OOD Case Study: More Security Requirements

25

The objective of this chapter is to apply the security design principles discussed in Chap. 24 to the development of the case studies.

25.1 OOD: Preconditions

The following should be true prior to starting this chapter.

- You have been introduced to the 13 security design principles that researchers have identified as critical to thinking about and including when developing a software solution. These principles are economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, psychological acceptability, secure the weakest link, defend in depth, be reluctant to trust, promote privacy, and use your resources.
- 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 describe an object-oriented software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented 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_25

25.2 OOD: ABA Security Design

We will add some requirements to our address book case study to demonstrate application of some of the security design principles described in the previous chapter. The new requirements are in *italics*.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person, a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Allow for display of contact data in the address book.
8. *When the ABA starts, the user shall create a password. The password must contain at least eight characters with at least one uppercase letter, one lowercase letter, one digit, and one special character from the list ".!?@#$%&*_+−=".*
9. *Each request to create or display contact data shall first require the user to reenter their password. Three incorrect entries of the user's password shall result in the ABA exiting.*
10. *The data shall be non-persistently stored in encrypted form. A symmetric cryptographic algorithm shall be used.*

Table 25.1 lists the 13 security design principles, maps these to the above requirements when applicable, and describes the design approach for the principle.

25.2.1 OOD: ABA Design Models

The statechart in Fig. 25.1 shows the user actions for this version of the ABA. First, the user enters a password to control access to the ABA data. This password is entered via a GUI dialog shown in Fig. 25.2, then validated against the password rules. Once the password is valid, a text-based menu is displayed to allow the user to create a new contact, display an existing contact, or exit the ABA. When the user chooses to create or display, the user must reenter their password before being allowed to complete their requested transaction. Figure 25.3 shows what this dialog window looks like. When the password is verified, the ABA continues by either prompting the user for the three data values (see the sub-machine diagram in Fig. 25.4 for Get Contact Data) or prompting the user for the contact name whose data should be displayed. When the user has a third bad attempt on verifying their password, the ABA will exit.

The high-level IDEF0 function model in Fig. 25.5 shows the various security controls and mechanisms used by this version of the ABA. Since the model component is responsible for storing the password and contact data, this component contains logic

Table 25.1 Map security design principles to requirements and design

Principle	Requirement	Design approach
Economy of mechanism		We want to develop design models which express enough details to be meaningful but also be simple to read and understand. Our design models cannot be too abstract as this would produce a design that would not show the security features
Fail-safe defaults	9	Keep design for creating and verifying a user's password separate from entry and display of contact data. In the design described below, a graphical user interface (GUI) is used to allow entry of the password while a text-based user interface (TUI) is used to obtain and display contact data
Complete mediation	9	This requirement exemplifies this principle; the user must enter a correct password before being allowed to create or display contact data
Open design		While the design is not open source, it is available to anyone with access to this book
Separation of privilege		No requirement identifies a need for this and the ABA design will not address this principle. An example of a security feature for this principle is the use of two-factor authentication
Least privilege	9	The design for requirement 9 should ensure that each request for data has been authorized and the password created by the user is only valid for the address book data created during a single application instance (since the data is non-persistently stored)
Least common mechanism	8 & 10	The design for creating and verifying a password will use the SHA-256 hash function, where SHA stands for Secure Hash Algorithm. The design for encrypting contact data will use the AES (Advanced Encryption Standard) symmetric key algorithm. The design of these two security mechanisms has no connection with each other
Psychological acceptability	8 & 9	Entry of a password uses a GUI user control to display each character entered as an asterisk. This is consistent with many applications using an asterisk to mask the input of each character entered as part of a password value
Secure the weakest link		The secret key generated and used to encrypt and decrypt contact data is stored in memory within the model component. A malicious user may be able to scan memory to find this key value, allowing this individual to decrypt and view/change contact data
Defend in depth	8, 9 & 10	Creating a valid password must be done before the ABA is used. Entering the same password must be done prior to creating or displaying contact data. Finally, the model component stores the phone number and email address in encrypted form. Only when the user requests display of this data is it decrypted
Be reluctant to trust	8 & 9	The requirement and design for creating a password will make it more difficult for someone to guess a password. The requirement and design for allowing only three invalid attempts at verifying a password ensures a malicious user cannot guess an unlimited number of times
Promote privacy	10	The design encrypts the phone number and email address for each contact person
Use your resources		Not applicable

for the SHA-256 hash function and AES symmetric encryption algorithm. However, the password and contact domain data are validated by the controller before being given to the model for storage. The view component also shows the use of a GUI dialog to obtain a password and text-based menus and prompts to obtain the user's request and associated data.

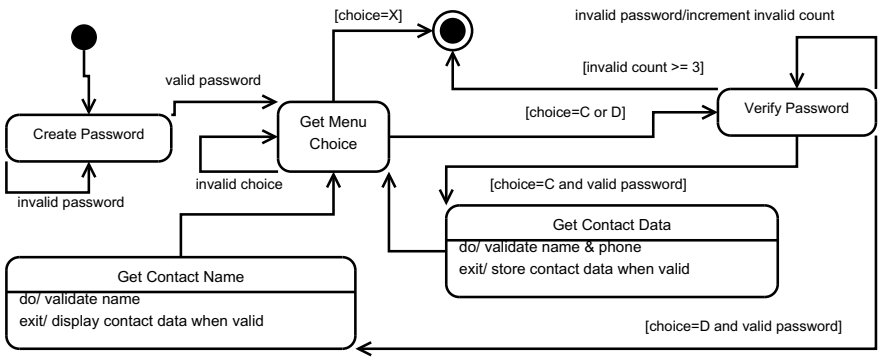


Fig. 25.1 ABA security design: statechart

Fig. 25.2 ABA security design: create password dialog

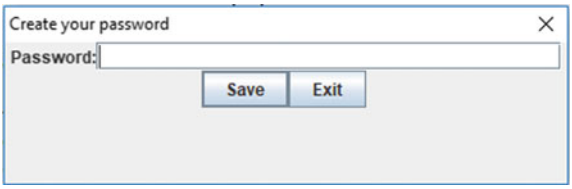
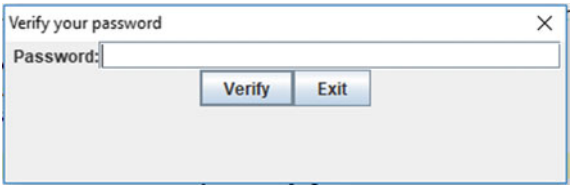


Fig. 25.3 ABA security design: verify password dialog



The package diagram in Fig. 25.6 shows the classes and relationship between the model, view, and controller components. The classes responsible for implementing security controls and mechanisms include the following.

ABA_A_viewPassword: Contains the plaintext password entered by the user via the GUI dialog. The model class is dependent on this class; this relationship is not shown on the package diagram.

ABA_A_viewGUI_Password: Represents the logic associated with the two password dialogs shown in Figs. 25.2 and 25.3.

ABA_A_modelCipherData: The model now stores instances of this object in a TreeMap. This object contains the contact name in plaintext and the phone and email in cipher(encrypted)text.

ABA_A_modelCrypto: Contains logic for the AES symmetric key algorithm.

ABA_A_modelGenKey: Generates a key used by the AES symmetric algorithm.

ABA_A_modelPassword: Generates and stores a message digest representing a valid password created by the user.

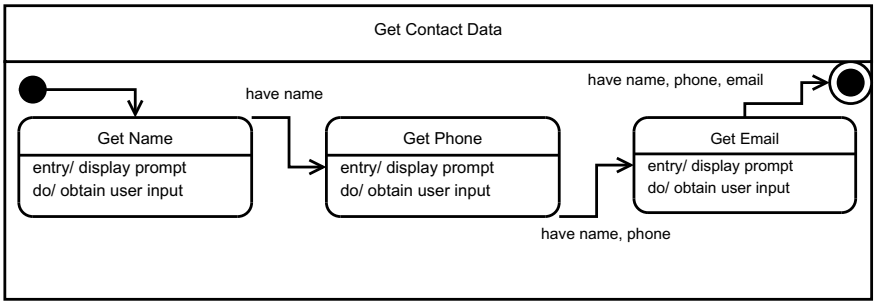


Fig. 25.4 ABA security design: sub-machine statechart for Get Contact Data

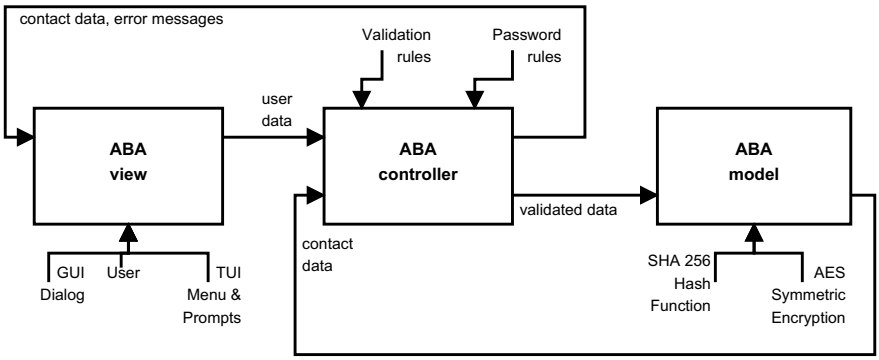


Fig. 25.5 ABA security design: IDEF0 function model

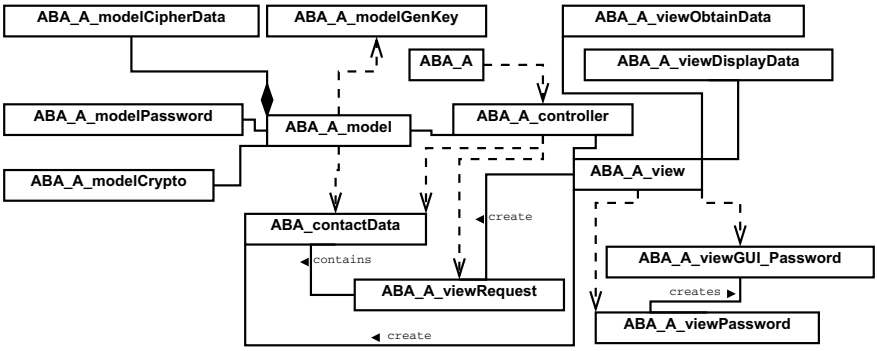


Fig. 25.6 ABA security design: package diagram

25.2.1.1 ABA Detailed Design and Implementation

This section provides more details on the design and implementation of the security controls and mechanisms used in this version of the ABA. We'll start by describing the changes to the view to support entry of a password. Figure 25.7 shows the classes in the view component. This class diagram shows attributes but hides methods for

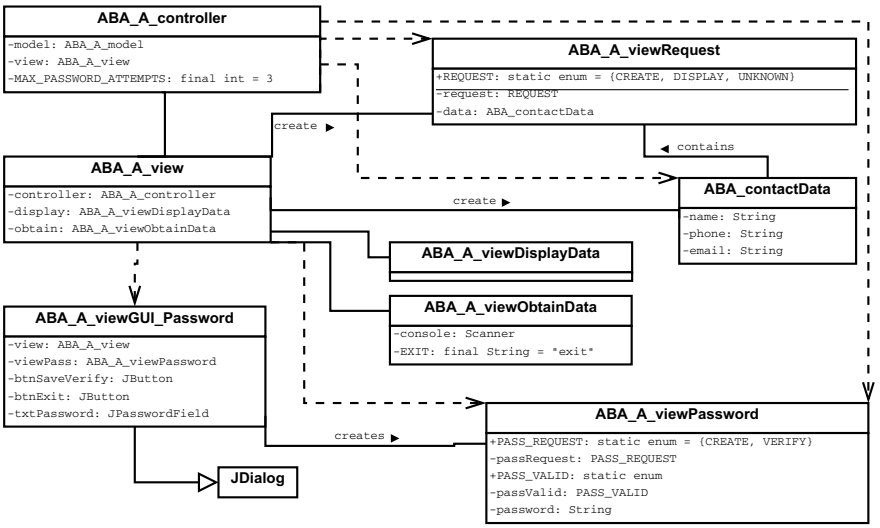


Fig. 25.7 ABA security design: class diagram view component

each class. The ABA_A_viewGUI_Password class inherits from the JDialog class, which is part of the Java swing package. This class uses a JPasswordField to hide the actual text being entered as a password. The ABA_A_viewPassword class acts as a container object passed to the controller for validation when creating a password, or for verification when verifying a password prior to the user creating or displaying contact data.

Listing 25.1 shows the creation of a JPasswordField user control. Note the method call setEchoChar('*'). This tells the control to display an asterisk each time the user enters a character.

Listing 25.1 ABA MVC Design version A - create JPasswordField

```
private JPanel createPasswordControls() {
  //Create JPanel that will contain the labels and text fields
  JPanel panel = new JPanel();
  panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

  txtPassword = new JPasswordField(TXT_WIDTH);
  txtPassword.setEchoChar('*');

  Dimension dim = txtPassword.getPreferredSize();
  txtPassword.setMaximumSize(dim);

  panel.add(createLblTxtPair(new JLabel(LBL_PASSWORD),
     txtPassword));

  return panel;
}
```

Two additional items are worth mentioning at this point. First, the ABA is a stand-alone application, wholly contained and executed on a single device. The ABA_A_viewPassword class contains the actual String value entered by the user for the password. An ABA_A_viewPassword object is created by the view and given to the controller for processing. Assuming that the user is creating their password, the controller validates it to ensure it adheres to the password rules (i.e., see requirement 8 above). When the password is valid, the controller gives the ABA_A_viewPassword object to the model. Only at this point is the plaintext password changed to a message digest. If the ABA were a distributed application, i.e., the view component runs on a client device and the model component runs on a server, then this design solution would be insufficient since the plaintext of the password would be traveling across a network from client to server. In this case, the client-side ABA application would contain the view and a partial controller implementation. The controller would validate the password (when it is being created), and then apply the hash function to produce a message digest. The message digest would be sent over the network to the server for storage.

Second, the SHA-256 hash function computes a message digest quickly. This can result in a successful dictionary attack if the password is made from a common word or phrase. In the case of the ABA, requirement 8 (*password must contain at least eight characters with at least one uppercase letter, one lowercase letter, one digit, and one special character*) makes a dictionary attack very unlikely to succeed. Regarding the speed of the SHA-256 algorithm [1], this processing starts by splitting the data to be hashed into 512-bit blocks. A password is likely going to contain fewer than 64 characters, resulting in the password being completely contained in one 512-bit block. The algorithm would then iterate 64 times to compute eight intermediate 32-bit values, which are then concatenated to produce the 256-bit message digest.

The class diagram for the model component is shown in Fig. 25.8. This class diagram shows attributes but hides methods for each class. The ABA_A_modelPassword class is used to transform the plaintext password value into a message digest. The private computeHash method, shown in Listing 25.2, shows the use of the Java MessageDigest class to create an instance associated with the "SHA-256" algorithm. This instance is then used to compute a 256-bit message digest using a `byte[]` representation of the plaintext password value.

Listing 25.2 ABA MVC Design version A - compute message digest

```
private byte[] computeHash(String pass)
{
  byte[] computedHashDigest = null;
  try
  {
    byte[] data = pass.getBytes();
    //use the SHA-256 algorithm
    MessageDigest md = MessageDigest.getInstance(SHA256);
    //Give hash function the byte data
    md.update(data);
    //Compute the hash digest
    computedHashDigest = md.digest();
```

```
    }
    catch (NoSuchAlgorithmException ex)
    {
      System.out.println("ABA_A_modelPassword: no such algorithm " +
          "exception in computeHash.");
      System.out.println(ex);
    }
    catch (Exception ex)
    {
      System.out.println("ABA_A_modelPassword: exception in " +
          "computeHash.");
      System.out.println(ex);
    }
    return computedHashDigest;
}
```

As noted above, the encryption and decryption of contact data is being done in the model component. The ABA_A_modelGenKey constructor method is used to generate a secret key used by the AES algorithm. Listing 25.3 shows the use of the KeyGenerator and SecureRandom Java API classes. The key variable is the only instance variable in this class. Not shown in the model class diagram (see Fig. 25.8), the public getKey method is called to obtain the SecretKey object.

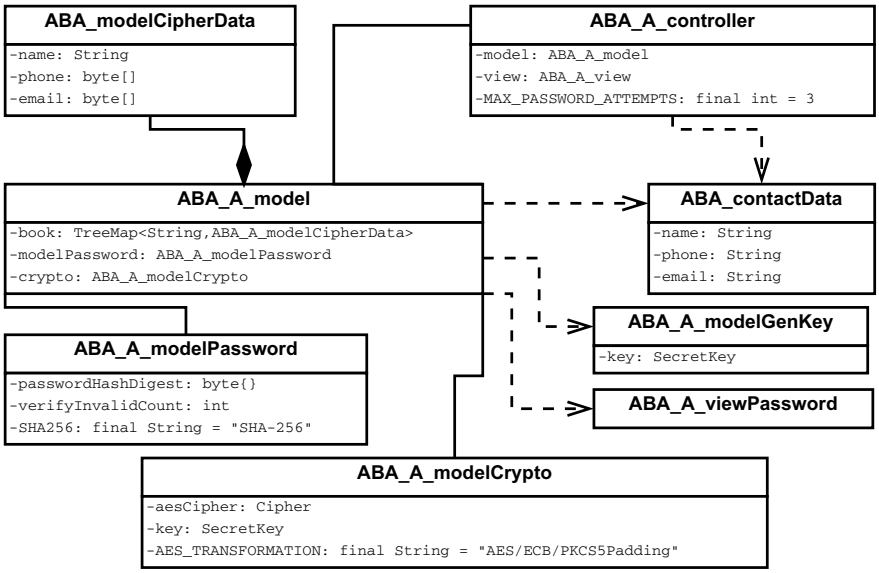


Fig. 25.8 ABA security design: class diagram model component

Listing 25.3 ABA MVC Design version A - generate secret key

```
public ABA_A_modelGenKey()
{
  final String AES = "AES";
  final String DSA_ALGORITHM_NAME = "SHA1PRNG";
  final String DSA_PROVIDER_NAME = "SUN";
  final byte[] MY_SEED_VALUE =
      "The ABA is a small but useful case study!".getBytes();

  try
  {
    //Get a key generator object for the AES algorithm
    KeyGenerator keyGen = KeyGenerator.getInstance(AES);
    //Do an algorithm-independent initialization of key generator
    SecureRandom random = SecureRandom.getInstance(
      DSA_ALGORITHM_NAME, DSA_PROVIDER_NAME);
    //Include a user supplied seed as part of this randomness.
    random.setSeed(MY_SEED_VALUE);
    //Initialize the key generator.
    keyGen.init(random);
    //Generate a secret key
    key = keyGen.generateKey();
  }
  catch (Exception ex)
  {
    System.out.println("demoSymK_Generate_Key: " + ex);
    key = null;
  }
}
```

The model constructor method in Listing 25.4 shows how an ABA_A_modelGen Key object is constructed and then used to obtain the secret key, which is used to construct an ABA_A_modelCrypto object. The ABA_A_modelCrypto object is then used to encrypt and decrypt the phone number and email address for contact. The encrypt method is shown in Listing 25.5. Also note in Listing 25.4, the TreeMap data structure now stores an ABA_A_modelCipherData object as the associated data value for a contact name, which is stored as a String object. As shown in the class diagram in Fig. 25.8, the phone and email instance variables in the ABA_A_modelCipherData are a byte[] data type, which is the type of data returned by the encrypt method.

Listing 25.4 ABA MVC Design version A - model constructor method

```
public ABA_A_model()
{
  this.book = new TreeMap<String,ABA_A_modelCipherData>();
  this.modelPassword = new ABA_A_modelPassword();
  //Create an AES symmetric key.
  ABA_A_modelGenKey genKey = new ABA_A_modelGenKey();
  //Create crypto object for AES..
  this.crypto = new ABA_A_modelCrypto(genKey.getKey());
}
```

Listing 25.5 ABA MVC Design version A - encrypt method

```
public byte[] encrypt(byte[] plaintext)
{
  byte[] ciphertext = null;
  try
  {
    //initialize cipher for encryption.
    aesCipher.init(Cipher.ENCRYPT_MODE, key);
    //encrypt plaintext.
    ciphertext = aesCipher.doFinal(plaintext);
  }
  catch (Exception ex)
  {
    System.out.println("ABA_A_modelCrypto.encrypt exception:");
    System.out.println(ex);
  }
  return ciphertext;
}
```

25.3 OOD Top-Down Design Perspective

We'll use the personal finances case study to reinforce design choices when thinking about security design principles as part of a top-down design approach.

25.3.1 OOD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 12, are listed below. Note the complete absence of security requirements.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.

- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

25.3.2 OOD Personal Finances: Security Design

Since this case study is storing sensitive financial information, we would expect the data to be persistently stored as ciphertext (i.e., in encrypted form). Likewise, we may want the user to provide evidence they are who they say they are. At the very least a pin number or password should be required. If we are concerned about the ease in which someone can guess the user's pin or password, we would design and implement rules regarding the length and content of a pin or password. We can also enforce a second factor of authentication, like sending a code to a cell phone and expecting the user to enter this code before they gain access to the personal finances application. These security controls and mechanisms may be sufficient when personal finances is a stand-alone application.

The security description above addresses the following security design principles.

Fail-safe defaults: Requiring a pin or password to gain access.
Separation of privilege: Use of a second factor for authentication.
Psychological acceptability (make security usable): We'll assume that entry of a pin or password is masked.
Promote privacy: Encrypting all of the data persistently stored.

If we assume that the personal finances application utilizes a cloud service for persistent storage of the data, this adds another layer of security concerns. In this case, the personal finances application is split between a client device (running a view component along with part of the controller) and a server device (running part of the controller and the model component). Our design should now include the encryption of data traveling between the client and server devices. This encryption should use a different algorithm, or at least a different key, than what is being used to encrypt the data for persistent storage. One approach would be to use the HTTPS protocol handshaking to establish a secret key between web browser and web server. This secret key would then be used during the duration of the current user session. Two benefits arise from this design solution. First, the handshaking between client and server within HTTPS is a well-established and tested protocol. It has shown remarkable resiliency to attack. Second, this guarantees that the secret key used for encrypting the data while in motion would be different from a secret key used to persistently store the data. This guarantee exists since the HTTPS handshaking generates a secret key each time a secured connection is established, while the secret key used for persistent storage would likely never change.

The security description for a distributed version of the personal finances application addresses the following additional security design principles.

Least common mechanism: Use of one secret key to encrypt data in motion while using a different secret key to encrypt data at rest.
Secure the weakest link: Encrypting data in motion and at rest tends to be overlooked in many of today's systems.
Defend in depth: Using encryption in different parts of the system.

Below are a few design ideas for some of the remaining security design principles.

Complete mediation: While it would be annoying to most users to always force entry of their pin/password before they can do something in the personal finances application, it may be appropriate to develop a compromise. The design could include a timer to compute the idle time between user actions. When the user is idle for more than N minutes, the software could force the user to reenter their pin/password. This idle detection mechanism could also use two-factor authentication.
Least privilege: Since the personal finances application is used by an individual, or perhaps by a few people using the same financial accounts, there is no reason to distinguish different types of users and their privilege levels.
Be reluctant to trust: If we assume the cloud service being used provides encryption services, this should be periodically tested by the team responsible for the personal finances application. If trust in the cloud provider cannot be verified, it's more secure to (redundantly) build encryption of persistent data into your design.

25.4 OOD: Post-conditions

The following should have been learned when completing this chapter.

- You understand how many of the 13 security design principles were applied to the case study software design.
- You've applied many of the 13 security design principles to a software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented 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.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a UML class diagram, UML package diagram, IDEF0 function model, and data-flow diagram are design models used in object-oriented solutions to illustrate the structure of your software design.

- You understand that an IDEF0 function model, data-flow diagram, UML communication diagram, and UML statechart are design models used to illustrate the behavior of your software design.
- You have created and/or modified models that describe an object-oriented software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Discussion Questions

1. The HTTPS protocol uses an asymmetric algorithm to establish a secret key between the browser and server. Both the client and server then use the same secret key (i.e., using a symmetric algorithm) to encrypt and decrypt the data in motion. Why does the HTTPS protocol do this? Why not simply use an asymmetric algorithm to encrypt and decrypt the data in motion?
2. In the description of *be reluctant to trust* for the personal finances application, why should the team responsible for the application *periodically* verify the encryption being used by the cloud service? Why not verify this once and be done with it?
3. Can you identify a software application that implements *complete mediation*?
4. Describe a type of software application where *least privilege* would be important to design and implement.
5. The argument for having an *open design* is supported by the phrase *two heads are better than one*. Having many people looking at your design would likely result in finding more defects and vulnerabilities. Can you think of a scenario where *open design* would result in more security risks, not less?

Hands-on Exercises

1. Modify the design and code for the ABA.

    a. Remove the requirement that a user must enter their password each time they request to create or display contact data. Replace this with a design and implementation of the *idle detection mechanism* briefly described for the personal finances application.

2. Modify the design for the personal finances application.

a. Update the class diagrams from Chap. 15 to show how the security controls and mechanisms described in this chapter may be included in the personal finances object-oriented design.

3. Using an existing code solution you've developed, develop alternative security design models to show ways in which security design principles could be included in your application.
4. Use your development of an application you started in Chap. 3 for this exercise. Modify your design to use some security controls, and then evaluate your design using the 13 security design principles.
5. Continue Hands-on Exercise 3 from Chap. 12 by developing a design that satisfies many of the 13 security design principles. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 12 for details on each domain.

- Airline reservation and seat assignment
- Automated teller machine (ATM)
- Bus transportation system
- Course-class enrollment
- Digital library
- Inventory and distribution control
- Online retail shopping cart
- Personal calendar
- Travel itinerary

Reference

1. National Institute of Standards and Technology: Secure Hash Standard (SHS). NIST (2015) https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. Accessed June 26 2019