Learning Objectives: Differentiating Access

Learners will be able to…

- **Explain what ACL is**
- **Describe how user permissions work on UNIX-systems**
- **Tell about differentiating access on user(different abilities for different user roles) and application level (different database connections for different role with different rights)**
- **Write an application with ACL**

Access Control List

Access Control List (ACL) - an access control list that determines who or what can access an object (program, process or file), and what operations are allowed or prohibited to be performed by the subject (user, user group).

First introduced in OS Multics in 1965, since then, multiple implementations have become widespread in almost all types of programs with distributed access.

There are a lot of types ACLs:

- Filesystem ACLs - filter access to files and/or directories. Filesystem ACLs tell operating systems which users can access the system, and what privileges the users are allowed
- Networking ACLs - filter access to the network. Networking ACLs tell routers and switches which type of traffic can access the network, and which activity is allowed
- Active Directory ACLs - Microsoft's Active Directory service implementation
- SQL implementations - ACL algorithms have been ported to SQL and to relational database systems

A file system ACL is a table that tells the computer's operating system the access rights a user has to a particular object, including a single file or file directory. Each object has a security property that associates it with its own access control list. The list contains an entry for each user with access rights to the system.

Linux File Permissions

In Linux, file permissions, attributes, and ownership control the access level that the system processes and users have to files. This ensures that only authorized users and processes can access specific files and directories.

The Linux filesystem gives us three types of permissions.
- **User** (or user owner)
- **Group** (or owner group)
- **Other** (everyone else)

Three file permissions types apply to each class of users:

- The **r**ead permission.
- The **w**rite permission.
- The e**x**ecute permission.

To view file permissions you could type in terminal: `ls -al file_name`. Output will be something like:

```
-rw-r--r-- 1 codio codio 1421 Oct 28 20:23 README.md
```

The first character indicates the file type. It can be a regular file `(-)`, directory `(d)`, a symbolic link `(l)`, or other special types of files. The following nine characters represent the file permissions, three triplets of three characters each. The first triplet shows the owner permissions, the second one group permissions, and the last triplet shows everybody else permissions.

Example above (`-rw-r--r--`) show, that owner could read and write into the file, the group and others - only read.

Effect of Permissions on Files

Read
`-` The file is not readable. You cannot view the file contents.
`r` The file is readable.

Write
`-` The file cannot be changed or modified.
`w` The file can be changed or modified.

Execute
`-` The file cannot be executed.
`x` The file can be executed.

You could change file permission(if you have permissions to do it) via `chmod` command.

```
chmod [OPTION]... MODE[,MODE]... FILE...
```

The format of a symbolic mode is `[ugoa...][[-+=][perms...]...]`, where perms is either zero or more letters from the set rwxXst, or a single letter from the set ugo. Multiple symbolic modes can be given, separated by commas.

A combination of the letters ugoa controls which users' access to the file will be changed: the user who owns it (u), other users in the file's group (g), other users not in the file's group (o), or all users (a). If none of these are given, the effect is as if (a) were given, but bits that are set in the umask are not affected.

The operator `+` causes the selected file mode bits to be added to the existing file mode bits of each file; `-` causes them to be removed; and `=` causes them to be added and causes unmentioned bits to be removed except that a directory's unmentioned set user and group ID bits are not affected.

Examples:

Remove writable flag for all users

```
chmod a-w filename
```

Add executable flag for all users

```
chmod a+x filename
```

Owner of file could be changed via `chown` command.

```
chown [OPTION]... [OWNER][:[GROUP]] FILE...
```

Examples:

Change file owner. Note: user must be exists in the system!

```
chown user file_name
```

File `change-me` in folder `file-permissions` have `rwx` permissions for all users, change it to only read permissions for all users. You could check files and permissions with `ls -al` command.

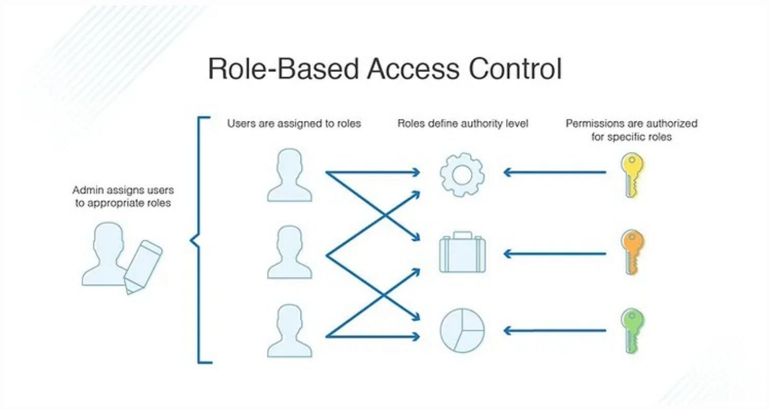
Role-based access control

Access control theories have their origins in the military. There are two types on which the other types are built: **DAC** and **MAC**.

Discretionary Access Control (DAC) - The owner of a protected system or resource sets policies defining who can access it. It is mostly how file permission in Unix works - if you own a file in Unix, you can set the file permissions to grant other people read or write or execute access to it.

Mandatory Access Control (MAC) - differentiation of access of subjects to objects, based on the assignment of a sensitivity label for information contained in objects, and the issuance of official permissions (permission) to subjects to access information of this level of confidentiality. You do not have ability to do something doesn’t let you share that ability with others.

Role-based access control (RBAC)
Role-based access control (RBAC), also known as role-based security, is a mechanism that restricts system access. It involves setting permissions and privileges to enable access to authorized users. Most large organizations use role-based access control to provide their employees with varying levels of access based on their roles and responsibilities.
RBAC is a subset of MAC. It’s a particular type of MAC, but it’s more concrete, so it’s easier to talk about and work with.


Role-Based Access Control
Admin assigns users to appropriate roles
Users are assigned to roles
Roles define authority level
Permissions are authorized for specific roles

rbac

RBAC is foundation of all role based access models. As seen in above diagram, it has a number of roles depicted as a capital N. Each role has a set of permissions associated with roles and in turn with permissions.

The main entities that are modelled in the core RBAC are:

User
The individuals which are identified with a unique user identification (UID).

Role
A named job function (indicates the level of authority).

Permission
Equivalent to access rights a role has on application.

Session
A mapping between a user and a set of roles to which the user is assigned in the context of a working time, that is the duration in which they access your application.

Object
A system resource that requires permission to access.

Operation
Any action in the protected network that is performed by a user.

RBAC vs ACL

For most business applications, RBAC is superior to ACL in terms of security and administrative overhead. ACL is better suited for implementing security at the individual user level and for low-level data, while RBAC better serves a company-wide security system with an overseeing administrator. An ACL can, for example, grant write access to a specific file, but it cannot determine how a user might change the file.

Access Control in NodeJs App

There are many libraries for access control, depending on the programming language and purpose.
For NodeJs we will take a look into accesscontrol library - it allow to implement Role-Based Access Control.

For uset it you need to create instance of access controll class.

```
const AccessControl = require('accesscontrol')
const ac = new AccessControl()
```

After it you need to describe required roles:

1. Create user role

```
ac.grant('user') // we define a new role
  .createOwn('book') // grant permissions to book resource
  .deleteOwn('book')
  .readAny('book')
```

2. Create admin role - extend user role with new permissions:

```
ac.grant('admin') // we define a new role
  .extend('user') // by extending user role
  .updateAny('book') // and add new permissions to make it admin
  .deleteAny('book')
```

3. now we could check access of role to resource

```
let permission = ac.can('user').createOwn('book')
console.log(permission.granted);    // —> true

permission = ac.can('admin').updateAny('book')
console.log(permission.granted);    // —> true

permission = ac.can('user').updateAny('book')
console.log(permission.granted);    // —> false
```

Access Control Playground

Here a playground for check access to resources. The code contains two users: `admin` and `user` and resource `book`. You could check if specific user could perform operations: `createOwn`, `readOwn`, `updateOwn`, `deleteOwn`, `createAny`, `readAny`, `updateAny`, `deleteAny`.

Try to add new role: `moderator` who could `updateAny` resource.