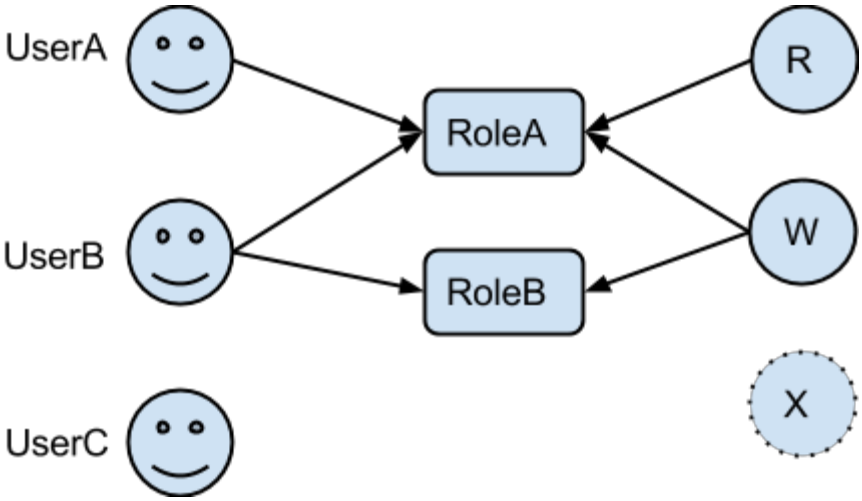
CSE 509, 2014 Fall
Kelong Wang 109600226
Sep 8, 2014

SBRACK
Role-Based Access Control LSM

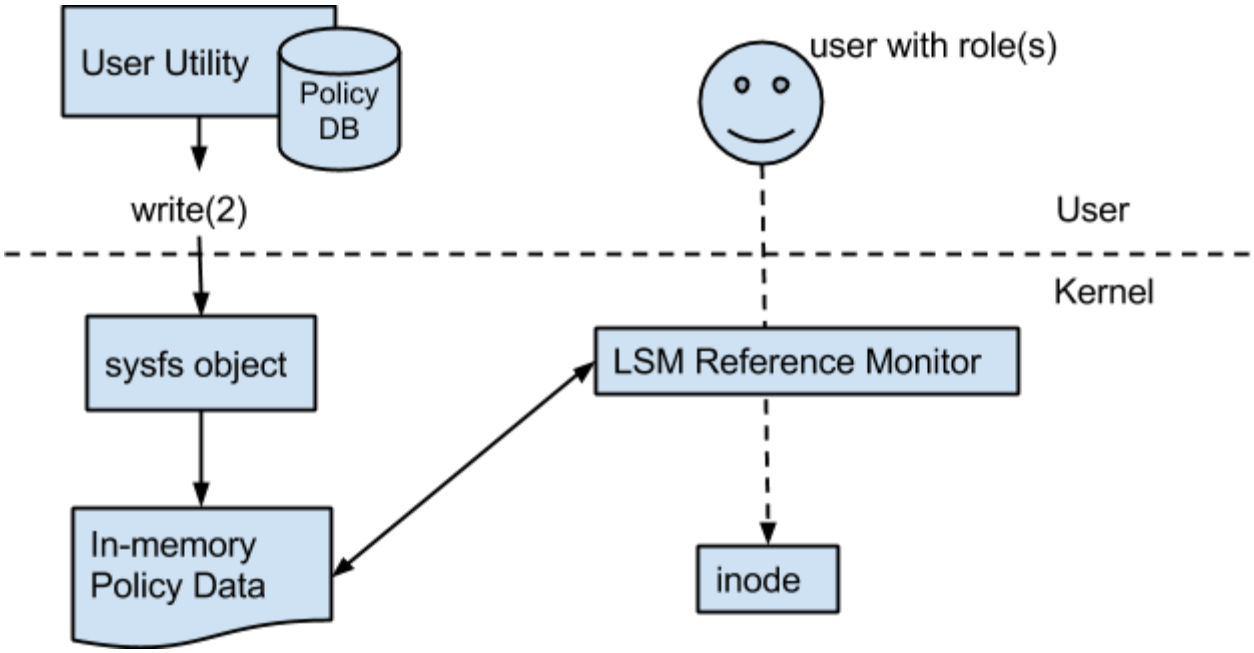
Design



Assumption

- SBRACK only checks the access of inode operations by the users who belong to "*stonybrook*" group on Linux. So our LSM does not interfere with other normal user (e.g., *root*)'s operation. Be default, when a user joins "*stonybrook*" group without being assigned roles, this user has no permission to do anything (to inode).
- For simplicity, each role in SBRACK can have three types of permission: *read*(R), *write*(W) and *execute*(X). Combination of these permissions is allowed. A user can be assigned with multiple roles. The user will pass the permission check as long as one of its role is qualified.
- In order to test *read* & *write* permissions with system utilities such as *ls*, *cat* and *echo*, the *execute* permission is not checked at this time[1]. Login may also require such permission.
- We categorize the mediated inode operations into read operation and write operation. For example, *inode_unlink()* is a write operation and *inode_getattr()* is a read operation. One exception is *inode_permission()* which will be checked with its *mask* argument passed in.

[1] This check can be enabled in main.c : fs_mask_to_sbrack_mask(), if you want.

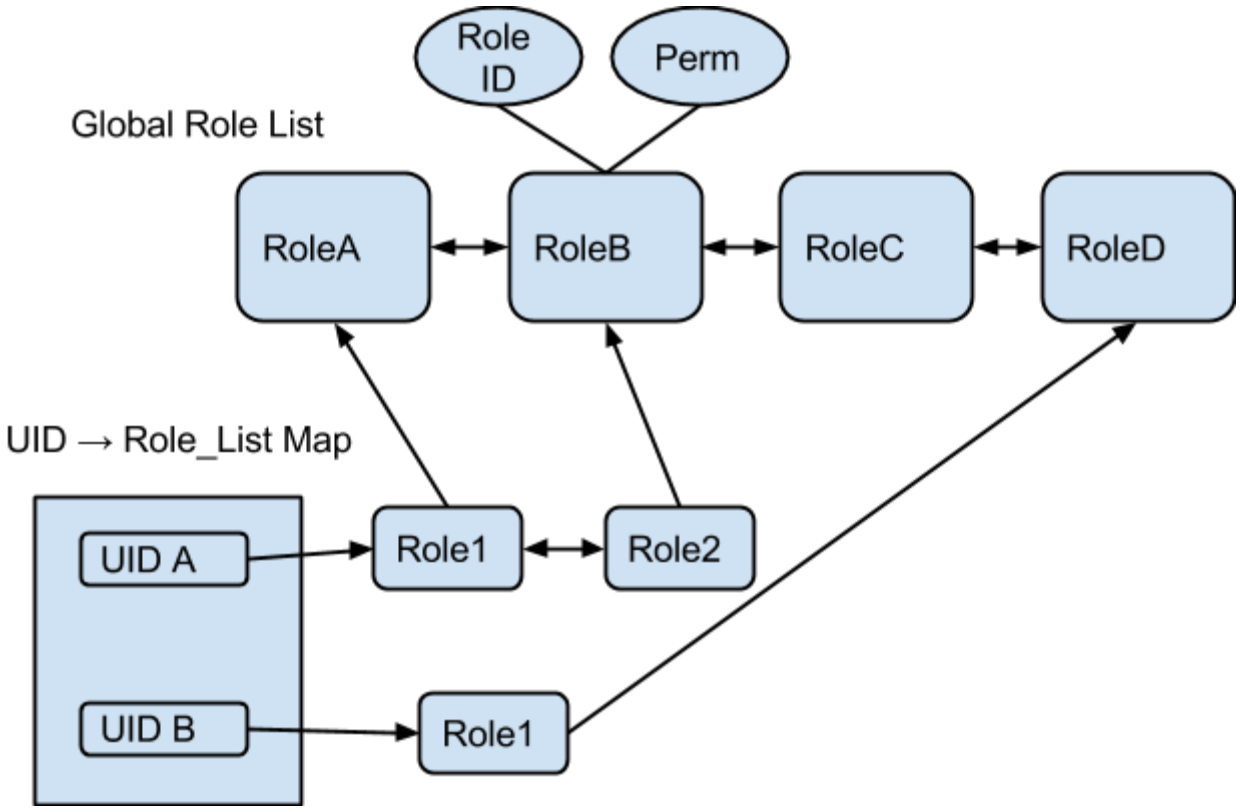


Component and Implementation

- In kernel LSM module (*main.c*)
  - We patched Linux kernel to make LSM module dynamically loadable.
  - By registering the corresponding inode hooks, VFS will call back SBRACK authentication functions.
  - Authentication is quite straightforward:
    - Get current user's effective UID and its role list, also make sure its supplementary groups contain "*stonybrook*".
    - Check the operation's required permission (say *inode_unlink()* requires *write* permission) against each role in its role list until satisfied.
  - In (kernel) memory data structure is figured below.
- API based on sysfs[2] (*kmanage.c*)
  - With sysfs, SBRACK will create two objects *"/sys/kernel/sbrack/role"* and *"/sys/kernel/sbrack/user"*. By writing to these virtual files administrators are able to communicate with kernel to set the policy.
  - Invalid argument (e.g., deleting a role that does not exist) will result in EINVAL.
- Userspace control utility (*sbrack_ctl.py*)
  - We provide a user-level program to manage the policy data. It maintains a local policy database (serialization of python objects simply for proof of concept). When a command is committed to kernel, a policy copy is also recorded locally and thus persistently.

[2] https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt

- This program is also a wrapper writing commands to sysfs objects in order to notify kernel the changes of policy.
- If local policy data is not synchronized with kernel policy data (e.g., after system reboots), this utility can load local data into kernel. See “Usage” section below for details.



Usage

install

```
# cd linux-3.14.17/security/sbrack
(apply make-LSM-loadable.patch, recompile kernel and reboot to allow
dynamic LSM)
# make
# insmod sbrack.ko
# chmod +x sbrack_ctl.py
# ./sbrack_ctl.py reset_db      # reset local data policy
```

(or use ./install_module.sh instead to load the module)

add, del, edit & list role

```
# ./sbrack_ctl.py add_role staff NONE
# ./sbrack_ctl.py add_role student RWX
# ./sbrack_ctl.py del_role student
# ./sbrack_ctl.py add_role student R
# ./sbrack_ctl.py edit_role staff RW
# ./sbrack_ctl.py list_role
[3]student      R
[1]staff        RW
```

(*dmesg* is useful for debugging if you need to confirm the success of command.)

add, del & list user

```
# groupadd --gid 20000 stonybrook  # see assumption 1
# useradd bob --groups stonybrook --home-dir /home/bob --create-home
# useradd david --groups stonybrook --home-dir /home/david
--create-home
# ./sbrack_ctl.py add_user bob
# ./sbrack_ctl.py add_user david
# ./sbrack_ctl.py del_user david
# ./sbrack_ctl.py add_user david
# ./sbrack_ctl.py list_user
[1004]bob       roles:
[1005]david     roles:
```

assign role(s) to user

```
# ./sbrack_ctl.py assign bob student
# ./sbrack_ctl.py list_user
[1004]bob       roles:student
[1005]david     roles:
# echo "hello world" > /tmp/test      # create a test file
# chmod 777 /tmp/test
```

(login as bob via `ssh`. Here `sudo su -` and `login` are not working with limited privilege.)

```
(bob)$ touch foo
touch: cannot touch ‘foo’: Permission denied
(bob)$ cat /tmp/test
hello world
(bob)$ echo 'try to edit' >> /tmp/test
cannot create /tmp/test: Permission denied

# ./sbrack_ctl.py assign david student
# ./sbrack_ctl.py assign david staff
# ./sbrack_ctl.py list_user
[1004]bob       roles:student
[1005]david     roles:student,staff
(david)$ touch foo
(david)$ cat /tmp/test
hello world
(david)$ echo 'try to edit' >> /tmp/test
(david)$ cat /tmp/test
hello world
try to edit
```

revoke role of user

```
# ./sbrack_ctl.py revoke david staff
# ./sbrack_ctl.py revoke bob student
# ./sbrack_ctl.py list_user
[1004]bob       roles:
[1005]david     roles:student
(david)$ touch foo
touch: cannot touch ‘foo’: Permission denied
(bob)$ cat /tmp/test
cat: Permission denied     (even have no access to cat program)
```

reload policy to kernel

```
# ./sbrack_ctl.py init_kernel_from_db
```

This doesn’t reset kernel data. So it should be only used for a newly loaded kernel module.