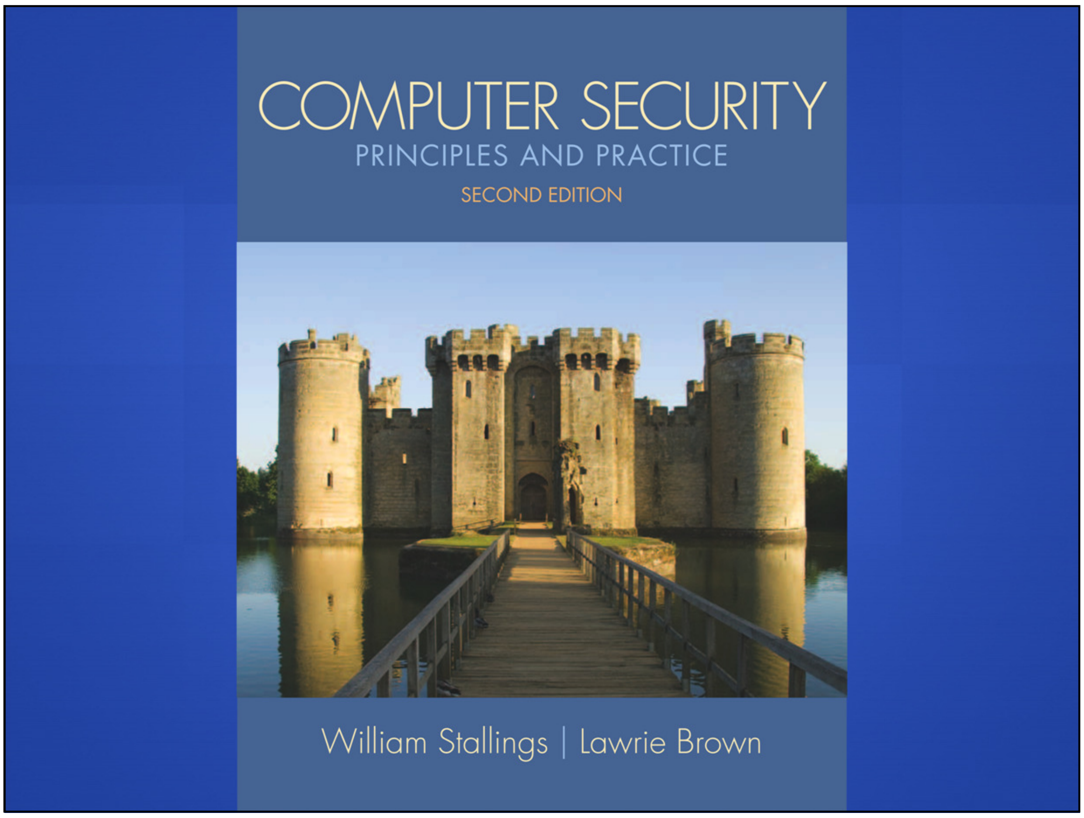
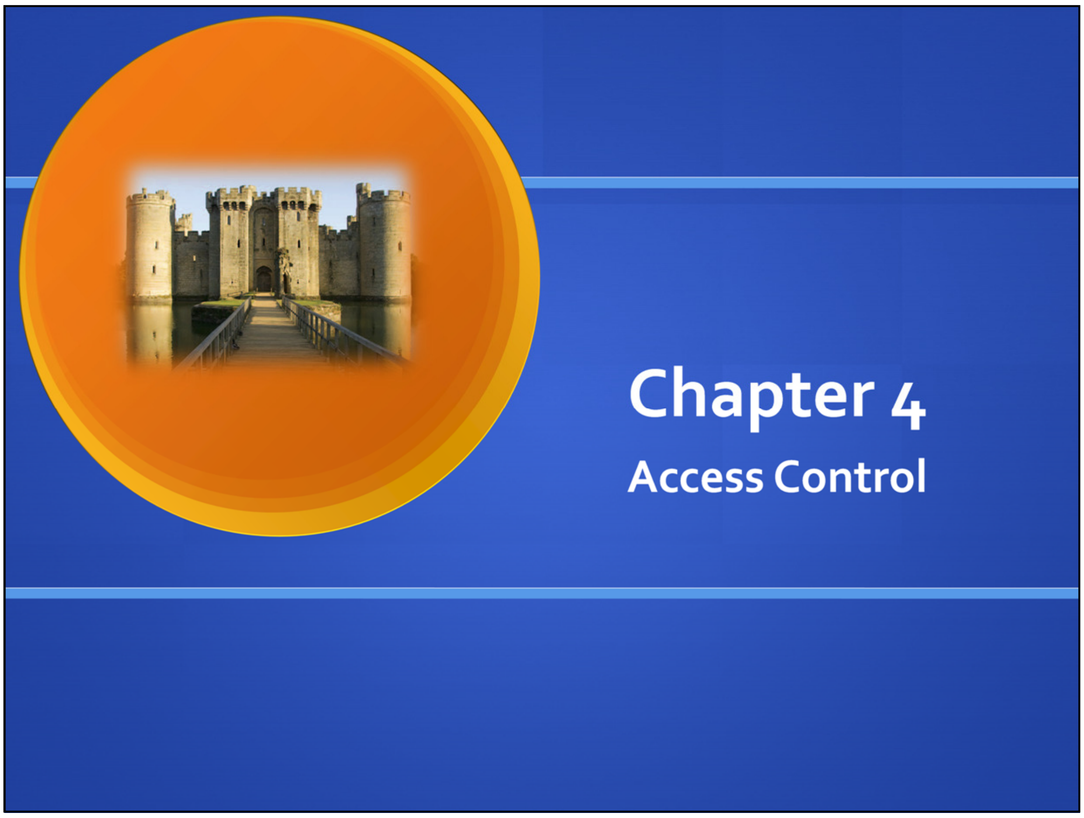
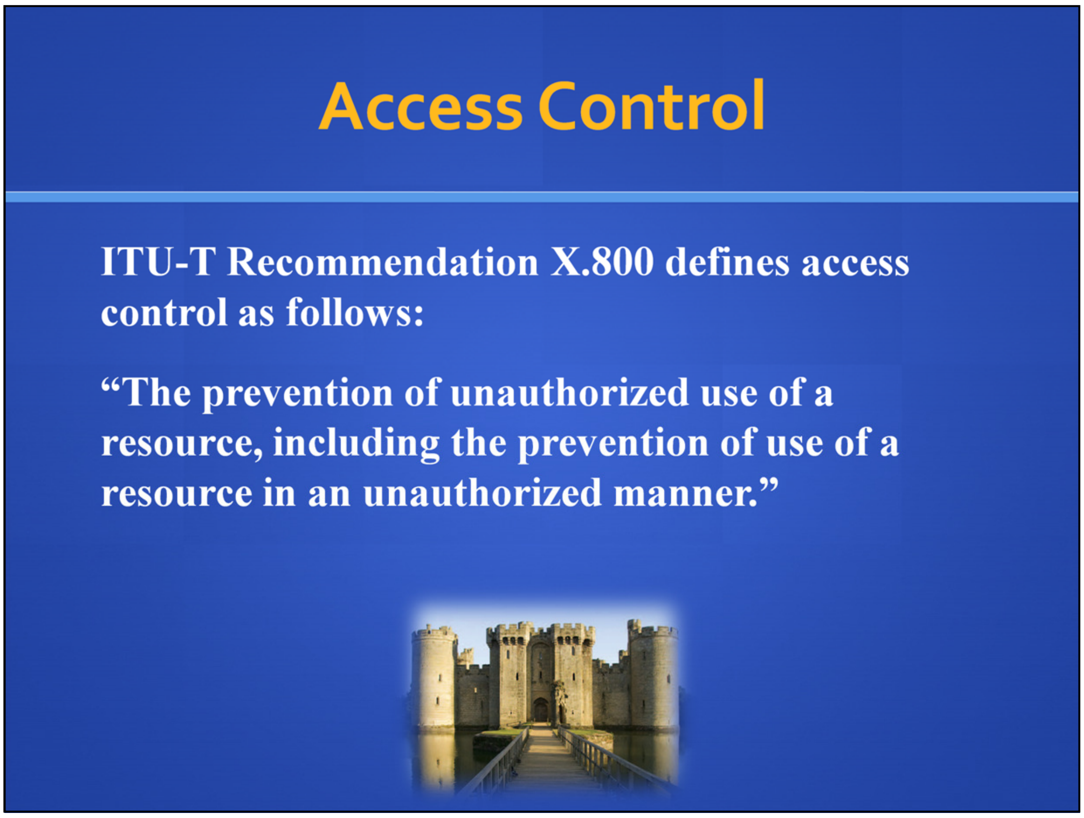


Lecture slides prepared for “Computer Security: Principles and Practice”, 2/e, by William Stallings and Lawrie Brown, Chapter 4 “Overview”.



# Chapter 4
## Access Control

This chapter focuses on access control enforcement within a computer system.
The chapter considers the situation of a population of users and user groups that are able to authenticated to a system and are then assigned access rights to certain resources
on the system. A more general problem is a network or Internet-based environment, in which there are a number of client systems, a number of server systems, and a number of users who may access servers via one or more of the client systems. This more general
context introduces new security issues and results in more complex solutions than those addressed in this chapter. We cover these topics in Chapter 23.



## Access Control

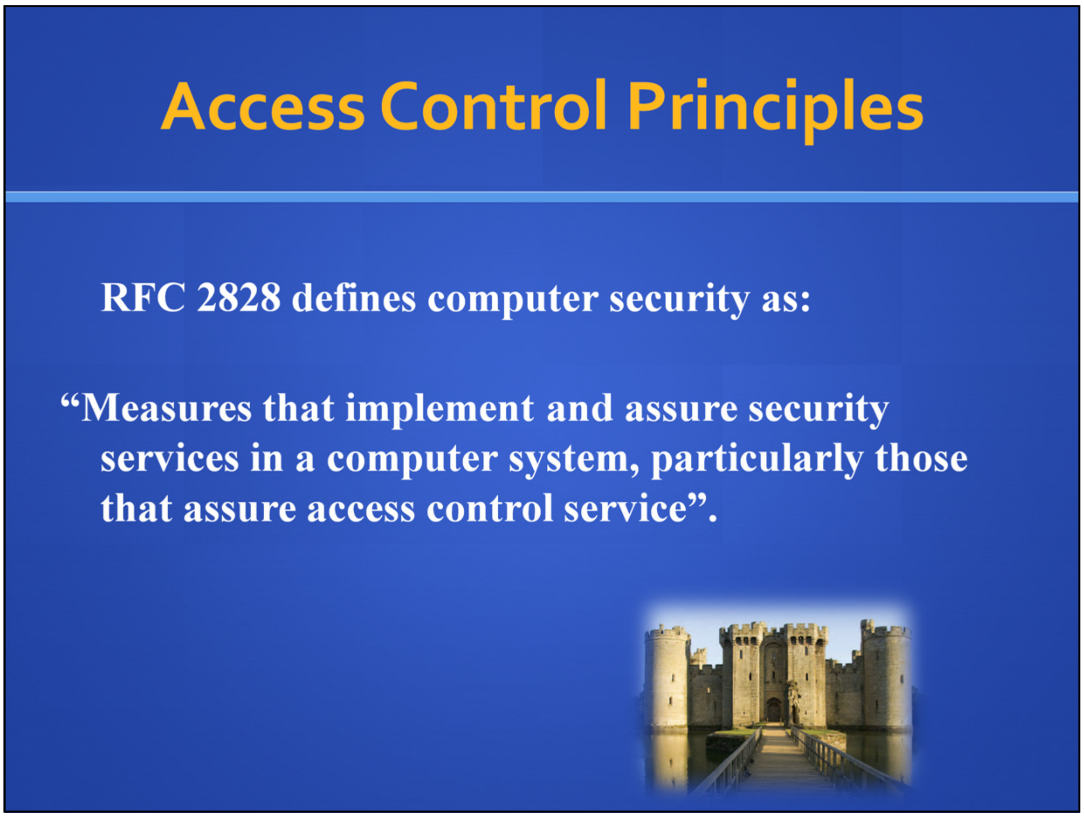
ITU-T Recommendation X.800 defines access control as follows:

"The prevention of unauthorized use of a resource, including the prevention of use of a resource in an unauthorized manner."

ITU-T Recommendation X.800 defines access control as follows:

**Access Control: The prevention of unauthorized use of a resource, including the**
prevention of use of a resource in an unauthorized manner

We can view access control as the central element of computer security. The principal objectives of computer security are to prevent unauthorized users from gaining access to resources, to prevent legitimate users from accessing resources in an unauthorized manner, and to enable legitimate users to access resources in an authorized manner.



## Access Control Principles

**RFC 2828 defines computer security as:**

**"Measures that implement and assure security services in a computer system, particularly those that assure access control service".**

In a broad sense, all of computer security is concerned with access control. Indeed,

RFC 2828 defines computer security as follows: Measures that implement and assure

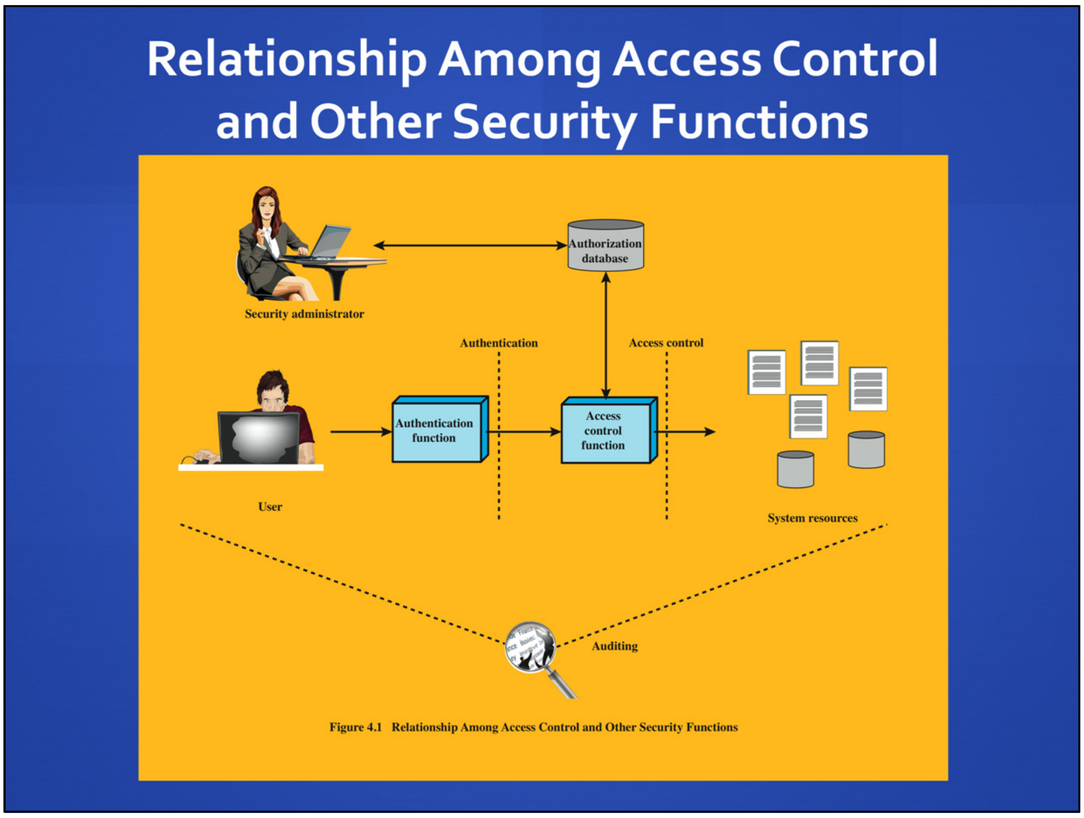
security services in a computer system, particularly those that assure access control

service. This chapter deals with a narrower, more specific concept of access control:

Access control implements a security policy that specifies who or what (e.g., in the

case of a process) may have access to each specific system resource and the type of

access that is permitted in each instance.



## Relationship Among Access Control and Other Security Functions

Figure 4.1 Relationship Among Access Control and Other Security Functions

Figure 4.1 shows a broader context of access control. In addition to access control, this context involves the following entities and functions:

• **Authentication: Verification that the credentials of a user or other system** entity are valid.

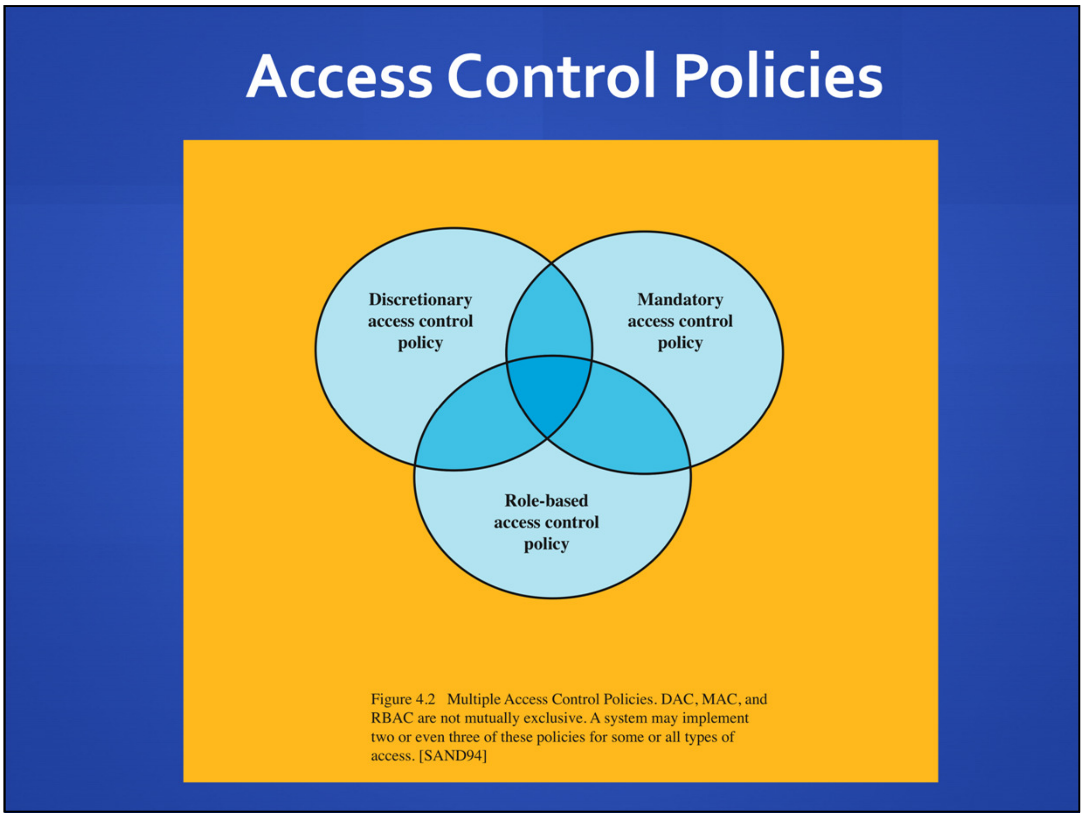
**Authorization: The granting of a right or permission to a system entity to** access a system resource. This function determines who is trusted for a given purpose.

• **Audit: An independent review and examination of system records and activities**
in order to test for adequacy of system controls, to ensure compliance with established policy and operational procedures, to detect breaches in security, and to recommend any indicated changes in control, policy and procedures.

An access control mechanism mediates between a user (or a process executing on behalf of a user) and system resources, such as applications, operating systems,

firewalls, routers, files, and databases. The system must first authenticate an entity
seeking access. Typically, the authentication function determines whether the user
is permitted to access the system at all. Then the access control function determines
if the specific requested access by this user is permitted. A security administrator
maintains an authorization database that specifies what type of access to which
resources is allowed for this user. The access control function consults this database
to determine whether to grant access. An auditing function monitors and keeps a
record of user accesses to system resources.

In the simple model of Figure 4.1, the access control function is shown as
a single logical module. In practice, a number of components may cooperatively
share the access control function. All operating systems have at least a rudimentary,
and in many cases a quite robust, access control component. Add-on security
packages can supplement the native access control capabilities of the OS. Particular
applications or utilities, such as a database management system, also incorporate
access control functions. External devices, such as firewalls, can also provide access
control services.


Discretionary access control policy
Mandatory access control policy
Role-based access control policy

## Access Control Policies

Figure 4.2 Multiple Access Control Policies. DAC, MAC, and RBAC are not mutually exclusive. A system may implement two or even three of these policies for some or all types of access. [SAND94]

An access control policy, which can be embodied in an authorization database, dictates what types of access are permitted, under what circumstances, and by whom. Access control policies are generally grouped into the following categories:

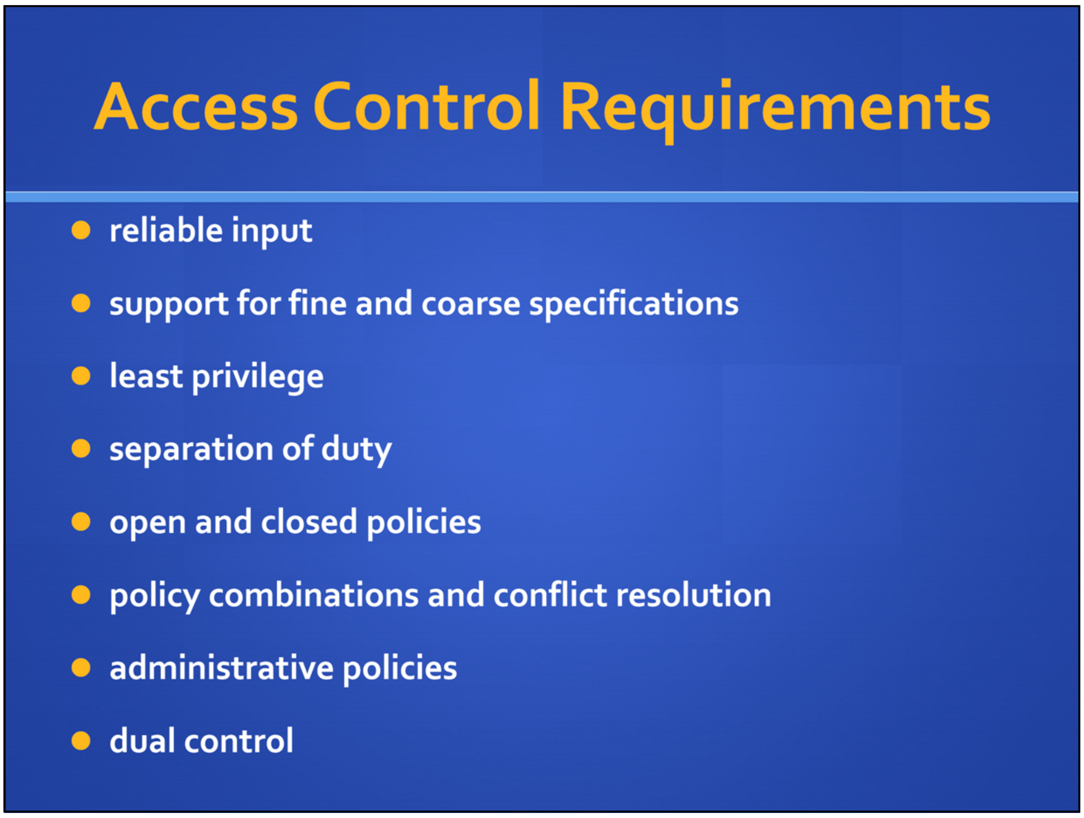
• **Discretionary access control (DAC): Controls access based on the identity**
of the requestor and on access rules (authorizations) stating what requestors are (or are not) allowed to do. This policy is termed *discretionary because an* entity might have access rights that permit the entity, by its own volition, to enable another entity to access some resource.

• **Mandatory access control (MAC): Controls access based on comparing** security labels (which indicate how sensitive or critical system resources are) with security clearances (which indicate system entities are eligible to access certain resources). This policy is termed *mandatory because an entity that has* clearance to access a resource may not, just by its own volition, enable another entity to access that resource.

• **Role-based access control (RBAC): Controls access based on the roles that**
users have within the system and on rules stating what accesses are allowed to users in given roles.

DAC is the traditional method of implementing access control, and is examined in Section 4.3. MAC is a concept that evolved out of requirements for military information security and is best covered in the context of trusted systems, which we
deal with in Chapter 13. RBAC has become increasingly popular and is covered in Section 4.5.

These three policies are not mutually exclusive (Figure 4.2). An access control mechanism can employ two or even all three of these policies to cover different classes of system resources.



## Access Control Requirements

- reliable input
- support for fine and coarse specifications
- least privilege
- separation of duty
- open and closed policies
- policy combinations and conflict resolution
- administrative policies
- dual control

[VIME06] lists the following concepts and features that should be supported by an
access control system.

• **Reliable input: The old maxim garbage-in-garbage-out applies with special**
force to access control. An access control system assumes that a user is authentic; thus, an authentication mechanism is needed as a front end to an access control system. Other inputs to the access control system must also be reliable. For example, some access control restrictions may depend on an address, such as a source IP address or medium access control address. The overall system must have a means of determining the validity of the source for such restrictions to operate effectively.

• **Support for fine and coarse specifications: The access control system should**
support fine-grained specifications, allowing access to be regulated at the level of individual records in files, and individual fields within records. The system should also support fine-grained specification in the sense of controlling each individual

access by a user rather than a sequence of access requests. System administrators
should also be able to choose coarse-grained specification for some classes of
resource access, to reduce administrative and system processing burden.

• **Least privilege: This is the principle that access control should be implemented**
so that each system entity is granted the minimum system resources and authorizations
that the entity needs to do its work. This principle tends to limit damage
that can be caused by an accident, error, or fraudulent or unauthorized act.

• **Separation of duty: This is the practice of dividing the steps in a system function**
among different individuals, so as to keep a single individual from subverting the
process. This is primarily a policy issue; separation of duty requires the appropriate
power and flexibility in the access control system, including least privilege and
fine-grained access control. Another useful tool is history-based authorization,
which makes access dependent on previously executed accesses.

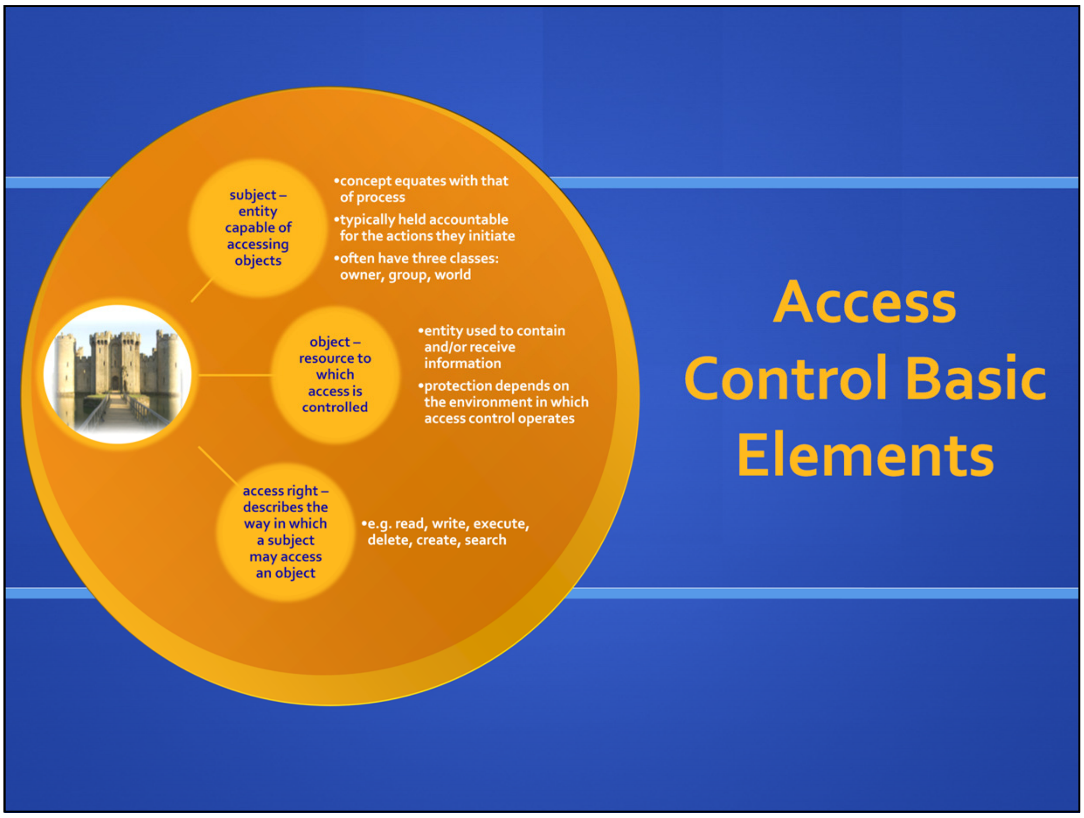
• **Open and closed policies: The most useful, and most typical, class of access**
control policies are closed policies. In a closed policy, only accesses that
are specifically authorized are allowed. In some applications, it may also be
desirable to allow an open policy for some classes of resources. In an open
policy, authorizations specify which accesses are prohibited; all other accesses
are allowed.

• **Policy combinations and conflict resolution: An access control mechanism**
may apply multiple policies to a given class of resources. In this case, care must
be taken that there are no conflicts such that one policy enables a particular
access while another policy denies it. Or, if such a conflict exists, a procedure
must be defined for conflict resolution.

• **Administrative policies: As was mentioned, there is a security administration**
function for specifying the authorization database that acts as an input to the

access control function. Administrative policies are needed to specify who can add, delete, or modify authorization rules. In turn, access control and other control mechanisms are needed to enforce the administrative policies.

• **Dual control: When a task requires two or more individuals working in tandem.**



## Access Control Basic Elements

subject – entity capable of accessing objects
- concept equates with that of process
- typically held accountable for the actions they initiate
- often have three classes: owner, group, world

object – resource to which access is controlled
- entity used to contain and/or receive information
- protection depends on the environment in which access control operates

access right – describes the way in which a subject may access an object
- e.g. read, write, execute, delete, create, search

The basic elements of access control are: subject, object, and access right.

A **subject is an entity capable of accessing objects. Generally, the concept of**

subject equates with that of process. Any user or application actually gains access to

an object by means of a process that represents that user or application. The process

takes on the attributes of the user, such as access rights.

A subject is typically held accountable for the actions they have initiated,

and an audit trail may be used to record the association of a subject with security relevant

actions performed on an object by the subject.

Basic access control systems typically define three classes of subject, with

different access rights for each class:

• **Owner: This may be the creator of a resource, such as a file. For system resources,** ownership may belong to a system administrator. For project resources, a project administrator or leader may be assigned ownership.

• **Group: In addition to the privileges assigned to an owner, a named group of**
users may also be granted access rights, such that membership in the group is sufficient to exercise these access rights. In most schemes, a user may belong to multiple groups.

• **World: The least amount of access is granted to users who are able to access the**
system but are not included in the categories owner and group for this resource.

An **object is a resource to which access is controlled. In general, an object**
is an entity used to contain and/or receive information. Examples include records, blocks, pages, segments, files, portions of files, directories, directory trees, mailboxes,
messages, and programs. Some access control systems also encompass, bits, bytes, words, processors, communication ports, clocks, and network nodes.

The number and types of objects to be protected by an access control system depends on the environment in which access control operates and the desired tradeoff
between security on the one hand and complexity, processing burden, and ease of use on the other hand.

An **access right describes the way in which a subject may access an object.**
Access rights could include the following:

• **Read: User may view information in a system resource (e.g., a file, selected**
records in a file, selected fields within a record, or some combination). Read access includes the ability to copy or print.

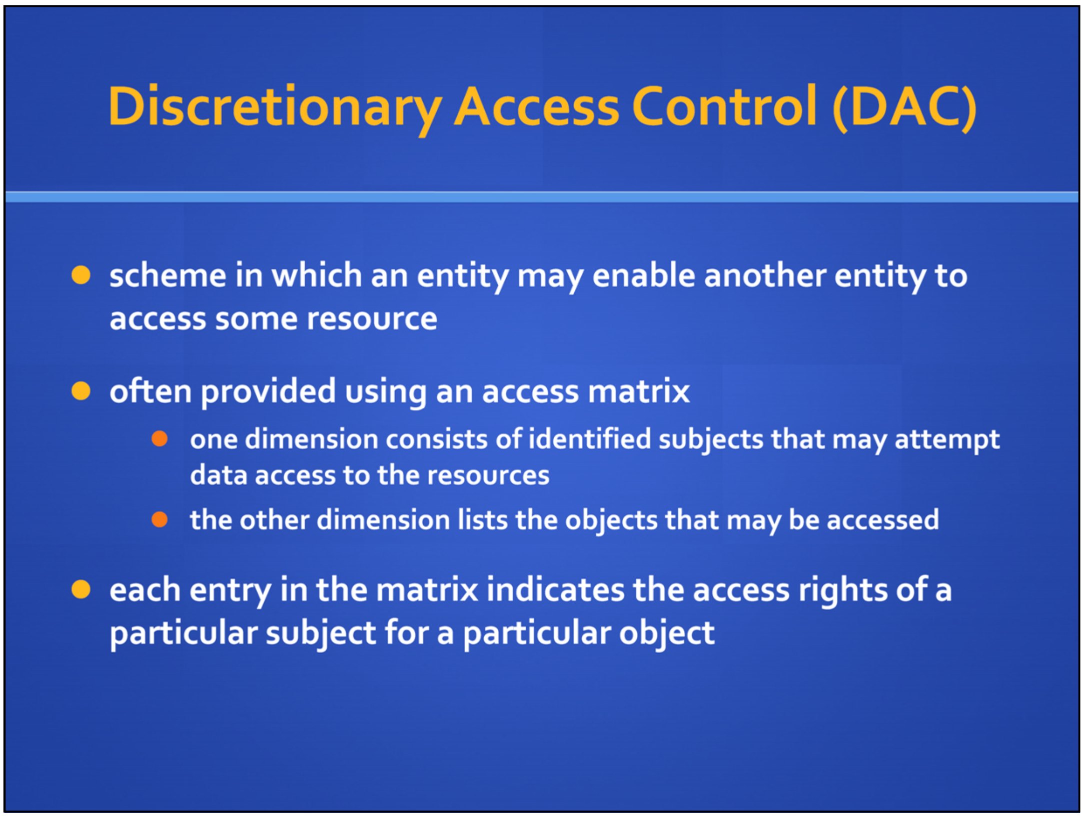
• **Write: User may add, modify, or delete data in system resource (e.g., files,**
records, programs). Write access includes read access.

• **Execute: User may execute specified programs.**

**• Delete: User may delete certain system resources, such as files or records.**

**• Create: User may create new files, records, or fields.**

**• Search: User may list the files in a directory or otherwise search the directory.**



## Discretionary Access Control (DAC)

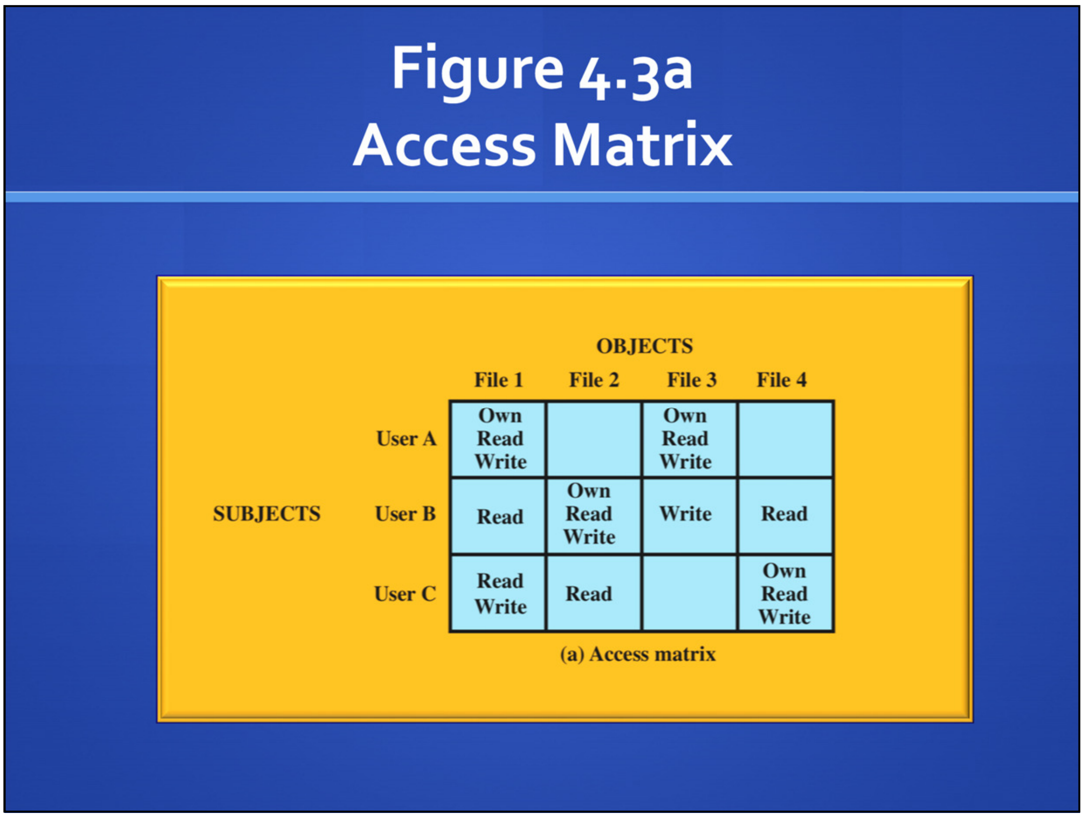
- scheme in which an entity may enable another entity to access some resource
- often provided using an access matrix
  - one dimension consists of identified subjects that may attempt data access to the resources
  - the other dimension lists the objects that may be accessed
- each entry in the matrix indicates the access rights of a particular subject for a particular object

As was previously stated, a discretionary access control scheme is one in which an
entity may be granted access rights that permit the entity, by its own volition, to
enable another entity to access some resource. A general approach to DAC, as
exercised by an operating system or a database management system, is that of an
**access matrix. The access matrix concept was formulated by Lampson [LAMP69,**
LAMP71], and subsequently refined by Graham and Denning [GRAH72, DENN71]
and by Harrison et al. [HARR76].

One dimension of the matrix consists of identified subjects that may attempt
data access to the resources. Typically, this list will consist of individual users or
user groups, although access could be controlled for terminals, network equipment,
hosts, or applications instead of or in addition to users. The other dimension lists
the objects that may be accessed. At the greatest level of detail, objects may be
individual data fields. More aggregate groupings, such as records, files, or even the

entire database, may also be objects in the matrix. Each entry in the matrix indicates

the access rights of a particular subject for a particular object.



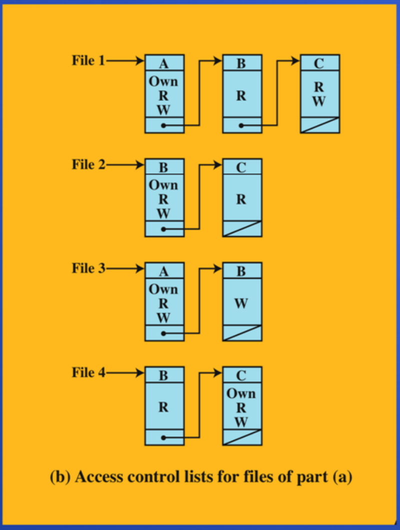
## Figure 4.3a
## Access Matrix

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

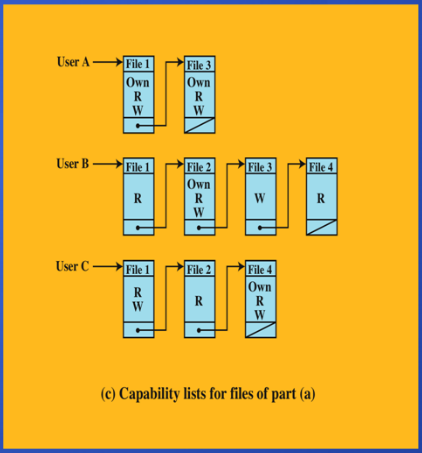
(a) Access matrix

Figure 4.3a, based on a figure in [SAND94], is a simple example of an access matrix. Thus, user A owns files 1 and 3 and has read and write access rights to those
files. User B has read access rights to file 1, and so on.

## Figures 4.3b and c
### Example of Access Control Structures



(b) Access control lists for files of part (a)



(c) Capability lists for files of part (a)

In practice, an access matrix is usually sparse and is implemented by decomposition

in one of two ways. The matrix may be decomposed by columns, yielding

**access control lists (ACLs); see Figure 4.3b. For each object, an ACL lists users and**

their permitted access rights. The ACL may contain a default, or public, entry. This

allows users that are not explicitly listed as having special rights to have a default

set of rights. The default set of rights should always follow the rule of least privilege

or read-only access, whichever is applicable. Elements of the list may include

individual users as well as groups of users.

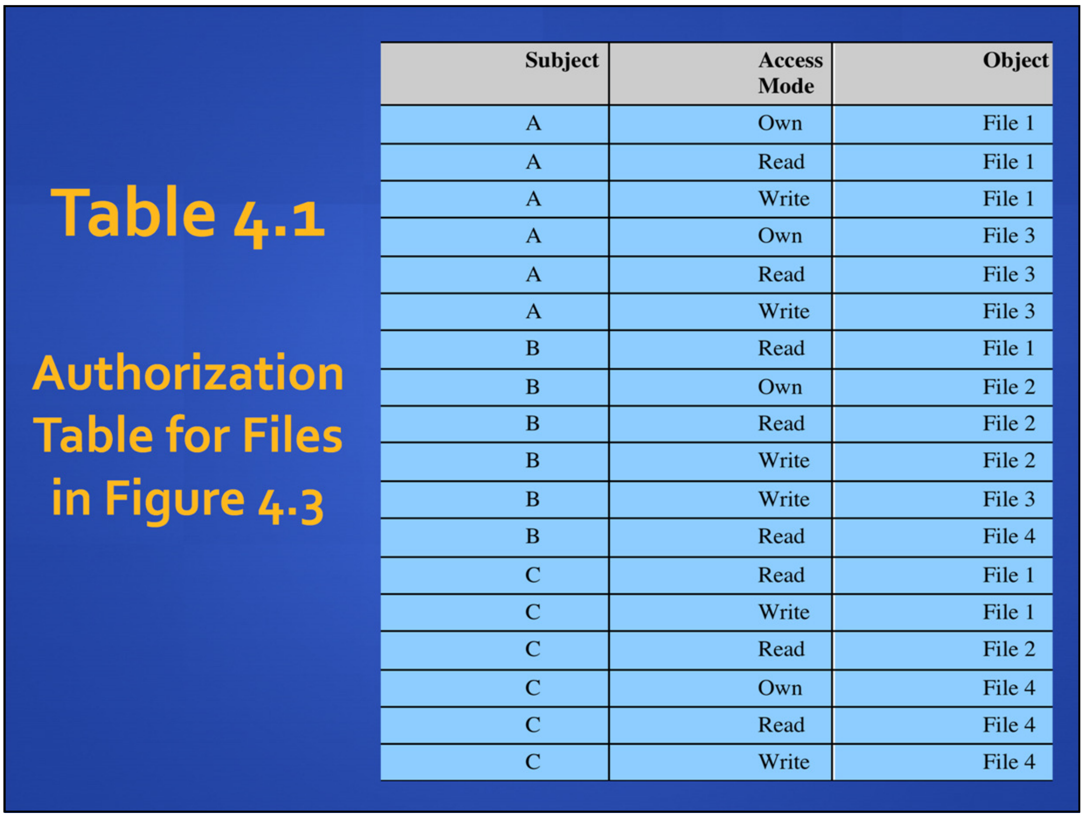
When it is desired to determine which subjects have which access rights to a particular

resource, ACLs are convenient, because each ACL provides the information

for a given resource. However, this data structure is not convenient for determining

the access rights available to a specific user.

Decomposition by rows yields **capability tickets (Figure 4.3c). A capability** ticket specifies authorized objects and operations for a particular user. Each user has a number of tickets and may be authorized to loan or give them to others. Because tickets may be dispersed around the system, they present a greater security
problem than access control lists. The integrity of the ticket must be protected, and guaranteed (usually by the operating system). In particular, the ticket must be unforgettable. One way to accomplish this is to have the operating system hold all tickets on behalf of users. These tickets would have to be held in a region of memory inaccessible to users. Another alternative is to include an unforgeable token in the capability. This could be a large random password, or a cryptographic message authentication code. This value is verified by the relevant resource whenever
access is requested. This form of capability ticket is appropriate for use in a distributed environment, when the security of its contents cannot be guaranteed. The convenient and inconvenient aspects of capability tickets are the opposite of those for ACLs. It is easy to determine the set of access rights that a given user has, but more difficult to determine the list of users with specific access rights for a specific resource.



## Table 4.1

## Authorization Table for Files in Figure 4.3

| Subject | Access Mode | Object |
|---|---|---|
| A | Own | File 1 |
| A | Read | File 1 |
| A | Write | File 1 |
| A | Own | File 3 |
| A | Read | File 3 |
| A | Write | File 3 |
| B | Read | File 1 |
| B | Own | File 2 |
| B | Read | File 2 |
| B | Write | File 2 |
| B | Write | File 3 |
| B | Read | File 4 |
| C | Read | File 1 |
| C | Write | File 1 |
| C | Read | File 2 |
| C | Own | File 4 |
| C | Read | File 4 |
| C | Write | File 4 |

[SAND94] proposes a data structure that is not sparse, like the access matrix,
but is more convenient than either ACLs or capability lists (Table 4.1). An authorization
table contains one row for one access right of one subject to one resource.
Sorting or accessing the table by subject is equivalent to a capability list. Sorting or
accessing the table by object is equivalent to an ACL. A relational database can
easily implement an authorization table of this type.

This section introduces a general model for DAC developed by Lampson, Graham,
and Denning [LAMP71, GRAH72, DENN71]. The model assumes a set of subjects,
a set of objects, and a set of rules that govern the access of subjects to objects. Let us
define the protection state of a system to be the set of information, at a given point in
time, that specifies the access rights for each subject with respect to each object. We can
identify three requirements: representing the protection state, enforcing access rights,

and allowing subjects to alter the protection state in certain ways. The model addresses
all three requirements, giving a general, logical description of a DAC system.

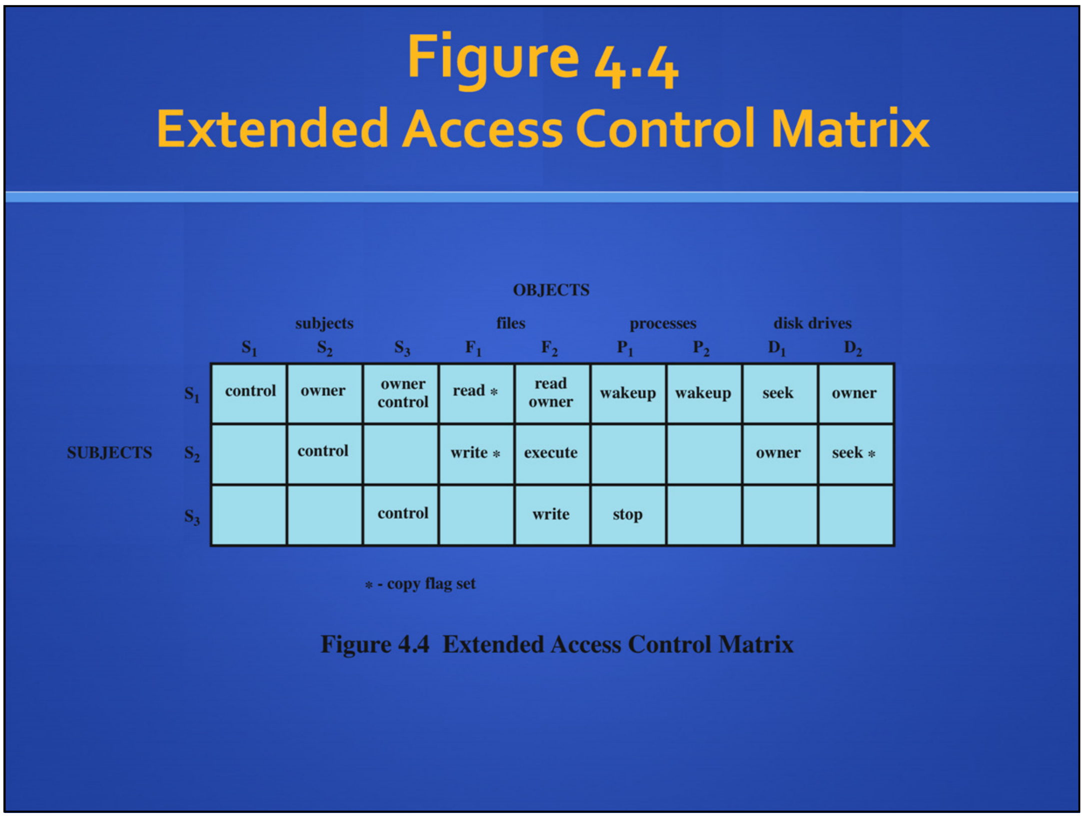
To represent the protection state, we extend the universe of objects in the access control matrix to include the following:

• **Processes: Access rights include the ability to delete a process, stop (block),**
and wake up a process.

• **Devices: Access rights include the ability to read/write the device, to control**
its operation (e.g., a disk seek), and to block/unblock the device for use.

• **Memory locations or regions: Access rights include the ability to read/write**
certain regions of memory that are protected such that the default is to disallow access.

• **Subjects: Access rights with respect to a subject have to do with the ability**
to grant or delete access rights of that subject to other objects, as explained subsequently.



# Figure 4.4
## Extended Access Control Matrix

OBJECTS

| | | subjects | | | files | | processes | | disk drives | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $S_1$ | $S_2$ | $S_3$ | $F_1$ | $F_2$ | $P_1$ | $P_2$ | $D_1$ | $D_2$ |
| | $S_1$ | control | owner | owner control | read * | read owner | wakeup | wakeup | seek | owner |
| SUBJECTS | $S_2$ | | control | | write * | execute | | | owner | seek * |
| | $S_3$ | | | control | | write | stop | | | |

* - copy flag set

Figure 4.4 Extended Access Control Matrix

Figure 4.4 is an example. For an access control matrix *A, each entry A[S, X]* contains strings, called access attributes, that specify the access rights of subject *S to* object *X. For example, in Figure 4.4, $S_1$ may read file $F_2$, because 'read' appears in $A[S_1, F_1]$.*

From a logical or functional point of view, a separate access control module is associated with each type of object (Figure 4.5). The module evaluates each request by a subject to access an object to determine if the access right exists. An access attempt triggers the following steps:

**1. A subject $S_0$ *issues a request of type α for object X.***

**2. The request causes the system (the operating system or an access control interface**
module of some sort) to generate a message of the form ($S_0$*, α, X) to the* controller for *X.*

**3. The controller interrogates the access matrix A to determine if α is in *$A[S_0, X]$.***
If so, the access is allowed; if not, the access is denied and a protection violation occurs. The violation should trigger a warning and appropriate action.

## Figure 4.5

## Access Control Function

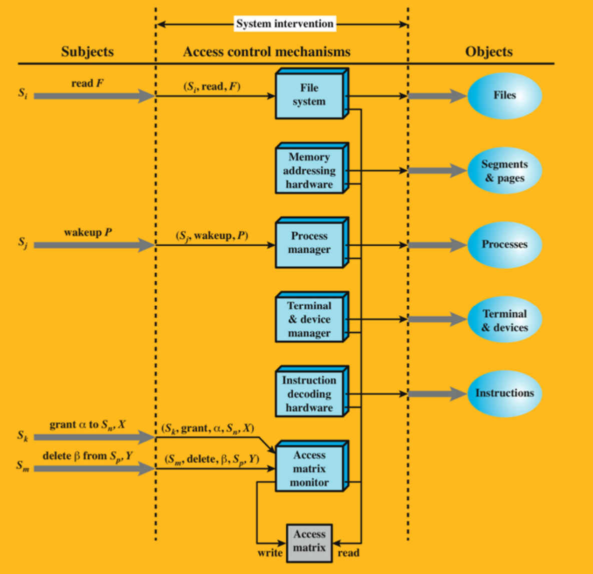
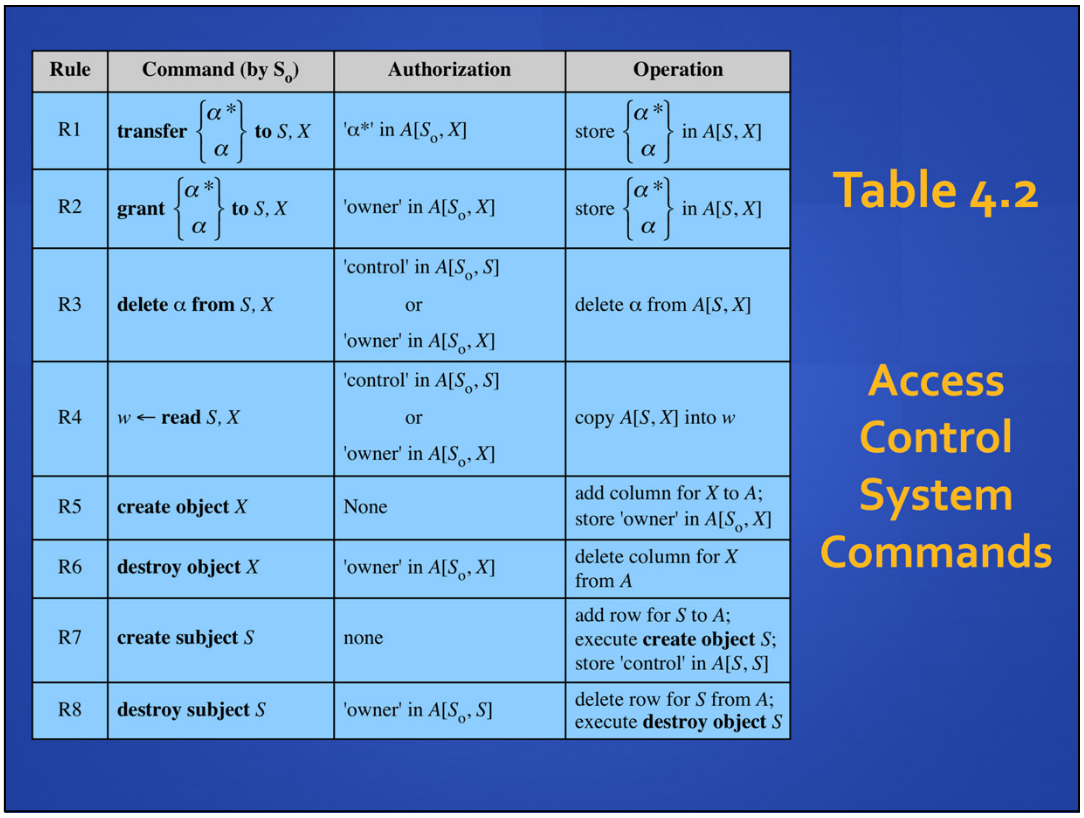


**Figure 4.5 An Organization of the Access Control Function**

Figure 4.5 suggests that every access by a subject to an object is mediated
by the controller for that object, and that the controller's decision is based on the
current contents of the matrix. In addition, certain subjects have the authority to
make specific changes to the access matrix. A request to modify the access
matrix is
treated as an access to the matrix, with the individual entries in the matrix treated
as
objects. Such accesses are mediated by an access matrix controller, which
controls
updates to the matrix.



| Rule | Command (by $S_o$) | Authorization | Operation |
|---|---|---|---|
| R1 | **transfer** $\left\{\begin{matrix}\alpha^*\\ \alpha\end{matrix}\right\}$ **to** $S, X$ | '$\alpha^*$' in $A[S_o, X]$ | store $\left\{\begin{matrix}\alpha^*\\ \alpha\end{matrix}\right\}$ in $A[S, X]$ |
| R2 | **grant** $\left\{\begin{matrix}\alpha^*\\ \alpha\end{matrix}\right\}$ **to** $S, X$ | 'owner' in $A[S_o, X]$ | store $\left\{\begin{matrix}\alpha^*\\ \alpha\end{matrix}\right\}$ in $A[S, X]$ |
| R3 | **delete** $\alpha$ **from** $S, X$ | 'control' in $A[S_o, S]$ or 'owner' in $A[S_o, X]$ | delete $\alpha$ from $A[S, X]$ |
| R4 | $w \leftarrow$ **read** $S, X$ | 'control' in $A[S_o, S]$ or 'owner' in $A[S_o, X]$ | copy $A[S, X]$ into $w$ |
| R5 | **create object** $X$ | None | add column for $X$ to $A$; store 'owner' in $A[S_o, X]$ |
| R6 | **destroy object** $X$ | 'owner' in $A[S_o, X]$ | delete column for $X$ from $A$ |
| R7 | **create subject** $S$ | none | add row for $S$ to $A$; execute **create object** $S$; store 'control' in $A[S, S]$ |
| R8 | **destroy subject** $S$ | 'owner' in $A[S_o, S]$ | delete row for $S$ from $A$; execute **destroy object** $S$ |

## Table 4.2

## Access Control System Commands

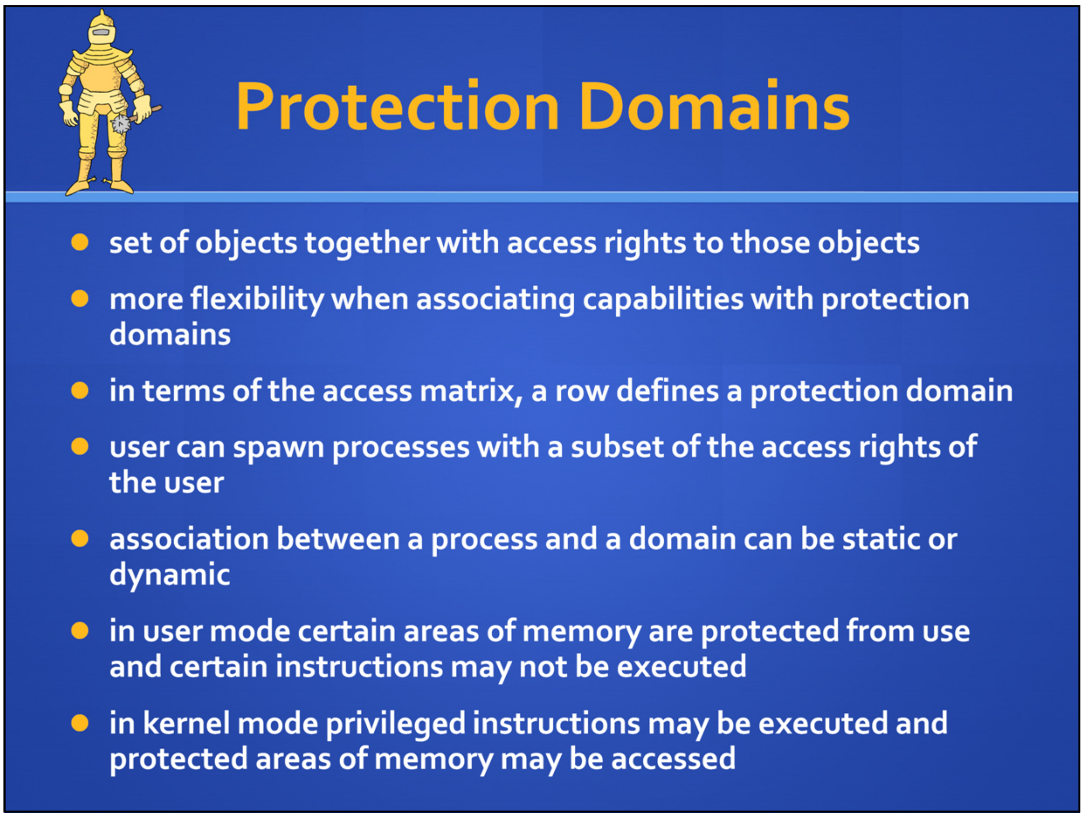
The model also includes a set of rules that govern modifications to the access matrix, shown in Table 4.2. For this purpose, we introduce the access rights 'owner' and 'control' and the concept of a copy flag, explained in the subsequent paragraphs.

The first three rules deal with transferring, granting, and deleting access rights. Suppose that the entry α* exists in *$A[S_0, X]$. This means that $S_0$ has access right α to* subject *X and, because of the presence of the copy flag, can transfer this right, with* or without copy flag, to another subject. Rule R1 expresses this capability. A subject would transfer the access right without the copy flag if there were a concern that the new subject would maliciously transfer the right to another subject that should not have that access right. For example, *$S_1$ may place 'read' or 'read*' in any matrix* entry in the *$F_1$ column. Rule R2 states that if $S_0$ is designated as the owner of object X, then $S_0$ can grant an access right to that object for any other subject. Rule 2 states* that *$S_0$ can add any access right to A[S, X] for any S, if $S_0$ has 'owner' access to x.* Rule R3 permits *$S_0$ to delete any access right from any matrix entry in a row for* which *$S_0$ controls the subject and for any matrix entry in a column for which $S_0$ owns* the object. Rule R4 permits a subject to read that portion of the matrix that it owns or controls.

The remaining rules in Table 4.2 govern the creation and deletion of subjects and objects. Rule R5 states that any subject can create a new object, which it owns, and can then grant and delete access to the object. Under rule R6, the owner of an object can destroy the object, resulting in the deletion of the corresponding column of the access matrix. Rule R7 enables any subject to create a new subject; the creator owns the new subject and the new subject has control access to itself. Rule R8 permits the owner of a subject to delete the row and column (if there are subject columns) of the access matrix designated by that subject.

The set of rules in Table 4.2 is an example of the rule set that could be defined for an access control system. The following are examples of additional or alternative rules that could be included. A transfer-only right could be defined, which results in the transferred right being added to the target subject and deleted from the transferring subject. The number of owners of an object or a subject could limited to one by not allowing the copy flag to accompany the owner right.

The ability of one subject to create another subject and to have ‘owner’ access right to that subject can be used to define a hierarchy of subjects. For example, in Figure 4.4, *$S_1$ owns $S_2$ and $S_3$, so that $S_2$ and $S_3$ are subordinate to $S_1$. By the rules* of Table 4.2, *$S_1$ can grant and delete to $S_2$ access rights that $S_1$ already has. Thus,* a subject can create another subject with a subset of its own access rights. This might be useful, for example, if a subject is invoking an application that is not fully trusted and does not want that application to be able to transfer access rights to other subjects.



## Protection Domains

- set of objects together with access rights to those objects
- more flexibility when associating capabilities with protection domains
- in terms of the access matrix, a row defines a protection domain
- user can spawn processes with a subset of the access rights of the user
- association between a process and a domain can be static or dynamic
- in user mode certain areas of memory are protected from use and certain instructions may not be executed
- in kernel mode privileged instructions may be executed and protected areas of memory may be accessed

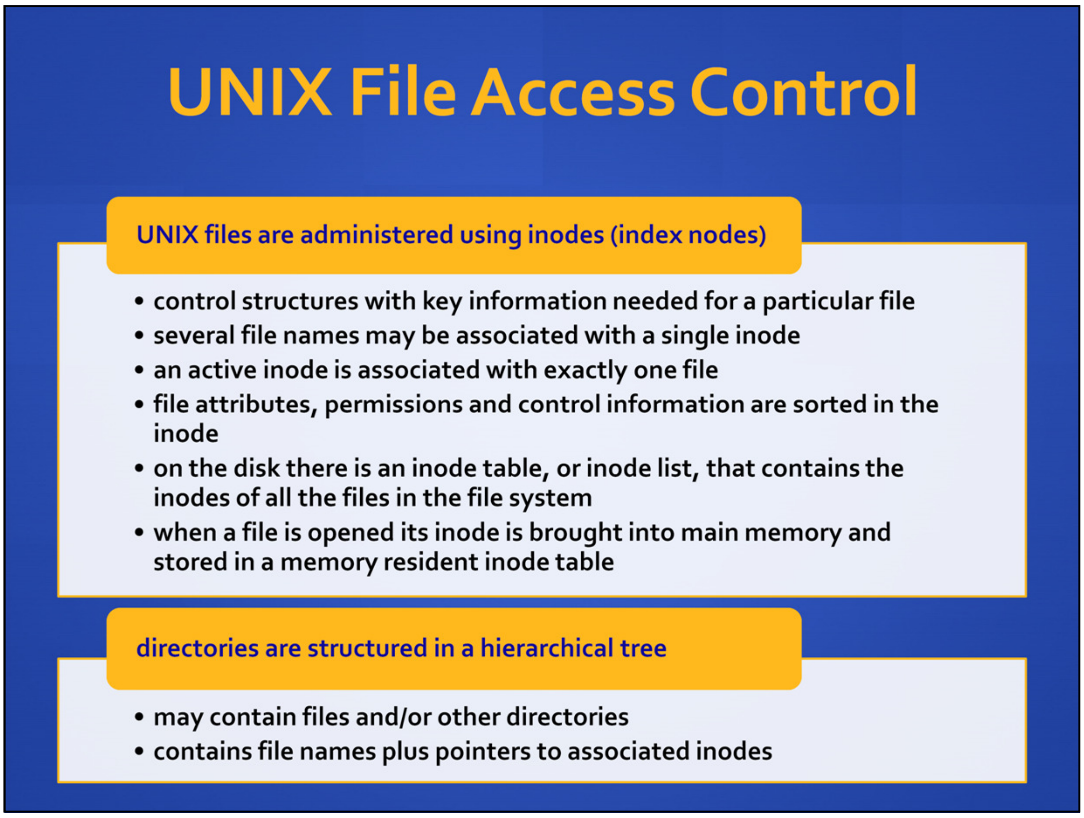
The access control matrix model that we have discussed so far associates a set of
capabilities with a user. A more general and more flexible approach, proposed
in [LAMP71], is to associate capabilities with protection domains. A protection
domain is a set of objects together with access rights to those objects. In terms
of the access matrix, a row defines a protection domain. So far, we have equated
each row with a specific user. So, in this limited model, each user has a protection
domain, and any processes spawned by the user have access rights defined by the
same protection domain.

A more general concept of protection domain provides more flexibility. For
example, a user can spawn processes with a subset of the access rights of the user,
defined as a new protection domain. This limits the capability of the process.
Such a scheme could be used by a server process to spawn processes for different
classes of users. Also, a user could define a protection domain for a program that
is not fully trusted, so that its access is limited to a safe subset of the user's

access
rights.

The association between a process and a domain can be static or dynamic.
For example, a process may execute a sequence of procedures and require different
access rights for each procedure, such as read file and write file. In general,
we would like to minimize the access rights that any user or process has at any
one time; the use of protection domains provides a simple means to satisfy this
requirement.

One form of protection domain has to do with the distinction made in many
operating systems, such as UNIX, between user and kernel mode. A user program
executes in a **user mode, in which certain areas of memory are protected from the**
user's use and in which certain instructions may not be executed. When the user
process calls a system routine, that routine executes in a system mode, or what has
come to be called **kernel mode, in which privileged instructions may be executed**
and in which protected areas of memory may be accessed.



## UNIX File Access Control

**UNIX files are administered using inodes (index nodes)**

- control structures with key information needed for a particular file
- several file names may be associated with a single inode
- an active inode is associated with exactly one file
- file attributes, permissions and control information are sorted in the inode
- on the disk there is an inode table, or inode list, that contains the inodes of all the files in the file system
- when a file is opened its inode is brought into main memory and stored in a memory resident inode table

**directories are structured in a hierarchical tree**

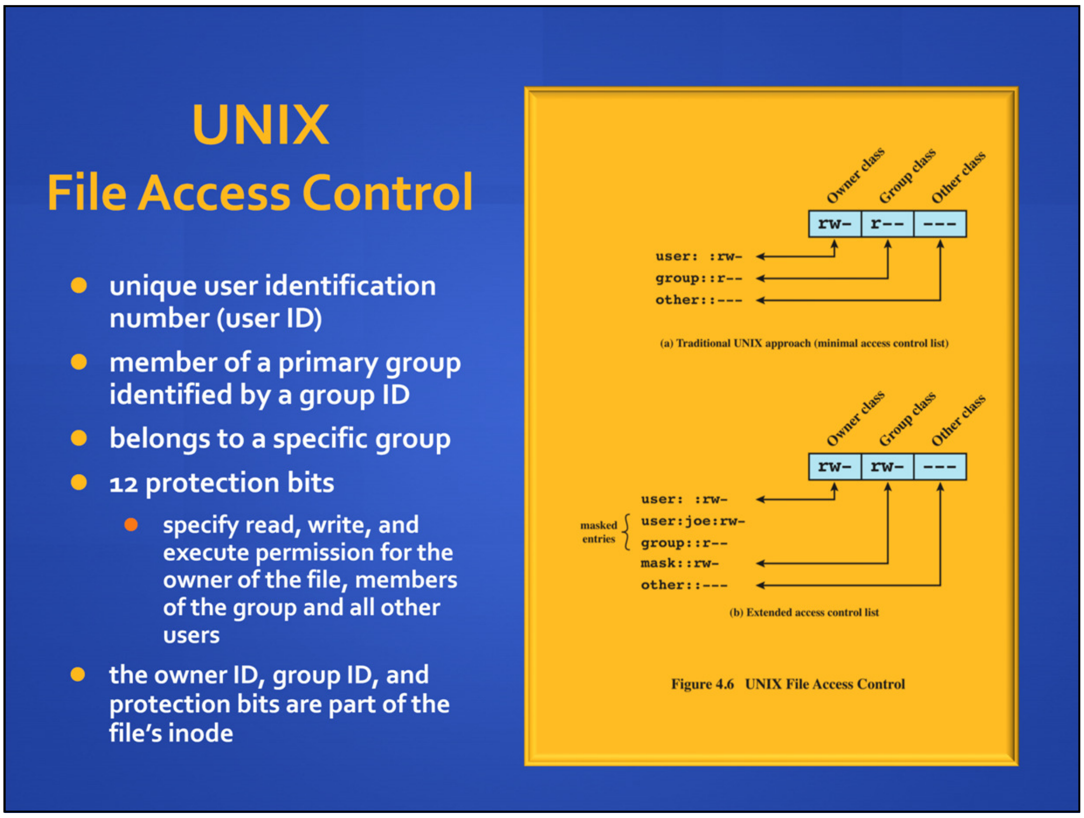
- may contain files and/or other directories
- contains file names plus pointers to associated inodes

For our discussion of UNIX file access control, we first introduce several basic concepts concerning UNIX files and directories.

All types of UNIX files are administered by the operating system by means of inodes. An inode (index node) is a control structure that contains the key information
needed by the operating system for a particular file. Several file names may be associated with a single inode, but an active inode is associated with exactly one file,
and each file is controlled by exactly one inode. The attributes of the file as well as
its permissions and other control information are stored in the inode. On the disk, there is an inode table, or inode list, that contains the inodes of all the files in the file
system. When a file is opened, its inode is brought into main memory and stored in
a memory-resident inode table.

Directories are structured in a hierarchical tree. Each directory can contain files and/or other directories. A directory that is inside another directory is referred to as a subdirectory. A directory is simply a file that contains a list of file names plus

pointers to associated inodes. Thus, associated with each directory is its own inode.



## UNIX File Access Control

- unique user identification number (user ID)
- member of a primary group identified by a group ID
- belongs to a specific group
- 12 protection bits
  - specify read, write, and execute permission for the owner of the file, members of the group and all other users
- the owner ID, group ID, and protection bits are part of the file's inode

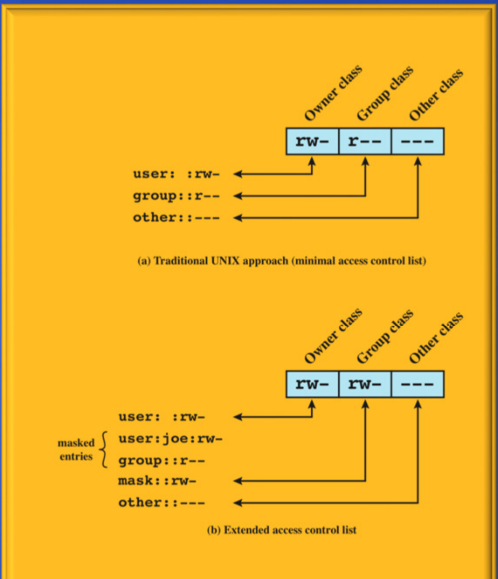


Figure 4.6 UNIX File Access Control

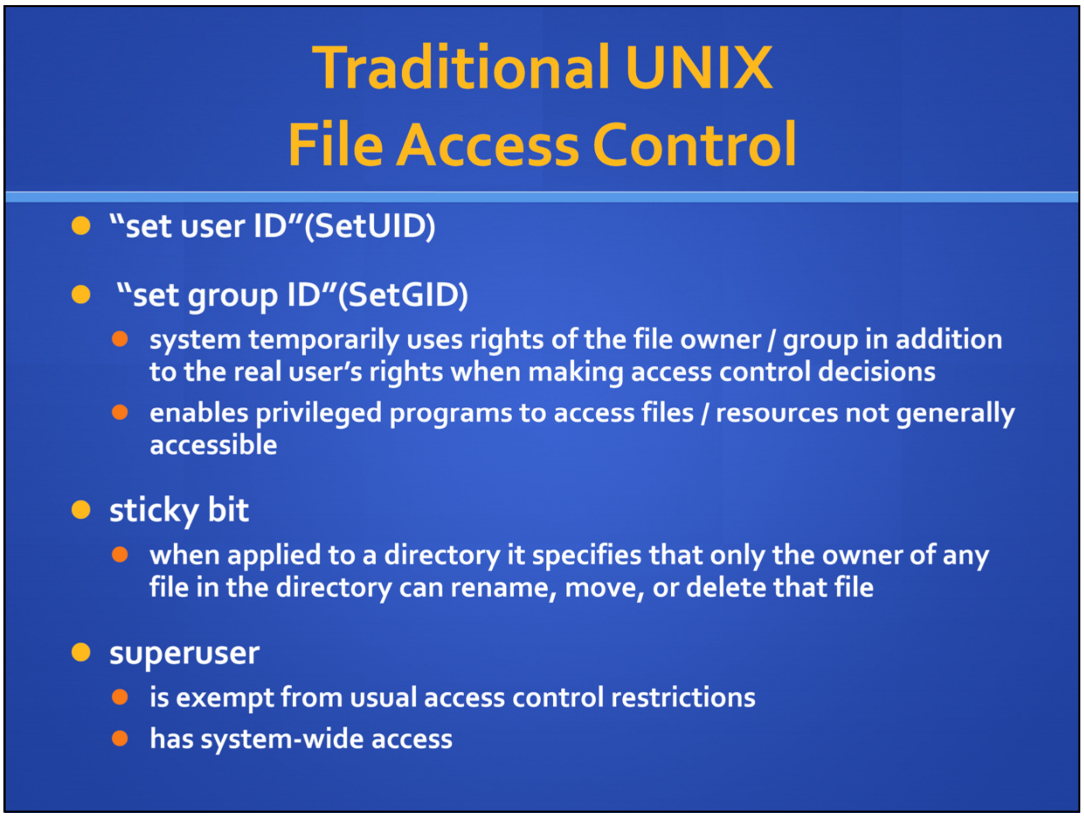
Most UNIX systems depend on, or at least are based on, the file access control scheme introduced with the early versions of UNIX. Each UNIX user is assigned a unique user identification number (user ID). A user is also a member of a primary group, and possibly a number of other groups, each identified by a group ID. When a file is created, it is designated as owned by a particular user and marked with that user's ID. It also belongs to a specific group, which initially is either its creator's primary group, or the group of its parent directory if that directory has SetGID permission set. Associated with each file is a set of 12 protection bits. The owner ID, group ID, and protection bits are part of the file's inode.

Nine of the protection bits specify read, write, and execute permission for the owner of the file, other members of the group to which this file belongs, and all other users. These form a hierarchy of owner, group, and all others, with the highest relevant set of permissions being used. Figure 4.6a shows an example in which the file owner has read and write access; all other members of the file's group have read access, and users

outside the group have no access rights to the file. When applied to a directory, the read

and write bits grant the right to list and to create/rename/delete files in the directory.

The execute bit grants to right to descend into the directory or search it for a filename.



## Traditional UNIX File Access Control

- "set user ID"(SetUID)
- "set group ID"(SetGID)
  - system temporarily uses rights of the file owner / group in addition to the real user's rights when making access control decisions
  - enables privileged programs to access files / resources not generally accessible
- sticky bit
  - when applied to a directory it specifies that only the owner of any file in the directory can rename, move, or delete that file
- superuser
  - is exempt from usual access control restrictions
  - has system-wide access

The remaining three bits define special additional behavior for files or directories.

Two of these are the "set user ID" (SetUID) and "set group ID" (SetGID)

permissions. If these are set on an executable file, the operating system functions as

follows. When a user (with execute privileges for this file) executes the file, the system

temporarily allocates the rights of the user's ID of the file creator, or the file's group,

respectively, to those of the user executing the file. These are known as the "effective

user ID" and "effective group ID" and are used in addition to the "real user ID" and

"real group ID" of the executing user when making access control decisions for this

program. This change is only effective while the program is being executed. This feature

enables the creation and use of privileged programs that may use files normally

inaccessible to other users. It enables users to access certain files in a controlled fashion.

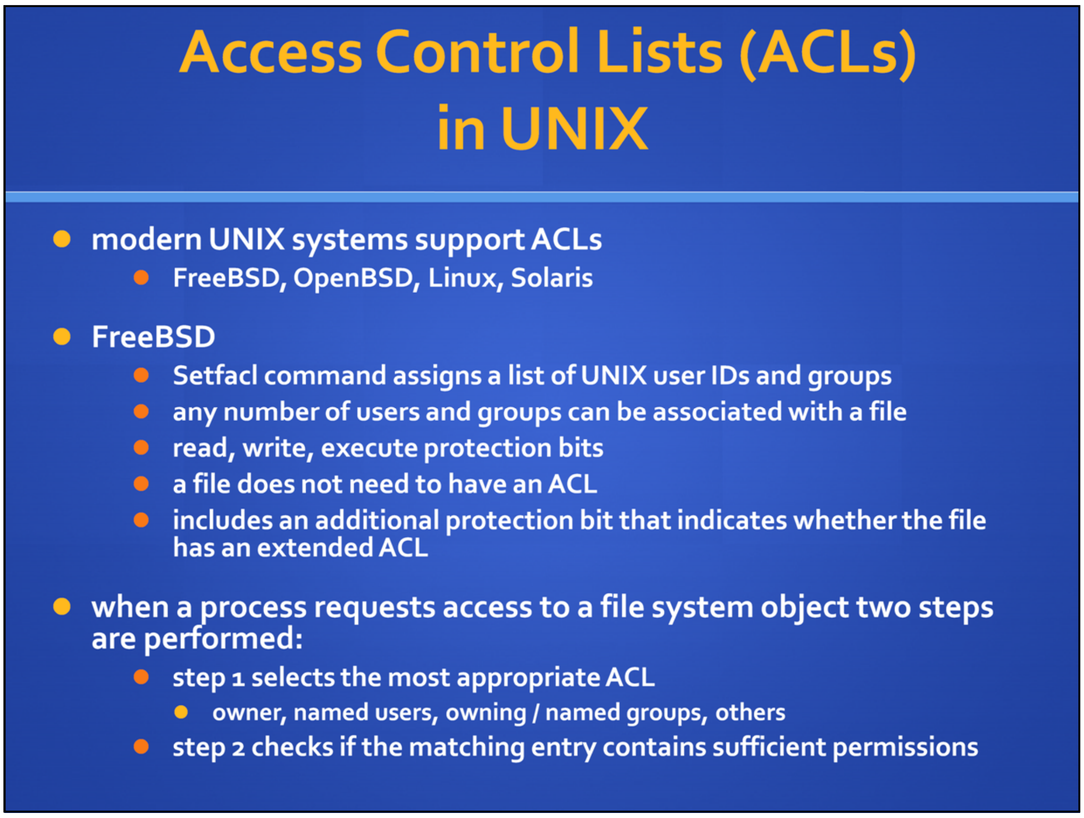
Alternatively, when applied to a directory, the SetGID permission indicates that newly created files will inherit the group of this directory. The SetUID permission is ignored.

The final permission bit is the “Sticky” bit. When set on a file, this originally indicated that the system should retain the file contents in memory following execution.
This is no longer used. When applied to a directory, though, it specifies that only the owner of any file in the directory can rename, move, or delete that file. This
is useful for managing files in shared temporary directories.

One particular user ID is designated as “superuser.” The superuser is exempt from the usual file access control constraints and has systemwide access. Any program that is owned by, and SetUID to, the “superuser” potentially grants unrestricted access to the system to any user executing that program. Hence great
care is needed when writing such programs.

This access scheme is adequate when file access requirements align with users and a modest number of groups of users. For example, suppose a user wants to give
read access for file X to users A and B and read access for file Y to users B and C. We
would need at least two user groups, and user B would need to belong to both groups
in order to access the two files. However, if there are a large number of different groupings of users requiring a range of access rights to different files, then a very large
number of groups may be needed to provide this. This rapidly becomes unwieldy and
difficult to manage, even if possible at all. One way to overcome this problem is to use
access control lists, which are provided in most modern UNIX systems.

A final point to note is that the traditional UNIX file access control scheme implements a simple protection domain structure. A domain is associated with the user, and switching the domain corresponds to changing the user ID temporarily.



## Access Control Lists (ACLs) in UNIX

- modern UNIX systems support ACLs
  - FreeBSD, OpenBSD, Linux, Solaris
- FreeBSD
  - Setfacl command assigns a list of UNIX user IDs and groups
  - any number of users and groups can be associated with a file
  - read, write, execute protection bits
  - a file does not need to have an ACL
  - includes an additional protection bit that indicates whether the file has an extended ACL
- when a process requests access to a file system object two steps are performed:
  - step 1 selects the most appropriate ACL
    - owner, named users, owning / named groups, others
  - step 2 checks if the matching entry contains sufficient permissions

Many modern UNIX and UNIX-based operating systems support access control lists, including FreeBSD, OpenBSD, Linux, and Solaris. In this section, we describe
FreeBSD, but other implementations have essentially the same features and interface.
The feature is referred to as extended access control list, while the traditional UNIX approach is referred to as minimal access control list.

FreeBSD allows the administrator to assign a list of UNIX user IDs and groups to a file by using the setfacl command. Any number of users and groups can be associated with a file, each with three protection bits (read, write, execute), offering a
flexible mechanism for assigning access rights. A file need not have an ACL but may be
protected solely by the traditional UNIX file access mechanism. Free BSD files include
an additional protection bit that indicates whether the file has an extended ACL.

FreeBSD and most UNIX implementations that support extended ACLs use the following strategy (e.g., Figure 4.6b):

**1. The owner class and other class entries in the 9-bit permission field have the**

same meaning as in the minimal ACL case.

**2. The group class entry specifies the permissions for the owner group for this file.**

These permissions represent the maximum permissions that can be assigned to named users or named groups, other than the owning user. In this latter role, the group class entry functions as a mask.

**3. Additional named users and named groups may be associated with the file,**

each with a 3-bit permission field. The permissions listed for a named user or named group are compared to the mask field. Any permission for the named user or named group that is not present in the mask field is disallowed.

When a process requests access to a file system object, two steps are per formed.

Step 1 selects the ACL entry that most closely matches the requesting process. The ACL

entries are looked at in the following order: owner, named users, (owning or named)

groups, others. Only a single entry determines access. Step 2 checks if the matching entry

contains sufficient permissions. A process can be a member in more than one group; so

more than one group entry can match. If any of these matching group entries contain the

requested permissions, one that contains the requested permissions is picked (the result

is the same no matter which entry is picked). If none of the matching group entries contains

the requested permissions, access will be denied no matter which entry is picked.

## Figure 4.7

## Role-Based Access Control (RBAC)

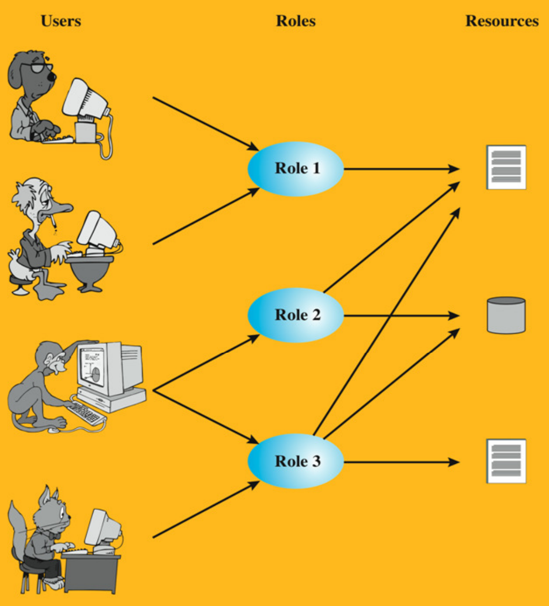


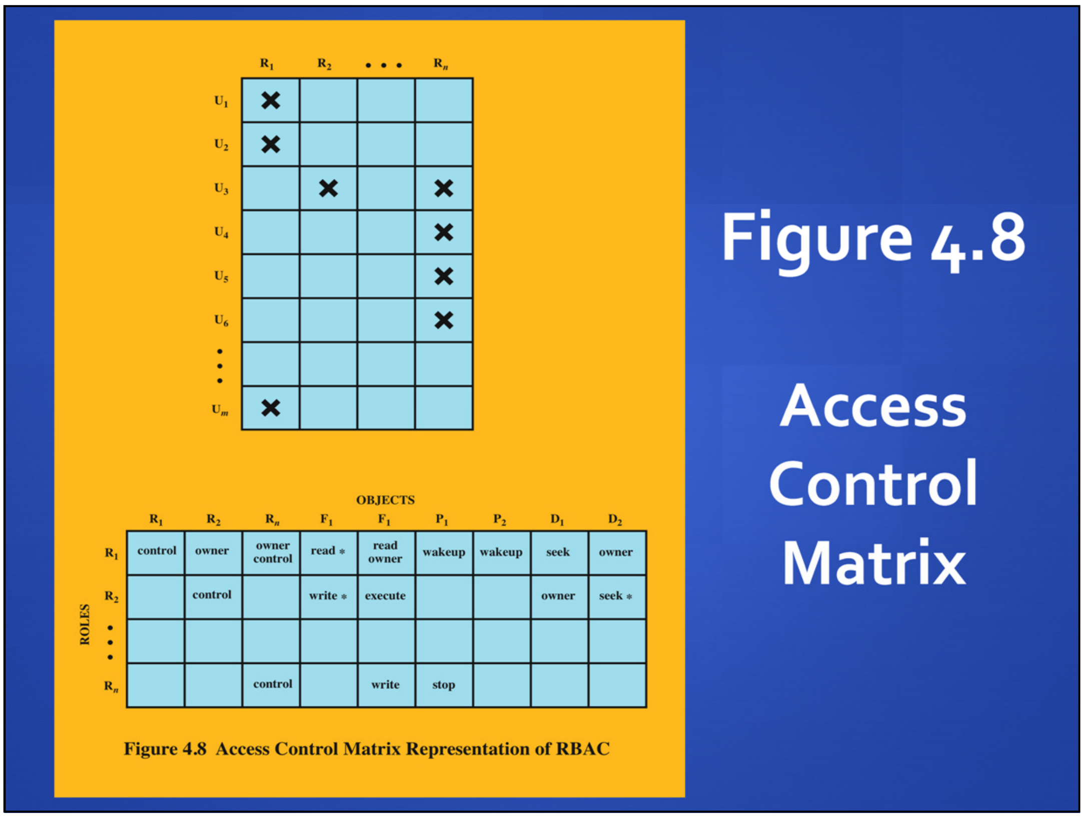
Figure 4.7 Users, Roles, and Resources

Traditional DAC systems define the access rights of individual users and groups of users. In contrast, RBAC is based on the roles that users assume in a system rather than the user's identity. Typically, RBAC models define a role as a job function
within an organization. RBAC systems assign access rights to roles instead of individual users. In turn, users are assigned to different roles, either statically or dynamically, according to their responsibilities.

RBAC now enjoys widespread commercial use and remains an area of active research. The National Institute of Standards and Technology (NIST) has issued a
standard, *Security Requirements for Cryptographic Modules (FIPS PUB 140-2, May*
25, 2001), that requires support for access control and administration through roles.

The relationship of users to roles is many to many, as is the relationship of roles to resources, or system objects (Figure 4.7). The set of users changes, in some
environments frequently, and the assignment of a user to one or more roles may

also be dynamic. The set of roles in the system in most environments is relatively
static, with only occasional additions or deletions. Each role will have specific
access
rights to one or more resources. The set of resources and the specific access
rights
associated with a particular role are also likely to change infrequently.



## Figure 4.8

## Access Control Matrix

| | $R_1$ | $R_2$ | • • • | $R_n$ |
|---|---|---|---|---|
| $U_1$ | ✖ | | | |
| $U_2$ | ✖ | | | |
| $U_3$ | | ✖ | | ✖ |
| $U_4$ | | | | ✖ |
| $U_5$ | | | | ✖ |
| $U_6$ | | | | ✖ |
| • • • | | | | |
| $U_m$ | ✖ | | | |

OBJECTS

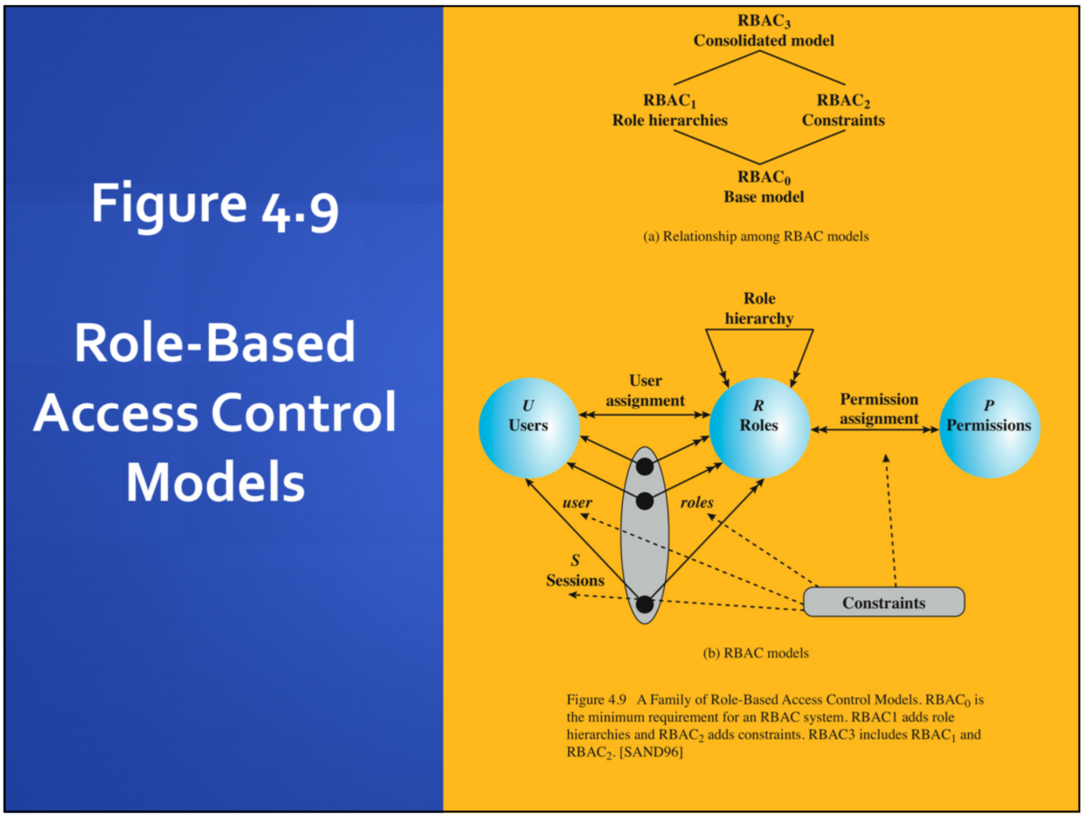
| ROLES | $R_1$ | $R_2$ | $R_n$ | $F_1$ | $F_1$ | $P_1$ | $P_2$ | $D_1$ | $D_2$ |
|---|---|---|---|---|---|---|---|---|---|
| $R_1$ | control | owner | owner control | read * | read owner | wakeup | wakeup | seek | owner |
| $R_2$ | | control | | write * | execute | | | owner | seek * |
| • • • | | | | | | | | | |
| $R_n$ | | | control | | write | stop | | | |

**Figure 4.8 Access Control Matrix Representation of RBAC**

We can use the access matrix representation to depict the key elements of an RBAC system in simple terms, as shown in Figure 4.8. The upper matrix relates individual users to roles. Typically there are many more users than roles. Each matrix
entry is either blank or marked, the latter indicating that this user is assigned to this
role. Note that a single user may be assigned multiple roles (more than one mark in a
row) and that multiple users may be assigned to a single role (more than one mark in
a column). The lower matrix has the same structure as the DAC access control matrix,
with roles as subjects. Typically, there are few roles and many objects, or resources.
In this matrix the entries are the specific access rights enjoyed by the roles. Note that a
role can be treated as an object, allowing the definition of role hierarchies.

RBAC lends itself to an effective implementation of the principle of least privilege, referred to in Section 4.1. Each role should contain the minimum set of

access rights needed for that role. A user is assigned to a role that enables him or her
to perform only what is required for that role. Multiple users assigned to the same
role, enjoy the same minimal set of access rights.



## Figure 4.9

## Role-Based Access Control Models

Figure 4.9 A Family of Role-Based Access Control Models. RBAC$_0$ is the minimum requirement for an RBAC system. RBAC1 adds role hierarchies and RBAC$_2$ adds constraints. RBAC3 includes RBAC$_1$ and RBAC$_2$. [SAND96]

A variety of functions and services can be included under the general RBAC approach. To clarify the various aspects of RBAC, it is useful to define a set of abstract models of RBAC functionality.

The solid lines in Figure 4.9b indicate relationships, or mappings, with a single arrowhead indicating one and a double arrowhead indicating many. Thus, there is a many-to-many relationship between users and roles: One user may have multiple
roles, and multiple users may be assigned to a single role. Similarly, there is a many-to-
many relationship between roles and permissions. A session is used to define a temporary one-to-many relationship between a user and one or more of the roles to
which the user has been assigned. The user establishes a session with only the roles
needed for a particular task; this is an example of the concept of least privilege.

The many-to-many relationships between users and roles and between roles and permissions provide a flexibility and granularity of assignment not found in conventional DAC schemes. Without this flexibility and granularity, there is a

greater
risk that a user may be granted more access to resources than is needed because of
the limited control over the types of access that can be allowed. The NIST RBAC
document gives the following examples: Users may need to list directories and modify
existing files without creating new files, or they may need to append records to a file
without modifying existing records.

[SAND96] defines a family of reference models that has served as the basis
for ongoing standardization efforts. This family consists of four models that are
related to each other as shown in Figure 4.9a. and Table 4.3. $RBAC_0$ contains the
minimum functionality for an RBAC system. $RBAC_1$ includes the $RBAC_0$ functionality
and adds role hierarchies, which enable one role to inherit permissions
from another role. $RBAC_2$ includes $RBAC_0$ and adds constraints, which restrict
the ways in which the components of a RBAC system may be configured. $RBAC_3$
contains the functionality of $RBAC_0$, $RBAC_1$, and $RBAC_2$.

the ways in which the components of a RBAC system may be configured. $RBAC_3$
contains the functionality of $RBAC_0$, $RBAC_1$, and $RBAC_2$.

**Base Model—$RBAC_0$ Figure 4.9b, without the role hierarchy and constraints,**
contains the four types of entities in an $RBAC_0$ system:

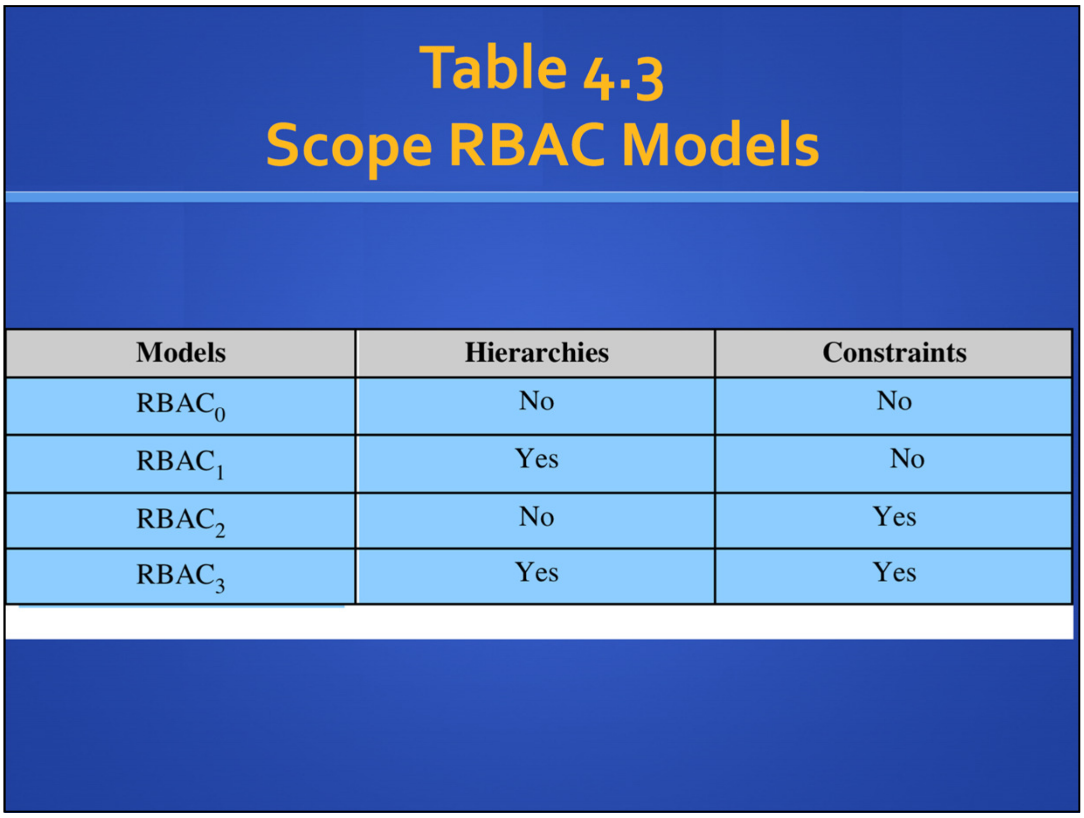
• **User: An individual that has access to this computer system. Each individual**
has an associated user ID.

**Role: A named job function within the organization that controls this computer**
system. Typically, associated with each role is a description of the authority and
responsibility conferred on this role, and on any user who assumes this role.

• **Permission: An approval of a particular mode of access to one or more objects.**
Equivalent terms are *access right, privilege, and authorization.*

• **Session: A mapping between a user and an activated subset of the set of roles**

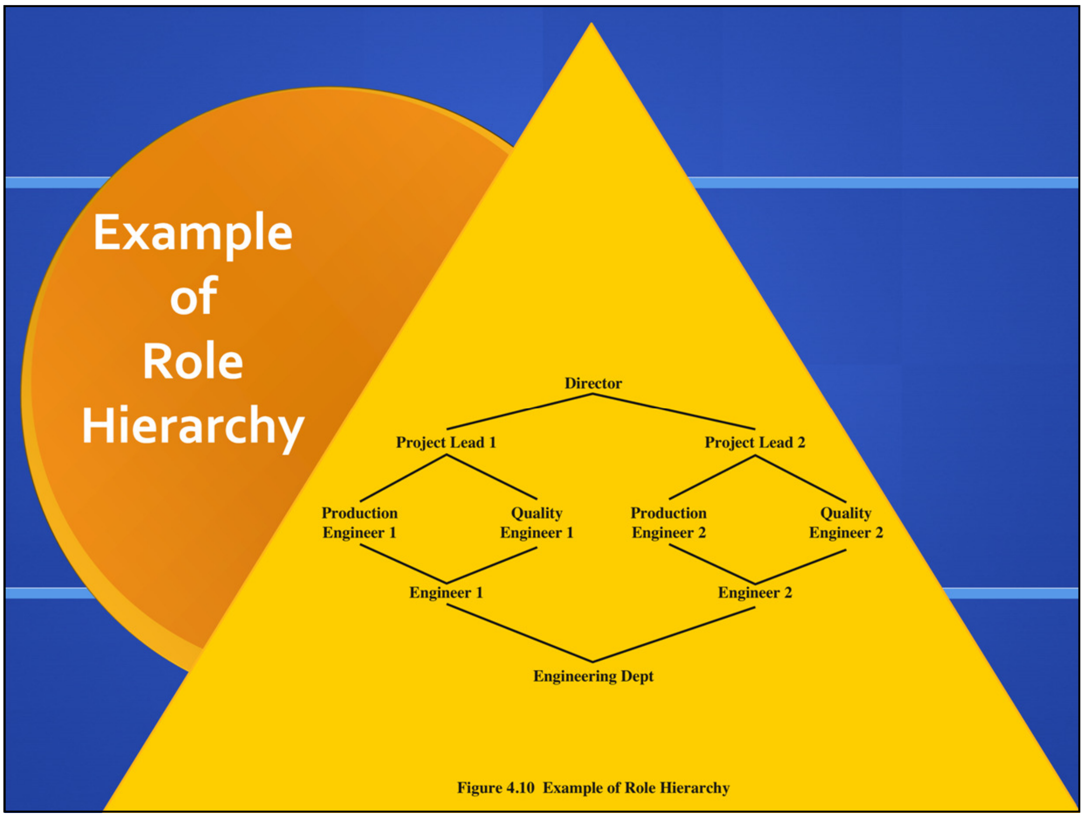
to which the user is assigned.



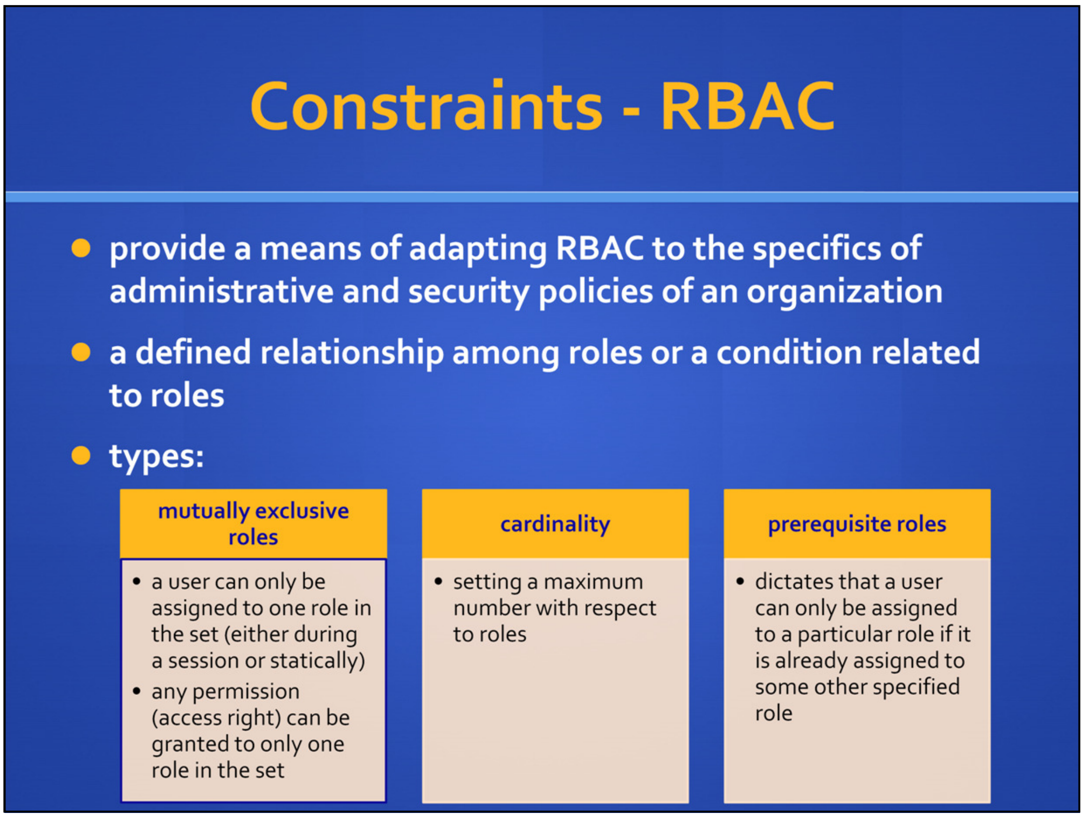
## Table 4.3
## Scope RBAC Models

| Models | Hierarchies | Constraints |
|---|---|---|
| $RBAC_0$ | No | No |
| $RBAC_1$ | Yes | No |
| $RBAC_2$ | No | Yes |
| $RBAC_3$ | Yes | Yes |

Scope RBAC Models.



Role hierarchies provide a means of reflecting
the hierarchical structure of roles in an organization. Typically, job functions with greater responsibility have greater authority to access resources. A subordinate job function may have a subset of the access rights of the superior job function. Role hierarchies make use of the concept of inheritance to enable one role to implicitly include access rights associated with a subordinate role.
Figure 4.10 is an example of a diagram of a role hierarchy. By convention, subordinate roles are lower in the diagram. A line between two roles implies that the
upper role includes all of the access rights of the lower role, as well as other access rights not available to the lower role. One role can inherit access rights from multiple subordinate roles. For example, in Figure 4.10, the Project Lead role includes all of the access rights of the Production Engineer role and of the Quality Engineer role. More than one role can inherit from the same subordinate role. For example, both the Production Engineer role and the Quality Engineer role include all of the access rights of the Engineer role. Additional access rights are also assigned to the Production Engineer Role and a different set of additional access rights are assigned to the Quality Engineer role. Thus, these two roles have overlapping access rights, namely the access rights they share with the Engineer role.



## Constraints - RBAC

- provide a means of adapting RBAC to the specifics of administrative and security policies of an organization
- a defined relationship among roles or a condition related to roles
- types:

| mutually exclusive roles | cardinality | prerequisite roles |
|---|---|---|
| • a user can only be assigned to one role in the set (either during a session or statically)<br>• any permission (access right) can be granted to only one role in the set | • setting a maximum number with respect to roles | • dictates that a user can only be assigned to a particular role if it is already assigned to some other specified role |

Constraints provide a means of adapting RBAC to the
specifics of administrative and security policies in an organization. A constraint is
a defined relationship among roles or a condition related to roles. [SAND96] lists
the following types of constraints: mutually exclusive roles, cardinality, and prerequisite
roles.

**Mutually exclusive roles are roles such that a user can be assigned to only**
one role in the set. This limitation could be a static one, or it could be dynamic, in
the sense that a user could be assigned only one of the roles in the set for a session.
The mutually exclusive constraint supports a separation of duties and capabilities
within an organization. This separation can be reinforced or enhanced by use of
mutually exclusive permission assignments. With this additional constraint, a mutually
exclusive set of roles has the following properties:

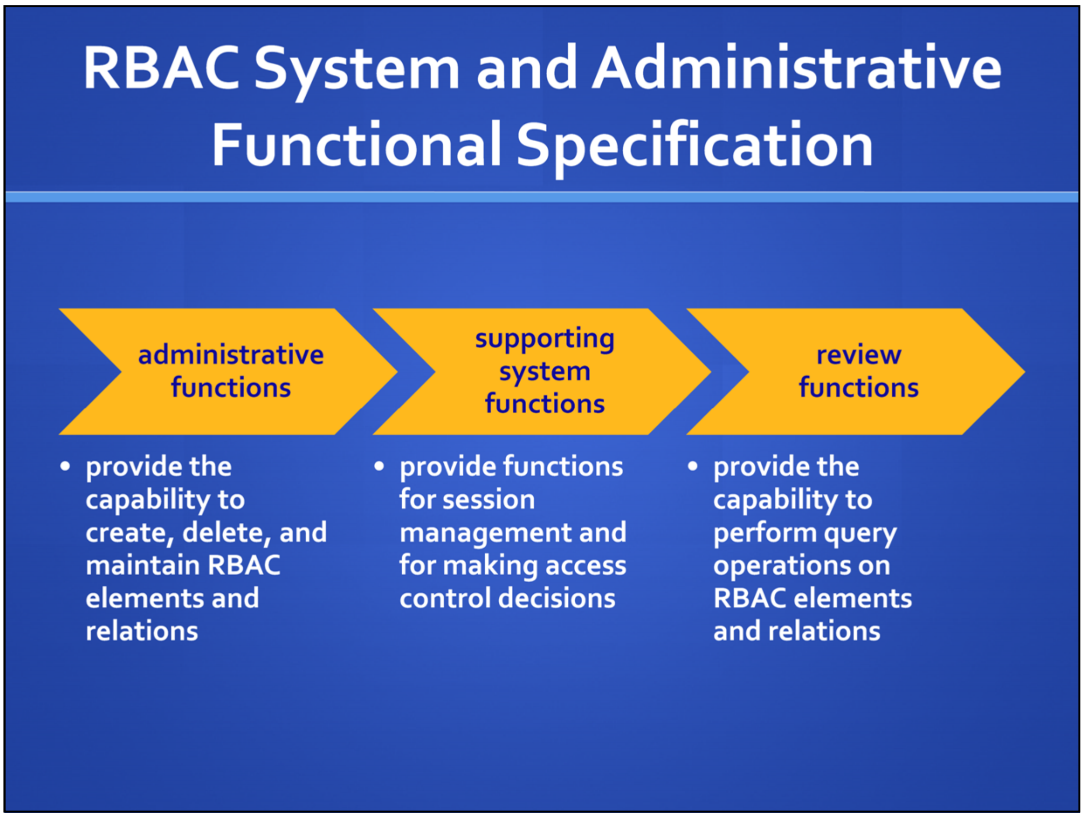
**1. A user can only be assigned to one role in the set (either during a session or**
statically).

**2. Any permission (access right) can be granted to only one role in the set.**

Thus the set of mutually exclusive roles have non-overlapping permissions. If two
users are assigned to different roles in the set, then the users have non-overlapping
permissions while assuming those roles. The purpose of mutually exclusive roles is to
increase the difficulty of collusion among individuals of different skills or divergent job
functions to thwart security policies.

**Cardinality refers to setting a maximum number with respect to roles. One** such constraint is to set a maximum number of users that can be assigned to a given
role. For example, a project leader role or a department head role might be limited to a single user. The system could also impose a constraint on the number of roles that a user is assigned to, or the number of roles a user can activate for a single session.
Another form of constraint is to set a maximum number of roles that can be granted a particular permission; this might be a desirable risk mitigation technique for a sensitive or powerful permission.

A system might be able to specify a **prerequisite, which dictates that a user can**
only be assigned to a particular role if it is already assigned to some other specified
role. A prerequisite can be used to structure the implementation of the least privilege
concept. In a hierarchy, it might be required that a user can be assigned to a senior
(higher) role only if it is already assigned an immediately junior (lower) role. For example, in Figure 4.10 a user assigned to a Project Lead role must also be assigned
to the subordinate Production Engineer and Quality Engineer roles. Then, if the user
does not need all of the permissions of the Project Lead role for a given task, the user
can invoke a session using only the required subordinate role. Note that the use of prerequisites tied to the concept of hierarchy requires the $RBAC_3$ model.



## RBAC System and Administrative Functional Specification

administrative functions
supporting system functions
review functions

- provide the capability to create, delete, and maintain RBAC elements and relations
- provide functions for session management and for making access control decisions
- provide the capability to perform query operations on RBAC elements and relations

In 2001, NIST proposed a consensus model for RBAC, based on the original work in [SAND96] and later contributions. The model was further refined within the RBAC community and has been adopted by the American National Standards Institute, International Committee for Information Technology Standards (ANSI/INCITS) as ANSI INCITS 359–2004.

The main innovation of the NIST standard is the introduction of the *RBAC System and Administrative Functional Specification, which defines the features* required for an RBAC system. This specification has a number of benefits. The specification provides a functional benchmark for vendors, indicating which capabilities must be provided to the user and the general programming interface for those functions. The specification guides users in developing requirements documents and in evaluating vendor products in a uniform fashion. The specification also provides a baseline system on which researchers and implementers can build enhanced features. The specification defines features, or functions, in three categories:

• **Administrative functions: Provide the capability to create, delete, and maintain** RBAC elements and relations

• **Supporting system functions: Provide functions for session management and** for making access control decisions

• **Review functions: Provide the capability to perform query operations on** RBAC elements and relations

Examples of these functions are presented in the following discussion.

## Figure 4.11
## NIST RBAC Model

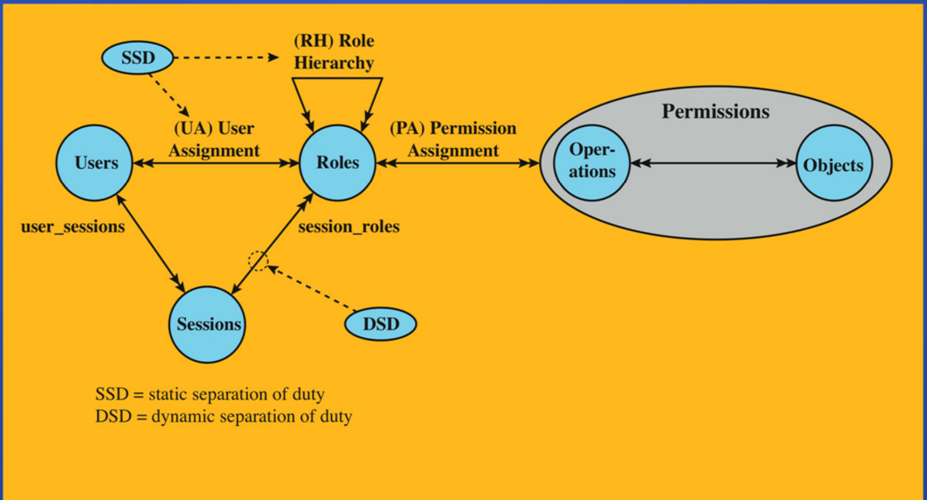
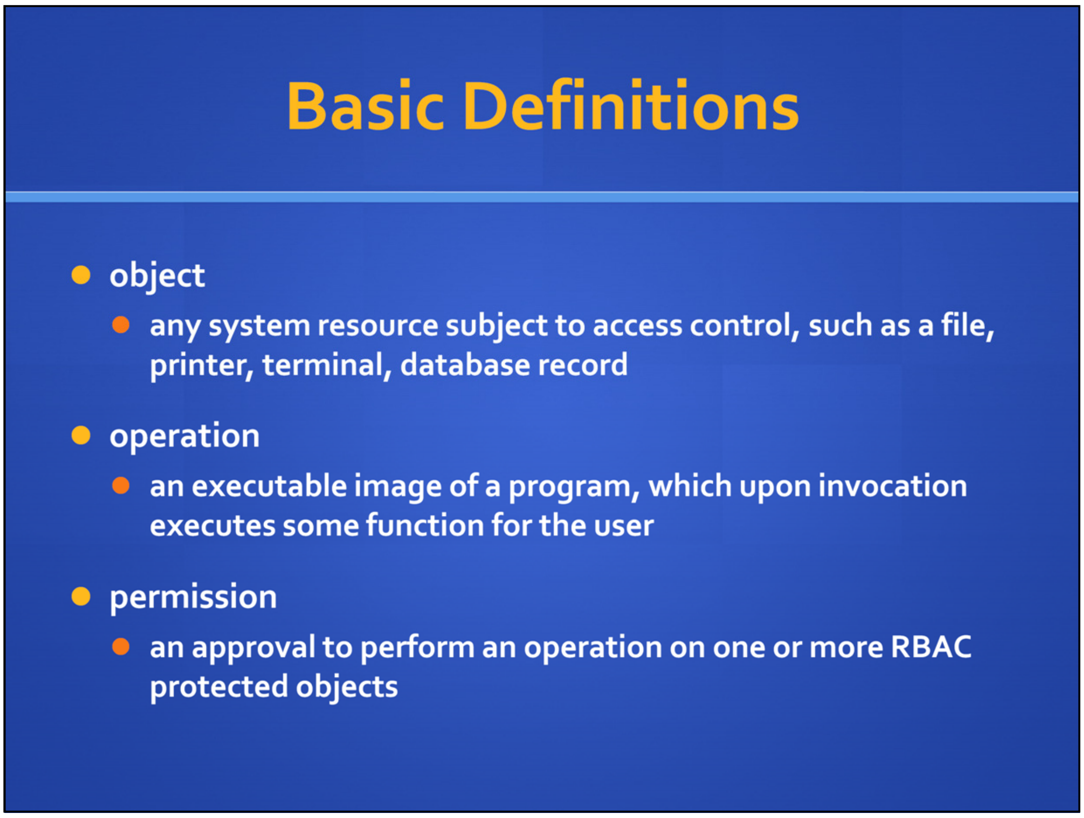


**Figure 4.11 NIST RBAC Model**

The NIST RBAC model comprises four model components (Figure 4.11): core RBAC, hierarchical RBAC, static separation of duty (SSD) relations, and dynamic separation of duty (DSD) relations. The last two components correspond to the constraints component of the model of Figure 4.9.



## Basic Definitions

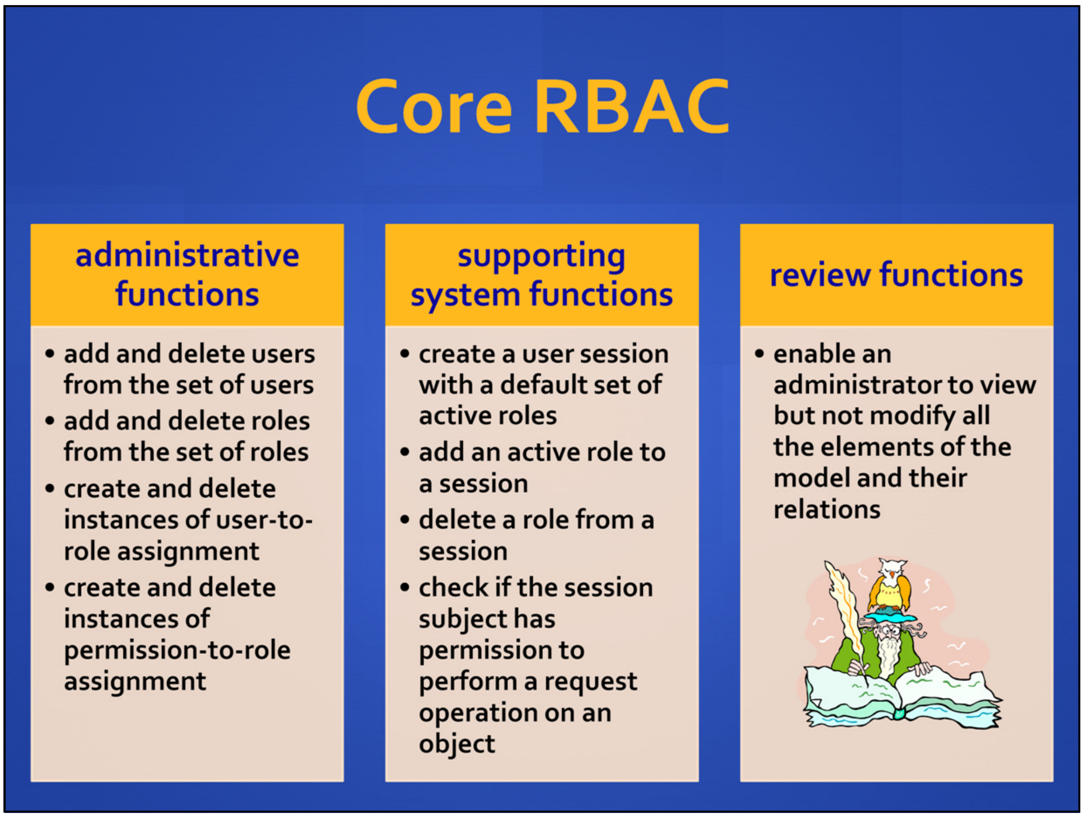
- object
  - any system resource subject to access control, such as a file, printer, terminal, database record
- operation
  - an executable image of a program, which upon invocation executes some function for the user
- permission
  - an approval to perform an operation on one or more RBAC protected objects

The elements of core RBAC are the same as those of RBAC$_0$
described in the preceding section: users, roles, permissions, and sessions. The NIST
model elaborates on the concept of permissions by introducing two subordinate
entities: operations and objects. The following definitions are relevant:

• **Object: Any system resource subject to access control, such as a file, printer,**
terminal, database record, and so on

• **Operation: An executable image of a program, which upon invocation**
executes some function for the user

• **Permission: An approval to perform an operation on one or more RBAC**
protected objects

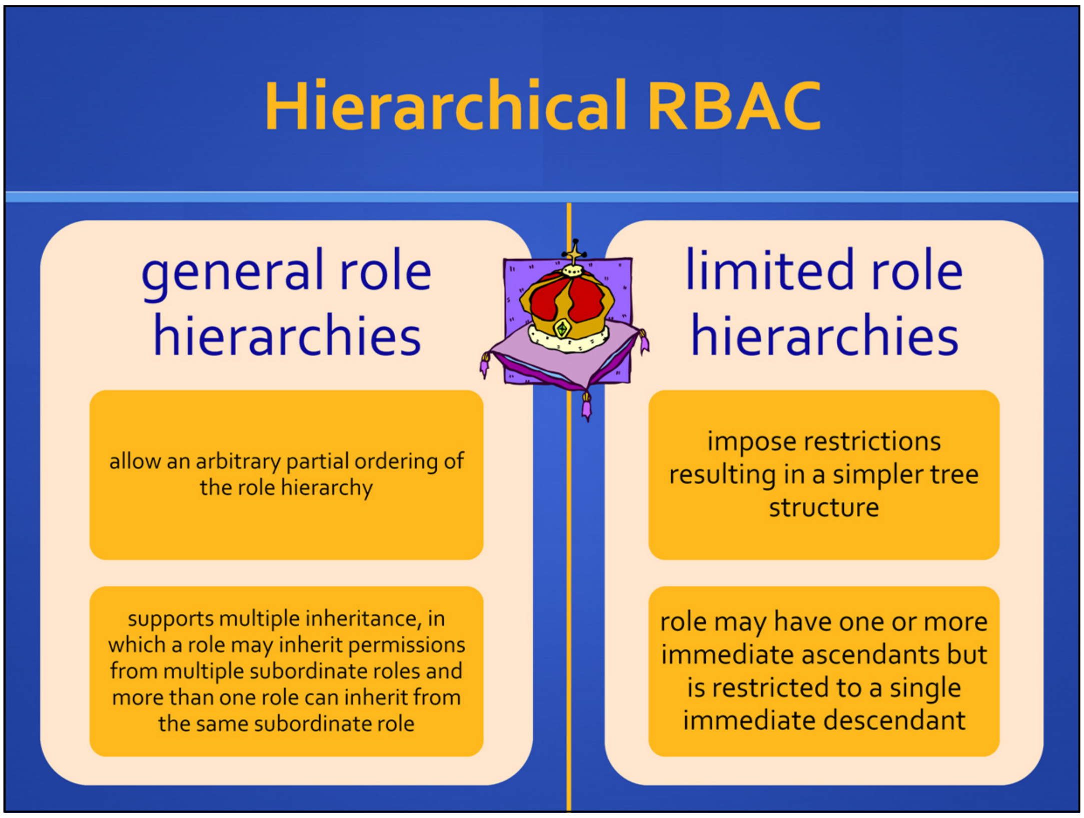


## Core RBAC

| administrative functions | supporting system functions | review functions |
|---|---|---|
| • add and delete users from the set of users<br>• add and delete roles from the set of roles<br>• create and delete instances of user-to-role assignment<br>• create and delete instances of permission-to-role assignment | • create a user session with a default set of active roles<br>• add an active role to a session<br>• delete a role from a session<br>• check if the session subject has permission to perform a request operation on an object | • enable an administrator to view but not modify all the elements of the model and their relations<br> |

The **administrative functions for Core RBAC include the following: add and** delete users from the set of users; add and delete roles from the set of roles; create and delete instances of user-to-role assignment; and create and delete instances of permission-to-role assignment. The **supporting system functions include the following:** create a user session with a default set of active roles; add an active role to a session; delete a role from a session; and check if the session subject has permission to perform a request operation on an object. The **review functions enable an administrator to view** but not modify all the elements of the model and their relations, including users, roles, user assignments, role assignments, and session elements.

Core RBAC is a minimal model that captures the common features found in the current generation of RBAC systems.



## Hierarchical RBAC

**general role hierarchies**

allow an arbitrary partial ordering of the role hierarchy

supports multiple inheritance, in which a role may inherit permissions from multiple subordinate roles and more than one role can inherit from the same subordinate role

**limited role hierarchies**

impose restrictions resulting in a simpler tree structure

role may have one or more immediate ascendants but is restricted to a single immediate descendant

Hierarchical RBAC includes the concept of inheritance
described for $RBAC_1$. In the NIST standard, the inheritance relationship includes
two aspects. Role $r_1$ *is said to be a descendant of* $r_2$ *if* $r_1$ *includes (inherits) all of the*
permissions from $r_2$ *and all users assigned to* $r_1$ *are also assigned to* $r_2$*. For example,*
in Figure 4.10, any permission allowed in the Project Lead 1 role is also allowed in the
Director role, and a user assigned to the Director role is also assigned to the Project
Lead 1 role.

The NIST model defines two types of role hierarchies:

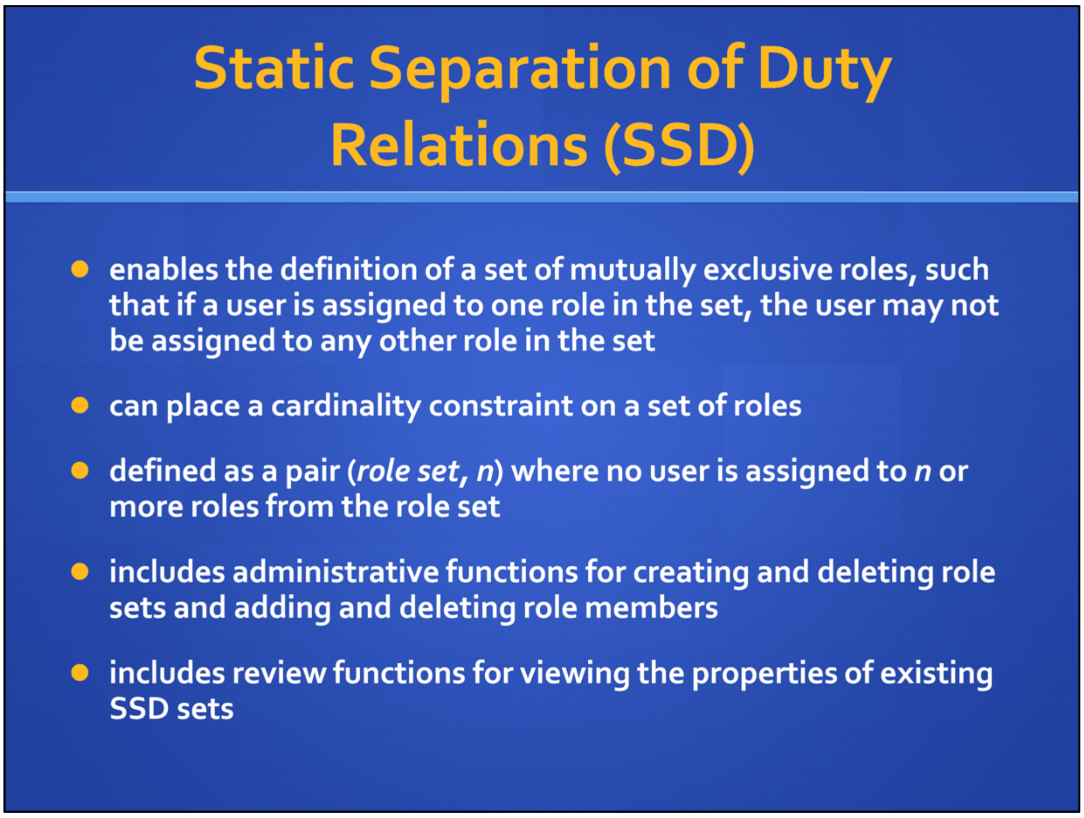
**General role hierarchies: Allow an arbitrary partial ordering of the role**
hierarchy. In particular, this type supports multiple inheritance, in which a
role may inherit permissions from multiple subordinate roles and more than
one role can inherit from the same subordinate role.

• **Limited role hierarchies: Impose restrictions resulting in a simpler tree structure.**
The limitation is that a role may have one or more immediate ascendants
but is restricted to a single immediate descendant.

The rationale for role hierarchies is that the inheritance property greatly simplifies
the task of defining permission relationships. Roles can have overlapping permissions,
which means that users belonging to different roles may have some shared permissions.
In addition, it is typical in an organization that there are many users that share
a set of common permissions, cutting across many organizational levels. To avoid the

necessity of defining numerous roles from scratch to accommodate various users,
role hierarchies are used in a number of commercial implementations. General role
hierarchies provide the most powerful tool for this purpose. The standard incorporates
limited role hierarchies, which are also useful, to allow for a simpler implementation
of role hierarchies.

Hierarchical RBAC adds four new administrative functions to Core RBAC:
add a new immediate inheritance relationship between two existing roles; delete
an existing immediate inheritance relationship; create a new role and add it as
an immediate ascendant of an existing role; and create a new role and add it as
an immediate descendant of an existing relationship. The hierarchical RBAC review
functions enable the administrator to view the permissions and users associated with
each role either directly or by inheritance.



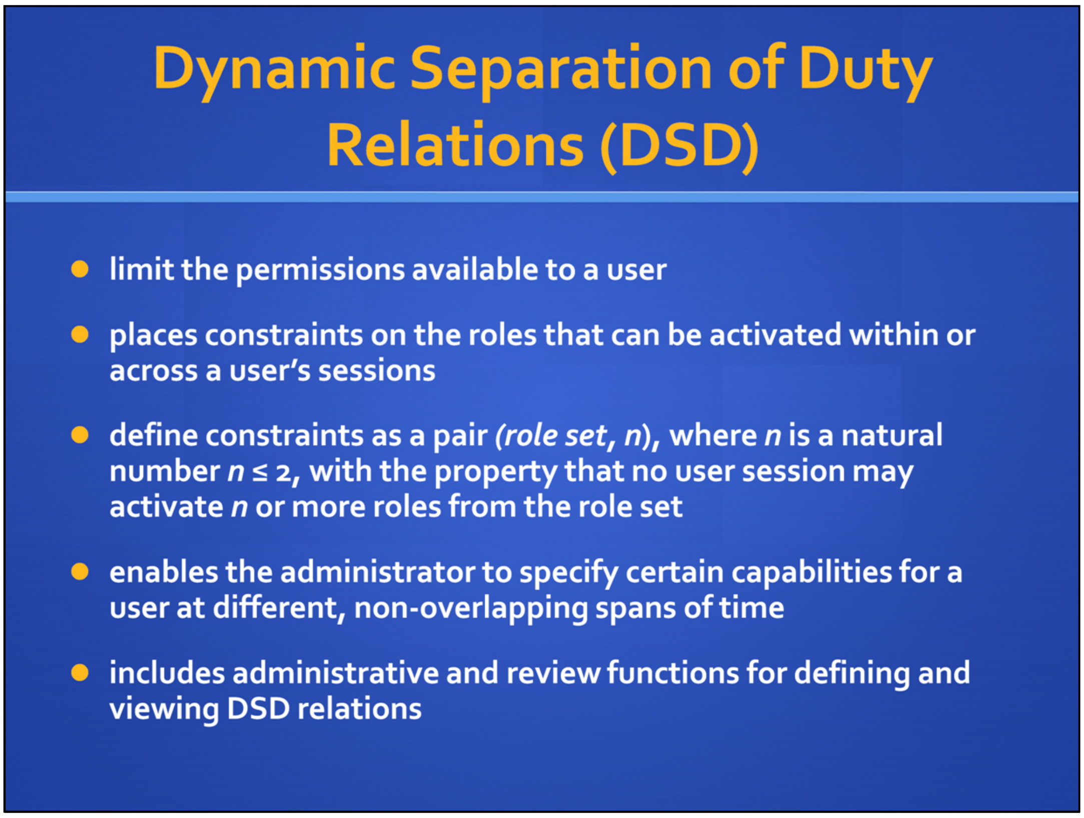
## Static Separation of Duty Relations (SSD)

- enables the definition of a set of mutually exclusive roles, such that if a user is assigned to one role in the set, the user may not be assigned to any other role in the set
- can place a cardinality constraint on a set of roles
- defined as a pair (*role set*, *n*) where no user is assigned to *n* or more roles from the role set
- includes administrative functions for creating and deleting role sets and adding and deleting role members
- includes review functions for viewing the properties of existing SSD sets

SSD and DSD are two components that add
constraints to the NIST RBAC model. The constraints are in the form of separation of duty relations, used to enforce conflict of interest policies that organizations may employ to prevent users from exceeding a reasonable level of authority for their positions.

SSD enables the definition of a set of mutually exclusive roles, such that if
a user is assigned to one role in the set, the user may not be assigned to any other role in the set. In addition, SSD can place a cardinality constraint on a set of roles.
A cardinality constraint associated with a set of roles is a number greater than one specifying a combination of roles that would violate the SSD policy. For example, the permissions associated with the purchasing function could be organized as a set of four roles, with the constraint the no user may be assigned more than three roles in the set. A concise definition of SSD is that SSD is defined as a pair (*role set, n)* where no user is assigned to *n or more roles from the role set.*

SSD includes administrative functions for creating and deleting role sets and adding and deleting role members. It also includes review functions for viewing the properties of existing SSD sets.

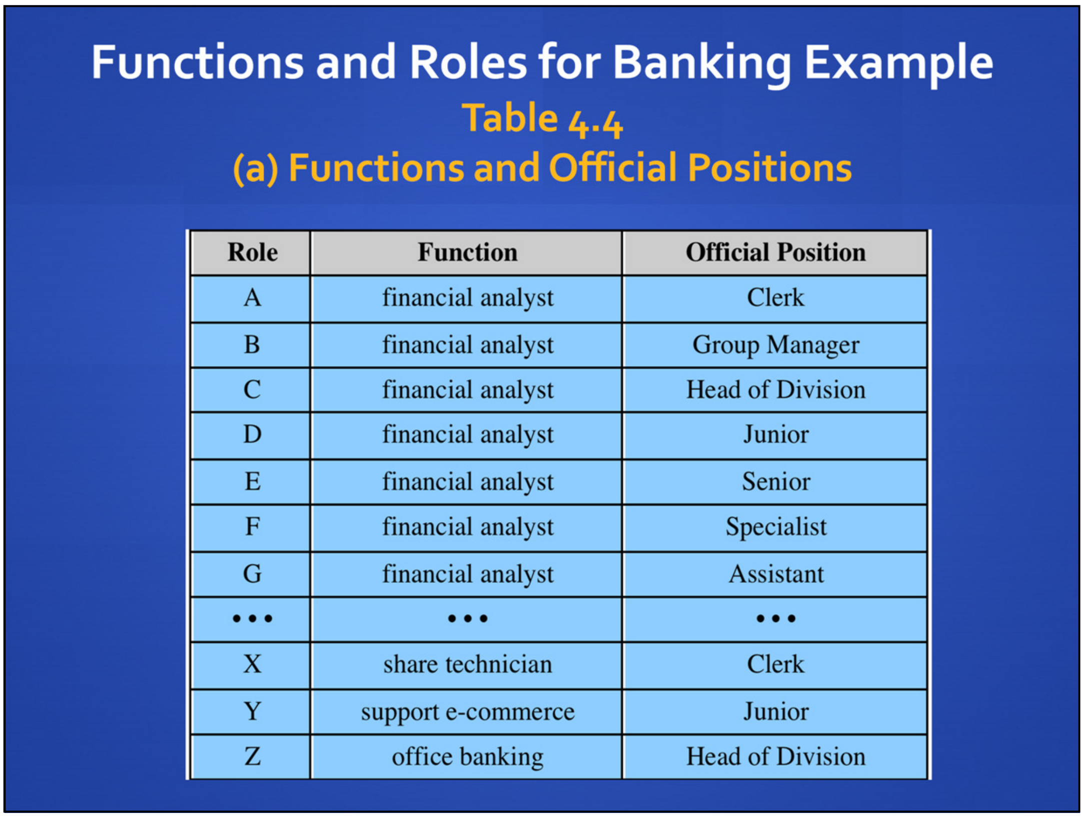


## Dynamic Separation of Duty Relations (DSD)

- limit the permissions available to a user
- places constraints on the roles that can be activated within or across a user's sessions
- define constraints as a pair *(role set, n)*, where *n* is a natural number *n* ≤ 2, with the property that no user session may activate *n* or more roles from the role set
- enables the administrator to specify certain capabilities for a user at different, non-overlapping spans of time
- includes administrative and review functions for defining and viewing DSD relations

As with SSD, DSD relations limit
the permissions available to a user. DSD specifications limit the availability of the
permissions by placing constraints on the roles that can be activated within or across
a user's sessions. DSD relations define constraints as a pair (*role set, n), where n is a*
natural number *n  2, with the property that no user session may activate n or more*
roles from the role set.

DSD enables the administrator to specify certain capabilities for a user at
different, non-overlapping spans of time. As with SSD, DSD includes administrative
and review functions for defining and viewing DSD relations.



## Functions and Roles for Banking Example

Table 4.4

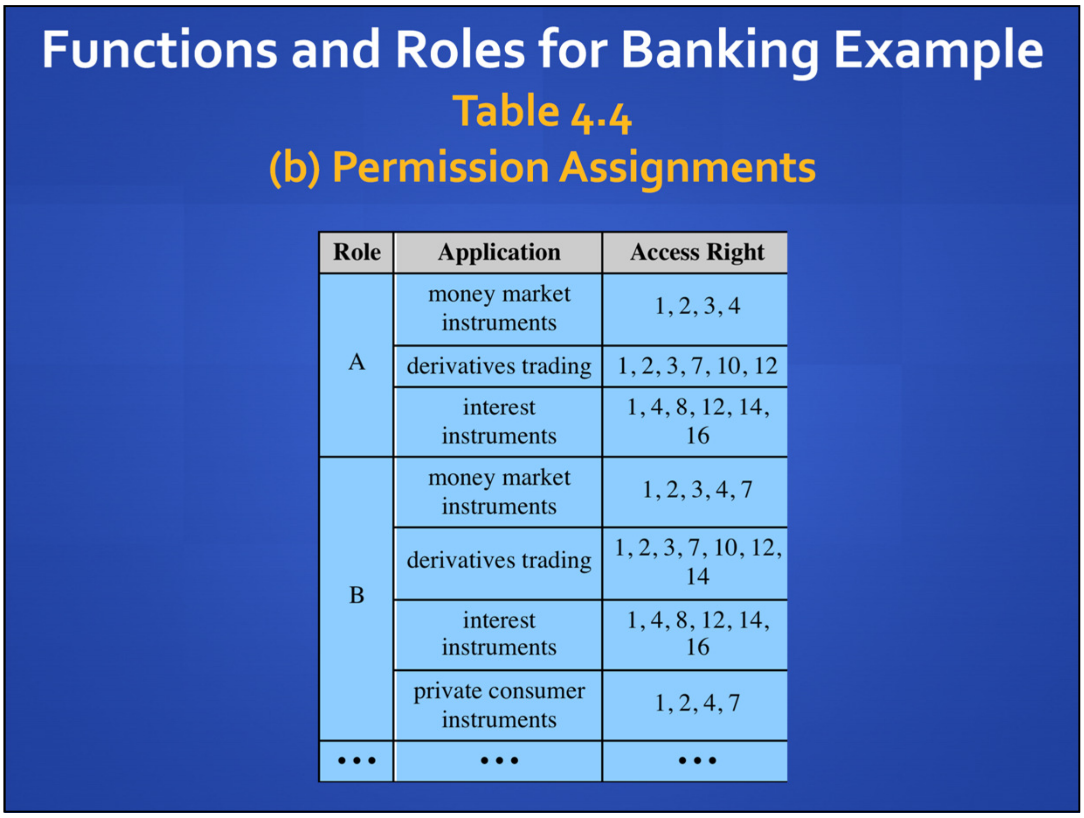
(a) Functions and Official Positions

| Role | Function | Official Position |
|---|---|---|
| A | financial analyst | Clerk |
| B | financial analyst | Group Manager |
| C | financial analyst | Head of Division |
| D | financial analyst | Junior |
| E | financial analyst | Senior |
| F | financial analyst | Specialist |
| G | financial analyst | Assistant |
| • • • | • • • | • • • |
| X | share technician | Clerk |
| Y | support e-commerce | Junior |
| Z | office banking | Head of Division |

The Dresdner Bank has implemented an RBAC system that serves as a useful practical
example [SCHA01]. The bank uses a variety of computer applications. Many
of these were initially developed for a mainframe environment; some of these older
applications are now supported on a client-server network while others remain on
mainframes. There are also newer applications on servers. Prior to 1990, a simple
DAC system was used on each server and mainframe. Administrators maintained
a local access control file on each host and defined the access rights for each employee
on each application on each host. This system was cumbersome, time-consuming,
and error-prone. To improve the system, the bank introduced an RBAC scheme,
which is systemwide and in which the determination of access rights is compartmentalized
into three different administrative units for greater security.

Roles within the organization are defined by a combination of official position
and job function. Table 4.4a provides examples. This differs somewhat from the

concept of role in the NIST standard, in which a role is defined by a job function. To some extent, the difference is a matter of terminology. In any case, the bank's role structuring leads to a natural means of developing an inheritance hierarchy based on official position. Within the bank, there is a strict partial ordering of official positions within each organization, reflecting a hierarchy of responsibility and
power. For example, the positions Head of Division, Group Manager, and Clerk are
in descending order.



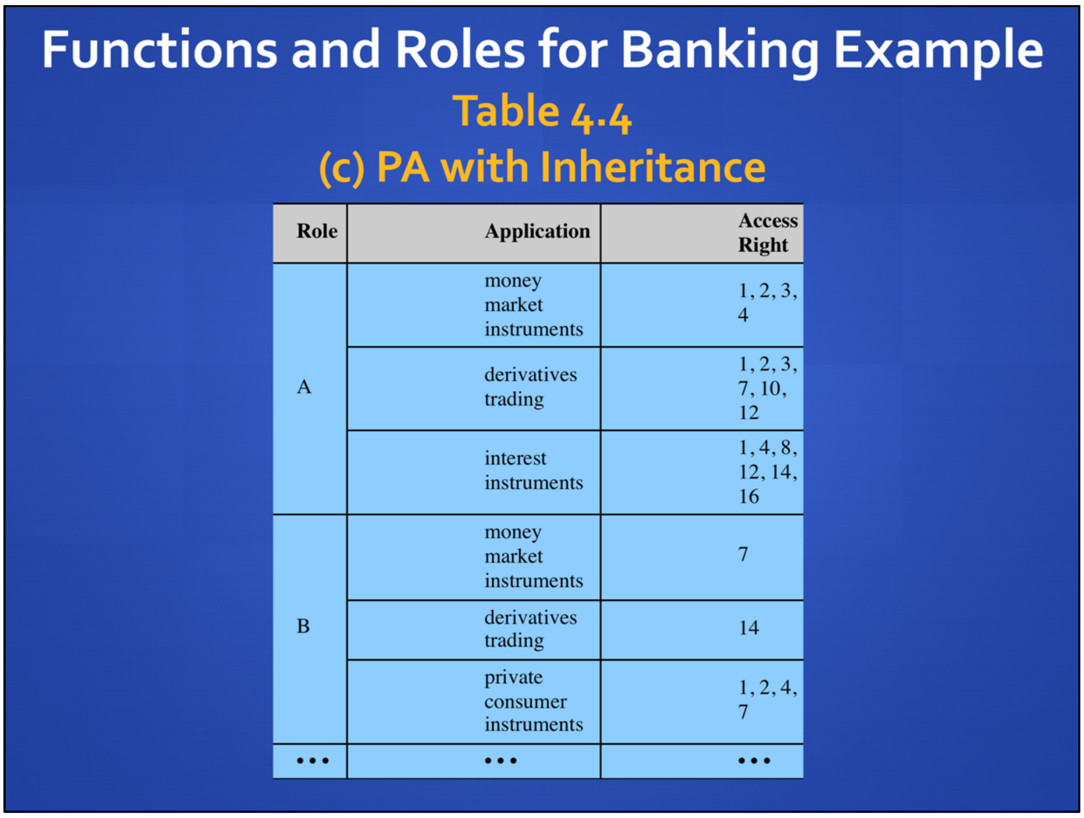
## Functions and Roles for Banking Example

### Table 4.4

### (b) Permission Assignments

| Role | Application | Access Right |
|---|---|---|
| A | money market instruments | 1, 2, 3, 4 |
| | derivatives trading | 1, 2, 3, 7, 10, 12 |
| | interest instruments | 1, 4, 8, 12, 14, 16 |
| B | money market instruments | 1, 2, 3, 4, 7 |
| | derivatives trading | 1, 2, 3, 7, 10, 12, 14 |
| | interest instruments | 1, 4, 8, 12, 14, 16 |
| | private consumer instruments | 1, 2, 4, 7 |
| • • • | • • • | • • • |

When the official position is combined with job function, there
is a resulting ordering of access rights, as indicated in Table 4.4b. Thus, the financial
analyst/Group Manager role (role B) has more access rights than the financial
analyst/Clerk role (role A). The table indicates that role B has as many or more
access rights than role A in three applications and has access rights to a fourth
application. On the other hand, there is no hierarchical relationship between office
banking/Group Manager and financial analyst/Clerk because they work in different
functional areas. We can therefore define a role hierarchy in which one role is superior
to another if its position is superior and their functions are identical.



## Functions and Roles for Banking Example

Table 4.4

(c) PA with Inheritance

| Role | Application | Access Right |
|---|---|---|
| A | money market instruments | 1, 2, 3, 4 |
|  | derivatives trading | 1, 2, 3, 7, 10, 12 |
|  | interest instruments | 1, 4, 8, 12, 14, 16 |
| B | money market instruments | 7 |
|  | derivatives trading | 14 |
|  | private consumer instruments | 1, 2, 4, 7 |
| • • • | • • • | • • • |

The role

hierarchy makes it possible to economize on access rights definitions, as suggested

in Table 4.4c.

## Figure 4.12 Example of Access Control Administration

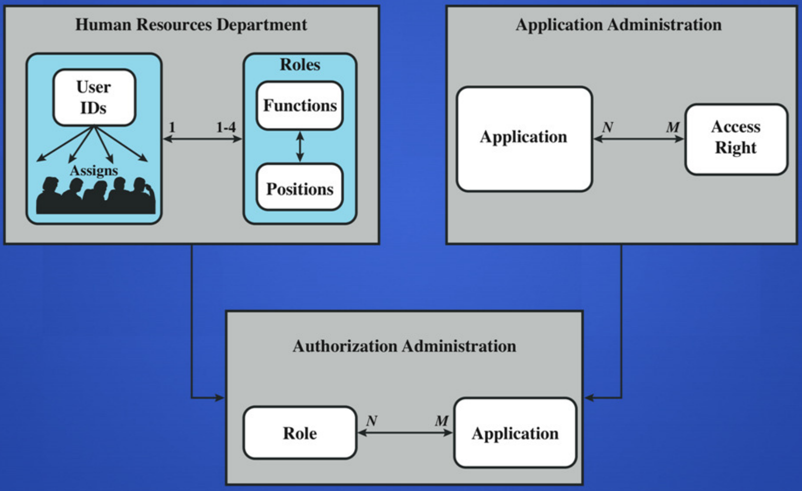


Figure 4.12 Example of Access Control Administration

In the original scheme, the direct assignment of access rights to the individual user occurred at the application level and was associated with the individual application. In the new scheme, an application administration determines the set of access rights associated with each individual application. However, a given user performing a given task may not be permitted all of the access rights associated with the application. When a user invokes an application, the application grants access on the basis of a centrally provided security profile. A separate authorization administration associated access rights with roles and creates the security profile for a use on the basis of the user's role.

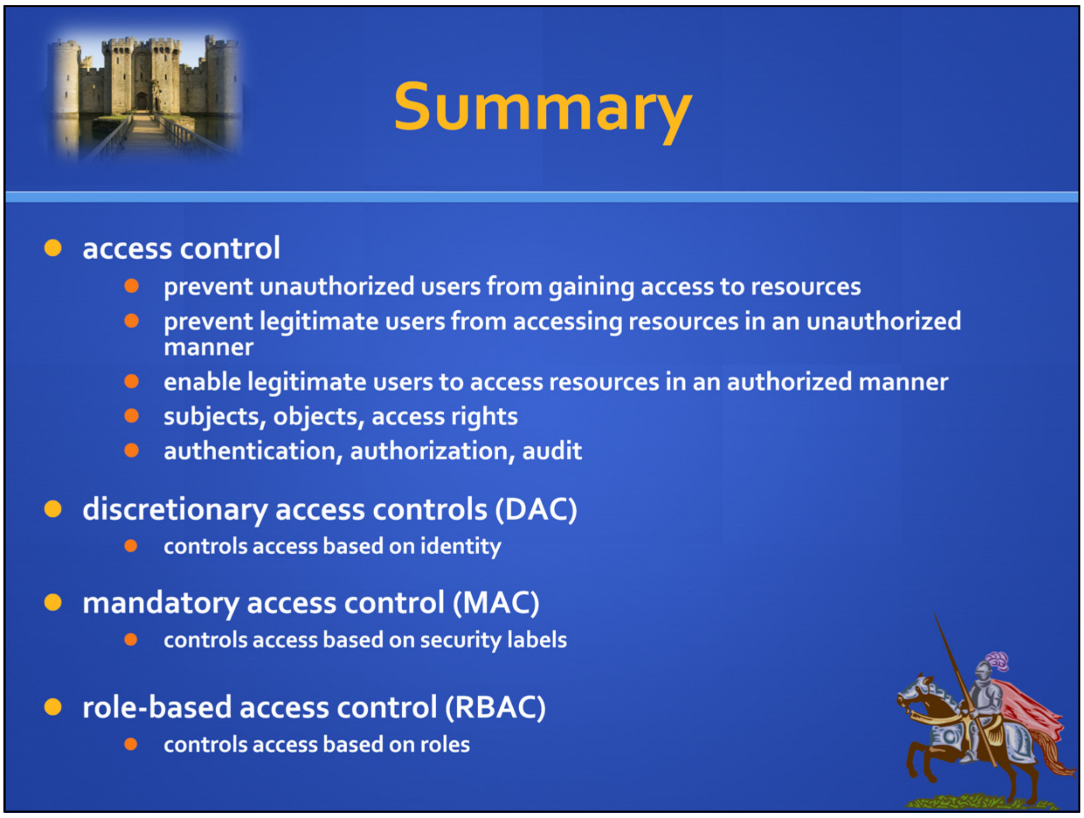
A user is statically assigned a role. In principle (in this example), each user may be statically assigned up to four roles and select a given role for use in invoking a particular application. This corresponds to the NIST concept of session. In practice, most users are statically assigned a single role based on the user's position and job function.

All of these ingredients are depicted in Figure 4.12. The Human Resource Department assigns a unique User ID to each employee who will be using the system. Based on the user's position and job function, the department also assigns one or more roles to the user. The user/role information is provided to the Authorization Administration, which creates a security profile for each user that associates the User ID and role with a set of access rights. When a user invokes an application, the application consults the security profile for that user to determine what subset of the application's access rights are in force for this user in this role.

A role may be used to access several applications. Thus, the set of access rights associated with a role may include access rights that are not associated with one of the applications the user invokes. This is illustrated in Table 4.4b. Role A has numerous access rights, but only a subset of those rights are applicable to each of the three applications that role A may invoke.

Some figures about this system are of interest. Within the bank, there are 65 official positions, ranging from a Clerk in a branch, through the Branch Manager, to a

Member of the Board. These positions are combined with 368 different job functions
provided by the human resources database. Potentially, there are 23,920 different roles, but the number of roles in current use is about 1300. This is in line with the experience other RBAC implementations. On average, 42,000 security profiles are distributed to applications each day by the Authorization Administration module.



## Summary

- access control
  - 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
  - subjects, objects, access rights
  - authentication, authorization, audit
- discretionary access controls (DAC)
  - controls access based on identity
- mandatory access control (MAC)
  - controls access based on security labels
- role-based access control (RBAC)
  - controls access based on roles

Chapter 4 summary.