CHAPTER 13

Internal Protection Mechanisms

13.1 THE ACCESS CONTROL ENVIRONMENT
13.2 INSTRUCTION-LEVEL ACCESS CONTROL
13.3 HIGH-LEVEL ACCESS CONTROL
13.4 INFORMATION FLOW CONTROL

The user-authentication mechanisms presented in the previous chapter govern who can enter the system as a valid user. Once established and represented by a legally created process, the user may access data maintained by the system and use its services. To prevent any security violation, the system must provide a set of protection mechanisms and policies to govern the behavior of all processes with respect to the system resources and to each other. In this chapter, we examine the two major problem domains that must be supported by the system. The first addresses the problem of controlling the access by a process to various system resources including programs, data files, or hardware devices. The second is concerned with controlling the flow of information between different entities within a system, such as processes or files.

13.1 THE ACCESS CONTROL ENVIRONMENT

In the course of its execution, a process must access various resources, both hardware and software. Hardware resources include the CPU, its status and data registers, main memory, and a variety of possible secondary storage and communication devices. Software resources include program and data files, system tables, runtime data structures, and other processes, each of which may reside in registers, main memory, or secondary storage. The collection of all resources—hardware and software—accessible to a process at a given time is defined as the process’ current execution environment.

The current execution environment may be static, i.e., constant for the lifespan of a process, or it may vary dynamically during its execution. Dynamic execution environments are more difficult to implement but provide more flexibility in enforcing different protection policies. In particular, they can decrease the extent of damage a process may cause to itself or to other processes. Systems patterned after the “need-to-know” principle enforced by the military require that processes operate at all times in the smallest environment necessary to carry out the current tasks. Only a highly dynamic execution environment can satisfy this requirement.

We differentiate between two fundamental levels of access to the different resources of the current execution environment:

- **Instruction-level access**: Certain machine instructions use sensitive hardware resources, such as CPU registers, and must only be executed by authorized system
Section 13.2 Instruction-Level Access Control

processes. In addition, machine instructions that access memory (physical or virtual) must be prevented from specifying addresses that lie outside of the memory areas (partitions, pages, or segments) allocated to the process.

- **System-level access**: High-level abstract entities, e.g., files or logical devices, are accessed by issuing kernel calls to the OS. Controlling the use of such entities is the responsibility of the subsystems that handle the various kernel calls. In particular, the file system is responsible for enforcing legal accesses to files, as prescribed by the system policies.

Next, we explore the possible hardware and software mechanisms that support access control at the two different levels, and thus ensure that processes always remain confined to their respective execution environments.

### 13.2 INSTRUCTION-LEVEL ACCESS CONTROL

A machine instruction can access data in three types of memory: 1) hardware registers within the CPU; 2) hardware registers within I/O device controllers; and 3) main memory. We must prevent unauthorized access to data at each of the three levels to prevent different forms of security violations.

#### 13.2.1 Register and Input/output Protection

**Central Processing Unit Registers**

The CPU contains a number of hardware registers and flags. Some of these are read and written implicitly as part of instruction execution; these include various condition codes that reflect the result of the last operation, such as the sign bit or overflow. Others are accessed explicitly as operands of various machine instruction.

We can subdivide the directly accessible registers into two classes for the purposes of protection. The first includes registers that programs use to hold intermediate data or address values of the ongoing computation. These must be accessible to all programs. The second class includes registers that make up the CPU’s *internal state* and must only be accessed by trusted system programs to prevent a variety of possible security violations. For example, the ability to explicitly set the program counter could give a process the ability to branch to any portion of memory. Altering the contents of timer registers used to generate time-sharing interrupts, or disabling interrupts altogether, would enable a process to illegally monopolize the CPU for its own purposes; this capability could result in unauthorized use of the computational resources or in the denial of service to others. In addition to the CPU registers, we also must prevent unauthorized access to I/O registers that comprise the interface to the various device controllers.

To support the necessary separation of user processes from authorized system processes, the instruction set of the processor is divided into two classes, *privileged* and *nonprivileged* instructions, respectively. The former include those that access and manipulate the CPU state, control I/O devices, or perform other sensitive operations, and may be executed only by authorized system programs. The remaining, nonprivileged instructions are used for general computation and are available to all programs.

The CPU hardware provides a special bit to differentiate between two distinct modes of execution: the *system* mode, also referred to as the *supervisor* or *kernel*
mode, and the user mode. When executing in the system mode, any instruction is valid. In the user mode, only the nonprivileged instructions may be executed; an attempt to invoke a privileged instruction causes an interrupt, which transfers control to the OS to take appropriate actions. Generally, the offending process is terminated and an error is reported.

The switching between different processor modes, which amounts to setting or resetting the special privilege bit in the CPU, must of course be controlled. Switching from the privileged system mode to the nonprivileged user mode can be done with a machine instruction that changes the mode bit. However, there can be no such instruction for switching from user to system mode; otherwise, any process could simply switch to the privileged mode, rendering the mechanism useless. The solution is a special instruction, generally called a supervisor or kernel call, that sets the bit to privileged mode and simultaneously transfers control to the OS. Thus, switching to system mode automatically takes control away from the invoking user process. When the OS decides to resume this process, it resets the bit to user mode prior to transferring control to it.

Some systems distinguish more than two modes. For example, the long-lasting VAX-11 architecture distinguished four processor states—kernel, executive, supervisor, and user—each capable of using a different subset of the existing instructions. This permits a finer subdivision of the OS into separate levels for further protection.

Note that the availability of two or more execution modes provides for a dynamic execution environment at the process level. The current execution of the process changes whenever it switches between different execution modes in that the set of usable machine instructions grows or shrinks with the current mode.

Input/Output Devices

Chapter 11 surveyed the many communication and storage devices that are attached to computer systems. To communicate with such I/O devices, the CPU must read and write various registers provided by the device controllers. Access to these registers may be through either a memory-mapped interface or a set of special I/O instructions. In the first case, access to devices may be controlled through the same mechanisms provided for memory protection. That means, only authorized system processes, such as the device drivers, are allowed to read or write the portion of memory corresponding to the device registers.

When special instructions are used for I/O, they must be made privileged to guarantee that I/O devices will not be accessed by user processes directly. Instead, a process that wishes to use a device issues a kernel call that transfers control to the OS in system mode. The appropriate parts of the OS, generally the device driver and the scheduler, carry out the I/O operation on behalf of the invoking user process. These programs have been specially designed to work with the devices in the most efficient and safe manner and are permitted to use the privileged I/O instructions. When the operation is completed, control is returned to the invoking user mode. Thus, from the user point of view, the kernel call is simply a single high-level I/O instruction.

13.2.2 Main Memory Protection

Memory protection facilities are required to control process access to both its own instructions and data, and to those belonging to other processes, i.e., processes must be protected
from themselves and from others. This includes the OS, which needs protection from
damage or unauthorized execution by user processes.

There are two main problems to be solved. First, processes must be confined to
areas of main memory that have been assigned to them by the OS. Enforcement of such
confinement depends greatly on the type of memory management scheme implemented
in a given system. The second problem is concerned with the type of access by a process
to different areas of main memory. Ideally, each process should have its own set of rights
with respect to a given memory area.

**EXAMPLE: Read/Write/Execute Access**
The most common set of rights are read, write, and execute. If we represent the three
rights as a triple \( rwx \) of Boolean values, then eight possible combinations of rights may
be applied to a region of memory:

- 000 no access
- 100 read only
- 010 write only
- 110 read and write
- 001 execute only
- 101 read and execute
- 011 write and execute
- 111 unrestricted access

Note that write and execute (011) is of little practical value and is almost never
used. Write only (010) is also rare. A possible scenario for this latter combination is
when a designated memory area is being used to collect data from multiple processes,
each process must be able to deposit its own data but not be able to read the data written
by other processes.

Let us now distinguish between systems with and without virtual memory. For
each, we examine the two problems of confining processes into specified memory areas
and permitting different types of access to an area.

**Systems with Static Relocation**
In the absence of relocation registers, paging hardware, or segmentation facilities, pro-
grams address physical memory directly. To guarantee that processes remain within their
own partitions, each physical address must be checked for its validity; it must refer to
only the area assigned to the process. This can be enforced through: 1) **bounds registers**
that specify the upper and lower addresses of a contiguous memory area; 2) **base
and length indicators** containing the starting point and the size of a memory area; or
3) identification **locks and keys** associated with memory blocks and user processes.
Chapter 13  Internal Protection Mechanisms

FIGURE 13-1. Bounds registers for main memory.

The use of bounds registers is illustrated in Figure 13-1. Each time a reference to a physical address \( pa \) is made, the hardware performs the check:

\[
LR \leq pa \leq UR
\]

where \( LR \) and \( UR \) point to the first (lower) and the last (upper) memory cell assigned to the process. Only if the check is successful is the reference carried out; otherwise, an error condition is signaled and the process is aborted. Instead of the upper-bound register, a length register, \( L \), could be used. In this case, the check performed by the hardware is:

\[
LR \leq pa < LR + L.
\]

The use of identification keys requires that memory be divided into blocks, usually of equal size. A combination of \( n \) bits is associated with each block as a lock. Each process has a pattern of \( n \) bits, the key, as part of its process state. Upon each reference, the hardware compares the current key with the lock of the block to be accessed; only if a match is found is the process permitted to proceed with the operation.

The use of bounds or length registers is the simplest form of access control; it permits only an "all-or-nothing" type of protection to a given area, without being able to distinguish different types of access. The use of memory locks and keys offers potentially greater flexibility in that each lock could incorporate the type of access permitted to the corresponding memory block. Associating protection information directly with physical memory, however, is very restrictive when programs or data regions are to be shared. A much greater flexibility is possible in systems with virtual memory, where protection is associated with the logical name spaces rather than physical memory.

Systems with Relocation Registers

In systems that employ simple relocation registers to implement dynamic address binding, the problem of confining processes to assigned areas is similar to that of systems with static relocation. A given logical address \( la \) is first transformed into the corresponding physical address by adding to it the content of the relocation register \( RR \). The resulting physical address is then compared with the contents of the upper- and lower-bounds registers. The modified address mapping function has the following form:
Section 13.2  Instruction-Level Access Control 447

address_map(la) {
    pa = la + RR;
    if (!(LR <= pa && (pa <= UR))) error;
    return (pa);
}

Alternatively, a length indicator can be used instead of the upper-bound register UR. Both mechanisms prevent a process from accessing information outside its legal bounds. The problem of allowing different types of access to a given memory area, however, still persists. This can be solved by including the necessary access information with the process state. The address translation mechanisms perform the necessary checks as part of each memory access. This permits each process to have a different set of access rights for a given memory area.

Systems with Virtual Memory

A segment or a page of a virtual memory system is accessed indirectly via an entry in the page or segment table (or both). Such tables are private to a process; thus, by incorporating protection information into individual table entries, each process can have a different set of rights with respect to any segment or page (private or shared) accessible by that process. The necessary checks are performed at each address translation step as outlined below. Let us consider the general case of segmentation followed by paging, where each table occupies at most one page. Access information is specified on a per-segment basis and, therefore, it is kept in the segment table.

We extend the segment and page tables as shown in Figure 13-2. Each segment table entry, s, consists of the fields:

- **access** records the type of access (e.g., rwx) for the segment;
- **len** records the length of segment s (in bytes);
- **valid** indicates whether s is valid (i.e., exists);
- **resident** indicates whether the page table of s is currently in memory;
- **base** is a pointer to the beginning of the page table of segment s.

![ FIGURE 13-2. Memory protection through segment and page tables.](image)
Chapter 13  Internal Protection Mechanisms

Similarly, each page table entry, \( p \), consists of the fields:

- **resident** indicates whether the corresponding page \( p \) is currently in memory;
- **base** is a pointer to the beginning of \( p \).

The physical address computation is extended as follows to ensure that every virtual address \((s, p, w)\) remains confined within the legal memory space allocated to the process and that the type of access requested is allowed. We represent the type of access by an additional parameter, \( a \).

\[
\text{address_map}(s, p, w, a) = \begin{cases} 
\text{if (a not element of *(STR+s).access) invalid access type;} \\
\text{if (*(STR+s).valid == false) invalid segment number;} \\
\text{if (*(STR+s).resident == false) segment fault;} \\
\text{if (*(STR+s).len/\text{pg size} < p) invalid page number;} \\
\text{if (*((STR+s).base+p).resident == false) page fault;} \\
\text{if (*(STR+s).len \% pg size < w) invalid displacement;} \\
\text{return *((STR+s).base+p).base+w;}
\end{cases}
\]

The first if-statement checks whether the requested access type \( a \) is recorded as a valid access for this segment; if not, an error condition is raised, and the computation terminates. The second if-statement checks whether the segment is valid; if not, a different error condition is raised, and the computation again terminates. The third if-statement ensures the segment’s page table is resident before proceeding. The fourth if-statement determines the number of valid pages for the segment. This is done by dividing the segment length (kept in \(* (STR+s).len \) by the size of memory pages \( (pg\text{ size}) \). If the result is less than \( p \), the page number is illegal. Assuming a valid page number, the next if-statement makes sure the page is resident before proceeding. The last if-statement checks the validity of the displacement: The modulo operation \(* (STR+s).len \% pg\text{ size} \) computes the number of words the segment occupies on its last page. If the displacement is larger than this number, it would refer to data past the current extent of the segment.

Note that the last check is important to prevent illegal memory browsing. Since memory is always allocated in multiples of pages, the last page of every segment is, on average, only half occupied. The remaining portion does not belong to the segment, but, because it may contain sensitive information from previous computations, it should not be accessible to the process. If the check is not performed, an intruder could exploit this by systematically creating very small segments, say one byte each, and examining the contents of the allocated pages. Each segment would give the intruder nearly a full page of data left there by earlier computations. For a safer system, the contents of any page could be explicitly erased before the page is allocated to a process.

**CASE STUDY: PAGE ACCESS CHECKS IN WINDOWS 2000**

Windows 2000 uses paging without segmentation, so only security checks applicable to pages can be performed. The system distinguishes between several types of
errors, which *abort* the process, and several types of page faults, which are *handled transparently* by the system.

**Possible errors:**

- The system distinguished between *kernel-mode* and *user-mode* pages. Kernel mode pages can be accessed only when executing in kernel mode and are used for system-wide data structures and other memory areas. A user process can access these only by invoking the appropriate kernel functions.
- The system (Win32 API) differentiates between the following types of pages: no-access, read-only, read-write, execute-only, execute-read, and execute-read-write. (Some architectures do not provide the hardware support to differentiate between a read and an execute access; in such cases, only the first three types are supported.) An attempt to access a page in an incompatible mode results in an error.
- Any page in a process address space is in one of three states: free, reserved, or committed. This is recorded in the corresponding page table entry. A free page is one that has not been allocated to the process, i.e., it is invalid. Any attempt to access such a page generates an invalid page number error. Accessing reserved or committed pages may result in different types of page faults, as explained below.

**Possible page faults:**

- A committed page is a regular valid page that has been allocated to the process and may be accessed by the process, subject to any read/write/execute or kernel-mode restrictions. If the page is not currently resident in memory, a page fault causes the page to be loaded.
- A process may reserve a number of pages for future use. Such pages have valid entries in the page table but no actual pages have yet been allocated; this is done only when the page is accessed for the first time. A special type of page fault first allocates a page and modifies its page table entry from reserved to committed. Reserved pages are used for dynamically expanding data structures (e.g., the user stack) that must occupy a contiguous area of the virtual memory. Initially, each process gets a single page for its stack, but additional neighboring pages are reserved and allocated as the stack grows beyond the current page.
- The memory manager employs a special optimization technique, called *copy-on-write*. When a page must be duplicated, e.g., as a result of a *fork* command, the page is only marked as *copy-on-write*, but both processes point to the same page. This allows the same copy of the page to be shared by multiple processes, but only as long as the page is not being modified. As soon as a process writes into the page, a special page fault is triggered, which creates a separate copy of the page for the process.

**Sandboxing**

In most systems, a function invoked by a process automatically inherits all access privileges of the invoking process. In particular, it has access to the entire virtual memory of the process. If the function cannot be trusted, as is frequently the case when an applet or service program is downloaded through the Internet or borrowed from another
user, such unrestricted access is undesirable. The imported function could be a “Trojan horse” or simply contain errors that could be harmful to the invoking process. Similar concerns arise when programs may be sent to and started on a remote machine, or when a mobile agent hops to a different machine and continues its execution in a new environment.

To limit the scope of the potential damage caused by untrusted programs, the system may provide support to restrict the privileges to only a small subset of those granted to the invoking process. The reduced-access environment is generally referred to as a sandbox. One of the most important aspects of sandboxing is to confine the execution of the program to a small area of memory—a memory sandbox. Any attempt to access data outside of this sandbox or to branch to a location outside of this sandbox causes a trap to the OS, and the application is terminated.

A more powerful variant of memory sandboxing is to provide two separate sandboxes for each program—one for its code and the other for its data. The program is allowed only to read and write in the data sandbox, and to only execute in (fetch instructions from) the code sandbox. This prevents the program from modifying itself at runtime and circumventing certain validity checks that may have been performed on the code prior to loading it.

Memory sandboxing can be implemented in a manner similar to paging. The virtual address space is divided into equal-size regions, each corresponding to a different sandbox. This is accomplished by dividing every virtual address into two components, \((b, w)\), where the number of bits allocated to \(b\) and \(w\) determine the number of sandboxes and the size of each sandbox, respectively. When a program is assigned to a given sandbox, \(s\), the system can easily check the validity of any address \((b, w)\) generated by the program by simply comparing \(b\) with \(s\). If the two values do not match, the address is outside of the assigned sandbox, and the program is terminated.

13.3 HIGH-LEVEL ACCESS CONTROL

13.3.1 The Access Matrix Model

Information about who can access what in a computer system can be represented in the form of an access matrix. Each entry of the matrix records the rights that a given entity (e.g., a process) has to access a resource. This basic concept was developed into a formal model that allows the systems designer or administrator to reason about and prove protection properties of the system (Harrison et al. 1976). Specifically, the access matrix model tries to answer the most basic question concerning system safety: In a given situation, can a particular subject gain access to a particular resource? For this purpose, we must formally define the possible operations that can be performed on the matrix and prove whether or not there exists a sequence of operations that can modify the matrix in such a way as to give the subject the necessary rights for the resource.

We will use the access matrix only informally as a framework to discuss different implementations of access control mechanisms and their respective advantages and drawbacks. The access matrix model has the following main components:

- **Resources**: Each resource \(R_i\) represents an entity (e.g., a file or a device) that is to be protected, i.e., accessed in a controlled manner. (The literature frequently
uses the term object instead of resource. We have chosen to use the latter to avoid any confusion with object-oriented programming and to maintain a consistent terminology throughout all chapters.)

- **Subjects**: Each subject \( S_j \) represents an active entity (e.g., a process) that can access resources.

- **Access rights**: Each right \( r_k \) represents an operation (e.g., read, write, or execute) that can be applied to a resource.

**EXAMPLE: Access Matrix**

Figure 13-3a shows the relationships between subjects, resources, and access rights in the form of a sample access matrix. Each column of the matrix represents a resource \( R_i \), and each row represents a subject \( S_j \). The intersection of the \( i \)th column with the \( j \)th row contains a (possibly empty) set of rights. These are the rights a subject \( S_j \) has with respect to the resource \( R_i \). The letters \( r \), \( w \), and \( x \) stand for read, write, and execute rights, respectively. For example, subject \( S_1 \) is allowed to read and write resource \( R_1 \), and read, write, and execute resource \( R_2 \). Subject \( S_2 \) can only execute \( R_2 \), but it has all three rights for the resources \( R_3 \) and \( R_4 \). (The access and capability lists will be explained shortly.)

<table>
<thead>
<tr>
<th></th>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>( R_3 )</th>
<th>( R_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( rw )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S_2 )</td>
<td></td>
<td>( x )</td>
<td>( rwx )</td>
<td></td>
</tr>
<tr>
<td>( S_3 )</td>
<td></td>
<td></td>
<td></td>
<td>( r )</td>
</tr>
</tbody>
</table>

(a)

**Access lists:**

- \( R_1 \): \((S_1, rw), (S_3, rwx)\)
- \( R_2 \): \((S_1, rw), (S_2, rwx), (S_3, x)\)
- \( R_3 \): \((S_2, rwx)\)
- \( R_4 \): \((S_2, rwx), (S_3, r)\)

**Capability lists:**

- \( S_1 \): \((R_1, rw), (R_2, rwx)\)
- \( S_2 \): \((R_2, x), (R_3, rwx), (R_4, rwx)\)
- \( S_3 \): \((R_2, rwx), (R_3, r)\)

(b) (c)

FIGURE 13-3. (a) access matrix; (b) implemented as access lists; and (c) implemented as capability lists.

Having defined the basic structure of the access matrix, we now consider the operations that can be applied to use and manipulate it. In particular, it is necessary to add or remove the rows or columns of the matrix, and to enter or remove rights from the entries of the matrix. The techniques for performing these operations depend greatly on how the access matrix is implemented. The obvious way—storing the matrix as an actual two-dimensional data structure of \( n \) rows and \( m \) columns—is very wasteful. That is because \( n \) and \( m \) can be large, and yet the matrix is sparsely populated since a typical subject has access to only a small subset of the total set of resources. A more efficient scheme represents the matrix as a set of lists. These can take on two different forms, as discussed next.
13.3.2 Access Lists and Capability Lists

The access matrix maintains information about the rights of subjects to access resources. We can segregate this information along columns or rows. For the former, all nonempty entries of each column \( j \) form a list associated with the corresponding resource \( R_j \). Each list is the access list of the resource. The second approach associates all nonempty entries of each row \( i \) with the corresponding subject \( S_i \). This list is the capability list of the subject.

By analogy, a capability may be viewed as a ticket (e.g., a theater ticket) that entitles its holder to exercise certain privileges, such as entering a theater. An access list, on the other hand, is comparable to a reservation list (e.g., in a restaurant); only parties whose names appear on that list are permitted to enter the establishment.

EXAMPLE: Access List vs. Capability List

To illustrate the differences between access lists and capability lists, consider the access matrix in Figure 13-3a. Figure 13-3b shows the same matrix organized as a set of access lists, one for each resource. For example, the access list for \( R_1 \) consists of the two entries comprising the first column; this shows that \( S_1 \) is allowed to read or write \( R_1 \), and \( S_3 \) has all three access rights. Figure 13-3c shows the same matrix organized as a set of capability lists, one for each subject. For example, the capability list associated with \( S_1 \) gives it read and write access to \( R_1 \), and unrestricted access to \( R_2 \).

On the surface, there appears to be little difference between capability lists and access lists, since the same amount of information is recorded in both cases. However, because the information is segregated differently and is associated with different types of entities, there are fundamental differences in how each approach can manage a number of important problems in terms of both functionality and efficiency of implementation.

Granularity of Subjects

In principle, a subject could be a user, a process, or an individual procedure (function). Access lists rely on recording the identity of each subject within the access list. Consequently, it would not be practical for the creator of a resource to identify all possible procedures that could access the resource. Similarly, the creator cannot identify all possible processes, since these are generally created dynamically. Thus, the only choice for a subject in an access list is the user, whose identity must be verified at each access.

In contrast, capabilities are not associated with the resources being accessed but are viewed as unforgeable tickets held by the different subjects. The possession of a capability is taken as proof that the holder is allowed access to the corresponding resource. Thus, controlling the propagation of capabilities is the main issue, which must involve authentication, but there is no authentication of the holder at the time of access. Consequently, capabilities can be maintained at a finer granularity than access list entries. In particular, when a new resource is created, the capability may be associated with a user, a process, or an individual procedure.
EXAMPLE: Granularity of Subjects

The above important difference is illustrated in Figure 13-4. Assume that a process running on behalf of a user, \( U \), executes two procedures, \( P1 \) and \( P2 \). Procedure \( P1 \) must read data from a file \( Q \), whereas \( P2 \) does not. Figure 13-4a shows that the access list associated with \( Q \) contains \( U \) as a valid subject authorized to read \( Q \). It would not be practical for the owner of \( Q \) to record individual procedures, like \( P1 \), as valid subjects. First, the owner would not know about all possible procedures that should access \( Q \), but, more importantly, the individual procedure must somehow be authenticated at runtime for each access to \( Q \). Since a system can generally authenticate the user process but no individual procedures, functions, or programs the process consists of, a procedure-level access granularity is not possible with access lists.

![Diagram showing access lists and capability lists](image)

**FIGURE 13-4.** Granularity of subjects using: (a) access lists; and (b) capability lists.

Figure 13-4b shows the same scenario using capabilities. This allows the user \( U \) to distributed capabilities only to those portions of the process that actually need them and vary the extent of the current access environment dynamically. In this example, \( P1 \) holds a read capability for \( Q \) and may perform the access. \( P2 \), on the other hand, would be denied access to \( Q \), because it lacks the necessary capability. As long as the capabilities are unforgeable, there is no need for any runtime authentication.
Chapter 13 Internal Protection Mechanisms

Static Versus Dynamic Environments

One of the main disadvantages of a user-level granularity of access lists is that the current execution environment of a process remains static during its entire execution. The process may, of course, switch to system mode by issuing a kernel call, in which case it is allowed to execute privileged instructions. However, its accesses will be subject to the same access lists as in user mode. For example, the process in Figure 13-4a is able to access $Q$ regardless of which procedure it is executing in. To support a dynamic environment while keeping the subject granularity at the user level, the system may allow a process to temporarily change its user identity. This must be done in a highly controlled manner to prevent an imposter from assuming another user identity.

**CASE STUDY: AMPLIFICATION OF PRIVILEGES IN UNIX**

In UNIX, every process is assigned a user ID, which identifies the user responsible for the process. As a process invokes different programs (files) in the course of its execution, the user ID associated with the process may change. This is controlled by a special permission, called the *set-user-id* flag, which is associated with every executable file $f$. When this permission is turned off, a process invoking $f$ will retain its current user ID. However, when the permission is turned on, the invoking process will temporarily inherit the user ID of $f$’s owner. The original user ID is restored automatically when the process exits the file $f$.

This is a powerful mechanism that can be used to temporarily amplify the privileges of a process for the duration of a call to a service routine. Assume, for example, that a user $U$ wishes to invoke a service program $P$ that is provided by another user $S$. Assume further that $P$ must access another file, $D$, to perform its function, but this file should not be accessible to $U$ in general. This can be accomplished by excluding $U$ from the access list of $P$, but enabling the *set-user-id* flag associated with $P$. When the process of user $U$ invokes $P$, the user ID changes to that of user $S$, which gives $P$ the necessary access privileges for $D$. Thus, the current execution environment of a process may change dynamically as it enters and exits different programs.

When a user process makes a kernel call, its current execution environment changes in two different ways. First, the processor is switched to system mode, which allows the process to execute privileged instructions. Second, if the *set-user-id* flag of the invoked system file is set, the process temporarily assumes the identity of the file owner, which is the OS itself (the root). This gives it greatly increased access privileges to systems resources.

List Sizes

Although a finer granularity of subjects results in more flexibility in expressing access constraints, a coarser granularity results in fewer entries and less space needed to maintain the lists. Reducing the number of entries in the lists is an important goal. Access lists lend themselves well to combining of subjects into groups, allowing a potentially large number of entries to be reduced to a few. For example, a file that should be generally readable by any user would include an entry for every possible user in its access list.
Providing a generic entry, “all users,” which acts as a wild-card for matching purposes, reduces the number of entries to just one.

**EXAMPLE: Grouping of Subjects**

Figure 13-5 extends the matrix of Figure 13-3a by adding an extra row labeled “*”, which is interpreted as a wild-card character. Thus, any subject, including newly created ones, automatically has all rights listed in this column. The matrix shows a new resource \( R_5 \), for which \( S_2 \) has read, write, and execution rights. In addition, the \( r \) right in the “*” row guarantees that all subjects, including those yet to be created, will be able to read the resource \( R_5 \).

<table>
<thead>
<tr>
<th></th>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>( R_3 )</th>
<th>( R_4 )</th>
<th>( R_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( r)w</td>
<td>( r)wx</td>
<td>( r)w</td>
<td>( r)w</td>
<td>( r)</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>( x)</td>
<td>( r)w</td>
<td>( r)w</td>
<td>( r)w</td>
<td>( r)w</td>
</tr>
<tr>
<td>( S_3 )</td>
<td>( r)wx</td>
<td>( r)w</td>
<td>( r)w</td>
<td>( r)w</td>
<td>( r)</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

**FIGURE 13-5.** Access matrix extended with a group subject.

When the matrix is implemented as an access list, adding such new rows is simple. The access list of the corresponding resource is extended to contain a new entry with “*” as the subject. In the above example, the extended access list for \( R_5 \) would have the form \((S_2, rwx), (*, r)\).

In contrast, implementing a group-access row in a capability-based system is more difficult. If the capability lists are maintained by the system, the new right could be entered into the capability lists of all subjects. Alternately, a separate capability list associated with the “*” could be maintained and included automatically with every subject capability list. However, when capabilities are implemented as unforgeable tickets maintained by their owners and dispersed throughout the system, there is no easy way to implement the new group-access row. The new right to be granted by default to all subjects needs to be explicitly propagated to all capability lists, but their owners and locations are unknown. Worse yet, newly created subjects would not automatically inherit such group rights as a default.

**CASE STUDY: ACCESS LISTS IN UNIX**

UNIX employs access lists to implement the access matrix and is able to conveniently support grouping of access rights. The following scheme has been adopted. A file access list distinguishes three types of subjects: 1) the file owner; 2) members of a specifically named group; and 3) all users. The first entry refers to the original creator of the file. The purpose of a group is to permit a selected team of users to access the file with rights different from those granted to the owner or to the general public. Every file belongs to one group, and every user belongs to one or more groups. Finally, all users are members of the general public; the group designated as “*” in the previous discussion. This gives all users a set of default rights with respect to a resource.
Within each of the three user types, a file may be readable (r), writable (w), or executable (x). For example, a file designated as rwxr-x--x permits unrestricted access (rwx) by the owner, read and execute access by any member of the group (r-x), and execute-only access by all others (--x).

Combining subjects into a small number of groups also solves the problem of authentication in systems where subjects are individual procedures. Access lists only maintain information in terms of groups, rather than individual procedures. Thus, the maximum number of entries per access list is limited by the number of groups. Each entry records the rights that apply equally to all subjects (procedures) of a group. This permits the use of access lists with fine granularity of subjects and supports a dynamic execution environment. At the same time, authentication is necessary only when assigning a procedure to a given group.

The Multics OS pioneered the idea of grouping or levels of protection for access control (Schoeder and Saltzer 1972). Any segment, including executable segments and data segments, is assigned to one of \( n \) possible groups, numbered 0 to \( n - 1 \). The groups are ordered so that segments in group 0 have the most access privileges, and segments in group \( n - 1 \) have the least privileges. A convenient way to visualize the groups is as an arrangement of concentric rings, with the most protected ring 0 at the center.

Figure 13-6 shows the initial ring assignments in Multics. The OS occupies the first three rings with the most critical part—the nucleus—occupying ring 0. Most of the system is in ring 1, and the remainder—the least sensitive segment of the OS—resides in ring 2. Rings 3 to \( n - 1 \) may be employed by user processes.

The rings control access to segments using this rule: When a process executing a program \( S \) in ring \( i \) attempts to read or write a segment \( T \) in ring \( j \), the access is allowed to proceed only when \( j \geq i \) and the access list of \( T \) contains the necessary read/write rights for the calling process.

The rings also are used to limit transfer of control between segments, but the rules are more complicated. A call to a segment with the same or higher ring number is allowed (provided the caller has the execute right), but it may be necessary to copy the parameters passed from the calling segment \( S \) to a ring with lower protection to enable the called segment \( T \) to access them. A call to a segment with a lower ring number may or may not be allowed; this is determined by a special ring limit number prescribed as part of \( T \). When the caller’s current ring number is greater than this limit, the call is disallowed.

Only a very small number of the originally planned 64 Multics rings were implemented, indicating some problems with the scheme. One of the main constraints is the linearity of the ring-based protection mechanisms; resources must be ordered according...
to the rights they have to one another. Such ordering may not always be possible or convenient since, in general, resource references could form an arbitrarily interconnected (e.g., cyclic) graph.

**Adding/Removing Resources and Subjects**

Adding or removing a resource in an access matrix representation corresponds to adding or removing a column. Let us consider how this can be handled in access list and capability list implementations. Whenever a new resource $R$ is created, its creator is generally given the responsibility to determine how the resource should be accessed. With access lists, the creator is able to include (explicitly or by default) any subject that should be given access to $R$. With capability lists, the creator is given the initial capability for the resource, which it may pass on, perhaps with a reduced set of rights, to other subjects.

The deletion of a resource could be controlled by a specific **delete** right that the creator automatically obtains at resource creation. More commonly, an **owner** right is defined that entitles the creator to any operation on the resource. In Figure 13-7, the owner right is represented by the character $o$. Thus, $S_1$ is the owner of $R_2$, $S_2$ is the owner of $R_3$, $R_4$, and $R_5$, and $S_3$ is the owner of $R_1$.

The ability to add a new subject, which corresponds to creating a new row of the matrix, depends greatly on the subject granularity. With access lists, subjects are
generally synonymous with users. Thus, adding a new subject can be performed only by the OS, in conjunction with an administrative user-authorization action. To be granted access to any resources, the new user must be included in existing access lists (explicitly or by default). Similarly, removing a subject can be done only by the OS.

With capability lists, subjects can be individual procedures. Since these are themselves resources that require access control, adding a row of the matrix automatically adds a column. The capability list associated with this new subject is normally empty initially and is subsequently populated by propagating capabilities to it. Similarly, the capability for the new subject may be propagated to other subjects. Thus, the main issue in the creation of new subjects is the propagation of capabilities, which is discussed next. The destruction of a subject can generally be performed by its creator, as long as the creator retains the original capability for the subject.

Adding/Removing Rights

In the access matrix model, adding and removing rights amounts to modifying individual entries of the matrix. With access lists, the responsibility for adding and removing rights is delegated to the resource owner, designated by the special owner right. This right gives the owner unrestricted ability to include or exclude any subject in the resource access lists and to specify the rights each subject should have. For example, subject $S$ could extend any rights for resource $R$ to any other subject by modifying $R$’s access list accordingly. Similarly, $S$ could later restrict or remove the previously granted access rights. A subject can even remove its own rights. For example, it could remove its $w$-right for a resource to prevent an accidental modification or deletion of that resource. As long as the subject retains its $o$-right, it can always add the $w$-right back to the access list.

With capability lists, the problem of rights management is more difficult to handle because the capabilities are managed by various subjects, rather than the resource owners. To support dynamic rights, it is necessary to guarantee that:

1. a capability is unforgeable;
2. a subject cannot pass copies of capabilities it possesses to other subjects in an uncontrollable manner;
3. a subject can cancel or revoke a previously granted capability.

Making Capabilities Unforgeable

There are several approaches to protecting capabilities from being fabricated by unauthorized subjects. In a centralized system, the safest method is to use a tagged architecture, where each memory word contains a special hardware tag, indicating whether
it contains a capability or another type of data. Only special privileged instructions, running in system mode, are allowed to create or modify memory locations tagged as capabilities.

If the architecture does not support memory tags, the capabilities may be segregated in an area inaccessible to user processes and managed by the OS. A subject can refer to its capabilities only indirectly, e.g., using an index into an array of capabilities. This is similar to managing memory pages or segments in virtual memory; the user specifies a page or segment by its page or segment number, and the system maintains the actual internal addresses.

In a distributed system, capabilities must be passed between different machines. One simple technique to prevent capabilities from being fabricated or tampered with applies an idea similar to that used for password protection: The capabilities are represented by unique strings or bit patterns chosen from a potentially large name space. This decreases the chances of an intruder guessing a valid capability by trial and error.

A second, and safer, method uses cryptography. For any newly created capability \((R, \text{rights})\), where \(R\) is the resource and \(\text{rights}\) is the list of applicable rights, the system generates a random number, \(N\). The system records this number along with the capability and generates a ticket of the form \(H(R, \text{rights}, N)\), where \(H\) is a one-way encryption function. This ticket is given to the subject and must be presented to the system along with the original capability to validate its authenticity. Assume a subject presents a capability \((R', \text{rights'})\) and a ticket \(t\). The system looks up the number \(N'\) saved with the presented capability and computes the value \(t' = H(R', \text{rights'}, N')\). If \(t'\) matches the ticket \(t\) presented by the subject, the capability is valid and the requested operation is carried out, subject to the access rights; otherwise, the capability is rejected as a forgery.

Controlling Propagation of Capabilities

One of the advantages of capability lists over access lists is that the resource creator need not specify and manage a complete list of subjects that are allowed to access the resource. Instead, a capability for the newly created resource is passed to a subject, from which it propagates to other subjects as needed. To permit this, subjects must be able to move or copy capabilities to other subjects, but this must be done in a controlled manner to prevent protection violations.

For this task, some additional mechanism must exist that allows a subject to pass a capability to another subject, but restrict it in such a way that it cannot be propagated any further. In other words, the subject must be able to restrict the copying of a capability to only a one-time occurrence. We accomplish this by adding a special nonpropagation right to the capability. Unlike other rights, the nonpropagation right does not apply to the resource pointed to by the capability, but instead applies to the capability itself. A capability lacking this special right is prevented (by the OS) from being copied into any other capability list and is unable to propagate through the system.

Note that a capability can be propagated in one of two ways. First, a subject \(A\) holding the capability could copy it into the capability list of another subject, \(B\), provided \(A\) has the necessary rights to write into \(B\)'s capability list. Second, the subject \(B\) could copy the capability from \(A\)'s capability list, provided \(B\) has the necessary rights to read from \(A\)'s capability lists. The special nonpropagation right attached to a capability prevents both types of propagation.
CASE STUDY: CONTROLLING PROPAGATION OF CAPABILITIES

The Hydra OS (Cohen and Jefferson 1975) calls the nonpropagation right the environment right (e-right, for short). A capability can be copied only if it includes an e-right. Figure 13-8 shows how the lack of the e-right prevents a capability from being copied.

The figure shows five entities, A, B, C, D, and E. A has a capability for C with the s-right. This right, called the store right, permits a subject to copy capabilities from its own capability list to the capability list of the resource pointed to by the capability. Thus, A can store copies of its own capabilities into C’s capability list. In particular, A can copy the capability for D (including the erw rights) into C’s list; this transfer is possible because the capability being copied has the e-right. A capability lacking the e-right, such as A’s capability for the resource E, cannot be copied. Thus, C cannot be given access to E. Figure 13-8b shows the updated capability list of C, where the crossed-out dashed pointer represents the capability for E that could not be propagated to C. Note that A could have decided to remove the e-right from the capability given to C. This would allow C to read and write D as before, but would prevent it from propagating the capability for D any further.

Figure 13-8a also shows that B has a capability for A with an l-right. This right, called the load right, enables its holder to load rights from the resource pointed to by the capability. Thus, B can read and copy capabilities from A’s capability list. In particular, it can copy the capability for D (including the erw-rights), but it is unable to copy the capabilities for either C or E, because both lack the e-right. Figure 13-8b shows the new capability list for B.

FIGURE 13-8. Propagation of capabilities (a) initial state; (b) after transferring capabilities for D to B and C.
Revocation of Capabilities

Revocation of capabilities is the inverse of capability propagation. This concept is difficult to implement directly since many copies of the same capability may be dispersed throughout the system and may not be easy to find. Indirection can be used to eliminate this problem. A dummy or alias resource is created that contains the capability for a resource. Individual users are given capabilities for the alias instead of the resource itself (Fig. 13-9). When revocation of access to the resource is desired, the alias is simply destroyed, breaking all indirect connections to the resource. Note that with this approach, all users lose their capabilities to the resource; it is not possible to remove privileges from users on an individual basis.

![Figure 13-9. Use of alias for revocation of access rights.](image)

13.3.3 A Comprehensive Example: Client/Server Protection

The majority of protection problems arise because various subsystems must cooperate. To illustrate the variety of potential problems and their solution, we consider a general client/server scenario, as shown in Figure 13-10. Assume that an owner offers a service program that may be called by legitimate users of the system. To do so, a user must supply information to the service in the form of parameters. After completing its work, the service returns the results of the computation to its caller. In addition, the service must communicate with its owner, e.g., it may report billing or performance data.

The owner and the user of the service have different concerns regarding their own security and the security of the service itself. The following requirements must be satisfied in general. The possible solutions depend on whether access lists or capability lists are employed:
Chapter 13  Internal Protection Mechanisms

FIGURE 13-10. Example of mutually suspicious applications.

**Requirement 1:** No user should be able to steal (obtain a copy) or damage the service in any way.

Theft and destruction of information are solved through the enforcement of execute-only privileges for the resource, in this case, the service. These privileges are normally provided in both access list-oriented and capability list-oriented protection schemes.

**Requirement 2:** No user should be able to employ the service without the owner's permission.

Unauthorized use of the service is a problem of right propagation. With access lists, propagation is not possible since only the owner of a resource can extend any rights to it. With capabilities, specific mechanisms must be implemented, e.g., the e-right in the Hydra OS.

**Requirement 3:** The owner should be able to revoke access to the service by an authorized user.

This problem requires the removal of previously granted rights. With access lists, this is accomplished simply by modifying the list associated with the service. With capability lists, the reference to the service can be set up as an alias that can later be broken to deny further access. Otherwise, the only convenient way to ensure no further use is to destroy the service itself and recreate it with a new capability.

**Requirement 4:** No user (authorized or not) should be able to prevent authorized users from employing the service.

This addresses the problem of service denial. The simplest form of service denial results when the service is destroyed or damaged. Both access lists and capability lists provide means to prevent such actions by not granting write/modify rights to any user. A more subtle form of service denial occurs any time a legitimate user is impeded from making sufficient progress. This intuitive measure is, unfortunately, subjective since insufficient progress may not necessarily be caused by malicious actions on the part of a user or a system malfunction; it could simply be the result of an unexpectedly high demand on a shared resource or a general system overload. Hence, in general, it will not be possible to guarantee that denial of service does not occur. The best we can do is provide mechanisms to detect denial of service for specific critical resources and inform the user or a higher authority of such situations so that corrective actions can be initiated. The detection can be accomplished by associating a maximum service time with each service to be monitored. When this time limit is exceeded, the process is considered to be making insufficient progress, incurring denial of service.
Requirement 5: The service should be able to access its own private files or other resources when performing the service, without granting the user access to these resources.

Right amplification provides the desired control. When executing the service procedure, a user process is given additional access rights, in particular, permitting it to use resources needed by the service. The current execution environment must be dynamic for this solution to work.

With access list-based systems, right amplification can be done by allowing a process to temporarily change its user or group ID (as in Section 13.3.2) or to provide resource groupings such that the service would belong to a more privileged group (e.g., having a lower ring number in Multics.) With capabilities, this is accomplished elegantly by simply associating the capabilities for the private resources with the service but not with the user.

Requirement 6: The service should not be able to steal, destroy, or otherwise compromise any information or services that the user did not explicitly supply to the service.

This addresses the Trojan horse problem (Section 12.1.3), where the service could take unauthorized and unadvertised actions to harm the user. The problem cannot be solved in systems with static execution environments, since any borrowed program executes with the same privileges as the invoking process. In a dynamically changing environment, we could temporarily change the user ID to that of the service owner. Alternately, we could place the service in a group with lower privileges than the user. For example, Multics could assign a higher ring number to the service than to the user. In both cases, the service would be prevented from accessing any of the user resources. However, most services must be supplied with parameters or other resources belonging to the user to perform their tasks. To make any such parameters accessible to the service, they must be down-protected (at least temporarily) to the current level of the service. All these solutions to satisfy Requirement 6 are rather awkward to implement in systems using access lists.

Systems based on capabilities solve this problem in a more elegant manner. Since capabilities generally may be passed as parameters, the user can pass the necessary information to the service at the time of the call. This method gives the service the necessary rights, while preventing it from accessing any other resources not passed explicitly to it.

13.3.4 Combining Access Lists and Capability Lists

The more common way to organize protection information is in the form of access lists. However, capabilities also are used in many systems, frequently in conjunction with access lists to reap the benefits of both approaches.

A good example of a combined approach is with files (Section 10.3). In most file systems, a file must first be opened before its contents can be accessed. The open command verifies that the subject has the required type of access to the file by consulting the file access list. If the access is authorized, the open command returns a handle (pointer, or file descriptor in UNIX terminology) for the open file, that is subsequently used to read, write or otherwise access the file. The file handle represents an unforgeable capability for the open file, and is used without verifying the subject’s identity upon every access to the file.
Another example is the dynamic linking of segments presented in Section 9.4. When a segment is referenced for the first time, the system verifies that the invoking process has the necessary rights by examining the access list associated with the segment and maintained by the file system. If the access is valid, a segment number \( s \) is assigned to the segment and its memory address is entered into the segment table with the valid access rights. Thus, the segment table may be viewed as the process capability list; each entry points to a valid resource (segment in memory) and records the valid rights for this segment. On subsequent accesses to the same segment, the segment number \( s \) is used as an index into the segment table to select the capability for the segment.

A third example is the use of the tickets in the Kerberos authentication system, discussed in Section 12.4.2. Initially, a client wishing to use a service must authenticate itself using a password. When this is successful, it is issued a ticket-granting ticket, which is a capability for the TGS. The capability is protected from tampering by being encrypted with a secret key known only to TGS. When presented to TGS, the capability enables the client to obtain other tickets (capabilities), also protected by encryption, which, in turn, enable the client to use the desired system services.

### 13.4 INFORMATION FLOW CONTROL

Consider the client/server scenario introduced in Section 13.3.3, and assume that the user of this service would like to impose the following additional requirement on the service: *The service must not be able to leak any information entrusted to it by the user to the owner or any other party.*

Such a breach of security can occur easily if the service is a Trojan horse program. Note that this is different from Requirement 6 of Section 13.3.3, which was concerned with protecting information not supplied to a service. The present concern is to prevent sensitive information actually entrusted to the service from being compromised. This is a matter of information flow control rather than access control.

Figure 13-11 illustrates the difference between the two problems. In Figure 13-11a, an unauthorized process is not permitted to read sensitive information—a problem of access control. In Figure 13-11b, a process is allowed to access the sensitive information but must be prevented from passing it to an unauthorized process.

#### 13.4.1 The Confinement Problem

Information may be passed between processes in many ways without authorization, depending on the interprocess communication facilities available. In most instances, this involves copying data into an area that can be read by an unauthorized process. To prevent such illegal information flow through a service procedure, we must deny write access by the service to any areas accessible by other processes. This is the *confinement problem*, so-named because the service must be surrounded by a boundary beyond which it cannot pass any information. The challenge is how to define and enforce such a boundary.

In one method, the service is not allowed to modify any resource unless that resource was explicitly passed to it by the caller or client. This gives the caller complete control over which resources can be modified by the service, and stops the flow of any information beyond the modifiable resources.

Consider how this policy could be enforced using capabilities. The basic idea is to extend the procedure-calling mechanisms such that the caller can disable all writing
Section 13.4 Information Flow Control 465

FIGURE 13-11. Flow of information (a) through direct read (b) via an authorized process.

capabilities of the service for the duration of the call, except those supplied explicitly as parameters. The Hydra OS implemented a special right for this purpose: the modify right or m-right for short. The m-right must be used in conjunction with a right that modifies a resource, such as a write or append right. Without the presence of the m-right, all such rights are automatically disabled. As part of the procedure call to a service, the caller may request that the m-right be masked out from all capabilities currently held by the service. At the same time, it may supply to the service capabilities as parameters; these may retain their m-rights and may be modified by the service.

CASE STUDY: CONFINEMENT OF A SERVICE

Figure 13-12 shows an example of the confinement of a service during its invocation by a user. We assume that the user capability list currently contains four capabilities: the first points to its own code segment; the second points to its own data segment; the third points to the service procedure to be called by the user code; and the fourth points to a data segment to be passed to the service procedure as a parameter. The latter may contain the caller’s sensitive information, which must be passed to the service but must not be leaked to any other process, including the service owner.

The service procedure has a capability list containing three capabilities: one for its own code segments; one for its data segment; and the third for a resource labeled common that it shares with the service owner. The service intends to use this resource for
the illegal passing of information to its owner, who has read access to it. Figure 13-12a presents the processes before the call is issued.

At the invocation of the service, the caller requests that all m-rights be masked out from the service capability list. Figure 13-12b shows the capabilities that apply during the execution of the service. All previously held m-rights of the service are disabled, which renders all corresponding w-rights ineffective. Specifically, the service cannot write into the common resource, since the lack of the m-rights disables the w-right. The capability for the parameter resource, on the other hand, has been passed to the service with the m-right; hence, the w-right, as well as all other rights on that capability, are applicable. This confines the service such that it can read or execute any resources for which it had capabilities, but it can only modify those resources supplied by the caller as parameters.

Note that the m-right facility permits only a total confinement of the service; the service is unable to pass even nonsensitive information, e.g., billing or performance data, to its owner.
13.4.2 Hierarchical Information Flow

The security requirements of the U.S. DoD motivated much of the research into controlling the flow of information. The military, as well as a number of other institutions, base their information management policies on a hierarchy of security classes. Subjects may access resources in a given class only if they possess the corresponding clearance. One of the most important properties that must be enforced by such a hierarchical model is that all information flows only in one direction: from less sensitive to more sensitive security classes.

Security policies where access to resources is granted based on membership in security classes, rather than on the identity of subjects, are called nondiscretionary. These do not eliminate the need for discretionary policies, which grant privileges on an individual basis. Specifically, the military enforces the principle of "need to know," according to which each subject should be permitted to access only those resources that it needs to perform its duties. Such policies must be based on the subject's identity. A comprehensive security system must include both discretionary and nondiscretionary policies.

Most existing hierarchical security models are based on the fundamental principles formulated by Bell and LaPadula at the MITRE corporation (Bell and LaPadula, 1973a, b, c; 1984). Their model is one of the first attempts to formalize the problem of information flow control. The model extends the basic access matrix model (Section 13.3.1), which governs the discretionary policies, with levels of security, which govern the nondiscretionary policies. As before, each entry of the access matrix records the rights (e.g., read, write, or execute) that an individual subject has with respect to a given resource. In addition, each subject is assigned a security clearance, and each resource is assigned a classification level. Both security clearances and classification levels are taken from a fixed ordered set, such as unclassified, confidential, secret, top secret.

The requirement that information may flow only from lower to higher security levels is enforced by two important rules:

- **No Read Up.** A subject $S$ may not read from a resource $R$ unless the classification level of $R$ is less than or equal to the clearance of $S$. That means, $S$ may not read from any resource at a higher security level. For example, a subject cleared for secret documents also may read those classified as confidential or unclassified, but not top secret.

- **No Write Down.** A subject may not write a resource $R$ unless the classification level of $R$ is greater than or equal to the clearance of $S$. That means, $S$ may not write into any resource at a lower security level. For example, a subject cleared for secret information may create documents with only secret or top-secret status, but not confidential or unclassified.

Figure 13-13 depicts graphically the legal flow of information enforced by the two rules. Consider a subject $S$ with clearance $i$, ($1 \leq i \leq n$). The possible flow of information due to reading by subject $S$ is shown as dashed lines, and information flow due to writing by subject $S$ is shown as solid lines. In all cases, the information flows only horizontally (i.e., between classes with equal security levels) or upward (i.e., from lower to higher security levels). Note that a subject requiring both read and write privileges to a resource must have a clearance equal to that resource classification level.
EXAMPLE: Using Hierarchical Information Flow

The hierarchical information flow model can, in principle, be used to solve the confinement problem. It requires that the user of a service suspected to be a Trojan horse possesses higher security clearance than either the service owner or any other subject to whom the service should not disclose information supplied to it by the caller. Assume that the user’s clearance is \( i \), and the owner’s clearance is \( j \), where \( i > j \). Under such conditions, the user is able to invoke the service, which will execute at the user’s current clearance, \( i \). That permits the service to access any information belonging to the user.

FIGURE 13-13. Legal flow of information in a multilevel system.

FIGURE 13-14. Preventing illegal information flow (a) before invocation (b) during invocation.
(subject to the discretionary read/write/execute rights of the access matrix). The service, however, is not able to pass any of this information down to its owner, due to the owner’s lower security clearance.

Figure 13-14 contains an example of such a scenario. Operating at the secret level, the user wishes to invoke a service defined at the lower unclassified level. Figure 13-14a shows the situation prior to the call. In Figure 13-14b, the service is in use. During the execution of the service, it operates at the user’s secret level, which permits it to access the user’s data, but prevents it from writing into any files at the owner’s unclassified level.

13.4.3 The Selective Confinement Problem

The objective of the confinement problem as discussed in the previous section is to prevent a service program from disclosing any information entrusted to it by its caller. This is a total confinement, since no information whatsoever may be leaked by the service to any other party, including the service owner. Such total confinement is only a special case of the more general and difficult problem of selective confinement, which distinguishes between sensitive and nonsensitive information. The objective of selective confinement is to prevent only the disclosure of sensitive data. For example, the service example presented in Section 13.3.3 should be able to send accounting and other information to its owner, based solely on nonsensitive information, such as the user’s name and identity. It should not, however, be allowed to disclose any sensitive information.

Due to the difficulty of distinguishing between sensitive and nonsensitive information, and keeping these strictly segregated as part of the computation, few existing systems provide mechanisms that offer a satisfactory solution for the general case of selective confinement. In fact, it has been shown that, in its full generality, this problem is unsolvable (Fenton, 1974).

EXAMPLE: Limiting Information Flow

To illustrate the difficulty of tracking and limiting information flow, consider the following sequence of statements:

\[
Z = 1; \\
Y = 2; \\
\text{if}\ (X == 0) \ Z = Y;
\]

From the last assignment statement, it is clear that some information flows from \(Y\) to \(Z\). Thus, by testing the value of \(Z\) at the end of the code fragment, we can deduce some information about \(Y\). Such information flow is called explicit, since \(Y\) is used to compute \(Z\).

Much less obvious is the fact that we also can deduce some information about the value of \(X\) at the end of the three statements. This can be done by testing the new value of \(Z\). If \(Z\) is equal to 2, we know that \(X\) must be equal to zero. Thus, in addition to the explicit information from \(Y\) to \(Z\), some information also flows from \(X\) to \(Z\), regardless of whether the conditional statement is true of false. Such information flow is referred to as implicit.
Chapter 13 Internal Protection Mechanisms

The Lattice Model of Information Flow

To solve the selective confinement problem, the system must be able to restrict both explicit and implicit flow of information. The lattice model of information flow offers one approach to solve this difficult problem (Denning, 1976). The basic idea is similar to the hierarchical information flow model discussed in Section 13.4.2. A set of security classes is defined, and each subject and each resource is assigned to a class. The main difference is that the classes are not arranged according to a total ordering. Instead, a flow relation specifies the legal flow of information between any two security classes, and a class-combining operator specifies how any two security classes may be combined into a higher security class. The flow relation and the class-combining operator organize the security classes in a lattice, i.e., a partial order.

To enforce legal flow of information, any computation $RES = f(R_1, \ldots, R_n)$, where $f$ may be an assignment or a conditional statement, may proceed only if the security class of the result $RES$ is greater or equal to the security class of all the resources $R_1, \ldots, R_n$ used by that computation. This security class is derived using the class-combining operator. This policy guarantees that information flows only along the legal channels as defined by the flow relation.

EXAMPLE: Information Flow Lattice

Consider a system containing three types of records: medical, financial, and criminal. A resource can be classified as containing one or more of the three types of information, e.g., a resource could contain purely medical information, a combination of medical and criminal, or all three types. Information of a given type may flow only into resources classified as containing that information type. This can be expressed by the lattice of Figure 13-15. The lower bound of the lattice is the empty set ($\phi$); the flow relation ($\rightarrow$) is the subset operator ($\subseteq$); the class-combining operator is the set union operator ($\cup$); and the resulting upper bound of the lattice is the set $\{Medical, Financial, Criminal\}$.

Assume that a given computation combines a medical and criminal record into a new record, $R$. The class-combining operator produces the class $\{Medical\} \cup \{Criminal\} = \{Medical, Criminal\}$ for the record $R$. The flow relation dictates that this record may

![Diagram of Information Flow Lattice](image)

be placed only into the class \{Medical, Criminal\} or \{Medical, Financial, Criminal\}; placement into any other class would violate the flow relation and would be disallowed.

**Sneaky Signaling**

The objective of confinement (both total and selective) is to prevent sensitive information from escaping out of the service program. Unfortunately, there is a number of possible implicit covert channels through which information may be conveyed to an observer. The use of such channels for information disclosure is called **sneaky signaling**. In principle, an element with two different states (representing zero and one) is sufficient to encode and transmit any amount of information when interrogated repeatedly over time.

There are many possible patterns of behavior that a service might display to signal binary information to an observer. For example, to signal a 1, a process $S$ could open an agreed-upon file, say $A$; to signal a 0, it would open another file, $B$. An observer would then attempt to open the two files for writing. One of these open commands will fail because either $A$ or $B$ has already been opened by $S$. Depending on which file cannot be opened, the observer can deduce one bit of information signaled by $S$.

Other covert channels that can be employed in a similar manner for sneaky signaling include using different I/O devices or generating error messages that depend on the information content. Similarly, information may be encoded and passed by controlling the patterns of tape movement or of sound waves emanating from a printing device. The use of different time delays caused intentionally as a way to encode sensitive information is particularly difficult to detect and prevent. For example, the execution of a long loop could be initiated or suppressed based on some sensitive data, signaling one bit of information. Since such time delays are potentially unbounded; the problem of sneaky signaling and the confinement problem is quite impractical to solve, even when limiting assumptions about the system and its environment are made. In their full generality, confinement problems have proven to be unsolvable. Therefore, other means, particularly cryptography, are suggested for protecting sensitive and critical information.

**CONCEPTS, TERMS, AND ABBREVIATIONS**

The following concepts have been introduced in this chapter. Test yourself by defining and discussing each keyword or phrase.

<table>
<thead>
<tr>
<th>Access control</th>
<th>Read right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access list</td>
<td>Revocation of rights</td>
</tr>
<tr>
<td>Access matrix</td>
<td>Right amplification</td>
</tr>
<tr>
<td>Access rights</td>
<td>Rights</td>
</tr>
<tr>
<td>Capability</td>
<td>Rings of protection</td>
</tr>
<tr>
<td>Confinement problem</td>
<td>Store right</td>
</tr>
<tr>
<td>Environment right</td>
<td>Safety</td>
</tr>
<tr>
<td>Hierarchical information flow</td>
<td>Security classes</td>
</tr>
<tr>
<td>Information flow control</td>
<td>Security classification level</td>
</tr>
<tr>
<td>Instruction-level access control</td>
<td>Security clearance</td>
</tr>
<tr>
<td>I/O protection</td>
<td>Selective confinement</td>
</tr>
<tr>
<td>Lattice model</td>
<td>Sneaky signaling</td>
</tr>
</tbody>
</table>
Chapter 13  Internal Protection Mechanisms

| Modify right | Subject                  |
| Memory protection | System mode          |
| Owner right        | Total confinement       |
| Privileged instruction | User mode            |
| Processor mode    | Write right            |
| Propagation of capabilities | Execute right |

EXERCISES

1. Consider a paging system that divides each page into four sandboxes.
   (a) Assuming a virtual memory occupies 32 bits and the page size is 1024 words, what is the total number of sandboxes and how many bits are necessary to represent a box number?
   (b) The system enforces the confinement of a program to a sandbox \( b \) as follows. Any address \((p, w)\) generated by the program is automatically transformed into a legal address using the following bit manipulations. The logical AND removes the current box number, and the logical OR provides the assigned box number:

\[
(p, w) \land \text{mask} \lor (b, x)
\]

Determine the structure and content of the \text{mask} and the field labeled \text{x}.

2. Consider a system using segmentation with paging. The virtual address has the form \((s, p, w)\), where \(|s| = 5\) bits, \(|p| = 7\) bits, and \(|w| = 9\) bits. Assume there are currently five segments (0 through 4) with the following respective lengths: 50, 515, 2048, 1200, and 2049.
   (a) The following three address are all invalid; for each, give the specific reason why:
      • \((9, 0, 0)\)
      • \((4, 5, 6)\)
      • \((1, 1, 15)\)
   (b) For each of the five segments, give the valid range of \(p\) and \(w\).

3. Consider the logical-to-physical address conversion function of systems with both segmentation and paging (Section 13.2.2). Simplify this function for systems with only segmentation. Show the new function and the corresponding segment table entry format.

4. Repeat the previous exercise for systems with only paging. Show the new function and the corresponding page table entry format.

5. Extend the logical-to-physical address conversion function (Section 13.2.2) to work with systems where:
   (a) the segment table is paged
   (b) each page table is paged
Show the new functions and the corresponding table entry formats.

6. A list command in UNIX produced the following information about the current directory:

<table>
<thead>
<tr>
<th>protection</th>
<th>owner</th>
<th>group</th>
<th>file name</th>
</tr>
</thead>
<tbody>
<tr>
<td>-rwxr-x---</td>
<td>smith</td>
<td>opsys</td>
<td>f1.txt</td>
</tr>
<tr>
<td>-rwxr-----</td>
<td>smith</td>
<td>comp</td>
<td>f2.txt</td>
</tr>
</tbody>
</table>
Assume that smith is a member of all three groups, opsys, comp, and misc, and richards is a member of only misc.

(a) Show the above information in the form of an access matrix. (Hint: Represent the three groups, opsys, comp, and misc, as separate subjects.)

(b) Which files are accessible by smith? Which are accessible by richards?

(c) Show all changes to the matrix resulting from the following operations:
- smith performs: chown richards f2
- smith performs: chgrp misc f3
- richards performs: chmod 750 f4
- smith performs: chmod 444 f1

Which files are accessible by smith and by richards after all changes have been made?

(d) Consider the access matrix under (a) and assume it is to be implemented using capability lists. The OS keeps track of which users are members of which groups, but no capability lists are associated with groups. Instead, the corresponding capabilities must be replicated within each user list. Show the capability lists for smith and richards.

7. A system consists of two users, U1 and U2, and three resources, R1, R2, and R3. The process of U1 consists of two functions, P1 and Q1. The process of U2 consists of two other functions, P2 and Q2. The four different functions must access the resources as follows:
- P1 reads R1
- Q1 reads/writes R1 and R2
- P2 reads R2
- Q2 reads R3

Implement the above scheme using:
(a) access lists
(b) Multics protection rings
(c) capability list

In each case, give each subject the least amount of privileges necessary to do its job.

8. Consider five objects, A, B, D1, D2, and D3, in a capability-based system like Hydra.

(a) Show the graphical representation of the objects’ capabilities such that the following operations may be performed:
- A can call B;
- B can read and write data from/to D1;
- B can give its capability for D1 to another object D2;
- A can read and write the capability list of D1;
- A can read data from D3;
- B can read and write data from/to D3.

(b) Answer the following questions based on your diagram. If the answer is yes, give a sequence of operations to accomplish the task:
- Can data from D3 ever get into D1?
- Can data from D3 ever get into D2?
- Can data from D1 ever get into A?
- Can data from D2 ever get into A?
9. Consider the following capability lists in a system like Hydra:

<table>
<thead>
<tr>
<th>subject</th>
<th>capability for object</th>
<th>rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>X</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>D3</td>
<td>emrw</td>
</tr>
<tr>
<td>P2</td>
<td>D3</td>
<td>l</td>
</tr>
<tr>
<td>X</td>
<td>D1</td>
<td>emws</td>
</tr>
<tr>
<td></td>
<td>D2</td>
<td>emrws</td>
</tr>
</tbody>
</table>

(a) Show a graphic representation of the above configuration.
(b) Assume P1 calls X and passes to it the capability for D3 as a parameter. Show the current capability list of X assuming the m-right was masked out during the call.
(c) During the call from P1 to X, which entities can X:
- read data from;
- load capabilities from;
- write data into;
- store capabilities into.
(d) Repeat the previous question for the state after X returns control to P1.
(e) Show a sequence of operations by which either of the two capabilities held by P1 propagates to P2.

10. Consider a system with two subjects, S1 and S2, and four resources, R1 through R4.
(a) Assign clearances to the subjects and classification levels to the resources according to the Bell-LaPadula model, such that the following conditions hold:
- S1 can write only into R3 and R4;
- S2 can write only into R3.
(b) Determine which resources can be read by which subjects under the assignment of part (a).
(c) Modify the assignment such that S1 cannot read R4.

11. Consider the set of security classes, C = {00, 01, 10, 11}, in the Lattice model of information flow. Define the flow relation “→” and the class-combining operator to form the following four possible lattices:

(a) ![Lattice A](image-a)
(b) ![Lattice B](image-b)
(c) ![Lattice C](image-c)
(d) ![Lattice D](image-d)

12. Consider the set of security classes, C = {(XYZ)|X, Y, Z ∈ {0, 1}}. Derive possible lattices analogous to those of the previous exercise. For each, specify the flow relation, the class-combining operator, and the lower and upper bounds.