系统软件课程设计

项目报告

项目名称：基于LSM 的程序运行权限管理

项目组成员：

傅雨东 F1103602 5110369036

院系：信息安全工程学院

学期：2014-2015学年第1学期

报告完成日期：2015年1月

项目摘要

Linux主要依据进程的启动用户判断进程的操作权限，这意味着进程可以共享启动用户的所有权限。这种权限授权方式违背信息安全的最小特权原则，最小特权原则指明按一个主体（进程等）完成它正常任务所需要的最小权限进程授权，如果一种权限对该主体正常运行不是必需的，就不要赋予该主体这种权限。

基于LSM框架，可以实现接近于最小特权管理的一种权限控制机制，管理员可以给一个可执行程序配置响应的运行权限，这种机制保证按管理员的权限配置对该可执行程序（实际上是该执行程序对应的进程）进行运行权限控制。

针对应用程序的运行权限控制给该应用程序划定一个运行权限范围，防止应用程序程序有意或无意的越过该运行权限，对操作系统和其他应用程序的运行构成破坏。在国外的操作系统安全相关研究中，这类权限控制机制被称为sandbox，在多种安全操作系统中得到支持和应用。

整个项目分为两大块部分，分别为配置程序和内核模块。配置程序的功能是让用户能方便地配置各种控制参数，权限的配置操作包括权限的查看、添加、删除、修改、导入配置文件、导出配置文件。内核模块的功能是根据配置程序所设置的用户配置进行对相应程序的操作权限的判断及处理，包括文件访问类操作、

通信类操作、管理类操作三大部分。

项目从2014年10月开始直至2014年12月11日完成。在2014年11月14日时仅完成内核编译及配置程序，于2014年11月29日完成获取当前进程的绝对路径，之后直至完成每天分别完成一个钩子函数的挂钩。

最后项目实现了如下功能：配置程序对权限的查看、添加、删除、修改、导入配置文件、导出配置文件；内核模块对文件访问类操作（创建、运行、打开、删除、修改文件）、通信类操作（socket通信、共享内存、消息队列）、管理类操作（mount、umount）的挂钩。

经过测试上述所有功能都成功实现，但是文件访问类操作中的修改文件操作挂钩了file\_permission函数，直接导致系统运行一段时间会发生卡死的现象。

目录

[第一章 需求分析和项目目标 4](#_Toc407376270)

[一、项目需求分析 4](#_Toc407376271)

[二、项目所要完成的功能目标 5](#_Toc407376272)

[三、项目可行性分析 6](#_Toc407376273)

[四、项目的参与人员和项目进度计划 6](#_Toc407376274)

[五、开发环境及开发工具的选择和确定 6](#_Toc407376275)

[六、测试方法的确定 7](#_Toc407376276)

[第二章 项目总体设计 8](#_Toc407376277)

[1、总体结构图 8](#_Toc407376278)

[2、项目的模块划分 8](#_Toc407376279)

[3、配置程序模块详细设计 10](#_Toc407376280)

[4、新设备文件的驱动模块详细设计 12](#_Toc407376281)

[5、辅助模块详细设计 15](#_Toc407376282)

[6、访问控制函数模块详细设计 17](#_Toc407376283)

[7、内核模块的初始化模块详细设计 19](#_Toc407376284)

[8、内核模块的注销模块详细设计 21](#_Toc407376285)

[第三章 系统实现 23](#_Toc407376286)

[1、 实现环境 23](#_Toc407376287)

[2、 开发工具 23](#_Toc407376288)

[3、 源文件情况 23](#_Toc407376289)

[4、 目标程序的组成及运行方式 25](#_Toc407376290)

[第四章 项目测试 26](#_Toc407376291)

[1、测试方法 26](#_Toc407376292)

[2、测试流程 26](#_Toc407376293)

[3、具体测试 26](#_Toc407376294)

[4、测试结论 37](#_Toc407376295)

[第五章 项目小结 38](#_Toc407376296)

# 第一章 需求分析和项目目标

## 一、项目需求分析

### 1、总体要求

#### 1.1 总体功能要求

完成配置程序使用户能方便地通过配置程序对规则进行查看、增加、修改、删除操作。

完成程序运行权限管理，包括文件访问类操作、通信类操作、管理类操作，其中以文件访问类操作为主。

#### 1.2 软件开发平台要求

①以Linux系统为宜

②内核版本不宜过高，因为过高会导致要用到的一些函数没导出

③2.6.10以上内核版本需要去除已经注册的其他安全机制

#### 1.3 运行环境要求

①以Linux系统为宜

②内核版本不宜过高，因为过高会导致要用到的一些函数没导出

③2.6.10以上内核版本需要去除已经注册的其他安全机制

### 2、外部接口需求

#### 2.1 用户界面：

命令行界面即可，不需要太友好的界面

#### 2.2 硬件接口：

无

#### 2.3 软件接口：

无

#### 2.4 通讯接口：

无

## 二、项目所要完成的功能目标

### 1、系统各子功能说明

#### 1.1. 权限配置程序

① 权限的配置操作（包括权限的查看、添加、删除、修改）

② 权限的配置文件操作（包括权限的配置文件的导入、导出）

#### 1.2. 程序运行权限管理

① 文件访问类操作

1. 各类文件（或目录）的创建
2. 各类文件（或目录）的打开
3. 各类文件（或目录）的写
4. 各类文件（或目录）的执行
5. 各类文件（或目录）的删除

② 通信类操作：

1. socket通信
2. 共享内存
3. 消息队列

③ 管理类操作：

1. mount
2. umount

### 2、输入/输出数据

#### 2.1输入

字符串，用户输入的控制信息

#### 2.2输出

无，表现为被设置的可执行程序的各项操作的权限设置

## 三、项目可行性分析

1. 本项目的工作量不大，均能够按期完成。
2. 本项目需要在Linux平台上进行，可在本人计算机上完成，不需要额外的支持设备。
3. 本项目的关键问题和难点主要在如何获取当前进程对应的程序的绝对路径，最后参考了<http://www.spinics.net/lists/newbies/msg19531.html>的方法。

## 四、项目的参与人员和项目进度计划

项目的参与人员：傅雨东

项目进度计划：

|  |  |
| --- | --- |
| 任务 | 完成时间 |
| 需求分析和总体设计 | 2014.10.20 |
| 新内核编译 | 2014.10.30 |
| 中期报告 | 2014.11.14 |
| 配置程序对规则的添加、修改 | 2014.11.26 |
| 配置程序列出当前规则 | 2014.11.27 |
| 配置程序导入、导出配置文件 | 2014.11.28 |
| 获取当前进程对应的程序的绝对路径 | 2014.11.29 |
| 对程序执行文件删除操作的挂钩 | 2014.11.30 |
| 对程序执行文件创建操作的挂钩 | 2014.12.1 |
| 对程序执行文件打开操作的挂钩 | 2014.12.1 |
| 对程序执行文件修改操作的挂钩 | 2014.12.1 |
| 对程序执行文件执行操作的挂钩 | 2014.12.2 |
| 对程序执行mount、umount操作的挂钩 | 2014.12.3 |
| 对程序执行socket通信操作的挂钩 | 2014.12.4 |
| 对程序执行共享内存操作的挂钩 | 2014.12.4 |
| 对程序执行消息队列操作的挂钩 | 2014.12.4 |
| 答辩 | 2014.12.12 |
| 结题报告 | 2014.12.26 |

## 五、开发环境及开发工具的选择和确定

虚拟机：VMware Workstation 8

操作系统：Fedora Core 6

Linux内核版本：Linux 2.6.18

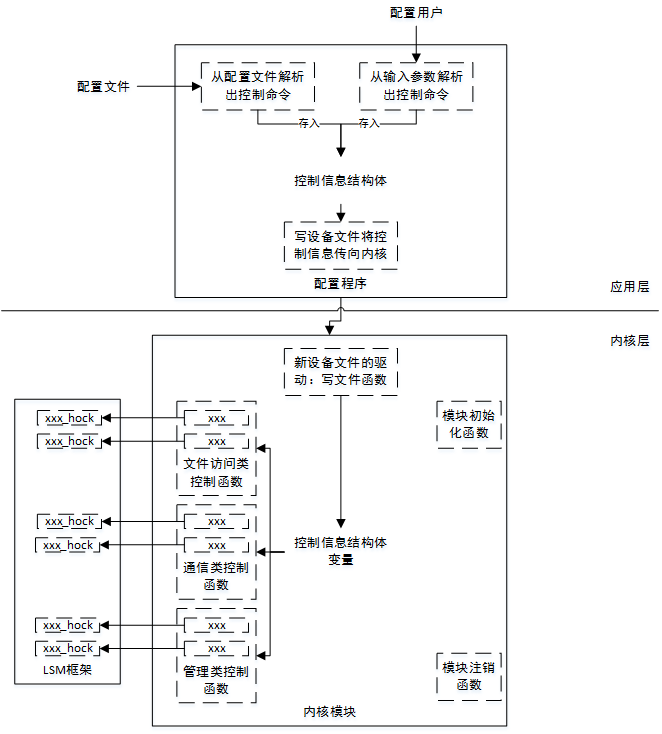
gcc version: gcc version 4.1.1 20061011 (Red Hat 4.1.1-30)

## 六、测试方法的确定

编写测试程序对比测试

# 第二章 项目总体设计

## 1、总体结构图



## 2、项目的模块划分

① 配置程序：负责从用户输入或配置文件中得到控制信息，并转换为一个结构体，并将该结构体通过写新的设备文件将控制信息从用户态传到核心态。

② 新设备文件的驱动：完成应用程序向内核模块发送配置信息，接收配置程序写入的控制信息，然后将该内容保存在控制信息结构体中。

③ 辅助模块：完成一些辅助功能，如根据PID得到实际的程序完整路径。

④ 访问控制函数：根据钩子函数传入的参数信息与辅助模块传入的进程信息，解析出哪个程序要执行什么操作，然后与控制信息结构体中的内容进行比对，如果要禁止则返回-1否则返回0.

⑤ 内核模块的初始化：完成两方面的初始化工作，一是注册新设备文件的驱动程序，二是将预先设计好的相关控制函数注册到LSM框架中。

⑥ 内核模块的注销：完成设备文件驱动的卸载，以及从LSM框架中注销所注册的钩子函数。

## 3、配置程序模块详细设计

### 3.1模块概述：

功能：配置规则

输入：字符串，用户键入的配置规则

输出：无

### 3.2模块的数据处理流程



### 3.3主要数据结构的设计

char filename[256]; //要控制的程序路径

char operation[256]; //对程序的控制行为

char result[512]; //最后的完整控制字符串

int fd; //设备文件打开时用到的文件描述符

struct stat buf; //文件状态结构缓冲区

### 3.4主要的处理函数

本模块依赖的外部标准库函数：

extern char \*strcat(char \*dest,char \*src);

extern char \*strcpy(char\* dest, const char \*src);

extern int strcmp(const char \*1, const char \*s2);

int \_stat(const char \*path,struct \_stat \*buffer)；

int system (char \*s); //run shell ”char \*s”

int write(int handle, void \*buf, int nbyte);

## 4、新设备文件的驱动模块详细设计

### 4.1模块概述

功能：用户态与内核态数据交互

输入：int fd, char \*buf, ssize\_t len

输出：int

### 4.2模块的数据处理流程



### 4.3主要数据结构的设计

char controlledRules[MAX\_RULE\_LENGTH][MAX\_LENGTH]; //规则集，字符串数组

### 4.4主要的处理函数

#### 4.4.1 write\_controlledRules

输入参数：int fd, char \*buf, ssize\_t len

输出类型：int

函数功能：从用户态读取数据传到内核态

依赖的外部函数：

unsigned long copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long n);

char \*strsep(char \*\*stringp, const char \*delim);

extern char \*strcat(char \*dest,char \*src);

extern char \*strcpy(char\* dest, const char \*src);

extern unsigned int strlen(char \*s);

int strncmp(char \*str1, char \*str2, int maxlen);

函数处理流程图：



#### 4.4.2 read\_controlledRules

输入参数：int fd, char \*buf, ssize\_t len

输出类型：int

函数功能：从内核态读取数据传到用户态

依赖的外部函数：

unsigned long copy\_to\_user(void \_\_user \*to, const void \*from, unsigned long n);

函数处理流程图：



## 5、辅助模块详细设计

### 5.1模块概述

功能：获取当前进程对应的程序的绝对路径

输入：无

输出：当前进程对应的程序的绝对路径

### 5.2模块的数据处理流程



### 5.3主要数据结构的设计

task\_struct \*current

### 5.4主要的处理函数

#### 5.4.1 get\_current\_process\_full\_path

输入参数：无

输出类型：char\*

函数功能：获取当前进程对应的程序的绝对路径

依赖的外部函数：

extern char \*d\_path(struct dentry \*, struct vfsmount \*, char \*, int);

void \*kmalloc(size\_t size, int flags);

void \*memset(void \*s, int ch, size\_t n);

void \*memmove(void \*dest, const void \*src, size\_t count);

void kfree(const void \*x);

函数处理流程图：



## 6、访问控制函数模块详细设计

### 6.1模块概述

功能：根据钩子函数传入的参数信息与辅助模块传入的进程信息，解析出哪个程序要执行什么操作，然后与控制信息结构体中的内容进行比对，如果要禁止则返回-1否则返回0

输入：char\* currentProcessFullPath, int authority

输出：int

### 6.2模块的数据处理流程



### 6.3主要数据结构的设计

char\* currentProcessFullPath //当前进程对应的程序的绝对路径

int authority //操作对应的权限值

char controlledRules[MAX\_RULE\_LENGTH][MAX\_LENGTH] //规则集，字符串数组

### 6.4主要的处理函数

#### 6.4.1 check

输入参数：char\* currentProcessFullPath, int authority

输出类型：int

函数功能：判断当前进程是否有执行对应操作的权限

依赖的外部函数：

char \*strsep(char \*\*stringp, const char \*delim);

extern char \*strcpy(char\* dest, const char \*src);

int strncmp(char \*str1, char \*str2, int maxlen);

long simple\_strtol(const char \*cp, char \*\*endp, unsigned int base);

函数处理流程图：



## 7、内核模块的初始化模块详细设计

### 7.1模块概述

功能：模块初始化

输入：无

输出：int

### 7.2模块的数据处理流程



### 7.3主要数据结构的设计

struct file\_operations fops = {

owner: THIS\_MODULE,

write: write\_controlledRules,

read: read\_controlledRules,

};

static struct security\_operations lsm\_ops=

{

.inode\_create = lsm\_inode\_create,

.inode\_unlink = lsm\_inode\_unlink,

.inode\_mkdir = lsm\_inode\_mkdir,

.inode\_rmdir = lsm\_inode\_rmdir,

.file\_permission = lsm\_file\_permission,

.inode\_permission = lsm\_inode\_permission,

.task\_create = lsm\_task\_create,

.socket\_create = lsm\_socket\_create,

.socket\_connect = lsm\_socket\_connect,

.shm\_shmat = lsm\_shm\_shmat,

.msg\_queue\_msgsnd = lsm\_msg\_queue\_msgsnd,

.sb\_mount = lsm\_sb\_mount,

.sb\_umount = lsm\_sb\_umount,

};

### 7.4主要的处理函数

三个系统函数：

int register\_security(&lsm\_ops);

int mod\_reg\_security(KBUILD\_MODNAME, &lsm\_ops);

int register\_chrdev(123, "/dev/controlfile", &fops);

## 8、内核模块的注销模块详细设计

### 8.1模块概述：

功能：模块注销

输入：无

输出：无

### 8.2模块的数据处理流程



### 8.3主要数据结构的设计

struct file\_operations fops = {

owner: THIS\_MODULE,

write: write\_controlledRules,

read: read\_controlledRules,

};

static struct security\_operations lsm\_ops=

{

.inode\_create = lsm\_inode\_create,

.inode\_unlink = lsm\_inode\_unlink,

.inode\_mkdir = lsm\_inode\_mkdir,

.inode\_rmdir = lsm\_inode\_rmdir,

.file\_permission = lsm\_file\_permission,

.inode\_permission = lsm\_inode\_permission,

.task\_create = lsm\_task\_create,

.socket\_create = lsm\_socket\_create,

.socket\_connect = lsm\_socket\_connect,

.shm\_shmat = lsm\_shm\_shmat,

.msg\_queue\_msgsnd = lsm\_msg\_queue\_msgsnd,

.sb\_mount = lsm\_sb\_mount,

.sb\_umount = lsm\_sb\_umount,

};

### 8.4主要的处理函数

三个系统函数：

int unregister\_security(&lsm\_ops);

int mod\_unreg\_security(KBUILD\_MODNAME, &lsm\_ops);

int unregister\_chrdev(123, "procinfo");

# 第三章 系统实现

## 实现环境

虚拟机：VMware Workstation 8

操作系统：Fedora Core 6

Linux内核版本：Linux 2.6.18

## 开发工具

gcc: gcc version 4.1.1 20061011 (Red Hat 4.1.1-30)

## 源文件情况

### 3.1 controlconf.c

代码长度：200行左右

实现功能：提供了用户输入控制命令的接口，便于用户输入控制规则。

主要函数：

int main(int argc, char \*argv[]);

int getCurrentRules(char currentRules[MAX\_RULE\_LENGTH][MAX\_LENGTH]);

主要数据结构：

char filename[256]; //要控制的程序路径

char operation[256]; //对程序的控制行为

char result[512]; //最后的完整控制字符串

int fd; //设备文件打开时用到的文件描述符

struct stat buf; //文件状态结构缓冲区

### 3.2 lsm.c

代码长度：474行左右

实现功能：

（1）模块主体部分：实现一组安全功能函数（钩子函数），该组函数是内核模块的主体。主要包括功能函数来传递操作的上下文进行处理。

（2）模块初始化函数：内核模块加载时注册钩子函数。

（3）模块注销函数：rmmod卸载模块时，注销钩子函数。

（4）write\_controlledRules函数：从用户态读取输入规则，并处理。

（5）read\_controlledRules函数：向用户态传入规则。

主要函数：

//LSM模块初始化和注销函数

static int \_\_init lsm\_init(void);

static void \_\_exit lsm\_exit(void);

//规则读入函数

int write\_controlledRules(int fd, char \*buf, ssize\_t len)

//规则写入函数

int read\_controlledRules(int fd, char \*buf, ssize\_t len)

//获取当前进程对应的程序的绝对路径的函数

static char\* get\_current\_process\_full\_path()

//检查权限函数

int check(char\* currentProcessFullPath, int authority)

//钩子函数

static int lsm\_inode\_rmdir(struct inode \*dir, struct dentry \*dentry)

static int lsm\_inode\_unlink(struct inode \*dir, struct dentry \*dentry)

static int lsm\_inode\_mkdir(struct inode \*dir, struct dentry \*dentry, umode\_t mode)

static int lsm\_inode\_create(struct inode \*dir, struct dentry \*dentry, umode\_t mode)

static int lsm\_inode\_permission(struct inode \*inode, int mask)

static int lsm\_file\_permission(struct file \*file, int mask)

static int lsm\_task\_create(unsigned long clone\_flags)

static int lsm\_sb\_mount(const char \*dev\_name, struct path \*path, const char \*type, unsigned long flags, void \*data)

static int lsm\_sb\_umount(struct vfsmount \* mnt, int flags)

static int lsm\_socket\_create(int family, int type, int protocol, int kern)

static int lsm\_socket\_connect(struct socket \*sock, struct sockaddr \*address, int addrlen)

static int lsm\_shm\_shmat(struct shmid\_kernel \*shp, char \_\_user \*shmaddr, int shmflg)

static int lsm\_msg\_queue\_msgsnd(struct msg\_queue \*msq, struct msg\_msg \*msg, int msqflg)

主要数据结构：

struct security\_operations;

struct file\_operations;

char controlledRules[MAX\_RULE\_LENGTH][MAX\_LENGTH];

### 3.3 测试程序

代码长度：共385行左右

实现功能：测试

主要函数：int main(int argc, char \*argv[]);

主要数据结构：无

## 目标程序的组成及运行方式

### 4.1 生成目标程序

内核模块生成：进入lsm目录make即可

配置程序生成：gcc –o命令即可

测试程序生成：进入test目录中对应目录gcc –o命令即可

### 4.2 运行方式

1. 使用/sbin/insmod lsm.ko加载内核模块
2. 使用./controlconf –i rule.txt可以导入预先设置好的规则集
3. 使用./controlconf –ls列出当前命令
4. 使用./controlconf –o filename 可以导出现在的规则集
5. 使用./controlconf program\_path authority\_value 设置对应程序的权限值
6. 使用./controlconf program\_path 取消对应程序的控制
7. 使用/sbin/rmmod lsm注销内核模块

### 4.3 权限值

共计12位，对应位为1表示运行，0表示禁止

#define MAX\_AUTHORITY "4095\0" //最大权限值

//文件访问类

#define REMOVE\_AUTHORITY 1 //删除文件权限值

#define MKDIR\_AUTHORITY 2 //创建文件夹权限值

#define OPEN\_AUTHORITY 4 //打开文件权限值

#define CREATE\_AUTHORITY 8 //创建文件权限值

#define WRITE\_AUTHORITY 16 //写文件权限值

#define EXEC\_AUTHORITY 32 //执行文件权限值

//通信类

#define SOCKET\_CREATE 256 //创建SOCKET权限值

#define SOCKET\_CONNECT 512 //连接SOCKET权限值

#define SHARED\_MEMORY 1024 //共享内存权限值

#define MESSAGE\_QUEUE 2048 //消息队列权限值

//管理类

#define MOUNT\_AUTHORITY 64 //挂载文件系统权限值

#define UMOUNT\_AUTHORITY 128 //卸载文件系统权限值

# 第四章 项目测试

## 1、测试方法

本项目的测试方法为编写一组测试程序，每一个测试程序对应一个操作，然后通过配置程序禁止这些测试程序执行其要完成的操作，看LSM模块是否达到预期要求。

## 2、测试流程

1. 加载LSM模块
2. 导入预设规则集
3. 依次测试
4. 配置程序操作
5. 导入
6. 查看
7. 新增
8. 修改
9. 导出

② 文件访问类操作

1. 各类文件（或目录）的创建

b) 各类文件（或目录）的打开

c) 各类文件（或目录）的写

d) 各类文件（或目录）的执行

e) 各类文件（或目录）的删除

③ 通信类操作：

a) socket通信

b) 共享内存

c) 消息队列

④ 管理类操作：

a) mount

b) umount

1. 关闭控制，查看原始效果
2. 注销LSM模块

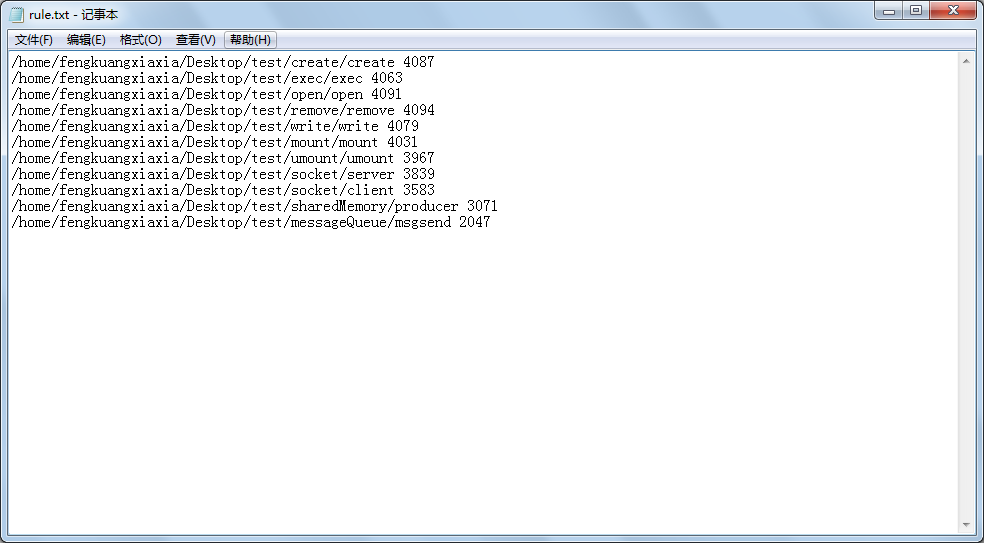
## 3、具体测试

### 3.1导入规则

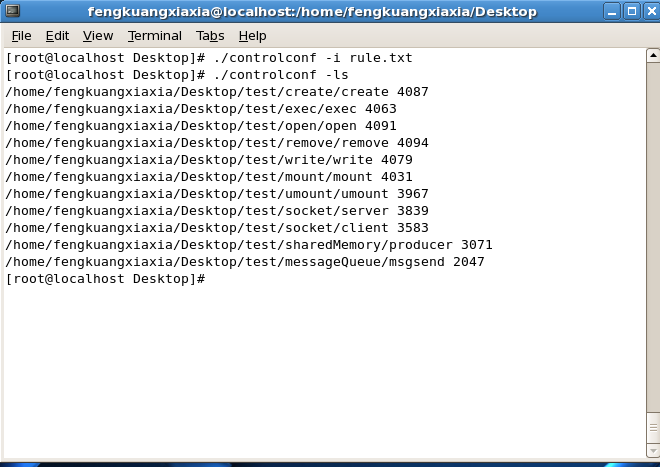
使用./controlconf –i rule.txt导入配置文件

通过./controlconf –ls查看导入的配置是否成功

配置文件：

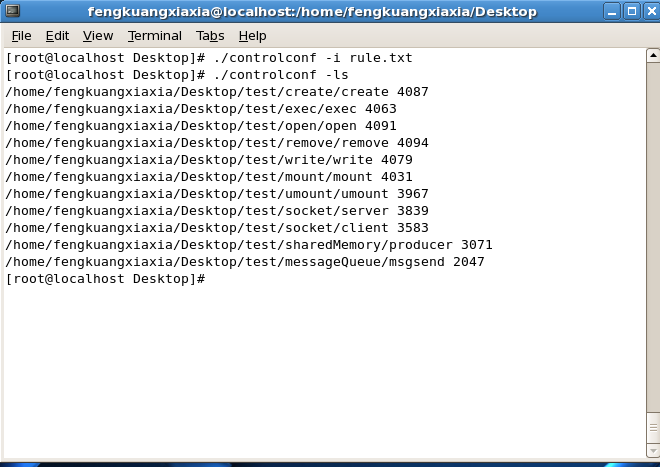


测试结果：



### 3.2查看规则

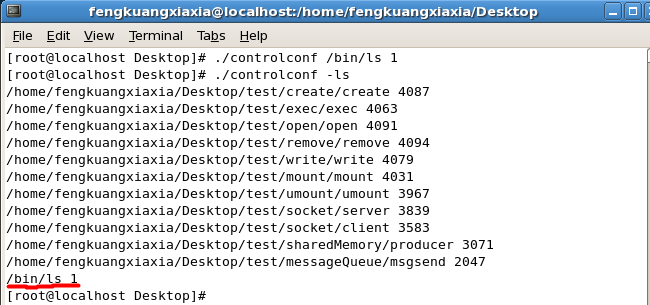
使用./controlconf –ls查看当前命令



### 3.3新增规则

使用./controlconf /bin/ls 1新增一条规则

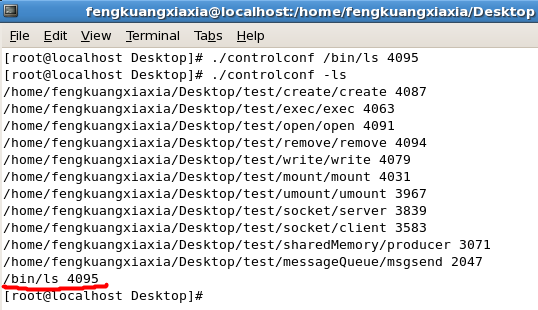
通过./controlconf –ls查看是否成功



### 3.4修改规则

使用./controlconf /bin/ls 4095修改原来的规则

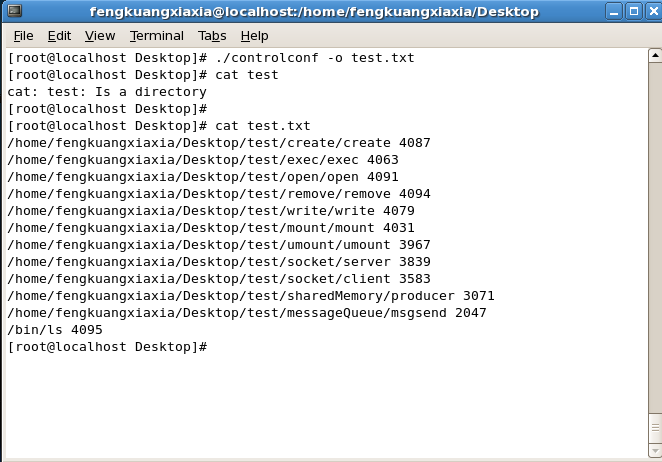
通过./controlconf –ls查看是否成功



### 3.5导出规则

使用./controlconf –o test.txt导出规则

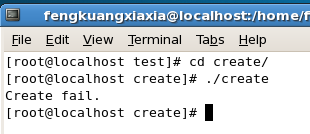
通过cat test.txt查看是否成功



### 3.6控制创建文件

由于已导入了配置文件，不需要手动添加

使用create测试

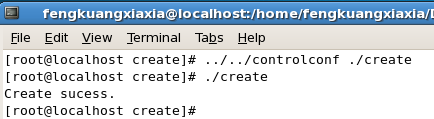


如图发现操作失败

查看dmesg



去除该规则，再次测试

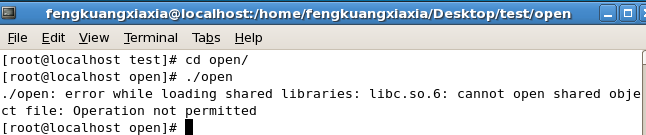


如图成功执行

### 3.7控制打开文件

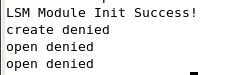
由于已导入了配置文件，不需要手动添加

使用open测试

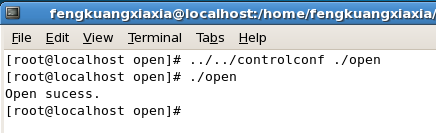


如图发现操作失败

查看dmesg



去除该规则，再次测试

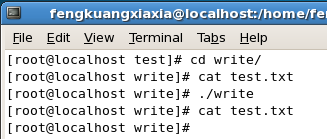


如图成功执行

### 3.8 控制写文件

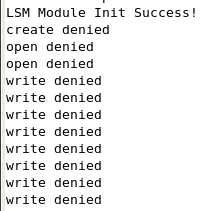
由于已导入了配置文件，不需要手动添加

使用write测试

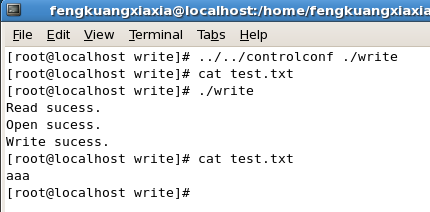


如图发现操作失败

查看dmesg



去除该规则，再次测试

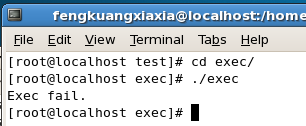


如图成功执行

### 3.9 控制执行文件

由于已导入了配置文件，不需要手动添加

使用exec测试

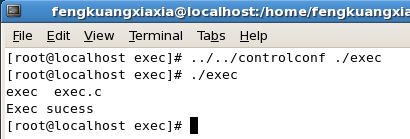


如图发现操作失败

查看dmesg



去除该规则，再次测试

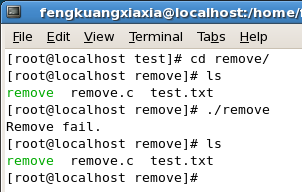


如图成功执行

### 3.10控制删除文件

由于已导入了配置文件，不需要手动添加

使用remove测试

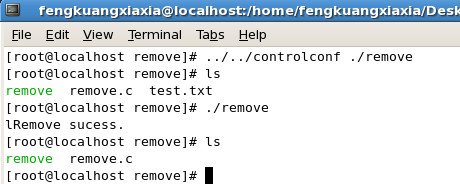


如图发现操作失败

查看dmesg



去除该规则，再次测试

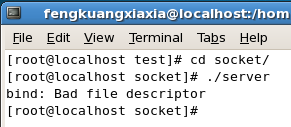


如图成功执行

### 3.11 控制socket 通信

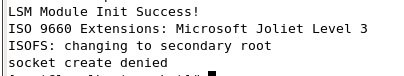
由于已导入了配置文件，不需要手动添加

使用server测试

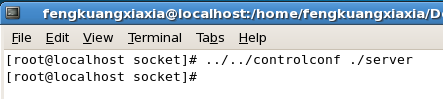


如图发现操作失败

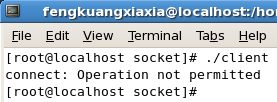
查看dmesg



去除该规则

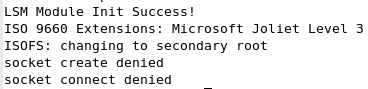


使用client测试

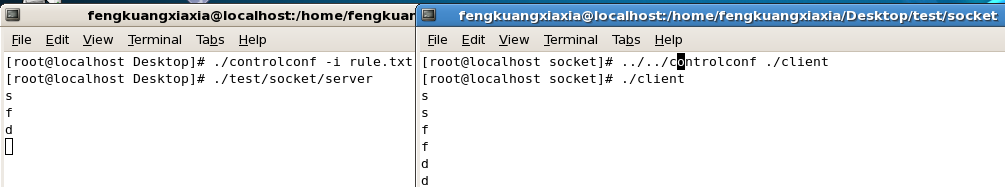


如图发现操作失败

查看dmesg



去除该规则，再次测试

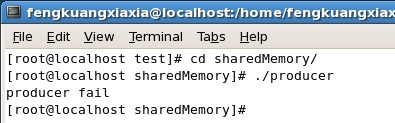


如图成功执行

### 3.12控制共享内存

由于已导入了配置文件，不需要手动添加

使用producer和consumer测试

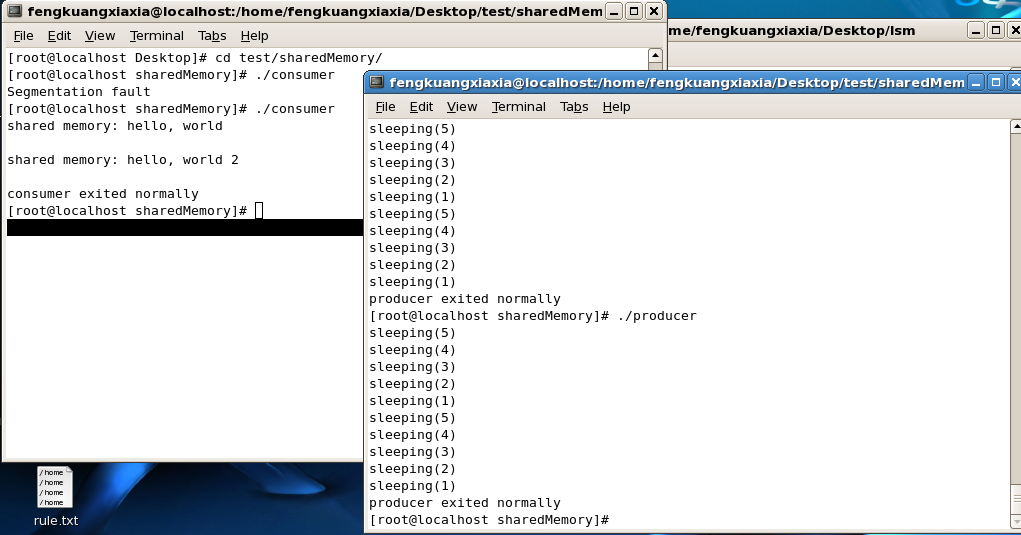


如图发现操作失败

查看dmesg



去除该规则，再次测试

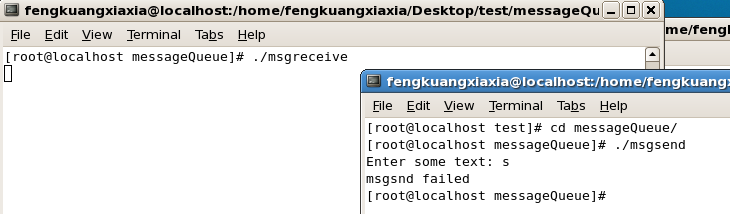


如图成功执行

### 3.13 控制消息队列

由于已导入了配置文件，不需要手动添加

使用msgsend和msgreceive测试

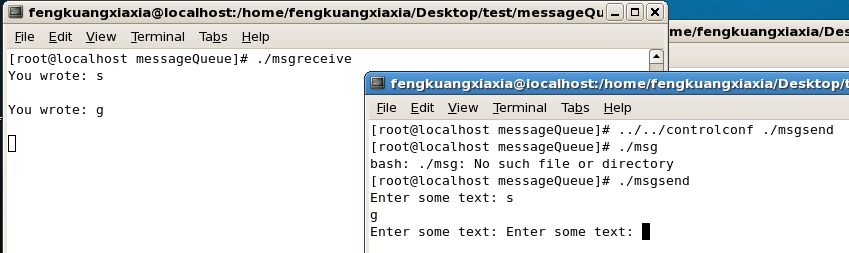


如图发现操作失败

查看dmesg



去除该规则，再次测试

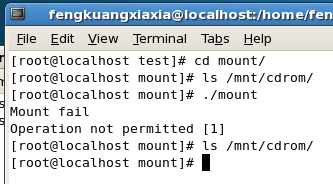


如图成功执行

### 3.14 控制mount

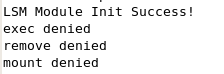
由于已导入了配置文件，不需要手动添加

使用mount测试

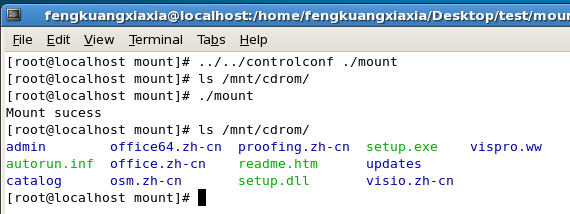


如图发现操作失败

查看dmesg



去除该规则，再次测试

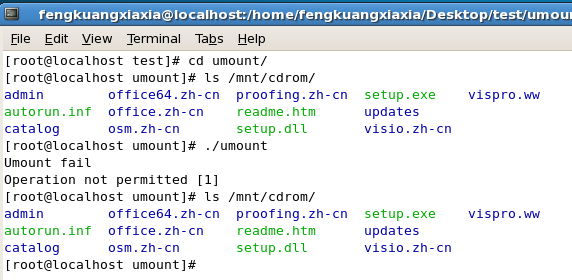


如图成功执行

### 3.15 控制umount

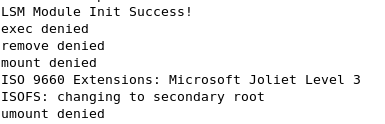
由于已导入了配置文件，不需要手动添加

使用umount测试

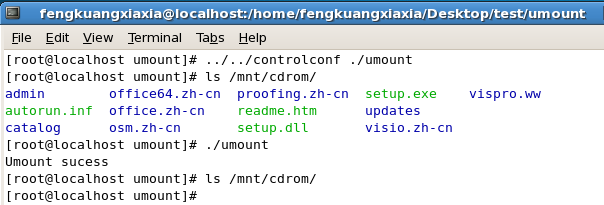


如图发现操作失败

查看dmesg



去除该规则，再次测试



如图成功执行

## 4、测试结论

上述所有测试均成功，可以认为基于LSM 的程序运行权限管理基本实现。

# 第五章 项目小结

从最后完成后的时间点来看，整个项目的实施主要遇到了两次瓶颈，第一次瓶颈是LSM模块的编译，另一次是获取当前进程对应的程序的绝对路径。一开始我试图在Ubuntu上实现，但发现在Ubuntu上做首先编译内核加引导项就更为复杂，另外每次修改LSM模块后需要连带内核一起重新编译，耗时巨大，因此最后还是参照书本编译了一个FC6新内核，当然编译过程中也需要不少由于2.6.18内核源码自身源码导致的问题，但那相对而言还是很好解决的。相较内核编译问题，获取当前进程对应的程序的绝对路径就要难办很多，期间经历了试图从current指针获取到试图从/proc/pid/exe获取再回到current指针获取的曲折，最后还是终于从网上找到了解答再加以改造成功完成了这个功能。突破了这两道大难题后剩下的就好办很多了，虽然从项目进度来看后续我仍然花了近一周完成剩余问题，但其实实际总耗时最后也只有8小时，我只是偷懒把每个功能拆分到每天完成而已。

项目完成了基本所有能想到的功能，LSM中常用的钩子也基本被挂了个遍，但还是有不少不足之处。首先最大的一个不足就是每次运行一段时间就会卡死，这个问题主要的原因是file\_permission这个钩子上，很多操作都要经过这个操作，所以运行时间一长，整个系统就会卡死，这个我也无法测试加强电脑性能能否解决这个问题，只能先搁置。另一个不足之处就在于我能够控制某个程序能够执行哪些操作，但尚未能控制某个程序能对某个文件执行哪些操作，也就是说现在的控制粒度仍属于比较大的，但考虑到有些操作是不存在客体只存在主体的，这样对于规则的保存和修改就会变得异常麻烦，因此这个问题也没有解决。

整个项目做下来最大的感受还是内核模块的编写好困难，首先用c语言来实现就已经很让我这种用惯更高级语言的人头疼了，其次就是在编写内核模块的时候很多常用的库函数都不能使用，这着实很让人抓狂。而最最大的困难无疑是调试了，由于我没什么内核调试经验，因此每次出错调试只能使用printk来先找到出错的位置，然后在针对不同情况使用printk输出不同结果来慢慢调试，比起原来习惯的try、catch或者断点调试复杂了不止一点点。虽然过程非常痛苦，但收获还是非常丰厚的，最大的收获就是对于linux的进程结构体有了非常充分的了解，在实现获取当前进程对应的程序的绝对路径的时候，我翻阅了大量关于linux进程结构体的资料，弄懂结构体中的各个属性是干什么的，即便在找到了解决方案后，对照着结构体要看懂代码仍有不少困难。另外一大收获自然就是对LSM的能挂钩的函数有了很多了解，对于一些常见的操作要经历的钩子函数也有了一定了解，比如使用c语言使用写模式打开一个文件要经历的钩子函数和顺序等等。

系统软件课程设计这门课可谓是大学本科四年生涯中少有的真正完全是信息安全相关的大作业了，因此我觉得这门课的存在意义绝对是毋庸置疑的。但我仍觉得这门课有可改进的地方。首先教程使用的FC6系统是一个相当古老的系统，当然我在做了后也明白使用一个古老的系统确实能减少很多高版本内核带来的问题，但我还是觉得做安全应该紧跟时代，能够有新系统下的安全性方面的框架给大家实践开发。另外就是我感觉结题报告模板中的需求分析和总体设计和原来的需求分析和总体设计的模板有不少对应不上的地方，另外就是需求分析和总体设计模板里自身也有很多前后对应不上的部分，我感觉模板还是应该有一份更标准一点的，前后一致能对应上的比较好。

\*参考资料：

[1]《信息安全技术解析与开发实践》，訾小超等，清华大学出版社

[2]http://www.spinics.net/lists/newbies/msg19531.html