Chapter 14
Design Principles

FALSTAFF: If I had a thousand sons, the
first human principle I would teach them should
be, to forswear thin potations and to addict
themselves to sack.
— *The Second Part of King Henry the Fourth*, IV, iii, 133–136.

Specific design principles underlie the design and implementation of mechanisms for supporting security policies. These principles build on the ideas of simplicity and restriction. This chapter discusses those basic ideas and design principles.

14.1 Underlying Ideas

Saltzer and Schroeder [1644] describe eight principles for the design and implementation of security mechanisms; Saltzer and Kaashoek [1643] later refined them. The principles draw on the ideas of simplicity and restriction.

Simplicity makes designs and mechanisms easy to understand. More importantly, simple designs lead to fewer problems, and those that occur are usually easier to deal with. Minimizing the interaction of system components minimizes the number of sanity checks on data being transmitted from one component to another.

EXAMPLE: The program *sendmail* reads configuration data from a binary file. System administrators generated the binary file by "freezing," or compiling, a text version of the configuration file. This created three interfaces: the mechanism used to edit the text file, the mechanism used to freeze the file, and the mechanism *sendmail* used to read the frozen file. The second interface required manual intervention and was often overlooked. To minimize this problem, *sendmail* checked that the frozen file was newer than the text file. If not, it warned the user to update the frozen configuration file.

The security problem lies in the assumptions that *sendmail* made. For example, the compiler would check that a particular option had an integer value. However, *sendmail* would not recheck this; it assumed that the compiler had done the checking. Errors in the compiler checks, or *sendmail*'s assumptions being inconsistent with those of the compiler, could produce security problems. If the compiler allowed the default UID to be a user name (say, daemon with a UID of 1), but *sendmail* assumed that it was an integer UID, then *sendmail* would scan the string "daemon" as though it were an integer. Most input routines would recognize that this string is not an integer and would default the return value to 0. Thus, *sendmail* would deliver mail with the root UID rather than with the desired daemon UID.

Simplicity also reduces the potential for inconsistencies within a policy or set of policies.

EXAMPLE: A college rule requires any teaching assistant who becomes aware of cheating to report it. A different rule ensures the privacy of student files. A TA contacts a student, pointing out that some files for a program were not submitted. The student tells the TA that the files are in the student's directory, and asks the TA to get the files. The TA does so, and while looking for the files notices two sets, one with names beginning with "x" and the other set not. Unsure of which set to use, the TA takes the first set. The comments show that they were written by a second student. The TA gets the second set, and the comments show that they were written by the first student. On comparing the two sets, the TA notes that they are identical except for the names in the comments. Although concerned about a possible countercharge for violation of privacy, the TA reports the student for cheating. As expected, the student charges the TA with violating his privacy by reading the first set of files. The rules conflict. Which charge or charges should be sustained?

Restriction minimizes the power of an entity. The entity can access only information it needs.

EXAMPLE: Government officials are denied access to information for which they have no need (the "need to know" policy). They cannot communicate that which they do not know.

Entities can communicate with other entities only when necessary, and in as few (and narrow) ways as possible.

EXAMPLE: All communications with prisoners are monitored. Prisoners can communicate with people on a list (given to the prison warden) through personal visits or mail, both of which are monitored to prevent the prisoners from receiving contraband such as files for cutting through prison bars or weapons to help them break out. The only exception to the monitoring policy is when prisoners meet with their attorneys. Such communications are privileged and so cannot be monitored.

"Communication" is used in its widest possible sense, including that of imparting information by not communicating.

EXAMPLE: Bernstein and Woodward, the reporters who broke the Watergate scandal, describe an attempt to receive information from a source without the source's directly answering the question. They suggested a scheme in which the source would hang up if the information was inaccurate and remain on the line if the information was accurate. The source remained on the line, confirming the information [178].

14.2 Principles of Secure Design

The principles of secure design discussed in this section express common-sense applications of simplicity and restriction in terms of computing. We will discuss detailed applications of these principles throughout the remainder of Part V and in Part VIII, "Practicum." However, this chapter mentions specific examples.

14.2.1 Principle of Least Privilege

This principle restricts how privileges are granted.

> **Definition 14–1.** The *principle of least privilege* states that a subject should be given only those privileges that it needs in order to complete its task.

If a subject does not need an access right, the subject should not have that right. Furthermore, the *function* of the subject (as opposed to its identity) should control the assignment of rights. If a specific action requires that a subject's access rights be augmented, those extra rights should be relinquished *immediately* on completion of the action. This is the analogue of the "need to know" rule: if the subject does not need access to an object to perform its task, it should not have the right to access that object. More precisely, if a subject needs to append to an object, but not to alter the information already contained in the object, it should be given append rights and not write rights.

In practice, most systems do not have the granularity of privileges and permissions required to apply this principle precisely. The designers of security mechanisms then apply this principle as best they can. In such systems, the consequences of security problems are often more severe than the consequences for systems that adhere to this principle.

EXAMPLE: The UNIX operating system does not apply access controls to the user *root*. That user can terminate any process and read, write, or delete any file. Thus, users who create backups can also delete files. The administrator account on Windows has the same powers.

This principle requires that processes should be confined to as small a protection domain as possible.

EXAMPLE: A mail server accepts mail from the Internet and copies the messages into a spool directory; a local server will complete delivery. The mail server needs the rights to access the appropriate network port, to create files in the spool directory, and to alter those files (so it can copy the message into the file, rewrite the delivery address if needed, and add the appropriate "Received" lines). It should surrender the right to access the file as soon as it has finished writing the file into the spool directory, because it does not need to access that file again. The server should not be able to access any user's files, or any files other than its own configuration files.

14.2.1.1 Principle of Least Authority

Closely related to the principle of least privilege is the principle of least authority [1349]. The two are often treated as meaning the same. However, some authors make a distinction between "permission" and "authority." They treat permission as determining what actions a process can take on objects directly, and authority as determining that effects a process may have on an object, either directly (as with permission) or indirectly through its interactions with other processes or subsystems.

Miller and Shapiro [1349] give a good example from the Take-Grant Protection Model. In that model, the rights would represent actions that subjects could take over objects, and so represent permissions. But the *de facto* rules of that model, which govern information transfer, show how information can flow from a subject to an object that is not directly connected to the subject. Hence the subject does not have permission to write information into the object, but it does have permission to pass the information to a second subject, and that subject can write the information into the object.[1]

Definition 14–2. The *principle of least authority* states that a subject should be given only the authority that it needs in order to complete its task.

If one reads the principle of least privilege as speaking to *permissions*, then this principle is somewhat different. But if it speaks to *authority*, the two are the same.

14.2.2 Principle of Fail-Safe Defaults

This principle restricts how privileges are initialized when a subject or object is created.

---

[1]This is the *find* rule described by Bishop and Snyder [213, 233].

Definition 14–3. The *principle of fail-safe defaults* states that, unless a subject is given explicit access to an object, it should be denied access to that object.

This principle requires that the default access to an object is *none*. Whenever access, privileges, or some security-related attribute is not *explicitly* granted, it should be denied. Moreover, if the subject is unable to complete its action or task, it should undo those changes it made to the security state of the system before it terminates. This way, even if the program fails, the system is still safe.

EXAMPLE: If the mail server is unable to create a file in the spool directory, it should close the network connection, issue an error message, and stop. It should not try to store the message elsewhere or to expand its privileges to save the message in another location, because an attacker could use that ability to overwrite other files or fill up other disks (a denial of service attack). The protections on the mail spool directory itself should allow create and write access only to the mail server and read and delete access only to the local server. No other user should have access to the directory.

In practice, most systems will allow an administrator access to the mail spool directory. By the principle of least privilege, that administrator should be able to access only the subjects and objects involved in mail queueing and delivery. As we have seen, this constraint minimizes the threats if that administrator's account is compromised. The mail system can be damaged or destroyed, but nothing else can be.

Because many users do not change default access control permissions, this rule applies to the default settings for both the system and for users.

14.2.3 Principle of Economy of Mechanism

This principle simplifies the design and implementation of security mechanisms.

Definition 14–4. The *principle of economy of mechanism* states that security mechanisms should be as simple as possible.

If a design and an implementation are simple, fewer possibilities exist for errors. The checking and testing process is less complex, because fewer components and cases need to be tested. Complex mechanisms often make assumptions about the system and environment in which they run. If these assumptions are incorrect, security problems may result.

EXAMPLE: The *ident* protocol [1807] sends the user name associated with a process that has a TCP connection to a remote host. A mechanism on host *nob* that allows access based on the results of an *ident* protocol result makes the assumption that the originating host is trustworthy. If host *toadflax* decides to attack host *nob*, it can connect and then send any identity it chooses in response to the *ident* request. This is an example of a mechanism making an incorrect assumption about the environment (specifically, that host *toadflax* can be trusted).

Interfaces to other modules are particularly suspect, because modules often make implicit assumptions about input or output parameters or the current system state; should any of these assumptions be wrong, the module's actions may produce unexpected and erroneous results. Interaction with external entities, such as other programs, systems, or humans, amplifies this problem.

EXAMPLE: The *finger* protocol transmits information about a user or system [2106]. Many client implementations assume that the server's response is well-formed. However, if an attacker were to create a server that generated an infinite stream of characters, and a *finger* client were to connect to it, the client would print all the characters. As a result, log files and disks could be filled up, resulting in a denial of service attack on the querying host. This is an example of incorrect assumptions about the input to the client.

14.2.4 Principle of Complete Mediation

This principle restricts the caching of information, which often leads to simpler implementations of mechanisms.

> **Definition 14–5.** The *principle of complete mediation* requires that all accesses to objects be checked to ensure that they are allowed.

Whenever a subject attempts to read an object, the operating system should mediate the action. First, it determines if the subject is allowed to read the object. If so, it provides the resources for the read to occur. If the subject tries to read the object again, the system should check that the subject is still allowed to read the object. Most systems would not make the second check. They would cache the results of the first check and base the second access on the cached results.

EXAMPLE: When a UNIX process tries to read a file, the operating system determines if the process is allowed to read the file. If so, the process receives a file descriptor encoding the allowed access. Whenever the process wants to read the file, it presents the file descriptor to the kernel. The kernel then allows the access.

If the owner of the file disallows the process permission to read the file after the file descriptor is issued, the kernel still allows access. This scheme violates the principle of complete mediation, because the second access is not checked. The cached value is used, resulting in the denial of access being ineffective.

The mediator should check that the request comes from the claimed source (authenticity) and that it has not been tampered with (integrity). After those are validated, the access should be granted if, and only if, the access is authorized. Failure to check authenticity and integrity can cause security problems.

EXAMPLE: The Domain Name Service (DNS) caches information mapping host names into IP addresses. If an attacker is able to "poison" the cache by implanting records associating a bogus IP address with a name, one host will route connections to another host incorrectly. Section 15.6.1.2 discusses this in more detail.

14.2.5 Principle of Open Design

This principle suggests that security should not depend solely on secrecy.

> **Definition 14–6.** The *principle of open design* states that the security of a mechanism should not depend on the secrecy of its design or implementation.

Designers and implementers of a program must not depend on secrecy of the details of their design and implementation to ensure security. Others can ferret out such details either through technical means, such as disassembly and analysis, or through nontechnical means, such as searching through garbage receptacles for source code listings (called "dumpster-diving"). If the strength of the program's security depends on the ignorance of the user, a knowledgeable user can defeat that security mechanism. The term "security through obscurity" captures this concept exactly.

This is especially true of cryptographic software and systems. Because cryptography is a highly mathematical subject, companies that market cryptographic software or use cryptography to protect user data frequently keep their algorithms secret. Experience has shown that such secrecy adds little if anything to the security of the system. Worse, it gives an aura of strength that is all too often lacking in the actual implementation of the system.

Keeping cryptographic keys and passwords secret does not violate this principle, because a key is not an algorithm. However, keeping the enciphering and deciphering algorithms secret would violate it.

Issues of proprietary software and trade secrets complicate the application of this principle. In some cases, companies may not want their designs made public, lest their competitors use them. The principle then requires that the design and implementation be available to people barred from disclosing it outside the company.

EXAMPLE: The Content Scrambling System (CSS) is a cryptographic algorithm that protects DVD movie disks from unauthorized copying. The DVD disk has an authentication key, a disk key, and a title key. The title key is enciphered with the disk key. A block on the DVD contains several copies of the disk key, each enciphered by a different player key, and a checksum of the disk key. When a DVD is inserted into a DVD player, the algorithm reads the authentication key and then authenticates the device (presumably to verify it is allowed to read the following keys). It then deciphers the disk keys using the DVD player's unique key. When it finds a deciphered key with the correct hash, it uses that key to decipher the title key, and it uses the title key to decipher the movie [1824]. (Figure 14–1 shows the layout of the keys.)

The authentication and disk keys are not located in the file containing the movie, so if one copies the file, one still needs the DVD disk in the DVD player to be able to play the movie.

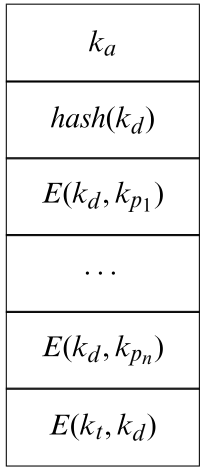

Figure 14–1 DVD key layout. k_a is the authentication key, k_t the title key, k_d the disk key, and k_{p_i} the key for DVD player i.

In 1999, a group in Norway acquired a (software) DVD playing program that had an unenciphered key. They also derived an algorithm completely compatible with the CSS algorithm from the software. This enabled them to decipher any DVD movie file. Software that could perform these functions rapidly became available throughout the Internet, much to the discomfort of the DVD Copyright Control Association, which promptly sued to prevent the code from being made public [643, 1465]. As if to emphasize the problems of providing security by concealing algorithms, the plaintiff's lawyers filed a declaration containing the source code of an implementation of the CSS algorithm. When they realized this, they requested that the declaration be sealed from public view. By then, the declaration—with the source code—had been posted on several Internet sites, including one that had more than 21,000 downloads of the declaration before the court sealed it [1278].

14.2.5.1 Minimize Secrets

The principle of open design implies that the designer should minimize secrets. Secrets can leak no matter how confidential one thinks they are—and mistakes do occur that sometimes reveal them, as in the above example. Protecting the confidentiality of a few secrets is typically simpler than protecting the confidentiality of many secrets.

This rule also suggests that designers should plan for the compromise of any secrets. When a secret is compromised, it should be simple and quick to restore the system to a state where the (formerly) secret data has no value. Minimizing the number of secrets reduces the number of these contingency plans, simplifying management.

14.2.6 Principle of Separation of Privilege

This principle is restrictive because it limits access to system entities.

> **Definition 14–7.** The *principle of separation of privilege* states that a system should not grant permission based on a single condition.

This principle is equivalent to the separation of duty principle discussed in Section 6.1. Company checks for more than $75,000 must be signed by two officers of the company. If either does not sign, the check is not valid. The two conditions are the signatures of both officers.

Similarly, systems and programs granting access to resources should do so only when more than one condition is met. This provides a fine-grained control over the resource as well as additional assurance that the access is authorized.

EXAMPLE: On Berkeley-based versions of the UNIX operating system, the program *su*, which enables users to change from their accounts to the *root* account, requires two conditions to be met. The first condition is that the user knows the *root* password. The second condition is that the user is in the *wheel* group (the group with GID 0). Meeting either condition is not sufficient to acquire *root* access; meeting both conditions is required.

14.2.7 Principle of Least Common Mechanism

This principle is restrictive because it limits sharing.

> **Definition 14–8.** The *principle of least common mechanism* states that mechanisms used to access resources should not be shared.

Sharing resources provides a channel along which information can be transmitted, and so such sharing should be minimized. In practice, if the operating system provides support for virtual machines, the operating system will enforce this privilege automatically to some degree (see Chapter 18, "Confinement Problem"). Otherwise, it will provide some support (such as a virtual memory space) but not complete support (because the file system will appear as shared among several processes).

EXAMPLE: A website provides electronic commerce services for a major company. Attackers want to deprive the company of the revenue it obtains from that website. They flood the site with messages and tie up the electronic commerce services. Legitimate customers are unable to access the website and, as a result, take their business elsewhere.

Here, the sharing of the Internet with the attackers' sites caused the attack to succeed. The appropriate countermeasure would be to restrict the attackers' access to the segment of the Internet connected to the website. Techniques for doing this include proxy servers such as the Purdue SYN intermediary [1695] or traffic throttling (see Section 7.4, "Availability and Network Flooding"). The former targets suspect connections; the latter reduces the load on the relevant segment of the network indiscriminately.

Minimizing the number of shared mechanisms also reduces the scope of an attack that compromises such a mechanism. If all versions of an operating system use the same program, then compromising that single program enables attackers to compromise any system of that type. But if the systems each use a slightly different version of the program, then compromise becomes more difficult.

EXAMPLE: Attack tools assume an underlying structure or configuration of a system or program. In order to invalidate this assumption, researchers have studied how to inject artificial diversity effectively into programs and systems. Then the attack tools will not work properly.

Object code obfuscation tools scramble the flow of execution and the placement of data in memory. For example, many attacks target the return address for function calls, which is stored on a stack and thus in a predictable location. Adding a layer of indirection requires changing the function call and return sequence. Then an attempt to overwrite the return address will change the index into the table instead. By appropriately constraining that value and obscuring how the actual return addresses are stored, the attacker will be unlikely to guess the actual location of the return address, defeating this class of attacks [381]. Other techniques randomize the order of variables and functions in memory or introduce random gaps between formerly contiguous areas of storage, and locations of memory regions. This renders ineffective attack tools that rely on the memory layout of the program [193].

14.2.8 Principle of Least Astonishment

This principle recognizes the human element in computer security.

> **Definition 14–9.** The *principle of least astonishment* states that security mechanisms should be designed so that users understand the reason that the mechanism works the way it does and that using the mechanism is simple.

This principle requires security mechanisms to use a model that the target audience (users and system administrators, typically) can easily understand. If the audience's mental model is too different than that used by the designers and implementers, then their confusion may undermine the security mechanisms.

Thus, configuring and executing a program should be as easy and as intuitive as possible, and any output should be clear, direct, and useful. If security-related software is too complicated to configure, system administrators may unintentionally set up the software in a nonsecure manner. Similarly, security-related user programs must be easy to use and must output understandable messages. If a user is changing a password, and the proposed password is rejected, the password changing program should state why it was rejected rather than giving a cryptic error message. If a configuration file has an incorrect parameter, the error message should describe the proper parameter.

EXAMPLE: The *ssh* program [131, 2058] allows a user to set up a public key mechanism for enciphering communications between systems. The installation and configuration mechanisms for the UNIX version allow one to arrange that the public key be stored locally without any password protection. In this case, one need not supply a password to connect to the remote system, but will still obtain the enciphered connection. This mechanism satisfies the principle of least astonishment.

On the other hand, security requires that the messages impart no unnecessary information.

EXAMPLE: When a user supplies the wrong password during login, the system should reject the attempt with a message stating that the login failed. If it were to say that the password was incorrect, the user would know that the account name was legitimate. If the "user" were really an unauthorized attacker, she would then know the name of an account for which she could try to guess a password.

Balancing the needs of security and the mental models of users requires that the designers and implementers take into account the environment in which the security mechanisms are used.

EXAMPLE: A mainframe system allows users to place passwords on files. Accessing the files requires that the program supply the password. Although this mechanism violates the principle as stated, it is considered sufficiently minimal to be acceptable. On an interactive system, where the pattern of file accesses is more frequent and more transient, this requirement would be too great a burden to be acceptable.

14.2.8.1 Psychological Acceptability

The principle of least astonishment is similar to one of Saltzer's and Schroeder's original principles, the *principle of psychological acceptability*. That principle stated that that security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present. The difference between that principle and the principle of least astonishment is that the former expressed an ideal, whereas the latter recognizes that security mechanisms may add additional steps to accessing the resource. The question is whether those additional steps are unnecessarily difficult to take to the particular population of users of the system.

14.3 Summary

The design principles discussed in this chapter are fundamental to the design and implementation of security mechanisms. They encompass not only technical details but also human interaction. Several principles come from nontechnical environments, such as the principle of least privilege. Each principle involves the restriction of privilege according to some criterion, or the minimization of complexity to make the mechanisms less likely to fail.

14.4 Research Issues

These principles pervade all research touching on the design and implementation of secure systems. The principle of least privilege raises the issue of granularity of privilege. Is a "write" privilege sufficient, or should it be fragmented—for example, into "write" and "write at the end" or "append," or into the ability to write to specific blocks? How does the multiplicity of rights affect system administration and security management? How does it affect architecture and performance? How does it affect the user interface and the user's model of the system?

Least common mechanism problems arise when dealing with denial of service attacks, because such attacks exploit shared media. The principle of least common mechanism plays a role in handling covert channels, which are discussed further in Chapter 18.

Separation of privilege arises in the creation of user and system roles. How much power should administrative accounts have? How should they work together? These issues arise in role-based access control, which is discussed in Section 8.4.

The principle of complete mediation runs counter to the philosophy of caching. One caches data to keep from having to retrieve the information when it is next needed, but complete mediation requires the retrieval of access permissions. How are these conflicting forces balanced in practice?

Research in software and systems design and implementation studies the application of the principle of economy of mechanism. How can interfaces be made simple and consistent? How can the various design paradigms lead to better-crafted, simpler software and systems?

Whether "open source" software (software the source of which is publicly available) is more secure than other software is a complex question. Analysts can check open source software for security problems more easily than they can software for which no source is available. Knowing that one's coding will be available for public scrutiny should encourage programmers to write better, tighter code. On the other hand, attackers can also look at the source code for security flaws, and various pressures (such as time to market) weigh against careful coding. Furthermore, the debate ignores security problems introduced by misconfigured software, or software used incorrectly.

Experimental data for the debate about the efficacy of open source software is lacking. An interesting research project would be to design an experiment that would provide evidence either for or against the proposition that if source code for software is available, then that software has (or causes) fewer security problems than software for which source code is not available. Part of the research would be to determine how to make this question precise, what metrics and statistical techniques should be used to analyze the data, and how the data should be collected.

An understanding of people's world views, and mental models of how computers and security should work, are the basis for applying the principle of least astonishment. The user interface of many security mechanisms, and the details that users must master, differ from their real-world counterparts for a variety of reasons. Thus, understanding how to communicate security issues to people, and tailoring mechanisms to interpret user commands properly, is an area of active research in both the security and human factors communities.

14.5 Further Reading

Many papers discuss the application of these principles to security mechanisms. Succeeding chapters will present references for this aspect of the principles. Other papers present different sets of principles. These papers are generally specializations or alternative views of the principles in this chapter, tailored for particular environments. Abadi and Needham [4] and Anderson and Needham [60] discuss principles for the design of cryptographic protocols; Syverson discusses their limits [1847], and Moore [1377] and Abadi [2] describe problems in cryptographic protocols. Wood [2019, 2020] discusses principles for secure systems design with an emphasis on groupware. Shapiro and Hardy elaborate on a set of principles underlying the design of the operating system EROS [1729]. Bonyun [268] focuses on architectural principles. Landwehr and Goldschlag [1135] consider Internet security. Other examples are for authentication protocols used in the infrastructure of the power grid [1049], for designing privacy constraints into systems [367], and for computer forensics [1506].

Principles for interacting with people are also under study. Yee discusses principles for user interfaces for secure systems [2051]. Peisert et al. [1510] identify principles of authentication that correspond to physical validation of identity. Stajano and Wilson [1808] present some principles underlying successful computer scams, and from them derive principles for protecting people. Motiee et al. [1387] examine user considerations about the use of the principle of least privilege.

14.6 Exercises

1. The PostScript language [18] describes page layout for printers. Among its features is the ability to request that the interpreter execute commands on the host system.
    a. Describe a danger that this feature presents when the language interpreter is running with administrative or root privileges.
    b. Explain how the principle of least privilege could be used to ameliorate this danger.
2. A common technique for inhibiting password guessing is to disable an account after three consecutive failed login attempts (see Section 13.4.2).
    a. Discuss how this technique might prevent legitimate users from accessing the system. Why is this action a violation of the principle of least common mechanism?
    b. One can argue that this is an example of fail-safe defaults, because by blocking access to an account under attack, the system is defaulting to a known, safe state. Do you agree or disagree with this argument? Justify your answer.
3. Kernighan and Plauger [1041] argue a minimalist philosophy of tool building. Their thesis is that each program should perform exactly one task, and more complex programs should be formed by combining simpler programs. Discuss how this philosophy fits in with the principle of economy of mechanism. In particular, how does the advantage of the simplicity of each component of a software system offset the disadvantage of a multiplicity of interfaces among the various components?
4. Design an experiment to determine the performance impact of checking access permissions for each file access (as opposed to once at the file's opening). If you have access to a system on which you can modify the file access mechanism, run your experiment and determine the impact.
5. A company publishes the design of its security software product in a manual that accompanies the executable software.
    a. In what ways does this satisfy the principle of open design? In what ways does it not?
    b. Given that the design is known, what advantages does keeping the source code unavailable give the company and those who purchase the software? What disadvantages does it cause?

6. Assume that processes on a system share no resources. Is it possible for one process to block another process's access to a resource? Why or why not? From your answer, argue that denial of service attacks are possible or impossible.

7. Given that the Internet is a shared network, discuss whether preventing denial of service attacks is inherently possible or not possible. Do systems connected to the Internet violate the principle of least common mechanism?

8. A program called *lsu* [219] gives access to role accounts. The user's access rights are checked, and the user is required to enter her password. If access rules allow the change and the user's password is correct, *lsu* allows the change. Given that Mary uses *lsu* from her account, why does *lsu* require her to enter her password? Name the principles involved, and why they require this.

9. Recall the S/Key one-time password algorithm discussed in Section 13.5.1. When a user prints a list of S/Key passwords for future use, the system encodes each hash value as a set of six short words and prints them. Why does it not merely print out the hash values?

10. The program *su* enables a UNIX user to access another user's account. Unless the first user is the superuser, *su* requires that the password of the second user be given. A (possibly apocryphal) version of *su* would ask for the user's password and, if it could not determine if the password was correct because the password file could not be opened, *immediately* grant superuser access so that the user could fix the problem. Discuss which of the design principles this approach meets, and which ones it violates.

11. Among the design principles Yee [2051] identifies is the *principle of expected ability*, which says that the interface must not lead the user to believe it is possible to do something that cannot be done. Which of the design principles in this chapter support this principle?