University of BRISTOL

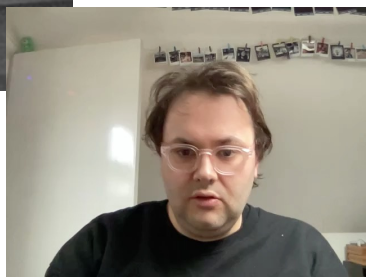
COMS20012: Introduction to OS Security and Access Control Mechanisms

Joseph Hallett

bristol.ac.uk

Whats all this about?

- Originally you didn't have your *own* computer…



Early timesharing systems

- Originally compute was a *really limited resource*
- Programs would be given as *punched cards* to the system admin
  - They would run them for you and give you the output later
- We still have systems like this!
  - If you play with the *Blue Crystal supercomputer* you'll have to submit a job to run on it and wait till the timesharing system loads and runs it for you
- No need for security*… only one thing runs at a time
  - *: This is wrong in all important ways.

As computers got more powerful…

- Programs as cards was replaced with programs in memory…
- Sysadmin to load programs was replaced by a scheduler program
- Computer started to have more than one terminal!
  - Multiple users could login at the same time

- …Suddenly we **do** need security (*isolation*)
  - You don't want Alice or Bob reading your files
  - You don't want Alice or Bob's programs crashing yours
  - …Except when we do…

  - How do we get the computer to let us write and enforce these policies?

PROTECTION

Butler W. Lampson
Xerox Corporation
Palo Alto, California

The original motivation for putting protection mechanisms into computer systems was to keep one user's malice or error from harming other users. Harm can be inflicted in several ways:

a) by destroying or modifying another user's data;

b) by reading or copying another user's data without permission;

c) by degrading the service another user gets, e.g. using up all the disk space or getting more than a fair share of the processing time. An extreme case is a malicious act or accident which crashes the system - this might be considered the ultimate degradation.

More recently it has been realized that all of the above reasons for wanting protection are just as strong if the word 'user' is replaced by 'program'. This line of thinking leads in two main directions:

a) towards enforcing the rules of modular programming so that it is possible, using the protection system, to guarantee that errors in one module will not affect another one. With this kind of control it is much easier to gain confidence in the reliability of a large system, since the protection provides fire walls which prevent the spread of trouble [4,9];

b) towards the support of proprietary programs, so that a user can buy a service in the form of a program which he can only call, but not read [9]. A simple example might be a proprietary compiler whose service is sold by number of statements compiled. A more complex case is a program which runs test cases against a proprietary data base as well.

Access Control

- General framework for expressing and keeping track of *who* is allowed to do *what* with the various *resources* we'd like to protect

- Vocab:
  - *Objects*. Stuff (resources/files) we want to protect
  - *Subjects*. Stuff (users/processes) that does things to objects
  - *Principals*. Subjects that set the policies
  - *Domains.* What a principal is allowed to set policies about
  - *Policies*. Rules about how subjects and objects interact

Access Control Matrices

- Idea proposed by Lampson…
- For every object on the system create a big table as to who is allowed to do what with it.
- Check the table before doing anything
- Quickly gets silly for large numbers of objects

Subject	File /etc/passwd	File /home/alice/diary.txt
Alice	Read & write	Read & Write
Bob	Read	-

UNIX DAC

- *Discretionary* Access Controls

- Almost everything is a file
- All files have an *owner user* and a *group*
- Users can belong to more than one group
- File permissions live as metadata on a file
- Permissions can be granted to *read, write* or *execute* a file

- user group other owner group filename
- `-rw-r--r--.root root 3303 /etc/passwd`

Useful commands for *NIX

- If you're ever on a commandline on a some UNIX-y system these are the commands you need to interact with file permissions
  - chmod: change the permissions of a file you own
  - chown: change the owner and group of a file you can control
  - ls –l: show the permissions of a file

- These are the system calls
  - chmod/fchmod/fchmodat: as above
  - chown/fchown/fchownat: as above
  - stat/fstat/fstatat: get information about a file

Extended File Access Control Lists

- Linux has also moved beyond just the older UNIX DAC
- Files can have more precise policies than just User/Group/Other
  - Requires a filesystem mounted with support for extended attributes and additional file metadata (BTRFS/EXT4)

```
$ ls -l /dev/audio
crw-rw----+ 1 root audio 14, 4 Nov. 9 12:49 /dev/audio

$ getfacl /dev/audio
# file: dev/audio
# owner: root
# group: audio
user::rw-
user:joseph:r--
group::rw-
mask::rw-
other::---
```

UNIX DAC Issues and Strengths

- The UNIX DAC is *surprisingly* flexible (even without ACL)
- It is still the basis for most OS's access control mechanisms.
- But…

- Trusts the owner of files to set the file permissions correctly
  - What if the system administrator wants to set the rules?

- What if you want to share a file between Alice, Bob and Charlie but no one else?
  - Oh well apart from Diveena but she's only allowed to read
  - Quickly gets unruly!

Solving the systems admin problem

- We'd like a systems administrator to make arbitrary tweaks to a system's access control matrix
- If all access control decisions are made by the *owner* of the files
  - Then a files owner can make something more permissive than the system admin might like…
- This starts to be a bigger problem with military/corporate systems…
  - How do you stop someone leaking secrets?

Mandatory Access Controls (MAC)

- Systems administrator sets the policy as to who can access what
- Typical implementations of you might see:
  - SELinux: NSA's patches for Linux to add a MAC. Complex but powerful.
  - AppArmour: Canonical's simplified MAC system based on paths
  - TOMOYO: another simplified system based on paths

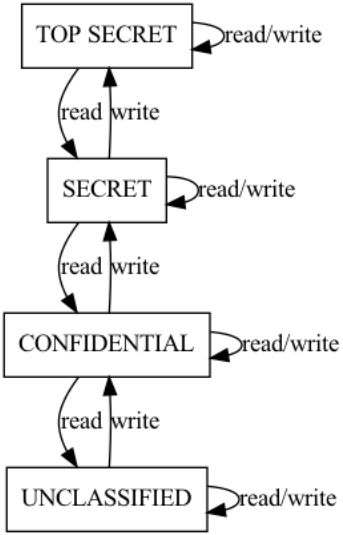
- Policies can be applied to more than just files
  - For example:
    "The web browser can access files in the downloads folder ONLY"
- Usually, will work on top of the DAC and defer the permit decision if the MAC is willing to allow the operation

So why on earth do you need a MAC?

- Government, military types and companies are really keen on Security classifications
  - e.g. UNCLASSIFIED < CONFIDENTIAL < SECRET < TOP SECRET

- In particular there is the an access control model called *Bell-LaPadula* that they'd really like to enforce…

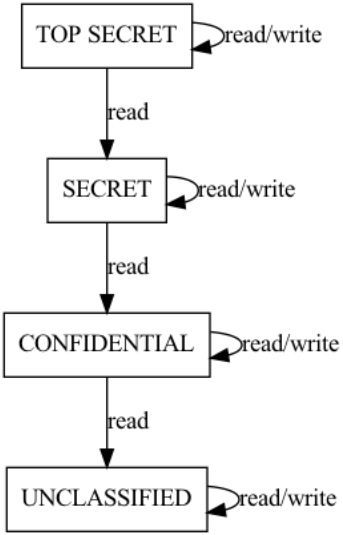
Bell LaPadula Model



- Model of access control developed in the US Department of Defense based on *multi-level security.*

- All objects and subjects have a security clearance
- Two rules about the system:
  - *Simple security property:* You cannot read something with more clearance than you. (**No read up**) (preserves secrecy)
  - ★*-property:* You cannot write to something with less clearance than you (**No write down**) (prevents leaking)

Bell LaPadula Model (2)



- Model of access control developed in the US Department of Defense based on *multi-level security.*

- All objects and subjects have a security clearance
- Two rules about the system:
  - *Simple security property:* You cannot read something with more clearance than you. (**No read up**) (preserves secrecy)
  - *★-property:* You cannot write to something with less clearance than you (**No write down**) (prevents leaking)
  - *Strong ★-property:* You can only write to something with the same level of clearance as you (prevents influencing people with more clearance)

But where do you go beyond MAC & DAC?

- Role-based Access Control: grant permissions based on what job someone is currently fulfilling

- For example:
  - the security guard can decide who is allowed into the building
  - During the day it's the day guard, during the night it's the night guard
  - The rota says whether Alice or Bob is allowed to activate the role of the security guard

But where do you go beyond MAC & DAC?

- Attribute-based Access Control: grant permissions based on properties of the requesting subject

- For example:
  - This program can execute if a checking program says that it has no unsafe operations contained within it

Logic Programming

- Make access control decisions based on arbitrary logical inferences
  - Programming languages like *Prolog* and *Datalog* especially suited to this sort of programming
  - Frameworks like XACML can implement policies… but never really took off

- *Extremely* powerful…But can be problematic!
- We would hope for three properties:
  - Soundness: You only grant permissions to things that should have access to permissions
  - Completeness: Everything you should be able to make a decision about you can and do
  - Timeliness: You do all the above in a reasonable amount o

(*hope* is the operative word here…)

Files and Processes

- So far we've been talking mostly about *files* as the *objects*
- …but what about the *subjects?* How do they work?

UIDs and GIDs

- Like files all processes are owned by a *user* and *group*
- …But it's a *bit* more complex

- *UID* of a process is the user id of the person who is running it
- *GID* of a process is the *active* group id of the person running it

- When a process starts it inherits the UID and GID of the process that started it

- Access control decisions made on the basis of these UID/GIDs

Real, Effective and *Saved* UIDs and GIDs

- All process have *3 separate* UIDs/GIDs

- *Real UID/GID*: the UID/GID of the person running the process
- *Effective UID/GID*: the UID/GID that policies get tested against
- *Saved UID/GID*: the UID/GID of the process last time it changed

Why do we need 3 UID/GIDs?

- Because sometimes we need to change who the user is!

- Consider what happens when you turn on your computer…
  - The kernel is loaded by the CPU/Bootloader
  - Which spawns `init` running as root (UID 0)
  - Which eventually sets up terminals for users to login
  - …which run as root
  - …But which change to the logged in user if the user logs in successfully

- So how do you do that transition?
  - `setuid / setguid / setresuid / setresgid`

Dropping privileges

- Policies govern which users are allowed to switch to which others
- Common use-case:
  - Start with a process that needs to do something privileged
  - Drop to a user with less permissions as soon as possible after to protect the system in case of overreach

  - Login needs to run as root to be able to switch to any other user
  - But once logged in you shouldn't be able to switch to any other user

Sometimes you need to gain privileges

- Suppose a user wants to update their password…
- Password hashes are stored in `/etc/shadow` which is only accessible by the root user
- How do you let a user run the password updating program `passwd` as root but nothing else?
  - Program is marked as setuid (`chmod ug+s /usr/bin/passwd`)
  - When started the effective and saved permission are set to the owner / group of the `passwd` program

```
-r-sr-xr-x   1    root bin  /usr/bin/passwd
```

(`su`/`sudo`/`doas` use the same mechanisms)

Loads of gotchas!

- Systems have policies to describe when these transitions are allowed to be made and under what circumstances
- Typical ones you *may* see:
  - Only allowed a setuid program if program read only and on special disks
  - Only root/special users are allowed to drop privileges
  - Cannot transition from a privileged to a less privileged user
  - If a privileged process `execs` an interactive program (e.g. `bash`) then the permissions of the real UID are transferred not the effective permissions
- Ultimately the OS/sysadmin sets the policy in place
  - Read the manual ;-)

Review

- *Files* are access control *objects; Processes* are access control *subjects; Policies* are the rules.
- *Objects* have *owners*, *process* have *users*
- In discretionary access control systems file owners say who can read their files, in mandatory access control systems the sysadmin writes the rules
- Bell LaPadula is multi-level security
  - *Unclassified* up to *top secret*
  - *No read up* for secrecy, *no write down* for avoiding leaks
- More complex schemes are possible… Prolog sounds neat!
- Read the manual for specific system details!