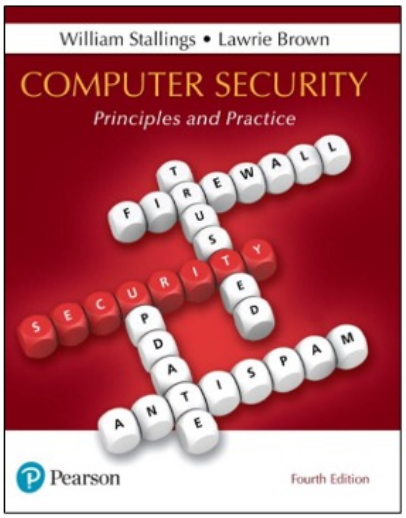
Computer Security: Principles and Practice

Fourth Edition



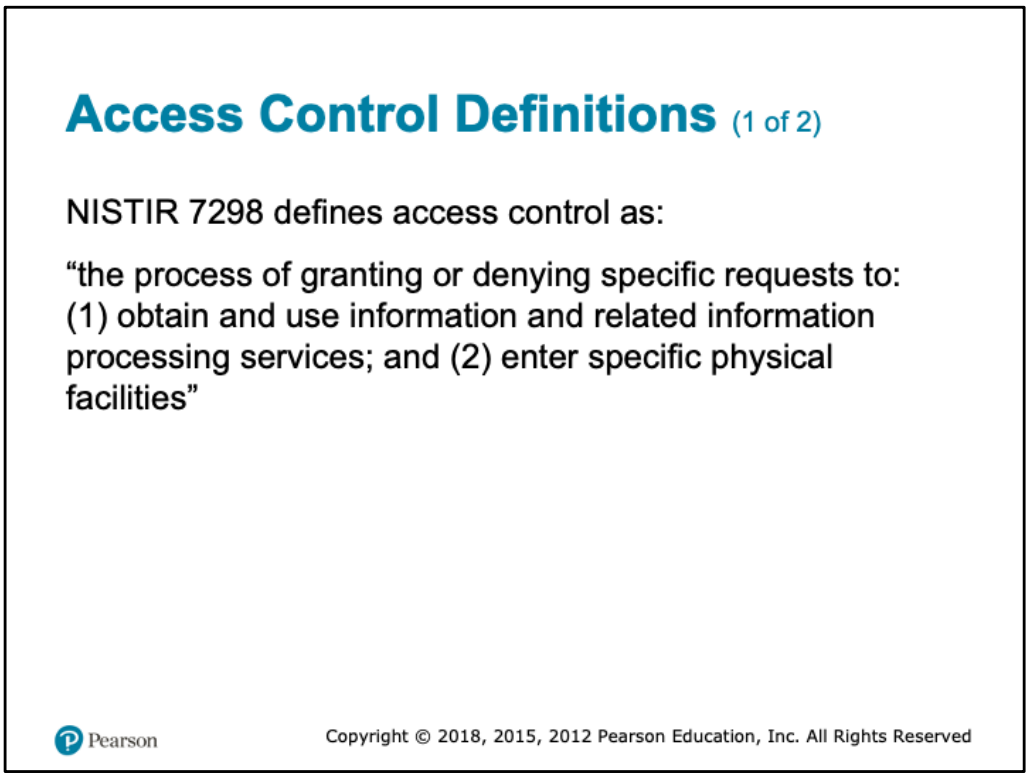
Chapter 4

Access Control

If this PowerPoint presentation contains mathematical equations, you may need to check that your computer has the following installed:
1) MathType Plugin
2) Math Player (free versions available)
3) NVDA Reader (free versions available)

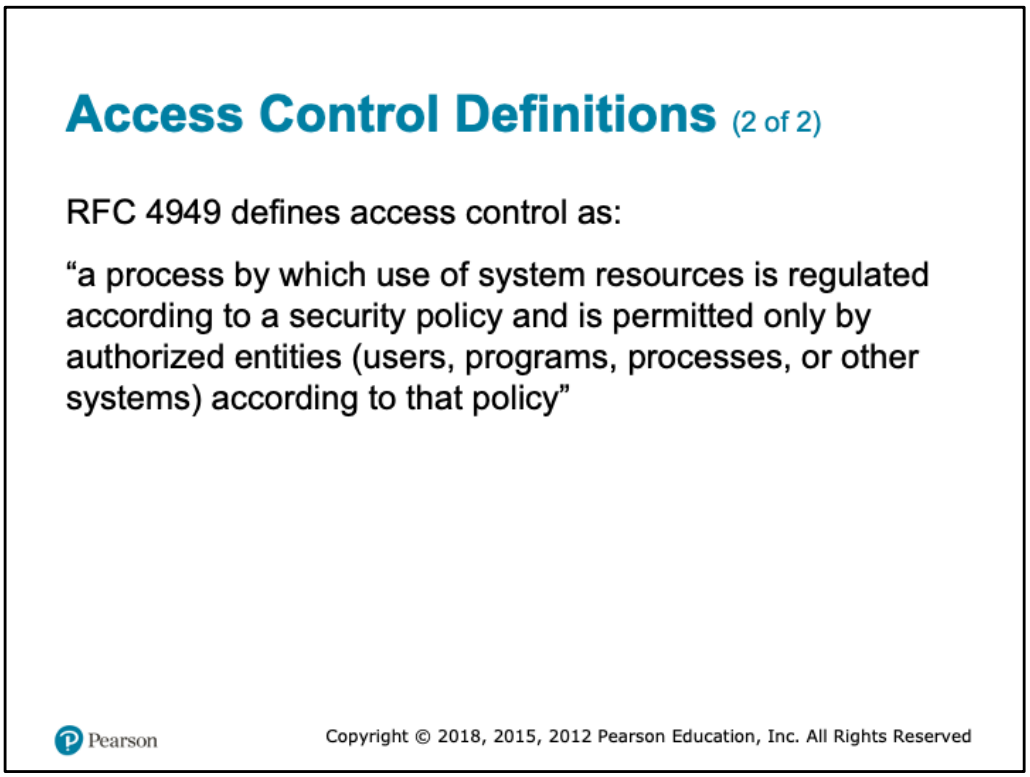
We can view access control as a central element of computer security. The principal objectives of computer security are to prevent unauthorized users from gaining access to resources, to prevent legitimate users from accessing resources in an unauthorized manner, and to enable legitimate users to access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

We begin this chapter with an overview of some important concepts. Next we look at three widely used techniques for implementing access control policies. We then turn to a broader perspective of the overall management of access control using identity, credentials, and attributes. Finally, the concept of a trust frameworkis introduced.

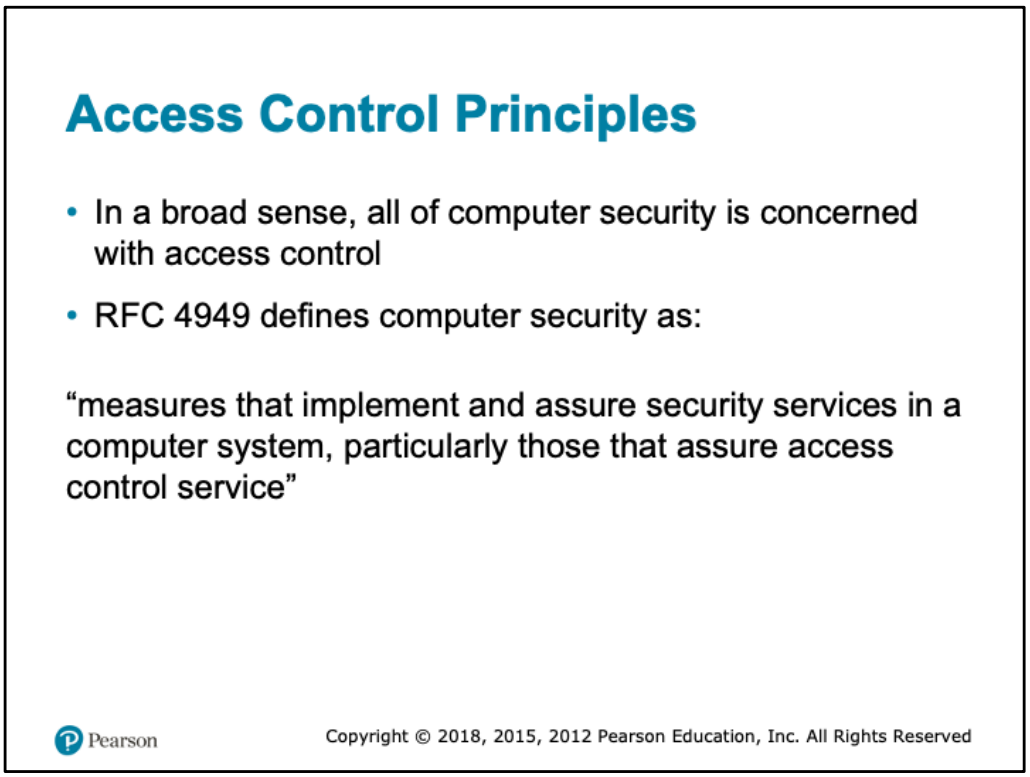


Two definitions of access control are useful in understanding its scope.

1. NISTIR 7298 (*Glossary of Key Information Security Terms* , May 2013), defines access control as the process of granting or denying specific requests to: (1) obtain and use information and related information processing services; and (2) enter specific physical facilities.

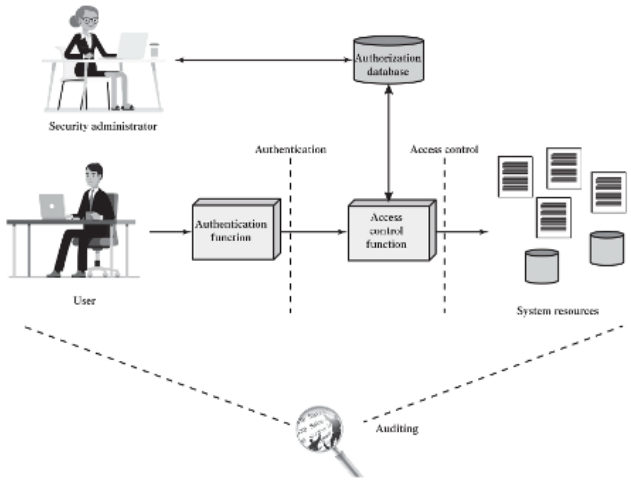


2. RFC 4949, *Internet Security Glossary* , defines access control as a process by which use of system resources is regulated according to a security policy and is permitted only by authorized entities (users, programs, processes, or other systems) according to that policy.



In a broad sense, all of computer security is concerned with access control. Indeed, RFC 4949 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.

Figure 4.1 Relationship Among Access Control and Other Security Functions



Source: Based on [SAND94].

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.

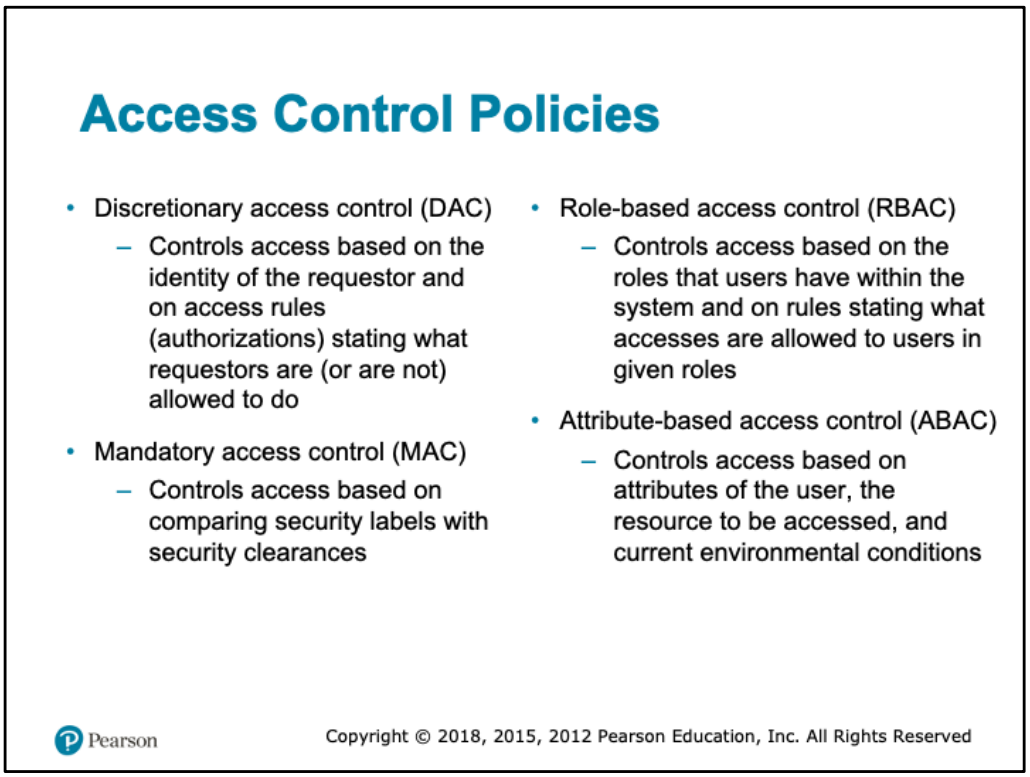
• **Audit:** An independent review and examination of system records and activities in order to test for adequacy of system controls, to ensure compliance with established policy and operational procedures, to detect breaches in security, and to recommend any indicated changes in control, policy and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

The security administrator creates and maintains the authorization database. The user accesses the systems resources after being authenticated by the authentication function. The access is controlled via access control function. A data exchange occurs between access control function and authorization database. Audit consists of an independent review and examination of system records and activities.



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.

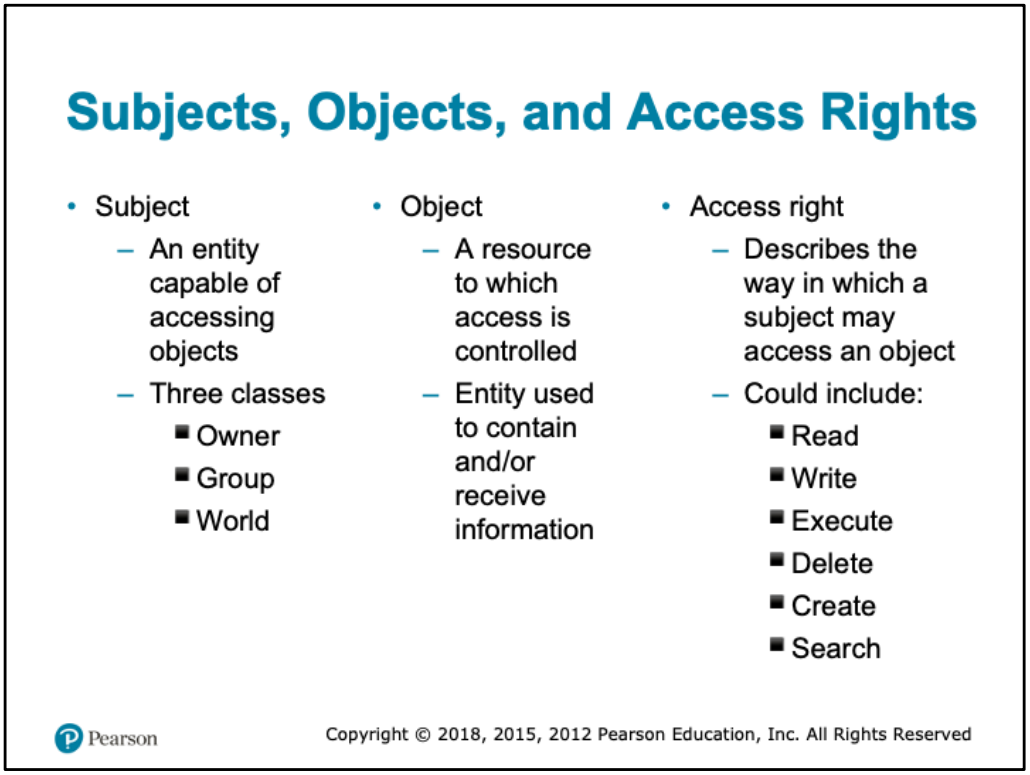
• **Mandatory access control** (MAC): Controls access based on comparing security labels (which indicate how sensitive or critical system resources are) with security clearances (which indicate system entities are eligible to access certain resources). This policy is termed *mandatory because an entity 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.

• **Attribute-based access control** (ABAC): Controls access based on attributes of the user, the resource to be accessed, and current environmental conditions.

DAC is the traditional method of implementing access control, and is examined in Sections 4.3 and 4.4. 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 27. Both RBAC and ABAC have become increasingly popular, and are examined in Sections 4.5 and 4.6, respectively.

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



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.

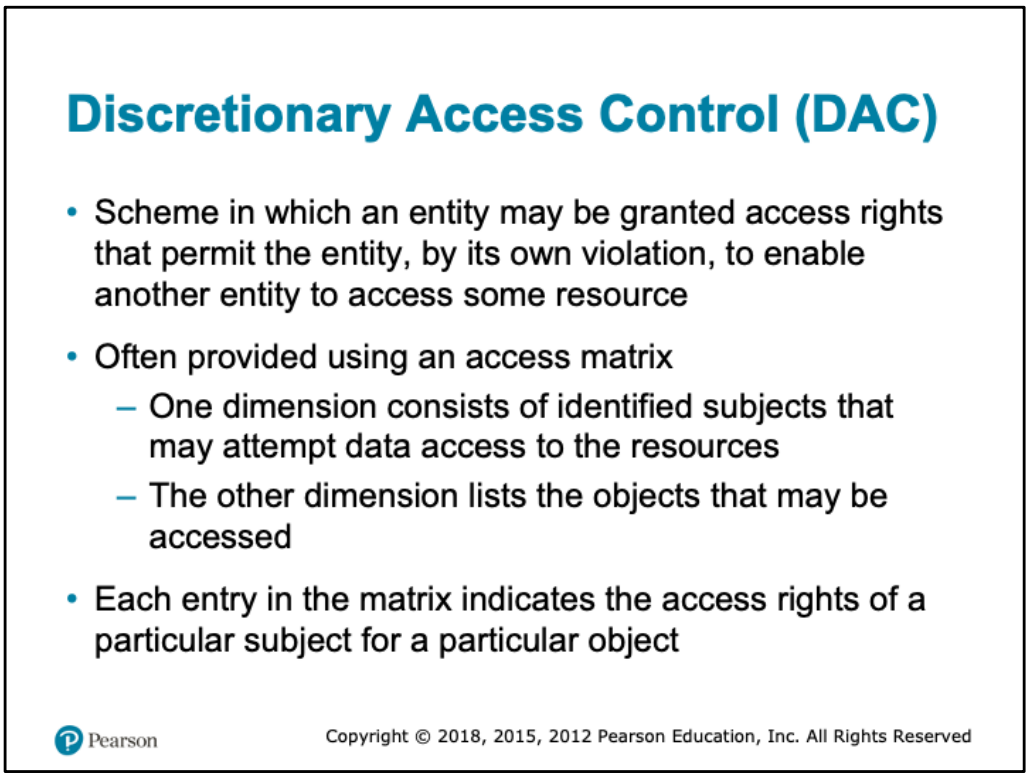
• **Write**: User may add, modify, or delete data in system resource (e.g., files, records, programs). Write access includes read access.

• **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.

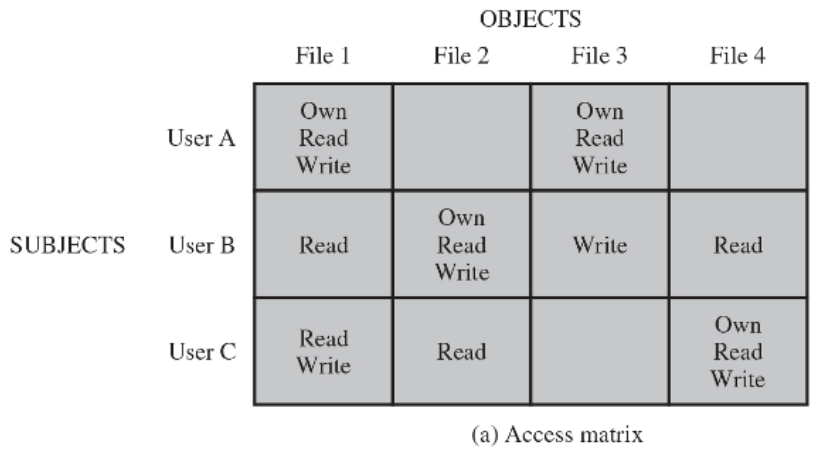
• **Search:** User may list the files in a directory or otherwise search the directory.



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 4.2 Example of Access Control Structures (1 of 2)



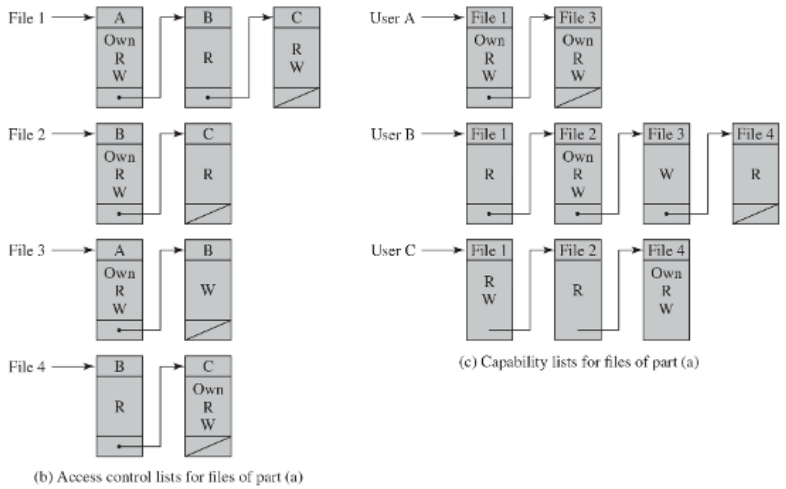
		OBJECTS			
		File 1	File 2	File 3	File 4
SUBJECTS	User A	Own Read Write		Own Read Write	
	User B	Read	Own Read Write	Write	Read
	User C	Read Write	Read		Own Read Write

(a) Access matrix

Figure 4.2a, 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.

The columns have the following headings from left to right. Subjects, Objects, file 1, Objects File 2, Objects, file 3, Objects, file 4, . The row entries are as follows. Row 1. User A, own read write, Blank, Own, read, write, Blank. Row 2. User B, Read, Own, read, write, Write, Read. Row 3. User C, Read write, Read, Blank, Own read write.

Figure 4.2 Example of Access Control Structures (2 of 2)



(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 4.2b. 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 4.2c). 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 unforgeable. 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.

The diagram is as follows. b. Access control lists for files of part a. Control lists for four files, from 1 to 4, are presented. The control lists have three parts. The control lists for file 1 has lists for user A, user B, and user C. The user A has own, read, and write, access rights. A node connects to user B list. The user B has read access right, and a node connects to user C list. The user C has read and write access rights, and the loop is closed. The file 2 has two user list. User list B has own, read, and write access rights, and a node connects the user B list connects to user C list. The iser C list has read access right, and the loop is closed. The file 3 has two user list. User list A has own, read, and write access rights, and a node connects the user A list connects to user B list. The user B list has write access right, and the loop is closed. The file 4 has two user list. User list B has read access right, and a node connects the user B list connects to user C list. The user C list has own, read, and write, access right, and the loop is closed. C. Capability lists for files of part a. The capability lists are presented for User A, user B, and user C. The user A has two capability lists. The list for file 1 has own, read, and write, access rights, and a node connects to list file 3. The list for file 3 has own, read, and write, access rights, and the loop is closed. The user B has four capability lists. The list for file 1 has read access right, and a node connects to list file 2. The file 2 has own, read, and write, access rights, and a node connects to file 3 list. The file 3 list has write access right, and a node connects to file list 4. The file 4 list has read access right, and the loop is closed. The user C has three capability lists. The file 1 list has read and write access right, and the process moves to file 2 list. The file 2 list has read access right, and the process moves to file list 4. The file list has own, read, and write, access right.

Table 4.2 Authorization Table for Files in Figure 4.2

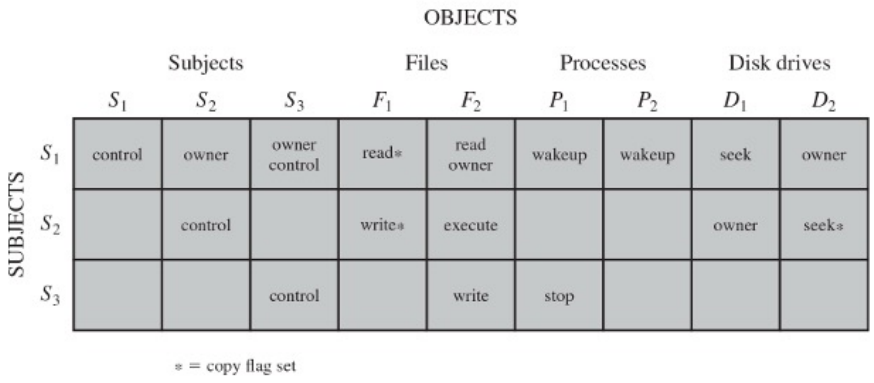
Subject	Access Mode	Object
A	Own	File 1
A	Read	File 1
A	Write	File 1
A	Own	File 3
A	Read	File 3
A	Write	File 3
B	Read	File 1
B	Own	File 2
B	Read	File 2
B	Write	File 2
B	Write	File 3
B	Read	File 4

Subject	Access Mode	Object
C	Read	File 1
C	Write	File 1
C	Read	File 2
C	Own	File 4
C	Read	File 4
C	Write	File 4

Pearson

[SAND94] proposes a data structure that is not sparse, like the access matrix, but is more convenient than either ACLs or capability lists (Table 4.2). 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.

Figure 4.3 Extended Access Control Matrix



SUBJECTS	OBJECTS: Subjects S_1	Subjects S_2	Subjects S_3	Files F_1	Files F_2	Processes P_1	Processes P_2	Disk drives D_1	Disk drives D_2
S_1	control	owner	owner control	read*	read owner	wakeup	wakeup	seek	owner
S_2		control		write*	execute			owner	seek*
S_3			control		write	stop			

* = copy flag set

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 4.3 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 4.3, S1 may read file F1, because ‘read’ appears in A[S1, F1].

From a logical or functional point of view, a separate access control module is associated with each type of object (Figure 4.4). 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:

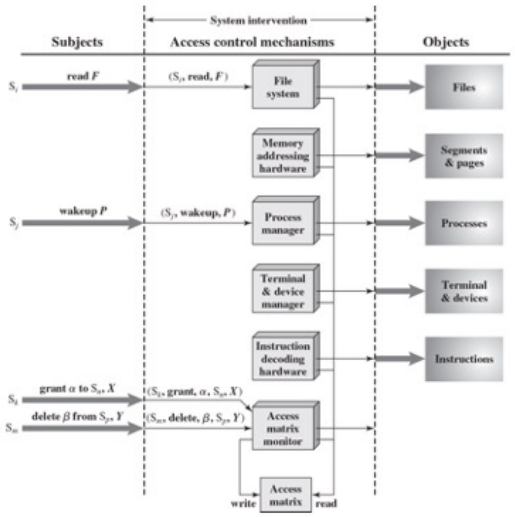
1. A subject S0 issues a request of type α for object X.

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 (S0, α, X) to the controller for X.

3. The controller interrogates the access matrix A to determine if α is in A[S0, 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.

The columns have the following headings from left to right. Subjects, Objects, Subjects, S sub 1, Objects, Subjects, S sub 2, Objects, Subjects, S sub 3, Objects, Files, F sub sub 1, Objects, Files, F sub sub 2, Objects, Processes, P sub 1, Objects, Processes, P sub 2, Objects, Disk Drives, D sub 1, Objects, Disk Drives, D sub 2. The row entries are as follows. Row 1. S sub 1, Control, Owner, Owner control, Read asterisk, Read owner, Wakeup, Wakeup, Seek, Owner. Row 2. S sub 2, Blank, Control, Blank, write asterisk, execute, Blank, Blank, Owner, Seek asterisk. Row 3. S sub 3, Blank, Blank, Control, Blank, Write, Stop, Blank, Blank, Blank.

Figure 4.4 An Organization of the Access Control Function



Pearson

Figure 4.4 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.

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. The subject S sub i executes read F. The command for S sub i to read F enters file system in access control mechanism. The file system accesses the object, files. The memory addressing hardware accesses objects, in segments and pages. The subject S sub j executes wakeup P, the command enters process manager, which accesses the processes in objects. The terminal and device manager and instruction decoding hardware accesses objects terminal and devices and instructions, respectively. The subject S sub k executes grant alpha to S sub n, X, and the subject S sub m wants to delete beta from S sub p, Y. The commands enter access matrix monitor. The access matrix monitor writes and reads the commands in the access matrix. All the commands in the system control mechanisms are executed by reading the

access matrix.

Table 4.3 Access Control System Commands (1 of 2)

Rule	Command (by S_0)	Authorization	Operation
R1	**transfer** $\begin{Bmatrix}\alpha^* \\ \alpha\end{Bmatrix}$ **to** S, X	"α^*" in $A[S_0, X]$	store $\begin{Bmatrix}\alpha^* \\ \alpha\end{Bmatrix}$ in $A[S, X]$
R2	**grant** $\begin{Bmatrix}\alpha^* \\ \alpha\end{Bmatrix}$ **to** S, X	'owner' in $A[S_0, X]$å	store $\begin{Bmatrix}\alpha^* \\ \alpha\end{Bmatrix}$ in $A[S, X]$
R3	**delete** α **from** S, X	'control' in $A[S_0, S]$ or 'owner' in $A[S_0, X]$	delete α from $A[S, X]$
R4	$w \leftarrow$ **read** S, X	'control' in $A[S_0, S]$ or 'owner' in $A[S_0, X]$	copy $A[S, X]$ into w
R5	**create object** X	None	add column for X to A; store 'owner' in $A[S_0, X]$

Pearson

The model also includes a set of rules that govern modifications to the access matrix, shown in Table 4.3. 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.

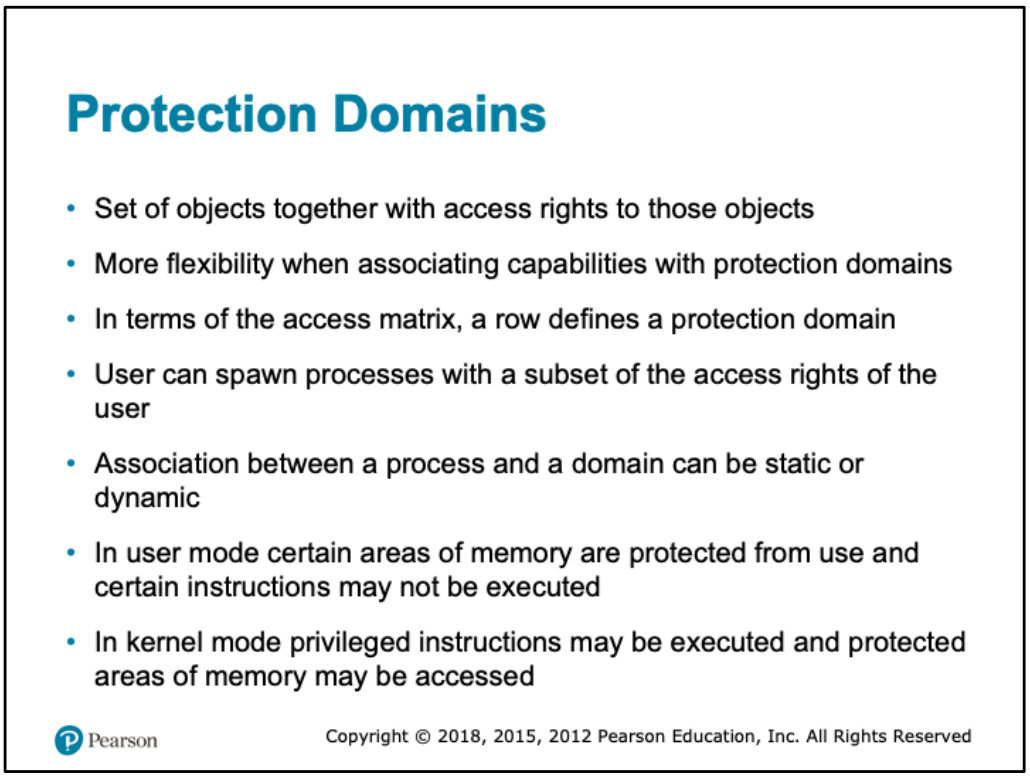
The remaining rules in Table 4.3 govern the creation and deletion of subjects and objects. Rule R5 states that any subject can create a new object, which it owns, and can then grant and delete access to the object. Under rule R6, the owner of an object can destroy the object, resulting in the deletion 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 4.3 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 4.3, *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 4.3, *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.

Table 4.3 Access Control System Commands (2 of 2)

Rule	Command (by S_0)	Authorization	Operation
R6	**destroy object** *X*	'owner' in $A[S_0, X]$	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 $A[S_0, S]$	delete row for *S* from *A*; execute **destroy object** *S*



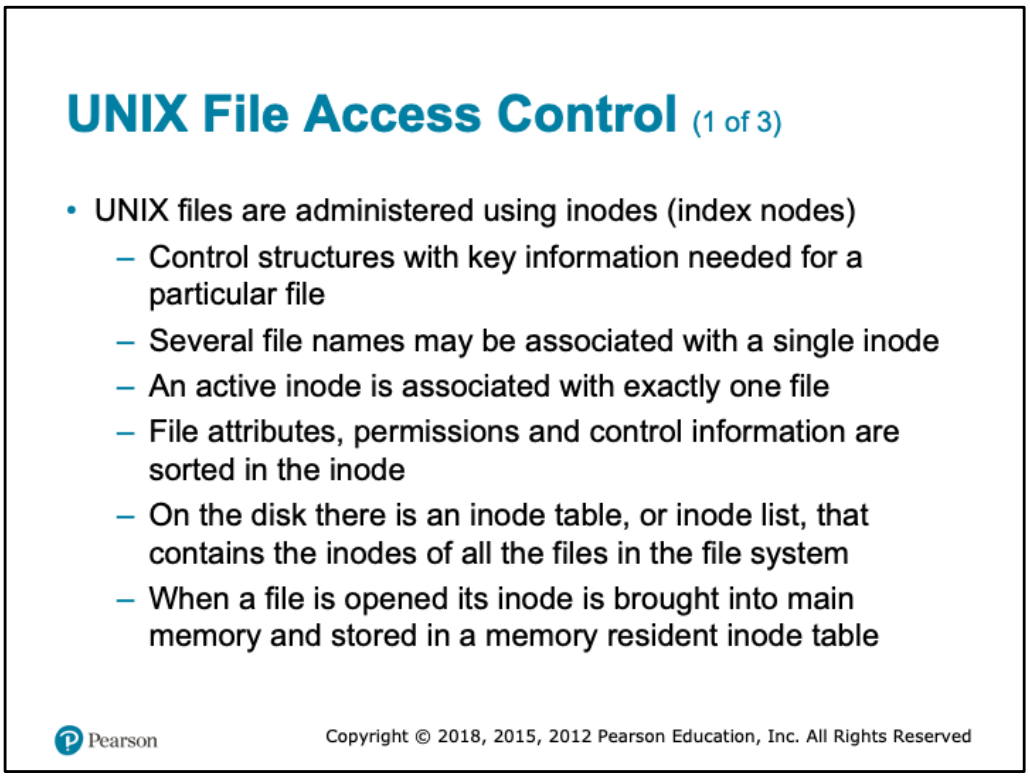
The access control matrix model that we have discussed so far associates a set of capabilities with a user. A more general and more flexible approach, proposed in [LAMP71], is to associate capabilities with protection domains. A protection domain is a set of objects together with access rights to those 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.



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.

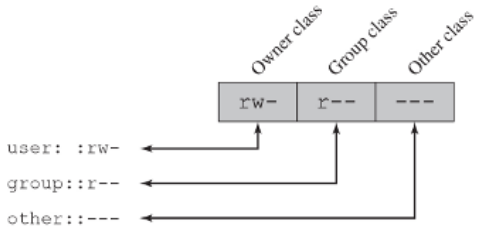
Directories are structured in a hierarchical tree. Each directory can contain files and/or other directories. A directory that is inside another directory is referred to as a subdirectory. A directory is simply a file that contains a list of file names plus pointers to associated inodes. Thus, associated with each directory is its own inode.

UNIX File Access Control (2 of 3)

- Directories are structured in a hierarchical tree
  - May contain files and/or other directories
  - Contains file names plus pointers to associated inodes

UNIX File Access Control (3 of 3)

- 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



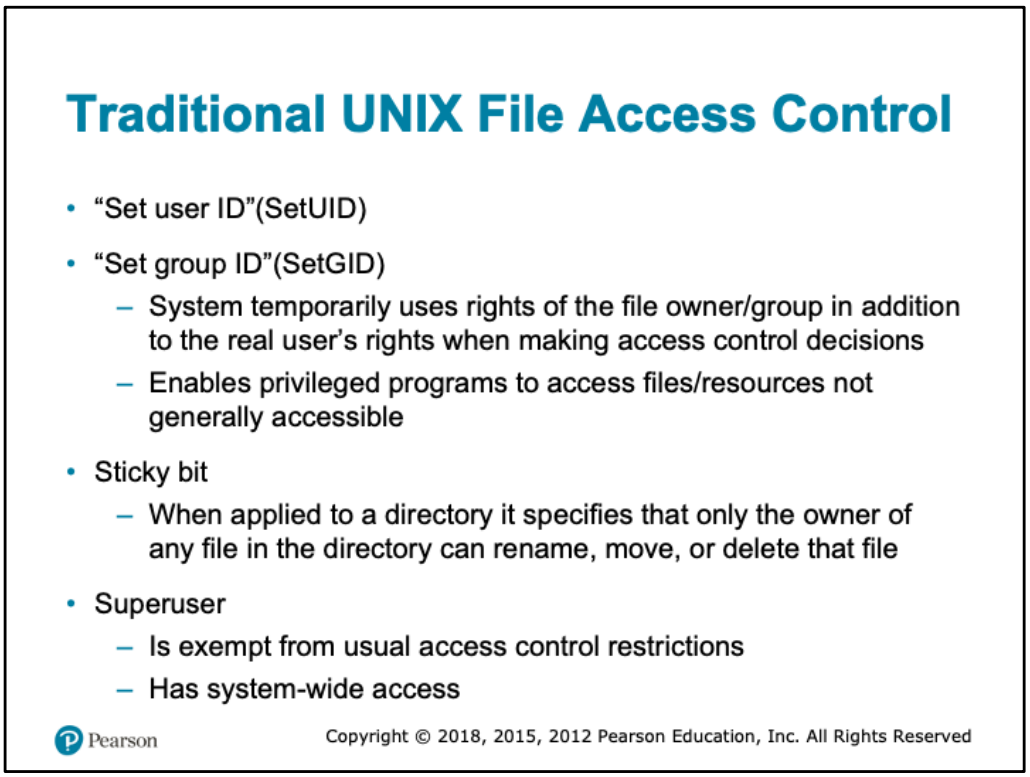
(a) Traditional UNIX approach (minimal access control list)

Figure 4.5 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 4.5a 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.

The list has access control assigned for user, group, and other. The user is assigned read and write access control. The group class is assigned read access control. The other class is assigned no access control.



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

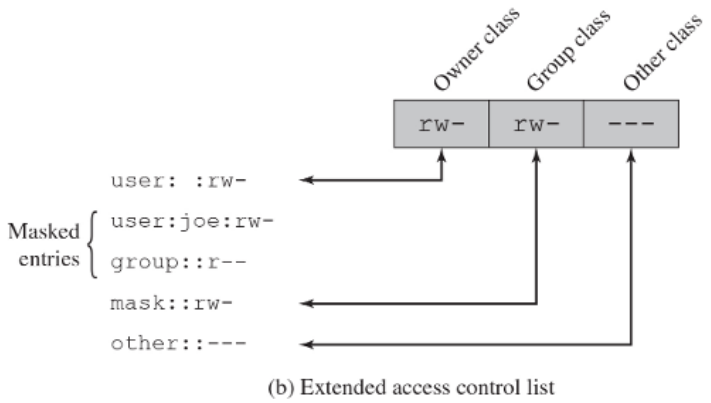
directories.

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.

Figure 4.5 UNIX File Access Control



(b) Extended access control list

FreeBSD and most UNIX implementations that support extended ACLs use the following strategy (e.g., Figure 4.5b):

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.

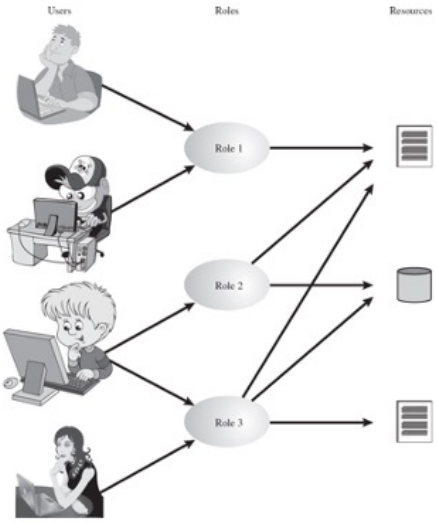
3. Additional named users and named groups may be associated with the file, each with a 3-bit permission field. The permissions listed for a named user or named group are compared to the mask field. Any permission for the named user or named group that is not present in the mask field is disallowed.

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.

The list has access control assigned for user, group, and other. The user is assigned read and write access control for owner class. The masked entries as follows. user joe, with read and write access right, and group has read access right. The group class is assigned read and write access control. The mask class is assigned to read write read write control in group class. The other control is assigned to no access control in other class.

Figure 4.6 Users, Roles, and Resources



Pearson

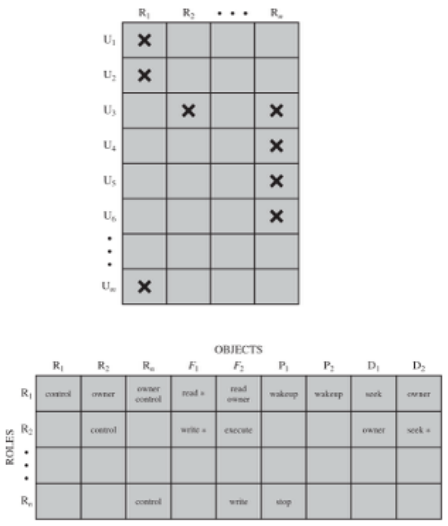
Traditional DAC systems define the access rights of individual users and groups of users. In contrast, RBAC is based on the roles that users assume in a system rather than the user's identity. Typically, RBAC models define a role as a job function within an organization. RBAC systems assign access rights 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, FIPS PUB 140-3, (Security Requirements for Cryptographic Modules September 2009), 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 4.6). 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.

The relationship of users to roles is many to many, as is the relationship of roles to resources, or system objects. The set of users changes, in some environments frequently, and the assignment of a user to one or more roles may also be dynamic. 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 4.7 Access Control Matrix Representation of RBAC



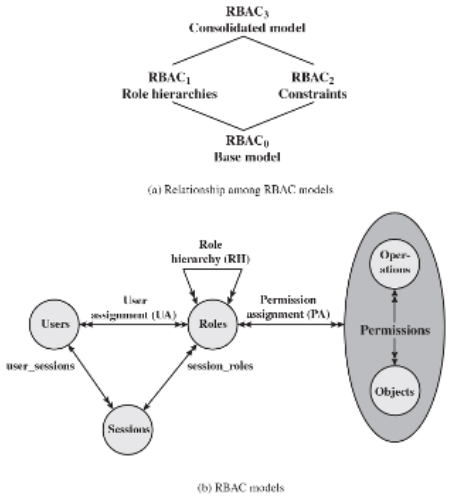
We can use the access matrix representation to depict the key elements of an RBAC system in simple terms, as shown in Figure 4.7. The upper matrix relates individual users to roles. Typically there are many more users than roles. Each matrix entry is either blank or marked, the latter indicating that 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 Chapter 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.

The first table lists the users and roles. The columns have the following headings from left to right. Users, R sub 1, R sub 2, Ellipsis, R sub n, . The row entries are as follows. Row 1. U sub 1, Cross, Blank, Blank, Blank. Row 2.

U sub 2, Cross, Blank, Blank, Blank. Row 3. U sub 3, Blank, Cross, Blank, Cross. Row 4. U sub 4, Blank, Blank, Blank, Cross. Row 5. U sub 5, Blank, Blank, Blank, Cross. Row 6. U sub 6, Blank, Blank, Blank, Cross. Row 7. Ellipsis, Blank, Blank, Blank. Row 8. U sub m, Cross, Blank, Blank, Blank. The second table lists the roles and objects. The columns have the following headings from left to right. Roles, Objects, R sub 1, Objects, R sub 2, Objects, R sub n, Objects, F sub 1, Objects, F sub 2, Objects, P sub 1, Objects, P sub 2, Objects, D sub 1, Objects, D sub 2. The row entries are as follows. Row 1. R sub 1, Control, Owner, Owner, control, Read asterisk, Read, owner, Wakeup, Wakeup, Seek, Owner. Row 2. R sub 2, Blank, Control, Blank, Write asterisk, execute, Blank, Blank, Owner, Seek astersik. Row 3. Ellipsis, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank, Blank. Row 4. R sub n, Blank, Blank, Control, Blank, Write, Stop, Blank, Blank, Blank.

Figure 4.8 A Family of Role-Based Access Control Models



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.

[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 4.8a. and Table 4.4. RBAC0 contains the minimum functionality for an RBAC system. RBAC1 includes the RBAC0 functionality and adds role hierarchies, which enable one role to inherit permissions from another role. RBAC2 includes RBAC0 and adds constraints, which restrict the ways in which the components of a RBAC system may be configured. RBAC3 contains the functionality of RBAC0, RBAC1, and RBAC2.

RBAC0 Figure 4.8b, without the role hierarchy and constraints, contains the four types of entities in an RBAC0 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.

The arrowed lines in Figure 4.8b 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.

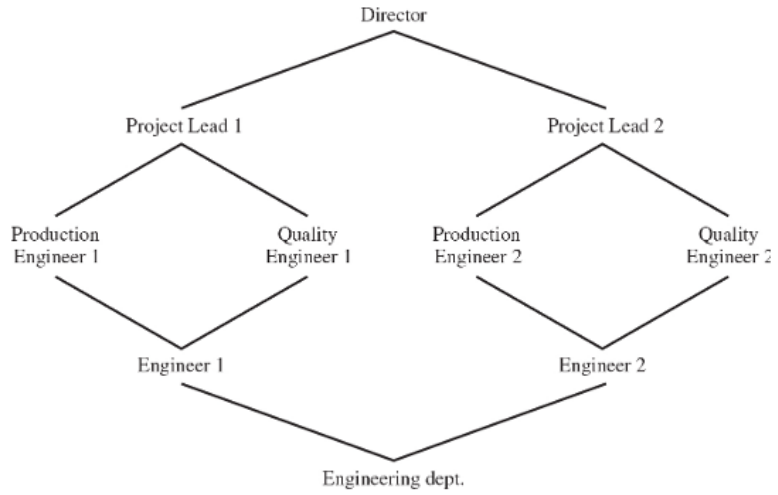
The first diagram illustrates the relationships among R B A C models. A base model, R B A C sub 0, contains the minimum functionality for an R B A C system. R B A C sub 1 includes the R B A C sub 0 functionality and adds role hierarchies. R B A C sub 2 includes R B A C sub 0 and adds constraints. R B A C sub 3 contains the functionality of R B A C sub 0, R B A C sub 1, and R B A C sub 2. The second diagram illustrates the R B A C models. The model has 4 types of entities, which are as follows. An entity, permission, has operations and objects, roles, sessions, and user. Double bidirectional arrows are present between entities with assignments, which are as follows. Operations and objects, permissions and roles labeled, permission assignment P A, users and roles user assignment U A, and a loop at roles labeled, role hierarchy R H. Arrows between users and sessions and roles and session are labeled, user underscore sessions and session underscore roles, respectively.

Table 4.4 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.

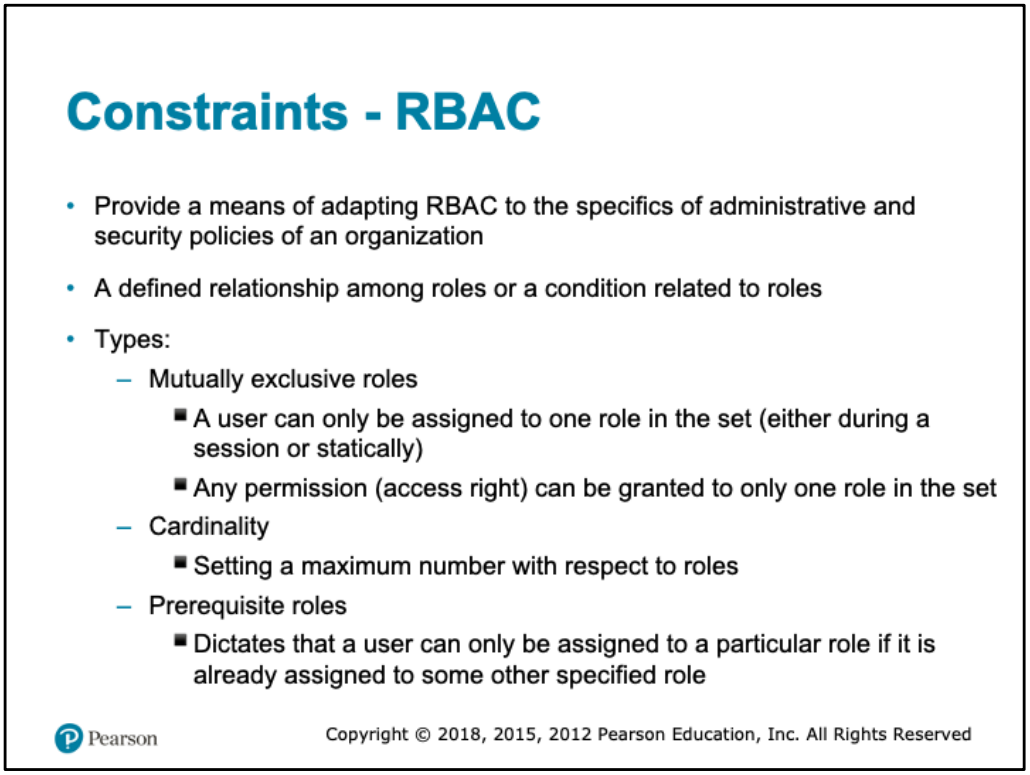
Figure 4.9 Example of Role Hierarchy



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 4.9 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 4.9, 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.

Director branches into project lead 1 and project lead 2. Project lead 1 branches into production engineer 1 and quality engineer 1. Production engineer 1 and quality engineer 1 combine to engineer 1. Project lead 2 branches into production engineer 2 and quality engineer 2. Production engineer 2 and quality engineer 2 combine to engineer 2. Engineer 1 and engineer 2 combine into engineering department.



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).

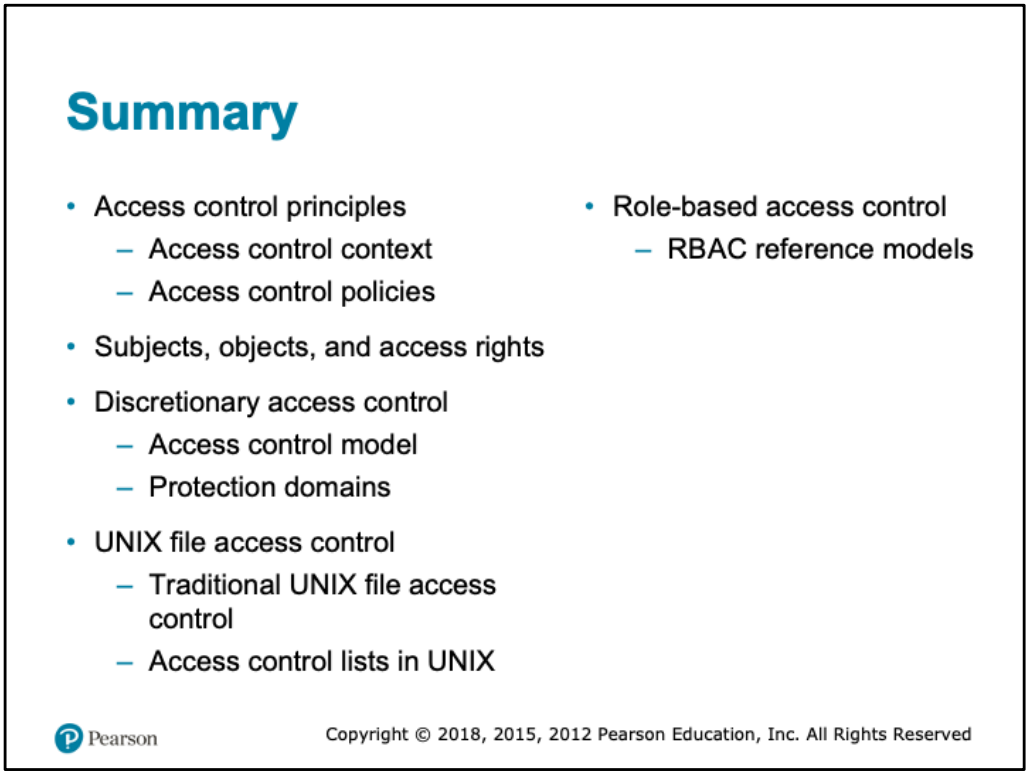
2. Any permission (access right) can be granted to only one role in the set.

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 role**, 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 4.9 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.



Chapter 4 summary.

Copyright