rbac-lsm：基于角色的访问控制安全模块

郝淼

2024-04-19

目录

1 策略 - RBAC

本实验中，用户（User）与角色（Role）是一对一关系，角色与权限（Permission）是多对多关系。因此，用户、角色与权限的关系可以表示为：

$$\text{用户} \rightarrow \text{角色} \rightrightarrows \text{权限}$$

进一步可以将权限分解：权限 = 接受性 + 操作 + 客体

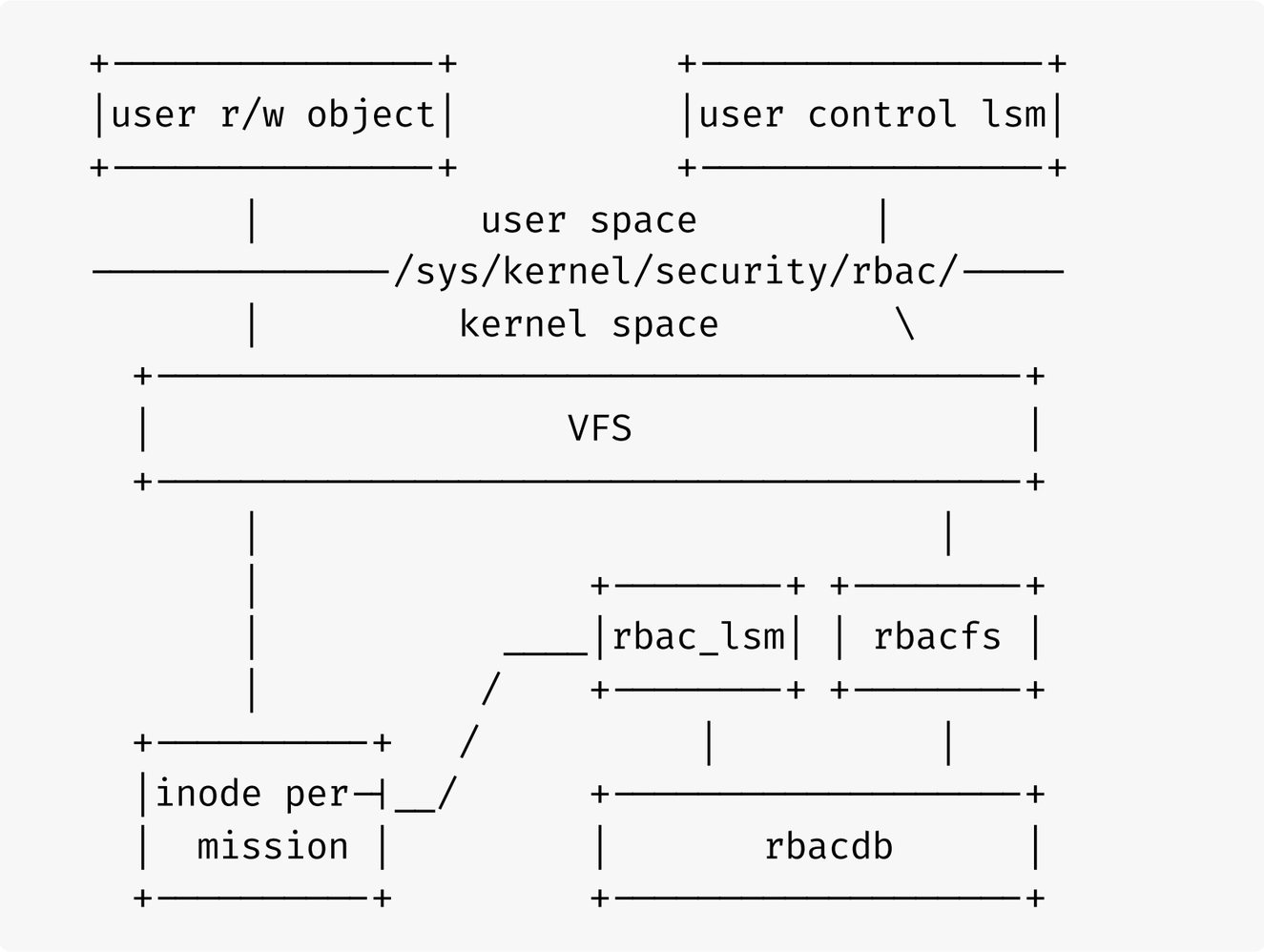
1 策略 - 模型表示

- 可以用多重元组$c \in U \times R \times A \times Op \times O$表示“一个用户具有某个权限”。
- 其中 U 表示用户（User），R 表示角色（Role），A 表示接受性（Acceptability），Op 表示操作（Operation），O 表示客体（Object）。
- 数据库中存储若干这样的五元组，当 U_i 通过 Op_j 访问 O_k 时，检查数据库中是否存在 $(U_i, R_l, A_m, Op_j, O_k)$：
  - 若存在且 A_m 为允许，则允许访问；
  - 若存在且 A_m 为拒绝，则拒绝访问；
  - 特别地，如果在数据库中没有查询到对应的元组，则允许访问。

2 实现 - 实验平台

- 内核版本：Linux 5.14.0-rc2
- 根文件系统：BusyBox
- 虚拟机平台：LoongArch 32 Reduced 架构的 QEMU 平台

2 实现 - 总体架构


user r/w object
user control lsm
user space
/sys/kernel/security/rbac/
kernel space
VFS
rbac_lsm
rbacfs
inode per-
mission
rbacdb

2 实现 - db 层

db 层负责实现内核数据对象的管理，向上提供接口供 fs 层和 lsm 进行使用。rbac-lsm 实现的内核数据对象包括：

- `rbac_user`：用户管理，用 `uid` 标识，每个用户可以关联到一个角色

```
struct rbac_user {
        uid_t     uid;
        struct rbac_role  *role;
        struct list_head  entry;
};
```

2 实现 - db 层

- `rbac_role`：角色管理，用 `name` 标识，每个角色可以关联至多 `ROLE_MAX_PERMS` 个权限

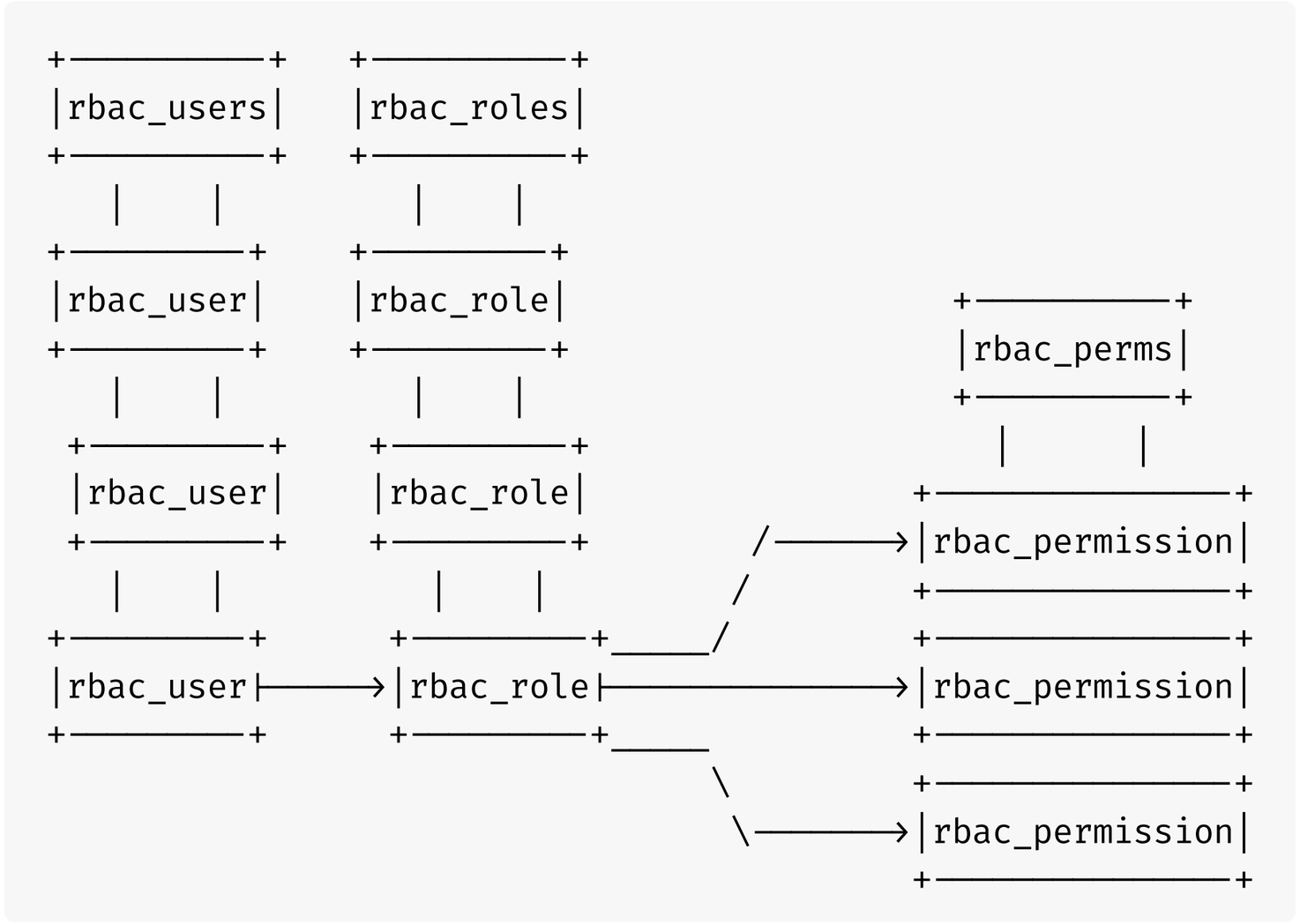
```
struct rbac_role {
        char       name[ROLE_NAME_LEN];
        struct rbac_permission  *perms[ROLE_MAX_PERMS];
        refcount_t    ref;
        struct list_head  entry;
};
```

2 实现 - db 层

- `rbac_permission`：权限管理，用 `id` 标识，每个权限包含接受性、操作和客体三个关键域

```
struct rbac_permission {
        int     id;
        rbac_acc_t    acc;
        rbac_op_t   op;
        rbac_obj_t    obj;
        char      *obj_path;
        refcount_t    ref;
        struct list_head  entry;
};
```

2 实现 - db 层


rbac_users
rbac_roles
rbac_user
rbac_role
rbac_perms
rbac_user
rbac_role
rbac_user
rbac_role
rbac_permission
rbac_permission
rbac_permission

2 实现 - fs 层

fs 层基于 securityfs 机制，在 `/sys/kernel/security/rbac` 下建立了一些列虚拟文件，实现了用户与 lsm 的交互功能：

文件名	属性	说明
`enable`	读写	获取/改变 rbac-lsm 使能
`user`	只读	获取已添加的用户信息
`role`	只读	获取已添加的角色信息
`perm`	只读	获取已添加的权限信息
`ctrl`	只写	添加、删除内核数据对象，改变其间的关系

2 实现 - `enable`

可以通过向 `enable` 文件中写 `0` 或 `1` 改变 rbac-lsm 的使能状态，其中 `0` 表示关闭；`1` 表示开启。

```
$ echo 0 > enable
$ cat enable
rbac: disabled
$ echo 1 > enable
$ cat enable
rbac: enabled
```

2 实现 - `rbac_enable_read`

```
static ssize_t rbac_enable_read(struct file *file, char __user *buf,
                                size_t size, loff_t *ppos)
{
        ...
        sprintf(kbuf, "rbac: %s\n", rbac_enable ? "enabled" : "disabled");
        ...
}
```

2 实现 - `rbac_enable_write`

```
static ssize_t rbac_enable_write(struct file *file, const char __user *buf,
                                 size_t size, loff_t * ppos)
{
        ...
        switch (kbuf[0]) {
        case '0':
                rbac_enable = 0;
                break;
        case '1':
                rbac_enable = 1;
                break;
        default:
                ret = -EINVAL;
        }
        ...
}
```

2 实现 - `rbac_fs_init`

```
static int __init rbac_fs_init(void)
{
        ...
        for (i = RBAC_ENABLE; i < RBAC_FP_TYPE_NUM; i++) {
                rbac_files[i].fp =
                        securityfs_create_file(rbac_files[i].namep,
                                               rbac_files[i].mode, rbac_dir,
                                               NULL, &rbac_files[i].ops);
                if (IS_ERR(rbac_files[i].fp)) {
                        ret = PTR_ERR(rbac_files[i].fp);
                        goto out;
                }
        }
        ...
}

fs_initcall(rbac_fs_init)
```

2 实现 - `user`

读 `user` 以获取目前 rbac-lsm 已知的用户信息。

```
$ cat user
uid: 0 acts as role "admin"
uid: 1000
```

目前 rbac-lsm 已经添加了 2 个用户，其中 `uid` 为 `0` 的用户已经绑定到了名为 `admin` 的角色上； `uid` 为 `1000` 的用户未绑定到任何角色。

2 实现 - `role`

读 `role` 以获取目前 rbac-lsm 中已添加的角色信息。

```
$ cat role
admin
        perm[0] id: 0
guest
```

目前 rbac-lsm 已经添加了 2 个角色，其中名为 `admin` 的角色已经绑定了一个 `id` 为 `0` 的权限；名为 `guest` 的角色未绑定任何权限。

2 实现 - `perm`

读 `perm` 以获取目前 rbac-lsm 中已添加的权限。

```
$ cat perm
[0]: deny write on /init
[1]: accept read on /
[2]: deny read on /init
[3]: deny write on /
```

目前 rbac-lsm 已经添加了 4 条权限，其中 `id` 为 `0` 的权限表示不允许写 `/init`；`id` 为 `1` 的权限表示允许读 `/`；`id` 为 `2` 的权限表示不允许读 `/init`；`id` 为 `3` 的权限表示不允许写 `/`。

2 实现 - `ctrl`

向 `ctrl` 写命令以实现对内核数据对象的修改，支持的命令包括：

- [add|remove] user UID：[添加|删除]用户，其 `uid` 为 UID
- [add|remove] role NAME：[添加|删除]名字为 NAME 的角色
- [add|remove] perm ACC OP OBJ：[添加|删除]接受性为 ACC，操作为 OP，客体为 OBJ 的权限
- (un)register UID NAME：将 `uid` 为 UID 的 user 和名字为 NAME 的 role （解）绑定
- bind ID NAME：将 id 为 ID 的权限绑定到名字为 NAME 的 role 上
- unbind RID NAME：将 id 为 ID 的权限到名字为 NAME 的 role 的绑定解除

2 实现 - `ctrl`

目前支持的参数可选值为：

名称	可选值
UID	任意非负整数
NAME	任意字符串
ACC	`a` 表示接受；`d` 表示拒绝
OP	`r` 表示读；`w` 表示写
OBJ	任意绝对路径
RID	将权限绑定到角色后，权限在角色上的 id

2 实现 - lsm

lsm 通过将权限检查钩子挂在 `inode_permission` 上以接受/拒绝某次访问：

```
static int rbac_inode_permission(struct inode *inode, int mask)
{
        ...
        if (rbac_enable == 0)
                goto out;

        cred = current_cred();
        uid = from_kuid(cred->user_ns, cred->euid);
        ret = rbac_check_access(uid, inode, mask);
out:
        return ret;
}

static struct security_hook_list rbac_hooks[] = {
        LSM_HOOK_INIT(inode_permission, rbac_inode_permission),
};
```

```
int rbac_check_access(uid_t uid, struct inode *inode, int mask)
{
        ...
        /* 通过 `uid` 查询用户信息，进一步获取当前用户的角色 `role` */
        user = rbac_get_user_by_uid(uid);
        role = user->role;
        /*
         * 遍历 `role` 中包含的每一条权限，根据 `inode` 和 `mask`
         * 找到客体和操作与本次访问相同的权限
         */
        for (i = 0; i < ROLE_MAX_PERMS; i++) {
                perm = role->perms[i];
                if (perm != NULL && perm->obj == inode) {
                        if (mask & MAY_READ && perm->acc == ACC_DENY && perm->op == OP_READ) {
                                /* 拒绝访问 */
                                ret = -EPERM;
                                goto out;
                        }
                }
        }
        /* 接受/拒绝访问 */
out:
        return ret;
}
```

3 实验结果

3 实验结果 - 获取 rbac-lsm 初始状态

```
$ cat user && cat role && cat perm
uid: 0 acts as role "admin"
admin
        perm[0] id: 0
[0]: deny write on /init
```

此时包含一个用户，它将 `uid` 为 `0` 的用户（即 root）绑定到了角色 `admin` 上；`admin` 绑定了 `id` 为 `0` 的权限；该权限禁止写 `/init`

3 实验结果 - 检查 rbac-lsm 是否工作

```
$ cat enable
rbac: enabled
$ cat /init
#!/bin/sh
mount -t devtmpfs none /dev
...
exec /bin/busybox
$ echo "add a new line" > /init
-sh: can't create /init: Operation not permitted
```

此时可以读 `/init` 但是无法写入内容

3 实验结果 - 添加并修改权限

添加一条对 `/init` 读的禁止权限，并将原来的权限替换为这条新的权限

```
$ echo add perm d r /init > ctrl && echo unbind 0 admin > ctrl && echo bind 1 admin > ctrl
$ cat user && cat role && cat perm
uid: 0 acts as role "admin"
admin
        perm[0] id: 1
[0]: deny write on /init
[1]: deny read on /init
$ cat /init
cat: can't open '/init': Operation not permitted
$ echo "add a new line" >> /init
```

`admin` 具有权限的 `id` 更新为了 `1`，此时无法读 `/init`，但可以写 `/init`

3 实验结果 - 将权限改回

解绑 `admin` 的禁止读权限，改为禁止写 `/init`

```
$ echo unbind 0 admin > ctrl && echo bind 0 admin > ctrl
$ cat user && cat role && cat perm
uid: 0 acts as role "admin"
admin
        perm[0] id: 0
[0]: deny write on /init
[1]: deny read on /init
$ cat /init
#!/bin/sh
mount -t devtmpfs none /dev
...
exec /bin/busybox
add a new line
```

此时又可以对 `/init` 进行读操作了，并且其中包含了刚刚写入的内容

3 实验结果 - 客体为目录

如果客体变为目录，则实验结果体现为能否读取该目录下包含的文件，或在该目录下创建文件，结果在此略过。

谢谢！