CHAPTER
# 14

# RESOURCE SECURITY AND PROTECTION: ACCESS AND FLOW CONTROL

## 14.1 INTRODUCTION

Security and protection deal with the control of unauthorized use and the access to hardware and software resources of a computer system. Business organizations and government agencies heavily use computers to store information to which unauthorized access must be prevented. For example, in business organizations, this information includes financial or personnel records, monetary transactions, legal contracts, payrolls, product information, future planning and strategies, etc. With the prevalent use of electronic fund transfers, the banking industry has become highly susceptible to malicious access and use. Examples in government agencies include strategic military information, CIA files, FBI files, blueprints of military hardware, information about military installations, etc.

Clearly, an unauthorized use of a company's confidential information can have catastrophic financial consequences and the unauthorized use of secret information of a government can have serious implications for the security of that nation. Therefore, with the widespread use of computers in business and government organizations, the security and protection of computer systems have become extremely important factors.

Note that not only should the misuse of secret information be prevented, but the destruction of such information should be prevented as well. For example, the destruction of information about customer account balances and bank transactions can have serious socioeconomic ramifications.

In this chapter, we study models of protection and techniques to enforce security and protection in computer systems.

## 14.2 PRELIMINARIES

### 14.2.1 Potential Security Violations

Anderson [2] has classified the potential security violations into three categories:

**Unauthorized information release.** This occurs when an unauthorized person is able to read and take advantage of the information stored in a computer system. This also includes the unauthorized use of a computer program.

**Unauthorized information modification.** This occurs when an unauthorized person is able to alter the information stored in a computer. Examples include changing student grades in a university database and changing account balances in a bank database. Note that an unauthorized person need not read the information before changing it. Blind writes can be performed.

**Unauthorized denial of service.** An unauthorized person should not succeed in preventing an authorized user from accessing the information stored in a computer. Note that services can be denied to authorized users by some internal actions (like crashing the system by some means, overloading the system, changing the scheduling algorithm) and by external actions (such as setting fire or disrupting electrical supply).

### 14.2.2 External vs. Internal Security

Computer systems security can be divided into two parts: external security and internal security. External security, also called physical security, deals with regulating access to the premises of computer systems, which include the physical machine (hardware, disks, tapes, power supply, air conditioning), terminals, computer console, etc. External security can be enforced by placing a guard at the door, by giving a key or secret code to authorized persons, etc.

Internal security deals with the access and use of computer hardware and software information stored in the computer system. Aside from external and internal securities, there is an issue of *authentication* by which a user "logs into" the computer system to access the hardware and the software resources.

Clearly, issues involved in external security are simple and administrative in nature. In this chapter, we will mainly be concerned with the internal security in computer systems, which is more challenging and subtle.

### 14.2.3 Policies and Mechanisms

Recall from Chap. 1 that policies refer to what should be done and mechanisms refer to how it should be done. A protection mechanism provides a set of tools that can be

used to design or specify a wide array of protection policies, whereas a policy gives assignment of the access rights of users to various resources. The separation of policies and mechanisms enhances design flexibility.

Protection in an operating system refers to mechanisms that control user access to system resources, whereas policies decide which user can have access to what resources. Policies can change with time and applications. Thus, a protection scheme must be amenable to a wide variety of policies to enforce security in computer systems. In this chapter, we will mainly be concerned with the design of protection mechanisms in operating systems.

**PROTECTION VS. SECURITY.** Hydra [39] designers make a distinction between protection and security. According to them, *protection is a mechanism and security is a policy.* Protection deals with mechanisms to build secure systems and security deals with policy issues that use protection mechanisms to build secure systems.

### 14.2.4 Protection Domain

The protection domain of a process specifies the resources that it can access and the types of operations that the process can perform on the resources. In a typical computation, the control moves through a series of processes. To enforce security in the system, it is good policy to allow a process to access only those resources that it requires to complete its task. This eliminates the possibility of a process breaching security maliciously or unintentionally (such as by a software bug) and increases accountability.

The concept of protection domain of a process enables us to achieve the policy of limiting a process's access to only needed resources. Every process executes in its protection domain and protection domain is switched appropriately whenever control jumps from a process to another process.

### 14.2.5 Design Principles for Secure Systems

Saltzer and Schroeder [34] gave the following principles for designing a secure computer system:

**Economy.** A protection mechanism should be economical to develop and use. Its inclusion in a system should not result in substantial cost or overhead to the system. One easy way to achieve economy is to keep the design as simple and small as possible [34].

**Complete Mediation.** The design of a completely secure system requires that every request to access an object be checked for the authority to do so.

**Open Design.** A protection mechanism should not stake its integrity on the ignorance of potential attackers concerning the protection mechanism itself (i.e., the underlying principle used to achieve the security). A protection mechanism should work even if its underlying principles are known to an attacker.

**Separation of Privileges.** A protection mechanism that requires two keys to unlock a lock (or gain access to a protected object) is more robust and flexible than one

that allows only a single key to unlock a lock. In computer systems, the presence of two keys may mean satisfying two independent conditions before an access is allowed.

**Least Privilege.** A subject should be given the bare minimum access rights that are sufficient for it to complete its task. If the requirement of a subject changes, the subject should acquire it by switching the domain. (Recall that a domain defines access rights of a subject to various objects.)

**Least Common Mechanism.** According to this principle, the portion of a mechanism that is common to more than one user should be minimized, as any coupling among users (through shared mechanisms and variables) represents a potential information path between users and is thus a potential threat to their security.

**Acceptability.** A protection mechanism must be simple to use. A complex and obscure protection mechanism will deter users from using it.

**Fail-Safe Defaults.** Default case should mean lack of access (because it is safer this way). If a design or implementation mistake is responsible for denial of an access, it will eventually be discovered and be fixed. However, the opposite is not true.

## 14.3 THE ACCESS MATRIX MODEL

A model of protection abstracts the essential features of a protection system so that various properties of it can be proven. A protection system consists of mechanisms to control user access to system resources or to control information flow in the system. In this section, we study the most fundamental model of protection—the access matrix model—in computer systems. Advanced models of protection are covered in Sec. 14.6. A survey of models for protection in computer systems can be found in a paper by Landwehr [23].

The access matrix model was first proposed by Lampson [21]. It was further enhanced and refined by Graham and Denning [18] and Harrison et al. [19]. The description of the access matrix model in this section is based on the work of Harrison et al. [19]. This model consists of the following three components:

**Current Objects.** Current objects are a finite set of entities to which access is to be controlled. The set is denoted by '$O$'. A typical example of an object is a file.

**Current Subjects.** Current subjects are a finite set of entities that access current objects. The set is denoted by '$S$'. A typical example of a subject is a process. Note that $S \subseteq O$. That is, subjects can be treated as objects and can be accessed like an object by other subjects.

**Generic Rights.** A finite set of generic rights, $R = \{r_1, r_2, r_3, ..., r_m\}$, gives various access rights that subjects can have to objects. Typical examples of such rights are read, write, execute, own, delete, etc.

**THE PROTECTION STATE OF A SYSTEM.** The *protection state* of a system is represented by a triplet $(S, O, P)$, where $S$ is the set of current subjects, $O$ is the set of current objects, and $P$ is a matrix, called the *access matrix*, with a row for every current subject and a column for every current object. A schematic diagram of an access matrix

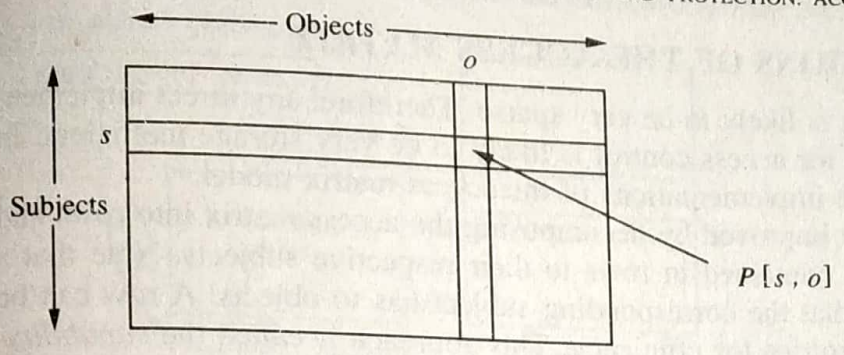


**FIGURE 14.1**
A schematic of an access matrix.

is shown in Fig. 14.1. Note that the access matrix $P$ itself is a protected object. Let variables $s$ and $o$ denote a subject and an object, respectively. Entry $P[s, o]$ is a subset of $R$, the generic rights, and denotes the access rights which subject $s$ has to object $o$.

**ENFORCING A SECURITY POLICY.** A security policy is enforced by validating every user access for appropriate access rights. Every object has a monitor that validates all accesses to that object in the following manner.

1. A subject $s$ requests an access $\alpha$ to object $o$.
2. The protection system presents triplet $(s, \alpha, o)$ to the monitor of $o$.
3. The monitor looks into the access rights of $s$ to $o$. If $\alpha \in P[s, o]$, then the access is permitted; Else it is denied.

**Example 14.1.** Figure 14.2 illustrates an access matrix that represents the protection state of a system with three subjects, $s_1$, $s_2$, $s_3$, and five objects, $o_1$, $o_2$, $s_1$, $s_2$, $s_3$. In this protection state, subject $s_1$ can read and write object $o_1$, delete $o_2$, send mail to $s_2$, and receive mail from $s_3$. Subject $s_3$ owns $o_1$ and can read and write $o_2$.

The access matrix model of a protection system is very popular because of its simplicity, elegant structure, and amenability to various implementations. We next discuss implementations of the access matrix model.

| | $o_1$ | $o_2$ | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|---|---|
| $s_1$ | *read, write* | *own, delete* | *own* | *sendmail* | *recmail* |
| $s_2$ | *execute* | *copy* | *recmail* | *own* | *block, wakeup* |
| $s_3$ | *own* | *read, write* | *sendmail* | *block, wakeup* | *own* |

**FIGURE 14.2**
An access matrix representing a protection state.

## 14.4 IMPLEMENTATIONS OF THE ACCESS MATRIX

Note that the access matrix is likely to be very sparse. Therefore, any direct implementation of the access matrix for access control is likely to be very storage inefficient. In this section, we study three implementations of the access matrix model.

The efficiency can be improved by decomposing the access matrix into rows and assigning the access rights contained in rows to their respective subjects. Note that a row denotes access rights that the corresponding subject has to objects. A row can be collapsed by deleting null entries for efficiency. This approach is called the *capability-based* method. An orthogonal approach is to decompose the access control matrix by columns and assign the columns to their respective objects. Note that a column denotes access rights of various subjects to the object. A column can be collapsed by deleting null entries for higher efficiency. This technique is called the *access control list* method. The third approach, called the *lock-key* method, is a combination of the first two approaches.

### 14.4.1 Capabilities

The capability based method corresponds to the row-wise decomposition of the access matrix. Each subject $s$ is assigned a list of tuples $(o, P[s, o])$ for all objects $o$ that it is allowed to access. The tuples are referred to as *capabilities*. If subject $s$ possesses a capability $(o, P[s, o])$, then it is authorized to access object $o$ in manners specified in $P[s, o]$. Possession of a capability by a user is treated as prima facie evidence that the user has authority to access the object in the ways specified in the capability. The list of capabilities assigned to subject $s$ corresponds to access rights contained in the row for subject $s$ in the access matrix. At any time, a subject is authorized to access only those objects for which it has capabilities. Clearly, one must not be able to forge capabilities.

A schematic view of a capability is shown in Fig. 14.3. A capability has two fields. First, an object descriptor, which is an identifier for the object and second, access rights, which indicate the allowed access rights to the object. The object descriptor can very well be the address of the corresponding objects and therefore, aside from providing protection, capabilities can also be used as an addressing mechanism by the system. The main advantage of using a capability as an addressing mechanism is that it provides an address that is context independent. That is, it provides an absolute address [14]. However, when a capability is used as an addressing mechanism, the system must allow the embedding of capabilities in user programs and data structures, as a capability will be a part of the address.

**CAPABILITY-BASED ADDRESSING.** Capability-based addressing is illustrated in Fig. 14.4. A user program issues a request to access a word within an object. The

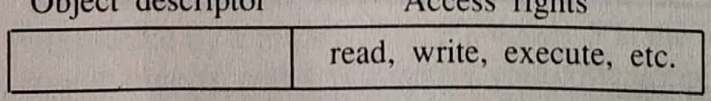


**FIGURE 14.3**
A schematic view of a capability.

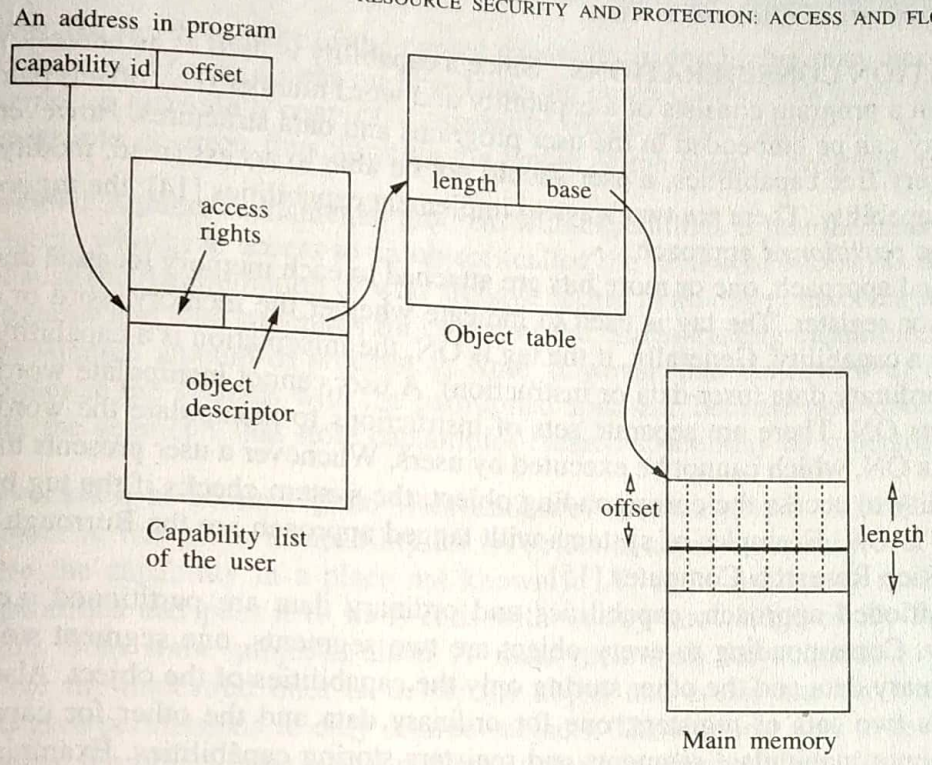


**FIGURE 14.4**
An illustration of capability-based addressing.

address of the request contains the capability ID of the object (which tells what object in the main memory is to be accessed) and an offset within the object (which gives the relative location of the word in the object to be accessed). The system uses the capability ID to search the capability list of the user to locate the capability that contains the allowed access rights and an object descriptor. The system checks if the requested access is permitted by checking the access rights in the capability. The object descriptor is used to search the object table to locate the entry for the object. The entry consists of the base address of the object in main memory and the length of the object. The system adds the base address to the offset in the request to determine the exact memory location of the accessed word.

Capability-based addressing has two salient features, relocatability and sharing. An object can be relocated anywhere in the main memory without making any change to the capabilities that refer to it. (For every relocation, only the base field of the object needs be changed in the object table.) Sharing is made easy as several programs can share the same object (program or data) with different names (object descriptors) for the object. Note that this type of sharing and relocatability is achieved by introducing a level of indirection (via the object table) in addressing the objects—the object descriptor in a capability contains the address of the address of the object.

If there is a separate object table for each process or subject, then the resolution of an object descriptor is done in the context of a process. If there is a global object table, then the resolution of an object descriptor is done in a single global context.

**IMPLEMENTATION CONSIDERATIONS.** Since a capability is used as an address, a typical address in a program consists of a capability and word number (i.e., offset) pair, and the capability can be embedded in the user programs and data structures. However, to maintain forgery-free capabilities, a user should not be able to access (read, modify, or construct) a capability. There are two ways to implement capabilities [14]: the *tagged* approach and the *partitioned* approach.

In the tagged approach, one or more bits are attached to each memory location and to every processor register. The tag is used to indicate whether the memory word or a register contains a capability. Generally, if the tag is ON, the information is a capability; otherwise, it is ordinary data (user data or instruction). A user cannot manipulate words with their tag bits ON. There are separate sets of instructions to manipulate the words with their tag bits ON, which cannot be executed by users. Whenever a user presents the system a capability to access the corresponding object, the system checks if the tag bit of the capability is ON. Examples of systems with tagged approach are the Burrough's B6700 and the Rice Research Computer [15].

In the partitioned approach, capabilities and ordinary data are partitioned, i.e., stored separately. Corresponding to every object are two segments, one segment storing only the ordinary data and the other storing only the capabilities of the object. Also, the processor has two sets of registers, one for ordinary data and the other for capabilities. Users cannot manipulate segments and registers storing capabilities. Examples of systems with the partitioned approach are the Chicago Magic Number Machine [13] and Plessey System 250 [12].

**ADVANTAGES OF CAPABILITIES.** The capability-based protection system has three main advantages [34]: efficiency, simplicity, and flexibility. It is efficient because the validity of an access can be easily tested; an access by a subject is implicitly valid if it has the capability. It is simple due to the natural correspondence between the structural properties of capabilities and the semantic properties of addressing variables. It is flexible because a capability system allows users to define certain parameters. For example, a users can decide which of his addresses contain capabilities. Also, a user can define any data structure with an arbitrary pattern of access authorization.

## DRAWBACK OF CAPABILITIES

**Control of propagation.** When a subject passes a copy of a capability for an object to another subject, the second subject can pass copies of the capability to many other subjects without the first subject's knowledge. In some applications, it may be desirable (to induce unrestricted sharing), while in other applications, it may be necessary to control the propagation of capabilities for the purpose of accountability as well as security.

The propagation of a capability can be controlled by adding a bit, called the *copy* bit, in a capability that indicates whether the holder of the capability has permission to copy (and distribute) the capability. The propagation of a capability can be prevented by setting this bit to OFF when providing a copy of the capability to other users. Another way to limit the propagation is to use a depth counter [34]. A depth counter is attached to each capability (whose initial value is one). Every time a copy of a capability is

made, the depth counter of the copied capability is one higher than that of the original capability. There is a limit on how large the depth counter can grow (say, four). Any attempt to generate a copy of a capability whose depth counter has reached the limit results into an error, thus, limiting the length of the chain a capability can propagate.

**Review.** Another fundamental problem with capabilities is that the determination of all subjects who have access to an object (called the *review* of access) is difficult. This is because the determination of who all have access to an object involves searching all the programs and data structures for copies of the corresponding capabilities. This requires a substantial amount of processing. Note, however, that the review of access becomes simpler in the systems with the partitioned approach because now one needs to search only the segments that store capabilities (search space may be substantially reduced).

**Revocation of access rights.** Revocation of access rights is difficult because once a subject $X$ has given a capability for an object to some other subject $Y$, subject $Y$ can store the capability in a place not known to $X$, or $Y$ itself may make copies of the capabilities and pass it to its friends without any knowledge of $X$. To revoke access rights from some subjects, either $X$ must review all the accesses to that object and delete the undesired ones or delete the object and create another copy of the object and give permissions to only desired subjects. The simplest way to revoke access is to destroy the object, which will prevent all the undesired subjects from accessing it. (Of course, the accesses by other users will also be denied).

**Garbage collection.** When all the capabilities for an object disappear from the system, the object is left inaccessible to users and becomes garbage. This is called the *garbage collection* or the *lost object* problem. One solution to this problem is to have the creator of an object or the system keep a count of the number of copies of each capability and recover the space taken by an object when its capability count becomes zero.

### 14.4.2 The Access Control List Method

The access control list method corresponds to the column-wise decomposition of the access matrix. Each object $o$ is assigned a list of pairs $(s, P[s, o])$ for all subjects $s$ that are allowed to access the object. Note that the set $P[s, o]$ denotes the access rights that subject $s$ has to object $o$. The access list assigned to object $o$ corresponds to all access rights contained in the column for object $o$ in the access matrix. A schematic diagram of an access control list is shown in Figure 14.5.

When a subject $s$ requests access $\alpha$ to object $o$, it is executed in the following manner:

- The system searches the access control list of $o$ to find out if an entry $(s, \Phi)$ exists for subject $s$.
- If an entry $(s, \Phi)$ exists for subject $s$, then the system checks to see if the requested access is permitted (i.e., $\alpha \in \Phi$).
- If the requested access is permitted, then the request is executed. Otherwise, an appropriate exception is raised.

| Subjects | Access rights |
|---|---|
| Smith | read, write, execute |
| Jones | read |
| Lee | write |
| Grant | execute |
| ⋮ | ⋮ |
| White | read, write |

**FIGURE 14.5**
A schematic of an access control list.

Clearly, the execution efficiency of the access control list method is poor because an access control list must be searched for every access to a protected object.

Major features of the access control list method include:

**Easy Revocation.** Revocation of access rights from a subject is very simple, fast, and efficient. It can be achieved by simply removing the subject's entry from the object's access control list.

**Easy Review of an Access.** It can be easily determined what subjects have access rights to an object by directly examining the access control list of that object. However, it is difficult to determine what objects a subject has access to.

**IMPLEMENTATION CONSIDERATIONS.** There are two main issues in the implementation of the access control list method:

**Efficiency of execution.** Since the access control list need be searched for every access to a protected object, it can be very slow.

**Efficiency of storage.** Since an access control list contains the names and access rights of all the subjects that can access the corresponding protected object, a list can require huge amounts of storage. However, note that the aggregate storage requirement is about the same as that required for capabilities. In an access control list, the total is taken across objects and in capabilities, the total is taken across users.

The first problem can be solved in the following way. When a subject makes its first access to an object, the access rights of the subject are fetched from the access control list of the object and stored in a place, called the *shadow register*, with the subject. This fetched information in the shadow register acts like a capability. Consequently, the subject can use that capability for all subsequent accesses to that object, dispensing with the need to search the access control list for every access. However, this method has negative implications for the revocability of access rights in the access control list method in that, merely revoking access rights from the access control lists will not revoke the access rights loaded in the shadow registers of processes. Of course, a simple way to get around this problem is to clear all shadow registers whenever an

access right is revoked from an access control list. Obviously, this will be followed by a large number of access control list searches to rebuild the shadow registers.

The second problem, large storage requirement, is caused by a large number of users as well as the numerous types of access rights. The large storage requirement due to a large number of users can be solved using the *protection group* technique discussed below. This technique limits the number of entries in an access control list by lumping users into groups.

Note that each entry in an access control list contains allowed access rights. If there are a large number of access rights, their coding and inclusion in an entry will be cumbersome. It will require large space and complex memory management. This problem can be solved by limiting the access rights to only a small number and assigning a bit in a vector for every access type.

**PROTECTION GROUPS.** The concept of protection group was introduced to reduce the overheads of storing (and searching) lengthy access control lists [34]. Subjects (users) are divided into protection groups and the access control list consists of the names of groups along with their access rights. Thus, the number of entries in an access control list is limited by the number of protection groups, and therefore, high efficiency is achieved. However, the granularity at which access rights can be assigned becomes coarse—all subjects in a protection group have identical access rights to the object. To access an object, a subject gives its protection group and requested access to the system.

**AUTHORITY TO CHANGE AN ACCESS CONTROL LIST.** The authority to change the access control list raises the question of who can modify the access control information (contained in an access control list). Note that in a capability-based system, this issue is rather vague—any process which has a capability may make a copy and give it to any other process. The access control list method, however, provides a more precise and structured control over the propagation of access rights.

The access control list method provides two ways to control propagation of access rights [34]: *self control* and *hierarchical control*. In the self control policy, the owner process of an object has a special access right by which it can modify the access control list of the object (i.e., can revoke or grant access rights to the object). Generally, the owner is the creator process of an object. A drawback of the self control method is that the control is centralized to one process.

In the hierarchical control, when a new object is created, its owner specifies a set of other processes which have the right to modify the access control list of the new object. Processes are arranged in a hierarchy and a process can modify the access control list associated with all the processes below it in the hierarchy.

### 14.4.3 The Lock-Key Method

The lock-key method is a hybrid of the capability-based method and the access control list method [7], [25]. This method has features of both these methods.

In the lock-key method, every subject has a capability list that contains tuples of the form $(O, k)$, indicating that the subject can access object $O$ using key $k$. Every