INF226 – Software Security

Håkon Robbestad Gylterud

2019–09-11

Recap: Access control models

Access control

Access control has different aspects:

- Physical
- Logical
- Cryptographic
- Social

We are currently studying the *logical aspects of access control*.

MAC vs DAC

Mandatory access control: Central, non-tranferrable access.

- Example: Classified documents.
- Example: Disk quotas.

Discretionary access control: At least some priviledges are transferrable.

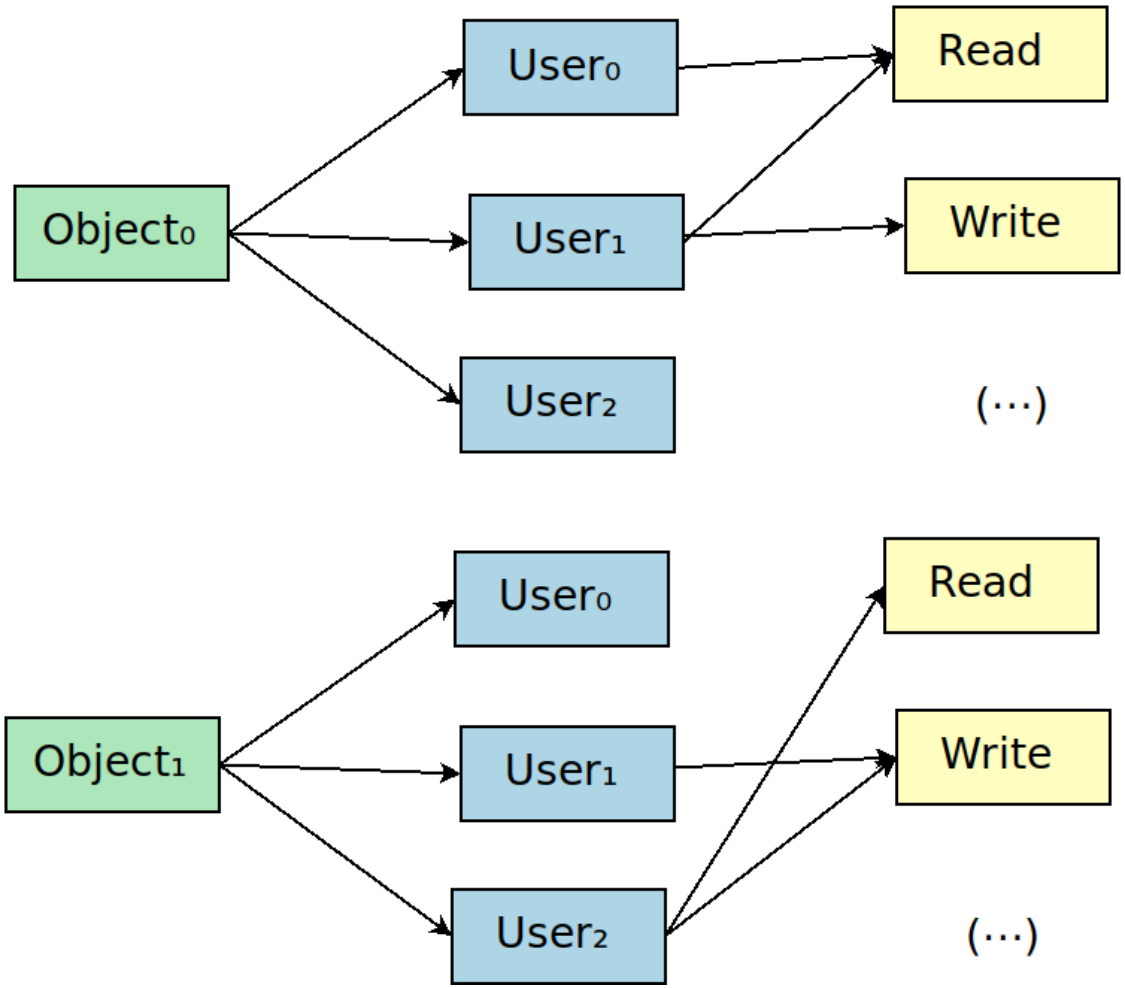
- Example: File owners can decide who can read and write to the file.
- Example: Open file descriptors can be tranferred to other processes.

Models of access control

We have seen three models of access control:

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

Access control lists



Rôle based access control

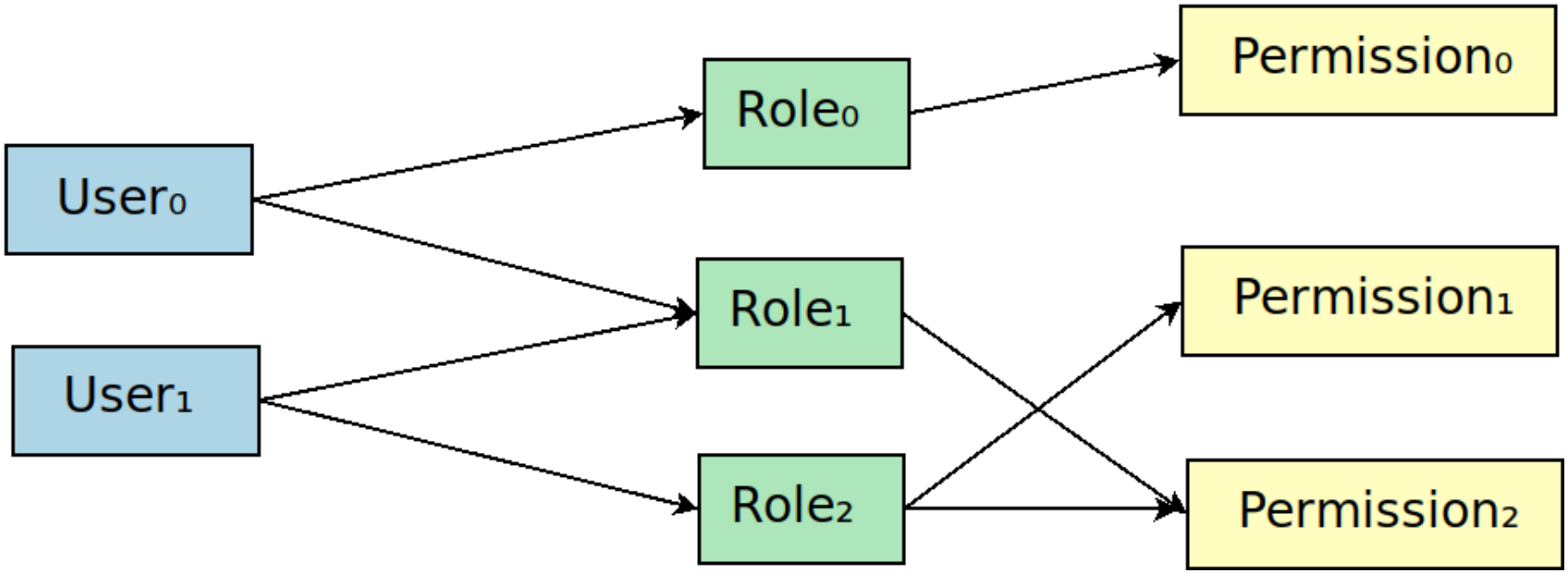
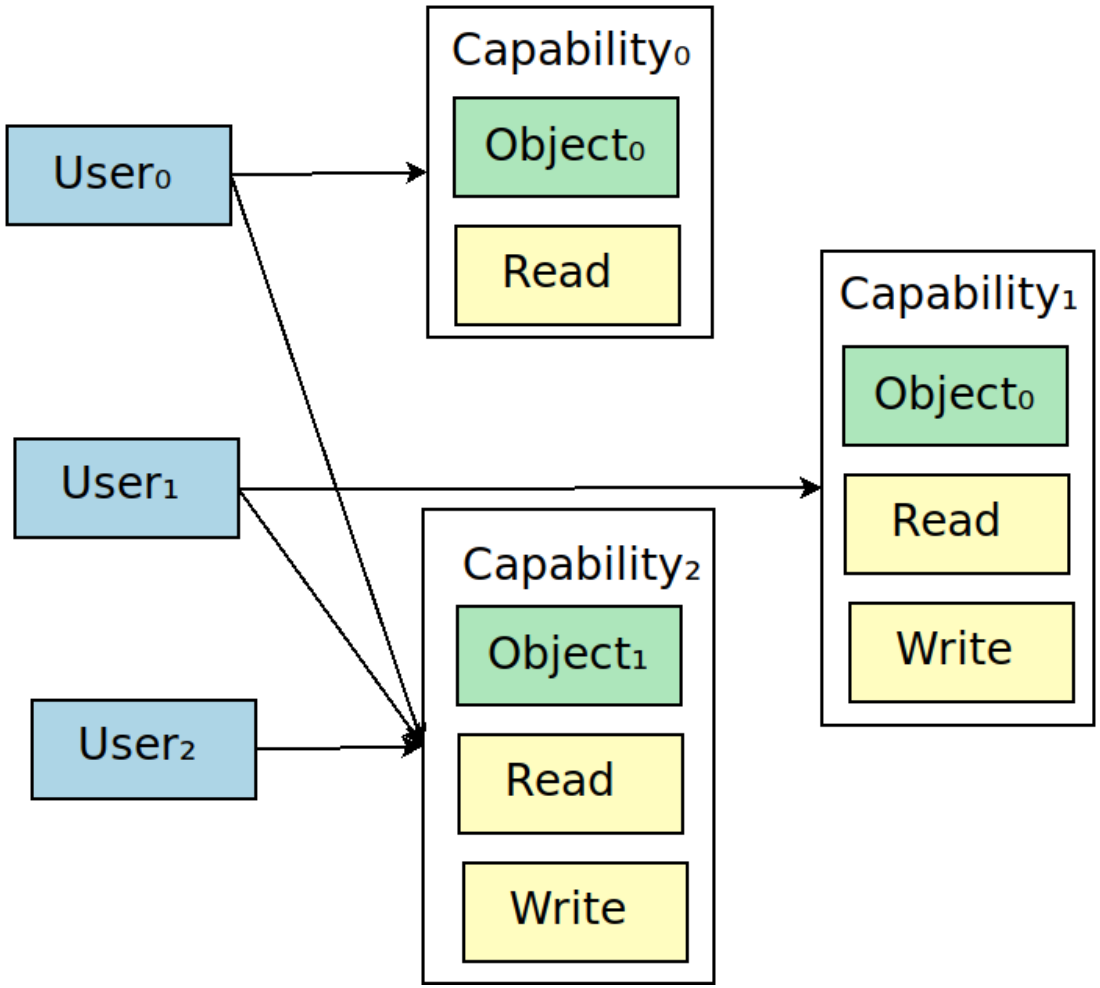


Figure 2: Rôle based access control

Capability based access control



Operating system and application security

Running example: Web server

Let us imagine that we are designing a web server.

- Handles a lot of untrusted user input.
- Is written in C, or runs CGI binaries written in C.

A database will be running on the same server, and we worry that a breach in the web server could compromise the database.

What preventive measures are available from the operating system?

Threat model: Defence in depth

Assumption: Some process on the system is misbehaving.

- *Example*: A buffer overflow in a service has caused an attacker to gain remote code execution.

Threat model: Defence in depth

Assumption: Some process on the system is misbehaving.

- *Example*: A buffer overflow in a service has caused an attacker to gain remote code execution.

Requirement: Limit the impact of this break-in.

Threat model: Defence in depth

Assumption: Some process on the system is misbehaving.

- *Example*: A buffer overflow in a service has caused an attacker to gain remote code execution.

Requirement: Limit the impact of this break-in.

Mechanisms: ?

Operating systems

What is the rôle of the operating system?

Operating systems

What is the rôle of the operating system?

- Orchestrate processes (software)
- Provide an abstract interface for hardware (drivers)

System calls

Programs can communicate with the OS through **system calls**, which interrupts the program and returns the control to the OS.

Examples

- The system call `open` opens (or creates) a file and returns a file descriptor to the program.
- The system calls `socket` and `connect` creates a network connect and returns a file descriptor to the program.
- The system call `write` writes bytes to a given file descriptor.

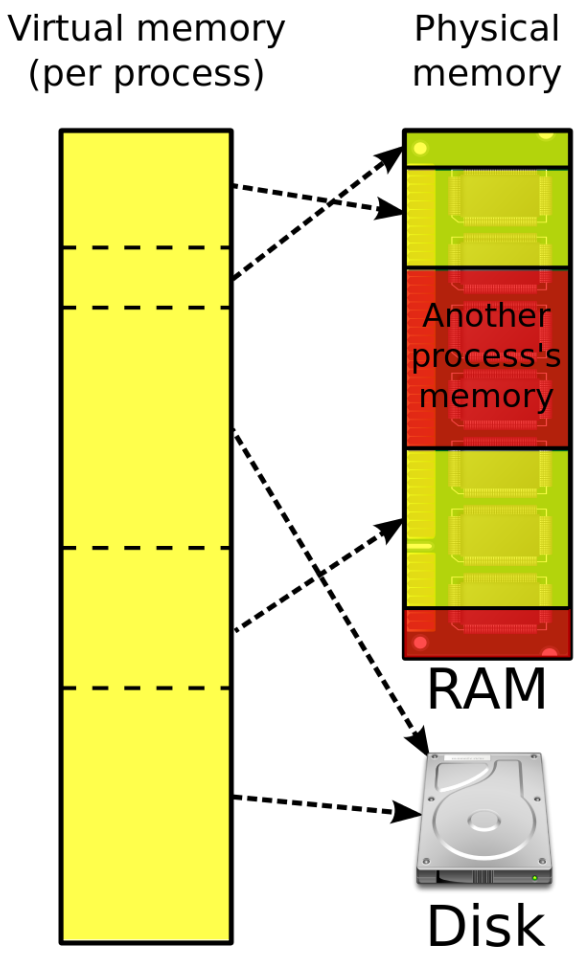
OS level priviledge separation

On the OS level individual processes have different protections for different resources:

- Memory:
  - Virtual memory mapping
  - Limits
- CPU
  - Scheduling priority
- File system:
  - Permissions
  - `chroot` or other visibility restrictions
  - quotas
- Open files/sockets/network connections:
  - file descriptors
  - limits

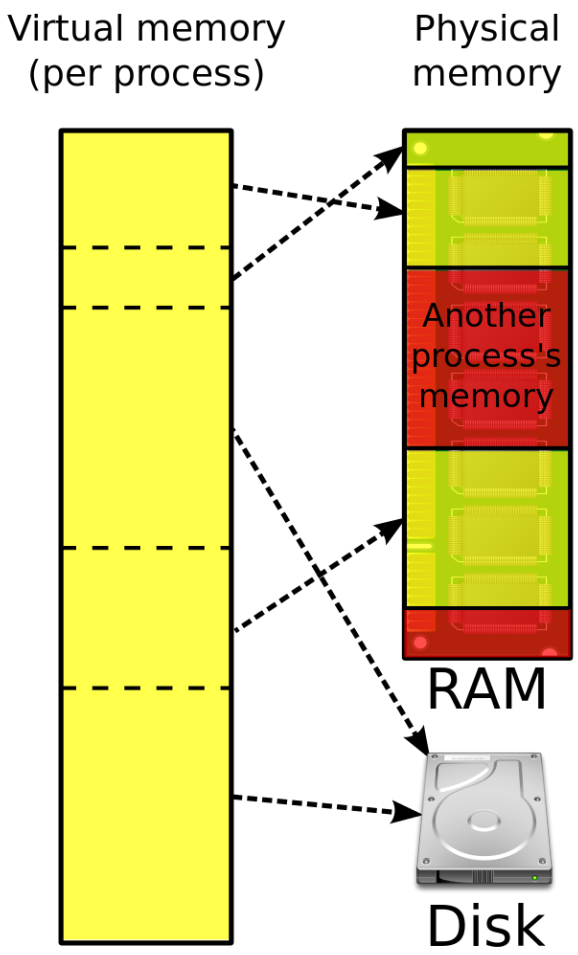
Memory protection

Virtual memory mapping



Not primarily for security: Letting programs use the physical memory addresses is *really inconvenient*.

Virtual memory mapping



Not primarily for security: Letting programs use the physical memory addresses is *really inconvenient*.

With virtual paged memory:

- Each program gets their own virtual address space.
- Memory locations not decided at compile time.
- Memory fragmentation hidden from programs.
- Easy to page out to swap (store memory on hard drive)

But, as a consequence: processes cannot directly address or access the memory of other processes

Virtual memory mapping

Exceptions:

1. Processes can allocate shared memory.
2. A process can attach them selves as a debugger to another process.

Number 2 is allowed by default in Linux for processes with the same UID. (See `ptrace(2)` manual page)

Example

Question: What do we need to do in order to prevent the compromised web server from accessing the memory of the database?

File system abstraction

The unix file system

The *unix file system* provides a unified way to access file systems based in **the root directory /**

Directories group the files into logical parts:

- `/bin`: programs
- `/sbin`: administrative programs
- `/etc`: system configuration
- `/dev`: virtual file system of devices
- `/home`: individual user's home folders
- `/tmp`: temporary files
- ...

Chroot

The operating system can restrict process file acess by **changing the root dir to a different directory** (`chroot`).

- **Example:** After `chroot /home/bar` the path `/bin/foo` translates to `/home/bar/bin/foo`.
- **Note:** a UID=0 (root) process easily access resources outside the new root.

This provides a form of file system virtualisation.

Chroot

Usefulness of `chroot` is limited by the restriction that only `root` can do it.

- Imagine that a user could set up a root folder with a forged `/etc/passwd` and `/etc/shadow`.
- Then they could fool a `SUID` program (such as `su`) to give them a `UID=0` shell.

Chroot

Chroot is not a complete sandboxing solution, it does not:

- Restrict network access
- Restrict usage of other system resources
- Prevent communication with/taking over other processes

Compare to:

- Plan9's concept of per process file system (approx. 1990)
- FreeBSD's `jail(8)` or Docker.
- OpenBSD's new `unveil(2)` (2018, work-in-progress)

Example

How can we use `chroot` to help prevent the web-server from accessing the database?

System call restrictions

System calls

Linux has more than *three hundred* system calls:

accept(2)	2.0	See notes on socketcall(2)
accept4(2)	2.6.28	
access(2)	1.0	
acct(2)	1.0	
add_key(2)	2.6.10	
adjtimex(2)	1.0	
alarm(2)	1.0	
alloc_hugepages(2)	2.5.36	Removed in 2.5.44
arch_prctl(2)	2.6	x86_64, x86 since 4.12
bdflush(2)	1.2	Deprecated (does nothing) since 2.6
bind(2)	2.0	See notes on socketcall(2)
bpf(2)	3.18	
brk(2)	1.0	
cacheflush(2)	1.2	Not on x86
capget(2)	2.2	
capset(2)	2.2	
chdir(2)	1.0	
chmod(2)	1.0	
chown(2)	2.2	See chown(2) for version details
chown32(2)	2.4	
chroot(2)	1.0	

OpenBSD pledge

System calls form a huge surface for an attacker to work with.

Difficult to restrict which system calls are allowed by a process, but OpenBSD's `pledge` gives such a mechanism:

- Calling pledge with a list of system call groups **restrics the process from accessing most system calls** not on the list.
- Further restrictions can be made at a later point.

Compare: `seccomp` for Linux or `capsicum` for FreeBSD.

OpenBSD pledge example

The openBSD `httpd` has pledge in `server.c`:

```
419     if (pledge("stdio rpath inet unix recvfd", NULL) == -1)
420         fatal("pledge");
```

OpenBSD pledge example

The openBSD `httpd` has pledge in `server.c`:

```
419     if (pledge("stdio rpath inet unix recvfd", NULL) == -1)
420         fatal("pledge");
```

Would this help prevent the compromised web server from compromising the database?

Example summary

What preventive measures do we have so far?

What kind of threats to they prevent?

What assumptions are they based on?

OS virtualisation

Linux kernel namespaces

Kernel namespaces allow processes to be grouped so that each group has:

- individual filesystem mount tables
- individual process tables
- individual network stack
- individual UID tables

More kinds of namespaces are being added to Linux.

Virtualisation

In stead of/in addition to manually separate priviledges using OS mechanisms, we can:

- Abstract away the specific operating system systematically:
  - **Operating system virtualization** (Docker or FreeBSD `jail(8)`)
- Abstrace away the specific hardware:
  - **Full virtualization**: can run a different ISA
  - **Paravirtualization**: Runs the CPU instructions natively.

Containers

Docker is an open source OS-virtualisation program which runs software packages called **containers**.

- Containers are *systematically separated* using OS mechanisms.
- Containers are templated by *images*:
  - An image programmatically constructs a containers.
  - Provides a predictable environment for the processes in the container.
  - Popular way to deploy web-services
- Container construction and administration through the daemon `dockerd`

Separation mechanisms

Docker is based on Linux' OS-level separation mechanisms:

- Chroot
- `namespaces` which gives each container:
  - individual mount tables
  - individual process tables
  - individual network stack
  - individual UID tables
- `cgroups` which limits the resource use of each container

Container capabilities

Each container has a specific set of capabilities[1]:

- An abstraction of the OS level restrictions
- Capabilities are whitelisted (not black listed)

[1] Not the same kinds of capabilities that we previously discussed

Docker security

Docker security can be decomposed into:

- Security of the underlying OS level separation mechanisms
- The dockerd daemon attack surface.
- Security of the container configuration.

Attack scenario: An attacker takes over a process running in a container.

A container is not a VM

While often branded as lightweight VMs, containers are fundamentally different from VMs:

Example: A container with `CAP_SYS_TIME` will set the time of the whole host, not just container.

In general, if the resource is not namespaced by the Linux kernel, it is global and can be affected by the container.

References

- Book: Operating Systems: Three easy pieces.
- Linux and OpenBSD man pages for specific syscals / protection mechanisms.

Muddiests point

Next week

Read the article *Preventing Priviledge Escalation* (link on Syllabus on MittUiB).

- Discusses in depth how priviledge separation is done in OpenSSH, zlib and several other programs.

Then we will discuss *authentication*.

Exercise for Monday

Read the article *Preventing Priviledge Escalation*, and answer the questions:

1. Which operating system mechanisms does this approach to priviledge separation rely upon?
2. Why does the slave process have to restart when going from pre-authentication phase to post-authentication phase?
3. What does the `P_SUGID` flag do?