****

# 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 4](#_Toc102040760)

[4. Secure Communications 4](#_Toc102040761)

[5. Secondary Testing 4](#_Toc102040762)

[6. Functional Testing 4](#_Toc102040763)

[7. Summary 4](#_Toc102040764)

[8. Industry Standard Best Practices 4](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **6/18/2025** | **Kileigh Adams** |  |

## 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

Kileigh Adams

## Algorithm Cipher

SHA-256 is listed in Oracle’s official Java security documentation as a supported algorithm for the MessageDigest class, which means it’s already built into Java and follows the platform’s security standards (Oracle, 2017). It’s a 256-bit hash function that doesn’t use keys, unlike symmetric or asymmetric encryption methods. Since it’s one-way and deterministic, it’s perfect for checking whether data has been altered without needing to decrypt anything. Older hash functions like MD5 and SHA-1 were used for this in the past, but they’ve been phased out due to known vulnerabilities. SHA-256 is still considered secure and is used in everything from SSL certificates to blockchain, so it’s a solid fit for a financial application that needs reliable protection.

## Certificate Generation

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a certificate

AI-generated content may be incorrect.

## Deploy Cipher

A screenshot of a computer

AI-generated content may be incorrect.

## Secure Communications

A screenshot of a computer

AI-generated content may be incorrect.

## Secondary Testing

* Maven build success

A screenshot of a computer program

AI-generated content may be incorrect.

* Original code scanned for dependencies:

A screenshot of a computer

AI-generated content may be incorrect.

* Refactored code scanned for dependencies:

A screenshot of a computer

AI-generated content may be incorrect.

To make sure my refactored code didn’t introduce new security risks, I ran a dependency check both before and after making changes. The baseline scan of the original code (version 5.3.0 of the plugin) found 221 known vulnerabilities across 21 vulnerable dependencies. After implementing the SHA-256 hashing route and updating the application to use HTTPS, I ran the check again using version 8.4.0. The new scan reported 176 vulnerabilities across 20 vulnerable dependencies. Since I didn’t add any new external libraries and only used built-in Java security features and Spring Boot configurations, this confirms that no new vulnerabilities were introduced as a result of the refactoring. If anything, the decrease in total vulnerabilities reflects improved dependency versions or configuration settings in the updated scan.

## Functional Testing

A screen shot of a computer

AI-generated content may be incorrect.

## Summary

To strengthen Artemis Financial’s web application, I refactored the code to include a checksum verification feature and added support for secure HTTPS communication. These updates address multiple areas outlined in the Vulnerability Assessment Process Flow Diagram, including data integrity, secure communication, and secure configuration. I created a new /hash route that returns a SHA-256 hash of a fixed string using Java’s MessageDigest class. I also generated a self-signed certificate using the Java Keytool and configured the application to run over HTTPS by updating the application.properties file.

In order to remove the browser’s “Not Secure” warning, I exported the .cer file and added it to my system’s trusted root certification authorities. This allowed the browser to recognize the certificate as trusted, which confirmed that encryption was properly configured during testing. Functionally, the updated application launched and returned the expected checksum through a secure HTTPS connection. For static testing, I ran OWASP Dependency-Check before and after making changes. The original project had 221 known vulnerabilities. After refactoring, the number dropped to 176. Since I did not introduce any new libraries, the results show that no new security issues were added.

## Industry Standard Best Practices

The updates I made align with secure software development practices commonly used in the industry, especially for web applications that handle sensitive client data. Artemis Financial requested a checksum feature and secure communication. For the checksum, I used SHA-256, which is still considered a reliable hash function for verifying data integrity. It's resistant to collision attacks and is widely used in security-critical systems. Since Artemis needed a way to confirm that transferred data wasn’t altered, not full encryption, SHA-256 was the right fit. It avoids unnecessary complexity while still meeting integrity requirements.

For secure transmission, I implemented HTTPS using a self-signed certificate. Although self-signed certificates are not used in production, they’re a standard choice for development and testing environments. This ensured data was encrypted in transit without requiring a third-party certificate authority. I also exported the certificate and added it to my system’s trusted root authorities to avoid browser security warnings, which reflects good development hygiene when testing HTTPS locally.

To confirm that my changes didn’t introduce new vulnerabilities, I ran OWASP Dependency-Check before and after refactoring. This tool is an industry-recognized way to monitor risks in third-party libraries. Since I kept the scope of changes focused on built-in Java security libraries and Spring Boot configurations, the risk of new vulnerabilities was low. Running static scans and avoiding new dependencies helped maintain the application's existing security posture while delivering the features Artemis needed.

**References**

Oracle. (2017). *Java Cryptography Architecture Standard Algorithm Name Documentation for JDK 9*. Oracle. <https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#cipher-algorithm-names>