

Lecture 11

User protection

Most OS are designed to be multi-user.

In this lecture we will study how the OS make sure an user won't adversely interfere another, while allowing them to cooperate.

References:

- [OSC] Section 18.1–18.5, 19.2–19.4.
- [ULK] pp 553–558 (Process Credentials and Capabilities).
- Secure UNIX Programming FAQ (e.g., <http://www.faqs.org/faqs/unix-faq/programmer/secure-programming/>)

POS(0230A)

POS(0230A)-11.1

Principles of protection

Basic principles (“rule-of-thumbs”):

- At any time, a process (or a user, or any active entity) should only be allowed to access resources it needs to complete its task. (The *need-to-know* principle)

E.g., there should be a way for a process to specify that it needs only to read a file, so that the file will not be accidentally modified.

- The OS defines and enforces **mechanisms** for protection.
- The administrators and users must define their protection **policy** using the mechanism provided by the OS.

E.g., the admin define a group `student` shared by all students, a user defines that one of his file can be read but not written by `student`'s, and the OS enforces this by denying writes from `student` group members.

POS(0230A)-11.2

POS(0230A)-11.3

What are protection domains?

Protection domain can be

- **Process.** Each process has exactly one row in the access-matrix.

To support the *need-to-know* principle, the process can **modify the access matrix** through the OS. Such requests are granted or rejected based on other information (e.g., a “password”, an access list, etc).

Access matrix can only be modified through the OS. Otherwise any user can change the list so that it contains the any desired capability.

- **User-ID.** Each process has a set of “user-IDs” (or something similar), and the user-ID is used to consult the access matrix.

Apart from modifying the access matrix, a process can **change the protection domain** to achieve the need-to-know principle.

Again, we need some mechanism to check whether to grant a request to switch to another user-ID.

POS(0230A)-11.4

Why we need protection mechanisms?

- To prevent **intentional violation of access restriction**, making it safe to allow untrustworthy users to **share resources**.
E.g., A student can use the shared file-server to do homework, load their program into the RAM of a shared server, etc., without the fear that another student would steal his work.
- To **increase reliability**. By restricting access, a **malfunctioning** subsystem has less chance to interfere a functioning one.
E.g., We run multiple programs in the same computer, some of which probably buggy, without the fear that one program will bring down another.
- To **allow execution of untrusted programs**. By restricting access, we are reasonably safe to run **programs with dubious source**.
E.g., A student can compile and run his own program without the authorization of the system administrator, since the student cannot affect other users by running his program.

Conceptual definition of protection policy: Access matrix

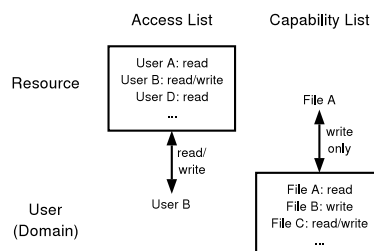
- The requestor of the resources is identified by a **protection domain**.
- Each **resource** defines a set of **operations** that can be individually allowed or denied.
- Conceptually, we view a protection policy as a table in this form:

Domain	File1	File2	File3	Soundcard
D_1	read		read	
D_2				write (=play)
D_3	read write	read		

- Ideally, the table entries are specified by the **resource owners** and the **resource users** in conjunction.

Illustration

The OS will not implement the access matrix directly, since it involves **huge space and administration cost**. Two common implementations:



- **Access list:** Store a list in the *resources* specifying who can access it.
- **Capability:** Store a list in the *domain* specifying what it can access.

POS(0230A)-11.5

Combining access-list with capabilities

- With **access list**, it is easy for **resource owners** to **change their entries**.
- This is significantly **more difficult with capabilities**, since the information is spread over all domains of the system.
- But most systems use **both** access-list and capability. E.g., Unix:
 1. To start using a resource, the **access-list** of the resource is checked. E.g., file/shared memory permission, etc.
 2. This results in a **capability**: opened file, attached memory, etc. This can be used to access the resource without further checks.
- Such a hybrid design is **more flexible**, e.g., a process can switch to another user, open a needed file, switch back and continue using the file. Side effect: revocation becomes nearly impossible.

POS(0230A)-11.6

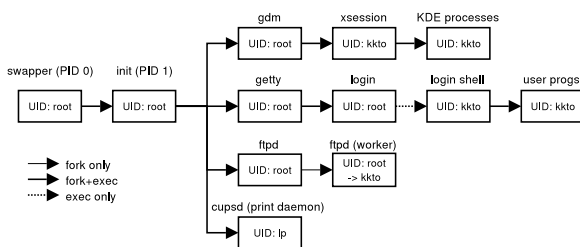
Unix UIDs and GIDs

- In Unix, users are identified by a **16-bit or 32-bit number** called **UID** (User Identifier).
- The `/etc/passwd` file stores some information about the user:
 - **GID** for the user. Better systems allocate one GID for each user; others won't and force users to be in a large group (e.g., `students`).
 - **Login ID** (the short 8-character name).
 - **Home directory**. Normally all files of that user are stored there.
 - **Shell program** to execute when the user login.
 - Other ("GECO") info, e.g., full name. See the man page `passwd(5)` for more information.
- Each actual user is allocated one UID. There are also some "pseudo-users" with UIDs. E.g., the manual page cache (`man`), the print spooler (`lp`), etc.

POS(0230A)-11.7

Basic idea of user IDs

Normally, a process **cannot switch** to a different user **unless** it has the **UID root**. So it **forks** out a process and let the child switches UID.



Init have a very high permission (root, UID=0), and fork out processes which **switches to the appropriate UID** after authentication.

POS(0230A)-11.8

The complete Unix domain

Dynamic domain of a process is more than just a UID. It includes:

- 3 UIDs: RUID (**real**), EUID (**effective**) and SUID (**saved**). When one logins, the `login` program set all to the user's uid.
- 3 GIDs: RGID, EGID and SGID; and some "supplementary gids". Defined in `/etc/passwd`, except the last which is specified in `/etc/groups`.

What are their normal uses?

- The effective UID, effective GID and supplementary GIDs are used to **check the permission modes** of resources when they are opened.
- The real UID and GID should always store the UID and GID of the one who **login** and created the process.
- The saved UID and GID holds an **additional UID/GID that the process can switch to...** (Will get back to this in a moment.)

POS(0230A)-11.9

Resource in Unix

- Each **resource** has an **owner user**, an **owner group**, and a **9-bit permission mode**.
- What resource we are talking about? Actually, quite a lot...
 - **Files** in the **filesystem**. This includes all kind of filesystem objects: regular files, sockets, device files, etc. Some POSIX IPC implementation also track the IPC constructs like shared memory in the filesystem. Use `ls -l` to see their UID/GID and permissions.
 - System V IPC constructs: **Shared memory** created by `shmget()`, **semaphores** created by `semget()`, etc. Use `ipcs` to see their UID/GID and permissions.
- Files have some more bits in the permission mode, though...

POS(0230A)-11.10

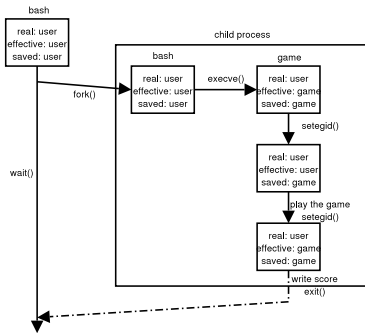
Changing domains: motivation

Suppose you write a game that keeps a highscore file.

- Of course, you **don't want players to arbitrarily change the highscore file**, but how that can be prevented?
- The highscore file must be owned by the game, not by the user. The permission modes must prevent the user from changing it. So the "others" bits should not allow writes.
- But then... **how the game can change the file itself?** It is the player who execute the game, so the game runs in the UID/GID of the player!
- The solution: the game program file has a special permission bit, the **SGID-bit** (apart from the 9-bit permission modes), set.
- When the program is executed, the effective and saved GID is set to that of the game program file, which is the same as that of the highscore file.

POS(0230A)-11.11

Illustration



Normally, a process can't arbitrarily change to another domain using `setegid()`.

But since the process has saved GID being `game` and real GID being `user`, it can switch between them.

Good programs will only use the `game` GID if it is needed, i.e., when opening the high-score file.

Many other programs are `setuid` or `setgid`. E.g., "su", "man", "X", "at", "crontab", "passwd", "write", many terminal programs, etc. Use `which` to find the program file, and `ls -l` to see what user/group they switch to.

POS(0230A)-11.12

Other extensions

• "Access Control Lists" (ACL)

Allow more fine-grained protection, some filesystems allow augmenting the bits with a list of user-operation pairs, e.g., (`tmchan`, `read`).
Read the `setfacl(1)` man page in Solaris.

• "Kernel capability"

Allow `root` owned process to drop some special capabilities. E.g., an `ftp` process only needs the ability to switch to any user, no need for ability to write to all files, etc. This limits the harm if the `ftp` server has a bug.

The full list of capabilities: `/usr/include/linux/capability.h`. The syscalls to manipulate it are `capget()` and `capset()`.

POS(0230A)-11.13

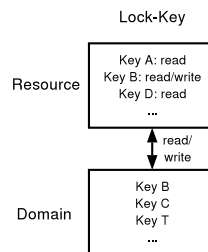
Weakness of access-list and capability based system

- Not very general: what if I need more than 2 users?
Imagine the game also need another user-id to access a device.
- Root is frequently needed, in particular to switch to arbitrary user.
The `ftp` server, login daemon, etc., must be executed as `root` (so that it can call `setuid()` after receiving password). What if they have a bug? An intruder get root access.
- Only admin can **define domains**, users can't.
E.g., as users we cannot run native code from untrusted web pages because we cannot prevent them from doing things that harm us. They have to code in a language like Java, which we translate to native code byte-by-byte and intercept all calls to the system: slow.

POS(0230A)-11.14

The lock-key scheme

A lock-key scheme (e.g., WinNT, Hurd) addresses these weaknesses.



- Resources and servers recognize not **UIDs**, but **locks** and **keys**.
- A list of **locks** in the *resources* and a list of **keys** in the *domain*, access is granted if and only if one key of the domain matches a lock of the resources.
- A process may have an **any number** of keys, and can **give them up**. It can also **request a key** from an "authentication server" to gain access to selected resources.
- A user process can **ask the OS to create a new key**.

POS(0230A)-11.15

Example use

`ftp` servers don't need to run as `root` under lock-key system...

- Initially, the server has **no permission**, apart from using the network.
- A user sends his username and password. The `ftp` server passes them to the authentication server, **requesting for the key of the user**.
- The authentication server sends the key to the `ftp` server, so that it can now access the files of the user.

User processes can run untrusted programs...

- The user process requests that a new (temporary) key be generated.
- Some resources are selected, and the access list of them are modified to allow processes with the new key to access them.
- The process `fork`, drops its own key, and run the untrusted program. Now that program can only access the selected resources.

POS(0230A)-11.16