

Chapter 5

Principles of a Security Architecture

Building a computer system requires striking a balance among a number of requirements such as capability, flexibility, performance, ease of use, and cost. While there is nothing inherently conflicting about these requirements, features intended to satisfy them often work against each other and require you to make tradeoffs in the system design. Security is simply another requirement; and where they conflict, security features must likewise be traded off against other features, based on the importance of security to the system.

As a purist whose primary goal is to make your system secure, you might not be willing to give up a single security feature in favor of any other. But with such an outlook you are likely to fail: others will treat you as a security fanatic who is ignorant of what it means to build a practical system. By being adamant about security to the detriment of other features, you will lose most arguments over system design alternatives, and the system you are trying to influence will probably end up with few meaningful security capabilities. You are more likely to succeed in your goal of establishing a secure system if you remain pragmatic, keeping the primary goals of the system in mind and compromising on nonessential points at appropriate times. Even if you are building a security kernel for which everyone agrees that security is the most important goal, performance is almost always very close behind.

If you approach the design of a system with the attitude that you are willing to give in when necessary, your strategy should be to steer the design of the system in a direction that will avoid conflicts where possible. Many security features need not adversely affect other features. You can achieve most of your security goals without conflicts if you establish ground rules or principles to guide the system design. Once everyone on the design team agrees to these principles, the design will naturally follow a secure path.

The key to the control of the design process is the security architecture—a detailed description of all aspects of the system that relate to security, along with a set of principles to guide the design. The security architecture is not a description of the functions of the system; such detail belongs in a functional specification. A good security architecture is more like a design overview, describing at an abstract level the relationships between key elements of the system architecture in a way that satisfies the security requirements. The security architecture should also describe the aspects of the system development process (see section 4.3) through which adherence to the security requirements is assured. The architecture should not constrain the design in ways that do not affect security.

In the early conceptual stage of system development—even before requirements have been completely defined—a security architecture can be written that deals with high-level security issues: the system security policy, the degree of assurance desired, the impact of security on the development process, and overall guiding principles. A security architecture written at this early stage is generic, with few details about the specific system to be designed.

When the system architecture is later solidified, the security architecture should be enhanced to reflect the structure of the system. As the design progresses through stages of more and more detail, the security architecture becomes increasingly specific. While the security architecture must evolve in parallel to the system development effort, the architecture must keep ahead of that effort so that it can help guide the work to be done.

Of course, writing down a security architecture does no good unless people stick to it. The security architecture must play a dominant role in the development process, and all the developers must subscribe to it. Even during the implementation phase of a project, individual programmers will be affected by guidelines distilled from the architecture, through programming standards, code reviews, and testing.

5.1 CONSIDER SECURITY FROM THE START

Except in research projects, few systems are designed with security as the primary goal from the start. All too often the approach of the developers is “build it first, secure it later.” From such a beginning, security is unlikely to be well-integrated into the system. Most designers fail to appreciate the great cost of retrofitting security.

You do not have to make security your number one goal in order to develop a secure system, but you do have to think about security from the beginning. Usually several ways are available to structure a system to satisfy a given set of requirements—some good for security, and some not. Without a security architecture to guide the early decisions, it is easy to choose a fundamentally flawed option, after which the cost of adding security controls is many times greater than would have been necessary had an equally sound alternative been selected.

This book contains many examples of situations where adding security to an existing system is made difficult by unfortunate design decisions. Practical experience in developing large systems has shown that, unless security considerations have influenced the early stages of design, little meaningful security is achieved in the final system. It cannot be stressed too strongly that, if you have any intention to incorporate security into a system, regardless of the priority of that security requirement, you must begin to think about it on the first day.

5.2 ANTICIPATE FUTURE SECURITY REQUIREMENTS

The security architecture should attempt to be far-sighted, addressing potential security features even if there is no immediate plan to use them. Usually it costs little to allow for future security enhancements, and therefore little is lost if the anticipated security is never needed.

But you must not be overly specific about anticipating security enhancements. For example, you might allow for an additional field in a protocol header to handle a security label; but when the time comes to implement the security feature, you may find it necessary to implement a third-party connection authorization scheme in order to validate the label—a feature that requires a different protocol design and can affect all existing implementations of that protocol. Another example involves error handling: while you may have made it easy to add security checks in many

places in the system, such checks may introduce the possibility of new combinations of failures that existing software does not expect and cannot gracefully handle. A classic example of a new kind of failure is the inability of software to read a file even though it recognizes that the file exists.

The keys to incorporating the appropriate hooks for future security enhancements are to understand computer security requirements in general and to include those requirements explicitly as possible future needs in a security architecture. Sufficient detail in the handling of future security needs must be worked out, and such detail must be part of the design.

Anticipating security requirements not only affects the level of effort needed to make the system more secure in the future, it may also determine whether security in the system can ever be improved. Experience has shown that the security of many systems cannot be improved because the functions of the system have been defined in such a way as to depend on fundamentally insecure characteristics of the system. If the characteristics are changed, the system will no longer work as expected. In many cases, plugging security holes fixes the operating system but breaks the applications.

Consider a system that provides a scratchpad directory for use by applications programs that need to create temporary files, as is done by some versions of Unix. The directory will contain files belonging to many users on whose behalf the applications are running. But placing many users' files in a single directory readable by all users might not be secure. Even the most rudimentary security enhancements require a separate directory per user, and yet making such a separation in a clean way could be a vast undertaking if it involves modifying all applications that use temporary files.¹

An area that demands particularly careful design planning is the definition of the security policy. A change in the security policy can have a catastrophic effect on previously good applications that violate the new policy, even when the change made in the operating system to implement the policy is simple. Had the applications been built with the new policy in mind (even if it were not enforced by the system at the time), the change would have been transparent. Of course, a documented but unenforced policy can easily be violated; you must exercise strong discipline over the applications developers. Among the applications that tend to be affected by a change in security policy are those that manage distributed information, those that maintain databases accessible to more than one user, and those that implement communications between users. Classic examples include electronic mail and database management systems. The most serious problems for such applications are caused by mandatory access control policies (see section 6.3).

5.3 MINIMIZE AND ISOLATE SECURITY CONTROLS

To achieve a high degree of confidence in the security of a system, the designer should minimize the size and complexity of the security-relevant parts of the internal design. A major reason why operating systems are not secure is that their large size leads to overall incomprehensibility. Of

1. Solutions to this particular problem in Unix have been proposed that do not require modifying all applications, but the solutions are not clean.

course, size is also the reason why operating systems are never totally free of bugs, and so will always be liable to behave unpredictably. But given that complex functional needs outside your control require you to have a big system, you still have the freedom to structure the system so that at least some parts (those that have to do with security) are small and well-defined.

If you are enhancing a system to add new security features, you may still follow this minimization principle, but constraints imposed by the existing architecture will certainly limit your flexibility. Needless to say, if improving the security of a system requires as much new mechanism as the system had in the first place, reliability of the new mechanism will be no higher than that of the original system (unless there are also significant improvements in the software engineering techniques used in those enhancements). You can always add new and useful security features, but the level of assurance may not change.

The key to minimizing the security-relevant parts of an operating system is to design the system to use only a small number of different types of security enforcement mechanisms, thereby forcing security-relevant actions to be taken in a few isolated sections. This goal, sometimes called economy of mechanism (Saltzer and Schroeder 1975), is simply a matter of good software engineering, but it is hard to attain for security in an operating system. The reason for this difficulty is that security permeates many different functional areas of a system- file system handling, memory management, process control, input/ output, and a large number of administrative functions-so that you do not have a security module in a system as you do a device driver or scheduler.

An example illustrating the proliferation of redundant security mechanisms typical in older systems that have evolved over the years is the control of file access. One set of applications may manage its files by requesting a password before opening a file, another may use an access control list for each file, and another may use a set of access rights assigned in advance to each user (see section 6.2). Granularity of access control may also differ between applications: a DBMS worries about access on the record level; a message-handling system worries about access at the message level; and a document-processing system worries about whole files. Access control software will therefore be sprinkled throughout these applications. Not only will the security-relevant software be difficult to find and isolate, each application will have its own definition of security. Even if all the security software can be isolated in some way, the variety of mechanisms makes it difficult to design a common security solution. There is little hope for substantially improving the security in such an environment without thoroughly reexamining large parts of the system.

All operating systems-even old and complex ones-have some security mechanisms that control access to basic objects such as files, but often the common mechanism is too inflexible to be useful for many applications. The message-handling system designed to control access to individual messages within a file must bypass the operating system's access controls at the file level (by giving default read/write access to files for its users) and will provide its own security controls. If two such applications exist on the same system-for example, a data-base management system and a message-handling system-there will probably be two different approaches. Even systems such as Honeywell's Multics, whose design is based on the economy-of-mechanism principle, are forced to implement duplicative security controls to handle messages (Whitmore et al. 1973).

There are other reasons why we find multiple security mechanisms that do almost the same thing. When new and more flexible mechanisms are introduced in an older system, they are often incompatible with existing mechanisms; nonetheless, the older mechanism must be retained for compatibility. Some newer sophisticated mechanisms needed for certain applications are too inefficient for general use, so they are implemented as optional features. (An optional feature is not going to receive widespread use: when users are allowed to choose among several mechanisms, the decision is more likely to be based on the sophistication of the user than on the dictates of security.)

If the security-relevant mechanisms in the system are simple, easily identified, and isolated, it is usually possible to implement additional controls to protect them from damage by bugs in other portions of the system. Certainly the code that makes security decisions should be write-protected so that it cannot be modified. The databases used to make the decisions should be isolated and, if possible, protected against modification by other parts of the system.

Isolation of data should not be carried to an extreme, however. Security attributes of files, for example, are best stored along with other attributes of files, rather than in a separate database, because the synchronization mechanism needed to maintain the separate database may be complex and prone to error. In the ultimate effort to isolate security controls that is made in the security kernel approach, extreme care is devoted to separating the security-relevant mechanisms into a hardware-protected kernel of an operating system. Security kernel designers go to great lengths to minimize the size of the kernel, even if it vitiates performance or requires a significantly more complicated operating system outside the kernel.

5.4 ENFORCE LEAST PRIVILEGE

Closely related to the concept of isolating the security mechanisms is the principle of least privilege: subjects should be given no more privilege than is necessary to enable them to do their jobs. In that way, the damage caused by erroneous or malicious software is limited. A strictly enforced least-privilege mechanism is necessary if any reasonable degree of assurance in the security of a system is to be attained.

The philosophy of least privilege has several dimensions. The usual meaning of privilege in a computer system relates to the hardware mechanism that restricts use of special instructions (such as input/output instructions) and access to certain areas of memory when the processor is not operating in a privileged mode or domain. A system with only two domains (privileged and unprivileged) has a difficult time enforcing least privilege except at the coarsest level: the privileges accorded are either all or none. An architecture with three or more states provides finer degrees of control, where each state has access to less memory than the previous state as you “move out” from the most privileged state. But the hierarchical nature of the domains does not always match the requirements of applications. A capability architecture with non hierarchical domains allows the finest degree of control but requires complex hardware support. Section 8.4 covers hardware protection features that support least privilege.

Similar to the hardware privileges are the software privileges assigned to certain programs by the operating system. These privileges permit programs to bypass the normal access controls

enforced on user programs, or to invoke selected system functions. There may be a number of such privileges, providing a fine granularity of control over what a program can and cannot do. For example, the system backup program may be allowed to bypass read restrictions on files, but it need not have the ability to modify files. The restore program might be allowed to write files but not to read them.

While a system with many types of software privileges allows a fine degree of control over least privilege, privileges should not be used as a catch-all to make up for deficient and inflexible access controls. It is usually possible to design the normal access controls to accommodate most system functions without privileges. For example, Multics does not require the backup process to bypass any controls; the backup process is treated just as any other process is that is explicitly given read access to files to be backed up. Users can revoke the backup's read access to a file if they choose, and thereafter the file will not be backed up. A system that relies on a bewildering variety of privileges to carry out routine system functions securely probably has poorly designed access controls.

Another dimension of least privilege is enforced by the way in which the system is built through techniques such as modular programming and structured design. By establishing programming standards that restrict access by procedures to global data, for example, a system designer can minimize the possibility that an error in one area will affect another area. Such conventions amount to no more than good programming practice; but where security is concerned, the motivation for strict adherence to these standards must be greater. In particular, use of a layered architecture (discussed in section 11.1) can go a long way toward increasing the reliability of a secure operating system.

A final dimension for least privilege involves actions of users and system administrators. Users and system managers should not be given more access than they need in order to do their jobs. Ensuring least privilege for a user means deciding exactly what the user's job is and configuring the system to constrain the user to his or her duties. The system must provide the necessary support through flexible, fine-grained access controls and through its overall architecture.

The area of administrative functions is a particularly good place to enforce least privilege. Many systems (Unix being the most notorious) have a single "superuser" privilege that controls all system administrative functions, whether they are security-relevant or not.

Examples of administrative functions that are not security-relevant include mounting tapes, taking dumps, starting and stopping printer queues, bringing up and bringing down various background system processes, reconfiguring hardware, and entering certain user attributes. The functions are privileged because a malicious user of them could wreak havoc on the system, but misuse is unlikely to compromise security.

Administrative security functions include assigning and resetting passwords, registering new users, specifying security attributes of users and files, and breaking into someone's account in emergency situations. Misuse of these functions (or even slight slips at the keyboard) could cause lasting security problems.

By isolating day-to-day administrative functions from security administrative functions, and by using separate privileges for the different types of functions, a system would provide the capability for a site to give more people access to the administrative functions without risk of compromising security. Only a very determined malicious user of administrator privilege would be able to affect the security functions. In a system based on a security kernel, we often go so far as to make it impossible for a person with administrator privilege to affect security.

5.5 STRUCTURE THE SECURITY-RELEVANT FUNCTIONS

In discussing the system development process in section 4.3, we observed the need to demonstrate that the functional specification of the system satisfies its security requirements. If such a demonstration entails careful scrutiny of hundreds of functions in a large system, the demonstration is not only difficult but of dubious value. It is essential that the architecture of the system permit the security-relevant aspects of the system to be easily identified so that large sections of the system can be examined quickly. With a good security architecture, this simply requires good documentation: the security controls will be isolated and minimized, and there should be a clean and easily specifiable interface to the security-relevant functions.

If we look at a description of system calls in an operating system, we usually find that many, if not most, functions have to make some security-relevant decisions. It is not possible to isolate all security-relevant activities in one place. Any function used to access an object has to determine access rights, and many functions must check their arguments for validity. Consequently, it is necessary to identify clearly which checks are security-relevant and which are courtesy checks for the programmer. In particular, many important checks to prevent denial of service are not relevant if the security requirements do not address denial of service.

5.6 MAKE SECURITY FRIENDLY

The following three principles should be kept in mind in any effort to design security mechanisms:

- Security should not affect users who obey the rules.
- It should be easy for users to give access.
- It should be easy for users to restrict access.

The first principle means that, in the average case of a user doing an assigned job, security should be transparent. When security repeatedly gets in the way, users lose productivity and may seek a way to bypass the controls. The security controls must be flexible enough to accommodate a wide range of user activities while fully enforcing the principle of least privilege.

No system can anticipate all possible user activities, so a user will occasionally need to understand and use the security mechanisms. In some systems the seemingly simple act of giving or restricting access to a file requires a system administrator action. Such systems view security as a concern only to system managers. Under such burdensome procedures, system administrators are

likely to give out more access than is needed, and only the most highly motivated users are likely to take the action needed to protect their information.

The second principle ensures that the user will provide access to information only when required and will not set up excessively permissive defaults to avoid complex procedures. The third principle increases the likelihood that the user will protect information when necessary.

A fourth design principle can be identified to help satisfy the preceding three principles:

- Establish reasonable defaults.

This includes both system-defined defaults and a mechanism for user-definable defaults.

A security administrator could argue that users of highly secure systems should constantly be made aware of their security responsibilities; otherwise, they might forget to take action to protect especially sensitive information when necessary. An obvious way to keep users on their toes is to configure the system so that the default action taken by the system is very restrictive (for example, arranging that nobody but the creator of a file can get access), while building in the option for a user to overrule the default when necessary. But if users are burdened with the need to override the default repeatedly in order to do their job, they will find a way to do so automatically; and an automatic override operates whether it is needed or not.

Some people insist that overriding the default controls should be difficult, requiring extraordinary effort. The government, in handling classified information, wants to make it extremely difficult for a user with access to the information to expose that information voluntarily within the computer, despite the fact that the user is fully trusted not to expose the information outside the computer. The rationale is that information in a computer is more subject to careless exposure than information on a sheet of paper. But as has been noted, making disclosure of one's own information extremely difficult only deters those who are not determined to make such a disclosure.

An improperly implemented user-defined default can be dangerous. In some versions of Unix, for example, the user can specify a default set of access modes to be assigned to all newly created files during a session. But the mechanism does not model the way people work: sometimes users operate on private files and sometimes they operate on public files, alternating between one and the other in the same session. A user who specifies a session default to make files publicly accessible is probably going to forget to turn off the default when creating private files.

A better design, used in Multics and eventually added to DEC's vms, allows the user to specify default access modes for files based on the directory in which the file is located. Users are inclined to use different directories, rather than different sessions, for different aspects of their job. People can easily adapt their work habits to such a mechanism and are more likely to give access only where necessary.

These arguments demonstrate that making security friendly requires a thorough understanding of the applications for which the system will be used. This is much easier to do in some systems than in others. Designers of general-purpose systems have a difficult time deciding what the users

will do, and in an attempt to please everyone they are likely to offer multiple redundant mechanisms, thereby violating the economy-of-mechanism principle.

5.7 DO NOT DEPEND ON SECRECY FOR SECURITY

Except in the handling of passwords and encryption keys, a primary goal for the security architecture of a system is to avoid depending on the secrecy of any part of the system's security mechanisms. In other words, it is unsafe to assume that users will not be able to break into a system because they do not have the manuals or source listings of the software. Of course, a penetration is certainly harder without the information, but you never know what information the penetrator has obtained, and the safest assumption is that the penetrator knows everything.

Fortunately secrecy of design is not a requirement for even the most highly secure systems. If you are building a system from the ground up, you have the opportunity to incorporate the necessary mechanisms so that even a person who helped develop the system cannot break into it. But if you are enhancing an existing system, you do not have that freedom, and you may have to make a guess as to how clever the penetrator will be. In such a case, you might well avoid publicly describing exactly what security enhancements you have made.

Revealing the internals of a system does not mean revealing ways to penetrate the system. Even the most secure systems have flaws, detected either as part of a penetration analysis or as a result of an actual penetration. No system will ever be free of all covert channels (see section 7.2).

Disclosing the design of a system's security mechanisms can actually improve security because it subjects the system to scrutiny by a much larger audience. Vendors often find that their customers report security problems in their systems as bugs before any serious penetration takes place. Of course, proprietary designs must be appropriately protected, but such protection should not be a requirement for system security.

REFERENCES

- Saltzer, J. H., and Schroeder, M. D. 1975. "The Protection of Information in Computer Systems." *Proceedings of the IEEE* 63(9):1278-1308. Reprinted in *Advances in Computer System Security*, vol. 1, ed. R. Turn, pp. 105-35. Dedham, Mass.: Artech House (1981).
A set of principles for the design of protection features in computers- particularly those used in Multics.
- Whitmore, J.; Bensoussan, A.; Green, P.; Hunt, D.; Kobziar, A.; and Stem, J. 1973. "Design for Multics Security Enhancements." ESD-TR-74-176. Hanscom AFB, Mass.: Air Force Electronic Systems Division. (Also available through National Technical Information Service, Springfield, Va., NTIS AD-A030801.)
A description of the enhancements incorporated into Multics to support mandatory security controls.