CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 12th, 2011

Access control

Topics to be covered in the next classes

- UNIX access control
- Windows access control
- ACM (access control matrix), RBAC (role based access control), Chinese Wall
- Capabilities
- Confused deputy problem
- Web security model
- Android security model

UNIX access control

Forty years back, UNIX ran mainly on mainframes. It was a multi-user system. A major goal was to **protect users from each other**.

Access control model:

- Each user is assigned a user ID (**uid**)
- Each user is assigned one or more group IDs (**gid**)
- Each file is **bound to a single uid**, representing its **owner**, and **single group id**, representing its **group**.
- A file has permission for its **owner**, its **group** and for the **other people/users**
    - There are three permission flags: **read**, **write**, **execute**
    - There are actually some extra flags too (later)

Example: `i_like_to_move_it_move_it.lol` $\frac{alinush}{rwx}, \frac{students}{rw-}, \frac{other}{r--}$

What does it mean?

- User `alinush` is the owner.
- He has given himself read, write and execute permissions for the file.
- He has set the file's group to `students`, with read and write permissions such that every user in the group can read and write the file, but they cannot execute it.
- For users that are neither `alinush` nor are they in the `students` group, the permission are set to read-only, such that they can never write or execute the file.

How are file permissions enforced by the OS?

For files

When a file is read, written or executed by user $alinush$, his UID is checked against the file's owner UID and against its group GID. If $alinush$'s UID matches one of these, then the permissions for what was matched are applied (either owner permissions or group permissions). Otherwise, the permissions set for the *others group* are applied.

```
if (process.uid === file.uid)
{
    check against owner permissions
}
else (file.gid in process.gids)
{
    check against group permissions
```

```
}
else
    check against other permissions
```

Note: This scheme can work against a user if he is added to a "denied" group for a file whose permissions are:

$$\frac{owner}{rwx}, \frac{group}{---}, \frac{other}{rwx}$$

For directories

- The r read permission allows an user to list the contents of a directory
- The w write permission allows an user to modify the content of directory
  - Creating files, deleting files, renaming them, moving them to another directory for which the user has w permission
- The x execute permission allows an user to look up the files or subdirectories in a directory
  - A user without the r permission and with the x permission can only query a directory to see if it contains a particular file. He can never list all the files in the directory unless he does a brute-force search to find all the files with names shorter than a specific length.
- The sticky bit s is set for a directory in which every user can only delete his own files
  - Mainly used for `/tmp` directory

Who can change ownership (`chown`)?
No one other than the **`root`** account with uid 0 can change a file's owner. If you're the owner of a file you can only change the group GID of the file.

Why? Allowing the owner to "give" the file to someone else could result in him being able to surpass the system quota enforced for users. He can just give big files to other users who will not even be aware of this, tricking the OS into thinking his quota is fine.

Access control lists

Access control lists are the most intuitive way to grant permissions: each file has a list of users and groups and their associated permissions for that file

Users	Groups
`alinush:rw`	`professors:rw`
`stonethumb:rw`	`students:R`

You can also have default entries such as `default:users` and `default:group`, which are the default ACLs put into newly created files and subdirectories into this directory.

UNIX works with both `chown`, `chmod` permissions and with ACLs. Somehow they were mixed together carefully so that they work together. This way you do not have to pick one scheme.

setuid and setgroupid programs

Suppose we had a print server running on a UNIX system and you want to enforce a print quota to your users. The print server keeps track of billing information and is connected to the printer.

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 12th, 2011
You could implement this with various permission schemes and user programs.

Scheme 1

`/dev/lpr` has permissions $\frac{root}{rw-}, \frac{root}{rw-}, \frac{other}{---}$ such that only root can use the `lpr` printing device

Then there is a program `/usr/bin/printfile` with permissions $\frac{root}{rwx}, \frac{root}{rwx}, \frac{other}{r-x}$

Usage: `printfile filename`

- Opens up the specified file, reads it and sends it to `/dev/lpr` for printing

```
// Scrap from the printfile program code
int main(int argc, char ** argv)
{
    // ...
    FILE * file_to_print = fopen(argv[0], "r");
    FILE * printer = fopen("/dev/lpr", "r");
    // ...
}
```

Suppose `alinush` executes `printfile /home/alinush/toprint.ps` with permissions $\frac{alinush}{rw-}, \frac{students}{---}, \frac{other}{---}$

A new `printfile` process will be created with `alinush`'s UID. The `.ps` file will be read since the permission allow it (`alinush` is the owner with `rw` permissions), but the `/dev/lpr` device won't open because `alinush` doesn't have permissions.

Note: Solutions to this problem will be covered later in the semester.

How to fix this?
We can use the `setuid` bit. An executable with the `setuid` bit enabled runs as its owner. So the `printfile` program will always run as `root` even if another user executes it.
Now our program `/usr/bin/printfile` will have permissions $\frac{root}{rws}, \frac{root}{rwx}, \frac{other}{r-x}$

Catch: The `setuid` bit can only be set by the owner of the file.
Also: There is also a `setguid` bit. An executable with the `setguid` bit enabled runs with its GID set to the file's group.

`setuid` programs **bind together privilege and policy**. The `printfile` program has the privilege to run as `root` and do anything on the system but its code will enforce the policy of `printfile` to just print stuff and not do anything crazy.

Problem: What if someone executes `printfile` *`secretfile`*? Turns out a user can print any file on the system since the `printfile` program executes as `root` and can thus read any file.

Scheme 2

The least privilege principle

`/dev/lpr` has permissions $\frac{root}{rw-}, \frac{print\_allowed}{rw-}, \frac{other}{---}$ such that users in the `print_allowed` group can use the `lpr` printing device.

The `/usr/bin/printfile` client program has permissions $\frac{root}{rwx}, \frac{print\_allowed}{rws}, \frac{other}{r-x}$

Because the `setguid` bit is set, whenever a user executes `printfile`, the GID for the `printfile` process will be set to the `print_allowed` group. This will allow the process to open `/dev/lpr` which can be opened by processes with the GID set to `print_allowed`.

Scheme 3

What if we had an `lpr-server` program that was always running in the background and had access to `/dev/lpr`. In addition, we also have a program `lpr-client` that all users can run and has no special privileges, it just opens a TCP/IP socket to the server and then sends the file over. Somehow, the server needs to authenticate the client, so that users cannot impersonate other users and use their quotas. There are a lot of ways of doing this authentication.

Alternative to `setuid` programs: you have a server program that authenticates client program requests ("economy of mechanism" and "separation of privileges").

Economy of mechanism: keep it simple (stupid)

Separation of privileges: `printfile` needed privilege from two sources, one to open the file to be printed, one to open the printer device. Now the `lpr-client` program can read any user files, and the `lpr-server` program can access the printer device.

Windows permissions

Windows uses ACLs to enforce access control. In Windows, an ACL can have allow and deny entries and this has interesting consequence in terms of how permissions are evaluated.

User name	Permission
alinush	r
students	deny
profs	rw

Should `alinush` be able to read the file if he is part of the `students` group? It turns out that in Windows the order of the access control entries (ACEs) matters when permissions are evaluated inside an ACL. In this case, Windows would give `alinush` permission, since it would first run into the ACE that allows `alinush` to read the file and it would stop right there.

Extra: You can look more into how Windows access control works here:

- How DACLs control access to an object
- Order of ACEs in a DACL
- Access control
- Access control model

In Windows, a process has a user ID and group ID credentials, but each one can have modes:

- ignore (this credential does not contribute to your ACL decision at all)
- normal (this credential can be matched against a positive ACL or a deny ACL)
- deny only (it can only be used to match ACL deny rules)