Protection and Security

Dr Lina Barakat

Department of Informatics
King's College London
Email: lina.barakat@kcl.ac.uk
Bush House (N) 6.04
Office Hours: Wednesday 13pm- 15pm

Chapters 14 and 15 of "Operating Systems Concepts" Silberschatz, Galvin, Gagne

Plus additional examples on Unix File Access Control

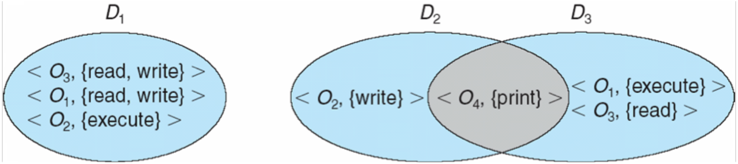
Aims and Objectives

- To discuss the goals and principles of protection in a modern computer system.
- To explain how protection domains, combined with an access matrix, are used to specify the resources a process may access.
- To look at an example implementation: Unix File Access Control.
- To discuss security threats and attacks.
- To explain the fundamentals of authentication.

- In one protection model, computer consists of a collection of objects, hardware or software.
- Each object has a unique name and can be accessed through a well-defined set of operations.
- **Protection problem**: ensure that each object is accessed correctly and only by those processes that are allowed to do so.
- Guiding principle: **principle of least privilege**
    - Programs, users and systems should be given just enough privileges to perform their tasks.
    - Limits damage if entity has a bug, gets abused.
    - **Need to know**, a similar concept regarding access to data.

Protection Domain

- A process operates within a protection domain.
- Domain = set of <object-name, access rights>



- Domain can be user, process, procedure

The Need to Know Principle

- The **need to know** principle:
  - a process should be allowed to access only those resources for which it has authorization; and
  - at any time, a process should be able to access only those resources that it currently requires to complete its task.

- If association between processes and domains is **fixed**:
  - a mechanism must be available to change the content of a domain, so that the domain always reflects the minimum necessary access rights.

- If association between processes and domains is **dynamic**:
  - a mechanism is available to allow **domain switching**, enabling the process to switch from one domain to another.
  - may also want to allow the content of a domain to be changed.

Access Matrix

- An abstract model of protection
- Rows represent **domains**
- Columns represent **objects**
- $Entry(i,j)$ is a set of **access rights**: the set of operations that a process executing in domain D_i, can invoke on object O_j.

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- When we switch a process from one domain to another, we are executing an operation **switch** on an object (the domain).
- Switching from D_i to D_j is allowed $\Leftrightarrow$ $switch \in Entry(i,j)$

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Additional **copy** operation (denoted by asterisk $*$):

- Allows an access right to be copied within the column (i.e. for the object) for which the right is defined.
- When a right R^* is copied from $Entry(i,j)$ to $Entry(k,j)$, we can have three variations:

  Copy: R^* remains in $Entry(i,j)$, and R^* is created in $Entry(k,j)$.

  Limited Copy: R^* remains in $Entry(i,j)$, and R is created in $Entry(k,j)$.

  Transfer: R^* is removed from $Entry(i,j)$, and R^* is created in $Entry(k,j)$.

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Additional **owner** operation:

- If $owner \in Entry(i,j)$, then a process executing in domain D_i can add and remove any right in any entry in column j (i.e. for object O_j).

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Access Matrix - Control over Content Change

Additional **control** operation:

- applicable only to domain objects.
- If $control \in Entry(i,j)$, then a process executing in domain D_i can remove any right from row j.

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

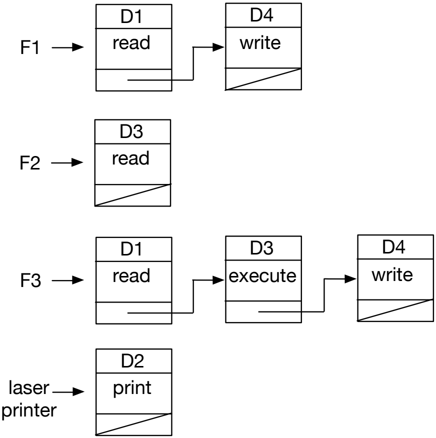
Implementation of the Access Matrix

- The access matrix is usually sparse

- Possible methods for implementation:
  - Store by **column** (Access Lists for Objects)
  - Store by **row** (Capability Lists for Domains)

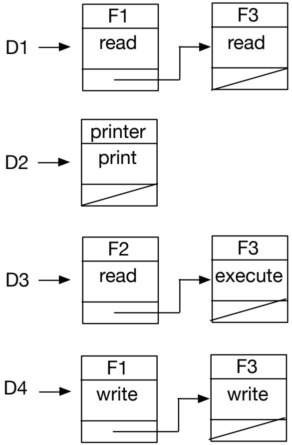
Access Lists for Objects

- Each column is implemented as an access list for one object.
- An access list for an object:
  - is a list of pairs $< domain, access\ rights >$, defining all domains with a nonempty set of access rights for that object.
  - may contain a **default** set of access rights for the object.



Capability Lists for Domains

- Each row is implemented as a capability list for one domain.
- A capability list for a domain:
  - is a list of the objects together with the operations allowed on these objects.
  - must be **unforgeable** (not directly accessible by a process in the domain).



Comparison of Implementations

- Access Lists for Objects:
  - *Good*: correspond to needs of users.
  - *Good*: revocation of access rights to an object is easy.
  - *Bad*: determining access rights by domain is difficult.
  - *Bad*: Every access to an object must be checked.

- Capability Lists for Domains:
  - *Good*: easy to localise information for a given process.
  - *Bad*: do not correspond directly to needs of users.
  - *Bad*: revocation of access rights might be inefficient.

- Most systems use a combination of both.

Traditional Unix File Access Control

- A simplified ACL-like system
- Only the owner (or **root**) sets file permissions
- Three classifications of users in connection with each file:
  - **Owner**: The user who created the file
  - **Group**: A set of users who are sharing the file and need similar access
  - **Universe**: All other users in the system
- Three file access modes:
  - **Read (r)**: view the content of the file.
  - **Write (w)**: modify, or remove the content of the file.
  - **Execute (x)**: run the file as a program.

owner	group	other
rw-	r - -	- - -

Traditional Unix File Access Control

- Use command **ls -l** to see the permissions of a file or directory.

```
$ls -l    testfile
-rwxrwxr--  1 lina  staff 1024  Feb 12 13:00  testfile

$ls -l    testdirectory
drwxr-----  2 lina  staff 25000  Sep 5 18:30  testdirectory
```

- Use command **chmod** (change mode) to change the permissions of a file or directory.
  - The symbolic mode
  - The absolute mode

Traditional Unix File Access Control

Using **chmod** in symbolic mode:

- Operator **+** : add the designated permission(s) to a file or directory.
- Operator **-** : remove the designated permission(s) from a file or directory.
- Operator **=** : set the designated permission(s).

```
$ls -l    testfile
-rwxrwxr--  1 lina  staff 1024  Feb 12 13:00  testfile

$chmod  o+wx    testfile
$ls -l    testfile
-rwxrwxrwx  1 lina  staff 1024  Feb 12 13:00  testfile


$chmod  u-x    testfile
$ls -l    testfile
-rw-rwxrwx  1 lina  staff 1024  Feb 12 13:00  testfile


$chmod  g=rx    testfile
$ls -l    testfile
-rw-r-xrwx  1 lina  staff 1024  Feb 12 13:00  testfile
```

Traditional Unix File Access Control

Using **chmod** in absolute mode:

- Number **0** : no permission (`---`)
- Number **1** : execute permission (`--x`)
- Number **2** : write permission (`-w-`)
- Number **3** : execute and write permission (`-wx`)
- Number **4** : read permission (`r--`)
- Number **5** : read and execute permission (`r-x`)
- Number **6** : read and write permission (`rw-`)
- Number **7** : all permissions (`rwx`)

Using **chmod** in absolute mode:

```
$ls -l    testfile
-rwxrwxr--  1 lina  staff 1024  Feb 12 13:00  testfile

$chmod  777    testfile
$ls -l    testfile
-rwxrwxrwx  1 lina  staff 1024  Feb 12 13:00  testfile


$chmod  677    testfile
$ls -l    testfile
-rw-rwxrwx  1 lina  staff 1024  Feb 12 13:00  testfile


$chmod  657    testfile
$ls -l    testfile
-rw-r-xrwx  1 lina  staff 1024  Feb 12 13:00  testfile
```

Traditional Unix File Access Control

Each file/directory is associated with a set of 12 protection bits:

- Nine of the bits specify the read/write/execute permissions for the owner/group/other
- Set User ID (**set UID**) and Set Group ID (**set GID**) are additional two bits for **privilege escalation**.
- Finally, the **sticky** bit:
  - If 0: if a user has write permission to the directory, they can rename/remove files therein
  - If 1: only the file owner, directory owner and root can do so

Example: /usr/bin/passwd

- The **passwd** command changes the password of a user
- To do this, a user must be able to write to **/etc/shadow**
  - File is owned by root, permissions `rw-------`
  - Regular user does not have read or write access to this file
- The **passwd** program has to give the user additional permissions so that they can write to the file **/etc/shadow**.
- This can be done via **set UID** and **set GID** bits.

Privilege Escalation

King's College London — University of London

- A process has a:
  - Real User ID (**RUID**): the user that started the process. This never changes.
  - Effective User ID (**EUID**): the user as which the process appears to run, for permissions purposes.
  - Similarly: Real Group ID, Effective Group ID.
- **Set UID** bit:

  If 1, when the process is started, effective user ID = the owner of the executable.
- **Set GID** bit:

  If 1, when the process is started, effective group ID = the group of the executable.

Privilege Escalation

- Example: when **/usr/bin/passwd** has **set UID** bit=1:
  - Executable file owned by root
  - Hence, **EUID** is then root, so can write to **/etc/shadow**

```
$ls -l  /usr/bin/passwd
-rwsr-xr-x  1 root  root 19031  Feb 7 13:47  /usr/bin/passwd
```

- The Risks of Set ID Bits:
  - Owner=root, **set UID**=1 means the process runs 'as' root (i.e. EUID=0).
  - Running as root permits essentially anything.

Access Control Lists in UNIX

- Main limitation of the owner/group/other model:
  - Must try to capture all use in these three cases
  - One group per file: cannot allow one group to read, another group to read/write
  - Only the owner/root can change permissions: cannot share responsibility with other users
- Many modern UNIX-based operating systems support access control lists

Example: POSIX ACL

- Still have an *owner*, *owning group*, and *other* permissions
- Can grant permissions for additional **named** groups/users
- Mask = best-case out of owning group and named users/groups

Entry Type	**Text Form**
Owner	`user::`*rwx*
Named user	`user:`*name*`:`*rwx*
Owning group	`group::`*rwx*
Named group	`group:`*name*`:`*rwx*
Mask	`mask::`*rwx*
Others	`other::`*rwx*

Example: POSIX ACL

```
$ls -l    testfile
-rwxr-----  1 lina  staff 1024  Feb 12 13:00  testfile

$ getfacl  testfile
# file: testfile
# owner: lina
# group: staff
user::rwx
group::r--
other::---
```

Example: POSIX ACL

```
$ setfacl -m user:mike:rwx testfile

$ getfacl  testfile
# file: testfile
# owner: lina
# group: staff
user::rwx
user:mike:rwx
group::r--
mask::rwx
other::---

$ls -l    testfile
-rwxrwx---+  1 lina  staff 1024  Feb 12 13:00  testfile
```

Example: POSIX ACL

```
$ chmod g-w testfile

$ ls -l testfile
-rwxr-x---+ 1 lina  staff 1024  Feb 12 13:00  testfile

$ getfacl testfile
# file: testfile
# owner: lina
# group: staff
user::rwx
user:mike:rwx           #effective:r-x
group::r--
mask::r-x
other::---
```

The Security Problem

- System is **secure** if resources are used and accessed as intended under all circumstances.
  - Unachievable
- **Intruders (crackers)** attempt to breach security
- **Threat** is potential security violation
- **Attack** is attempt to breach security
- Attack can be accidental or malicious
  - Easier to protect against accidental than malicious misuse

Security Violation Categories

- **Breach of confidentiality**:

  unauthorized reading of data (or theft of information).
- **Breach of integrity**:

  Unauthorized modification of data
- **Breach of availability**:

  Unauthorized destruction of data
- **Theft of service**:

  Unauthorized use of resources
- **Denial of service (DOS)**:

  Preventing legitimate use of the system

Security Violation Methods

- **Masquerading**

  Pretending to be an authorized user to escalate privileges (breach authentication)

- **Replay attack**:

  As is or with **message modification**

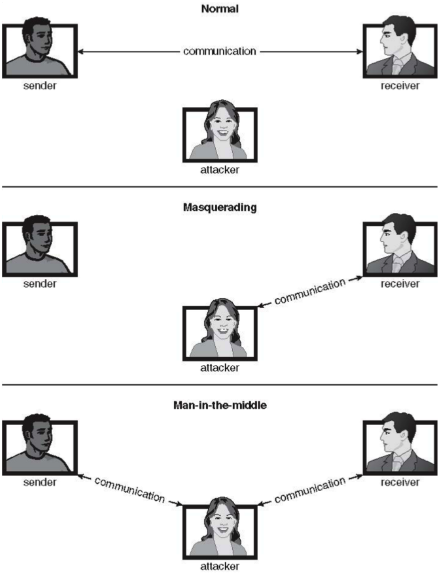
- **Man-in-the-middle attack**:

  Intruder sits in data flow, masquerading as sender to receiver and vice versa

- **Session hijacking**:

  Intercept an already-established session to bypass authentication

Standard Security Attacks



Security Measure Levels

Security must occur at four levels to be effective

- **Physical**:

  Data centers, servers, connected terminals

- **Human**

  Avoid social engineering, phishing, dumpster diving

- **Operating System**:

  Protection mechanisms, debugging

- **Network**:

  Intercepted communications, interruption, DOS

User Authentication

- Crucial to identify user correctly, as protection systems depend on user ID
- User identity most often established through **passwords**
- Password Vulnerabilities:
  - can be guessed
  - accidentally exposed (**shoulder surfing**)
  - **sniffed**
  - illegally transferred
- Passwords must be kept secret
  - Frequent change of passwords
  - History to avoid repeats
  - Use of *non-guessable* passwords