Part Seven

Protection and Security

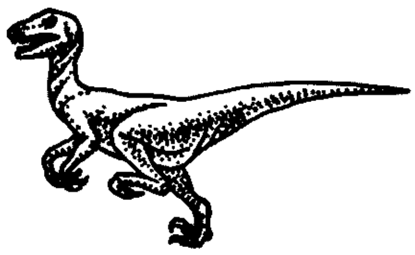
Protection mechanisms control access to a system by limiting the types of file access permitted to users. In addition, protection must ensure that only processes that have gained proper authorization from the operating system can operate on memory segments, the CPU, and other resources.

Protection is provided by a mechanism that controls the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specifying the controls to be imposed, together with a means of enforcing them.

Security ensures the authentication of system users to protect the integrity of the information stored in the system (both data and code), as well as the physical resources of the computer system. The security system prevents unauthorized access, malicious destruction or alteration of data, and accidental introduction of inconsistency.

CHAPTER

System Protection



The processes in an operating system must be protected from one another's activities. To provide such protection, we can use various mechanisms to ensure that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources of a system.

Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specifying the controls to be imposed, together with a means of enforcement. We distinguish between protection and security, which is a measure of confidence that the integrity of a system and its data will be preserved. Security assurance is a much broader topic than is protection, and we address it in Chapter 18.

17.1 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 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. 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, so that application designers can use them 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.

17.2

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 to fail, for example, but should not allow the execution of code from the 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, systems 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 backup 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 access control lists, as described in Section 10.6.2 and 17.6.

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.

17.3 [illegible]

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, a CPU can only be executed on. 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 where 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 discussed in Section 17.2 in that the goals of protection are to minimize the risks of possible security violations.

17.3.1 Domain Structure

To facilitate this scheme, 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*, *rights-set*>. For example, if domain D has the access right <*file F*, {read,write}>, 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 do not need to be disjoint; they may share access rights. For example, in Figure 17.1, we have three domains: D_1, D_2, and D_3. The access right <O_4, {print}> is shared by D_2 and D_3, implying that a process executing in either of these two domains can print object O_4. Note that a process must be executing in domain D_1 to read and write object O_1, while only processes in domain D_3 may execute object O_1.

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. Thus, the need-to-know principle is violated. We must allow the contents of a domain to be modified so that it 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.

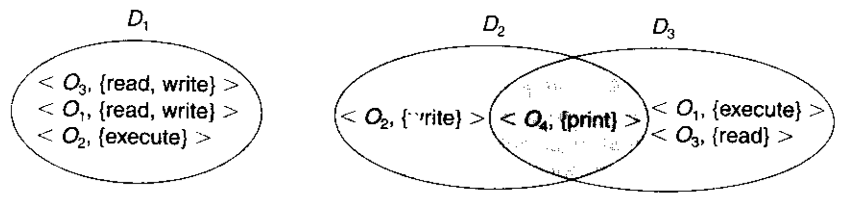


Figure 17.1 System with three protection domains.

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.

We discuss domain switching in greater detail in Section 17.4.

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 these concepts have been implemented there.

17.3.2 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 user ID is set to that of the owner of the file; when the bit is *off*, however, the user ID 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 to that 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 user IDs 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 user ID to change when the program was run. The user ID would change to that of a user with network access privilege (such as *root*, the most powerful user ID). One problem with this method is that if a user manages to create a file with user ID *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 other operating systems is to place privileged programs in a special directory. The operating system would be designed to change the user ID of any program run from this directory, either to the equivalent of *root* or to the user ID of the owner of the directory. This eliminates one security problem with setuid programs in which crackers create and hide (using obscure file or directory names) them for later use. 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 user ID. 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 user ID. 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. We discuss security in Chapter 18.

17.3.3 An Example: MULTICS

In the MULTICS system, the protection domains are organized hierarchically into a ring structure. Each ring corresponds to a single domain (Figure 17.2). The rings are numbered from 0 to 7. Let D_i and D_j be any two domain rings. If $j < i$, then D_i is a subset of D_j. That is, a process executing in domain D_j has more privileges than does a process executing in domain D_i. A process executing in domain D_0 has the most privileges. If only two rings exist, this scheme is equivalent to the monitor–user mode of execution, where monitor mode corresponds to D_0 and user mode corresponds to D_1.

MULTICS has a segmented address space; each segment is a file, and each segment is associated with one of the rings. A segment description includes an entry that identifies the ring number. In addition, it includes three access bits to control reading, writing, and execution. The association between segments and rings is a policy decision with which we are not concerned here.

A *current-ring-number* counter is associated with each process, identifying the ring in which the process is executing currently. When a process is executing in ring i, it cannot access a segment associated with ring j ($j < i$). It can access a segment associated with ring k ($k \geq i$). The type of access, however, is restricted according to the access bits associated with that segment.

Domain switching in MULTICS occurs when a process crosses from one ring to another by calling a procedure in a different ring. Obviously, this switch must be done in a controlled manner; otherwise, a process could start executing in ring 0, and no protection would be provided. To allow controlled domain

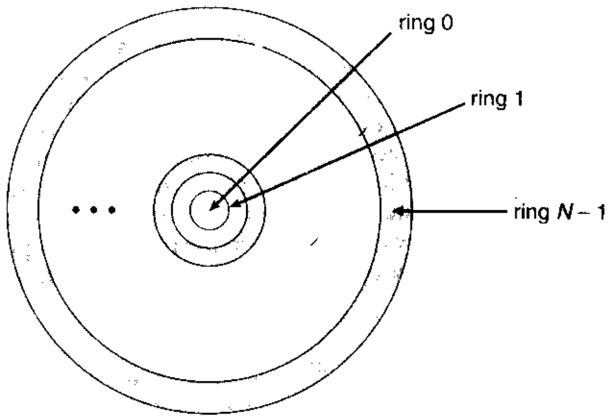


Figure 17.2 MULTICS ring structure.

switching, we modify the ring field of the segment descriptor to include the following:

- **Access bracket.** A pair of integers, *b1* and *b2*, such that $b1 \leq b2$.
- **Limit.** An integer *b3* such that $b3 > b2$.
- **List of gates.** Identifies the entry points (or **gates**) at which the segments may be called.

If a process executing in ring *i* calls a procedure (or segment) with access bracket (*b1*,*b2*), then the call is allowed if $b1 \leq i \leq b2$, and the current ring number of the process remains *i*. Otherwise, a trap to the operating system occurs, and the situation is handled as follows:

- If $i < b1$, then the call is allowed to occur, because we have a transfer to a ring (or domain) with fewer privileges. However, if parameters are passed that refer to segments in a lower ring (that is, segments not accessible to the called procedure), then these segments must be copied into an area that can be accessed by the called procedure.
- If $i > b2$, then the call is allowed to occur only if *b3* is greater than or equal to *i* and the call has been directed to one of the designated entry points in the list of gates. This scheme allows processes with limited access rights to call procedures in lower rings that have more access rights, but only in a carefully controlled manner.

The main disadvantage of the ring (or hierarchical) structure is that it does not allow us to enforce the need-to-know principle. In particular, if an object must be accessible in domain D_j but not accessible in domain D_i, then we must have $j < i$. But this requirement means that every segment accessible in D_i is also accessible in D_j.

The MULTICS protection system is generally more complex and less efficient than are those used in current operating systems. If protection interferes with the ease of use of the system or significantly decreases system performance, then its use must be weighed carefully against the purpose of the system. For instance, we would want to have a complex protection system on a computer used by a university to process students' grades and also used by students for classwork. A similar protection system would not be suited to a computer being used for number crunching, in which performance is of utmost importance. We would prefer to separate the mechanism from the protection policy, allowing the same system to have complex or simple protection depending on the needs of its users. To separate mechanism from policy, we require a more general model of protection.

17.4 [illegible]

Our 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 D_i can invoke on object O_j.

To illustrate these concepts, we consider the access matrix shown in Figure 17.3. There are four domains and four objects—three files (F_1, F_2, F_3) and one laser printer. A process executing in domain D_1 can read files F_1 and F_3. A process executing in domain D_4 has the same privileges as one executing in domain D_1; but in addition, it can also write onto files F_1 and F_3. Note that the laser printer can be accessed only by a process executing in domain D_2.

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 indeed hold. More specifically, we must ensure that a process executing in domain D_i 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

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 17.3 Access matrix.

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 O_j, the column O_j 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 the 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 may 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. Domain switching from domain D_i to domain D_j is allowed if and only if the access right `switch` $\in$ `access`(i, j). Thus, in Figure 17.4, a process executing in domain D_2 can switch to domain D_3 or to domain D_4. A process in domain D_4 can switch to D_1, and one in domain D_1 can switch to domain D_2.

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 copying of the access right only within the column (that is, for the object) for which the right is defined. For example, in Figure 17.5(a), a process executing in domain D_2 can copy the read operation into any

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure 17.4 Access matrix of Figure 17.3 with domains as objects.

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Figure 17.5 Access matrix with *copy* rights.

entry associated with file F_2. Hence, the access matrix of Figure 17.5(a) can be modified to the access matrix shown in Figure 17.5(b).

This scheme has two variants:

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

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 D_k 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 D_i can add and remove any right in any entry in column j. For example, in Figure 17.6(a), domain D_1 is the owner of F_1 and thus can add and delete any valid right in column F_1. Similarly, domain D_2 is the owner of F_2 and F_3 and thus can add and remove any valid right within these two columns. Thus, the access matrix of Figure 17.6(a) can be modified to the access matrix shown in Figure 17.6(b).

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 D_i can remove any access right from row j. For example, suppose that, in Figure 17.4, we include the *control* right in

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write*

(b)

Figure 17.6 Access matrix with *owner* rights.

access(D_2, D_4). Then, a process executing in domain D_2 could modify domain D_4, as shown in Figure 17.7.

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 Bibliographical Notes for references).

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 the implementation and control of 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 is here; system designers and users must make the policy decisions concerning which domains are to have access to which objects in which ways.

17.5

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

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Figure 17.7 Modified access matrix of Figure 17.4.

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.

17.5.1 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 O_j within domain D_i, the global table is searched for a triple $<D_i, O_j, R_k>$, with $M \in R_k$. 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, it must have a separate entry in every domain.

17.5.2 Access Lists for Objects

Each column in the access matrix can be implemented as an access list for one object, as described in Section 10.6.2. 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 O_j is attempted in domain D_i, we search the access list for object O_j, looking for an entry $<D_i, R_k>$ with $M \in R_k$. 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.

17.5.3 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 O_j, the process executes the operation M, specifying the capability (or pointer) for object O_j 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 multiprogrammed computer systems came of age. The idea of an inherently protected pointer provides a foundation for protection that can be extended up to the applications 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 its type either as a capability or as 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 1 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 (Section 8.6) is useful to support this approach.

Several capability-based protection systems have been developed; we describe them briefly in Section 17.8. The Mach operating system also uses a version of capability-based protection; it is described in Appendix B.

17.5.4 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.

17.5.5 Comparison

We now compare the various techniques for implementing an access matrix. 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 the operations allowed. However, because access-rights 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; they are useful, however, 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 (Section 17.7).

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 (Section 17.7).

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

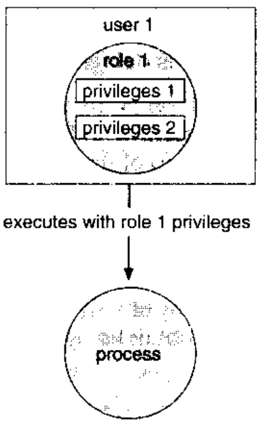


Figure 17.8 Role-based access control in Solaris 10.

violation by comparing the requested operation with the capability in the file-table entry.

17.6 A[illegible]

In Section 10.6.2, we described how access controls can be used on files within a file system. Each file and directory are 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 Sun Microsystems 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 17.8. This implementation of privileges decreases the security risk associated with superusers and setuid programs.

Notice that this facility is similar to the access matrix described in Section 17.4. This relationship will be further explored in the exercises at the end of the chapter.

17.7 Revocation of Access Rights

In a dynamic protection system, we may sometimes need to revoke access rights to objects shared by different users. Various questions about revocation may arise:

- **Immediate versus delayed**. Does revocation occur immediately, or is it delayed? If revocation is delayed, can we find out when it will take place?
- **Selective versus general**. When an access right to an object is revoked, does it affect *all* the users who have an access right to that object, or can we specify a select group of users whose access rights should be revoked?
- **Partial versus total**. Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?
- **Temporary versus permanent**. Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again?

With an access-list scheme, revocation is easy. The access list is searched for any access rights to be revoked, and they are deleted from the list. Revocation is immediate and can be general or selective, total or partial, and permanent or temporary.

Capabilities, however, present a much more difficult revocation problem. Since the capabilities are distributed throughout the system, we must find them before we can revoke them. Schemes that implement revocation for capabilities include the following:

- **Reacquisition**. Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.
- **Back-pointers**. A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary. This scheme was adopted in the MULTICS system. It is quite general, but its implementation is costly.
- **Indirection**. The capabilities point indirectly, not directly, to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. Then, when an access is attempted, the capability is found to point to an illegal table entry. Table entries can be reused for other capabilities without difficulty, since both the capability and the table entry contain the unique name of the object. The object for a capability and its table entry must match. This scheme was adopted in the CAL system. It does not allow selective revocation.
- **Keys**. A key is a unique bit pattern that can be associated with a capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process owning the capability. A **master key** is associated with each object; it can be defined or replaced with the `set-key` operation. When a capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared with the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition

is raised. Revocation replaces the master key with a new value via the `set-key` operation, invalidating all previous capabilities for this object.

This scheme does not allow selective revocation, since only one master key is associated with each object. If we associate a list of keys with each object, then selective revocation can be implemented. Finally, we can group all keys into one global table of keys. A capability is valid only if its key matches some key in the global table. We implement revocation by removing the matching key from the table. With this scheme, a key can be associated with several objects, and several keys can be associated with each object, providing maximum flexibility.

In key-based schemes, the operations of defining keys, inserting them into lists, and deleting them from lists should not be available to all users. In particular, it would be reasonable to allow only the owner of an object to set the keys for that object. This choice, however, is a policy decision that the protection system can implement but should not define.

17.8 Capability-Based Systems

In this section, we survey two capability-based protection systems. These systems vary in their complexity and in the types of policies that can be implemented on them. Neither system is widely used, but they are interesting proving grounds for protection theories.

17.8.1 An Example: Hydra

Hydra is a capability-based protection system that provides considerable flexibility. A fixed set of possible access rights is known to and interpreted by the system. These rights include such basic forms of access as the right to read, write, or execute a memory segment. In addition, a user (of the protection system) can declare other rights. The interpretation of user-defined rights is performed solely by the user's program, but the system provides access protection for the use of these rights, as well as for the use of system-defined rights. These facilities constitute a significant development in protection technology.

Operations on objects are defined procedurally. The procedures that implement such operations are themselves a form of object, and they are accessed indirectly by capabilities. The names of user-defined procedures must be identified to the protection system if it is to deal with objects of the user-defined type. When the definition of an object is made known to Hydra, the names of operations on the type become **auxiliary rights**. Auxiliary rights can be described in a capability for an instance of the type. For a process to perform an operation on a typed object, the capability it holds for that object must contain the name of the operation being invoked among its auxiliary rights. This restriction enables discrimination of access rights to be made on an instance-by-instance and process-by-process basis.

Hydra also provides **rights amplification**. This scheme allows a procedure to be certified as *trustworthy* to act on a formal parameter of a specified type on behalf of any process that holds a right to execute the procedure. The rights held by a trustworthy procedure are independent of, and may exceed, the rights held by the calling process. However, such a procedure must not be regarded as universally trustworthy (the procedure is not allowed to act on other types, for instance), and the trustworthiness must not be extended to any other procedures or program segments that might be executed by a process.

Amplification allows implementation procedures access to the representation variables of an abstract data type. If a process holds a capability to a typed object *A*, for instance, this capability may include an auxiliary right to invoke some operation *P* but would not include any of the so-called kernel rights, such as read, write, or execute, on the segment that represents *A*. Such a capability gives a process a means of indirect access (through the operation *P*) to the representation of *A*, but only for specific purposes.

When a process invokes the operation *P* on an object *A*, however, the capability for access to *A* may be amplified as control passes to the code body of *P*. This amplification may be necessary to allow *P* the right to access the storage segment representing *A* so as to implement the operation that *P* defines on the abstract data type. The code body of *P* may be allowed to read or to write to the segment of *A* directly, even though the calling process cannot. On return from *P*, the capability for *A* is restored to its original, unamplified state. This case is a typical one in which the rights held by a process for access to a protected segment must change dynamically, depending on the task to be performed. The dynamic adjustment of rights is performed to guarantee consistency of a programmer-defined abstraction. Amplification of rights can be stated explicitly in the declaration of an abstract type to the Hydra operating system.

When a user passes an object as an argument to a procedure, we may need to ensure that the procedure cannot modify the object. We can implement this restriction readily by passing an access right that does not have the modification (write) right. However, if amplification may occur, the right to modify may be reinstated. Thus, the user-protection requirement can be circumvented. In general, of course, a user may trust that a procedure performs its task correctly. This assumption is not always correct, however, because of hardware or software errors. Hydra solves this problem by restricting amplifications.

The procedure-call mechanism of Hydra was designed as a direct solution to the *problem of mutually suspicious subsystems*. This problem is defined as follows. Suppose that a program is provided that can be invoked as a service by a number of different users (for example, a sort routine, a compiler, a game). When users invoke this service program, they take the risk that the program will malfunction and will either damage the given data or retain some access right to the data to be used (without authority) later. Similarly, the service program may have some private files (for accounting purposes, for example) that should not be accessed directly by the calling user program. Hydra provides mechanisms for directly dealing with this problem.

A Hydra subsystem is built on top of its protection kernel and may require protection of its own components. A subsystem interacts with the kernel through calls on a set of kernel-defined primitives that define access rights to resources defined by the subsystem. The subsystem designer can define policies for use of these resources by user processes, but the policies are enforceable by use of the standard access protection afforded by the capability system.

A programmer can make direct use of the protection system after acquainting herself with its features in the appropriate reference manual. Hydra provides a large library of system-defined procedures that can be called by user programs. A user of the Hydra system would explicitly incorporate calls on these system procedures into the code of her programs or would use a program translator that had been interfaced to Hydra.

17.8.2 An Example: Cambridge CAP System

A different approach to capability-based protection has been taken in the design of the Cambridge CAP system. CAP's capability system is simpler and superficially less powerful than that of Hydra. However, closer examination shows that it, too, can be used to provide secure protection of user-defined objects. CAP has two kinds of capabilities. The ordinary kind is called a **data capability**. It can be used to provide access to objects, but the only rights provided are the standard read, write, and execute of the individual storage segments associated with the object. Data capabilities are interpreted by microcode in the CAP machine.

The second kind of capability is the so-called **software capability**, which is protected, but not interpreted, by the CAP microcode. It is interpreted by a *protected* (that is, a privileged) procedure, which may be written by an application programmer as part of a subsystem. A particular kind of rights amplification is associated with a protected procedure. When executing the code body of such a procedure, a process temporarily acquires the right to read or write the contents of a software capability itself. This specific kind of rights amplification corresponds to an implementation of the `seal` and `unseal` primitives on capabilities. Of course, this privilege is still subject to type verification to ensure that only software capabilities for a specified abstract type are passed to any such procedure. Universal trust is not placed in any code other than the CAP machine's microcode. (See Bibliographical Notes for references.)

The interpretation of a software capability is left completely to the subsystem, through the protected procedures it contains. This scheme allows a variety of protection policies to be implemented. Although a programmer can define her own protected procedures (any of which might be incorrect), the security of the overall system cannot be compromised. The basic protection system will not allow an unverified, user-defined, protected procedure access to any storage segments (or capabilities) that do not belong to the protection environment in which it resides. The most serious consequence of an insecure protected procedure is a protection breakdown of the subsystem for which that procedure has responsibility.

The designers of the CAP system have noted that the use of software capabilities allowed them to realize considerable economies in formulating and implementing protection policies commensurate with the requirements of abstract resources. However, a subsystem designer who wants to make use of this facility cannot simply study a reference manual, as is the case with Hydra. Instead, she must learn the principles and techniques of protection, since the system provides her with no library of procedures.

17.9 Language-Based Protection

To the degree that protection is provided in existing computer systems, it is usually achieved through an operating-system kernel, which acts as a security agent to inspect and validate each attempt to access a protected resource. Since comprehensive access validation is potentially a source of considerable overhead, either we must give it hardware support to reduce the cost of each validation or we must accept that the system designer may compromise the goals of protection. Satisfying all these goals is difficult if the flexibility to implement protection policies is restricted by the support mechanisms provided or if protection environments are made larger than necessary to secure greater operational efficiency.

As operating systems have become more complex, and particularly as they have attempted to provide higher-level user interfaces, the goals of protection have become much more refined. The designers of protection systems have drawn heavily on ideas that originated in programming languages and especially on the concepts of abstract data types and objects. Protection systems are now concerned not only with the identity of a resource to which access is attempted but also with the functional nature of that access. In the newest protection systems, concern for the function to be invoked extends beyond a set of system-defined functions, such as standard file-access methods, to include functions that may be user-defined as well.

Policies for resource use may also vary, depending on the application, and they may be subject to change over time. For these reasons, protection can no longer be considered a matter of concern to only the designer of an operating system. It should also be available as a tool for use by the application designer, so that resources of an applications subsystem can be guarded against tampering or the influence of an error.

17.9.1 Compiler-Based Enforcement

At this point, programming languages enter the picture. Specifying the desired control of access to a shared resource in a system is making a declarative statement about the resource. This kind of statement can be integrated into a language by an extension of its typing facility. When protection is declared along with data typing, the designer of each subsystem can specify its requirements for protection, as well as its need for use of other resources in a system. Such a specification should be given directly as a program is composed, and in the language in which the program itself is stated. This approach has several significant advantages:

- Protection needs are simply declared, rather than programmed as a sequence of calls on procedures of an operating system.
- Protection requirements can be stated independently of the facilities provided by a particular operating system.
- The means for enforcement need not be provided by the designer of a subsystem.
- A declarative notation is natural because access privileges are closely related to the linguistic concept of data type.

A variety of techniques can be provided by a programming-language implementation to enforce protection, but any of these must depend on some degree of support from an underlying machine and its operating system. For example, suppose a language is used to generate code to run on the Cambridge CAP system. On this system, every storage reference made on the underlying hardware occurs indirectly through a capability. This restriction prevents any process from accessing a resource outside of its protection environment at any time. However, a program may impose arbitrary restrictions on how a resource can be used during execution of a particular code segment. We can implement such restrictions most readily by using the software capabilities provided by CAP. A language implementation might provide standard protected procedures to interpret software capabilities that would realize the protection policies that could be specified in the language. This scheme puts policy specification at the disposal of the programmers, while freeing them from implementing its enforcement.

Even if a system does not provide a protection kernel as powerful as those of Hydra or CAP, mechanisms are still available for implementing protection specifications given in a programming language. The principal distinction is that the *security* of this protection will not be as great as that supported by a protection kernel, because the mechanism must rely on more assumptions about the operational state of the system. A compiler can separate references for which it can certify that no protection violation could occur from those for which a violation might be possible, and it can treat them differently. The security provided by this form of protection rests on the assumption that the code generated by the compiler will not be modified prior to or during its execution.

What, then, are the relative merits of enforcement based solely on a kernel, as opposed to enforcement provided largely by a compiler?

- **Security**. Enforcement by a kernel provides a greater degree of security of the protection system itself than does the generation of protection-checking code by a compiler. In a compiler-supported scheme, security rests on correctness of the translator, on some underlying mechanism of storage management that protects the segments from which compiled code is executed, and, ultimately, on the security of files from which a program is loaded. Some of these considerations also apply to a software-supported protection kernel, but to a lesser degree, since the kernel may reside in fixed physical storage segments and may be loaded from only a designated file. With a tagged-capability system, in which all address computation is performed either by hardware or by a fixed microprogram, even greater security is possible. Hardware-supported protection is also relatively immune to protection violations that might occur as a result of either hardware or system software malfunction.

- **Flexibility**. There are limits to the flexibility of a protection kernel in implementing a user-defined policy, although it may supply adequate facilities for the system to provide enforcement of its own policies. With a programming language, protection policy can be declared and enforcement provided as needed by an implementation. If a language does not provide sufficient flexibility, it can be extended or replaced with less disturbance of a system in service than would be caused by the modification of an operating-system kernel.

- **Efficiency**. The greatest efficiency is obtained when enforcement of protection is supported directly by hardware (or microcode). Insofar as software support is required, language-based enforcement has the advantage that static access enforcement can be verified off-line at compile time. Also, since an intelligent compiler can tailor the enforcement mechanism to meet the specified need, the fixed overhead of kernel calls can often be avoided.

In summary, the specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources. A language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable. In addition, it can interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.

One way of making protection available to the application program is through the use of a software capability that could be used as an object of computation. Inherent in this concept is the idea that certain program components might have the privilege of creating or examining these software capabilities. A capability-creating program would be able to execute a primitive operation that would seal a data structure, rendering the latter's contents inaccessible to any program components that did not hold either the seal or the unseal privilege. They might copy the data structure or pass its address to other program components, but they could not gain access to its contents. The reason for introducing such software capabilities is to bring a protection mechanism into the programming language. The only problem with the concept as proposed is that the use of the `seal` and `unseal` operations takes a procedural approach to specifying protection. A nonprocedural or declarative notation seems a preferable way to make protection available to the application programmer.

What is needed is a safe, dynamic access-control mechanism for distributing capabilities to system resources among user processes. To contribute to the overall reliability of a system, the access-control mechanism should be safe to use. To be useful in practice, it should also be reasonably efficient. This requirement has led to the development of a number of language constructs that allow the programmer to declare various restrictions on the use of a specific managed resource. (See the Bibliographical Notes for appropriate references.) These constructs provide mechanisms for three functions:

1. Distributing capabilities safely and efficiently among customer processes: In particular, mechanisms ensure that a user process will use the managed resource only if it was granted a capability to that resource.

2. Specifying the type of operations that a particular process may invoke on an allocated resource (for example, a reader of a file should be allowed only to read the file, whereas a writer should be able both to read and to write): It should not be necessary to grant the same set of rights to every user process, and it should be impossible for a process to enlarge

its set of access rights, except with the authorization of the access-control mechanism.

- Specifying the order in which a particular process may invoke the various operations of a resource (for example, a file must be opened before it can be read): It should be possible to give two processes different restrictions on the order in which they can invoke the operations of the allocated resource.

The incorporation of protection concepts into programming languages, as a practical tool for system design, is in its infancy. Protection will likely become a matter of greater concern to the designers of new systems with distributed architectures and increasingly stringent requirements on data security. Then the importance of suitable language notations in which to express protection requirements will be recognized more widely.

17.9.2 Protection in Java

Because Java was designed to run in a distributed environment, the Java virtual machine—or JVM—has many built-in protecion mechanisms. Java programs are composed of **classes**, each of which is a collection of data fields and functions (called **methods**) that operate on those fields. The JVM loads a class in response to a request to create instances (or objects) of that class. One of the most novel and useful features of Java is its support for dynamically loading untrusted classes over a network and for executing mutually distrusting classes within the same JVM.

Because of these capabilities of Java, protection is a paramount concern. Classes running in the same JVM may be from different sources and may not be equally trusted. As a result, enforcing protection at the granularity of the JVM process is insufficient. Intuitively, whether a request to open a file should be allowed will generally depend on which class has requested the open. The operating system lacks this knowledge.

Thus, such protection decisions are handled within the JVM. When the JVM loads a class, it assigns the class to a protection domain that gives the permissions of that class. The protection domain to which the class is assigned depends on the URL from which the class was loaded and any digital signatures on the class file. (Digital signatures are covered in Section 18.4.1.3.) A configurable policy file determines the permissions granted to the domain (and its classes). For example, classes loaded from a trusted server might be placed in a protection domain that allows them to access files in the user's home directory, whereas classes loaded from an untrusted server might have no file access permissions at all.

It can be complicated for the JVM to determine what class is responsible for a request to access a protected resource. Accesses are often performed indirectly, through system libraries or other classes. For example, consider a class that is not allowed to open network connections. It could call a system library to request the load of the contents of a URL. The JVM must decide whether or not to open a network connection for this request. But which class should be used to determine if the connection should be allowed, the application or the system library?

The philosophy adopted in Java is to require the library class to explicitly permit the network connection to load the requested URL. More generally, in order to access a protected resource, some method in the calling sequence that resulted in the request must explicitly assert the privilege to access the resource. By doing so, this method *takes responsibility* for the request; presumably, it will also perform whatever checks are necessary to ensure the safety of the request. Of course, not every method is allowed to assert a privilege; a method can assert a privilege only if its class is in a protection domain that is itself allowed to exercise the privilege.

This implementation approach is called **stack inspection**. Every thread in the JVM has an associated stack of its ongoing method invocations. When its caller may not be trusted, a method executes an access request within a `doPrivileged` block to perform the access to a protected resource directly or indirectly. `doPrivileged()` is a static method in the `AccessController` class that is passed a class with a `run()` method to invoke. When the `doPrivileged` block is entered, the stack frame for this method is annotated to indicate this fact. Then, the contents of the block are executed. When an access to a protected resource is subsequently requested, either by this method or a method it calls, a call to `checkPermissions()` is used to invoke stack inspection to determine if the request should be allowed. The inspection examines stack frames on the calling thread's stack, starting from the most recently added frame and working toward the oldest. If a stack frame is first found that has the `doPrivileged()` annotation, then `checkPermissions()` returns immediately and silently, allowing the access. If a stack frame is first found for which access is disallowed based on the protection domain of the method's class, then `checkPermissions()` throws an `AccessControlException`. If the stack inspection exhausts the stack without finding either type of frame, then whether access is allowed depends on the implementation (for example, some implementations of the JVM may allow access, other implementations may disallow it).

Stack inspection is illustrated in Figure 17.9. Here, the `gui()` method of a class in the *untrusted applet* protection domain performs two operations, first a `get()` and then an `open()`. The former is an invocation of the

protection domain:	**untrusted applet**	**URL loader**	**networking**
socket permission:	none	*.lucent.com:80, connect	any
class:	gui: … get(url); open(addr); …	get(URL u): … doPrivileged { open('proxy.lucent.com:80'); } <request u from proxy> …	open(Addr a): … checkPermission (a, connect); connect (a); …

Figure 17.9 Stack inspection.

get() method of a class in the *URL loader* protection domain, which is permitted to open() sessions to sites in the lucent.com domain, in particular a proxy server proxy.lucent.com for retrieving URLs. For this reason, the untrusted applet's get() invocation will succeed: the checkPermissions() call in the networking library encounters the stack frame of the get() method, which performed its open() in a doPrivileged block. However, the untrusted applet's open() invocation will result in an exception, because the checkPermissions() call finds no doPrivileged annotation before encountering the stack frame of the gui() method.

Of course, for stack inspection to work, a program must be unable to modify the annotations on its own stack frame or to do other manipulations of stack inspection. This is one of the most important differences between Java and many other languages (including C++). A Java program cannot directly access memory. Rather, it can manipulate only an object for which it has a reference. References cannot be forged, and the manipulations are made only through well-defined interfaces. Compliance is enforced through a sophisticated collection of load-time and run-time checks. As a result, an object cannot manipulate its run-time stack, because it cannot get a reference to the stack or other components of the protection system.

More generally, Java's load-time and run-time checks enforce **type safety** of Java classes. Type safety ensures that classes cannot treat integers as pointers, write past the end of an array, or otherwise access memory in arbitrary ways. Rather, a program can access an object only via the methods defined on that object by its class. This is the foundation of Java protection, since it enables a class to effectively **encapsulate** and protect its data and methods from other classes loaded in the same JVM. For example, a variable can be defined as private so that only the class that contains it can access it or protected so that it can be accessed only by the class that contains it, subclasses of that class, or classes in the same package. Type safety ensures that these restrictions can be enforced.

17.10 [illegible]

Computer systems contain many objects, and they need to be protected from misuse. Objects may be hardware (such as memory, CPU time, and I/O devices) or software (such as files, programs, and semaphores). An access right is permission to perform an operation on an object. A domain is a set of access rights. Processes execute in domains and may use any of the access rights in the domain to access and manipulate objects. During its lifetime, a process may be either bound to a protection domain or allowed to switch from one domain to another.

The access matrix is a general model of protection that provides a mechanism for protection without imposing a particular protection policy on the system or its users. The separation of policy and mechanism is an important design property.

The access matrix is sparse. It is normally implemented either as access lists associated with each object or as capability lists associated with each domain. We can include dynamic protection in the access-matrix model by considering domains and the access matrix itself as objects. Revocation of access rights in a dynamic protection model is typically easier to implement with an access-list scheme than with a capability list.

Real systems are much more limited than the general model and tend to provide protection only for files. UNIX is representative, providing read, write, and execution protection separately for the owner, group, and general public for each file. MULTICS uses a ring structure in addition to file access. Hydra, the Cambridge CAP system, and Mach are capability systems that extend protection to user-defined software objects. Solaris 10 implements the principle of least privilege via role-based access control, a form of the access matrix.

Language-based protection provides finer-grained arbitration of requests and privileges than the operating system is able to provide. For example, a single Java JVM can run several threads, each in a different protection class. It enforces the resource requests through sophisticated stack inspection and via the type safety of the language.

[illegible]

17.1 Consider the ring protection scheme in MULTICS. If we were to implement the system calls of a typical operating system and store them in a segment associated with ring 0, what should be the values stored in the ring field of the segment descriptor? What happens during a system call when a process executing in a higher-numbered ring invokes a procedure in ring 0?

17.2 The access-control matrix could be used to determine whether a process can switch from, say, domain A to domain B and enjoy the access privileges of domain B. Is this approach equivalent to including the access privileges of domain B in those of domain A?

17.3 What hardware features are needed in a computer system for efficient capability manipulation? Can these be used for memory protection?

17.4 Discuss the strengths and weaknesses of implementing an access matrix using access lists that are associated with objects.

17.5 Explain why a capability-based system such as Hydra provides greater flexibility than the ring protection scheme in enforcing protection policies.

17.6 Discuss the need for rights amplification in Hydra. How does this practice compare with the cross-ring calls in a ring protection scheme?

17.7 Discuss which of the following systems allow module designers to enforce the need-to-know principle.

a. The MULTICS ring protection scheme

b. Hydra's capabilities

c. JVM's stack-inspection scheme

17.8 Describe how the Java protection model would be sacrificed if a Java program were allowed to directly alter the annotations of its stack frame.

17.9 How does the principle of least privilege aid in the creation of protection systems?

17.10 How can systems that implement the principle of least privilege still have protection failures that lead to security violations?

The access-matrix model of protection between domains and objects was developed by Lampson [1969] and Lampson [1971]. Popek [1974] and Saltzer and Schroeder [1975] provided excellent surveys on the subject of protection. Harrison et al. [1976] used a formal version of this model to enable them to prove properties of a protection system mathematically.

The concept of a capability evolved from Iliffe's and Jodeit's *codewords*, which were implemented in the Rice University computer (Iliffe and Jodeit [1962]). The term *capability* was introduced by Dennis and Horn [1966].

The Hydra system was described by Wulf et al. [1981]. The CAP system was described by Needham and Walker [1977]. Organick [1972] discussed the MULTICS ring protection system.

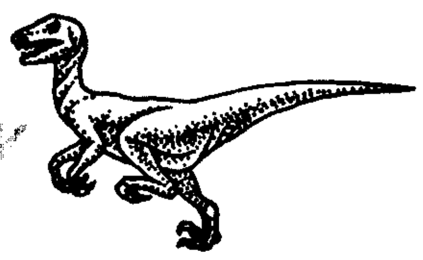
Revocation was discussed by Redell and Fabry [1974], Cohen and Jefferson [1975], and Ekanadham and Bernstein [1979]. The principle of separation of policy and mechanism was advocated by the designer of Hydra (Levin et al. [1975]). The confinement problem was first discussed by Lampson [1973] and was further examined by Lipner [1975].

The use of higher-level languages for specifying access control was suggested first by Morris [1973], who proposed the use of the seal and unseal operations discussed in Section 17.9. Kieburtz and Silberschatz [1978], Kieburtz and Silberschatz [1983], and McGraw and Andrews [1979] proposed various language constructs for dealing with general dynamic-resource-management schemes. Jones and Liskov [1978] considered how a static access-control scheme can be incorporated in a programming language that supports abstract data types. The use of minimal operating-system support to enforce protection was advocated by the Exokernel Project (Ganger et al. [2002], Kaashoek et al. [1997]). Extensibility of system code through language-based protection mechanisms was discussed in Bershad et al. [1995]. Other techniques for enforcing protection include sandboxing (Goldberg et al. [1996]) and software fault isolation (Wahbe et al. [1993b]). The issues of lowering the overhead associated with protection costs and enabling user-level access to networking devices were discussed in McCanne and Jacobson [1993] and Basu et al. [1995].

More detailed analyses of stack inspection, including comparisons with other approaches to Java security, can be found in Wallach et al. [1997] and Gong et al. [1997].

CHAPTER

System Security



Protection, as we discussed in Chapter 17, is strictly an *internal* problem: How do we provide controlled access to programs and data stored in a computer system? **Security**, on the other hand, requires not only an adequate protection system but also consideration of the *external* environment within which the system operates. A protection system is ineffective if user authentication is compromised or a program is run by an unauthorized user.

Computer resources must be guarded against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. These resources include information stored in the system (both data and code), as well as the CPU, memory, disks, tapes and networking that are the computer. In this chapter, we start by examining ways in which resources may be accidentally or purposefully misused. We then explore a key security enabler—cryptography. Finally, we look at mechanisms to guard against or detect attacks.

18.1 [illegible]

In many applications, ensuring the security of the computer system is worth considerable effort. Large commercial systems containing payroll or other financial data are inviting targets to thieves. Systems that contain data pertaining to corporate operations may be of interest to unscrupulous competitors. Furthermore, loss of such data, whether by accident or fraud, can seriously impair the ability of the corporation to function.

In Chapter 17, we discussed mechanisms that the operating system can provide (with appropriate aid from the hardware) that allow users to protect their resources, including programs and data. These mechanisms work well only as long as the users conform to the intended use of and access to these resources. We say that a system is **secure** if its resources are used and accessed as intended under all circumstances. Unfortunately, total security cannot be achieved. Nonetheless, we must have mechanisms to make security breaches a rare occurrence, rather than the norm.

Security violations (or misuse) of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse than against malicious misuse. For the most part, protection mechanisms are the core of protection from accidents. The following list includes forms of accidental and malicious security violations. We should note that in our discussion of security, we use the terms *intruder* and *cracker* for those attempting to breach security. In addition, a **threat** is the potential for a security violation, such as the discovery of a vulnerability, whereas an **attack** is the attempt to break security.

- **Breach of confidentiality**. This type of violation involves unauthorized reading of data (or theft of information). Typically, a breach of confidentiality is the goal of an intruder. Capturing secret data from a system or a data stream, such as credit-card information or identity information for identity theft, can result directly in money for the intruder.
- **Breach of integrity**. This violation involves unauthorized modification of data. Such attacks can, for example, result in passing of liability to an innocent party or modification of the source code of an important commercial application.
- **Breach of availability**. This violation involves unauthorized destruction of data. Some crackers would rather wreak havoc and gain status or bragging rights than gain financially. Web-site defacement is a common example of this type of security breach.
- **Theft of service**. This violation involves unauthorized use of resources. For example, an intruder (or intrusion program) may install a daemon on a system that acts as a file server.
- **Denial of service**. This violation involves preventing legitimate use of the system. **Denial-of-service**, or **DOS**, attacks are sometimes accidental. The original Internet worm turned into a DOS attack when a bug failed to delay its rapid spread. We discuss DOS attacks further in Section 18.3.3.

Attackers use several standard methods in their attempts to breach security. The most common is **masquerading**, in which one participant in a communication pretends to be someone else (another host or another person). By masquerading, attackers breach **authentication**, the correctness of identification; they can then can gain access that they would not normally be allowed or escalate their privileges—obtain privileges to which they would not normally be entitled. Another common attack is to replay a captured exchange of data. A **replay attack** consists of the malicious or fraudulent repeat of a valid data transmission. Sometimes the replay comprises the entire attack—for example, in a repeat of a request to transfer money. But frequently it is done along with **message modification**, again to escalate privileges. Consider the damage that could be done if a request for authentication had a legitimate user's information replaced with an unauthorized user's. Yet another kind of attack is the **man-in-the-middle attack**, in which an attacker sits in the data flow of a communication, masquerading as the sender to the receiver, and vice versa. In a network communication, a man-in-the-middle attack may be preceded by a **session hijacking**, in which an active communication session is intercepted. Several attack methods are depicted in Figure 18.1.

As we have already suggested, absolute protection of the system from malicious abuse is not possible, but the cost to the perpetrator can be made

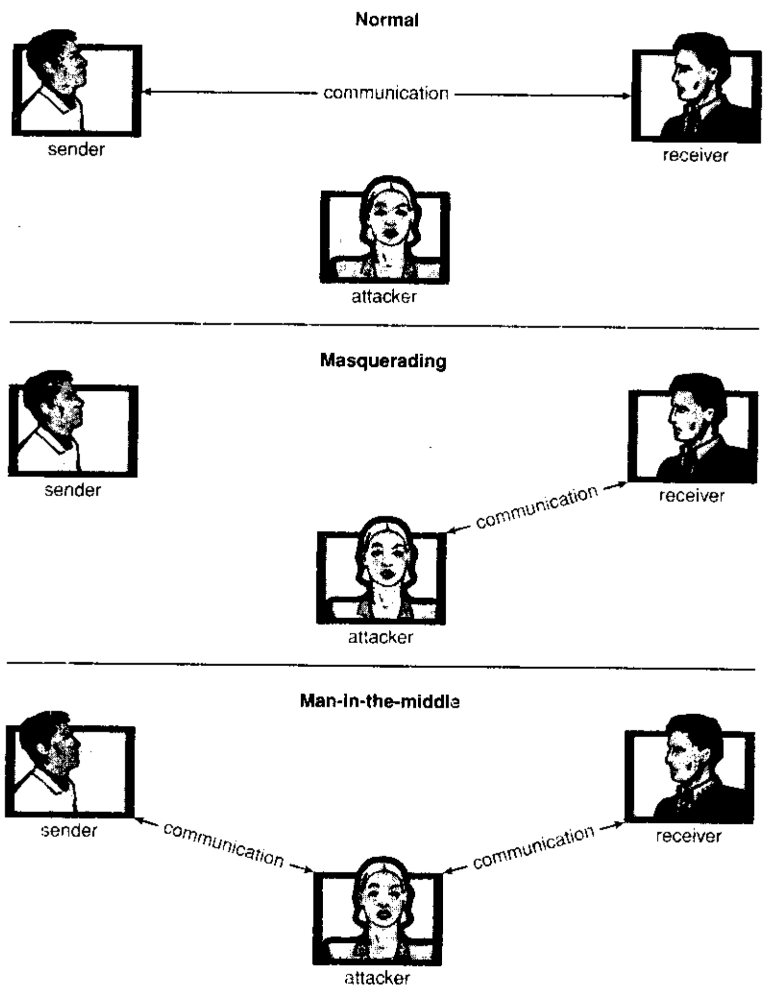


Figure 18.1 Standard security attacks.

sufficiently high to deter most intruders. In some cases, such as a denial-of-service attack, it is preferable to prevent the attack but sufficient to detect the attack so that countermeasures can be taken.

To protect a system, we must take security measures at four levels:

Physical. The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders. Both the machine rooms and the terminals or workstations that have access to the machines must be secured.

Human. Authorizing users must be done carefully to assure that only appropriate users have access to the system. Even authorized users, however, may be "encouraged" to let others use their access (in exchange for a bribe, for example). They may also be tricked into allowing access via **social engineering**. One type of social-engineering attack is **phishing**. Here, a legitimate-looking e-mail or web page misleads a user into entering confidential information. Another technique is **dumpster diving**, a general term for attempting to gather information in order to gain unauthorized access to the computer (by looking through trash, finding phone books, or finding notes containing passwords, for example). These security problems are management and personnel issues, not problems pertaining to operating systems.

3. **Operating system**. The system must protect itself from accidental or purposeful security breaches. A runaway process could constitute an accidental denial-of-service attack. A query to a service could reveal passwords. A stack-overflow could allow the launching of an unauthorized process. The list of possible breaches is almost endless.

4. **Network**. Much computer data in modern systems travels over private leased lines, shared lines like the Internet, wireless connections, or dial-up lines. Intercepting these data could be just as harmful as breaking into a computer; and interruption of communications could constitute a remote denial-of-service attack, diminishing users' use of and trust in the system.

Security at the first two levels must be maintained if operating-system security is to be ensured. A weakness at a high level of security (physical or human) allows circumvention of strict low-level (operating-system) security measures. Thus, the old adage that a chain is as weak as its weakest link is especially true of system security. All of these aspects must be addressed for security to be maintained.

Furthermore, the system must provide protection (Chapter 17) to allow the implementation of security features. Without the ability to authorize users and processes, to control their access, and to log their activities, it would be impossible for an operating system to implement security measures or to run securely. Hardware protection features are needed to support an overall protection scheme. For example, a system without memory protection cannot be secure. New hardware features are allowing systems to be made more secure, as we shall discuss.

Unfortunately, little in security is straightforward. As intruders exploit security vulnerabilities, security countermeasures are created and deployed. This causes intruders to become more sophisticated in their attacks. For example, recent security incidents include the use of spyware to provide a conduit for spam through innocent systems (we discuss this practice in Section 18.2). This cat-and-mouse game is likely to continue, with more security tools needed to block the escalating intruder techniques and activities.

In the remainder of this chapter, we address security at the network and operating-system levels. Security at the physical and human levels, although important, is for the most part beyond the scope of this text. Security within the operating system and between operating systems is implemented in several ways, ranging from passwords for authentication through guarding against viruses to detecting intrusions. We start with an exploration of security threats.

18.2 [illegible]

Processes, along with the kernel, are the only means of accomplishing work on a computer. Therefore, writing a program that creates a breach of security, or causing a normal process to change its behavior and create a breach, is a common goal of crackers. In fact, even most nonprogram security events have as their goal causing a program threat. For example, while it is useful to log in to a system without authorization, it is quite a lot more useful to leave behind a **back-door** daemon that provides information or allows easy access even if the original exploit is blocked. In this section, we describe common methods by which programs cause security breaches. Note that there is considerable variation in the naming conventions of security holes and that we use the most common or descriptive terms.

18.2.1 Trojan Horse

Many systems have mechanisms for allowing programs written by users to be executed by other users. If these programs are executed in a domain that provides the access rights of the executing user, the other users may misuse these rights. A text-editor program, for example, may include code to search the file to be edited for certain keywords. If any are found, the entire file may be copied to a special area accessible to the creator of the text editor. A code segment that misuses its environment is called a **Trojan horse**. Long search paths, such as are common on UNIX systems, exacerbate the Trojan-horse problem. The search path lists the set of directories to search when an ambiguous program name is given. The path is searched for a file of that name, and the file is executed. All the directories in such a search path must be secure, or a Trojan horse could be slipped into the user's path and executed accidentally.

For instance, consider the use of the "." character in a search path. The "." tells the shell to include the current directory in the search. Thus, if a user has "." in her search path, has set her current directory to a friend's directory, and enters the name of a normal system command, the command may be executed from the friend's directory instead. The program would run within the user's domain, allowing the program to do anything that the user is allowed to do, including deleting the user's files, for instance.

A variation of the Trojan horse is a program that emulates a login program. An unsuspecting user starts to log in at a terminal and notices that he has apparently mistyped his password. He tries again and is successful. What has happened is that his authentication key and password have been stolen by the login emulator, which was left running on the terminal by the thief. The emulator stored away the password, printed out a login error message, and exited; the user was then provided with a genuine login prompt. This type of attack can be defeated by having the operating system print a usage message at the end of an interactive session or by a non-trappable key sequence, such as the `control-alt-delete` combination used by all modern Windows operating systems.

Another variation on the Trojan horse is **spyware**. Spyware sometimes accompanies a program that the user has chosen to install. Most frequently, it comes along with freeware or shareware programs, but sometimes it is included with commercial software. The goal of spyware is to download ads to display on the user's system, create **pop-up browser windows** when certain sites are visited, or capture information from the user's system and return it to a central site. This latter mode is an example of a general category of attacks known as **covert channels**, in which surreptitious communication occurs. As a current example, the installation of an innocuous-seeming program on a Windows system could result in the loading of a spyware daemon. The spyware could contact a central site, be given a message and a list of recipient addresses, and deliver the spam message to those users from the Windows machine. This process continues until the user discovers the spyware. Frequently, the spyware is not discovered. In 2004, it was estimated that 80 percent of spam was being delivered by this method. This theft of service is not even considered a crime in most countries!

Spyware is a micro example of a macro problem: violation of the principle of least privilege. Under most circumstances, a user of an operating system does not need to install network daemons. Such daemons are installed via two mistakes. First, a user may run with more privileges than necessary (for example, as the administrator), allowing programs that she runs to have more access to the system than is necessary. This is a case of human error—a common security weakness. Second, an operating system may allow by default more privileges than a normal user needs. This is a case of poor operating-system design decisions. An operating system (and, indeed, software in general) should allow fine-grained control of access and security, but it must also be easy to manage and understand. Inconvenient or inadequate security measures are bound to be circumvented, causing an overall weakening of the security they were designed to implement.

18.2.2 Trap Door

The designer of a program or system might leave a hole in the software that only she is capable of using. This type of security breach (or **trap door**) was shown in the movie *War Games*. For instance, the code might check for a specific user ID or password, and it might circumvent normal security procedures. Programmers have been arrested for embezzling from banks by including rounding errors in their code and having the occasional half-cent credited to their accounts. This account crediting can add up to a large amount of money, considering the number of transactions that a large bank executes.

A clever trap door could be included in a compiler. The compiler could generate standard object code as well as a trap door, regardless of the source code being compiled. This activity is particularly nefarious, since a search of the source code of the program will not reveal any problems. Only the source code of the compiler would contain the information.

Trap doors pose a difficult problem because, to detect them, we have to analyze all the source code for all components of a system. Given that software systems may consist of millions of lines of code, this analysis is not done frequently, and frequently it is not done at all!

18.2.3 Logic Bomb

Consider a program that initiates a security incident only under certain circumstances. It would be hard to detect because under normal operations, there would be no security hole. However, when a predefined set of parameters were met, the security hole would be created. This scenario is known as a **logic bomb**. A programmer, for example, might write code to detect if she is still employed; if that check failed, a daemon could be spawned to allow remote access, or code could be launched to cause damage to the site.

18.2.4 Stack and Buffer Overflow

The stack- or buffer-overflow attack is the most common way for an attacker outside the system, on a network or dial-up connection, to gain unauthorized access to the target system. An authorized user of the system may also use this exploit for privilege escalation.

Essentially, the attack exploits a bug in a program. The bug can be a simple case of poor programming, in which the programmer neglected to code bounds checking on an input field. In this case, the attacker sends more data than the program was expecting. Using trial and error, or by examining the source code of the attacked program if it is available, the attacker determines the vulnerability and writes a program to do the following:

Overflow an input field, command-line argument, or input buffer—for example, on a network daemon—until it writes into the stack.

Overwrite the current return address on the stack with the address of the exploit code loaded in step 3.

Write a simple set of code for the next space in the stack that includes the commands that the attacker wishes to execute—for instance, spawn a shell.

The result of this attack program's execution will be a root shell or other privileged command execution.

For instance, if a web-page form expects a user name to be entered into a field, the attacker could send the user name, plus extra characters to overflow the buffer and reach the stack, plus a new return address to load onto the stack, plus the code the attacker wants to run. When the buffer-reading subroutine returns from execution, the return address is the exploit code, and the code is run.

Let's look at a buffer-overflow exploit in more detail. Consider the simple C program shown in Figure 18.2. This program creates a character array of size `BUFFER_SIZE` and copies the contents of the parameter provided on the command line—`argv[1]`. As long as the size of this parameter is less than `BUFFER_SIZE` (we need one byte to store the null terminator), this program works properly. But consider what happens if the parameter provided on the command line is longer than `BUFFER_SIZE`. In this scenario, the `strcpy()` function will begin copying from `argv[1]` until it encounters a null terminator (\0) or until the program crashes. Thus, this program suffers from a potential buffer-overflow problem in which copied data overflow the `buffer` array.

Note that a careful programmer could have performed bounds checking on the size of `argv[1]` by using the `strncpy()` function rather than `strcpy()`, replacing the line "`strcpy(buffer, argv[1]);`" with "`strncpy(buffer,`

```
#include <stdio.h>
#define BUFFER_SIZE 256

int main(int argc, char *argv[])
{
   char buffer[BUFFER_SIZE];

   if (argc < 2)
      return -1;
   else {
      strcpy(buffer,argv[1]);
      return 0;
   }
}
```

Figure 18.2 C program with buffer-overflow condition.

argv[1], sizeof(buffer)-1);". Unfortunately, good bounds checking is the exception rather than the norm.

Furthermore, lack of bounds checking is not the only possible cause of the behavior of the program in Figure 18.2. The program could instead have been carefully designed to compromise the integrity of the system. We now consider the possible security vulnerabilities of a buffer overflow.

When a function is invoked in a typical computer architecture, the variables defined locally to the function (sometimes known as **automatic variables**), the parameters passed to the function, and the address to which control returns once the function exits are stored in a **stack frame**. The layout for a typical stack frame is shown in Figure 18.3. Examining the stack frame from top to bottom, we first see the parameters passed to the function, followed by any automatic variables declared in the function. We next see the **frame pointer**, which is the address of the beginning of the stack frame. Finally, we have the return address, which specifies where to return control once the function exits. The frame pointer must be saved on the stack, as the value of the stack pointer can

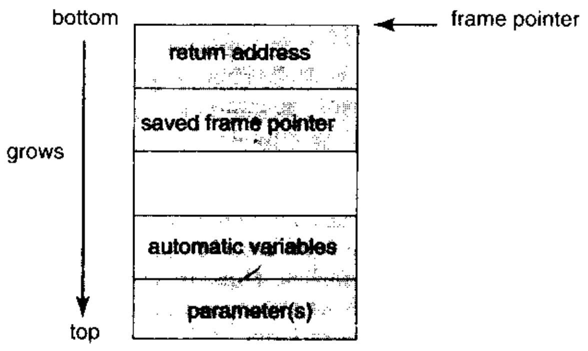


Figure 18.3 The layout for a typical stack frame.

vary during the function call; the saved frame pointer allows relative access to parameters and automatic variables.

Given this standard memory layout, a cracker could execute a buffer-overflow attack. Her goal is to replace the return address in the stack frame so that it now points to the code segment containing the attacking program.

The programmer first writes a short code segment such as the following:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    execvp(''\bin\sh'',''\bin \sh'', NULL);
    return 0;
}
```

Using the execvp() system call, this code segment creates a shell process. If the program being attacked runs with system-wide permissions, this newly created shell will gain complete access to the system. Of course, the code segment could do anything allowed by the privileges of the attacked process. This code segment is then compiled so that the assembly language instructions can be modified. The primary modification is to remove unnecessary features in the code, thereby reducing the code size so that it can fit into a stack frame. This assembled code fragment is now a binary sequence that will be at the heart of the attack.

Refer again to the program shown in Figure 18.2. Let's assume that when the main() function is called in that program, the stack frame appears as shown in Figure 18.4(a). Using a debugger, the programmer then finds the address of buffer[0] in the stack. That address is the location of the code the attacker wants executed, so the binary sequence is appended with the necessary

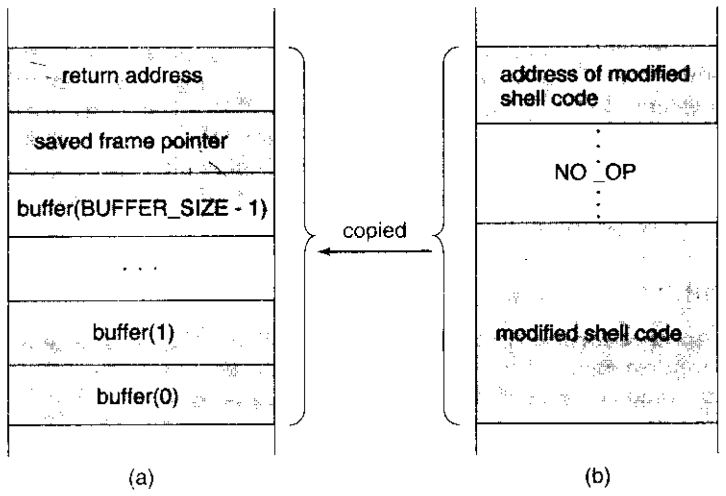


Figure 18.4 Hypothetical stack frame for Figure 18.2, (a) before and (b)

amount of NO-OP instructions (for `NO-OPeration`) to fill the stack frame up to the location of the return address; and the location of `buffer[0]`, the new return address, is added. The attack is complete when the attacker gives this constructed binary sequence as input to the process. The process then copies the binary sequence from `argv[1]` to position `buffer[0]` in the stack frame. Now, when control returns from `main()`, instead of returning to the location specified by the old value of the return address, we return to the modified shell code, which runs with the access rights of the attacked process! Figure 18.4(b) contains the modified shell code.

There are many ways to exploit potential buffer-overflow problems. In this example, we considered the possibility that the program being attacked—the code shown in Figure 18.2—ran with system-wide permissions. However, the code segment that runs once the value of the return address has been modified might perform any type of malicious act, such as deleting files, opening network ports for further exploitation, and so on.

This example buffer-overflow attack reveals that considerable knowledge and programming skill are needed to recognize exploitable code and then to exploit it. Unfortunately, it does not take great programmers to launch security attacks. Rather, one cracker can determine the bug and then write an exploit. Anyone with rudimentary computer skills and access to the exploit—a so-called **script kiddie**—can then try to launch the attack at target systems.

The buffer-overflow attack is especially pernicious because it can be run between systems and can travel over allowed communication channels. Such attacks can occur within protocols that are expected to be used to communicate with the target machine, and they can therefore be hard to detect and prevent. They can even bypass the security added by firewalls (Section 18.7).

One solution to this problem is for the CPU to have a feature that disallows execution of code in a stack section of memory. Recent versions of Sun's SPARC chip include this setting, and recent versions of Solaris enable it. The return address of the overflowed routine can still be modified; but when the return address is within the stack and the code there attempts to execute, an exception is generated, and the program is halted with an error.

Recent versions of AMD and Intel x86 chips include the NX feature to prevent this type of attack. The use of the feature is supported in several x86 operating systems, including Linux and Windows XP SP2. The hardware implementation involves the use of a new bit in the page tables of the CPUs. This bit marks the associated page as nonexecutable, disallowing instructions to be read from it and executed. As this feature becomes prevalent, buffer-overflow attacks should greatly diminish.

18.2.5 Viruses

Another form of program threat is a **virus**. Viruses are self-replicating and
to "infect" other programs. They can wreak havoc in a system
; or destroying files and causing system crashes and program
A virus is a fragment of code embedded in a legitimate program.
penetration attacks, viruses are very specific to architectures,
tems, and applications. Viruses are a particular problem for
after. JNIX and other multiuser operating systems generally are not
viruses because the executable programs are protected from

writing by the operating system. Even if a virus does infect such a program, its powers usually are limited because other aspects of the system are protected.

Viruses are usually borne via email, with spam the most common vector. They can also spread when users download viral programs from Internet file-sharing services or exchange infected disks.

Another common form of virus transmission uses Microsoft Office files, such as Microsoft Word documents. These documents can contain *macros* (or Visual Basic programs) that programs in the Office suite (Word, PowerPoint, and Excel) will execute automatically. Because these programs run under the user's own account, the macros can run largely unconstrained (for example, deleting user files at will). Commonly, the virus will also e-mail itself to others in the user's contact list. Here is a code sample that shows the simplicity of writing a Visual Basic macro that a virus could use to format the hard drive of a Windows computer as soon as the file containing the macro was opened:

```
Sub AutoOpen()
Dim oFS
    Set oFS = CreateObject(''Scripting.FileSystemObject'')
    vs = Shell(''c:
command.com /k format c:'',vbHide)
End Sub
```

How do viruses work? Once a virus reaches a target machine, a program known as a **virus dropper** inserts the virus onto the system. The virus dropper is usually a Trojan horse, executed for other reasons but installing the virus as its core activity. Once installed, the virus may do any one of a number of things. There are literally thousands of viruses, but they fall into several main categories. Note that many viruses belong to more than one category.

- **File**. A standard file virus infects a system by appending itself to a file. It changes the start of the program so that execution jumps to its code. After it executes, it returns control to the program so that its execution is not noticed. File viruses are sometimes known as parasitic viruses, as they leave no full files behind and leave the host program still functional.

- **Boot**. A boot virus infects the boot sector of the system, executing every time the system is booted and before the operating system is loaded. It watches for other bootable media (that is, floppy disks) and infects them. These viruses are also known as memory viruses, because they do not appear in the file system. Figure 18.5 shows how a boot virus works.

- **Macro**. Most viruses are written in a low-level language, such as assembly or C. Macro viruses are written in a high-level language, such as Visual Basic. These viruses are triggered when a program capable of executing the macro is run. For example, a macro virus could be contained in a spreadsheet file.

- **Source code**. A source code virus looks for source code and modifies it to include the virus and to help spread the virus.

- **Polymorphic**. This virus changes each time it is installed to avoid detection by antivirus software. The changes do not affect the virus's functionality

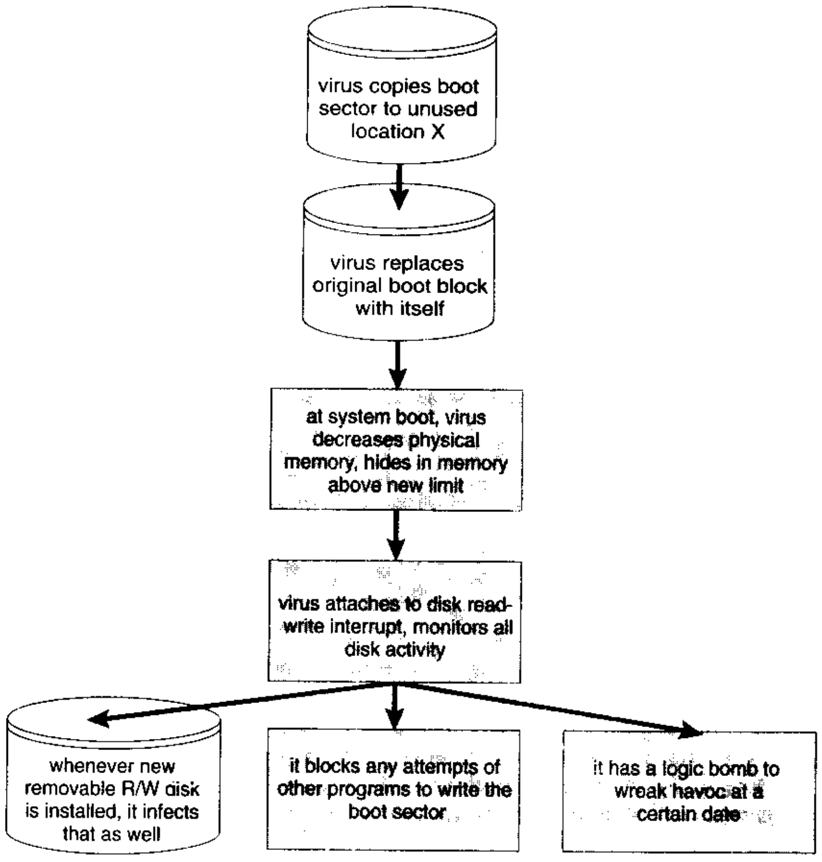

virus copies boot sector to unused location X
virus replaces original boot block with itself
at system boot, virus decreases physical memory, hides in memory above new limit
virus attaches to disk read-write interrupt, monitors all disk activity
whenever new removable R/W disk is installed, it infects that as well
it blocks any attempts of other programs to write the boot sector
it has a logic bomb to wreak havoc at a certain date

Figure 18.5 A boot-sector computer virus.

but rather change the virus's signature. A **virus signature** is a pattern that can be used to identify a virus, typically a series of bytes that make up the virus code.

- **Encrypted**. An encrypted virus includes decryption code along with the encrypted virus, again to avoid detection. The virus first decrypts and then executes.
- **Stealth**. This tricky virus attempts to avoid detection by modifying parts of the system that could be used to detect it. For example, it could modify the `read` system call so that if the file it has modified is read, the original form of the code is returned rather than the infected code.
- **Tunneling**. This virus attempts to bypass detection by an antivirus scanner by installing itself in the interrupt-handler chain. Similar viruses install themselves in device drivers.

Multipartite. A virus of this type is able to infect multiple parts of a system, including boot sectors, memory, and files. This makes it difficult to detect and contain.

Armored. An armored virus is coded to make itself hard for antivirus researchers to unravel and understand. It can also be compressed to avoid detection and disinfection. In addition, virus droppers and other full files that are part of a virus infestation are frequently hidden via file attributes or unviewable file names.

This vast variety of viruses is likely to continue to grow. In fact, in 2004 a new and widespread virus was detected. It exploited three separate bugs for its operation. This virus started by infecting hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server (IIS). Any vulnerable Microsoft Explorer web browser visiting those sites received a browser virus with any download. The browser virus installed several back-door programs, including a **keystroke logger**, which records all things entered on the keyboard (including passwords and credit-card numbers). It also installed a daemon to allow unlimited remote access by an intruder and another that allowed an intruder to route spam through the infected desktop computer.

Generally, viruses are the most disruptive security attack; and because they are effective, they will continue to be written and to spread. Among the active debates within the computing community is whether a **monoculture**, in which many systems run the same hardware, operating system, and/or application software, is increasing the threat of and damage caused by security intrusions. Within the debate is the issue of whether or not there even exists a monoculture today (consisting of Microsoft products).

18.3 System and Network Threats

Program threats typically use a breakdown in the protection mechanisms of a system to attack programs. In contrast, system and network threats involve the abuse of services and network connections. Sometimes a system and network attack is used to launch a program attack, and vice versa.

System and network threats create a situation in which operating-system resources and user files are misused. Here, we discuss some examples of these threats, including worms, port scanning, and denial-of-service attacks.

It is important to note that masquerading and replay attacks are also common over networks between systems. In fact, these attacks are more effective and harder to counter when multiple systems are involved. For example, within a computer, the operating system usually can determine the sender and receiver of a message. Even if the sender changes to the ID of someone else, there might be a record of that ID change. When multiple systems are involved, especially systems controlled by attackers, then such tracing is much harder.

The generalization is that sharing secrets (to prove identity and as keys to encryption) is required for authentication and encryption, and that is easier in environments (such as a single operating system) in which secure sharing methods exist. These methods include shared memory and interprocess communications. Creating secure communication and authentication is discussed in Sections 18.4 and 18.5.

18.3.1 Worms

A **worm** is a process that uses the **spawn** mechanism to ravage system performance. The worm spawns copies of itself, using up system resources and perhaps locking out all other processes. On computer networks, worms are particularly potent, since they may reproduce themselves among systems and thus shut down an entire network. Such an event occurred in 1988 to UNIX systems on the Internet, causing millions of dollars of lost system and system administrator time.

At the close of the workday on November 2, 1988, Robert Tappan Morris, Jr., a first-year Cornell graduate student, unleashed a worm program on one or more hosts connected to the Internet. Targeting Sun Microsystems' Sun 3 workstations and VAX computers running variants of Version 4 BSD UNIX, the worm quickly spread over great distances; within a few hours of its release, it had consumed system resources to the point of bringing down the infected machines.

Although Robert Morris designed the self-replicating program for rapid reproduction and distribution, some of the features of the UNIX networking environment provided the means to propagate the worm throughout the system. It is likely that Morris chose for initial infection an Internet host left open for and accessible to outside users. From there, the worm program exploited flaws in the UNIX operating system's security routines and took advantage of UNIX utilities that simplify resource sharing in local-area networks to gain unauthorized access to thousands of other connected sites. Morris's methods of attack are outlined next.

The worm was made up of two programs, a **grappling hook** (also called a **bootstrap** or **vector**) program and the main program. Named *l1.c,* the grappling hook consisted of 99 lines of C code compiled and run on each machine it accessed. Once established on the computer system under attack, the grappling hook connected to the machine where it originated and uploaded a copy of the main worm onto the *hooked* system (Figure 18.6). The main program proceeded to search for other machines to which the newly infected system could connect easily. In these actions, Morris exploited the UNIX networking utility `rsh` for easy remote task execution. By setting up special files that list host–login name pairs, users can omit entering a password each time they access a remote account on the paired list. The worm searched these special files for site names that would allow remote execution without a password. Where remote shells were established, the worm program was uploaded and began executing anew.

The attack via remote access was one of three infection methods built into the worm. The other two methods involved operating-system bugs in the UNIX `finger` and `sendmail` programs.

The `finger` utility functions as an electronic telephone directory; the command

```
finger user-name@hostname
```

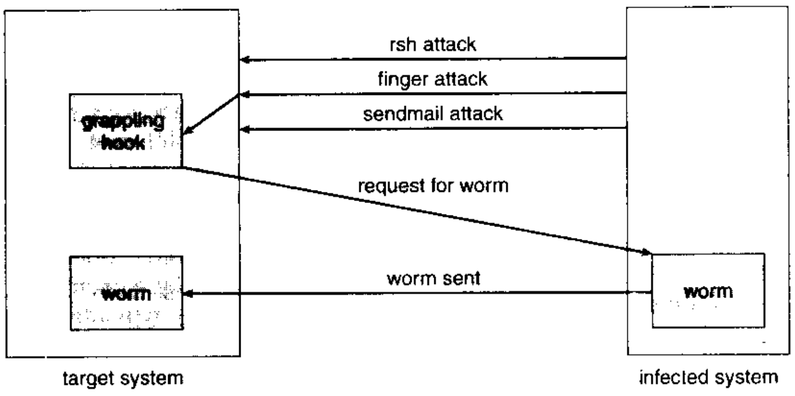


Figure 18.6 The Morris Internet worm.

returns a person's real and login names along with other information that the user may have provided, such as office and home address and telephone number, research plan, or clever quotation. Finger runs as a background process (or daemon) at each BSD site and responds to queries throughout the Internet. The worm executed a buffer-overflow attack on `finger`. The program queried `finger` with a 536-byte string crafted to exceed the buffer allocated for input and to overwrite the stack frame. Instead of returning to the *main* routine it was in before Morris's call, the `finger` daemon was routed to a procedure within the invading 536-byte string now residing on the stack. The new procedure executed `/bin/sh`, which, if successful, gave the worm a remote shell on the machine under attack.

The bug exploited in `sendmail` also involved using a daemon process for malicious entry. `sendmail` sends, receives, and routes electronic mail. Debugging code in the utility permits testers to verify and display the state of the mail system. The debugging option was useful to system administrators and was often left on. Morris included in his attack arsenal a call to debug that —instead of specifying a user address, as would be normal in testing—issued a set of commands that mailed and executed a copy of the grappling-hook program.

Once in place, the main worm undertook systematic attempts to discover user passwords. It began by trying simple cases of no password or of passwords constructed of account–user-name combinations, then used comparisons with an internal dictionary of 432 favorite password choices, and then went to the final stage of trying each word in the standard UNIX on-line dictionary as a possible password. This elaborate and efficient three-stage password-cracking algorithm enabled the worm to gain access to other user accounts on the infected system. The worm then searched for `rsh` data files in these newly broken accounts and used them as described previously to gain access to user accounts on remote systems.

With each new access, the worm program searched for already active copies of itself. If it found one, the new copy exited, except in every seventh instance. Had the worm exited on all duplicate sightings, it might have remained undetected. Allowing every seventh duplicate to proceed (possibly to confound efforts to stop its spread by baiting with *fake* worms) created a wholesale infestation of Sun and VAX systems on the Internet.

The very features of the UNIX network environment that assisted the worm's propagation also helped to stop its advance. Ease of electronic communication, mechanisms to copy source and binary files to remote machines, and access to both source code and human expertise allowed cooperative efforts to develop solutions quickly. By the evening of the next day, November 3, methods of halting the invading program were circulated to system administrators via the Internet. Within days, specific software patches for the exploited security flaws were available.

Why did Morris unleash the worm? The action has been characterized as both a harmless prank gone awry and a serious criminal offense. Based on the complexity of starting the attack, it is unlikely that the worm's release or the scope of its spread was unintentional. The worm program took elaborate steps to cover its tracks and to repel efforts to stop its spread. Yet the program contained no code aimed at damaging or destroying the systems on which it ran. The author clearly had the expertise to include such commands; in fact, data structures were present in the bootstrap code that could have been used to transfer Trojan-horse or virus programs. The behavior of the program may lead to interesting observations, but it does not provide a sound basis for inferring motive. What is not open to speculation, however, is the legal outcome: A federal court convicted Morris and handed down a sentence of three years' probation, 400 hours of community service, and a $10,000 fine. Morris's legal costs probably exceeded $100,000.

Security experts continue to evaluate methods to decrease or eliminate worms. A more recent event, though, shows that worms are still a fact of life on the Internet. It also shows that as the Internet grows, the damage that even "harmless" worms can do also grows and can be significant. This example occurred during August 2003. The fifth version of the "Sobig" worm, more properly known as "W32.Sobig.F@mm," was released by persons at this time unknown. It was the fastest-spreading worm released to date, at its peak infecting hundreds of thousands of computers and one in seventeen e-mail messages on the Internet. It clogged e-mail inboxes, slowed networks, and took a huge number of hours to clean up.

Sobig.F was launched by being uploaded to a pornography newsgroup via an account created with a stolen credit card. It was disguised as a photo. The virus targeted Microsoft Windows systems and used its own SMTP engine to e-mail itself to all the addresses found on an infected system. It used a variety of subject lines to help avoid detection, including "Thank You!" "Your details," and "Re: Approved." It also used a random address on the host as the "From:" address, making it difficult to determine from the message which machine was the infected source. Sobig.F included an attachment for the target e-mail reader to click on, again with a variety of names. If this payload was executed, it stored a program called WINPPR32.EXE in the default Windows directory, along with a text file. It also modified the Windows registry.

The code included in the attachment was also programmed to periodically attempt to connect to one of twenty servers and download and execute a program from them. Fortunately, the servers were disabled before the code could be downloaded. The content of the program from these servers has not yet been determined. If the code was malevolent, untold damage to a vast number of machines could have resulted.

18.3.2 Port Scanning

Port scanning is not an attack but rather is a means for a cracker to detect a system's vulnerabilities to attack. Port scanning typically is automated, involving a tool that attempts to create a TCP/IP connection to a specific port or a range of ports. For example, suppose there is a known vulnerability (or bug) in `sendmail`. A cracker could launch a port scanner to try to connect to, say, port 25 of a particular system or a range of systems. If the connection was successful, the cracker (or tool) could attempt to communicate with the answering service to determine if it was indeed `sendmail` and, if so, if it was the version with the bug.

Now imagine a tool in which each bug of every service of every operating system was encoded. The tool could attempt to connect to every port of one or more systems. For every service that answered, it could try to use each known bug. Frequently, the bugs are buffer overflows, allowing the creation of a privileged command shell on the system. From there, of course, the cracker could install Trojan horses, back-door programs, and so on.

There is no such tool, but there are tools that perform subsets of that functionality. For example, `nmap` (from http://www.insecure.org/nmap/) is a very versatile open-source utility for network exploration and security auditing. When pointed at a target, it will determine what services are running, including application names and versions. It can determine the host operating system. It can also provide information about defenses, such as what firewalls are defending the target. It does not exploit any known bugs.

`Nessus` (from http://www.nessus.org/) performs a similar function, but it has a database of bugs and their exploits. It can scan a range of systems, determine the services running on those systems, and attempt to attack all appropriate bugs. It generates reports about the results. It does not perform the final step of exploiting the found bugs, but a knowledgeable cracker or a script kiddie could.

Because port scans are detectable (see 18.6.3), they frequently are launched from **zombie systems**. Such systems are previously compromised, independent systems that are serving their owners while being used for nefarious purposes, including denial-of-service attacks and spam relay. Zombies make crackers particularly difficult to prosecute because determining the source of the attack and the person that launched it is challenging. This is one of many reasons that "inconsequential" systems should also be secured, not just systems containing "valuable" information or services.

18.3.3 Denial of Service

As mentioned earlier, DOS attacks are aimed not at gaining information or stealing resources but rather at disrupting legitimate use of a system or facility. Most denial-of-service attacks involve systems that the attacker has not penetrated. Indeed, launching an attack that prevents legitimate use is frequently easier than breaking into a machine or facility.

Denial-of-service attacks are generally network based. They fall into two categories. The first case is an attack that uses so many facility resources that, in essence, no useful work can be done. For example, a web-site click could download a Java applet that proceeds to use all available CPU time or to infinitely pop up windows. The second case involves disrupting the network of the facility. There have been several successful denial-of-service attacks of this kind against major web sites. They result from abuse of some of the fundamental functionality of TCP/IP. For instance, if the attacker sends the part of the protocol that says "I want to start a TCP connection," but never follows with the standard "The connection is now complete," the result can be partially started TCP sessions. Enough of these sessions can eat up all the network resources of the system, disabling any further legitimate TCP connections. Such attacks, which can last hours or days, have caused partial or full failure of attempts to use the target facility. These attacks are usually stopped at the network level until the operating systems can be updated to reduce their vulnerability.

Generally, it is impossible to prevent denial-of-service attacks. The attacks use the same mechanisms as normal operation. Even more difficult to prevent and resolve are **distributed denial-of-service attacks (DDOS)**. These attacks are launched from multiple sites at once, toward a common target, typically by zombies.

Sometimes a site does not even know it is under attack. It can be difficult to determine whether a system slowdown is just a surge in system use or an attack. Consider that a successful advertising campaign that greatly increases traffic to a site could be considered a DDOS.

There are other interesting aspects of DOS attacks. For example, programmers and systems managers need to fully understand the algorithms and technologies they are deploying. If an authentication algorithm locks an account for a period of time after several incorrect attempts, then an attacker could cause all authentication to be blocked by purposefully causing incorrect attempts to all accounts. Similarly, a firewall that automatically blocks certain kinds of traffic could be induced to block that traffic when it should not. Finally, computer science classes are notorious sources of accidental system DOS attacks. Consider the first programming exercises in which students learn to create subprocesses or threads. A common bug involves spawning subprocesses infinitely. The system's free memory and CPU resources don't stand a chance.

18.4 Cryptography as a Security Tool

There are many defenses against computer attacks, running the gamut from methodology to technology. The broadest tool available to system designers and users is cryptography. In this section we discuss the details of crypography and its use in computer security.

In an isolated computer, the operating system can reliably determine the sender and recipient of all interprocess communication, since it controls al communication channels in the computer. In a network of computers, the situation is quite different. A networked computer receives bits *from the wire* with no immediate and reliable way of determining what machine or application sent those bits. Similarly, the computer sends bits onto the network with no way of knowing who might eventually receive them.

Commonly, network addresses are used to infer the potential senders and receivers of network messages. Network packets arrive with a source address, such as an IP address. And when a computer sends a message, it names the intended receiver by specifying a destination address. However, for applications where security matters, we are asking for trouble if we assume that the source or destination address of a packet reliably determines who sent or received that packet. A rogue computer can send a message with a falsified source address, and numerous computers other than the one specified by the destination address can (and typically do) receive a packet. For example, all of the routers on the way to the destination will receive the packet, too. How, then, is an operating system to decide whether to grant a request when it cannot trust the named source of the request? And how is it supposed to provide protection for a request or data when it cannot determine who will receive the response or message contents it sends over the network?

It is generally considered infeasible to build a network of any scale in which the source and destination addresses of packets can be *trusted* in this sense. Therefore, the only alternative is somehow to eliminate the need to trust the network. This is the job of cryptography. Abstractly, **cryptography** is used to constrain the potential senders and/or receivers of a message. Modern cryptography is based on secrets called **keys** that are selectively distributed to computers in a network and used to process messages. Cryptography enables a recipient of a message to verify that the message was created by some computer possessing a certain key—the key is the *source* of the message. Similarly, a sender can encode its message so that only a computer with a certain key can decode the message, so that the key becomes the *destination*. Unlike network addresses, however, keys are designed so that it is not computationally feasible to derive them from the messages they were used to generate or from any other public information. Thus, they provide a much more trustworthy means of constraining senders and receivers of messages. Note that cryptography is a field of study unto itself, with large and small complexities and subtleties. Here, we explore the most important aspects of the parts of cryptography that pertain to operating systems.

18.4.1 Encryption

Because it solves a wide variety of communication security problems, encryption is used frequently in many aspects of modern computing. Encryption is a means for constraining the possible receivers of a message. An encryption algorithm enables the sender of a message to ensure that only a computer possessing a certain key can read the message. Encryption of messages is an ancient practice, of course, and there have been many encryption algorithms, dating back to before Caesar. In this section, we describe important modern encryption principles and algorithms.

Figure 18.7 shows an example of two users communicating securely over an insecure channel. We refer to this figure throughout the section. Note that the key exchange can take place directly between the two parties or via a trusted third party (that is, a certificate authority), as discussed in Section 18.4.1.4.

An encryption algorithm consists of the following components:

A set K of keys.

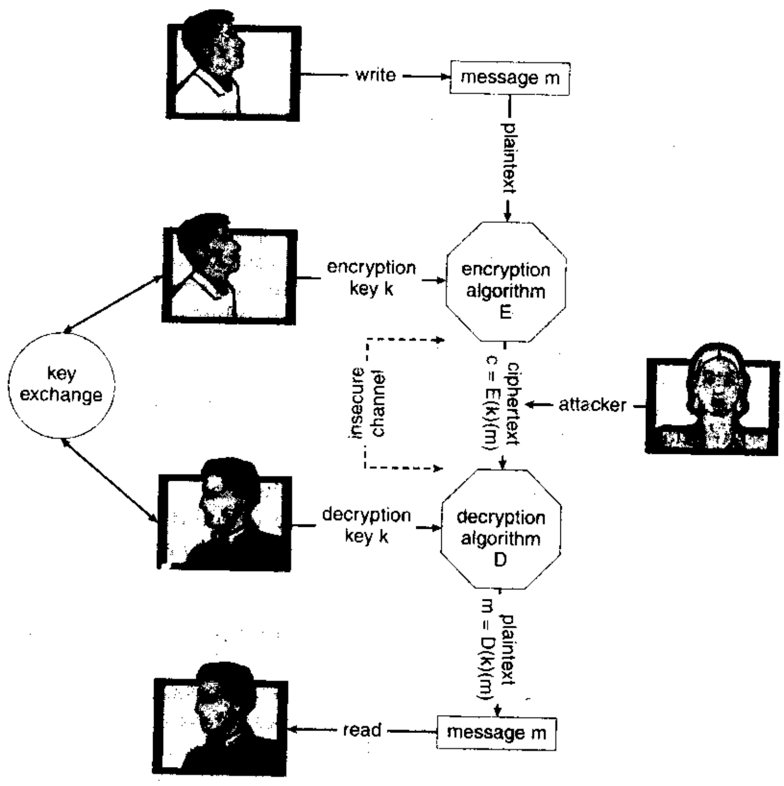


Figure 18.7 A secure communication over an insecure medium.

- A set M of messages.
- A set C of ciphertexts.
- A function $E : K \rightarrow (M \rightarrow C)$. That is, for each $k \in K$, $E(k)$ is a function fc generating ciphertexts from messages. Both E and $E(k)$ for any k shoul be efficiently computable functions.
- A function $D : K \rightarrow (C \rightarrow M)$. That is, for each $k \in K$, $D(k)$ is a function fc generating messages from ciphertexts. Both D and $D(k)$ for any k shoul be efficiently computable functions.

An encryption algorithm must provide this essential property: Given ciphertext $c \in C$, a computer can compute m such that $E(k)(m) = c$ only it possesses $D(k)$. Thus, a computer holding $D(k)$ can decrypt ciphertexts t the plaintexts used to produce them, but a computer not holding $D(k)$ cannc decrypt ciphertexts. Since ciphertexts are generally exposed (for example, ser on the network), it is important that it be infeasible to derive $D(k)$ from th ciphertexts.

There are two main types of encryption algorithms: symmetric and asymmetric. We discuss both types in the following sections.

18.4.1.1 Symmetric Encryption

In a **symmetric encryption algorithm**, the same key is used to encrypt and to decrypt. That is, $E(k)$ can be derived from $D(k)$, and vice versa. Therefore, the secrecy of $E(k)$ must be protected to the same extent as that of $D(k)$.

For the past 20 years or so, the most commonly used symmetric encryption algorithm in the United States for civilian applications has been the **data-encryption standard (DES)** adopted by the National Institute of Standards and Technology (NIST). DES works by taking a 64-bit value and a 56-bit key and performing a series of transformations. These transformations are based on substitution and permutation operations, as is generally the case for symmetric encryption transformations. Some of the transformations are **black-box transformations**, in that their algorithms are hidden. In fact, these so-called "S-boxes" are classified by the United States government. Messages longer than 64 bits are broken into 64-bit chunks, and a shorter block is padded to fill out the block. Because DES works on a chunk of bits at a time, is a known as a **block cipher**. If the same key is used for encrypting an extended amount of data, it becomes vulnerable to attack. Consider, for example, that the same source block would result in the same ciphertext if the same key and encryption algorithm were used. Therefore, the chunks are not just encrypted but also XORed with the previous ciphertext block before encryption. This is known as **cipher-block chaining**.

DES is now considered insecure for many applications because its keys can be exhaustively searched with moderate computing resources. Rather than giving up on DES, though, NIST created a modification called **triple DES**, in which the DES algorithm is repeated three times (two encryptions and one decryption) on the same plaintext using two or three keys—for example, $c = E(k_3)(D(k_2)(E(K_1)(m)))$. When three keys are used, the effective key length is 168 bits. Triple DES is in widespread use today.

In 2001, NIST adopted a new encryption algorithm, called the **advanced encryption standard (AES)**, to replace DES. AES is another symmetric block cipher. It can use key lengths of 128, 192, and 256 bits and works on 128-bit blocks. It works by performing 10 to 14 rounds of transformations on a matrix formed from a block. Generally, the algorithm is compact and efficient.

There are several other symmetric block encryption algorithms in use today that bear mentioning. The **twofish** algorithm is fast, compact, and easy to implement. It can use a variable key length of up to 256 bits and works on 128-bit blocks. **RC5** can vary in key length, number of transformations, and block size. Because it uses only basic computational operations, it can run on a wide variety of CPUs.

RC4 is perhaps the most common stream cipher. A **stream cipher** is designed to encrypt and decrypt a stream of bytes or bits rather than a block. This is useful when the length of a communication would make a block cipher too slow. The key is input into a pseudo-random-bit generator, which is an algorithm that attempts to produce random bits. The output of the generator when fed a key is a keystream. A **keystream** is an infinite set of keys that can be used for the input plaintext stream. RC4 is used in encrypting steams of data, such as in WEP, the wireless LAN protocol. It is also used in communications between web browsers and web servers, as we discuss below. Unfortunately, RC4 as used in WEP (IEEE standard 802.11) has been found to be breakable in a reasonable amount of computer time. In fact, RC4 itself has vulnerabilities.

18.4.1.2 Asymmetric Encryption

In an **asymmetric encryption algorithm**, there are different encryption and decryption keys. Here, we describe one such algorithm, known as *RSA* after the names of its inventors (Rivest, Shamir and Adleman.) The RSA cipher is a block-cipher public-key algorithm and is the most widely used asymmetrical algorithm. Asymmetrical algorithms based on elliptical curves are gaining ground, however, because the key length of such an algorithm can be shorter for the same amount of cryptographic strength.

It is computationally infeasible to derive $D(k_d, N)$ from $E(k_e, N)$, and so $E(k_e, N)$ need not be kept secret and can be widely disseminated; thus, $E(k_e, N)$ (or just k_e) is the **public key** and $D(k_d, N)$ (or just k_d) is the **private key**. N is the product of two large, randomly chosen prime numbers p and q (for example, p and q are 512 bits each). The encryption algorithm is $E(k_e, N)(m) = m^{k_e} \bmod N$, where k_e satisfies $k_e k_d \bmod (p-1)(q-1) = 1$. The decryption algorithm is then $D(k_d, N)(c) = c^{k_d} \bmod N$.

An example using small values is shown in Figure 18.8. In this example, we make $p = 7$ and $q = 13$. We then calculate $N = 7*13 = 91$ and $(p-1)(q-1) = 72$. We next select k_e relatively prime to 72 and < 72, yielding 5. Finally, we calculate k_d such that $k_e k_d \bmod 72 = 1$, yielding 29. We how have our keys: the public key, $k_e, N = 5, 91$, and the private key, $k_d, N = 29, 91$. Encrypting the message 69 with the public key results in the message 62, which is then decoded by the receiver via the private key.

The use of asymmetric encryption begins with the publication of the public key of the destination. For bidirectional communication, the source also must publish its public key. "Publication" can be as simple as handing over an electronic copy of the key, or it can be more complex. The private key (or "secret key") must be jealously guarded, as anyone holding that key can decrypt any message created by the matching public key.

We should note that the seemingly small difference in key use between asymmetric and symmetric cryptography is quite large in practice. Asymmetric cryptography is based on mathematical functions rather than transformations, making it much more computationally expensive to execute. It is much faster for a computer to encode and decode ciphertext by using the usual symmetric algorithms than by using asymmetric algorithms. Why, then, use an asymmetric algorithm? In truth, these algorithms are not used for general-purpose encryption of large amounts of data. However, they are used not only for encryption of small amounts of data but also for authentication, confidentiality, and key distribution, as we show in the following sections.

18.4.1.3 Authentication

We have seen that encryption offers a way of constraining the set of possible receivers of a message. Constraining the set of potential senders of a message is

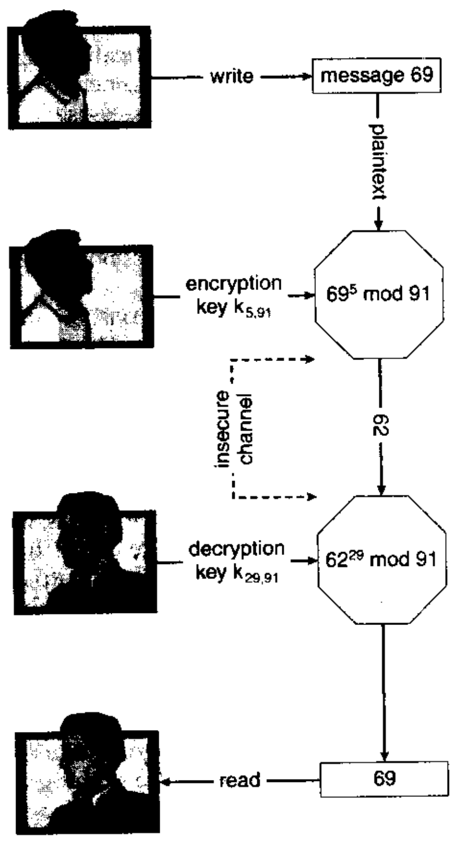


Figure 18.8 Encryption and decryption using RSA asymmetric cryptography.

called **authentication**. Authentication is thus complementary to encryption. In fact, sometimes their functions overlap. Consider that an encrypted message can also prove the identity of the sender. For example, if $D(k_d, N)(E(k_e, N)(m))$ produces a valid message, then we know that the creator of the message must hold k_e. Authentication is also useful for proving that a message has not been modified. In this section, we discuss authentication as a constraint on possible receivers of a message. Note that this sort of authentication is similar to but distinct from user authentication, which we discuss in Section 18.5.

An authentication algorithm consists of the following components:

- A set K of keys.
- A set M of messages.
- A set A of authenticators.
- A function $S : K \rightarrow (M \rightarrow A)$. That is, for each $k \in K$, $S(k)$ is a function for generating authenticators from messages. Both S and $S(k)$ for any k should be efficiently computable functions.

A function $V : K \rightarrow (M \times A \rightarrow \{\texttt{true}, \texttt{false}\})$. That is, for each $k \in K$, $V(k)$ is a function for verifying authenticators on messages. Both V and $V(k)$ for any k should be efficiently computable functions.

The critical property that an authentication algorithm must possess is this: For a message m, a computer can generate an authenticator $a \in A$ such that $V(k)(m, a) = \texttt{true}$ only if it possesses $S(k)$. Thus, a computer holding $S(k)$ can generate authenticators on messages so that any other computer possessing $V(k)$ can verify them. However, a computer not holding $S(k)$ cannot generate authenticators on messages that can be verified using $V(k)$. Since authenticators are generally exposed (for example, they are sent on the network with the messages themselves), it must not be feasible to derive $S(k)$ from the authenticators.

Just as there are two types of encryption algorithms, there are two main varieties of authentication algorithms. The first step in understanding these algorithms is to explore hash functions. A **hash function** creates a small, fixed-sized block of data, known as a **message digest** or **hash value**, from a message. Hash functions work by taking a message in n-bit blocks and processing the blocks to produce an n-bit hash. H must be collision resistant on m—that is, it must be infeasible to find an $m' \neq m$ such that $H(m) = H(m')$. Now, if $H(m) = H(m')$, we know that $m_1 = m_2$—that is, we know that the message has not been modified. Common message-digest functions include **MD5**, which produces a 128-bit hash, and **SHA-1**, which outputs a 160-bit hash.

Message digests are useful for detecting changed messages but are not useful as authenticators. For example, $H(m)$ can be sent along with a message; but if H is known, then someone could modify m and recompute $H(m)$, and the message modification would not be detected. Therefore, an authentication algorithm takes the message digest and encrypts it.

The first type of authentication algorithm uses symmetric encryption. In a **message-authentication code (MAC)**, a cryptographic checksum is generated from the message using a secret key. Knowledge of $V(k)$ and knowledge of $S(k)$ are equivalent: One can be derived from the other, so k must be kept secret. A simple example of a MAC defines $S(k)(m) = f(k, H(m))$, where f is a function that is one-way on its first argument (that is, k cannot be derived from $f(k, H(m))$). Because of the collision resistance in the hash function, we are reasonably assured that no other message could create the same MAC. A suitable verification algorithm is then $V(k)(m, a) \equiv (f(k, m) = a)$. Note that k is needed to compute both $S(k)$ and $V(k)$, so anyone able to compute one can compute the other.

The second main type of authentication algorithm is a **digital-signature algorithm**, and the authenticators thus produced are called **digital signatures**. In a digital-signature algorithm, it is computationally infeasible to derive $S(k_s)$ from $V(k_v)$; in particular, V is a one-way function. Thus, k_v is the public key and k_s is the private key.

Consider as an example the RSA digital-signature algorithm. It is similar to the RSA encryption algorithm, but the key use is reversed. The digital signature of a message is derived by computing $S(k_s)(m) = H(m)^{k_s} \bmod N$. The key k_s again is a pair $\langle d, N \rangle$, where N is the product of two large, randomly chosen prime numbers p and q. The verification algorithm is then $V(k_v)(m, a) \equiv (a^{k_v} \bmod N = H(m))$, where k_v satisfies $k_v k_s \bmod (p-1)(q-1) = 1$.

If encryption can prove the identity of the sender of a message, then why do we need separate authentication algorithms? There are three primary reasons.

Authentication algorithms generally require fewer computations (with the notable exception of RSA digital signatures). Over large amounts of plaintext, this efficiency can make a huge difference in resource use and the time needed to authenticate a message.

An authenticator of a message is almost always shorter than the message and its ciphertext. This improves space use and transmission time efficiency.

Sometimes, we want authentication but not confidentiality. For example, a company could provide a software patch and could "sign" that patch to prove that it came from the company and that it hasn't been modified.

Authentication is a component of many aspects of security. For example, it is the core of **nonrepudiation,** which supplies proof that an entity performed an action. A typical example of nonrepudiation involves the filling out of electronic forms as an alternative to the signing of paper contracts. Nonrepudiation assures that a person filling out an electronic form cannot deny that he did so.

18.4.1.4 Key Distribution

Certainly, a good part of the battle between cryptographers (those inventing ciphers) and cryptanalysts (those trying to break them) involves keys. With symmetric algorithms, both parties need the key, and no one else should have it. The delivery of the symmetric key is a huge challenge. Sometimes it is performed **out-of-band**—say, via a paper document or a conversation. These methods do not scale well, however. Also consider the key-management challenge. Suppose a user wanted to communicate with N other users privately. That user would need N keys and, for more security, would need to change those keys frequently.

These are the very reasons for efforts to create asymmetric key algorithms. Not only can the keys be exchanged in public, but a given user needs only one private key, no matter how many other people she wants to communicate with. There is still the matter of managing a public key per party to be communicated with, but since public keys need not be secured, simple storage can be used for that **key ring**.

Unfortunately, even the distribution of public keys requires some care. Consider the man-in-the-middle attack shown in Figure 18.9. Here, the person who wants to receive an encrypted message sends out his public key, but an attacker also sends her "bad" public key (which matches her private key). The person who wants to send the encrypted message knows no better and so uses the bad key to encrypt the message. The attacker then happily decrypts it.

The problem is one of authentication—what we need is proof of who (or what) owns a public key. One way to solve that problem involves the use of digital certificates. A **digital certificate** is a public key digitally signed by a trusted party. The trusted party receives proof of identification from some entity and certifies that the public key belongs to that entity. But how do we

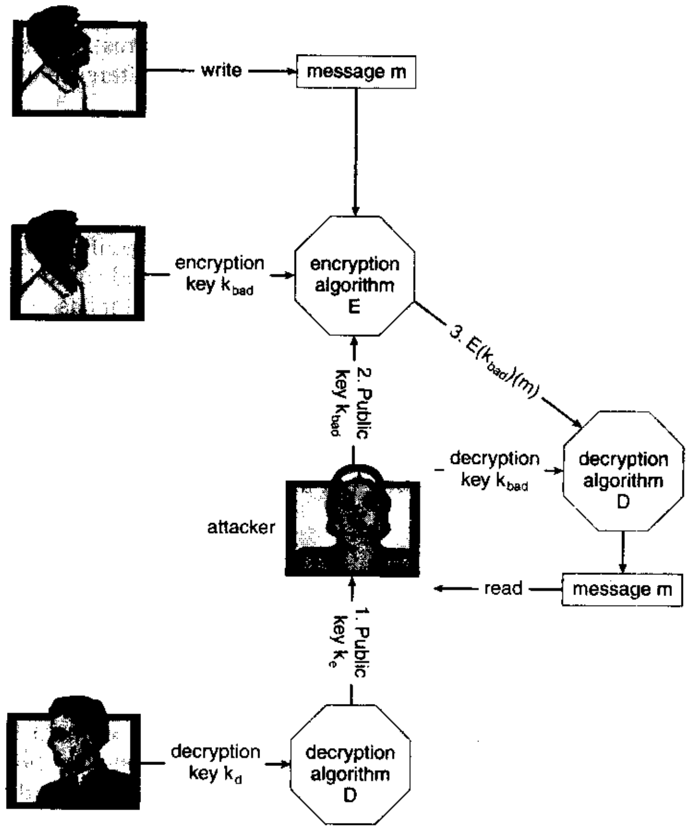


Figure 18.9 A man-in-the-middle attack on asymmetric cryptography.

know we can trust the certifier? These **certificate authorities** have their public keys included within web browsers (and other consumers of certificates) before they are distributed. These certificate authorities can then vouch for other authorities (digitally signing the public keys of these other authorities), and so on, creating a web of trust. The certificates can be distributed in a standard X.509 digital certificate format that can be parsed by computer. This scheme is used for secure web communication, as we discuss in Section 18.4.3.

18.4.2 Implementation of Cryptography

Network protocols are typically organized in **layers**, each layer acting as a client to the one below it. That is, when one protocol generates a message to send to its protocol peer on another machine, it hands its message to the protocol below it in the network-protocol stack for delivery to its peer on that machine. For example, in an IP network, TCP (a *transport-layer* protocol) acts as a client of IP (a *network-layer* protocol): TCP packets are passed down to IP for delivery to the TCP peer at the other end of the TCP connection. IP encapsulates the TCP packet in an IP packet, which it similarly passes down to the *data-link layer* to be transmitted across the network to its IP peer on the destination computer. This IP peer then delivers the TCP packet up to the TCP peer on that machine. All in all, the **ISO Reference Model**, which has been almost universally adopted as a model for data networking, defines seven such protocol layers. (You will read more about the ISO model of networking in Chapter 14; Figure 14.6 shows a diagram of the model.)

Cryptography can be inserted at almost any layer in the ISO model. SSL (Section 18.4.3), for example, provides security at the transport layer. Network-layer security generally has been standardized on **IPSec**, which defines IP packet formats that allow the insertion of authenticators and the encryption of packet contents. It uses symmetric encryption and uses the **IKE** protocol for key exchange. IPSec is becoming widely used as the basis for **virtual private networks (VPNs)**, in which all traffic between two IPSec endpoints is encrypted to make a private network out of one that may otherwise be public. Numerous protocols also have been developed for use by applications, but then the applications themselves must be coded to implement security.

Where is cryptographic protection best placed in a protocol stack? In general, there is no definitive answer. On the one hand, more protocols benefit from protections placed lower in the stack. For example, since IP packets encapsulate TCP packets, encryption of IP packets (using IPSec, for example) also hides the contents of the encapsulated TCP packets. Similarly, authenticators on IP packets detect the modification of contained TCP header information.

On the other hand, protection at lower layers in the protocol stack may give insufficient protection to higher-layer protocols. For example, an application server that runs over IPSec might be able to authenticate the client computers from which requests are received. However, to authenticate a user at a client computer, the server may need to use an application-level protocol—for example, the user may be required to type a password. Also consider the problem of e-mail. E-mail delivered via the industry standard SMTP protocol is stored and forwarded, frequently multiple times, before it is delivered. Each of these hops could go over a secure or insecure network. For e-mail to be secure, the e-mail message needs to be encrypted so that its security is independent of the transports that carry it.

18.4.3 An Example: SSL

SSL 3.0 is a cryptographic protocol that enables two computers to communicate securely—that is, so that each can limit the sender and receiver of messages to the other. It is perhaps the most commonly used cryptographic protocol on the Internet today, since it is the standard protocol by which web browsers communicate securely with web servers. For completeness, we should note that SSL was designed by Netscape and that it evolved into the industry standard TLS protocol. In this discussion, we use SSL to mean both SSL and TLS.

SSL is a complex protocol with many options. Here, we present only a single variation of it, and even then in a very simplified and abstract form, so as to maintain focus on its use of cryptographic primitives. What we are about to see is a complex dance in which asymmetric cryptography is used so that a client and server can establish a secure **session key** that can be used for symmetric encryption of the session between the two—all of this while avoiding man-in-the-middle and replay attacks. For added cryptographic strength, the session keys are forgotten once a session is completed. Another communication between the two would require generation of new session keys.

The SSL protocol is initiated by a **client** c to communicate securely with a **server**. Prior to the protocol's use, the server s is assumed to have obtained a certificate, denoted cert_s, from certification authority CA. This certificate is a structure containing the following:

- Various attributes attrs of the server, such as its unique *distinguished* name and its *common* (DNS) name
- The identity of a public encryption algorithm $E()$ for the server
- The public key k_e of this server
- A validity interval interval during which the certificate should be considered valid
- A digital signature a on the above information by the CA—that is, $a = S(k_{CA})(\langle \text{attrs}, E(k_e), \text{interval} \rangle)$

In addition, prior to the protocol's use, the client is presumed to have obtained the public verification algorithm $V(k_{CA})$ for CA. In the case of the Web, the user's browser is shipped from its vendor containing the verification algorithms and public keys of certain certification authorities. The user can add or delete these for certification authorities as she chooses.

When c connects to s, it sends a 28-byte random value n_c to the server, which responds with a random value n_s of its own, plus its certificate cert_s. The client verifies that $V(k_{CA})(\langle \text{attrs}, E(k_e), \text{interval}\rangle, a) = \text{true}$ and that the current time is in the validity interval interval. If both of these tests are satisfied, the server has proved its identity. Then the client generates a random 46-byte **premaster secret** pms and sends $\text{cpms} = E(k_s)(\text{pms})$ to the server. The server recovers $\text{pms} = D(k_d)(\text{cpms})$. Now both the client and the server are in possession of n_c, n_s, and pms, and each can compute a shared 48-byte **master secret** $\text{ms} = f(n_c, n_s, \text{pms})$, where f is a one-way and collision-resistant function. Only the server and client can compute ms, since only they know pms. Moreover, the dependence of ms on n_c and n_s ensures that ms is a *fresh* value—that is, a session key that has not been used in a previous communication. At this point, the client and the server both compute the following keys from the ms:

- A symmetric encryption key k_{cs}^{crypt} for encrypting messages from the client to the server
- A symmetric encryption key k_{sc}^{crypt} for encrypting messages from the server to the client
- A MAC generation key k_{cs}^{mac} for generating authenticators on messages from the client to the server
- A MAC generation key k_{sc}^{mac} for generating authenticators on messages from the server to the client

To send a message m to the server, the client sends

$$c = E(k_{cs}^{\text{crypt}})(\langle m, S(k_{cs}^{\text{mac}})(m)\rangle).$$

Upon receiving c, the server recovers

$$\langle m, a\rangle = D(k_{cs}^{\text{crypt}})(c)$$

and accepts m if $V(k_{cs}^{\text{mac}})(m, a) = \text{true}$. Similarly, to send a message m to the client, the server sends

$$c = E(k_{sc}^{\text{crypt}})(\langle m, S(k_{sc}^{\text{mac}})(m)\rangle)$$

and the client recovers

$$\langle m, a\rangle = D(k_{sc}^{\text{crypt}})(c)$$

and accepts m if $V(k_{sc}^{\text{mac}})(m, a) = \text{true}$.

This protocol enables the server to limit the recipients of its messages to the client that generated pms and to limit the senders of the messages it accepts to that same client. Similarly, the client can limit the recipients of the messages it sends and the sender of the messages it accepts to the party that knows $S(k_d)$ (that is, the party that can decrypt cpms). In many applications, such as web transactions, the client needs to verify the identity of the party that knows $S(k_d)$. This is one purpose of the certificate cert_s; in particular, the attrs field contains information that the client can use to determine the identity—for example, the domain name—of the server with which it is communicating. For applications in which the server also needs information about the client, SSL supports an option by which a client can send a certificate to the server.

In addition to its use on the Internet, SSL is being used for a wide variety of tasks. For example, IPSec VPNs now have a competitor in SSL VPNs. IPSec is good for point-to-point encryption of traffic—say, between two company offices. SSL VPNs are more flexible but not as efficient, so they might be used between an individual employee working remotely and the corporate office.

18.5

The discussion of authentication above involves messages and sessions. But what of users? If a system cannot authenticate a user, then authenticating that a message came from that user is pointless. Thus, a major security problem for operating systems is **user authentication**. The protection system depends on the ability to identify the programs and processes currently executing, which in turn depends on the ability to identify each user of the system. A user normally identifies herself. How do we determine whether a user's identity is authentic? Generally, user authentication is based on one or more of three things: the user's possession of something (a key or card), the user's knowledge of something (a user identifier and password), and/or an attribute of the user (fingerprint, retina pattern, or signature).

18.5.1 Passwords

The most common approach to authenticating a user identity is the use of **passwords**. When the user identifies herself by user ID or account name, she is asked for a password. If the user-supplied password matches the password stored in the system, the system assumes that the account is being accessed by the owner of that account.

Passwords are often used to protect objects in the computer system, in the absence of more complete protection schemes. They can be considered a special case of either keys or capabilities. For instance, a password could be associated with each resource (such as a file). Whenever a request is made to use the resource, the password must be given. If the password is correct, access is granted. Different passwords may be associated with different access rights. For example, different passwords may be used for reading files, appending files, and updating files.

In practice, most systems require only one password for a user to gain full rights. Although more passwords theoretically would be more secure, such systems tend not to be implemented due to the classic trade-off between security and convenience. If security makes something inconvenient, then the security is frequently bypassed or otherwise circumvented.

18.5.2 Password Vulnerabilities

Passwords are extremely common because they are easy to understand and use. Unfortunately, passwords can often be guessed, accidentally exposed, sniffed, or illegally transferred from an authorized user to an unauthorized one, as we show next.

There are two common ways to guess a password. One way is for the intruder (either human or program) to know the user or to have information about the user. All too frequently, people use obvious information (such as the names of their cats or spouses) as their passwords. The other way is to use brute force, trying enumeration—or all possible combinations of valid password characters (letters, numbers, and punctuation on some systems)—until the password is found. Short passwords are especially vulnerable to this method. For example, a four-decimal password provides only 10,000 variations. On average, guessing 5,000 times would produce a correct hit. A program that could try a password every millisecond would take only about 5 seconds to guess a four-digit password. Enumeration is less successful where systems allow longer passwords that include both uppercase and lowercase letters, along with numbers and all punctuation characters. Of course, users must take advantage of the large password space and must not, for example, use only lowercase letters.

In addition to being guessed, passwords can be exposed as a result of visual or electronic monitoring. An intruder can look over the shoulder of a user (**shoulder surfing**) when the user is logging in and can learn the password easily by watching the keyboard. Alternatively, anyone with access to the network on which a computer resides can seamlessly add a network monitor, allowing her to watch all data being transferred on the network (**sniffing**), including user IDs and passwords. Encrypting the data stream containing the password solves this problem. Even such a system could have passwords stolen, however. For example, if a file is used to contain the passwords, it could be copied for off-system analysis. Or consider a Trojan-horse program installed on the system that captures every keystroke before sending it on to the application.

Exposure is a particularly severe problem if the password is written down where it can be read or lost. As we shall see, some systems force users to select hard-to-remember or long passwords, which may cause a user to record the password or to reuse it. As a result, such systems provide much less security than systems that allow users to select easy passwords!

The final type of password compromise, illegal transfer, is the result of human nature. Most computer installations have a rule that forbids users to share accounts. This rule is sometimes implemented for accounting reasons but is often aimed at improving security. For instance, suppose one user ID is shared by several users, and a security breach occurs from that user ID. It is impossible to know who was using the ID at the time the break occurred or even whether the user was an authorized one. With one user per user ID, any user can be questioned directly about use of the account; in addition, the user might notice something different about the account and detect the break-in. Sometimes, users break account-sharing rules to help friends or to circumvent accounting, and this behavior can result in a system's being accessed by unauthorized users —possibly harmful ones.

Passwords can be either generated by the system or selected by a user. System-generated passwords may be difficult to remember, and thus users may write them down. As mentioned, however, user-selected passwords are often easy to guess (the user's name or favorite car, for example). Some systems will check a proposed password for ease of guessing or cracking before accepting it. At some sites, administrators occasionally check user passwords and notify a user if his password is easy to guess. Some systems also *age* passwords, forcing users to change their passwords at regular intervals (every three months, for instance). This method is not foolproof either, because users can easily toggle between two passwords. The solution, as implemented on some systems, is to record a password history for each user. For instance, the system could record the last N passwords and not allow their reuse.

Several variants on these simple password schemes can be used. For example, the password can be changed more frequently. In the extreme, the password is changed from session to session. A new password is selected (either by the system or by the user) at the end of *each* session, and that password must be used for the next session. In such a case, even if a password is misused, it can be used only once. When the legitimate user tries to use a now-invalid password at the next session, he discovers the security violation. Steps can then be taken to repair the breached security.

18.5.3 Encrypted Passwords

One problem with all these approaches is the difficulty of keeping the password secret within the computer. How can the system store a password securely yet allow its use for authentication when the user presents her password? The UNIX system uses encryption to avoid the necessity of keeping its password list secret. Each user has a password. The system contains a function that is extremely difficult—the designers hope impossible—to invert but is simple to compute. That is, given a value x, it is easy to compute the function value $f(x)$. Given a function value $f(x)$, however, it is impossible to compute x. This function is used to encode all passwords. Only encoded passwords are stored. When a user presents a password, it is encoded and compared against the stored encoded password. Even if the stored encoded password is seen, it cannot be decoded, so the password cannot be determined. Thus, the password file does not need to be kept secret. The function $f(x)$ is typically an encryption algorithm that has been designed and tested rigorously.

The flaw in this method is that the system no longer has control over the passwords. Although the passwords are encrypted, anyone with a copy of the password file can run fast encryption routines against it—encrypting each word in a dictionary, for instance, and comparing the results against the passwords. If the user has selected a password that is also a word in the dictionary, the password is cracked. On sufficiently fast computers, or even on clusters of slow computers, such a comparison may take only a few hours. Furthermore, because UNIX systems use a well-known encryption algorithm, a cracker might keep a cache of passwords that have been cracked previously. For these reason, new versions of UNIX store the encrypted password entries in a file readable only by the **superuser**. The programs that compare a presented password to the stored password run `setuid` to root; so they can read this file, but other users cannot. They also include a "salt," or recorded random number, in the encryption algorithm. The salt is added to the password to ensure that if two plaintext passwords are the same, they result in different ciphertexts.

Another weakness in the UNIX password methods is that many UNIX systems treat only the first eight characters as significant. It is therefore extremely important for users to take advantage of the available password space. To avoid the dictionary encryption method, some systems disallow the use of dictionary words as passwords. A good technique is to generate your password by using the first letter of each word of an easily remembered phrase using both upper and lower characters with a number or punctuation mark thrown in for good measure. For example, the phrase "My mother's name is Katherine" might yield the password "Mmn.isK!". The password is hard to crack but easy for the user to remember.

18.5.4 One-Time Passwords

To avoid the problems of password sniffing and shoulder surfing, a system could use a set of **paired passwords**. When a session begins, the system randomly selects and presents one part of a password pair; the user must supply the other part. In this system, the user is **challenged** and must **respond** with the correct answer to that challenge.

This approach can be generalized to the use of an algorithm as a password. The algorithm might be an integer function, for example. The system selects a random integer and presents it to the user. The user applies the function and replies with the correct result. The system also applies the function. If the two results match, access is allowed.

Such algorithmic passwords are not susceptible to reuse; that is, a user can type in a password, and no entity intercepting that password will be able to reuse it. In this variation, the system and the user share a secret. The secret is never transmitted over a medium that allows exposure. Rather, the secret is used as input to the function, along with a shared seed. A **seed** is a random number or alphanumeric sequence. The seed is the authentication challenge from the computer. The secret and the seed are used as input to the function $f(secret, seed)$. The result of this function is transmitted as the password to the computer. Because the computer also knows the secret and the seed, it can perform the same computation. If the results match, the user is authenticated. The next time the user needs to be authenticated, another seed is generated, and the same steps ensue. This time, the password is different.

In this **one-time password** system, the password is different in each instance. Anyone capturing the password from one session and trying to reuse it in another session will fail. One-time passwords are among the only ways to prevent improper authentication due to password exposure.

One-time password systems are implemented in various ways. Commercial implementations, such as SecurID, use hardware calculators. Most of these calculators are shaped like a credit card, a key-chain dangle, or a USB device; they include a display and may or may not also have a keypad. Some use the current time as the random seed. Others require that the user enters the shared secret, also known as a **personal identification number** or **PIN**, on the keypad. The display then shows the one-time password. The use of both a one-time password generator and a PIN is one form of **two-factor authentication**. Two different types of components are needed in this case. Two-factor authentication offers far better authentication protection than single-factor authentication.

Another variation on one-time passwords is the use of a **code book**, or **one-time pad**, which is a list of single-use passwords. In this method, each password on the list is used, in order, once, and then is crossed out or erased. The commonly used S/Key system uses either a software calculator or a code book based on these calculations as a source of one-time passwords. Of course, the user must protect his code book.

18.5.5 Biometrics

Another variation on the use of passwords for authentication involves the use of biometric measures. Palm- or hand-readers are commonly used to secure physical access—for example, access to a data center. These readers match stored parameters against what is being read from hand-reader pads. The parameters can include a temperature map, as well as finger length, finger width, and line patterns. These devices are currently too large and expensive to be used for normal computer authentication.

Fingerprint readers have become accurate and cost-effective and should become more common in the future. These devices read your finger's ridge patterns and convert them into a sequence of numbers. Over time, they can store a set of sequences to adjust for the location of the finger on the reading pad and other factors. Software can then scan a finger on the pad and compare its features with these stored sequences to determine if the finger on the pad is the same as the stored one. Of course, multiple users can have profiles stored, and the scanner can differentiate among them. A very accurate two-factor authentication scheme can result from requiring a password as well as a user name and fingerprint scan. If this information is encrypted in transit, the system can be very resistant to spoofing or replay attack.

Multi-factor authentication is better still. Consider how strong authentication can be with a USB device that must be plugged into the system, a PIN, and a fingerprint scan. Except for the user's having to place her finger on a pad and plug the USB into the system, this authentication method is no less convenient that using normal passwords. Recall, though, that strong authentication by itself is not sufficient to guarantee the ID of the user. An authenticated session can still be hijacked, if it is not encrypted.

18.6

Just as there are myriad threats to system and network security, there are many security solutions. The solutions run the gamut from improved user education, through technology, to writing bug-free software. Most security professionals subscribe to the theory of **defense in depth,** which states that more layers of defense are better than fewer layers. Of course, this theory applies to any kind of security. Consider the security of a house without a door lock, with a door lock, and with a lock and an alarm. In this section, we look at the major methods, tools, and techniques that can be used to improve resistance to threats.

18.6.1 Security Policy

The first step toward improving the security of any aspect of computing is to have a **security policy**. Policies vary widely but generally include a statement of what is being secured. For example, a policy might state that all outside-accessible applications must have a code review before being deployed, or that users should not share their passwords, or that all connection points between a company and the outside must have port scans run every six months. Without a policy in place, it is impossible for users and administrators to know what is permissible, what is required, and what is not allowed. The policy is a road map to security, and if a site is trying to move from less secure to more secure, it needs a map to know how to get there.

Once the security policy is in place, the people it affects should know it well. It should be their guide. The policy should also be a **living document** that is reviewed and updated periodically to ensure that it is still pertinent and still followed.

18.6.2 Vulnerability Assessment

How can we determine whether a security policy has been correctly implemented? The best way is to execute a vulnerability assessment. Such assessments can cover broad ground, from social engineering through **risk assessment** to port scans. For example, risk assessment endeavors to value the assets of the entity in question (a program, a management team, a system, or a facility) and determine the odds that a security incident will affect the entity and decrease its value. When the odds of suffering a loss and the amount of the potential loss are known, a value can be placed on trying to secure the entity.

The core activity of most vulnerability assessments is a **penetration test**, in which the entity is scanned for known vulnerabilities. Because this book is concerned with operating systems and the software that runs on them, we will concentrate on those aspects.

Vulnerability scans typically are done at times when computer use is relatively low, to minimize their impact. When appropriate, they are done on test systems rather than production systems because they can induce unhappy behavior from the target systems or network devices.

A scan within an individual system can check a variety of aspects of the system:

- Short or easy-to-guess passwords
- Unauthorized privileged programs, such as *setuid* programs
- Unauthorized programs in system directories
- Unexpectedly long-running processes
- Improper directory protections on user and system directories
- Improper protections on system data files, such as the password file, device drivers, or the operating-system kernel itself
- Dangerous entries in the program search path (for example, the Trojan horse discussed in Section 18.2.1)
- Changes to system programs detected with checksum values
- Unexpected or hidden network daemons

Any problems found by a security scan can be either fixed automatically or reported to the managers of the system.

Networked computers are much more susceptible to security attacks than are standalone systems. Rather than attacks from a known set of access points, such as directly connected terminals, we face attacks from an unknown and large set of access points—a potentially severe security problem. To a lesser extent, systems connected to telephone lines via modems are also more exposed.

In fact, the U.S. government considers a system to be only as secure as its most far-reaching connection. For instance, a top-secret system may be accessed only from within a building also considered top-secret. The system loses its top-secret rating if any form of communication can occur outside that environment. Some government facilities take extreme security precautions. The connectors that plug a terminal into the secure computer are locked in a safe in the office when the terminal is not in use. A person must have proper ID to gain access to the building and her office, must know a physical lock combination, and must know authentication information for the computer itself to gain access to the computer—an example of multi-factor authentication.

Unfortunately for systems administrators and computer-security professionals, it is frequently impossible to lock a machine in a room and disallow all remote access. For instance, the Internet network currently connects millions of computers. It is becoming a mission-critical, indispensable resource for many companies and individuals. If you consider the Internet a club, then, as in any club with millions of members, there are many good members and some bad members. The bad members have many tools they can use to attempt to gain access to the interconnected computers, just as Morris did with his worm.

Vulnerability scans can be applied to networks to address some of the problems with network security. The scans search a network for ports that respond to a request. If services are enabled that should not be, access to them can be blocked, or they can be disabled. The scans then determine the details of the application listening on that port and try to determine if each has any known vulnerabilities. Testing those vulnerabilities can determine if the system is misconfigured or is lacking needed patches.

Finally, though, consider the use of port scanners in the hands of a cracker rather than someone trying to improve security. These tools could help crackers find vulnerabilities to attack. (Fortunately, it is possible to detect port scans through anomaly detection, as we discuss next.) It is a general challenge to security that the same tools can be used for good and for harm. In fact, some people advocate **security through obscurity**, stating that tools should not be written to test security so that security holes will be harder to find (and exploit). Others believe that this approach to security is not a valid one, pointing out, for example, that crackers could write their own tools. It seems reasonable that security through obscurity be considered one of the layers of security only so long as it is not the only layer. For example, a company could publish its entire network configuration information; but keeping that information secret makes it harder for intruders to know what to attack or to determine what might be detected. Even here, though, a company assuming that such information will remain a secret has a false sense of security.

18.6.3 Intrusion Detection

Securing systems and facilities is intimately linked to intrusion detection. **Intrusion detection**, as its name suggests, strives to detect attempted or successful intrusions into computer systems and to initiate appropriate responses to the intrusions. Intrusion detection encompasses a wide array of techniques that vary on a number of axes. These axes include:

- The time that detection occurs. Detection can occur in real time (while the intrusion is occurring) or after the fact.
- The types of inputs examined to detect intrusive activity. These may include user-shell commands, process system calls, and network packet headers or contents. Some forms of intrusion might be detected only by correlating information from several such sources.
- The range of response capabilities. Simple forms of response include alerting an administrator to the potential intrusion or somehow halting the potentially intrusive activity—for example, killing a process engaged in apparently intrusive activity. In a sophisticated form of response, a system might transparently divert an intruder's activity to a **honeypot**—a false resource exposed to the attacker. The resource appears real to the attacker and enables the system to monitor and gain information about the attack.

These degrees of freedom in the design space for detecting intrusions have yielded a wide range of solutions, known as **intrusion-detection systems** (IDSs) and **intrusion-prevention systems** (IDPs). IDS systems raise an alarm when an intrusion is detected, while IDP systems act as routers, passing traffic unless an intrusion is detected (at which point that traffic is blocked).

But just what constitutes an intrusion? Defining a suitable specification of intrusion turns out to be quite difficult, and thus automatic IDSs and IDPs today typically settle for one of two less ambitious approaches. In the first, called **signature-based detection**, system input or network traffic is examined for specific behavior patterns (or **signatures**) known to indicate attacks. A simple example of signature-based detection is scanning network packets for the string */etc/passwd/* targeted for a UNIX system. Another example is virus-detection software, which scans binaries or network packets for known viruses.

The second approach, typically called **anomaly detection**, attempts through various techniques to detect anomalous behavior within computer systems. Of course, not all anomalous system activity indicates an intrusion, but the presumption is that intrusions often induce anomalous behavior. An example of anomaly detection is monitoring system calls of a daemon process to detect whether the system-call behavior deviates from normal patterns, possibly indicating that a buffer overflow has been exploited in the daemon to corrupt its behavior. Another example is monitoring shell commands to detect anomalous commands for a given user or detecting an anomalous login time for a user, either of which may indicate that an attacker has succeeded in gaining access to that user's account.

Signature-based detection and anomaly detection can be viewed as two sides of the same coin: Signature-based detection attempts to characterize dangerous behaviors and detects when one of these behaviors occurs, whereas anomaly detection attempts to characterize normal (or non-dangerous) behaviors and detects when something other than these behaviors occurs.

These different approaches yield IDSs and IDPs with very different properties, however. In particular, anomaly detection can detect previously unknown methods of intrusion (so-called **zero-day attacks**). Signature-based detection, in contrast, will identify only known attacks that can be codified in a recognizable pattern. Thus, new attacks that were not contemplated when the signatures were generated will evade signature-based detection. This problem is well known to vendors of virus-detection software, who must release new signatures with great frequency as new viruses are detected manually.

Anomaly detection is not necessarily superior to signature-based detection, however. Indeed, a significant challenge for systems that attempt anomaly detection is to benchmark "normal" system behavior accurately. If the system is already penetrated when it is benchmarked, then the intrusive activity may be included in the "normal" benchmark. Even if the system is benchmarked cleanly, without influence from intrusive behavior, the benchmark must give a fairly complete picture of normal behavior. Otherwise, the number of **false positives** (false alarms) or, worse, **false negatives** (missed intrusions) will be excessive.

To illustrate the impact of even a marginally high rate of false alarms, consider an installation consisting of a hundred UNIX workstations from which records of security-relevant events are recorded for purposes of intrusion detection. A small installation such as this could easily generate a million audit records per day. Only one or two might be worthy of an administrator's investigation. If we suppose, optimistically, that each such attack is reflected in ten audit records, we can then roughly compute the rate of occurrence of audit records reflecting truly intrusive activity as

$$\frac{2\frac{\text{intrusions}}{\text{day}} \cdot 10\frac{\text{records}}{\text{intrusion}}}{10^6\frac{\text{records}}{\text{day}}} = 0.00002.$$

Interpreting this as a "probability of occurrence of intrusive records," we denote it as $P(I)$; that is, event I is the occurrence of a record reflecting truly intrusive behavior. Since $P(I) = 0.00002$, we also know that $P(\neg I) = 1 - P(I) = 0.99998$. Now let A denote the raising of an alarm by an IDS. An accurate IDS should maximize both $P(I \mid A)$ and $P(\neg I \mid \neg A)$—that is, the probabilities that an alarm indicates an intrusion and that no alarm indicates no intrusion. Focusing on $P(I \mid A)$ for the moment, we can compute it using **Bayes' theorem**:

$$\begin{aligned} P(I \mid A) &= \frac{P(I) \cdot P(A \mid I)}{P(I) \cdot P(A \mid I) + P(\neg I) \cdot P(A \mid \neg I)} \\ &= \frac{0.00002 \cdot P(A \mid I)}{0.00002 \cdot P(A \mid I) + 0.99998 \cdot P(A \mid \neg I)} \end{aligned}$$

Now consider the impact of the false-alarm rate $P(A \mid \neg I)$ on $P(I \mid A)$. Even with a very good true-alarm rate of $P(A \mid I) = 0.8$, a seemingly good false-alarm rate of $P(A \mid \neg I) = 0.0001$ yields $P(I \mid A) \approx 0.14$. That is, fewer than one in every seven alarms indicates a real intrusion! In systems where a security administrator investigates each alarm, a high rate of false alarms—called a "Christmas tree effect"—is exceedingly wasteful and will quickly teach the administrator to ignore alarms.

This example illustrates a general principle for IDSs and IDPs: For usability, they must offer an extremely low false-alarm rate. Achieving a sufficiently low false-alarm rate is an especially serious challenge for anomaly-detection systems, as mentioned, because of the difficulties of adequately benchmarking normal system behavior. However, research continues to improve anomaly-detection techniques. Intrusion detection software is evolving to implement signatures, anomaly algorithms, and other algorithms and to combine the results to arrive at a more accurate anomaly-detection rate.

18.6.4 Virus Protection

As we have seen, viruses can and do wreak havoc on systems. Protection from viruses thus is an important security concern. Antivirus programs are often used to provide this protection. Some of these programs are effective against only particular known viruses. They work by searching all the programs on a system for the specific pattern of instructions known to make up the virus. When they find a known pattern, they remove the instructions, **disinfecting** the program. Antivirus programs may have catalogs of thousands of viruses for which they search.

Both viruses and antivirus software continue to become more sophisticated. Some viruses modify themselves as they infect other software to avoid the basic pattern-match approach of antivirus programs. Antivirus programs in turn now look for families of patterns rather than a single pattern to identify a virus. In fact, some anti-virus programs implement a variety of detection algorithms.

They can decompress compressed viruses before checking for a signature. Some also look for process anomalies. A process opening an executable file for writing is suspicious, for example, unless it is a compiler. Another popular technique is to run a program in a **sandbox**, which is a controlled or emulated section of the system. The antivirus software analyzes the behavior of the code in the sandbox before letting it run unmonitored. Some antivirus programs also put up a complete shield rather than just scanning files within a file system. They search boot sectors, memory, inbound and outbound e-mail, files as they are downloaded, files on removable devices or media, and so on.

The best protection against computer viruses is prevention, or the practice of **safe computing**. Purchasing unopened software from vendors and avoiding free or pirated copies from public sources or disk exchange offer the safest route to preventing infection. However, even new copies of legitimate software applications are not immune to virus infection: There have been cases where disgruntled employees of a software company have infected the master copies of software programs to do economic harm to the company selling the software. For macro viruses, one defense is to exchange Word documents in an alternative file format called **rich text format (RTF)**. Unlike the native Word format, RTF does not include the capability to attach macros.

Another defense is to avoid opening any e-mail attachments from unknown users. Unfortunately, history has shown that e-mail vulnerabilities appear as fast as they are fixed. For example, in 2000, the *love bug* virus became very widespread by appearing to be a love note sent by a friend of the receiver. Once the attached Visual Basic script was opened, the virus propagated by sending itself to the first users in the user's e-mail contact list. Fortunately, except for clogging e-mail systems and users' inboxes, it was relatively harmless. It did, however, effectively negate the defensive strategy of opening attachments only from people known to the receiver. A more effective defense method is to avoid opening any e-mail attachment that contains executable code. Some companies now enforce this as policy by removing all incoming attachments to e-mail messages.

Another safeguard, although it does not prevent infection, does permit early detection. A user must begin by completely reformatting the hard disk, especially the boot sector, which is often targeted for viral attack. Only secure software is uploaded, and a signature of each program is taken via a secure message-digest computation. The resulting filename and associated message-digest list must then be kept free from unauthorized access. Periodically, or each time a program is run, the operating system recomputes the signature and compares it with the signature on the original list; any differences serve as a warning of possible infection. This technique can be combined with others. For example, a high-overhead antivirus scan, such as a sandbox, can be used; and if a program passes the test, a signature can be created for it. If the signatures match the next time the program is run, it does not need to be virus-scanned again.

18.6.5 Auditing, Accounting, and Logging

Auditing, accounting, and logging can decrease system performance, but they are useful in several areas, including security. Logging can be general or specific. All system-call executions can be logged for analysis of program behavior (or misbehavior). More typically, suspicious events are logged. Authentication failures and authorization failures can tell us quite a lot about break-in attempts.

Accounting is another potential tool in a security administrator's kit. It can be used to find performance changes, which in turn can reveal security problems. One of the early UNIX computer break-ins was detected by Cliff Stoll when he was examining accounting logs and spotted an anomaly.

18.7 [illegible]

We turn next to the question of how a trusted computer can be connected safely to an untrustworthy network. One solution is the use of a firewall to separate trusted and untrusted systems. A **firewall** is a computer, appliance, or router that sits between the trusted and the untrusted. A network firewall limits network access between the two **security domains** and monitors and logs all connections. It can also limit connections based on source or destination address, source or destination port, or direction of the connection. For instance, web servers use HTTP to communicate with web browsers. A firewall therefore may allow only HTTP to pass from all hosts outside the firewall to the web server within the firewall. The Morris Internet worm used the `finger` protocol to break into computers, so `finger` would not be allowed to pass, for example.

In fact, a network firewall can separate a network into multiple domains. A common implementation has the Internet as the untrusted domain; a semi-trusted and semi-secure network, called the **demilitarized zone (DMZ)**, as another domain; and a company's computers as a third domain (Figure 18.10). Connections are allowed from the Internet to the DMZ computers and from the company computers to the Internet but are not allowed from the Internet or DMZ computers to the company computers. Optionally, controlled

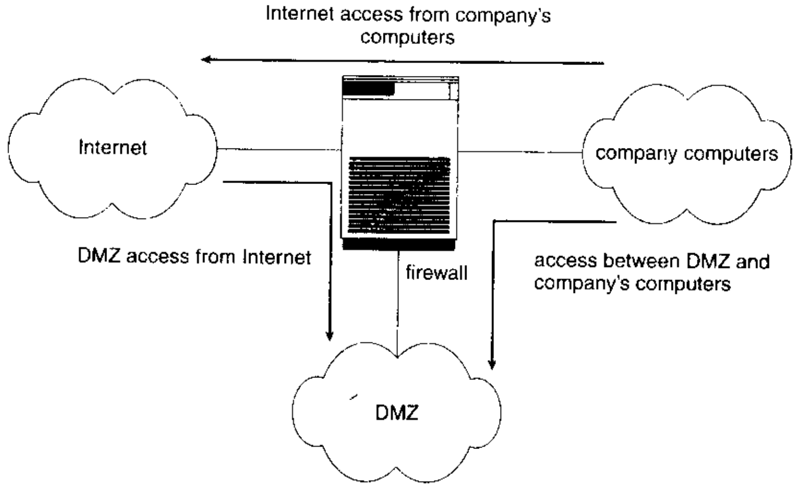

Internet access from company's computers
Internet
company computers
DMZ access from Internet
firewall
access between DMZ and company's computers
DMZ

Figure 18.10 Domain separation via firewall.

communications may be allowed between the DMZ and one company computer or more. For instance, a web server on the DMZ may need to query a database server on the corporate network. With a firewall, however, access is contained, and any DMZ systems that are broken into still are unable to access the company computers.

Of course, a firewall itself must be secure and attack-proof; otherwise, its ability to secure connections can be compromised. Furthermore, firewalls do not prevent attacks that **tunnel**, or travel within protocols or connections that the firewall allows. A buffer-overflow attack to a web server will not be stopped by the firewall, for example, because the HTTP connection is allowed; it is the contents of the HTTP connection that house the attack. Likewise, denial-of-service attacks can affect firewalls as much as any other machines. Another vulnerability of firewalls is **spoofing**, in which an unauthorized host pretends to be an authorized host by meeting some authorization criterion. For example, if a firewall rule allows a connection from a host and identifies that host by its IP address, then another host could send packets using that same address and be allowed through the firewall.

In addition to the most common network firewalls, there are other, newer kinds of firewalls, each with its pros and cons. A **personal firewall** is a software layer either included with the operating system or added as an application. Rather than limiting communication between security domains, it limits communication to (and possibly from) a given host. A user could add a personal firewall to her PC so that a Trojan horse would be denied access to the network to which the PC is connected. An **application proxy firewall** understands the protocols that applications speak across the network. For example, SMTP is used for mail transfer. An application proxy accepts a connection just as an SMTP server would and then initiates a connection to the original destination SMTP server. It can monitor the traffic as it forwards the message, watching for and disabling illegal commands, attempts to exploit bugs, and so on. Some firewalls are designed for one specific protocol. An **XML firewall**, for example, has the specific purpose of analyzing XML traffic and blocking disallowed or malformed XML. **System-call firewalls** sit between applications and the kernel, monitoring system-call execution. For example, in Solaris 10, the "least privilege" feature implements a list of more than fifty system calls that processes may or may not be allowed to make. A process that does not need to spawn other processes can have that ability taken away, for instance.

18.8 Computer-Security Classifications

The U.S. Department of Defense Trusted Computer System Evaluation Criteria specify four security classifications in systems: A, B, C, and D. This specification is widely used to determine the security of a facility and to model security solutions, so we explore it here. The lowest-level classification is division D, or minimal protection. Division D includes only one class and is used for systems that have failed to meet the requirements of any of the other security classes. For instance, MS-DOS and Windows 3.1 are in division D.

Division C, the next level of security, provides discretionary protection and accountability of users and their actions through the use of audit capabilities.

Division C has two levels: C1 and C2. A C1-class system incorporates some form of controls that allow users to protect private information and to keep other users from accidentally reading or destroying their data. A C1 environment is one in which cooperating users access data at the same levels of sensitivity. Most versions of UNIX are C1 class.

The sum total of all protection systems within a computer system (hardware, software, firmware) that correctly enforce a security policy is known as a **trusted computer base (TCB)**. The TCB of a C1 system controls access between users and files by allowing the user to specify and control sharing of objects by named individuals or defined groups. In addition, the TCB requires that the users identify themselves before they start any activities that the TCB is expected to mediate. This identification is accomplished via a protected mechanism or password; the TCB protects the authentication data so that they are inaccessible to unauthorized users.

A C2-class system adds an individual-level access control to the requirements of a C1 system. For example, access rights of a file can be specified to the level of a single individual. In addition, the system administrator can selectively audit the actions of any one or more users based on individual identity. The TCB also protects itself from modification of its code or data structures. In addition, no information produced by a prior user is available to another user who accesses a storage object that has been released back to the system. Some special, secure versions of UNIX have been certified at the C2 level.

Division-B mandatory-protection systems have all the properties of a class-C2 system; in addition, they attach a sensitivity label to each object. The B1-class TCB maintains the security label of each object in the system; the label is used for decisions pertaining to mandatory access control. For example, a user at the confidential level could not access a file at the more sensitive secret level. The TCB also denotes the sensitivity level at the top and bottom of each page of any human-readable output. In addition to the normal user-name–password authentication information, the TCB also maintains the clearance and authorizations of individual users and will support at least two levels of security. These levels are hierarchical, so that a user may access any objects that carry sensitivity labels equal to or lower than his security clearance. For example, a secret-level user could access a file at the confidential level in the absence of other access controls. Processes are also isolated through the use of distinct address spaces.

A B2-class system extends the sensitivity labels to each system resource, such as storage objects. Physical devices are assigned minimum and maximum security levels that the system uses to enforce constraints imposed by the physical environments in which the devices are located. In addition, a B2 system supports covert channels and the auditing of events that could lead to the exploitation of a covert channel.

A B3-class system allows the creation of access-control lists that denote users or groups *not* granted access to a given named object. The TCB also contains a mechanism to monitor events that may indicate a violation of security policy. The mechanism notifies the security administrator and, if necessary, terminates the event in the least disruptive manner.

The highest-level classification is division A. Architecturally, a class-A1 system is functionally equivalent to a B3 system, but it uses formal design specifications and verification techniques, granting a high degree of assurance that the TCB has been implemented correctly. A system beyond class A1 might be designed and developed in a trusted facility by trusted personnel.

The use of a TCB merely ensures that the system can enforce aspects of a security policy; the TCB does not specify what the policy should be. Typically, a given computing environment develops a security policy for **certification** and has the plan **accredited** by a security agency, such as the National Computer Security Center. Certain computing environments may require other certification, such as that supplied by TEMPEST, which guards against electronic eavesdropping. For example, a TEMPEST-certified system has terminals that are shielded to prevent electromagnetic fields from escaping. This shielding ensures that equipment outside the room or building where the terminal is housed cannot detect what information is being displayed by the terminal.

18.9 [illegible]

Microsoft Windows XP is a general-purpose operating system designed to support a variety of security features and methods. In this section, we examine features that Windows XP uses to perform security functions. For more information and background on Windows XP, see Chapter 22.

The Windows XP security model is based on the notion of **user accounts**. Windows XP allows the creation of any number of user accounts, which can be grouped in any manner. Access to system objects can then be permitted or denied as desired. Users are identified to the system by a *unique* security ID. When a user logs on, Windows XP creates a **security access token** that includes the security ID for the user, security IDs for any groups of which the user is a member, and a list of any special privileges that the user has. Examples of special privileges include backing up files and directories, shutting down the computer, logging on interactively, and changing the system clock. Every process that Windows XP runs on behalf of a user will receive a copy of the access token. The system uses the security IDs in the access token to permit or deny access to system objects whenever the user, or a process on behalf of the user, attempts to access the object. Authentication of a user account is typically accomplished via a user name and password, although the modular design of Windows XP allows the development of custom authentication packages. For example, a retinal (or eye) scanner might be used to verify that the user is who she says she is.

Windows XP uses the idea of a subject to ensure that programs run by a user do not get greater access to the system than the user is authorized to have. A **subject** is used to track and manage permissions for each program that a user runs; it is composed of the user's access token and the program acting on behalf of the user. Since Windows XP operates with a client–server model, two classes of subjects are used to control access: simple subjects and server subjects. An example of a **simple subject** is the typical application program that a user executes after she logs on. The simple subject is assigned a **security context** based on the security access token of the user. A **server subject** is a process implemented as a protected server that uses the security context of the client when acting on the client's behalf.

As mentioned in Section 18.7, auditing is a useful security technique. Windows XP has built-in auditing that allows many common security threats to be monitored. Examples include failure auditing for login and logoff events to detect random password break-ins, success auditing for login and logoff events to detect login activity at strange hours, success and failure write-access auditing for executable files to track a virus outbreak, and success and failure auditing for file access to detect access to sensitive files.

Security attributes of an object in Windows XP are described by a **security descriptor**. The security descriptor contains the security ID of the owner of the object (who can change the access permissions), a group security ID used only by the POSIX subsystem, a discretionary access-control list that identifies which users or groups are allowed (and which are not allowed) access, and a system access-control list that controls which auditing messages the system will generate. For example, the security descriptor of the file *foo.bar* might have owner avi and this discretionary access-control list:

avi—all access

group cs—read–write access

user cliff—no access

In addition, it might have a system access-control list of audit writes by everyone.

An access-control list is composed of access-control entries that contain the security ID of the individual and an access mask that defines all possible actions on the object, with a value of AccessAllowed or AccessDenied for each action. Files in Windows XP may have the following access types: `ReadData`, `WriteData`, `AppendData`, `Execute`, `ReadExtendedAttribute`, `WriteExtendedAttribute`, `ReadAttributes`, and `WriteAttributes`. We can see how this allows a fine degree of control over access to objects.

Windows XP classifies objects as either container objects or noncontainer objects. **Container objects**, such as directories, can logically contain other objects. By default, when an object is created within a container object, the new object inherits permissions from the parent object. Similarly, if the user copies a file from one directory to a new directory, the file will inherit the permissions of the destination directory. **Noncontainer objects** inherit no other permissions. Furthermore, if a permission is changed on a directory, the new permissions do not automatically apply to existing files and subdirectories; the user may explicitly apply them if she so desires.

The system administrator can prohibit printing to a printer on the system for all or part of a day and can use the Windows XP Performance Monitor to help her spot approaching problems. In general, Windows XP does a good job of providing features to help ensure a secure computing environment. Many of these features are not enabled by default, however, which may be one reason for the myriad security breaches on Windows XP systems. Another reason is the vast number of services Windows XP starts at system boot time and the number of applications that typically are installed on a Windows XP system. For a real multiuser environment, the system administrator should formulate a security plan and implement it, using the features that Windows XP provides and other security tools.

18.10

Protection is an internal problem. Security, in contrast, must consider both the computer system and the environment—people, buildings, businesses, valuable objects, and threats—within which the system is used.

The data stored in the computer system must be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. It is easier to protect against accidental loss of data consistency than to protect against malicious access to the data. Absolute protection of the information stored in a computer system from malicious abuse is not possible; but the cost to the perpetrator can be made sufficiently high to deter most, if not all, attempts to access that information without proper authority.

Several types of attacks can be launched against programs and against individual computers or the masses. Stack- and buffer-overflow techniques allow successful attackers to change their level of system access. Viruses and worms are self-perpetuating, sometimes infecting thousands of computers. Denial-of-service attacks prevent legitimate use of target systems.

Encryption limits the domain of receivers of data, while authentication limits the domain of senders. Encryption is used to provide confidentiality of data being stored or transferred. Symmetric encryption requires a shared key, while asymmetric encryption provides a public key and a private key. Authentication, when combined with hashing, can prove that data have not been changed.

User authentication methods are used to identify legitimate users of a system. In addition to standard user-name and password protection, several authentication methods are used. One-time passwords, for example, change from session to session to avoid replay attacks. Two-factor authentication requires two forms of authentication, such as a hardware calculator with an activation PIN. Multi-factor authentication uses three or more forms. These methods greatly decrease the chance of authentication forgery.

Methods of preventing or detecting security incidents include intrusion-detection systems, antivirus software, auditing and logging of system events, monitoring of system software changes, system-call monitoring, and firewalls,.

18.1 Buffer-overflow attacks can be avoided by adopting a better programming methodology or by using special hardware support. Discuss these solutions.

18.2 A password may become known to other users in a variety of ways. Is there a simple method for detecting that such an event has occurred? Explain your answer.

18.3 What is the purpose of using a "salt" along with the user-provided password? Where should the "salt" be stored, and how should it be used?

18.4 An experimental addition to UNIX allows a user to connect a **watchdog** program to a file. The watchdog is invoked whenever a program requests access to the file. The watchdog then either grants or denies access to the file. Discuss two pros and two cons of using watchdogs for security.

18.5 Discuss a means by which managers of systems connected to the Internet could have designed their systems to limit or eliminate the damage done by a worm. What are the drawbacks of making the change that you suggest?

18.6 Argue for or against the judicial sentence handed down against Robert Morris, Jr., for his creation and execution of the Internet worm discussed in Section 18.3.1.

18.7 What are two advantages of encrypting data stored in the computer system?

18.8 What commonly used computer programs are prone to man-in-the-middle attacks? Discuss solutions for preventing this form of attack.

18.9 Why doesn't $D(k_e, N)(E(k_d, N)(m))$ provide authentication of the sender? To what uses can such an encryption be put?

18.10 Discuss how the asymmetric encryption algorithm can be used to achieve the following goals.

a. Authentication: the receiver knows that only the sender could have generated the message

b. Secrecy: only the receiver can decrypt the message.

c. Authentication and secrecy: only the receiver can decrypt the message, and the receiver knows that only the sender could have generated the message.

General discussions concerning security are given by Hsiao et al. [1979], Landwehr [1981], Denning [1982], Pfleeger and Pfleeger [2003], Tanenbaum 2003, and Russell and Gangemi [1991]. Also of general interest is the text by Lobel [1986]. Computer networking is discussed in Kurose and Ross [2005].

Issues concerning the design and verification of secure systems are discussed by Rushby [1981] and by Silverman [1983]. A security kernel for a multiprocessor microcomputer is described by Schell [1983]. A distributed secure system is described by Rushby and Randell [1983].

Morris and Thompson [1979] discuss password security. Morshedian [1986] presents methods to fight password pirates. Password authentication with insecure communications is considered by Lamport [1981]. The issue of password cracking is examined by Seely [1989]. Computer break-ins are discussed by Lehmann [1987] and by Reid [1987]. Issues related to trusting computer programs are discussed in Thompson [1984].

Discussions concerning UNIX security are offered by Grampp and Morris [1984], Wood and Kochan [1985], Farrow [1986b], Farrow [1986a], Filipski and Hanko [1986], Hecht et al. [1988], Kramer [1988], and Garfinkel et al. [2003].

Bershad and Pinkerton [1988] present the watchdog extension to BSD UNIX. The COPS security-scanning package for UNIX was written by Farmer at Purdue University. It is available to users on the Internet via the FTP program from host ftp.uu.net in directory /pub/security/cops.

Spafford [1989] presents a detailed technical discussion of the Internet worm. The Spafford article appears with three others in a special section on the Morris Internet worm in *Communications of the ACM* (Volume 32, Number 6, June 1989).

Security problems associated with the TCP/IP protocol suite are described in Bellovin [1989]. The mechanisms commonly used to prevent such attacks are discussed in Cheswick et al. [2003]. Another approach to protecting networks from insider attacks is to secure topology or route discovery. Kent et al. [2000], Hu et al. [2002], Zapata and Asokan [2002], and Hu and Perrig [2004] present solutions for secure routing. Savage et al. [2000] examine the distributed denial-of-service attack and propose IP trace-back solutions to address the problem. Perlman [1988] proposes an approach to diagnose faults when the network contains malicious routers.

Information about viruses and worms can be found at http://www.viruslist.com, as well as in Ludwig [1998] and Ludwig [2002]. Other web sites containing up-to-date security information include http://www.trusecure.com and httpd://www.eeye.com. A paper on the dangers of a computer monoculture can be found at http://www.ccianet.org/papers/cyberinsecurity.pdf.

Diffie and Hellman [1976] and Diffie and Hellman [1979] were the first researchers to propose the use of the public-key encryption scheme. The algorithm presented in Section 18.4.1 is based on the public-key encryption scheme; it was developed by Rivest et al. [1978]. Lempel [1979], Simmons [1979], Denning and Denning [1979], Gifford [1982], Denning [1982], Ahituv et al. [1987], Schneier [1996], and Stallings [2003] explore the use of cryptography in computer systems. Discussions concerning protection of digital signatures are offered by Akl [1983], Davies [1983], Denning [1983], and Denning [1984].

The U.S. government is, of course, concerned about security. The *Department of Defense Trusted Computer System Evaluation Criteria* (DoD [1985]), known also as the *Orange Book*, describes a set of security levels and the features that an operating system must have to qualify for each security rating. Reading it is a good starting point for understanding security concerns. The *Microsoft Windows NT Workstation Resource Kit* (Microsoft [1996]) describes the security model of NT and how to use that model.

The RSA algorithm is presented in Rivest et al. [1978]. Information about NIST's AES activities can be found at http://www.nist.gov/aes/; information about other cryptographic standards for the United States can also be found at that site. More complete coverage of SSL 3.0 can be found at http://home.netscape.com/eng/ssl3/. In 1999, SSL 3.0 was modified slightly and presented in an IETF Request for Comments (RFC) under the name TLS.

The example in Section 18.6.3 illustrating the impact of false-alarm rate on the effectiveness of IDSs is based on Axelsson [1999]. A more complete description of the swatch program and its use with syslog can be found in Hansen and Atkins [1993]. The description of Tripwire in Section 18.6.5 is based on Kim and Spafford [1993]. Research into system-call-based anomaly detection is described in Forrest et al. [1996].

Part Eight

Our coverage of operating-system issues thus far has focused mainly on general-purpose computing systems. There are, however, special-purpose systems with requirements different from those of many of the systems we have described.

A *real-time system* is a computer system that requires not only that computed results be "correct" but also that the results be produced within a specified deadline period. Results produced after the deadline has passed—even if correct—may be of no real value. For such systems, many traditional operating-system scheduling algorithms must be modified to meet the stringent timing deadlines.

A *multimedia system* must be able to handle not only conventional data, such as text files, programs, and word-processing documents, but also multimedia data. Multimedia data consist of continuous-media data (audio and video) as well as conventional data. Continuous-media data—such as frames of video—must be delivered according to certain time restrictions (for example, 30 frames per second). The demands of handling continuous-media data require significant changes in operating-system structure, most notably in memory, disk, and network management.