****

# Practices for Secure Software Report

Table of Contents

[Document Revision History 3](#_Toc102040754)

[Client 3](#_Toc102040755)

[Instructions 3](#_Toc102040756)

[Developer 4](#_Toc102040757)

[1. Algorithm Cipher 4](#_Toc102040758)

[2. Certificate Generation 4](#_Toc102040759)

[3. Deploy Cipher 5](#_Toc102040760)

[4. Secure Communications 5](#_Toc102040761)

[5. Secondary Testing 6](#_Toc102040762)

[6. Functional Testing 7](#_Toc102040763)

[7. Summary 8](#_Toc102040764)

[8. Industry Standard Best Practices 9](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **8/14/2024** | **Armon Wilson** |  |

## Client



## Instructions

Submit this completed practices for secure software report. Replace the bracketed text with the relevant information. You must document your process for writing secure communications and refactoring code that complies with software security testing protocols.

* Respond to the steps outlined below and include your findings.
* Respond using your own words. You may also choose to include images or supporting materials. If you include them, make certain to insert them in all the relevant locations in the document.
* Refer to the Project Two Guidelines and Rubric for more detailed instructions about each section of the template.

## Developer

Armon Wilson

## Algorithm Cipher

The recommended solution is to use the SHA-256 cryptographic hash function. SHA-256 is a widely used and trusted method that takes any input and produces a fixed-size 256-bit hash value. This hash acts like a unique fingerprint for the data. Key features of SHA-256 include: it always produces the same hash for the same input, it's very difficult to find two different inputs that produce the same hash, it's very difficult to find the original input from the hash, and even a small change in the input causes a big change in the hash.

SHA-256 works by processing the input data in blocks and applying mathematical operations to create the final hash. The 256-bit output provides strong security against attacks. It's important to note that SHA-256 is a hash function, not an encryption algorithm. It doesn't use keys or random numbers; its job is to create a unique identifier for the data.

Encryption methods have evolved, with older ones like MD5 and SHA-1 becoming less secure. SHA-256 is part of the SHA-2 family and is considered secure. It's widely used for things like digital signatures, checking file integrity, and storing passwords.

SHA-256 is a good choice for Artemis Financial because it's strong, established, and provides high assurance of data integrity. Its 256-bit output makes it resistant to attacks, ensuring any data changes during transfer are detected. It's also widely supported and easy to implement.

While other hash functions might offer slightly better performance or security, SHA-256 balances security, performance, and ease of use well, making it suitable for Artemis Financial's needs.

## Certificate Generation

Insert a screenshot below of the CER file.

A computer screen with white text

Description automatically generated

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screenshot of a computer

Description automatically generated

## Secure Communications

Insert a screenshot below of the web browser that shows a secure webpage.

A screenshot of a computer

Description automatically generated

## Secondary Testing

Insert screenshots below of the refactored code executed without errors and the dependency-check report.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer screen

Description automatically generated

## Functional Testing

Insert a screenshot below of the refactored code executed without errors.

A screenshot of a computer program

Description automatically generated

## Summary

The code refactoring efforts significantly enhanced the application's security posture by addressing key vulnerabilities and aligning with industry best practices. The primary focus was on ensuring data integrity and establishing secure communication channels.

The implementation of the SHA-256 cryptographic hash algorithm provides a mechanism for verifying data integrity during transmission. By generating a unique checksum for the data, any unauthorized modifications or corruption during transit can be readily detected. The transition from HTTP to HTTPS, achieved through SSL/TLS configuration and keystore generation, makes certain that all communication between the client and server is encrypted, protecting sensitive information from eavesdropping and unauthorized access.

The refactoring process also included a thorough code review, which helped identify and address potential vulnerabilities such as inadequate error handling. Additionally, the use of the dependency-check tool played a role in identifying and mitigating vulnerabilities within third-party libraries, further strengthening the application's overall security.

* **Cryptography**: The core of the refactoring involved implementing the SHA-256 cryptographic hash algorithm to generate checksums. This directly addresses the "Cryptography: Encryption Use and Vulnerabilities" aspect of the diagram, ensuring data integrity during transmission.
* **Input Validation**: Although not explicitly implemented in the provided code, the discussion around input validation emphasizes its importance in preventing security issues like injection attacks. This aligns with the "Input Validation: Secure Input and Representations" area of the diagram.
* **Client/Server**: The conversion from HTTP to HTTPS, by configuring SSL/TLS and generating a keystore, directly addresses the "Client/Server: Secure Distributed Composing" aspect. This ensures that data transmitted between the client and server is encrypted and protected.
* **Code Quality**: The refactoring process also touched upon the "Code Quality: Secure Coding Practices/Patterns" area. The code review helped identify potential vulnerabilities.

The code has been refactored to incorporate security enhancements that address potential vulnerabilities and align with industry best practices. The primary focus was on data integrity and secure communication. The implementation of the SHA-256 cryptographic hash algorithm ensures that data transmitted between the client and server maintains its integrity, safeguarding against unauthorized modifications or corruption. The transition from HTTP to HTTPS, achieved through SSL/TLS configuration and keystore generation, encrypts data in transit, protecting it from eavesdropping and unauthorized access.

The refactoring process also involved a code review, which helped identify and address potential vulnerabilities such as inadequate error handling. The use of the dependency-check tool further strengthened the application's security by identifying and mitigating vulnerabilities in third-party libraries.

## Industry Standard Best Practices

The refactoring efforts in this project significantly enhanced the application's security by addressing vulnerabilities and aligning the code with industry best practices. The initial codebase lacked any explicit security measures for data integrity or secure communication. The primary vulnerability was the potential for data tampering or corruption during transmission, especially concerning sensitive client data and financial information. The application also relied on the insecure HTTP protocol, exposing data to eavesdropping and unauthorized access.

The refactoring process focused on two areas:

* **Data Integrity**: The introduction of the SHA-256 cryptographic hash algorithm directly addressed the vulnerability of data integrity during transmission. By generating a unique checksum for the data, any modifications or corruption can be easily detected, ensuring that the received data is identical to the data sent. This aligns with the "Cryptography: Encryption Use and Vulnerabilities" aspect of the Vulnerability Assessment Process Flow diagram.
* **Secure Communication**: The conversion from HTTP to HTTPS, achieved through SSL/TLS configuration and keystore generation, mitigated the risk of unauthorized access and eavesdropping by encrypting all data transmitted between the client and server. This directly corresponds to the "Client/Server: Secure Distributed Composing" aspect of the diagram.

In addition, the refactoring process also involved a code review that identified and addressed potential vulnerabilities related to error handling. While my original code included basic error handling, it lacked robustness and could potentially expose sensitive information in error messages. The refactored code includes improved error handling to mitigate this risk, aligning with the "Code Quality: Secure Coding Practices/Patterns" aspect of the diagram.

The use of the dependency-check-maven plugin further contributed to the application's security by identifying and addressing vulnerabilities in third-party libraries. This approach to dependency management helps assertion that the application's foundation remains secure and free from known vulnerabilities.

By implementing industry-standard best practices for secure coding, the project successfully mitigated known security risks and enhanced the overall protection of sensitive data. The layered approach to security, combining data integrity checks, secure communication protocols, and improved error handling, demonstrates a commitment to safeguarding user information and maintaining the confidentiality and integrity of data within the application.