****

# 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** | **6/19/25** | **Shamus Cerny** |  |

## 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

Shamus Cerny

## Algorithm Cipher

For securing Artemis Financials’ data transfers, I recommend using **AES-256** for encryption, paired with a **SHA-256** hash function for checksum-based data verification. AES (Advanced Encryption Standard) is a symmetric block cipher standardized by NIST, encrypting data in 128-bit blocks using key sizes of 128, 192, or 256 bits. **AES-256** offers the highest level of security and is trusted by institutions and governments for protecting sensitive information. It uses the same key for both encryption and decryption, making it ideal for internal systems and authenticated exchanges within Artemis Financials’ platform.

**SHA-256** is a cryptographic hash function that outputs a fixed 256-bit value, providing strong integrity checks and resistance to collisions or reversibility. When paired with AES-256, it ensures both **confidentiality** (via encryption) and **integrity** (via hashing). At the bit level, AES-256's 256-bit key makes brute-force attacks computationally infeasible, while SHA-256 provides a consistent and secure checksum for verifying data during transmission.

To support secure encryption, AES-256 must be implemented with **cryptographically secure random numbers** for generating unique initialization vectors (IVs) per encryption session. This guarantees that even identical plaintexts produce different ciphertexts each time. While AES is a **symmetric cipher**, requiring secure key exchange between parties, this can be handled safely using **asymmetric encryption**. In asymmetric cryptography, a public key is used to encrypt the AES key, and only the recipient’s private key can decrypt it. This hybrid approach combines the efficiency of AES with the secure key distribution of asymmetric algorithms like RSA or ECC.

AES was introduced in 2001 to replace the outdated and vulnerable DES cipher and remains unbroken when implemented correctly. Similarly, **SHA-256**, part of the SHA-2 family, was developed to address weaknesses in SHA-1 and has become a global standard for secure hashing. Today, AES-256 and SHA-256 are widely supported across programming languages, operating systems, and cryptographic libraries. They are used in critical industries such as finance, healthcare, and government, making them a strong, future-proof choice for Artemis Financials’ secure software infrastructure.

## Certificate Generation

Insert a screenshot below of the CER file.

A black screen with white text

AI-generated content may be incorrect.

A computer screen with white text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screen shot of a computer program

AI-generated content may be incorrect.

## Secure Communications

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

A screen shot 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.

No changes were required as there are no vulnerabilities related to the checksum endpoint. I do recommend updating **Spring Boot** and **Tomcat-embed-core**

A screenshot of a computer

AI-generated content may be incorrect. 

## Functional Testing

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

A screen shot of a computer program

AI-generated content may be incorrect.

## Summary

No major code changes were needed because the checksum feature I added was already secure. It uses Java’s built-in MessageDigest with the SHA-256 algorithm, which is a strong and modern choice for verifying data integrity. This helps make sure the data hasn’t been changed or tampered with.

I ran a dependency check to make sure nothing I added introduced security issues. The scan showed some known problems in Spring Boot and Tomcat, which are used by the project but not part of my custom code. To fix those, I recommended updating Spring Boot and Tomcat to newer versions.

The main security improvement from my code is that it adds a way to check if the data is valid using a secure checksum. This adds a layer of protection and follows good software security practices.

## Industry Standard Best Practices

To keep the software secure, I followed common best practices during development and testing. For the checksum feature, I used the SHA-256 hashing algorithm with Java’s built-in MessageDigest, which is a trusted and safe way to check data integrity. I avoided older, weaker algorithms like MD5 and SHA-1 to prevent common attacks.

I also practiced secure coding by not hardcoding secrets, limiting access to only needed endpoints, and keeping the code simple to lower the chance of bugs or vulnerabilities. After building the app, I ran a security scan with OWASP Dependency-Check to make sure no new issues were introduced. Based on the scan, I recommended updating Spring Boot and Tomcat to versions that fix known security problems, following the practice of keeping software up to date.

Using these secure coding practices helps protect the app from risks, builds customer trust, reduces chances of breaches, and avoids costly problems. It supports the company’s security and makes the software more stable and ready for future threats.