Chapter 1
Security Principles

Our objective in writing this book is to help readers improve their understanding of information security by carrying out hands-on experiments where they apply and deepen their knowledge. We will not cover all the related theory and assume that the reader has a basic knowledge of cryptography and information security, perhaps from other courses or books. Nevertheless, we will summarize some of the more central notions that are relevant for the experiments.

This first chapter is cross-cutting in that we summarize principles that are relevant for the coming chapters. We present 12 security principles that provide guidelines on how to incorporate security into system design. The principles are stated as generally as possible and should help the reader to discover commonalities among the more concrete design practices presented in the subsequent chapters.

1.1 Objectives

After reading this chapter you should understand the 12 principles and be able to explain each of them to an IT specialist who is unfamiliar with them. Moreover, you should be able to provide several examples of adherence to the principles or of how their violation results in security problems.

1.2 Problem Context

In this book, we will focus on security of systems, where the system typically is a computer platform that runs software, such as a web server. Security is often defined with respect to a policy describing which kinds of actions are authorized. Consider, for example, an e-mail service where only legitimate users who are registered with the service should be allowed to access their mailboxes. This security policy is vi-

D. Basin et al., *Applied Information Security*,
DOI 10.1007/978-3-642-24474-2_1,

olated if a malicious, illegitimate user can access the mailbox of a legitimate user, for example by acquiring his authentication credentials.

The example of the e-mail service illustrates the complexity of information security. Even a simple platform providing an e-mail service typically involves several subsystems, such as a web front end, a mail server and a database. The malicious user therefore has numerous possibilities for attacking the system. Aside from determining a valid user name and password combination, he may also exploit vulnerabilities in the web server, the database, the operating systems they reside on, etc. This simplifies the adversary's job as a single vulnerability may undermine the system's security. In contrast, those responsible for the system's security must consider the entire system and leave no stone unturned. Security cannot be isolated to a single system component, and engineering secure systems has ramifications through all phases of system development, deployment and operation.

In the next section, we present a set of security principles that have been taken directly or derived from the scientific and engineering literature [5, 12, 18, 22, 24]. The principles provide guidelines on how to account for security during the development process and how to evaluate existing solutions.

1.3 The Principles

Classic information security goals are confidentiality, integrity, availability and accountability. The following selection of security principles helps us achieve these goals and analyze systems with regard to their security. This selection is not intended to be comprehensive, but it does include those principles which are most essential, regardless of the actual domain. This means that they apply to operating system security as well as to application and network security.

To enable their broad use, we formulate these principles abstractly in terms of *subjects* and *objects*. Subjects are active entities, such as a user or a system acting on a user's behalf. Objects are passive containers that store information as data. Access to an object usually implies access to the data it contains. Examples of objects are records, blocks, pages, segments, files, directories and file systems.

For many of these principles, we refer to Saltzer and Schroeder, whose article [18] on the subject is a classic. Although they wrote their paper over 35 years ago, most of the principles they state are just as relevant and clear today as they were then.

1.3.1 Simplicity

Keep it simple.

This principle applies to any engineering and implementation task involved in designing or maintaining a system. More generally, it is a good guideline whenever a problem needs to be solved and often reflects the quality of a solution. The simpler a solution, the easier it is to understand.

Simplicity is desirable for all aspects of system design and development, for operation and maintenance as well as for security mechanisms. Simpler systems are less likely to contain flaws than complex ones. Moreover, simpler systems are easier to analyze and review, and it is thus easier to establish their trustworthiness.

Saltzer and Schroeder call this principle economy of mechanism. They point out its importance when protection mechanisms are inspected on the software and hardware levels. For such inspections to succeed, a small and simple design is essential.

1.3.2 Open Design

The security of a system should not depend on the secrecy of its protection mechanisms.

Saltzer and Schroeder describe this principle very accurately:

> The mechanisms should not depend on the ignorance of potential adversaries, but rather on the possession of specific, more easily protected, keys or passwords. This decoupling of protection mechanisms from protection keys permits the mechanisms to be examined by many reviewers without concern that the review may itself compromise the safeguards. . . . It is simply not realistic to attempt to maintain secrecy for any system which receives wide distribution.

In cryptography this is known as Kerckhoffs' principle [7], which states that a cryptosystem should be secure even if all aspects of the system (except the keys being used) are public knowledge.

Secrets are hard to protect. Secrets must be stored somewhere, for example, in human memory, in machine memory, on disks or external devices, and protected accordingly. The amount of information that needs to be kept secret should therefore be reduced to a minimum.

Example 1.1. We do not design doors that only authorized persons know how to open and close. Instead, we design standardized doors with standardized locks (both with different protection levels) and rely on the protection of the associated key.

1.3.3 Compartmentalization

Organize resources into isolated groups of similar needs.

Compartmentalization means organizing resources into groups (also called compartments or zones), each of which is isolated from the others, except perhaps for some limited and controlled means of exchanging information. The principle of compartmentalization is applied in different areas in computer science, for example, in programming, where functions and variables are grouped and put into separate modules or classes.

Example 1.2.

1. Sensitive applications are often run on separate computers in order to minimize the impact of an attack: If one computer and its application are compromised, this does not entail the compromise of other applications. The same applies to the different tiers in web-based applications. They are usually placed on separate servers, for example, to protect the database in case the business logic is compromised on the application server.
2. There are other ways to separate applications and operating systems. They are based on software solutions that separate applications on a single physical machine. Common technologies for achieving this are Kernel/User-Mode in Unix systems, VirtualBox, VMware, Xen, VServer, Jail, chroot or hard disk partitions.
3. Compartmentalization is often used in networks. Filtering devices such as firewalls are employed to partition a network into separate zones, and communication between zones is governed by a policy. The objective is to increase the difficulty of attacking machines or servers from a host in a different zone. A common zoning concept includes an internal network and a demilitarized zone (DMZ) with connections to external networks and the Internet.
4. Compartmentalization is also important in software development and is supported by most modern programming languages. Language-based mechanisms, such as encapsulation, modularization and object-orientation, encourage and enable compartmentalization. These mechanisms also help to build more secure systems.

Compartmentalization has several positive characteristics. First, it often facilitates simplification, since compartments contain resources with the same (or similar) needs. For example, coarse-grained access control is easier to understand, implement, review and maintain than fine-grained access control. Second, problems resulting from attacks, operational problems, unintended mishaps and the like are often isolated to a single compartment. Compartmentalization thus reduces the negative effects of the problem and provides mechanisms for tackling it. For example, one may disconnect an affected network compartment from the Internet, or stop a misbehaving process. Third, highly security-sensitive functionality can be placed in

one specific compartment in which special security measures and policies can be enforced.

An important application of compartmentalization in software engineering is the separation of data and code. Many protocols and data structures mix data and code which can lead to security vulnerabilities. Mixing data and code on the stack leads to buffer overflows. Mixing data and code in office documents can result in malicious documents, e.g., those containing macro viruses. Mixing data and code in web pages leads to cross-site scripting, while mixing data and code in SQL statements leads to SQL injections, and so on.

Problem 1.1 Can you name other examples where the failure to separate data and code leads to security problems?

While compartmentalization implies some form of separation, it is often infeasible to completely isolate resources, functions, information, communication channels or whatever the objects under consideration are. In many cases connections between compartments remain, and special care must be taken to tightly control the corresponding interfaces in such a way that an interface itself does not become a source of vulnerabilities.

Examples of such connections include mechanisms for interprocess communication in an operating system (such as pipes, sockets, temporary files and shared memory), network devices (such as a firewall) that connect one network area to another, or application programming interfaces.

Saltzer and Schroeder discuss the least common mechanism principle, which makes a similar statement:

> Minimize the amount of mechanism common to more than one user and depended on by all users. Every shared mechanism (especially one involving shared variables) represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security.

Example 1.3. Prisons are often compartmentalized to prevent prisoners from forming large groups. Submarines have compartments to ensure that one leak does not sink the entire vessel. Buildings have compartments to prevent fires from spreading, and different areas are allocated to different groups of fans at soccer games.

1.3.4 Minimum Exposure

Minimize the attack surface a system presents to the adversary.

This principle mandates minimizing the possibilities for a potential adversary to attack a system. It advocates the following:

1. Reduce external interfaces to a minimum.
2. Limit the amount of information given away.
3. Minimize the window of opportunity for an adversary, for example, by limiting the time available for an attack.

Example 1.4.

1. An important security measure in every system is to disable all unnecessary functionality. This applies in particular to functionality that is externally available. Examples include networked services in an operating system or connectivity options like infrared, WLAN or Bluetooth in a handheld device.
2. Some systems provide information that can help an adversary to attack a system. For instance, poorly configured web servers may reveal information about installed software modules, including their version and even their configuration. Minimizing such information leakage improves overall security.
3. Brute-force attacks against password-based authentication mechanisms work by repeatedly trying different user names and passwords until a valid combination is found. Security mechanisms to diminish the window of opportunity for the adversary include locking the account after several failed login attempts, increasing the time the user must wait between failed attempts, or introducing CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) to prevent a noninteractive program from automatically calling the login process.
4. The window of opportunity for an adversary is also diminished when an application supports sessions with an automatic session timeout or, on client computers, with an automatic screen lock. In both cases, the time available to reuse an old session by a user who neglected to logout is reduced.

Example 1.5. A medieval castle did not have a dozen entry points but instead just one or two. In World War II, people were instructed to shade the windows of their homes at night in order not to give away the location of cities to the enemy. Self-closing doors with a spring mechanism minimize the opportunity for a potential intruder to gain access after a legitimate person enters or leaves a building.

1.3.5 Least Privilege

Any component (and user) of a system should operate using the least set of privileges necessary to complete its job.

The principle states that privileges should be reduced to the absolute minimum. As a consequence subjects should not be allowed to access objects other than those really needed to complete their jobs.

Example 1.6.

1. Employees in a company do not generally require access to other employees' personnel files to get their jobs done. Therefore, in accordance with this principle, employees should not be given access to other employees' files. However, a human resources assistant may require access to these files and, if so, is granted the necessary access rights.
2. Most office workers do not need the privileges of installing new software on a corporate computer, creating new user accounts or sniffing network traffic. These employees can do their jobs with fewer privileges, e.g., access to office applications and a directory to store data.
3. A well-configured firewall restricts access to a network or a single computer system, to a limited set of TCP/UDP ports, IP addresses, protocols, etc. If a firewall protects a web server, it usually restricts access to TCP ports 80 (HTTP) and 443 (SSL). These are the least privileges that subjects (web users) require to perform their task (web browsing) on the object (web server).
4. A web server's processes need not run with administrative privileges; they should run with the privileges of a less privileged user-account.

Ensuring this principle requires understanding of the system design and architecture as well as of the tasks that should be carried out by system users. Implementing this principle is often difficult. In a perfect world, the information could be derived from a clearly defined security policy. However, fine-grained and well-defined security policies rarely exist, and necessary access must be identified on a case-by-case basis. Least privilege is a central principle because it helps to minimize the negative consequences of unexpected operation errors, and it reduces the negative effects of deliberate attacks carried out by subjects with privileges.

Example 1.7. Keys to an office building may constitute a mechanism for implementing least privilege: An assistant who only works regular office hours only needs the key to his own office. A senior employee may have keys to his office and the main door to the building. The janitor has keys to every door apart from safes.

1.3.6 Minimum Trust and Maximum Trustworthiness

Minimize trust and maximize trustworthiness.

The difference between a trustworthy and a trusted system is important. A user of a system has an expectation of how the system should behave. If the user trusts the system, he assumes the system will satisfy this expectation. This is just an assumption, however, and a trusted system may "misbehave", in particular by acting maliciously. In contrast, a trustworthy system satisfies the user's expectations.

This principle implies the need either to minimize expectations and thus to minimize the trust placed in a system or to maximize the system's trustworthiness. This is particularly important when interacting with external systems or integrating third-party (sub-)systems. Minimizing expectations can result in a complete loss of trust and therefore in using the external system only for non–security-relevant tasks. Maximizing trustworthiness means turning assumptions into validated properties. One way to do so is to rigorously prove that the external systems behave only in expected (secure) ways.

In general, trust should be avoided whenever possible. There is no guarantee that the assumptions made are justified. For example, in the case of a system relying on external input, it should not trust that it is given only valid inputs. Instead the system should verify that its input is actually valid and take corrective actions when this is not the case. This eliminates the trust assumption.

Example 1.8. Web applications are often susceptible to vulnerabilities caused by unvalidated input. Typically, web applications receive data from web clients, e.g., in HTTP headers (such as cookies). This data need not be benign: It could be tampered with to cause a buffer overflow, SQL injection, or a cross-site scripting attack. Web applications should not trust data received but should instead verify the validity of the data. This usually requires filtering all input and removing potentially "dangerous" symbols.

Trust is generally a transitive relation in system engineering. A system's behavior can depend on other systems that have no direct relation to it and are only related indirectly over a chain of trust. Indeed these other systems might not even be known to the system developers. A standard example of this is the use of remote login procedures like SSH, explained in Chap. 4. A user A on one computer who trusts a user B on another computer can use SSH to enable B to log in to his computer. This trust might be justified because both computers are in the same administrative domain. However, suppose B trusts an external user C, e.g., a supplier who delivers second-level support, and thereby also provides C SSH access to B's computer. By transitivity A also trusts C. This may or may not be acceptable and it must be taken into account when thinking about the security of A.

Problem 1.2 Extend some of the above examples to illustrate the problem of transitive trust.

Example 1.9. Parents tell their children not to take candies from strangers. Countries do not just trust individuals at their borders but base their trust on evidence which is hard to forge, like passports. Airlines and airports do not trust passengers' luggage but have it screened.

1.3.7 Secure, Fail-Safe Defaults

The system should start in and return to a secure state in the event of a failure.

Security mechanisms should be designed so that the system starts in a secure state and returns to a secure default state in case of failure. For example, a security mechanism can be enabled at system start-up and re-enabled whenever the system or a subsystem fails. This principle also plays an important role in access control: The default and fail-safe state should prevent any access. This implies that the access control system should identify conditions under which access is granted. If the conditions are not identified, access should be denied (the default). This is often called a *whitelist approach*: Permission is denied unless explicitly granted. The opposite (less secure) variant is the *blacklist approach*: Permission is granted unless explicitly denied.

Saltzer and Schroeder mention an additional reason why denying access as the default may be advantageous in case of failures:

> A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, which is a safe situation since it will be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use.

Example 1.10.

1. Many computers are equipped with security mechanisms like personal firewalls or antivirus software. It is essential to have these mechanisms enabled by default and to ensure that they are reactivated after a crash or a manual reboot. Otherwise an adversary only needs to provoke a reboot to deactivate these security mechanisms.
2. Most firewall rule sets follow the whitelist approach. The default rule is to deny access to any network packet. Thus a packet is allowed through only if some rule explicitly permits it to pass, and otherwise it is dropped.
3. Crashes and failures often leave marks on a system: core dumps, temporary files or the plaintext of encrypted information that was not removed before the crash. The principle of fail-safe defaults states that failures are taken into account and clean-up measures are taken during the error-handling process. The disadvantage, however, is that removing this information may complicate troubleshooting or forensics.

Example 1.11. Doors to a building often lock when closed and cannot be opened from the outside; indeed, they may not even be equipped with a doorknob on the outside. However, safety requirements may specify that doors must open in emergency situations (e.g., emergency exits or fire doors).

1.3.8 Complete Mediation

Access to any object must be monitored and controlled.

This principle states that each access to every security-relevant object within a system must be controlled. Thus, an access control mechanism must encompass all relevant objects and must be operational in any state the system can possibly enter. This includes normal operation, shutdown, maintenance mode and failure.

Care should be taken to ensure that the access control mechanism cannot be circumvented. Sensitive information should also be protected during transit and in storage, which often requires data to be encrypted in order to achieve complete mediation. A system that merely controls access to unencrypted objects can often be attacked by means of layer-below attacks. In a layer-below attack an adversary gains unauthorized access to an object by requesting it below the enforcement layer. Examples of this include booting a different operating system to circumvent file-system-based access control, or sniffing traffic to circumvent access control mechanisms imposed by a web application.

Saltzer and Schroeder additionally mention identification as a prerequisite to complete mediation, and they point out the risks of caching authorization information:

> [Complete mediation] implies that a foolproof method of identifying the source of every request must be devised. It also requires that proposals to gain performance by remembering the result of an authority check be examined skeptically. If a change in authority occurs, such remembered results must be systematically updated.

Example 1.12.

1. A computer's memory management unit is a hardware component that, among other responsibilities, enforces range checks on every memory access request.
2. An encrypted file system can help to ensure complete mediation in cases where file system access control is enforced by the operating system and therefore is not necessarily guaranteed when the operating system is not running.
3. Canonicalization is the process of converting data that has multiple representations into a normalized, canonical representation. Canonicalization issues are a common cause of problems for complete mediation; for example, a check for authority may fail due to a special encoding or alternate form. Examples are file system paths (`/bin/ls` vs. `/bin/..///bin/./ls`), host names (www.abc.com vs. 199.181.132.250) and URLs (www.xyz.com/abc vs. www.xyz.com/%61%62%63).

Example 1.13. Airport authorities take great care to ensure that each and every subject is properly authenticated before entering sensitive areas of the airport or boarding a plane. Airports ensure complete mediation by using architectural means and by controlling the flow of passengers throughout the airport.

1.3.9 No Single Point of Failure

Build redundant security mechanisms whenever feasible.

This principle states that security should not rely on a single mechanism. As a consequence, if one mechanism fails, there are others in place that can still prevent malice, so there is no single point of failure. This principle is also known as defense in depth. The principle itself does not stipulate how many redundant mechanisms should be employed. This must be determined on the basis of a cost-benefit analysis. Indeed, only a single mechanism may be possible due to other conflicting requirements such as financial resources, usability, performance, administrative overhead, etc. Still, it is good practice to try and prevent single points of failure when feasible.

A common technique for preventing single points of failure is separation of duties. Saltzer and Schroeder describe it as follows:

> Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key. ... The two keys can be physically separated and distinct programs, organizations, or individuals made responsible for them. From then on, no single accident, deception, or breach of trust is sufficient to compromise the protected information.

Example 1.14.

1. Two-factor authentication relies on two components, often a physical token (e.g., a one-time password generator) and a PIN. A single factor on its own is not sufficient for authentication.
2. Many companies install antivirus software on their mail servers, Internet proxies, file servers, client computers and server machines. One reason is that there are different infection vectors, not all of which can be addressed by an installation in just one place. Another reason is to prevent single points of failure: The antivirus software on the mail server could fail, perhaps because it crashed or somebody stopped it and forgot to restart it, or because a message is encrypted. In these cases, the antivirus software on the client can still detect the malware. Some companies even employ the software of two different antivirus software vendors. If one product fails to detect a certain type of malware, the other might still succeed.
3. It is good practice both to employ a firewall at the network perimeter and to secure internal applications by hardening their operating systems and running applications with minimal privileges. In the event that an adversary is able to circumvent the firewall, the individual servers and applications should still be able to withstand attacks.

Example 1.15. There are numerous examples of this principle from the physical world. Airplanes fly with four engines instead of just one or two. High-security safes are equipped with two locks and the corresponding keys are held by different people; both keys must be used to unlock the door (separation of duties). Credit

cards are delivered through postal mail and the corresponding PIN codes are often delivered in a separate letter a few days earlier or later. Thus, theft or loss of one letter alone does not compromise the security of the credit card.

1.3.10 Traceability

Log security-relevant system events.

A trace is a sign or evidence of past events. Traceability therefore requires that the system retains traces of activities. A frequently used synonym for trace is the term audit trail, which is defined as a record of a sequence of events, from which the system's history may be reconstructed.

Traceability is ensured by providing good log information. So when you design a system you must determine which information is relevant and provide proper logging infrastructure. This includes planning where and for how long log information is stored, and whether additional security measures are necessary. Additional security mechanisms include making backups of log information, logging to a tamper-resistant or tamper-evident device, using encryption to ensure confidentiality, or using digital signatures to ensure the integrity and authenticity of the logs. Good log information is useful for many purposes: to detect operational errors and deliberate attacks, to identify the approach taken by the adversary, to analyze the effects of an attack, to minimize the spread of such effects to other systems, to undo certain effects, and to identify the source of an attack.

Traceability is often an important prerequisite for accountability, i.e., linking an action to a subject that can be held responsible. An additional requirement for accountability is the use of unique identifiers for all subjects, especially users. Do not use shared accounts, as they cannot be linked to individuals.

The objective of linking actions to persons can interfere with individual privacy and data protection laws. Care must be taken when recording personal data, since the logging should comply with the relevant regulations. Additional security measures should be implemented to ensure data protection where appropriate. One possible measure is to record pseudonymous log information and to store the true user identities in a separate place, where each kind of information is accessible to different people. This is an example of separation of duties.

Example 1.16. Many hard-copy forms in companies have an audit trail. Invoices in particular may require signatures, stamps and other information as they flow through the administrative processes. They are archived afterwards so that it is later possible to determine who checked an invoice or who cleared it.

1.3.11 Generating Secrets

Maximize the entropy of secrets.

Following this principle helps to prevent brute-force attacks, dictionary-based attacks or simple guessing attacks. In short, it helps keep secrets secret.

Example 1.17. Use a good pseudorandom number generator and sufficiently large keys when generating session tokens (e.g., in web applications), passwords and other credentials (especially secrets that are shared between devices for mutual authentication), and all cryptographic keys. In case of passwords generated by humans, special care must be taken to ensure they are not guessable.

Example 1.18. A combination lock for a bicycle is the natural offline example. The smaller the key space (often 10^3), the easier it is to open the lock by trying all combinations. Similarly, if the key is predictable (e.g., a trivial combination like 000), the lock is also quickly opened.

1.3.12 Usability

Design usable security mechanisms.

Security mechanisms should be easy to use. The more difficult a security mechanism is to use, the more likely it is that users will circumvent it to get their job done or will apply it incorrectly, thereby introducing new vulnerabilities. Only a few examples of this principle appear in this book, but it is certainly important enough to be included.

Saltzer and Schroeder call this principle psychological acceptability:

> It is essential that the human[-computer] interface should be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Also, to the extent that the user's mental image of his protection goals matches the mechanisms he must use, mistakes will be minimized. If he must translate his image of his protection needs into a radically different specification language, he will make errors.

This principle is not only concerned with end users: It applies to all system staff, including system administrators, user administrators, auditors, support staff and software engineers. As with all other mechanisms, security mechanisms must be designed with these users and their limitations in mind.

Example 1.19.

- Most end users do not understand cryptographic mechanisms. They do not understand what a certificate is, its intended use and how to verify the authenticity of

a server certificate. As a consequence, it is possible to impersonate a web server even in settings where server certificates are used.
- With an overly restrictive operating system setup, system administrators often resort to using the root account by default to circumvent restrictions imposed upon their user accounts. Otherwise they cannot work efficiently.

Example 1.20. Some doors close and lock automatically and must be unlocked with a key to be reopened. If such doors are used frequently, people are likely to keep them from ever closing by blocking them with a chair or some other object.

1.4 Discussion

Most real systems do not adhere to all the principles we have given. But these principles help us to pay due attention to security when we are designing or analyzing systems or their components. The question of whether a principle has been applied can often not be answered with a simple yes or no. Principles can be realized on different levels, reflecting, for example, the manpower or financial resources available and the risks involved. Moreover, in many situations there are good reasons to deliberately not implement a principle.

> **Problem 1.3** Many of the principles discussed are not independent from the other principles. Some of them overlap, while others are in conflict. Identify at least three cases in which two or more principles overlap or are in conflict. Explain the reasons for this.

1.5 Assignment

When you work through the rest of this book recall the principles discussed in this chapter. Try to associate each topic and experiment with one or several of the principles, asking yourself "Which principle motivates this security measure?" or "Which principle was ignored here, resulting in a security vulnerability?".

1.6 Exercises

Question 1.1 Explain the principle of secure, fail-safe defaults using firewalls as an example.

Question 1.2 Explain the principle of least privilege and provide two applications of this principle in your daily life.

Question 1.3 Explain why security by obscurity typically does not lead to an increased level of security.

Question 1.4 In the terminology of Saltzer and Schroeder [18], complete mediation describes the security principle which requires access checks for any object each time a subject requests access. Explain why this principle may be difficult to implement and give examples that illustrate the difficulties you mention.

Question 1.5 In contrast to the open design principle, the minimum exposure principle demands that the information given away is reduced to a minimum. Are these principles contradictory? Provide examples that support your answer.

Question 1.6 Explain the principle of compartmentalization and give an example.

Question 1.7 Give examples of systems that are trustworthy and trusted, systems that are trusted but not trustworthy, and systems that are trustworthy but not trusted.

Question 1.8 What is a *whitelist approach* and how does it differ from a *blacklist approach*? Give examples of when one approach is preferable to the other.

Question 1.9 A company wants to provide a service to their customers over the Internet. Figure 1.1 shows two different system design solutions.

Solution a: The server providing the service is located in a demilitarized zone (DMZ) that is separated from the Internet and the intranet, controlled by a firewall.

Solution b: The server providing the service is located in the intranet that is separated from the Internet, controlled by a firewall.

Compare the two designs and give at least four arguments for or against them. Use the security principles to substantiate your arguments. Which of these designs would you prefer?