CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Protection

System must protect resources against both accidental and malicious misuse

Examples include reading data of other users, running certain programs, deleting files owned by others, ...

Requires some way of knowing who should have access to what

Important to distinguish between policy and mechanism

Policy is rules we wish to enforce -- e.g., only Alice should be able to delete files in her home directory

Mechanism is way we enforce rules -- e.g., maintaining list of users allowed to access certain file

Ideally, mechanism should be flexible enough to implement various policies

Most basic way to protect access in Linux is setting file permissions

Each file has one owner and one group

User may belong to more than one group

Different permissions can be specified for

- user ("owner")
- group
- other

For each of these, permission may be granted to

- read
- write
- execute

Any combination of read/write/execute permissions may be granted

```
$ ls -l
total 0
-rw-rw-rw- 1 nate nate 0 Dec  8 00:23 public.txt
-rw------- 1 nate nate 0 Dec  8 00:23 secret.c
-rwxrwx--- 1 nate team 0 Dec  8 00:24 team_only.sh
```

What if we need to be more precise?

Consider following scenario:

- I own code that anyone in my project group can modify
- Nobody else should see code

This is easy -- separate permissions for project group and rest of the world

In same scenario, visitor comes to inspect project

Want just this visitor to be able to *read* code, but not modify

This is not possible with simple permissions described

Finer-grained control possible using access-control list (ACL)

In addition to standard owner/group/other permission, add user-specific permissions for any user on system

In Linux, managed with `getfacl` and `setfacl`

Rules for owner, group, and other always match simple permissions

```
$ getfacl team_only.sh
# file: team_only.sh
# owner: nate
# group: nate
user::rwx
group::rwx
other::---
```

Exceptions for specific users or groups can be added

```
$ setfacl -m u:visitor:r team_only.sh
```

```
# file: team_only.sh
# owner: nate
# group: nate
user::rwx
user:visitor:r--
group::rwx
mask::rwx
other::---
```

Files with ACL denoted with + in `ls -l` output

```
$ ls -l
total 0
-rw-rw-rw-  1 nate nate 0 Dec  8 00:23 public.txt
-rwx------  1 nate nate 0 Dec  8 00:23 secret.c*
-rwxrwx---+ 1 nate nate 0 Dec  8 00:24 team_only.sh*
```

Another important consideration is `setuid`

Consider: what if non-root user needs to run program that can change system state?

Canonical example is `passwd` -- program to change user's own password

To do its job, `passwd` needs to modify passwords file, but this is not something normal user should be able to do

`setuid` allows user to run specific program with UID set to owner of that program

Gives temporarily escalated permissions

In case of `passwd`, user can modify password file when running `passwd` executable

Can be recognized with `ls -l`

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 63640 Sep  7 09:42 /usr/bin/passwd
```

`setgid` (group id) also exists and works similarly

`setuid` is very useful -- allows user to change system state, but only in controlled way

Very dangerous if program with `setuid` permission poorly written

If some input to `passwd` could, for example, trick it into running `/bin/bash`, user would have root access to entire system