# Software Design and Security

# 24

The objective of this chapter is to introduce information security as an important software design topic.

## 24.1 Preconditions

The following should be true prior to starting this chapter.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_24

## 24.2 Concepts and Context

There are many security concepts and topics that fall within the information security domain. This chapter will briefly describe a number of design principles that researchers suggest will lead to a more secure software system.

### 24.2.1 Information Security Foundations

The three goals when developing information security are listed below. These three goals form the acronym CIA.

Confidentiality: Avoid unauthorized disclosure of information.
Integrity: Information has been altered only in authorized way.
Availability: Information is accessible and modifiable in a timely fashion by authorized entities.

One of the challenges with information security is how to balance these three goals when improving one goal often reduces another. For example, implementing more security mechanisms to improve confidentiality will adversely affect availability. Two-factor authentication is an effective way to increase confidence that a person is who they say they are, but this affects availability when the person's second factor (e.g., cell phone) is not working or lost.

Four additional concepts are also important to understand when you design security into a software product.

Anonymity: A property that certain records or transactions are not attributable to any individual.
Assurance: How trust is provided and managed in computer systems using policies, permissions, and protections.
Authenticity: Ability to determine whether statements, policies, and permissions issued by persons or systems are genuine.
Non-repudiation: User is responsible for their actions and should not be able to deny what they have done.

Anonymity and non-repudiation represent opposing views on whether identifying data is maintained by a system. A crisis prevention hotline would likely use a system which does not require the caller to identify themselves. In contrast, a financial system will likely record information about a user when they perform an action resulting in changes to the data.

### 24.2.2 Designing Information Security in Software

The first set of eight design principles come from an article written by Saltzer and Schroeder that was published in 1975 [1].

Economy of mechanism: Your design should be as small and as simple as possible. This allows quality assurance methods (e.g., formal reviews, design walkthroughs) to have the greatest chance of finding security vulnerabilities.

Fail-safe defaults: Your design should base access to data on permission rather than exclusion. The default situation should be to deny access. Only when the protection scheme identifies conditions to permit access should the data be accessible. In contrast, creating a scheme which describes the conditions for refusing access presents the wrong psychological perspective for a secure software design. To state this another way, the rules needed to express permission are likely to be simpler to understand than the rules needed to refuse permission.

Complete mediation: Each access to an object must be checked for authority. This implies that a foolproof method of identifying the source of each request must be devised and suggests improvements made for performance reasons, e.g., by remembering the result of a previous authority check, be examined skeptically. Care must be taken when a change in authority occurs, to ensure the remembered results are systematically updated.

Open design: Your design should not be a secret. This allows reviewers to comment on the security mechanisms being used while protecting the keys or passwords used by the mechanisms. You should assume your design is not a secret.

Separation of privilege: Requiring two (or more) keys to unlock a protection mechanism is more robust than allowing access after presenting a single key. This allows the two (or more) keys to be physically held by different individuals, organizations, or systems. Examples of this security design principle may be seen in the use of bank safe deposit boxes and in launching a nuclear weapon.

Least privilege: Grant users/systems only those access rights needed to perform their tasks. When software must access an information asset, it should ideally be granted this access only for the moments in time when it is using the information asset. This limits the damage resulting from an accident or error.

Least common mechanism: Minimize the number of mechanisms used by more than one user/system. Each security mechanism shared among most/all users or shared among systems, especially when shared variables are used, represents a potential information path that may unintentionally compromise security. Do not share objects and protection mechanisms, instead create separate instances for each user or system interface.

Psychological acceptability (make security usable): Design the human–computer interface (HCI) for ease of use, to promote correct use of the protection mechanism. A well designed HCI will match the protection mechanism to the user's mental image of their protection goals.

One of the eye-opening aspects of Saltzer and Schroeder's research is the publication year was 1975. This leads one to wonder …what has the software industry been doing for the past 40+ years? Why are there so many vulnerabilities in software systems and applications? Thinking about these eight principles, none of them seem all that challenging to design and implement. And yet the software industry seems to be playing catch-up each time we hear of another large data breach. One thing is clear, the software industry needs to change some fundamental aspects of how software is developed in order to try to get ahead of the malicious attackers.

In 2013, Gary McGraw expanded on Saltzer and Schroeder's eight security design principles by adding five more [2].

Secure the weakest link: The suite of security mechanisms being used are only as good as the weakest security mechanism being used. The analogy often used is that a chain is only as strong as its weakest link. Likewise, a system is only as secure as its weakest security mechanism.

Defend in depth: Your design should include redundancy and layers of defense. This approach looks to manage security risks by using a diverse set of security mechanisms that provide redundant capabilities or are provided by different software layers.

Be reluctant to trust: Be skeptical of security protections not within your software system. The quote "trust but verify" often used to describe international agreements to limit nuclear weapons is a good motto to follow when it comes to placing your software within an operating environment. A cloud provider may claim to provide certain protections, but it is best to verify these as best you can.

Promote privacy: Your design needs to consider the types of personal information you are collecting from a user. Do you really need the information you are requesting? Should the personal information be encrypted? Does this data really need to be persistently stored?

Use your resources: Nobody knows everything about what a good software security design looks like. Talk to others about the design choices you are making. Have experts with different backgrounds review your design.

So what should the software industry do to address the large number of vulnerabilities found in software? It should, *no it must*, incorporate these 13 security design principles into the software development processes (SDP) it uses to design software. In addition, thinking about security should be an integral part of the entire process. Thus an SDP should describe tasks the development team should perform to think about security while developing a project plan, while documenting software needs, while developing a quality assurance plan, while designing the software, and while coding and testing the software.

## 24.2.3 Cryptographic Concepts

Cryptography is used to provide different security mechanisms. The following describes four uses of cryptography.

### 24.2.3.1 Cryptographic Systems

A cryptographic algorithm along with a key may be used to encrypt plaintext data into ciphertext, or to decrypt ciphertext back into plaintext. Two types of cryptographic algorithms are in use today.

Symmetric: The same key is used to encrypt plaintext and decrypt ciphertext.
Asymmetric: One key is used to encrypt plaintext and a different key is used to decrypt ciphertext.

Asymmetric algorithms use a key-pair with one key being public, i.e., known by everyone, and one key being private. The private key is (hopefully) known only by the individual or organization that owns the key-pair. For use as an encryption scheme, a system sending data to the individual or organization will encrypt their plaintext data using the public key. The individual or organization will receive the ciphertext and use their private key to decrypt, producing the original plaintext. Asymmetric encryption is also know as *public-key encryption*.

#### 24.2.3.2 Digital Signatures

A digital signature uses public-key encryption to verify who sent the data. In this scenario, the sender will encrypt their plaintext data using their private key. Typically, the plaintext data includes some identifying information about the sender, which the receiver is expecting to see. The receiver will use the sender's public key to decrypt the data, then look for the *sender's* identifying information in the decrypted plaintext. Since only the keys in the key-pair may be used, and since in theory only the sender knows their private key, a digital signature may be used to verify who sent the data.

#### 24.2.3.3 Cryptographic Hash Functions

A hash function computes a checksum on a data value. A checksum value contains a fixed number of bits, which is usually much smaller than the data value fed into the hash function. A cryptographic hash function is defined mathematically as a one-way function. A one-way hash function produces a checksum given a data value, but it is hard to recreate the data value if all you have is a checksum value. The checksum produced by a cryptographic hash function is commonly called a *message digest*. A cryptographic hash function is an effective way to store passwords without having to store the plaintext value.

#### 24.2.3.4 Digital Certificates

A digital certificate is associated with an entity you want to communicate with. It is used to ensure a public key being used belongs to the entity you want to send data to. A digital certificate originates from a *certificate authority* and combines a public key with identifying information about the entity that owns the public key.

## 24.3 Use in Software Designs

A software design—structured and object-oriented—can use many of the modeling techniques discussed so far to express the inclusion of security principles into a design. In particular, using data-flow diagrams, IDEF0 function models, and UML state machine diagrams gives you the ability to express a design that adheres to many of the security principles just discussed. *In the following subsections, only those security design principles capable of being expressed by using the modeling technique are described.*

### 24.3.1 Data-Flow Diagram

A data-flow diagram (DFD) illustrates the structure and behavior of software by describing the processes, data flows, data stores, and external entities that are in the design.

Economy of mechanism: When a DFD expresses a design at a high level of abstraction, there are likely very few design elements in the diagram. This would give the impression the design is simple. In contrast, a DFD expressing significant design details is a good candidate to assess economy of mechanism. In this case, being able to express the details using a small number of DFD elements would represent a simple design.

Fail-safe defaults: A DFD showing detailed processing may express the use of permission rather than exclusion to access data. The challenge with data-flow diagrams is you only have input and output data flows, so some creativity may be needed to express fail-safe defaults.

Complete mediation: Like fail-safe defaults, a detailed DFD may show authorization being checked each time data is accessed.

Open design: Use various design models, including DFDs, to express and publicize your design.

Separation of privilege: Use processes, data flows and data stores to show a design requiring two (or more) keys to unlock a protection mechanism.

Defend in depth: Use DFDs to show redundancy and layers of defense in your design.

Be reluctant to trust: Use DFDs to show security protections which are outside the scope of your system (by using the external entities notation) and to show how you are skeptical of these security protections by having redundant security protections within your system.

Promote privacy: Use processes, data flows, and data stores to show the types of personal information being collected/stored and the security/privacy mechanisms designed to protect these data assets.

### 24.3.2 IDEF0 Function Model

An IDEF0 function model illustrates the structure and behavior of software by describing the functions and data flows in the design. With four types of data flows—inputs, controls, outputs, mechanisms—an IDEF0 function model can express a significant amount of design details. In particular, the controls data flows can express design constraints (e.g., password required) while the mechanism data flows can express the use of security mechanisms (e.g., AES, which stands for advanced encryption standard).

Economy of mechanism: When an IDEF0 function model expresses a design at a high level of abstraction, there are likely very few design elements in the diagram. This would give the impression the design is simple. In contrast, an IDEF0 function model expressing significant design details is a good candidate to assess economy of mechanism. In this case, being able to express the details using a small number of model elements would represent a simple design.

Fail-safe defaults: The use of control flows and mechanism flows may be used to express a design where access to data is based on permission.

Complete mediation: Like fail-safe defaults, a detailed IDEF0 function model may show authorization being checked each time data is accessed.

Open design: Use various design models, including IDEF0 function models, to express and publicize your design.

Separation of privilege Use functions and the four types of data flows to show a design requiring two (or more) keys to unlock a protection mechanism.

Least privilege: Use functions and the four types of data flows, particularly control and mechanism flows, to show a design which grants users/systems only those access rights needed to perform their tasks.

Defend in depth: Use IDEF0 function models to show redundancy and layers of defense in your design.

Be reluctant to trust: Use IDEF0 function models, in particular input, control and mechanism flows, to show security protections which are outside the scope of your system and to show how you are skeptical of these security protections by having redundant security protections within your system.

Promote privacy: Use functions and the four types of data flows to show the types of personal information being collected/stored and the security/privacy mechanisms designed to protect these data assets.

### 24.3.3 UML State Machine Diagram

A UML state machine illustrates the behavior of software by describing software states and transitions between those states. While a state is described using a name/label, it may also describe three actions—entry, do, and exit. The entry action occurs when transitioning into the state and the exit action occurs when transitioning out of the state. The do action is processing that is done while in the state. A transition is typically described using a trigger (i.e., what causes the transition to occur) but may also include an action and a guard condition. Any action specified is performed when the transition occurs. When a guard condition is specified, it must be true in order for the transition to occur. These UML state machine features allow this modeling technique to represent a wide range of behavior.

Economy of mechanism: When a UML state machine expresses a design at a high level of abstraction, it is likely that there are very few design elements in the diagram. This would give the impression that the design is simple. A UML state machine that expresses significant design details is a good candidate to assess economy of mechanism. In this case, being able to express the details using a small number of model elements would represent a simple design.
Fail-safe defaults: The use of trigger and guard conditions on a transition may be used to express a design where access to data is based on permission.
Complete mediation: The use of states and transitions may be used to express a design which shows how each access to an object is dependent on first checking for authority.
Open design: Use various design models, including state machine diagrams, to express and publicize your design.
Separation of privilege: Use states and transitions to show a design requiring two (or more) keys to unlock a protection mechanism.
Least privilege: Use states and transitions to show a design that grants users/systems only those access rights needed to perform their tasks.
Defend in depth: Use state machine diagrams to show redundancy and layers of defense in your design.

## 24.3.4 Using Other Design Models

The security design principles not listed in the data-flow diagram, IDEF0 function model, and state machine diagram sections are listed below.

Least common mechanism: In an object-oriented design, a UML communication or UML sequence diagram will show when object instances are created. To support this principle, each security mechanism object would need to have a distinct instance created instead of sharing one instance for many users or for many system interfaces. In a structured design, use of local variables within any function supporting a security mechanism would ensure the variables are not shared across function calls. However, the SD design models discussed in this book do not express the use of local variables.
Psychological acceptability: None of the design models introduced in this book are useful in describing a human–computer interface (HCI) design. Determining whether an HCI design is easy to use and promotes correct use of a protection mechanism is challenging, especially when the mechanism is new to users.
Secure the weakest link: This requires someone knows which security mechanism being used is the weakest. Performing a formal review, informal review, or a design walkthrough (refer to Chap. 23) and including cybersecurity experts in the review

is a good way to identify the weakest mechanism. Once the weakest mechanism is identified, discussions can determine how to mitigate this risk.

Use your resources: Clearly, this principle has nothing to do with using design models. Instead, this principle is about people. Take advantage of the knowledge and experience found in your colleagues, friends, and professional network. Getting different perspectives on developing more secure software can help you and your project team produce more robust software.

## 24.4 Post-conditions

The following should have been learned when completing this chapter.

- There are three information security goals: confidentiality, integrity, and availability.
- There are four additional information security concepts which are important to understand: anonymity, assurance, authenticity, and non-repudiation.
- There are 13 security design principles researchers have identified as critical to think about and include when developing a software solution.

1. Economy of mechanism: simpler designs are easier to verify and validate.
2. Fail-safe defaults: allow access to data only when permission (i.e., authentication) has been verified.
3. Complete mediation: each access to data should be checked to ensure proper authority.
4. Open design: allow anyone to review your design.
5. Separation of privilege: use more than one key to unlock a protection mechanism. Store these keys in different locations.
6. Least privilege: give users/systems the minimum access rights necessary to perform their tasks.
7. Least common mechanism—minimize the number of protection mechanisms shared across users/systems.
8. Psychological acceptability: make security usable; align the user's mental image of their protection goals with your HCI design.
9. Secure the weakest link: spend more time on improving the weakest security mechanism within your system.
10. Defend in depth: design security into different software layers of your system; use redundant capabilities to provide a more robust set of security mechanisms.
11. Be reluctant to trust: be skeptical of security mechanisms provided by third-party software—trust but verify these mechanisms.
12. Promote privacy: your design needs to consider the types of personal information being collected and stored.

13. Use your resources: talk to others about your design choices; no one knows everything about good software security.

- Cryptographic concepts include: cryptographic systems, digital signatures, cryptographic hash functions, and digital certificates.

## Exercises

### Discussion Questions

1. Use an existing code solution you've developed and identify portions of the program design which correctly implements one or more of the 13 security design principles.
2. Use an existing software design you've developed and identify portions of it which correctly implements one or more of the 13 security design principles.
3. Use an existing software design you've developed and identify the design changes you would need to make to adhere to applicable security design principles not currently in the design.
4. Use an existing software design you've developed and identify the design changes you would need to make to adhere to the security design principles not correctly implemented.

### Hands-on Exercises

1. Modify an existing code solution you've developed by implementing one or more of the 13 security design principles.
2. Modify an existing software design you've developed by adding one or more of the 13 security design principles.
3. Modify an existing software design you've developed by adding additional security design principles not correctly implemented or missing from the design.

## References

1. Saltzer JH, Schroeder MD (1975) The Protection of Information in Computer Systems. Proc IEEE 63(9)
2. McGraw G (2013) Thirteen principles to ensure enterprise system security. Available via https://searchsecurity.techtarget.com/opinion/Thirteen-principles-to-ensure-enterprise-system-security. Cited 28 July 2015