

Chapter 3

General Concepts

This chapter introduces, at an elementary level, some general concepts of computer security that apply to all applications; it also introduces terms that will be used repeatedly in later chapters. Many of the topics discussed here will be covered later in more detail.

3.1 INTERNAL AND EXTERNAL SECURITY

Most of this book addresses *internal security* controls that are implemented within the hardware and software of the system. For these internal controls to be effective, however, they must be accompanied by adequate *external security* controls that govern physical access to the system.

External controls cover all activities for maintaining security of the system that the system itself cannot address. External controls can be divided into three classes:

- Physical security
- Personnel security
- Procedural security

Physical security controls (locked rooms, guards, and the like) are an integral part of the security solution for a central computing facility, but they alone cannot address the security problems of multiuser distributed systems. As networking becomes a more and more pervasive part of computing, the role of physical security will continue to diminish. In a large heterogeneous network, it is probably impossible to guarantee (and risky to assume) that any system other than your own is physically protected.

Personnel security covers techniques that an employer uses in deciding whom to trust with the organization's system and with its information. Most governments have procedures whereby a level of security clearance is assigned to individuals based on a personal background investigation and (possibly) additional measures such as polygraph examinations. These procedures allow the government to assign different degrees of trust to different people, depending on the needs of their particular job and the depth of their investigation. Personnel screening in industry is far less formal than in government, and people are usually given "all or none" access. Where selective access to information is required, it is determined on a case-by-case basis.

Procedural security covers the processes of granting people access to machines, handling physical input and output (such as printouts and tapes), installing system software, attaching user terminals, and performing countless other details of daily system administration.

Internal and external controls go hand in hand, and it is possible to trade off a control in one area for a control in the other. For example, even the most primitive multiuser systems today have password protection. The password mechanism is an internal control that obviates the need for external controls such as locked terminal rooms. In designing a secure system, we generally strive to minimize the need for external controls, because external controls are usually far more expen-

sive to implement. Procedural controls are also notoriously errorprone, since they rely on people each time they are invoked.

3.2 THE SYSTEM BOUNDARY AND THE SECURITY PERIMETER

A system is a vague entity that comprises the totality of the computing and communications environment over which the developers have some control. Everything inside the system is protected by the system, and everything outside it is unprotected (fig. 3-1). What is important is not the generic definition of the term *system* but the definition as it applies in each particular case. In any effort to plan for security features, it is crucial to establish a clear understanding of the *system boundary* and to define the threats (originating outside the boundary) against which the system must defend itself. You cannot construct a coherent security environment without understanding the threats.

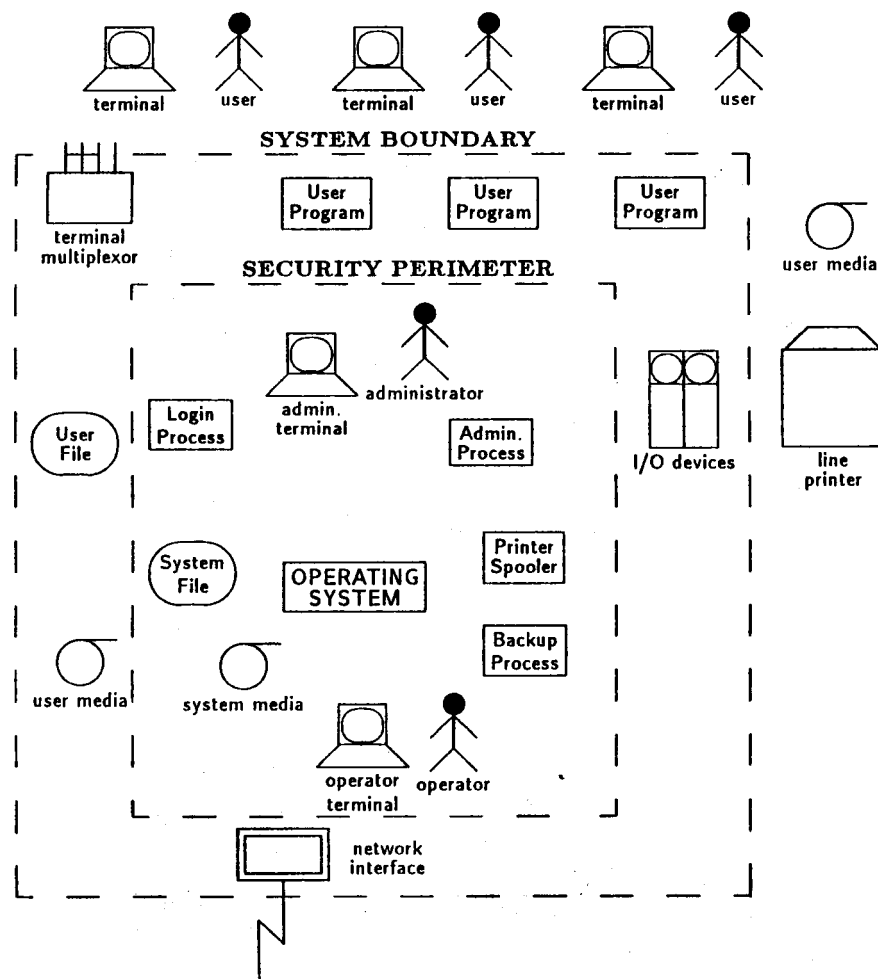


Figure 3-1. System Boundary and Security Perimeter. The entities collected inside the system are protected by the security-relevant portions within the security perimeter, as long as rules about access to the system from the outside are enforced by means of external security controls. Rules for access to the security perimeter interface are enforced by the internal controls implemented in the security perimeter.

Identifying the system boundary hinges on precisely specifying the interface between the system and the outside world. External security controls enforce this interface; and as long as those controls are in place, the internal controls protect information within the system against the specified threats. All bets are off, however, if something that should not be there bypasses the external controls and enters the system or if the system is threatened from the outside in an unanticipated way.

For example, a user might walk into the machine room and enter commands on the system console, or the system administrator might divulge a password to an outsider. These are failures of external controls that the system cannot defend against. It may, however, be able to defeat attempted incursions by unauthorized terminals, modems, or users who access the system remotely, as long as they are constrained to enter the system according to the rules of the system interface.

The components inside the system are of two types: those responsible for maintaining the security of the system (those, in other words, that are security-relevant), and all others. The security-relevant components implement the internal controls. Separating the two types of components is an imaginary boundary called the *security perimeter*. The operating system and computer hardware usually lie within the security perimeter; outside the perimeter are user programs, data, terminals, modems, printers, and the items that the system controls and protects. The nature of all components within the security perimeter must be precisely defined, because a malfunction in any one can lead to a security violation; in contrast, the nature of the components outside the perimeter is rather arbitrary, subject only to constraints enforced at the time they enter through the system boundary. A malfunction within the security perimeter has the effect of expanding the security perimeter to the system boundary, causing components previously outside the perimeter to become security-relevant.

Just as a precise interface must be identified across the system boundary, a well-defined interface across the security perimeter is crucial, as well. This interface is enforced by the security-relevant components. For example, the list of system calls in an operating system or the electrical specifications of a communications line are interfaces into the security perimeter. As long as the system boundary is enforced externally, the security perimeter will be maintained by the security-relevant components. In order to implement the components within the security perimeter, great care must go into defining a complete, consistent, and enforceable set of perimeter interface rules.

3.3 USERS AND TRUST

The *user* is the person whose information the system protects and whose access to information the system controls. A person who does not use the system, but who indirectly accesses the system through another user, is not a user as far as the system is concerned. For example, if your secretary is responsible for reading your electronic mail on your behalf, as well as the mail of others in your department, your secretary is the user and, as far as the system is concerned, this same user has access to all the mail in the department. You must trust your secretary, in addition to the system, to keep your mail separate from that of others.

3.3.1 Protecting the User from Self-betrayal

The system must assume that the user who owns a given piece of data or who has created that piece of data, is trusted not to disclose it willfully to another user who should not see it, nor to modify it in an inappropriate way. Of course, the user might be tricked into mishandling his data, but that's a different threat.

Though it may seem obvious, people often lose sight of the fact that computers cannot possibly protect information if the owner of the information wants to give it away.¹ It is in fact possible to design a system that does not allow users to give others access to their data, intentionally or otherwise; but such a design would be silly, because a person determined to disclose information doesn't need a computer to do so. The ability to read a file is tantamount to the ability to give that file to someone else.

While it does not make sense to go to great lengths to prevent a user from giving away information, it does make sense to ensure that the user knows when he or she is doing so. The access controls on the system must have a well-engineered user interface to minimize accidental disclosures.

3.3.2 Identification and Authentication

In order for a system to make meaningful decisions about whether a user should be allowed to access a file, the system (and other users must have a means of identifying each user. A *unique identifier* is a name for each user such as a last name, initials, or account number) that everyone knows, that nobody can forge or change, and that all access requests can be checked against. The identifier must be unique because that is the only way the system can tell users apart. The identifier must be unforgeable so that one user cannot impersonate another.

The act of associating a user (or more accurately, a program running on behalf of a user) with a unique identifier is called *authentication*. The authentication process almost always requires the user to enter a password, but some more advanced techniques, such as fingerprint readers, may soon be available. The process of identification (associating a user ID with a program) is easy to confuse with authentication (associating the real user with the user ID), but it is important to maintain the distinction. The system must separate authentication information (passwords) from identification information (unique IDs) to the maximum extent possible, because passwords are secret and user IDs are public. The password need only be presented when the user first accesses the system. Once the unique ID is determined, the system need not refer to the password again. The unique ID, on the other hand, is used many times to make access decisions. Since the entire security of the system may be based on the secrecy of the passwords, the fewer times and fewer places they are used, the less the risk of exposure will be.

Authentication and identification are general concerns that pertain to systems and programs as well as to users. Users may need to know which system or which programs on the system they are interacting with and they need to obtain this information in a way that cannot be forged by the

1. Various “copy protection” schemes attempt to prevent the user from copying a file (usually on a medium such as a floppy disk) in order to protect copyrighted software, but these schemes address an entirely different threat from the data protection threats that this book is about. (They also don’t work very well.)

system or the programs. Moreover, systems on a network may need to authenticate each other, as if each were a user of the other. In many cases, the ability of a program to impersonate another program—or of a system to impersonate another system—is a serious security concern. The authentication techniques for systems and programs are quite different from those for users. In particular, passwords make very poor authenticators for systems and programs because each use of a password results in disclosure to the recipient and (therefore) the potential for abuse. Section 10.4 describes ways that systems and programs identify themselves to users. Section 13.2.2 discusses system-to-system authentication within a network.

3.4 TRUSTED SYSTEMS

Although users must be trusted to protect data to which they have access, the same is not true for the computer programs that they run. Everybody knows that computer programs are not completely trustworthy. And no matter how much we trust certain users, we cannot let the programs they use have total freedom with the data. The best programmers would agree that even their own programs can make mistakes. It would be nice (but it is usually impractical) to give programs limited access rights on a case-by-case basis, depending on what the programs need.

We can group software into three broad categories of trust:

1. *Trusted* – The software is responsible for enforcing security, and consequently the security of the system depends on its flawless operation.
2. *Benign* – The software is not responsible for enforcing security but uses special privileges or has access to sensitive information, so it must be trusted not to violate the rules intentionally. Flaws in benign software are presumed to be accidental, and such flaws are not likely to affect the security of the system.
3. *Malicious* – The software is of unknown origin. From a security standpoint, it must be treated as malicious and likely to attempt actively to subvert the system.

The quality of software that falls into each of these groups varies greatly from system to system. Most software we use daily is benign, whether the software was written by a good programmer or by an incompetent programmer, and whether that software is a system program or an application. The software is not trusted because it is not responsible for enforcing security of the system, and it is not malicious because the programmer did not intend to deceive the user. Some systems trust software that has received minimal scrutiny, while others consider anything not written by a trusted system programmer to be malicious. Hence, one system's trusted software may be as unreliable as another system's malicious software.

Within a system, a fine line separates a malicious program from a benign program with many bugs: there is no guarantee that a buggy benign program will not give away or destroy data, unintentionally having the same effect as a malicious program. Lacking an objective way to measure the difference, we often (but not always) consider both benign and malicious software to be in a single category that we call *untrusted*. This interpretation is especially common in environments where extremely sensitive information is handled, and it constitutes a fundamental tenet of the security kernel approach to building a secure system.

In most cases, the operating system is trusted and the user programs and applications are not; therefore, the system is designed so that the untrusted software cannot cause harm to the operating system, even if it turns out to be malicious. A few systems are secure even if significant portions of the operating system are not trusted, while others are secure only if all of the operating system and a great deal of software outside the operating system are trusted.

When we speak of trusted software in a secure operating system, we are usually talking about software that first has been developed by trusted individuals according to strict standards and second has been demonstrated to be correct by means of advanced engineering techniques such as formal modeling and verification. Our standards for trust in a secure operating system far exceed the standards applied to most existing operating systems, and they are considerably more costly to implement. Trusting all the software in a large system to this extent is hopeless; hence, the system must be structured in a way that minimizes the amount of software needing trust. The trusted software is only the portion that is security-relevant and lies within the security perimeter, where a malfunction could have an adverse effect on the security of the system. The untrusted software is not security-relevant and lies outside the security perimeter: it may be needed to keep the system running, but it cannot violate system security.

Within a single system, it is normally not useful to distinguish between different degrees of trusted software. Software either is responsible for security or is not. It does no good to assign more trust to some security-relevant programs than to others, because any one of them can do your system in. Similarly, we usually try to avoid establishing degrees of untrustworthiness. In most conventional systems where the security perimeter is not precisely defined, however, it is useful to distinguish between benign and malicious programs. In some instances, certain programs need not work correctly to maintain security of the system, but they nonetheless have the potential to cause damage if they are malicious. Such benign programs fall into a gray area straddling the security perimeter.

3.4.1 Trojan Horses

Most people's model of how malicious programs do their damage involves a user—the penetrator—writing and executing such programs from a remote terminal. Certainly systems do have to protect against this direct threat. But another type of malicious program, called the *Trojan horse*, requires no active user at a terminal.

A Trojan horse is a program or subroutine that masquerades as a friendly program and is used by trusted people to do what they believe is legitimate work. A Trojan horse may be embedded in a wordprocessing program, a compiler, or a game. An effective Trojan horse has no obvious effect on the program's expected output, and its damage may never be detected. A simple Trojan horse in a text editor might discreetly make a copy of all files that the user asks to edit, and store the copies in a location where the penetrator—the person who wrote the program—can later access them. As long as the unsuspecting user can voluntarily and legitimately give away the file, there is no way the system can prevent a Trojan horse from doing so, because the system is unable to tell the difference between a Trojan horse and a legitimate program. A more clever Trojan horse in a text editor need not limit itself to the file the user is trying to edit; any file potentially accessible to the user via the editor is accessible to the Trojan horse.

The reason Trojan horses work is because a program run by a user usually inherits the same unique ID, privileges, and access rights as the user. The Trojan horse therefore does its dirty work without violating any of the security rules of the system—making it one of the most difficult threats to counter. Most systems not specifically designed to counter Trojan horses are able to do so only for limited environments. Chapter 7 presents a detailed discussion of the problem, along with some implications that may seem surprising.

3.5 SUBJECTS, OBJECTS, AND ACCESS CONTROL

All activities within a system can be viewed as sequences of operations on objects. You can usually think of an object as a file, but in general anything that holds data may be an object, including memory, directories, queues, interprocess messages, network packets, input/output (I/O) devices, and physical media.

Active entities that can access or manipulate objects are called *subjects*. At a high level of abstraction, users are subjects; but within the system, a subject is usually considered to be a process, job, or task, operating on behalf of (and as a surrogate for) the user. I/O devices can be treated as either subjects or objects, depending on the observer's point of view, as we will discuss in section 8.5. The concepts of authentication and identification, discussed in section 3.3.2, apply to all types of subjects, although authenticating subjects internal to the computer may be implicit. It is particularly important that all subjects have an unforgeable unique identifier. Subjects operating as surrogates for users inherit the unique ID of the user, but in some cases users may invoke subjects possessing another user's unique ID.

A computer program residing in memory or stored on disk is treated as an object, like any other type of data. But when the program is run, it becomes part of a subject or process. Distinguishing between the program and the process is important because the same program may be run simultaneously by different processes on behalf of different users, where each process possesses a different unique ID. Often we loosely identify a subject as a program rather than as the process in which the program executes, but it should usually be clear when we are talking about a running program as a subject versus a program as data.

Like subjects, objects should have unique IDs. Not all systems implement explicit unique IDs for objects, but doing so is important for a secure system. Section 11.4.2 discusses this topic further.

3.5.1 Access Control

The primary purpose for security mechanisms in a computer system is *access control*, which consists of three tasks:

- Authorization: determining which subjects are entitled to have access to which objects
- Determining the access rights (a combination of access modes such as read, write, execute, delete, and append)
- Enforcing the access rights

In a computer system, the term *access control* applies only to subjects and objects within the system, not to access to the system by outsiders. Techniques for controlling access to the system from outside fall under the topics of user authentication and identification discussed in section 3.3.2. Nonetheless, the access controls in a network of systems must deal with outsiders and remote systems, as well as with subjects inside the system. Network access control is covered in section 13.3.1.

While systems may implement many types of access modes, security concerns usually center on the difference between read and write. In addition, it is occasionally useful to define access modes that distinguish between the ability to delete a file and the ability to write zeros into it (for example) or between the ability to write random data anywhere into a file and the ability to append information to the end of it only.

Subjects grant or rescind access rights to objects. Usually, a subject that possesses the ability to modify the access rights of an object is considered the object's *owner*, although there may be multiple owners. Not all systems explicitly identify an owner; and often subjects other than the owner (such as system administrators) have the ability to grant access.

Associated with each object is a set of security attributes used to help determine authorization and access rights. A security attribute of an object may be something as simple as two bits of information—one for read and one for write—indicating the modes of access that all subjects have to the object. On the other hand, a security attribute may be complex, containing a lengthy access control list of individual subjects and their access rights to the object. Other examples of security attributes of objects are passwords, access bits, and security levels.

Some systems assign security attributes to subjects as well as to objects. These may consist of identifiers or security levels that are used, in addition to the subject's unique ID, as the basis for authorization.

Instead of using subject and object attributes as a basis for access control, some systems use capability lists. A *capability* is a key to a specific object: if a subject possesses the capability, it may access the object. Subjects may possess very long lists of capabilities. A more detailed discussion of capability lists is offered in section 6.2.2.

In talking about how access controls are implemented, we need to distinguish between the granting of access rights (which happens in advance) and the exercising of rights (which happens at the time of access), because security violations do not occur until an improper access takes place. For example, placing confidential information into a public file does not cause any harm until an unauthorized user reads the file. This distinction may seem rather subtle, but the design of some systems forces us to apply certain controls at the time access is granted and certain different controls when the access occurs.

3.5.2 Security Policy

In the real world, a security policy describes how people may access documents or other information. In order for the policy to be reflected in a computer environment, we must rewrite it using

terms such as subjects and objects that are meaningful to the computer. Strictly speaking, the computer obeys security properties, while people obey a security policy. We will, however, loosely talk about the computer's security properties as if they were a policy of the computer system. In cases where the distinction between security policy and security properties is especially important (as when we discuss formal models), we will use more precise terminology.

The computer's version of the policy consists of a precise set of rules for determining authorization as a basis for making access control decisions. Authorization depends on the security attributes of users and information, unique IDs, and perhaps other information about the current state of the system. While all systems have security properties, the properties are not always explicit, and the policy on which they are based may be difficult to deduce. Often the policy is a hodgepodge of ad hoc rules that have evolved over the years and are inconsistently enforced. Lack of a clear policy—and not programming errors—is a major reason why the security controls of many systems are flawed. Section 9.5.1 shows how a security policy is converted into security properties for a system.