Cybersecurity: Access Control

Jocelyne Elias, Ozalp Babaoglu

Authorization and access control

- From authentication to authorization
  - Once subjects have been authenticated, the next problem to confront is *authorization* or *access control*
- Access control is a central element of computer security whose objectives are:
  - prevent unauthorized users from gaining access to resources,
  - prevent legitimate users from accessing resources in an unauthorized manner,
  - enable legitimate users to access resources in an authorized manner
- Set of *policies* and *mechanisms* that serve to decide if a particular ***subject*** is allowed to perform certain ***operations*** on certain ***objects***

Access control

- Access control is achieved through a set of *policies* and a set of *mechanisms* to enforce the policies
- Access control *policy* dictates what types of access are permitted, under what circumstances, and by whom
- Basic elements of access control are:
  - ***subject***: an entity capable of accessing objects
  - ***object***: a resource to which access needs to be controlled
  - ***access right***: describes the way in which a subject may access an object

Access control policy types

- ***Discretionary access control*** (DAC): access based on the identity of subjects and on access rules stating what subjects are (or are not) allowed to do on which objects. *Discretionary* because subjects decide to grant (or deny) access to other subjects
- ***Mandatory access control*** (MAC): access based on comparing *security labels* (which indicate how sensitive or critical objects are) with *security clearances* of subjects. *Mandatory* because security labels and clearances are set by the system and cannot be modified by subjects
- ***Role-based access control*** (RBAC): access based on the roles that subjects have within the system and on rules stating what accesses are allowed for subjects in given roles

Fundamental principles for security policies

- "Open design"
- "Economy of mechanism"
- "Fail-safe defaults"
  - By default, subjects have no access privileges over any object
- "Complete mediation" (reference monitor)
  - Objects cannot be accessed directly; all accesses must be controlled
- "Least privilege"
  - Subjects have the minimum access privileges that are necessary to carry out the operations that are required for that phase of execution

Fundamental principles for security policies

- ***Least Privilege***: every subject should operate using the *minimum* set of privileges (access rights) that are necessary to perform its task
  - Limits damage that can result from an accident or error
  - Limits number of privileged programs
  - Helps in debugging
  - Increases assurance
  - Allows isolation of critical subsystems
- *Least Privilege* enforced through a *reference monitor* that implements *complete mediation* — every access to every object is checked

Notation

- Let S denote the set of *subjects*
- Let O denote the set of *objects*
  - Note that objects can be active and acts as subjects
- Let α denote the set of *access rights* that subjects have on objects

Access control – Protection domains

- A ***protection domain*** is a set of objects and the set of access rights for each one
- Formally, it is a set of tuples

`<object, set_of_access_rights>`

- Subjects are associated with a given protection domain in which they operate
- The association between subjects and protection domains can be *static* or *dynamic*

Access control – Protection domains

- "Kernel mode" vs "User mode" in operating systems can be seen as two protection domains that control access to main memory
  - Normally processes operate in ***user mode***
  - When they execute a system call, they switch to ***kernel mode*** and gain privileges that are required to carry out the system call
- This is an example of a dynamic association between subject and protection domain

Access Control Matrix model for DAC

- A model for *Discretionary Access Control* (DAC)
- Access Control Matrix
  - is a matrix M with domains as rows and objects as columns
  - each entry $M(i, j)$ contains the set of access rights α that domain D_i permits over object O_j
- When a new object is created
  - add a new column to the matrix
  - the contents of the column decided by the creator of the object

Access Control Matrix – Example

Assume each subject operates in their own protection domain

SUBJECTS	OBJECTS →	File 1	File 2	File 3	File 4
	User A	Own Read Write		Own Read Write	
	User B	Read	Own Read Write	Write	Read
	User C	Read Write	Read		Own Read Write

Access Control Matrix – Example

- User A in domain D_2 editing $File_2$, user B in D_3 editing $File_3$
- Users A and B turn on “spelling corrector” function based on $File_4$ which is a *dictionary*
- The *dictionary* is proprietary and should not be copied

	object / domain	$File_1$	$File_2$	$File_3$	$File_4$
	D_1	read			
A	D_2		read write	read	read
B	D_3			read write	read

But now A and B can make copies of the dictionary

Access Control Matrix – Example

- Introduce a new domain D_4 such that the dictionary can only be read in that domain and add new access right “switch”

	domain \ object	$File_1$	$File_2$	$File_3$	$File_4$	D_4
	D_1	read				
A	D_2		read write	read		switch
B	D_3			read write		switch
	D_4				read	

Access Control Matrix – Example

- But now users A and B cannot access the files they are editing ($File_2$ and $File_3$)
- “Switch” not only changes domains but also copies the access rights from the source domain to the destination domain
- Since there may be multiple users that switch to the same domain, they are kept logically distinct by creating multiple instances of the domain
- This mechanism effectively implements the “principle of least privilege”

Access Control Matrix – Example

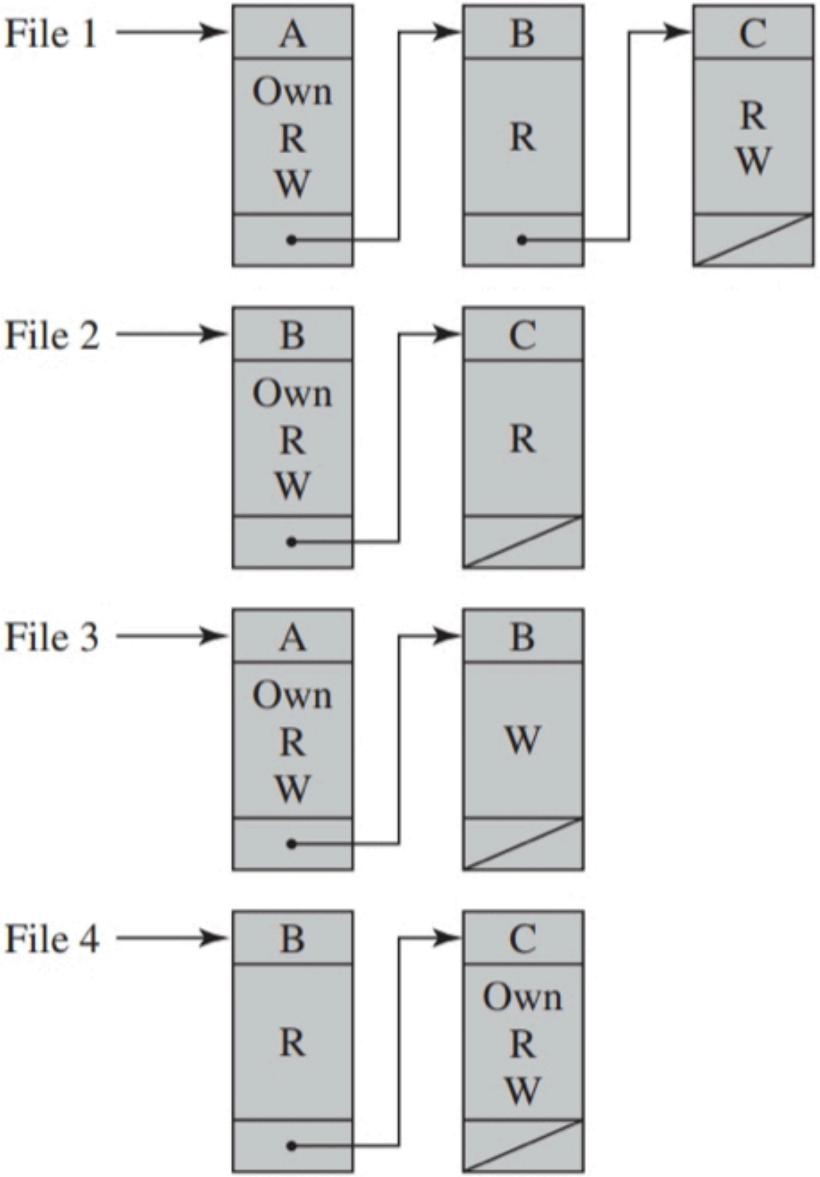
	domain \ object	$File_1$	$File_2$	$File_3$	$File_4$	D_4
	D_1	read				
A	D_2		read write	read		switch
B	D_3			read write		switch
A	D_4		read write	read	read	
B	D_4			read write	read	

Implementation

- As a global table:
  - store the matrix as a 2-dimensional array (table) with entries that are <set_of_access_rights>
- Advantages:
  - simple to implement
- Drawbacks:
  - table can be huge
  - difficult to maintain in a dynamic system where domains and objects are added/deleted and access rights change over time

Access Control Lists

- Access Control List (ACL)
  - the table is stored “per column”
  - with each object, associate a list of tuples that specify access rights for each domain
    `<domain, set_of_access_rights>`
- Optimizations for reducing the length of the list
  - include only domains that have access rights different from a default (e.g., no access)
  - group domains into a (small) number of sets and define access rights only for them
- ACL act like the “guest list” for a party that is checked by a guard at the door to decide who gets to enter



File 1
A
Own
R
W
B
R
C
R
W
File 2
B
Own
R
W
C
R
File 3
A
Own
R
W
B
W
File 4
B
R
C
Own
R
W

Access Control Lists

- Unix example:

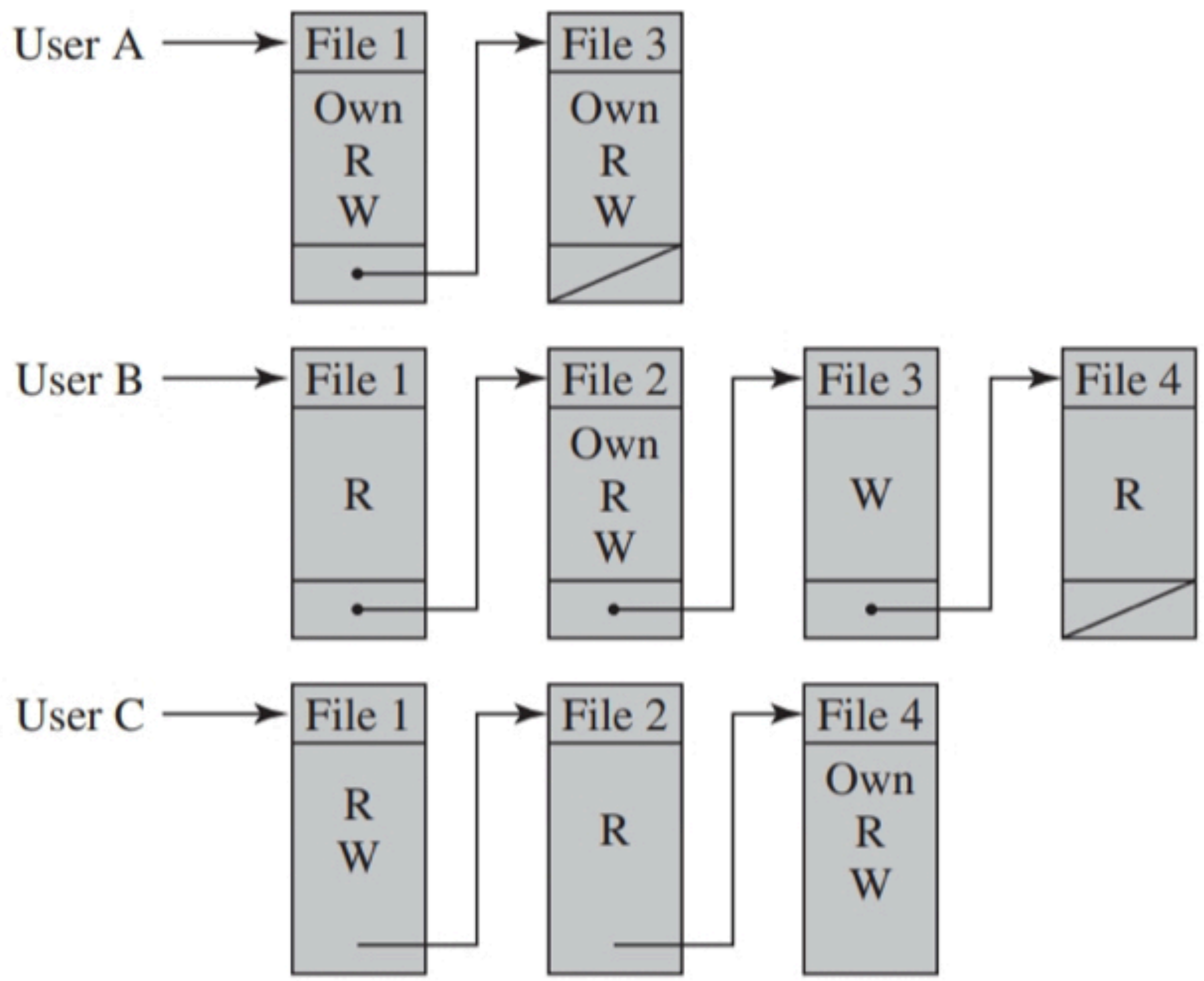
```
babaoglu% ls -l /etc/passwd
-rw-r--r--  1 root wheel 7579 Jan 1 2020 /etc/passwd
```

- Unix has only 3 domains: owner, group, others

Capability

- Capability
  - the table is stored “per row”
  - every domain is associated a list of access “rights”
    `<object, access_rights_for_object>`
  - such a tuple is called a ***capability***
- Who maintains capabilities?
  - processes that “present” them to exercise the access rights over the object
  - capabilities act like keys to open locks protecting objects or invitations that convince “bouncers” guarding a party

Capability


User A
File 1
Own
R
W
File 3
Own
R
W
User B
File 1
R
File 2
Own
R
W
File 3
W
File 4
R
User C
File 1
R
W
File 2
R
File 4
Own
R
W

Capability

For the capability mechanism to function, we must guarantee that:

- processes not be able to forge fake capabilities
- the object (reference monitor) is able to recognize if a capability is fake or authentic
- processes may be permitted or not to copy or transfer their capabilities

Capability implementation

- Capabilities can be implemented using public-key cryptography
- Processes are given capabilities in the form of triples:

`<object, access_rights_for_object, unique_code>`

after being signed with the private key of the object

- Processes can store and observe capabilities but cannot modify them since they cannot sign the modified version because they do not have the object’s private key (similar to certificates)

Capability

- When a process needs to access a resource, it presents to the object the capability it holds for that object
- When an object is presented a capability,
  - it verifies the signature,
  - checks its name,
  - checks the control code,
  - checks that the current access is permitted by the access rights listed in the capability
- N.B. the capability can be *copied* and *transferred* to another process but cannot be modified

Revocation of access rights

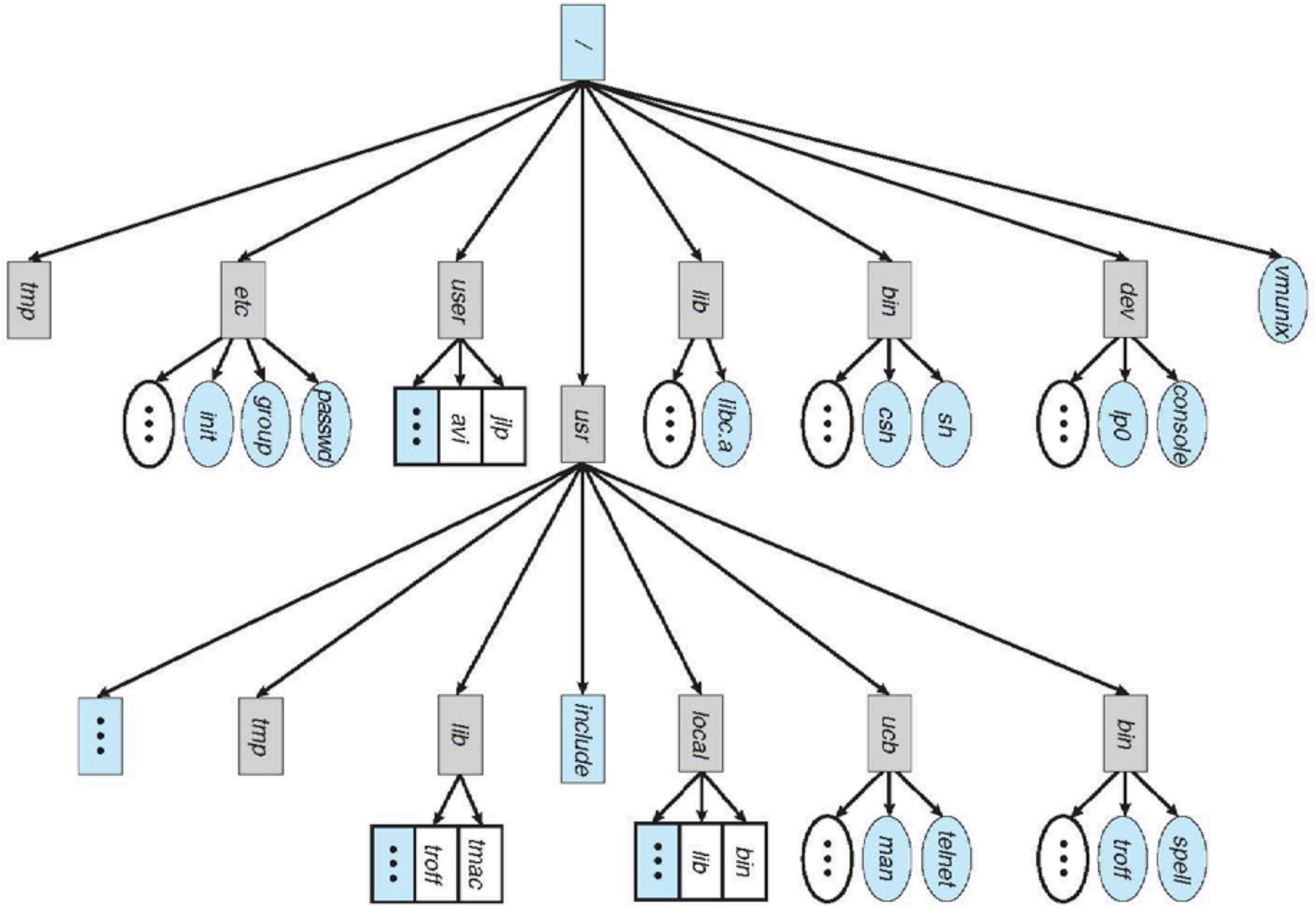
- Revocation can be:
  - immediate or delayed
  - selective or general
  - partial or total (all access rights or some)
  - temporary or permanent
- Revocation in ACL-based systems
  - Easy — it suffices to update the access rights found in the list associated with the object
- Revocation in capability-based systems
  - Difficult — since access rights are not held at the object but are distributed to processes through capabilities, modifying them requires that we first locate them — may be difficult or impossible

Revocation of access rights

- Time-limited capabilities:
  - capabilities have an “expiration date” after which they need to be renewed
  - by not renewing capabilities, we can achieve (delayed) revocation
- Indirect capabilities
  - capabilities do not point directly to objects but to entries in intermediate tables that point to objects
  - by modify the entries in the intermediate table, we can simulate (immediate) revocation

Access control example: UNIX file system

- Every object (resource) in UNIX is a ***file*** with a tree-structured naming scheme (e.g., `/usr/bin/spell`)



Access control example: UNIX file system

- Every file has:
  - owner — the user that created the file
  - group — a collection of users
- Every file has 9 bits of ***access rights*** corresponding to:
  - **r**ead, **w**rite, **e**xecute for ***owner***
  - **r**ead, **w**rite, **e**xecute for ***group***
  - **r**ead, **w**rite, **e**xecute for ***other***
- Examples:
  - **`rw-r--r--`** (644)
  - **`rwxr-xr-x`** (755)

Access control example: UNIX file system

- Users and groups are identified using integers found in the password file
  - *user-id*
  - *group-id*

```
mezzina:x:501:1000:Leonardo Mezzina:/home/mezzina:
trotter:x:502:1000:Guido Trotter:/home/trotter:
```

File ownership

- Each process created by the user (to execute commands) inherits her ***user-id*** and ***group-id*** as the process ***real-user-id*** and ***real-group-id***
- When a process creates a new file, its owner and group are set to the ***real-user-id*** and ***real-group-id*** of the process creating it
- Subsequently, the file's owner can be modified through the command

```
chown newusername file(s)
```

- Typically disabled (limited to root) in systems that maintain file quotas

Real vs Effective User ID

Each process has several IDs associated with it:

- *real-user-id, real-group-id*
  - identify the real user and group that launched the process
  - these values are read from the passwd file
  - do not change during the execution of the process
- *effective-user-id, effective-group-id*
  - set dynamically during the execution of the process through the **`setuid`** mechanism
  - are used to determine the ***access rights*** of the process when interacting with the file system

Hybrid access control

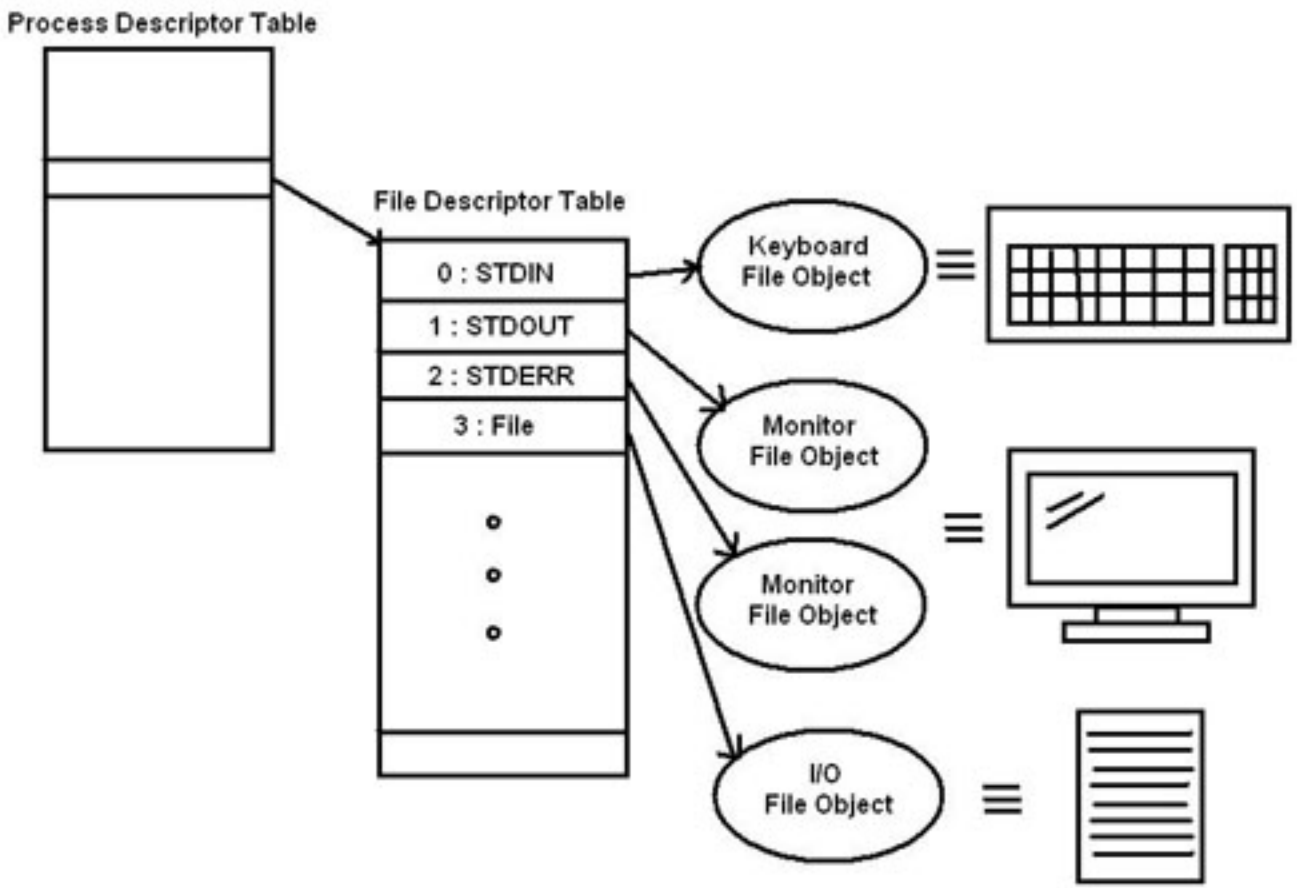
- Often, systems are not ***pure*** ACL-based or ***pure*** Capability-based
- ***Hybrid access control*** combines ACL and Capability mechanisms to obtain the advantages of both:
  - Access control based on identity — ACL
  - Ease of revocation — ACL
  - Efficiency of access — Capability

Hybrid access control example: UNIX

- Open system call
  - `int open(const char *pathname, int flags);`
- where flags is one of
  - `O_RDONLY`
  - `O_WRONLY`
  - `O_RDWR`
- The **`open()`** call checks that the named file exists, that the access requested (***`flags`***) is allowed for ***effective-user-id*** and ***effective-group-id*** of the executing process and returns a (small) integer called a ***file descriptor***
- For execute, there is a separate system call
  - `execv("/bin/cat", args);`

Hybrid access control example: UNIX

- The *file descriptor* returned by the **`open()`** system call is an index into a ***File Descriptor Table*** maintained in kernel space



Hybrid access control example: UNIX

- The *File Descriptor Table* is nothing more than a list of ***capabilities*** corresponding to the files that can be accessed by the process
- A process can use a capability by pointing to it in the *File Descriptor Table* but cannot modify it
- After a file has been **`opened`**, it can be accessed as many times as necessary through the system calls **`read()`** and **`write()`** ***without any further checks***
- In this manner, the cost of verifying access (which is high since it requires reading data structures on disk) is paid only ***once*** and this cost is amortized over many (thousands, millions) of **`read/write`** calls that are fast (do not perform any access control checks)

Hybrid access control example: UNIX

```
int main()
  {
    int fd;
    static char message[] = "Hello, world";

     fd = open(“foo.bar", O_WRONLY);
     if (fd == -1)
      {
         perror(“foo.bar");
         exit (1);
      }
     else
      write(fd, message, sizeof(message));
  }
```

Saved-user-ID

- In addition to ***real-user-id***, ***real-group-id***, ***effective-user-id*** and ***effective-group-id***, each process has a ***saved-user-id*** and ***saved-group-id*** that contain copies of the effective user id and effective group id that existed at the time a **`setuid`** program is executed
- ***saved-user-id*** and ***saved-group-id*** allow the process to return to its effective user/group id once the execution of the **`setuid`** program terminates

Set-user-id, Set-group-id

- Normally:
  - *effective-user-id* and *real-user-id* are the same
  - *effective-group-id* and *real-group-id* are the same
- At the time an executable file with the ***set-user-id*** bit of its permissions set is executed, the following occurs:
  - *saved-user-id* set to ***effective-user-id***
  - *effective-user-id* set to ***user id*** of the file's owner
- At the time an executable file with the ***set-group-id*** bit of its permissions set is executed, the following occurs:
  - *saved-group-id* set to ***effective-group-id***
  - *effective-group-id* set to ***group id*** of the file's owner

Set-user-id, Set-group-id

- These mechanisms allow any user to run the executable with the permissions of the executable's owner or group
- New permissions remain in effect only during the course of the execution
- When the execution terminates, permissions return to their previous state
- Allows a process to change its protection domain dynamically during its execution
- Can be used to implement "principle of least privilege"

Set-user-id example

- How to implement a command that allows users to change their passwords?
- A user *should* be able to change her own password, but should *not* be able to see (or modify) the passwords of others
- But in Unix, permissions are at the granularity of an entire file
- It is not possible to define permissions at the granularity of individual records (lines within the **`/etc/passwd`** file)
- To allow any user to modify her password, the permissions of the **`/etc/passwd`** file must be set to “read/write by all”
- But now anyone can see (and modify) the password of anyone else

Set-user-id example

- Use of the **`setuid`** mechanism to solve the password problem:
  - Root writes a command **`/bin/passwd`** that is owned by **`root`** with permissions `r-s--x--x` (the **`setuid`** bit is on)
  - The file **`/etc/passwd`** is owned by **`root`** with permissions `rw-------` (read/write root only)
  - When **`/bin/passwd`** is executed by a process, its *effective-user-id* changes to **`root`**
  - Therefore, the process can write the file **`/etc/passwd`** but only after having made all necessary checks implemented by the command **`/bin/passwd`**