

Chapter 4

Design Techniques

This chapter provides an overview of the aspects of computer system design that are important to security. It discusses both the architectures of computer systems and the methods by which systems are designed and built, and it introduces terms that will be used and more thoroughly covered in subsequent chapters.

4.1 SYSTEM STRUCTURES

In the last chapter we introduced two important interfaces: the system boundary and the security perimeter. To understand better the design implications of these interfaces, it is necessary to look closely at how systems are built. We shall group systems into two types: a computer system consisting of a single machine or closely coupled multiprocessors; and a distributed system that resembles a single computer system from the outside but actually consists of multiple computer systems.

The difference between a computer system and a distributed system is reflected in the internal system structure and may not be apparent to users on the outside. Indeed, some people insist that a good distributed system be indistinguishable from a computer system. It is sometimes difficult to decide whether a networked collection of computer systems should or should not be called a distributed system: the decision depends on the observer's point of view, which differs for each application on the network.

Our concern in identifying a distributed system is not so much with terminology as with the internal security architecture of its networking mechanisms. We view a *computer system* as a self-contained entity whose system boundary does not include other systems with which it might be communicating. Such a system must protect itself and does not rely on assistance from other systems; information that leaves the system is no longer protected. We view a *distributed system* as one whose system boundary includes physically separate and relatively autonomous processors that are cooperating in some way to present an integrated environment for at least some applications. Information passing from one processor to another remains within the system and is protected.

The remainder of this section discusses the structure of computer systems, introducing concepts that are fundamental to an understanding of computer security. Chapter 13 covers concepts that pertain specifically to distributed systems and networks.

4.1.1 Structure of a Computer System

The traditional decomposition of a computer system shows hardware, an operating system, and applications programs, as in figure 4-1.¹ There may be multiple applications running simultaneously and independently, one or more for each active user of the system. The applications may be the same or entirely different. The users of the system generally interact only with the applications and not directly with the operating system, though there are important exceptions. Each application running on behalf of a user can loosely be thought of as a *process*.

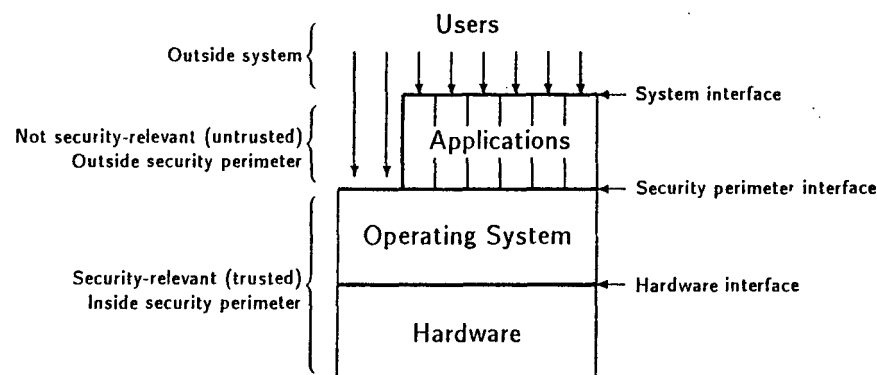


Figure 4-1. Generic Computer System Structure. Each layer uses the facilities of—and is subject to rules and restrictions enforced by—the layer below it. The interface between a pair of layers specifies the functions in the lower layer that are available to the higher layer. The operating system and hardware are security-relevant, lying within the security perimeter. The applications access the operating system through the perimeter by means of a well-defined set of system calls. The users are outside the system. They access the system through the applications or, on occasion, communicate directly with the operating system.

Years ago, the distinction between the hardware and the operating system was obvious: the operating system was implemented with bits in memory that could be easily changed, and the hardware was implemented with circuits that stayed fixed. With many machines containing microcode or firmware, however, the distinction is now blurred. While many agonize over spelling out the differences, the differences matter only when we talk about hardware or software verification—not when we discuss security mechanisms. For the most part, we will treat *hardware* in the conventional sense, as the term is applied to contemporary machines, regardless of whether that hardware is, in fact partially implemented in firmware or software.

The division between the applications and the operating system software is usually more obvious (and more important, from our point of view) than the division between hardware and the

1. Computer security enthusiasts seem to have a preference for drawing pictures that place the operating system underneath the applications, while much of the mainframe world puts the operating system above the applications. This book will adhere to the former, “top-down” tradition.

operating system, although the nature of the software division may vary from system to system. Most people think of an operating system as being distinct from the system applications or processes that are needed to support it. The latter include processes for handling login, backup, and network interfaces. For security purposes, whether or not such processes are implemented outside the operating system is unimportant: the processes still lie within the security perimeter and must be treated as logical parts of the operating system.

A good test to use in deciding whether or not a piece of software should be viewed as part of the operating system is to ask whether it requires any special, privileges to do its job—hardware privileges necessary to execute certain instructions or software privileges needed to gain access to certain data. Although utilities such as compilers, assemblers, and text editors are commonly provided by the vendor of the system (and written by system programmers), such applications do not require any privileges, because unprivileged users with programming skills can write and use their own versions.² Another common test is to check whether the software can have an adverse effect on the system if it misbehaves.

The horizontal lines in figure 4-1 separating the users, applications, operating system, and hardware represent precisely defined interfaces. Usually the security perimeter or operating system interface is described as a set of functions, or *system calls*, offered by the operating system; the hardware interface is described in the machine-language instruction manual. The operating system, together with the hardware, ensures that the security perimeter interface is accessed only in accordance with the rules of that interface. The system interface, on the other hand, is enforced through physical controls external to the system. There are few (if any) controls on the information that passes across that interface. For example, users are allowed to communicate freely with the applications in the system, but they can do so only through permitted physical connections such as terminal ports.

Database management systems, teleprocessing monitors, and other large applications often constitute minioperating systems of their own, running on top of the basic operating system and controlling the execution of several user applications (fig. 4-2). From the perspective of the operating system, the DBMS is just another application or process without special privileges.³ The DBMS may be responsible for enforcing its own security policy, or the operating system may do it all. In such a design, the designers must have a very precise definition of the security requirements in order to tell whether the DBMS is security-relevant. Section 11.3 discusses the security role of such subsystems.

While most of the security features we will be discussing are intended for systems that support multiple users simultaneously, the features are usually applicable to single-user systems such as personal computers that allow serial access by multiple users. Sometimes, however, it is important to distinguish between a PC whose user has physical control over all of the hardware and software

2. This is not universally true, however—especially for machines whose native language is a higher-order language, necessitating use of interpreters to execute the source code.

3. Again, this is an idealized view and is not universally true, since some operating systems do not provide the facilities to support multiuser applications without special privileges.

and a PC whose user does not have direct access to the operating system or hardware. This is because, without some physical security, no PC (or other computer) can protect itself and its data.

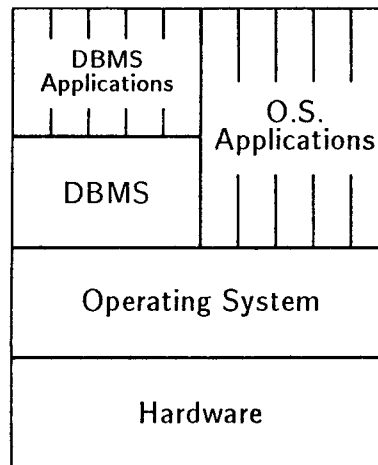


Figure 4-2. DBMS Process Structure. To the operating system, the DBMS appears as just another user application process, while the DBMS controls its own set of applications running as individual processes within the DBMS process.

4.1.2 System States

A system with the structure shown in figure 4-1 or 4-2 requires some built-in support to enforce the layering and proper use of the interfaces. While it may be possible to build a system in which all the layering is enforced by software, the enforcement is tenuous, depending on correct implementation of the software on both sides of each interface. In order for the operating system to enforce constraints on the applications successfully, the operating system must have some help from the hardware.

Most machines have at least two states, domains, or modes of operation: *privileged*, and *unprivileged*. The privileged mode may also be called *executive*, *master*, *system*, *kernel*, or *supervisor* mode; and the unprivileged mode may be called *user*, *application*, or *problem* mode. When the machine is running in privileged mode, software can execute any machine instruction and can access any location in memory. In unprivileged mode, software is prevented from executing certain instructions or accessing memory in a way that could cause damage to the privileged software or other processes. Once the operating system (running in privileged mode) loads certain registers, the machine runs applications software in unprivileged mode until that software makes a call into the operating system, at which time privileged mode is restored. Privileged mode is also entered when interrupts are serviced by the operating system. Without hardware-enforced modes of privilege, the only way the operating system can protect itself is to execute applications programs interpretively—a technique that slows the machine down by several orders of magnitude.

Many modern machines, including microprocessors, have more than two domains. Several members of DEC's PDP-11 family for example, have three protection domains: user, supervisor, and kernel. The kernel mode has the most access to memory and to privileged instructions, and the user mode has the least. Having three hardware domains allows for efficient implementation of the types of system structures shown in figure 4-2. When a machine has more than three domains the domains may be numbered, with the lowest numbered domain having the most privilege. Because the domains are usually hierarchical-in the sense that each domain has more privileges than the domain above it-it is convenient to think of the domains as a series of concentric rings, a concept introduced in Honeywell's Multics (Organick 1972). Multics once proposed as many as sixty-four rings, although in practice systems commonly do not use more than a handful.

4.2 THE REFERENCE MONITOR AND SECURITY KERNELS

The security of a system can be improved in many ways without fundamentally altering its architecture. There are also a number of ways to build a fairly secure system from scratch. But for maximum protection of extremely sensitive information, a rigorous development strategy and specialized system architecture are required. The security kernel approach is a method of building an operating system that avoids the security problems inherent in conventional designs (Ames, Gasser, and Schell 1983). Based on a set of strict principles that guide the design and development process, the security kernel approach can significantly increase the user's level of confidence in the correctness of the system's security controls. Though by no means universally accepted as the ideal solution, the security kernel approach has been used more times than any other single approach for systems requiring the highest levels of security. Following is a very brief overview of the security kernel approach; chapter 10 covers the topic much more thoroughly.

The security kernel approach to building a system is based on the concept of a reference monitor-a combination of hardware and software responsible for enforcing the security policy of the system. Access decisions specified by the policy are based on information in an abstract access control database. The access control database embodies the security state of the system and contains information such as security attributes and access rights. The database is dynamic, changing as subjects and objects are created or deleted, and as their rights are modified. A key requirement of the reference monitor is the control of each and every access from subject to object.

Fundamental to the security kernel approach is the theory that, in a large operating system, a relatively small fraction of the software is responsible for security. By restructuring the operating system so that all of the security-relevant software is segregated into a trusted kernel of an operating system, most of the operating system need not be responsible for enforcing security. The kernel must be suitably protected (tamperproof), and it must not be possible to bypass the kernel's access control checks. The kernel must be as small as possible so that its correctness is easy to verify.

Compare figure 4-1 (showing hardware, software, and an operating system) to figure 4-3. The security kernel in the latter figure consists of hardware and a new layer of software inserted between the hardware and the operating system. The kernel's software and the hardware are trusted and lie inside the security perimeter; in contrast, the operating system lies outside the security perimeter, along with the applications.

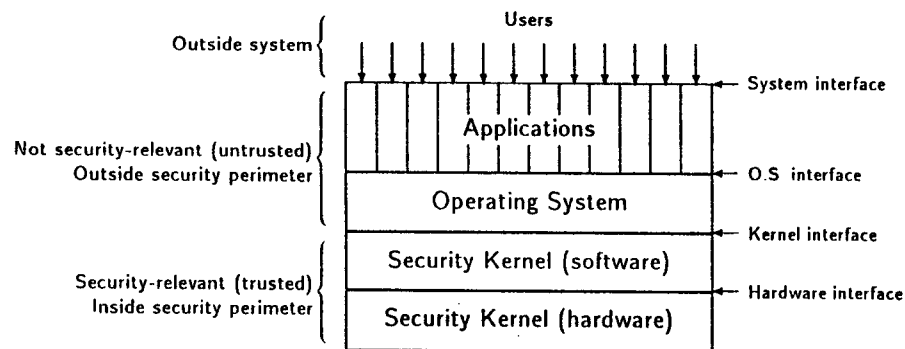


Figure 4-3. Security Kernel in a Computer System. The kernel maintains security by controlling the actions of the operating system, while the operating system maintains a level of service by controlling the actions of the applications.

In most respects, the security kernel is a primitive operating system. The security kernel performs services on behalf of the operating system much as the operating system performs services on behalf of the applications. And just as the operating system places constraints on the applications, the security kernel imposes constraints on the operating system. While the operating system plays no role in enforcing the security policy implemented by the kernel, the operating system is needed to keep the system running and to prevent denial of service due to errant or malicious applications. No error in either the applications or the operating system will lead to a violation of the kernel's security policy.

Building a security kernel does not require building an operating system above it: the security kernel could just as well implement all the functions of an operating system. But the more operating-system features a designer puts in a kernel, the larger the kernel becomes and the more it begins to look like a conventional operating system. In order for us to have any confidence that the kernel is more secure than an operating system, the kernel must be as small as possible. The smallness requirement must be ruthlessly enforced during design: the kernel should not contain any function not necessary to prevent a violation of the security, policy. Issues such as performance, features, and convenience lie below smallness on the list of kernel design priorities.

4.3 SYSTEM DEVELOPMENT PROCESS

The development of any system involves several steps:

- *Requirements*: establishing generic needs
- *Specification*: defining precisely what the system is supposed to do, including *specification verification*, which involves demonstrating that the specification meets the requirements
- *Implementation*: designing and building the system, including *implementation verification*, which involves demonstrating that the implementation meets the specification

We also use the word *correspondence* as another name for the verification substeps at which two descriptions of a system are shown to be in agreement. Usually the more detailed description (for example, the specification) at a low level is said to correspond to the less detailed description (for example, the requirements) at a higher level.

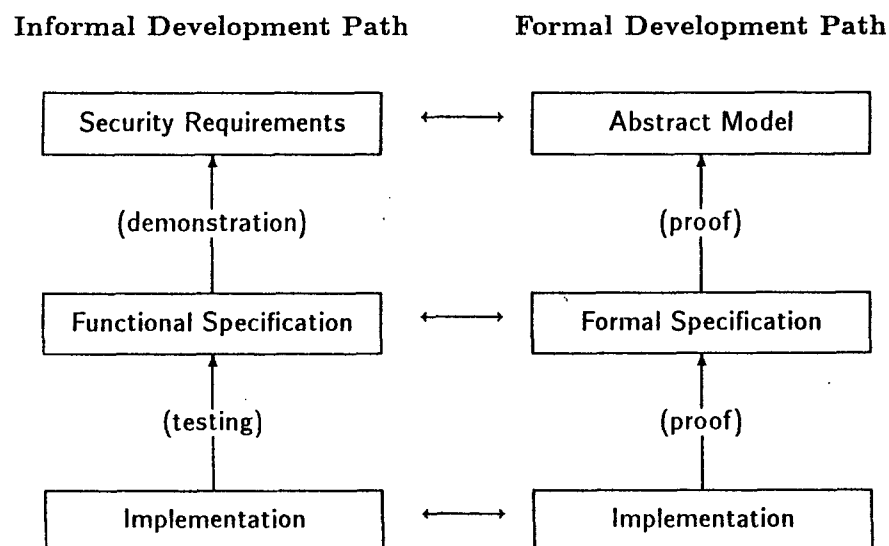


Figure 4-4. System Development Process for a Secure System. The security-relevant aspects of the system development process are shown in two parallel paths. The informal path is conventional; the functional specifications and implementation are shown to meet the security requirements through correspondence steps involving demonstration and testing. The formal path, using mathematical techniques, is employed for systems where an extremely high level of assurance regarding the security controls is desired.

The overall development of a system is guided by a *system architecture*. While most of us think of a system architecture as a description of the system as built, rather than thinking of it as a description of the process by which the system is built, a relationship exists between the result you want to achieve and the way you get there. Many of the desired characteristics of the system that

help dictate the architecture (such as reliability, maintainability, and performance) have a profound impact on the development strategy. The system architecture makes itself felt either directly (through explicit development guidelines) or indirectly (through system goals).

A *security architecture* describes how the system is put together to satisfy the security requirements. If the security requirements specify that the system must attain a given level of *assurance* as to the correctness of its security controls, the security architecture must dictate many details of the development process.

Security concerns do not add steps to the development process that are not part of conventional developments. Rather, the guidelines in the security architecture are a pervasive influence on all development steps. The left-hand (“informal”) side of figure 4-4 illustrates the conventional development process with one change: we have replaced *system requirements* at the top with the *security requirements*. The security requirements, a small extract of the total system requirements, are derived from the system’s security policy (not shown in the figure). The functional specification and the implementation shown in the figure are complete, not security-specific extracts. Verification of the functional specification against the security requirements—a process we call *demonstration* because it is based on informal arguments—is a far simpler task than verification of the specification against all functional requirements, since many functions described in the specification have little effect on security. Clearly verification is made easier if the functional specification is structured to locate security-relevant functions in as few (and as isolated) places as possible.

The bottom two phases in the informal path of figure 4-4, the implementation and its verification (testing), are conventional; there are no shortcuts to fully verifying the implementation against its specification, even if security is the only concern, because all functions must be examined or tested to be sure that they do not violate the security constraints.

The development process we have just discussed is called *informal* because there is no proof, in the mathematical sense, that the steps are correctly carried out. Because requirements and specification are written in a natural language (English, French, Latin) that is prone to ambiguities and omissions, formal mathematics cannot be applied to any of the correspondence steps.

The right-hand side of figure 4-4 shows a parallel formal development path that might be used to develop a highly secure system such as a security kernel. Each phase in the informal path has a formal equivalent. The implementation, consisting of computer programs and hardware, is unchanged because the programs and hardware are already formal.

The natural-language security requirements are expressed as an abstract model, written in a mathematical notation, that is derived from exactly the same security policy as are the security requirements. The natural-language specification is expressed in a formal specification language amenable to computer processing. The correspondence steps of demonstration and testing are replaced by mathematical proofs. The horizontal arrows between parallel phases in the figure indicate equivalence, although no objective proof of that equivalence can be made.

The arrows between layers in the figure are upward, indicating that, in each case, the lower-layer description of the system corresponds to, satisfies, or is an example of a system described in the higher layer. In the formal path, especially, all of the rules in the abstract model need not be expanded in the formal specification, and all of the functions in the formal specification need not exist in the implementation; it is only necessary that the lower layer avoid violating rules or requirements of its adjacent higher layer.

The formal path for development is intended as a supplement to, not a replacement for, the informal path. It augments the informal process enough to provide the appropriate level of assurance dictated in the security requirements. Which phases of the formal path you carry out and how thoroughly you do so vary with that degree of assurance.

When you are choosing a development strategy, it is most important that you avoid gaps in the correspondence process: the boxes in the figure must be connected. Although you can choose not to write a formal specification, it is a waste of time to develop either an abstract model or a formal specification without devoting proper effort to the correspondence process. For example, you may choose to use an abstract model as an adjunct to the security requirements, without a formal specification; but in that case, you must demonstrate informally that the functional specification corresponds to the model. Alternatively, you may want to develop both a model and a formal specification but omit the formal proofs; if so, you must then use informal arguments to demonstrate correspondence among the implementation, the formal specification, and the model.

Be warned that figure 4-4 is a bit deceiving: in practice, you cannot hope to prove fully that the implementation meets the formal specification. Such a proof is a theoretical possibility, but it is not yet feasible. In other words, today's technology does not permit the right-hand path to be entirely formal. Nonetheless, the existence of a formal specification allows you to make a much more convincing, semiformal argument for implementation correspondence than you could get by testing alone.

REFERENCES

- Ames, S. R., Jr.; Gasser, M.; and Schell, R. R. 1983. "Security Kernel Design and Implementation: An Introduction." *Computer* 16(7):14-22. Reprinted in *Advances in Computer System Security*, vol. 2, ed. R. Turn, pp. 170-77. Dedham, Mass.: Artech House (1984).
An overview of the reference monitor concept, the security model, and kernel implementation issues.
- Organick, E. I. 1972. *The Multics System: An Examination of Its Structure*. Cambridge, Mass.: MIT Press.
A description of Multics—at that time implemented on a processor without hardware-supported protection rings.