****

# 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 7](#_Toc102040764)

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

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **20 June 2025** | **Chase Knoblock** |  |

## 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

Chase Knoblock

## Algorithm Cipher

Encryption has been around for thousands of years with examples like the Caesar Cipher and the Enigma Machine. More recently, in the 1970s, DES (Data Encryption Standard) was adopted as one of the first modern encryption algorithms. The RSA algorithm was created around the same time. DES was later replaced by AES (Advanced Encryption Standard) and that has remained one of the most widely used symmetric encryption standards. Today, encryption is a foundational element for cybersecurity. Symmetric algorithms like AES, Asymmetric algorithms like RSA, and hash functions like SHA (introduced in 2001) are all very common and have proven sturdy against cyber-attacks.

For this project, I used SHA-256, developed by the National Security Agency and published by NIST. SHA-256 is a cryptographic hash function used for data integrity verification. Hashing converts. Input data into a fix-length string of characters that cannot be reversed or decrypted. This makes it perfect for verifying content with revealing the original data. SHA-256 will produce a 256-bit hash value that is displayed as a 64-character hexadecimal string. The “256” refers to the length of the hash output, determining collision resistance, which is how hard it is to find two different inputs that produce the same output.

SHA-256 does not use keys or random numbers. For example, AES uses symmetric encryption meaning it uses the same key for encryption and decryption. RSA uses asymmetric encryption meaning it uses a public/private key pair. There is also random number generation which is important for key generation and salting data before hashing. However, SHA-256 is a deterministic cryptographic hash function meaning the same input will always produce the same output and does not use randomness, secret keys or any hidden state.

## Certificate Generation

Insert a screenshot below of the CER file.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screenshot of a phone

AI-generated content may be incorrect.

## Secure Communications

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

A screenshot of a computer

AI-generated content may be incorrect.

## Secondary Testing

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

**Before Refactoring:**

A screen shot of a computer

AI-generated content may be incorrect.

A close-up of a screen

AI-generated content may be incorrect.

**After Refactoring:**

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer error

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

The build was successful after refactoring and no new vulnerabilities were found after introducing new code.

## Functional Testing

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

After manually reviewing the refactored code, the code runs with no syntax errors or warnings. All methods seem to compile and run correctly. The checksum returns what is expected and only the checksum is output, nothing else. The inputs are hardcoded for project and learning purposes, but a dynamic input validation could be used. HTTPS is enabled and a strong cipher is being used, SHA-256.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Summary

For this project, I refactored the original application to comply with modern software security practices. I focused on a few areas of security through code refactoring including data protection, secure communication, certificate-based encryption, and static vulnerability analysis.

For data protection, a cryptographic hash function using the SHA-256 algorithm was used to generate a checksum from a static input string. This ensures data integrity and demonstrates cryptographic functionality. For secure communication, the application was converted from HTTP to HTTPS by configuring the Spring Boot server to use TLS encryption. I utilized a self-signed SSL certificate, generated and loaded it using Java Keystore(.p12) to establish encrypted communications by the port 8443. The certificate was exported as a .cer file to prove the encryption source and satisfies the requirements for secure certificate generation and deployment. I used the OWASP Dependency-Check Mavin plugin for static security testing, conducting a static analysis for the project’s dependencies. The report confirmed that no new vulnerabilities were introduced with the refactored code. I also verified that the application compiled and executed successfully after manually reviewing the code for syntax or logic errors.

I used a layered security strategy refactoring the code. I enforced HTTPS to secure the data in transit. I added the hash algorithm to validate the data integrity. The SSL certificate allowed trust. The application.properties file was updated to securely manage the keystore access. The dependency analysis was used to identify any vulnerable third-party libraries.

## Industry Standard Best Practices

While refactoring the code, I did my best to adhere to industry standard best practices to maintain and enhance the applications existing security. For example, I used HTTPS with SSL, refactoring the server to run over HTTPS using the self-signed certificate loaded via Java Keystore, protecting data in transit. I applied SHA-256 to generate the checksum, ensuring data integrity. I used the application.properties to store sensitive properties, isolating configuration from code to reduce its exposure. The OWASP Dependency-Check plugin that was integrated was used to statically analyze project dependencies. And of course, I manually reviewed the existing and refactored code for syntax and logical errors.

Applying secure coding best practices will help protect the integrity of the application and will contribute to the reputation of Artemis Financial. By following security guidelines that have proven to be useful minimizes the risk of data breaches, legal exposure and allows clients to trust the company. Adhering to industry standards also helps align with industry regulations, something that is highly important for financial services. Writing clean and secure code makes it easier to maintain and update, reducing technical debt and enhances the lifecycle management.

Resources

Buchmann, J. (2019). Introduction to cryptography (3rd ed.). Gesellschaft für Informatik e.V.

<https://dl.gi.de/server/api/core/bitstreams/37f9e22b-a8e9-4d01-98a2-40463cd0f1fb/content>

Moskowitz, C. (2024, May 24). Hans Peter Luhn and the birth of the hashing algorithm. IEEE

Spectrum. <https://spectrum.ieee.org/hans-peter-luhn-and-the-birth-of-the-hashing-algorithm>