INF226 – Software Security

Håkon Robbestad Gylterud

2019–09-09

The second mandatory assignment

Goal: Analyse the security of a web-application

Method:

- Describe security model
- Security tools (SonarQube, ZAP)
- Manual inpection

Today

- Quick demonstration of the tools.
- Models of access control

Zed Attack Proxy

SonarQube

Access control

Access control

On a multi-user system **access control** decides which users can

- read/write to objects (files, databases tables, $\cdots$)
- perform operations (start processes, allocate memory, $\cdots$)
- grant/revoke access

Mandatory vs. discretionary

In a *Mandatory Access Control* (MAC) system, the access control policies are fixed by a central authority.

In a *Discretionary Access Control* (DAC) system, a user who has access to an object can specify permissions for it or transfer acess to another actor.

Examlpes of mandatory access control

Modern operating systems have mandatory access control on resources such as CPU, memory and storage.

Examlpes of mandatory access control

Modern operating systems have mandatory access control on resources such as CPU, memory and storage.

In additon there are systems for introducing more MAC based secruity:

- SELinux
- Linux Security Modules (AppArmor)
- Mandatory Integrity Control on Windows (Extending ACLs)
- Language based mechanisms (e.g. Java Security Manager)

Examples of discretionary access control

- File systems
- E-mail
- WIFI passwords
- . . .

Access control models

Mathematical models of access control

- Bell—LaPadula model (1973): Security levels “Top sectret”–“Unclassified”.
- Biba model (1975): Focussed on data integrity.
- Graham—Denning model: Concerned with object creation and ownership

Access control models

We will focus on three common access control models:

- Access control lists
- Rôle based access control
- Capability based access control

Access control lists

In a system with access control lists, permissions are assigned to objects:

- Each object has a list of permissions assigned to different users.

Access control lists

In a system with access control lists, permissions are assigned to objects:

- Each object has a list of permissions assigned to different users.

Typically (but not always), the access control list specifies an owner og the object.

Access control lists

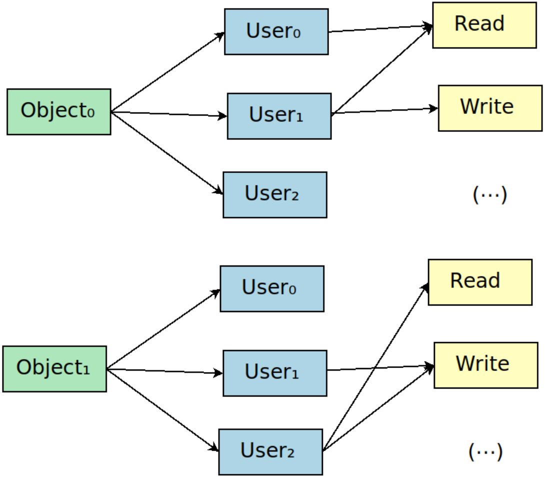

Figure 1: Access control lists

Example

In Unix-like systems:

- Subjects: processes
- Objects: files, sockets, processes, $\cdots$

Permissions are structured according to users and groups.

Users and groups

The system is divided into users and groups, identified by numbers:

- User ID (UID)
- Group ID (GID)

Special UID: 0 (root). Can ignore most permission restrictions.

Processes

When a program is run it is assigned a **Process ID** (PID).

- Processes are prevented from directly accessing each other's memory.

In addtion, the process is assigned to a specific UID and GID. (Usually inherited from parent process)

Files

Every file has:

- Owner UID
- Owner GID

File permissions

There are three kinds of file permissions:

- Read
- Write
- Execute

This gives a matrix

W\P	read	write	execute
user	□	□	□
group	□	□	□
other	□	□	□

Example

W╲P	read	write	execute	octal
user	1	1	0	$6 = 4^1 + 2^1 + 1^0$
group	1	0	0	4
other	0	0	0	0

Commandline: `chmod 640 filename`

Executables (SUID/SGID)

In additions to permissions, there are special flags:

- Set UID (**SUID**):
  - When executed, the UID of the process is set to file owner.
- Set GID (**SGID**):
  - When executed, the GID of the process is set to file group.
- Sticky-bit:
  - File can only be renamed/deleted by root or owner

SUID usage

SUID bits can be used to give a process higher or lower priviledges.

Warning: If a user can trick root into owning a specially crafted SUID program, user gains admin priviledges.

Directory permissions

Directories also have read, write and execute permissions.

- Read: list the content of the directory
- Write: create, rename and delete from directory
- Execute: Entering directory and access files

Question

Why can only root change ownership of a file?

Question

A user, `bob`, wants to share his file `/home/bob/secret` with user `alice`, but does not want to give any other users access.

How can he arrange this?

Rôle based access control

In a **rôle based access control** (RBAC) system, a set of *rôles* abstract the permissions from users.

Rôle based access control

In a **rôle based access control** (RBAC) system, a set of *rôles* abstract the permissions from users.

We have sets `P` (permissions), `R` (roles) and `U` (users):

- `RolePerm` $\subseteq$ `R` $\times$ `P` specifies permissions for rôle.
- `UserRoles` $\subseteq$ `U` $\times$ `P`

Rôle based access control

In a **rôle based access control** (RBAC) system, a set of *rôles* abstract the permissions from users.

We have sets `P` (permissions), `R` (roles) and `U` (users):

- `RolePerm` $\subseteq$ `R` × `P` specifies permissions for rôle.
- `UserRoles` $\subseteq$ `U` × `P`

Actions are always *performed by a rôle*. To change rôle user must reauthenticate.

Rôle based access control

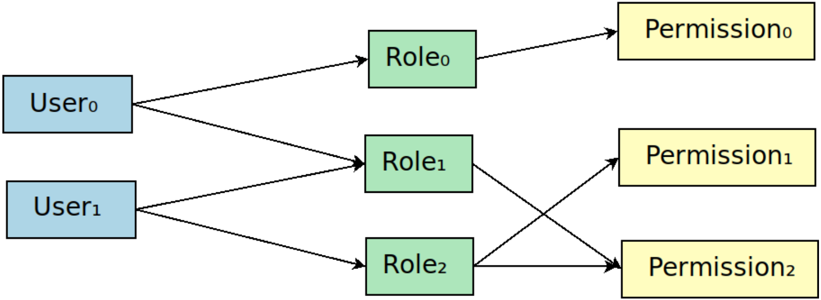

Figure 2: Rôle based access control

Example of RBAC

- `U = {alice,bob}` and
- `R = {doctor, patient}`
- `P = {writePerscription, withdrawMedicine}`
- `RolePerm = {(doctor,writePerscription), (patient, withdrawMedicine)}`
- `UserRoles = {(alice,doctor),(bob,patient),(alice,patient)}`

Capability based access control

In **cabability based access control**, users have capabilities.

A *capability* consists of:

- A **reference** to an object
- A set of **permissions** for that object

A capability is used **whenever a resource is accessed**.

Capability based access control

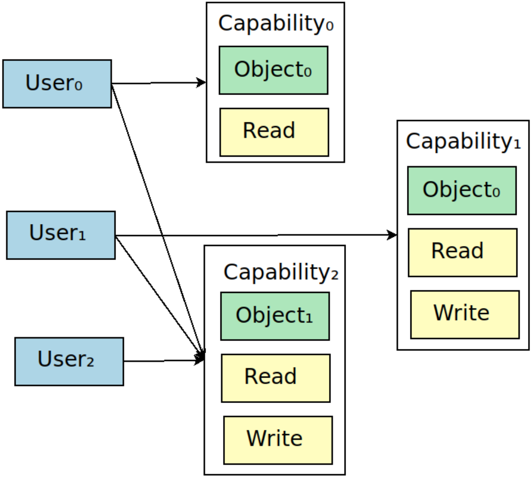

Figure 3: Capability based access control

File descriptors

A file descriptor is a *capability of accessing a file*.

Each process has its own **file-descriptor table**.

File descriptors

A file descriptor is a *capability of accessing a file*.

Each process has its own **file-descriptor table**.

Not only for accessing files:

- Files
- stdout/stdin/stderr
- pipes (inter-process communication)
- sockets (network access)

File descriptors

The OS checks permissions when opening a file and creating the descriptor.

- File descriptors can be transferred between processes
- The recipient process does not need to have permission to access the file to use the file-descriptor

This gives a fine-grained way to transfer capabilities between processes.

Example:

An HTTP server wants:

- to bind to port 80 (requires root),
- but processing HTTP request as root is dangerous

Priviledge dropping

1. Roots starts the `httpd`-program with UID=0.
2. `httpd` creates a socket and bind it to port 80.
3. `httpd` creates a child process with a less priviledged UID.
4. `httpd` hands the socket file descriptor to the child process
5. Child process handles the HTTP requests.