CITS3007 Secure Coding
Access control and setuid

Unit coordinator: Arran Stewart

Highlights

- Authentication and authorization
- Access control
    - What is it, how it helps with our security goals
- Access control lists and capabilities
- `setuid` and `setgid`
- "Confused deputy" and TOCTOU vulnerabilities

Security goals

On most systems, we have more than one user, and more than one program.

This makes achieving our "C I A" security goals more complicated.

Two mechanisms that help us are *authentication* and *authorization*.

authentication
: Verifying who a user is ("Who is this?")

authorization
: Checking whether a particular user is permitted to perform some action ("What can they do?")

Authentication

Simple (not necessarily most secure) way to implement authentication: identify every user using a **username** and a **password**.

(What if a user can change their username? Persistent **user IDs**, usually incrementing integers, help solve this.)

Multi-factor authentication

Stronger authentication relies on multiple methods (“factors”) of proving who you are. Traditionally, two or more of

something you have
: E.g. a smart card, or USB fob

something you know
: E.g. a password (or even better, a passphrase)

something you are
: E.g. your fingerprint, retina scan, or face

Authentication as part of the OS

Authentication provides only very weak guarantees on its own – it must be used in concert with other techniques.

Example

A desktop computer or laptop might ask for a username and password, but not encrypt disk storage used, and not use a secure boot process.[1]

That would mean anyone with physical access to the computer could boot from e.g. a USB thumb-drive, run their own OS, bypass my computer's normal authentication system, and read (or alter) data on disk.

[1] Usually, a secure boot process should (a) limit what devices the system can be booted from, (b) only allow the computer to be booted using OSs from trusted sources, and (c) attempt to detect possible tampering with the hardware.

Authentication best practices

- Passwords should not be stored as plain text – in fact, they should not be stored at all.
- Operating systems (and other software) normally instead store a cryptographic one-way **hash** of the password or passphrase. (We discuss hashes further when we look at cryptography.)
- Creating a good password hash algorithm is difficult and error-prone, so it's best to stick to a known and reliable one.
  - e.g. The default algorithm on many recent Linux distributions is an algorithm called yescrypt

Authorization

Once a user is authenticated, we need to decide what they are permitted to do (i.e. perform authorization).

- Authorization is enforced by an **access control system**
- The access control system assumes a user has already been authenticated in some way

Definition: access control system

- A collection of methods and components that determines who has access to particular system resources, and the type of access they have.
- Ensures all actions on resources are within the security policy
- Supports our goals of achieving confidentiality and integrity

Access control systems

Why do we cover this?

- So you know what's available when implementing software
  - OS-provided access control systems often have many features we can leverage
- Because multi-user software typically must implement its *own* access control system, and it's useful to know the basics
  - e.g. Multi-user software like a bulletin board, ride-sharing app, database, etc will need to model users, resources and rights

Terminology

principal (or **subject**)
: Representation of a user or a group of users

resource (or **object**)
: Something we want to protect, that a principal can access or operate on in some way – e.g. a file, a running process, the database of users.
(On Unix systems, many resources are represented as files.)

permission (or **right**)
: Some action that can be performed on a resource. E.g. for a file – **read**ing, **writ**ing and **execut**ing the file might be distinct permissions.

Access control system

Examples of questions we might expect an access control system to answer:

- Can Alice read the file "`/home/Bob/my-private-journal.txt`"?
- Can Bob open a TCP socket, listening on port 80?
- Can Carol write to row 15 of the database table "`USER-SALARIES`"?

Access control system design

- Principle to bear in mind: **Principle of Least Privilege**
  - Programs, users and systems should be given enough privileges to perform their tasks, and no more
- *Efficiency*:
  - We can have many file accesses occuring every second, so our system needs to be able to make decisions quickly
- *Expressiveness*:
  - We may want to express complex, high-level policies about who can do what

Matrix model

- Imagine that we have a matrix listing all principals (as rows) and all resources in the system (as columns)

	file1	file2	file3
alice	rwx	r	r
bob		rwx	
carol			rwx

Matrix model

	file1	file2	file3
alice	rwx	r	r
bob		rwx	
carol			rwx

- At the intersection, we list the **permissions** or rights for that principal and that resource (here, **{r,w,x}** = {read,write,execute})
- Called an **access control matrix** or **access matrix**

Matrix model

	file1	file2	file3
alice	rwx	r	r
bob		rwx	
carol			rwx

Terminology you might also see:

access control entry
A triplet of (principal,resource,permission list) – i.e. one cell from the matrix

access control list
All the access control entries for one resource – i.e., a column from the matrix

DAC vs MAC – who decides?

Who decides what rights subjects have for particular objects?

One answer:

- Individual users can control access to e.g. files that they own
  $\Rightarrow$ we have a **Discretionary Access Control** (DAC) system
- Some system mechanism controls access, and individual users can't alter it
  $\Rightarrow$ we have a **Mandatory Access Control** (MAC) system

There are other sorts as well – e.g. Role Based Access Control (RBAC) which we don't go into in this unit.

DAC vs MAC – who decides?

Discretionary Access Control (DAC) system:

- Owners of objects set the permissions
- Most common approach
- Poses difficulties for e.g. protecting audit logs from sysadmins

Mandatory Access Control (MAC) system:

- Enforced by the OS
- May be appropriate for e.g. Dept of Defence systems
- Implies that superusers/system administrators don't have ultimate control
- Used e.g. to ensure not even sysadmins can tamper with the OS kernel,
- To be effective, needs hardware support: else we can e.g. boot from a thumbdrive and take control

DACs

For DACs, there's usually one sort of right called "ownership", which grants the ability to add or remove rights

- e.g. granting others the right to read files in your home directory

DAC complications

- Suppose we're sysadmin: do we really trust users to get all permissions right?
- What if a user wants to download and run programs they found online – should they be able to?
- What if some users should be considered more trustworthy than others?

Mixed systems

Many OSs will actually implement aspects of both MACs and DACs.

Example: Windows provides a kind of support for some MAC-style, system-specified permissions.

Downloaded files

- Resources have an “integrity-level” label
- Default file label = “medium”, but web browser and downloaded files are “low”
- “Low”-labelled files may not alter higher-labelled files.
- ⇒ To run some program you downloaded, if it will change files on filesystem, user must explicitly upgrade it from “low” to “medium”.

Question

What is some (non-OS) software you have used recently which would have need of an access control system?

- Who are the principals? How are they grouped?
- What are the resources?
- How is authentication done?
  (Password? Fingerprint? Something else?)
- How would password hashes be stored?
- How would you implement it?

Third-party providers

- Implementing an access control system can be a complex task
- Often we may leverage libraries and services provided by third parties
    - e.g. Okta, Azure, Google Cloud
- This can solve some problems (How to securely store password hashes?) but raises others (How much can we trust the provider?)
    - Using the libraries is often complex, and many developers rely on copy-and-pasting code
- Doesn't alleviate us of the responsibility of making sensible design choices

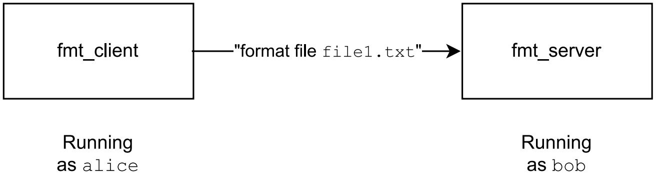
Deputies

- Suppose a user wants to change their password – stored in e.g. `/sys/PASSWORDS`
- We can't give every user read and write permissions to that file
- We might give particular *programs* the ability to take actions on behalf of particular users at a higher level of privilege than the user has.
  - e.g. We might specify that the `passwd` program, when it runs, runs as `root` and can read and alter this `sys/PASSWORDS` file.
  - Or we might run a program as a server, started by a privileged user – and other users just send *requests*.

Confused deputies

But if we do this carelessly, it leads to *confused deputies*.

Example: Alice sends a request to a server process started by Bob – "give me the contents of `file1`, nicely formatted"



It's *Bob's* permissions that are used to check whether a file can be read; so Alice could ask for the contents of `/home/bob/my_secret_file.txt`, which she shouldn't be allowed to have access to.

(See "The Confused Deputy (or why capabilities might have been invented)")

Confused deputies

An example that has actually occurred:

- A webserver – each student is allowed to create a `public_html` dir from which files for a website are served (e.g. `/home/student1/public_html`).
- The webserver process has privileged rights (e.g. because it needs to serve on port 80, which only `root` can do)
- A student creates a `public_html` in their directory, but it's a *symbolic link* (symlink) to `/etc/passwd` (or some other file only root should have access to).
- Because the webserver is running as root, it *does* have access to files like `/etc/passwd`; so it serves it up as a webpage.

Solutions to confused deputies: coming up later

Implementing an access control matrix

Suppose we want to implement an access control system – how should we store the information about rights?

"As a very big 2D array" is not a good idea – many cells would be empty (i.e. sparse array) or duplicated, array would be large

- e.g. Suppose Alice owns a file. Do we really want to store a list of all users at e.g. UWA who *don't* have permission to read/write it?
- We might want to have "default" permissions for things
  - it's then wasteful to list them explicitly for every user/subject
  - only explicitly list people who've specifically been granted *more* or *fewer* rights.

Implementing an access control matrix

Some options:

(a) Store by “column” (resource)
  - Each object is associated with a list of users, and what rights they have

(b) Store by “row” (user)
  - Each subject is associated with a list of objects, and what rights the user has for it

ACL vs capabilities

(a) By "column" (resource)
- Each object is associated with a list of users, and what rights they have

(b) By "row" (user)
- Each subject is associated with a list of objects, and what rights the user has for it

- Option (a) leads to the idea of an **access control list** (ACL)
- Option (b) leads to the idea of a **capability system** (though there's more to such a system than just this)
  - Rights to do things are held by subjects, and can be passed around to other subjects

---

NB: Don't confuse a "capability system" with `man 7 capabilities`

- a Linux approach to making superuser permissions finer-grained
- not actually a "capability system"

ACLs vs capabilities

ACL

- Store rights as e.g. file metadata
- Straightforward to implement
- Easy to e.g. revoke one user's rights to a file
- Difficult to determine all rights possessed by one user
- Difficult to e.g. revoke a user's right on *all* files
- Example: All popular OSs (Linux, Mac, Windows)

Capabilities

- Easy to determine all rights possessed by one user
- Easy to add and remove users, and to delegate rights
- Difficult to e.g. change rights of all users to one file
- Example: Various distributed OSs (e.g. Amoeba, experimental system developed in 90s)

ACLs vs capabilities

- In practice, ACLs don't list *every* user – doesn't scale well
- Also, we said access control needs to be efficient – in many systems, permissions are only checked when a file is *opened*, not each time the file content is accessed
- Many OSs combine aspects of ACLs and capability systems

Capability systems

We don't look at them in detail, but as noted, many OSs use aspects of capabilities.

Typically very powerful and flexible.

Particular subjects might have the ability to *copy* capabilities so they can be given to others – or perhaps only to *transfer* capabilities (i.e., the original subject no longer possesses them)

Unix approach

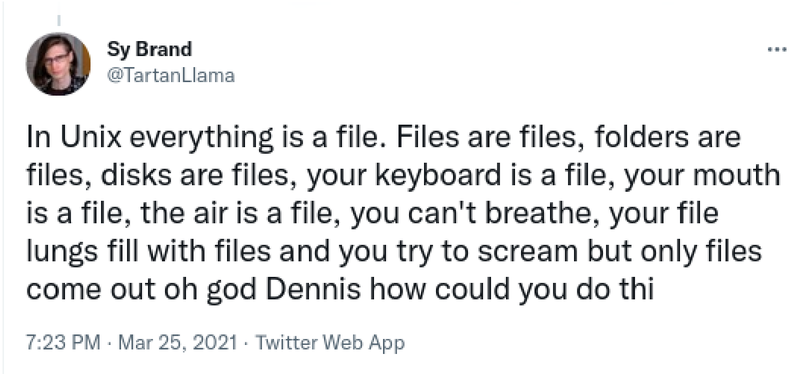
The Unix approach to *subjects* (principals):

- Users have a *user ID*, and one or more *groups*.
- The `root` user (with user ID 0) is the superuser
  - The first process starts as `root`, spawns others
- Every user has a primary group (stored in `/etc/passwd`), and can be a member of others (stored in `/etc/group`).
- A user `nobody` normally exists that owns no files, and can be used as a default user for unprivileged operations
- Processes execute with the permissions ("effective user ID") of the user that started them
- When determining rights to files, we use a coarse-grained approach and divide all principals into
  - the user/owner
  - the group owner
  - everyone else

Unix approach

The Unix approach to *objects* (principals):

- "Everything is a file"
- represent as many things as possible as *files*; then, we can use filesystem permissions to implement our access control system



Sy Brand
@TartanLlama

In Unix everything is a file. Files are files, folders are files, disks are files, your keyboard is a file, your mouth is a file, the air is a file, you can't breathe, your file lungs fill with files and you try to scream but only files come out oh god Dennis how could you do thi

7:23 PM · Mar 25, 2021 · Twitter Web App

Unix approach

- Classify file rights as “read”, “write” and “execute”
  - There are other rights needed to execute particular system calls (e.g. to kill a process)
  - The OS kernel will check whether a subject (a process) has rights to make particular system calls
  - For the `root` user, the answer is always “yes” (but see `man 7 capabilities`)
- Classify subjects as “user”, “group”, “everyone else”
- Processes have an *actual* user ID and group ID (based on the user that started them)
  - But can also have an *effective* user ID and group ID – differs for `setuid` and `setgid` programs
  - Check file permissions only on open
  - Effectively, file descriptors are a kind of “capability”, and can be passed around to e.g. to subprocesses, and even unrelated processes

Unix approach

- Doesn't easily allow for flexible rules
  - e.g. "Allow Alice's file `F` to be read by every user except Carol and Dan"
- The system calls for managing `setuid` and `setgid` programs are easy to get wrong
- Because `root` can do anything – difficult to create a sysadmin-untamperable audit trail/log
  - need to either store off-system/offsite, or modify the DAC approach

Note:

- Many file systems allow files to have "extended attributes" (see https://en.wikipedia.org/wiki/Extended_file_attributes) which allow more flexible policies to be implemented on top
- On modern systems, the Unix approach is typically agumented e.g. the SELinux (Security-Enhanced Linux) architecture

Solutions to confused deputies

Confused deputies arise when a process with high privileges is “fooled” into letting a less-privileged principal do something they shouldn’t.

One solution:

- Split the program into two interacting processes that communicate.

Solutions to confused deputies – client/server

e.g. Suppose a compiler needs to allow users to compile input files and write to output files, and **also** should write billing/audit information to `/sys/billing`.

- Split the compiler into two:
  - Compiler part runs as user, will only read or write files the user has access to
  - A separate process runs as (e.g. `root`), is communicated with by compiler process, writes to `/sys/billing`
- Better practice would be to create a dedicated `billing` user, not to use `root`
- Principle of Least Privilege: give principals only the rights they need to perform their job

Solutions to confused deputies – `setuid`

Client/server approach is not always appropriate (e.g. not especially fast)

`setuid` approach:

- Make the compiler a `setuid` program, which starts off running as `root`
- Open `/sys/billing`
- Immediately drop all root privileges
  (again: Principle of Least Privilege)
- Now do the job of e.g. reading input files, compiling and writing to output files

Solutions to confused deputies – setuid

Downsides:

- Relies on programmer to get things right
- On Unix systems, easy to get wrong
- Easy to create TOCTOU bugs – "time of check vs time of use"

```
if not actual_user_can_access("file1"):
  sys.exit(1)
fp = open("file1", "w")
fp.write("some data")
```

See:

- Bishop, "How to Write a Setuid Program" (PDF)
- Checklist for Security of Setuid Programs (PDF)