****

# 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** | **6/21/2025** | **Blake Springman** |  |

## 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

Blake Springman

## Algorithm Cipher

I recommend deploying the SHA-256 hashing algorithm in tandem with the AES (Advanced Encryption Standard) cipher. This dual approach ensures both integrity verification through hashing and confidentiality through encryption, strengthening the application's security profile.

SHA-256 is part of the SHA-2 family developed by the National Security Agency (NSA) and standardized by NIST. It produces a 256-bit (32-byte) fixed-length hash regardless of the input size, making it highly resistant to collisions and pre-image attacks. As a cryptographic hash function, SHA-256 is ideal for verifying data integrity through checksums since any tampering with the input data yields a drastically different hash value.

In terms of encryption, AES is the gold standard for symmetric encryption in modern applications. It operates on fixed block sizes of 128 bits with key sizes of 128, 192, or 256 bits. AES uses a substitution-permutation network and multiple rounds of transformation, making it both efficient and secure. Given that Artemis Financial processes sensitive financial data, AES-256 is strongly recommended for its high level of security.

Regarding key management, AES is a symmetric encryption cipher, meaning the same key is used for both encryption and decryption. This makes it faster and more computationally efficient than asymmetric ciphers (like RSA), though it requires careful handling of key distribution. In contrast, asymmetric algorithms rely on a key pair public and private keys for secure communication, but they typically serve best in key exchange and digital signatures rather than bulk data encryption.

Random number generation also plays a critical role in both hash-based operations and encryption. For AES, securely generated random initialization vectors (IVs) ensure ciphertext uniqueness, even when encrypting the same plaintext. SHA-256, while deterministic, benefits indirectly from randomization when used with salting techniques to protect against dictionary and rainbow table attacks.

Historically, cryptography began with simple ciphers like Caesar’s substitution cipher, evolving over centuries into complex mechanical systems (e.g., the Enigma machine in WWII). With the advent of computers, algorithms like DES and RSA emerged, ushering in the age of digital cryptography. DES was eventually replaced due to vulnerabilities, leading to the standardization of AES in 2001, which remains the dominant choice today. Likewise, SHA-1 was phased out due to collision vulnerabilities, giving way to the SHA-2 family. Today, algorithms like AES and SHA-256 represent the cryptographic backbone of secure digital systems, endorsed by government and industry standards alike.

By integrating SHA-256 for checksum validation and AES for secure data transmission, Global Rain can ensure that Artemis Financials software meets both modern security expectations and compliance standards safeguarding client data while maintaining high performance.

## Certificate Generation

Insert a screenshot below of the CER file.

A screenshot of a computer screen

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A white background with black text

AI-generated content may be incorrect.**I did everything I could to connect to the server but it is still refusing even after changing the server port a few times.**

## Secure Communications

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

A white page with black text

AI-generated content may be incorrect.**Still received the same error as I did in the last test.**

## Secondary Testing

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

**My refactored code does not introduce any additional insecure dependencies or critical CVE’s.**

A screenshot of a computer

AI-generated content may be incorrect.

A close-up of a computer code

AI-generated content may be incorrect.

## Functional Testing

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

A screenshot of a computer

AI-generated content may be incorrect.

## Summary

As part of the secure software development lifecycle, the application was refactored to improve code structure, maintainability, and alignment with industry security standards. The changes targeted key points within the Vulnerability Assessment Process Flow Diagram, including secure design, static analysis, and implementation review.

The refactoring process began with reorganizing the application’s components to follow Java’s best practices. Beforehand, multiple public classes were declared in a single file, which violates standard syntax rules and can interfere with Spring Boot’s component scanning. This was resolved by moving the HashController into its own file within the correct package, allowing Spring Boot to properly register and expose the /checksum endpoint. The use of annotations such as @RestController and @GetMapping ensured the method was available through a secured HTTPS connection.

From a security standpoint, SSL was configured using a self-signed certificate, enforcing encrypted communication between client and server. This addressed the “encryption enforced” phase in the vulnerability flow and ensured that all data in transit was securely handled. Additionally, the SHA-256 hashing algorithm was used to represent message integrity. While the input was static and not supplied by the user, this configuration minimized exposure to injection vulnerabilities and aligned with secure hashing practices.

To verify that no vulnerabilities were introduced during development, a static code analysis was performed using the OWASP Dependency-Check Maven plugin. This scan evaluated all project dependencies for known CVEs (Common Vulnerabilities and Exposures). The final report confirmed that the application did not introduce high-severity risks, supporting confidence in the system’s integrity. The results, including the report summary and runtime output, were captured via screenshots and included in the final report for documentation.

Finally, the entire codebase was reviewed manually for syntax correctness, logical flow, and potential security flaws. By consolidating the logic into modular, readable components, and limiting the number of accessible endpoints, the application’s attack surface was kept minimal. Unnecessary complexity was removed, and input handling remained tightly controlled. Altogether, these efforts reflect sound software engineering practices rooted in security awareness and design discipline.

## Industry Standard Best Practices

Throughout the development of the Spring Boot SSL server application, I applied industry-standard secure coding practices to both preserve the system’s integrity and proactively mitigate known vulnerabilities. By structuring the application properly, separating the controller from the main application class, using secure annotations like @RestController, and isolating logic in appropriately scoped methods, I ensured that the system remained organized and manageable, which is foundational to long-term security.

To uphold the existing security, I retained HTTPS support by configuring a self-signed SSL certificate, ensuring all traffic is encrypted via TLS. This provides confidentiality and integrity for transmitted data, even within a local environment. Additionally, I continued using SHA-256, a strong and widely recognized hashing algorithm, to preserve data integrity on the /checksum endpoint. The logic introduced no unsafe or unvalidated input handling, eliminating the risk of injection vulnerabilities or denial-of-service conditions.

Recognizing that third-party dependencies can introduce vulnerabilities, I also incorporated the OWASP Dependency-Check tool into my workflow. Running this tool allowed me to statically scan the application’s libraries for known CVEs. Verifying that no high-risk dependencies were introduced confirmed compliance with secure development protocols and further hardened the application against common threats.

Applying to these industry’s best practices is not just a technical necessity, it’s an organizational safeguard. Secure coding helps protect the company’s assets, brand, and customers by reducing the risk of data breaches, system downtime, and legal consequences. It builds trust with users and stakeholders while supporting long-term software maintainability. Ultimately, a culture of secure development doesn’t just prevent problems, it creates resilient systems that can evolve safely in a changing technological landscape.