**Goals of Protection**

As computer systems have become more sophisticated and pervasive in their applications, the need to protect their integrity has also grown. Protection was originally conceived as an adjunct to multiprogramming operating systems, so that untrustworthy users might safely share a common logical name space, such as a directory of files, or share a common physical name space, such as memory. Modern protection concepts have evolved to increase the reliability of any complex system that makes use of shared resources.

We need to provide protection for several reasons. The most obvious is the need to prevent the mischievous, intentional violation of an access restriction by a user. Of more general importance, however, is the need to ensure that each program component active in a system uses system resources only in ways consistent with stated policies. This requirement is an absolute one for a reliable system.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a malfunctioning subsystem. Also, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.

The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the management of a system. Still others are defined by the individual users to protect their own files and programs. A protection system must have the flexibility to enforce a variety of policies.

Policies for resource use may vary by application, and they may change over time. For these reasons, protection is no longer the concern solely of the designer of an operating system. The application programmer needs to use protection mechanisms as well, to guard resources created and supported by an application subsystem against misuse. In this chapter, we describe the protection mechanisms the operating system should provide, but application designers can use them as well in designing their own protection software.

Note that mechanisms are distinct from policies. Mechanisms determine how something will be done; policies decide what will be done. The separation of policy and mechanism is important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, every change in policy would require a change in the underlying mechanism. Using general mechanisms enables us to avoid such a situation.

**Principles of Protection**

Frequently, a guiding principle can be used throughout a project, such as the design of an operating system. Following this principle simplifies design decisions and keeps the system consistent and easy to understand. A key, time-tested guiding principle for protection is the principle of least privilege. It dictates that programs, users, and even systems be given just enough privileges to perform their tasks.

Consider the analogy of a security guard with a passkey. If this key allows the guard into just the public areas that she guards, then misuse of the key will result in minimal damage. If, however, the passkey allows access to all areas, then damage from its being lost, stolen, misused, copied, or otherwise compromised will be much greater.

An operating system following the principle of least privilege implements its features, programs, system calls, and data structures so that failure or compromise of a component does the minimum damage and allows the minimum damage to be done. The overflow of a buffer in a system daemon might cause the daemon process to fail, for example, but should not allow the execution of code from the daemon process’s stack that would enable a remote user to gain maximum privileges and access to the entire system (as happens too often today).

Such an operating system also provides system calls and services that allow applications to be written with fine-grained access controls. It provides mechanisms to enable privileges when they are needed and to disable them when they are not needed. Also beneficial is the creation of audit trails for all privileged function access. The audit trail allows the programmer, system administrator, or law-enforcement officer to trace all protection and security activities on the system.

Managing users with the principle of least privilege entails creating a separate account for each user, with just the privileges that the user needs. An operator who needs to mount tapes and back up files on the system has access to just those commands and files needed to accomplish the job. Some systems implement role-based access control (RBAC) to provide this functionality.

Computers implemented in a computing facility under the principle of least privilege can be limited to running specific services, accessing specific remote hosts via specific services, and doing so during specific times. Typically, these restrictions are implemented through enabling or disabling each service and through using access control lists.

The principle of least privilege can help produce a more secure computing environment. Unfortunately, it frequently does not. For example, Windows 2000 has a complex protection scheme at its core and yet has many security holes. By comparison, Solaris is considered relatively secure, even though it is a variant of UNIX, which historically was designed with little protection in mind. One reason for the difference may be that Windows 2000 has more lines of code and more services than Solaris and thus has more to secure and protect. Another reason could be that the protection scheme in Windows 2000 is incomplete or protects the wrong aspects of the operating system, leaving other areas vulnerable.

**Domain of Protection**

A computer system is a collection of processes and objects. By objects, we mean both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives) and software objects (such as files, programs, and semaphores). Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations. Objects are essentially abstract data types.

The operations that are possible may depend on the object. For example, on a CPU, we can only execute. Memory segments can be read and written, whereas a CD-ROM or DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.

A process should be allowed to access only those resources for which it has authorization. Furthermore, at any time, a process should be able to access only those resources that it currently requires to complete its task. This second requirement, commonly referred to as the need-to-know principle, is useful in limiting the amount of damage a faulty process can cause in the system. For example, when process p invokes procedure A(), the procedure should be allowed to access only its own variables and the formal parameters passed to it; it should not be able to access all the variables of process p. Similarly, consider the case in which process p invokes a compiler to compile a particular file. The compiler should not be able to access files arbitrarily but should have access only to a well-defined subset of files (such as the source file, listing file, and so on) related to the file to be compiled. Conversely, the compiler may have private files used for accounting or optimization purposes that process p should not be able to access. The need-to-know principle is similar to the principle of least privilege in that the goals of protection are to minimize the risks of possible security violations.

**Domain Structure**

To facilitate the scheme just described, a process operates within a protection domain, which specifies the resources that the process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object. The ability to execute an operation on an object is an access right. A domain is a collection of access rights, each of which is an ordered pair <object-name, right-set> . For example, if domain D has the access right <file F, {read, right}> , then a process executing in domain D can both read and write file F. It cannot, however, perform any other operation on that object.

Domains may share access rights. For example, in Figure 1.86, we have three domains: D1, D2, and D3. The access right <O4, {print}> is shared by D2 and D3, implying that a process executing in either of these two domains can print object O4. Note that a process must be executing in domain D1 to read and write object O1, while only processes in domain D3 may execute object O1.

The association between a process and a domain may be either static, if the set of resources available to the process is fixed throughout the process’s lifetime, or dynamic. As might be expected, establishing dynamic protection domains is more complicated than establishing static protection domains.

If the association between processes and domains is fixed, and we want to adhere to the need-to-know principle, then a mechanism must be available to change the content of a domain. The reason stems from the fact that a process may execute in two different phases and may, for example, need read access in one phase and write access in another. If a domain is static, we must define the domain to include both read and write access. However, this arrangement provides more rights than are needed in each of the two phases, since we have read access in the phase where we need only write access, and vice versa.

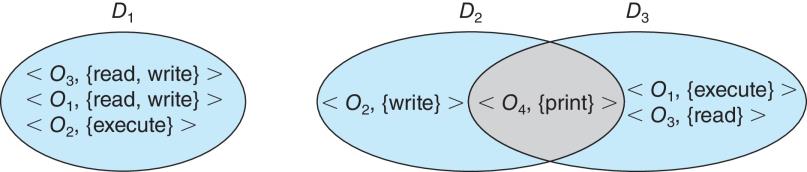


Figure 1.86 System with three protection domains

Thus, the need-to-know principle is violated. We must allow the contents of a domain to be modified so that the domain always reflects the minimum necessary access rights.

If the association is dynamic, a mechanism is available to allow domain switching, enabling the process to switch from one domain to another. We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.

A domain can be realized in a variety of ways:

• Each user may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed—generally when one user logs out and another user logs in.

• Each process may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.

• Each procedure may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

Consider the standard dual-mode (monitor–user mode) model of operating-system execution. When a process executes in monitor mode, it can execute privileged instructions and thus gain complete control of the computer system. In contrast, when a process executes in user mode, it can invoke only nonprivileged instructions. Consequently, it can execute only within its predefined memory space. These two modes protect the operating system (executing in monitor domain) from the user processes (executing in user domain). In a multiprogrammed operating system, two protection domains are insufficient, since users also want to be protected from one another. Therefore, a more elaborate scheme is needed. We illustrate such a scheme by examining two influential operating systems—UNIX and MULTICS —to see how they implement these concepts.

**An Example: UNIX**

In the UNIX operating system, a domain is associated with the user. Switching the domain corresponds to changing the user identification temporarily. This change is accomplished through the file system as follows. An owner identification and a domain bit (known as the setuid bit) are associated with each file. When the setuid bit is on, and a user executes that file, the userID is set to that of the owner of the file. When the bit is off, however, the userID does not change. For example, when a user A (that is, a user with userID = A) starts executing a file owned by B, whose associated domain bit is off, the userID of the process is set to A. When the setuid bit is on, the userID is set tothat of the owner of the file: B. When the process exits, this temporary userID change ends.

Other methods are used to change domains in operating systems in which userIDs are used for domain definition, because almost all systems need to provide such a mechanism. This mechanism is used when an otherwise privileged facility needs to be made available to the general user population. For instance, it might be desirable to allow users to access a network without letting them write their own networking programs. In such a case, on a UNIX system, the setuid bit on a networking program would be set, causing the userID to change when the program was run. The userID would change to that of a user with network access privilege (such as root, the most powerful userID). One problem with this method is that if a user manages to create a file with userID root and with its setuid bit on, that user can become root and do anything and everything on the system. The setuid mechanism is discussed further in Appendix A.

An alternative to this method used in some other operating systems is to place privileged programs in a special directory. The operating system is designed to change the userID of any program run from this directory, either to the equivalent of root or to the userID of the owner of the directory. This eliminates one security problem, which occurs when intruders create programs to manipulate the setuid feature and hide the programs in the system for later use (using obscure file or directory names). This method is less flexible than that used in UNIX, however.

Even more restrictive, and thus more protective, are systems that simply do not allow a change of userID. In these instances, special techniques must be used to allow users access to privileged facilities. For instance, a daemon process may be started at boot time and run as a special userID. Users then run a separate program, which sends requests to this process whenever they need to use the facility. This method is used by the TOPS-20 operating system.

In any of these systems, great care must be taken in writing privileged programs. Any oversight can result in a total lack of protection on the system. Generally, these programs are the first to be attacked by people trying to break into a system. Unfortunately, the attackers are frequently successful. For example, security has been breached on many UNIX systems because of the setuid feature.

**Access Matrix**

Our general model of protection can be viewed abstractly as a matrix, called an access matrix. The rows of the access matrix represent domains, and the columns represent objects. Each entry in the matrix consists of a set of access rights. Because the column defines objects explicitly, we can omit the object name from the access right. The entry access(i,j) defines the set of operations that a process executing in domain Di can invoke on object Oj .

To illustrate these concepts, we consider the access matrix shown in Figure 1.87. There are four domains and four objects—three files (F1, F2, F3) and one laser printer. A process executing in domain D1 can read files F1 and F3. A process executing in domain D4 has the same privileges as one executing in domain D1; but in addition, it can also write onto files F1 and F3. The laser printer can be accessed only by a process executing in domain D2.

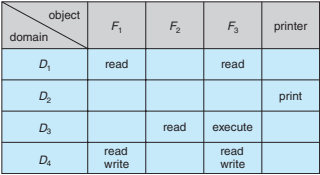


Figure 1.87 Access matrix.

The access-matrix scheme provides us with the mechanism for specifying a variety of policies. The mechanism consists of implementing the access matrix and ensuring that the semantic properties we have outlined hold. More specifically, we must ensure that a process executing in domain Di can access only those objects specified in row i, and then only as allowed by the access-matrix entries.

The access matrix can implement policy decisions concerning protection. The policy decisions involve which rights should be included in the (i, j)th entry. We must also decide the domain in which each process executes. This last policy is usually decided by the operating system.

The users normally decide the contents of the access-matrix entries. When a user creates a new object Oj , the column Oj is added to the access matrix with the appropriate initialization entries, as dictated by the creator. The user may decide to enter some rights in some entries in column j and other rights in other entries, as needed.

The access matrix provides an appropriate mechanism for defining and implementing strict control for both static and dynamic association between processes and domains. When we switch a process from one domain to another, we are executing an operation (switch) on an object (the domain). We can control domain switching by including domains among the objects of the access matrix. Similarly, when we change the content of the access matrix, we are performing an operation on an object: the access matrix. Again, we can control these changes by including the access matrix itself as an object. Actually, since each entry in the access matrix can be modified individually, we must consider each entry in the access matrix as an object to be protected. Now, we need to consider only the operations possible on these new objects (domains and the access matrix) and decide how we want processes to be able to execute these operations.

Processes should be able to switch from one domain to another. Switching from domain Di to domain Dj is allowed if and only if the access right switch ∈ access(i, j). Thus, in Figure 1.88, a process executing in domain D2 can switch to domain D3 or to domain D4. A process in domain D4 can switch to D1, and one in domain D1 can switch to D2.

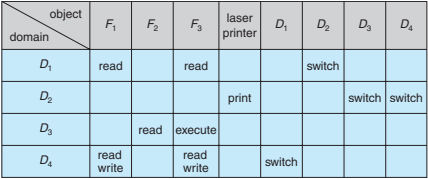


Figure 1.88 Access matrix of Figure 14.3 with domains as objects

Allowing controlled change in the contents of the access-matrix entries requires three additional operations: copy, owner, and control. We examine these operations next.

The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (\*) appended to the access right. The copy right allows the access right to be copied only within the column (that is, for the object) for which the right is defined. For example, in Figure 1.89(a), a process executing in domain D2 can copy the read operation into any entry associated with file F2. Hence, the access matrix of Figure 1.89(a) can be modified to the access matrix shown in Figure 1.89(b).

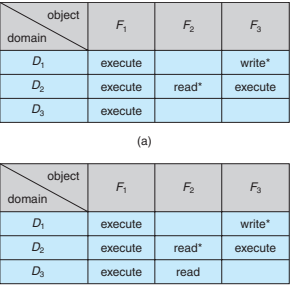


Figure 1.89 Access matrix with copy rights.

This scheme has two additional variants:

1. A right is copied from access(i, j) to access(k, j); it is then removed from access(i, j). This action is a of a right, rather than a copy.

2. Propagation of the copy right may be limited. That is, when the right R∗ is copied from access(i, j) to access(k, j), only the right R (not R∗) is created. A process executing in domain Dk cannot further copy the right R.

A system may select only one of these three copy rights, or it may provide all three by identifying them as separate rights: copy, transfer, and limited copy.

We also need a mechanism to allow addition of new rights and removal of some rights. The owner right controls these operations. If access(i, j) includes the owner right, then a process executing in domain Di can add and remove any right in any entry in column j. For example, in Figure 1.90(a), domain D1 is the owner of F1 and thus can add and delete any valid right in column F1. Similarly, domain D2 is the owner of F2 and F3 and thus can add and remove any valid right within these two columns. Thus, the access matrix of Figure 1.90(a) can be modified to the access matrix shown in Figure 1.90(b).

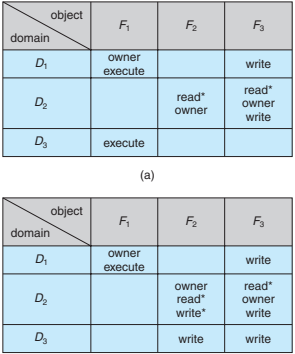


Figure 1.90 Access matrix with owner rights.

The copy and owner rights allow a process to change the entries in a column. A mechanism is also needed to change the entries in a row. The control right is applicable only to domain objects. If access(i, j) includes the control right, then a process executing in domain Di can remove any access right from row j. For example, suppose that, in Figure 1.88, we include the control right in access(D2, D4). Then, a process executing in domain D2 could modify domain D4, as shown in Figure 1.91.

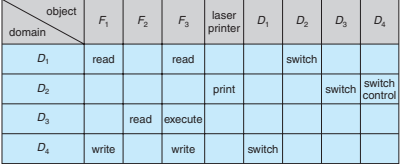


Figure 1.91 Modified access matrix of Figure 1.88.

The copy and owner rights provide us with a mechanism to limit the propagation of access rights. However, they do not give us the appropriate tools for preventing the propagation (or disclosure) of information. The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called the confinement problem. This problem is in general unsolvable (see the bibliographical notes at the end of the chapter).

These operations on the domains and the access matrix are not in themselves important, but they illustrate the ability of the access-matrix model to allow us to implement and control dynamic protection requirements. New objects and new domains can be created dynamically and included in the access-matrix model. However, we have shown only that the basic mechanism exists. System designers and users must make the policy decisions concerning which domains are to have access to which objects in which ways.

**Implementation of the Access Matrix**

How can the access matrix be implemented effectively? In general, the matrix will be sparse; that is, most of the entries will be empty. Although data structure techniques are available for representing sparse matrices, they are not particularly useful for this application, because of the way in which the protection facility is used. Here, we first describe several methods of implementing the access matrix and then compare the methods.

**Global Table**

The simplest implementation of the access matrix is a global table consisting of a set of ordered triples <domain, object, rights-set>. Whenever an operation M is executed on an object Oj within domain Di , the global table is searched for a triple <Di, Oj, Rk>, with M ∈ Rk . If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.

This implementation suffers from several drawbacks. The table is usually large and thus cannot be kept in main memory, so additional I/O is needed. Virtual memory techniques are often used for managing this table. In addition, it is difficult to take advantage of special groupings of objects or domains. For example, if everyone can read a particular object, this object must have a separate entry in every domain.

**Access Lists for Objects**

Each column in the access matrix can be implemented as an access list for one object. Obviously, the empty entries can be discarded. The resulting list for each object consists of ordered pairs <domain, rights-set>, which define all domains with a nonempty set of access rights for that object.

This approach can be extended easily to define a list plus a default set of access rights. When an operation M on an object Oj is attempted in domain Di , we search the access list for object Oj , looking for an entry <Di, Rk> with M ∈ Rk . If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs. For efficiency, we may check the default set first and then search the access list.

**Capability Lists for Domains**

Rather than associating the columns of the access matrix with the objects as access lists, we can associate each row with its domain. A capability list for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its physical name or address, called a capability. To execute operation M on object Oj , the process executes the operation M, specifying the capability (or pointer) for object Oj as a parameter. Simple possession of the capability means that access is allowed.

The capability list is associated with a domain, but it is never directly accessible to a process executing in that domain. Rather, the capability list is itself a protected object, maintained by the operating system and accessed by the user only indirectly. Capability-based protection relies on the fact that the capabilities are never allowed to migrate into any address space directly accessible by a user process (where they could be modified). If all capabilities are secure, the object they protect is also secure against unauthorized access.

Capabilities were originally proposed as a kind of secure pointer, to meet the need for resource protection that was foreseen as multi-programmed computer systems came of age. The idea of an inherently protected pointer provides a foundation for protection that can be extended up to the application level.

To provide inherent protection, we must distinguish capabilities from other kinds of objects, and they must be interpreted by an abstract machine on which higher-level programs run. Capabilities are usually distinguished from other data in one of two ways:

• Each object has a tag to denote whether it is a capability or accessible data. The tags themselves must not be directly accessible by an application program. Hardware or firmware support may be used to enforce this restriction. Although only one bit is necessary to distinguish between capabilities and other objects, more bits are often used. This extension allows all objects to be tagged with their types by the hardware. Thus, the hardware can distinguish integers, floating-point numbers, pointers, Booleans, characters, instructions, capabilities, and uninitialized values by their tags.

• Alternatively, the address space associated with a program can be split into two parts. One part is accessible to the program and contains the program’s normal data and instructions. The other part, containing the capability list, is accessible only by the operating system. A segmented memory space is useful to support this approach.

**A Lock –Key Mechanism**

The lock–key scheme is a compromise between access lists and capability lists. Each object has a list of unique bit patterns, called locks. Similarly, each domain has a list of unique bit patterns, called keys. A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

As with capability lists, the list of keys for a domain must be managed by the operating system on behalf of the domain. Users are not allowed to examine or modify the list of keys (or locks) directly

**Comparison**

As you might expect, choosing a technique for implementing an access matrix involves various trade-offs. Using a global table is simple; however, the table can be quite large and often cannot take advantage of special groupings of objects or domains. Access lists correspond directly to the needs of users. When a user creates an object, he can specify which domains can access the object, as well as what operations are allowed. However, because access-right information for a particular domain is not localized, determining the set of access rights for each domain is difficult. In addition, every access to the object must be checked, requiring a search of the access list. In a large system with long access lists, this search can be time consuming.

Capability lists do not correspond directly to the needs of users, but they are useful for localizing information for a given process. The process attempting access must present a capability for that access. Then, the protection system needs only to verify that the capability is valid. Revocation of capabilities, however, may be inefficient.

The lock–key mechanism, as mentioned, is a compromise between access lists and capability lists. The mechanism can be both effective and flexible, depending on the length of the keys. The keys can be passed freely from domain to domain. In addition, access privileges can be effectively revoked by the simple technique of changing some of the locks associated with the object.

Most systems use a combination of access lists and capabilities. When a process first tries to access an object, the access list is searched. If access is denied, an exception condition occurs. Otherwise, a capability is created and attached to the process. Additional references use the capability to demonstrate swiftly that access is allowed. After the last access, the capability is destroyed. This strategy is used in the MULTICS system and in the CAL system.

As an example of how such a strategy works, consider a file system in which each file has an associated access list. When a process opens a file, the directory structure is searched to find the file, access permission is checked, and buffers are allocated. All this information is recorded in a new entry in a file table associated with the process. The operation returns an index into this table for the newly opened file. All operations on the file are made by specification of the index into the file table. The entry in the file table then points to the file and its buffers. When the file is closed, the file-table entry is deleted. Since the file table is maintained by the operating system, the user cannot accidentally corrupt it. Thus, the user can access only those files that have been opened. Since access is checked when the file is opened, protection is ensured. This strategy is used in the UNIX system.

The right to access must still be checked on each access, and the file-table entry has a capability only for the allowed operations. If a file is opened for reading, then a capability for read access is placed in the file-table entry. If an attempt is made to write onto the file, the system identifies this protection violation by comparing the requested operation with the capability in the file-table entry.

**Access Control**

We have described how access controls can be used on files within a file system. Each file and directory is assigned an owner, a group, or possibly a list of users, and for each of those entities, access-control information is assigned. A similar function can be added to other aspects of a computer system. A good example of this is found in Solaris 10.

Solaris 10 advances the protection available in the operating system by explicitly adding the principle of least privilege via role-based access control (RBAC). This facility revolves around privileges. A privilege is the right to execute a system call or to use an option within that system call (such as opening a file with write access). Privileges can be assigned to processes, limiting them to exactly the access they need to perform their work. Privileges and programs can also be assigned to roles. Users are assigned roles or can take roles based on passwords to the roles. In this way, a user can take a role that enables a privilege, allowing the user to run a program to accomplish a specific task, as depicted in Figure 1.92. This implementation of privileges decreases the security risk associated with superusers and setuid programs.

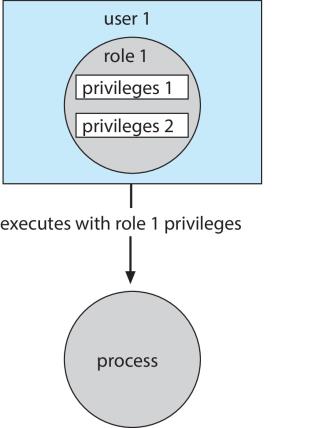


Figure 1.92 Role-based access control in Solaris 10