫 i_mode 的。
https://blog.csdn.net/jinking01/article/details/105771173 得知确实和权限相关，参考其他地方的 linux 源码，通过或操作实现给权限的功能，

```
i_mode |= S_IWUSR | S_IXUSR;
```

因此通过此方式给 S_IXUSR 权限即可。

最后是 mount -o loop test.img /mnt -t romfs #挂载镜像到/mnt 下，这个命令，-o 代表选项，loop 为伪设备，（https://zh.wikipedia.org/wiki/Loop%E8%AE%BE%E5%A4%87） （-t vfstype 指定文件系统的类型，通常不必指定，mount 会自动选择正确的类型。）最后插入的位置也就合理的选在获取 inode 变量代码之后。如下：

```
290     inode = romfs_iget(dir->i_sb, offset);
291
292     /*********modify********/
293     if (exec_file_name && !strcmp(exec_file_name, name))
294         inode->i_mode |= S_IXUSR;
295     /*********modify********/
296
297     break;
```

结果：
使用 run.sh 中的相关命令编译后：

```
root@wf-virtual-machine:/home/wf/linux_files/linux_exp/lab4# ls -l /mnt
总用量 0
-rw-r--r-- 1 root root  8 1月   1  1970 bb
-rwxr--r-- 1 root root 24 1月   1  1970 cc
root@wf-virtual-machine:/home/wf/linux_files/linux_exp/lab4# cat /mnt/bb
bcdfegh
       root@wf-virtual-machine:/home/wf/linux_files/linux_exp/lab4# /mnt/cc
pass
```

总结：

本次实验基于 Linux 内核虚拟文件系统的相关理论，实践了 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。虽然相对前几次实验较简单，但是也有一些函数和数据结构比较像，需要仔细查找资料区分和应用。

实验过程中的问题：

开始我只用了 super.c 编译，但是提示没有 internal.h,从 usr/src/目录中提取此文件复制到实验目录下，而又发现另一个报错说没有一些结构体，查阅后发现在 storage.c 中，遂复制一份并修改 makefile 文件一起编译。
然后就是一些 API 函数的功能查找和修改，总体没有耗费太多时间。