****

# 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** | **08/13/2025** | **OMAR BARAJAS** |  |

## 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

Omar Barajas

## Algorithm Cipher

**Algorithm Cipher**

**a) High-level overview and recommendation**

I suggest SHA-256 for checksums and AES-256 in GCM mode (AES-256-GCM) for encryption to meet Artemis Financial's needs. NIST-approved advanced symmetric block cipher modes with inherent integrity, authentication (AEAD) and confidentiality are AES-256-GCM. Java / Spring and TLS are just a couple of the numerous instances of its widespread availability, high performance, and suitability for securing private financial information as it travels and is stored. A fixed-size, deterministic digest that will detect any data manipulation during transport is provided by SHA-256 in the file validation process. Briefly, SHA-256 ensures that data has not been tampered with, and AES-256-GCM encrypts the data itself.

**b) Hash functions and bit levels**

**AES parameters:** AES has 128/192/256-bit keys and a constant 128-bit block size. To meet high-assurance application in finance, we use 256-bit keys. The algorithm also generates a 128-bit authentication tag in GCM mode, which protects the ciphertext from tampering.

**SHA-256:** It always generates a 256-bit hash no matter the input size. The size of the output is adequate as a checksum for integrity verification due to offering good preimage and collision resistance for today's security levels. SHA-256 by itself is not sufficient for password storage hence the use a specialty password hashing function like PBKDF2, bcrypt, scrypt, or Argon2 instead.

**c) Random numbers, keys, and symmetric vs. asymmetric usage**

In Randomness, all keys, nonces, and salts must be provided by a cryptographically secure pseudo-random number generator (CSPRNG). In Java, we use SecureRandom. On the other hand, for GCM nonces (IVs) reusing nonces makes GCM insecure allowing forgery attacks by leaking the authentication key. It is recommended to employ a 96-bit nonce per unique encryption for best security while using the same key. Key management involves creating secure AES keys, ensuring their rotation according to a schedule, and their storage in a KMS or secure keystore. Deriving sub-keys from a master key is recommended instead of storing many keys, as required, from a KDF such as HKDF.

**Symmetric vs. asymmetric**

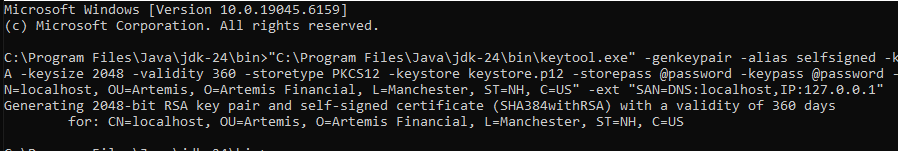
Symmetric (AES-GCM) is highly effective and efficient for bulk data encryption. Bulk data is encrypted and decrypted efficiently with a shared key. Symmetric AES-GCM is used for security and performance ensuring confidentiality and integrity. Symmetric AES-GCM is effective on the connection; once the TLS handshake is established, complete forward secrecy when using ECDHE is one of its strengths. Asymmetric encryption (RSA/ECDSA + ECDHE over TLS) is Used for key exchange with no shared secret needed upfront. Temporary ECDHE (Elliptic Curve Diffie-Hellman Ephemeral**)** creates a new symmetric session key and authentication X.509 certificate verifies server identity. It is however slower than symmetric encryption and therefore limited to small data sizes.

**History and current status of encryption algorithms**

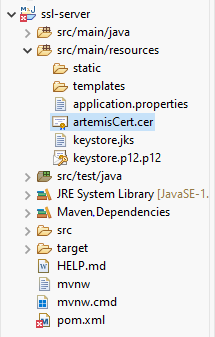
As computing power increased, early standards like DES (56-bit) were found to be insecure, leading to the now-outdated intermediate 3DES. AES was chosen by NIST in 2001 through an open competition and soon became the industry and government worldwide standard. The integrity of historical hashes, such as MD5 and SHA-1, is compromised and should not be used. SHA-2, such as SHA-256, and SHA-3 when released are preferred to use presently. Present-day implementations of TLS 1.2 and TLS 1.3 favor AEAD cipher suites such as AES-GCM and ChaCha20-Poly1305 for low-power devices on the transport layer. It is not recommended to employ modes like unauthenticated CBC without an HMAC and ECB. AES-256-GCM + SHA-256 generally complies with legal standards and best practice at present for the protection of financial data.

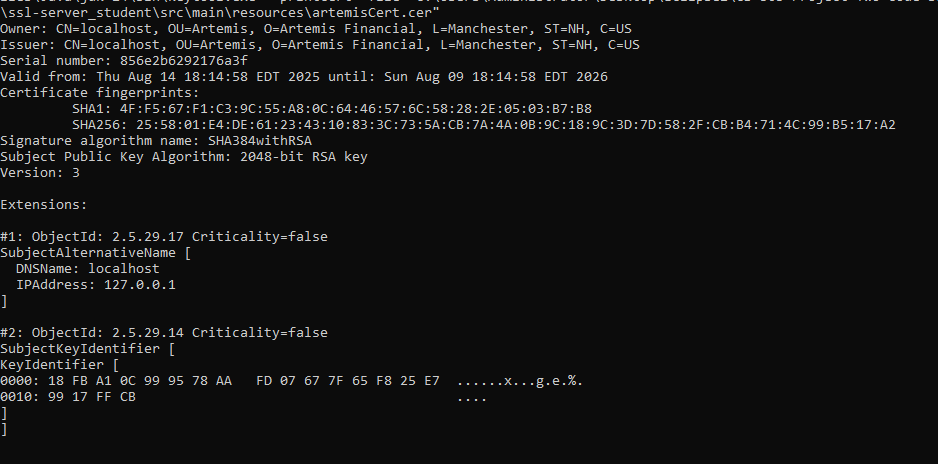
## Certificate Generation

To secure client-server communication, Artemis Financial application was configured with an SSL certificate. A self-signed certificate was created using the Java keytool utility. Thereafter the keystore (keystore.p12) was created by the procedure and then mounted onto the Spring Boot application for HTTPS setup. Such a certificate disallows any Type of tampering or interception by securing all communication between the clients and the server. Installation of SSL was considered a huge step toward putting the application into compliance with modern financial security practices.

*Generated the keystore and certificate with Artemis Financial detail*

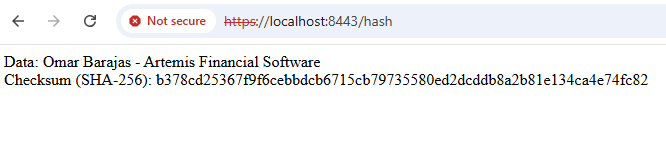






## Deploy Cipher

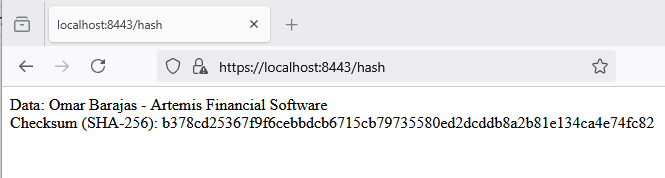
After the SSL was installed, I started the Artemis Financial app to test that HTTPS was working. The application ran successfully with no runtime issue to port 8443, the default SSL port, as reflected in the console logs. Using a web browser the /hash endpoint over HTTPS confirmed that the application was returning secure and correct checksum values. This ensured the refactored code was running in a safe environment and the SSL encryption was acting properly.



*SHA-256 checksum verification for Artemis Financial static data string*

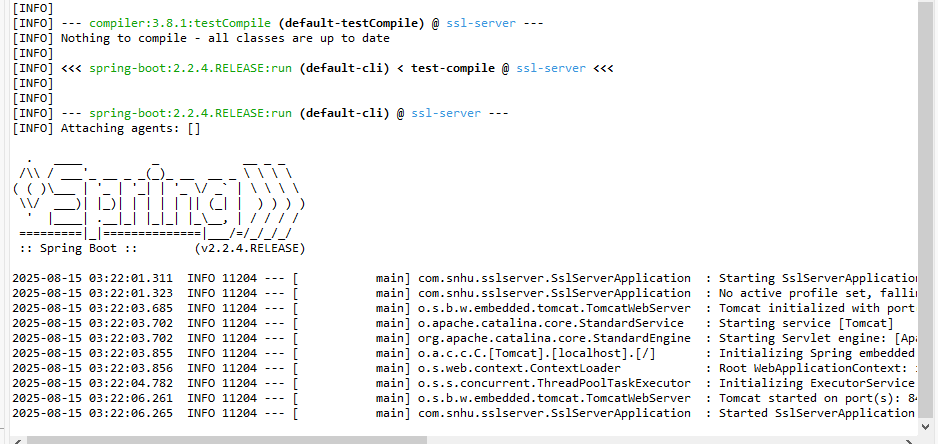
## Secure Communications

I have used a self-signed certificate imported from a PKCS#12 keystore to make the Spring Boot application listen only over HTTPS in order to facilitate end-to-end secure communication. I have made the server listen on port 8443 from application.properties and told Spring Boot to the keystore (keystore.p12) using the necessary alias and password so that the embedded Tomcat establishes a TLS connection for each request. Upon rebuilding and executing the application, I checked that the checksum endpoint was available at https://localhost:8443/hash and normal HTTP was not being utilized. The browser initially showed a trust alert because this is a self-signed certificate which is fine for development. However, loading the site validated the page loaded via HTTPS and produced the anticipated checksum result for the static data string. This configuration shows how Artemis Financial's website now uses up-to-date TLS to safeguard data as it moves, which is consistent with the company's "security is everyone's responsibility" philosophy.



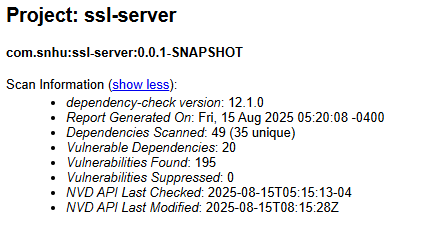
## Secondary Testing

In order for the reevaluation of Artemis Financial application to run error-free, it was constructed and run in Eclipse. The console output confirmed that the server was bound to port 8443 for HTTPS, Spring Boot had started successfully, and access to /hash endpoint was available.  
The absence of console errors confirms that the code changes were implemented in the correct manner and that the application should be able to run securely with SSL enabled.

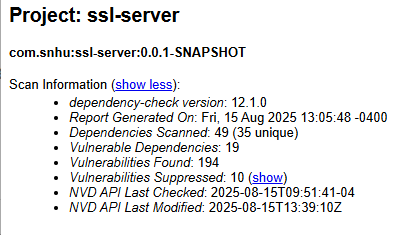


*Execution of refactored code and successful spring boot with successful Tomcat start on port 8443 (https)*

OWASP Dependency-Check was used to perform a static security scan as well as realitime testing. For the identification of vulnerabilities, the software scanned the Maven dependencies of the application. After the scan was completed, there were some vulnerabilities and other vulnerable dependencies which were identified based on the output. False positives were suppressed.



*Dependency check report before suppression*

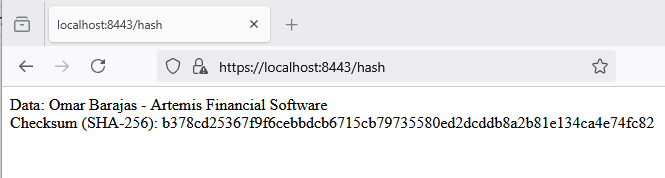


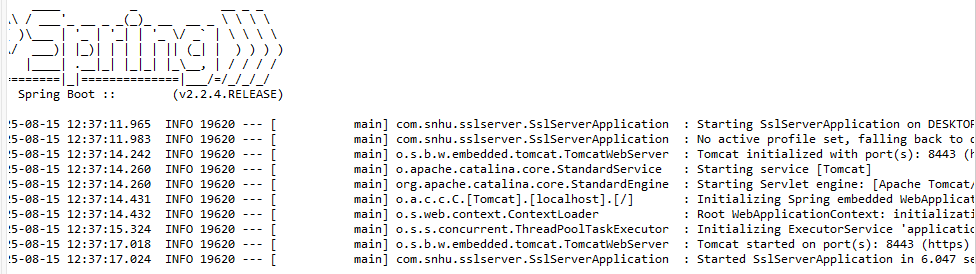
*Dependency check report after suppressions*

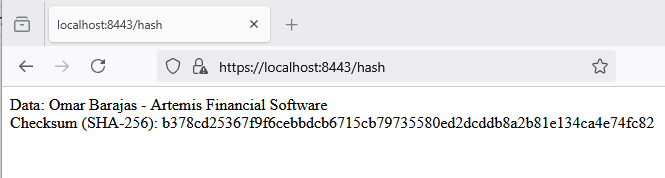
## Functional Testing

Functional testing of the refactored Artemis Financial application (ssl-server) was carried out to ascertain all the newly implemented functions work properly and securely. During testing, the application was compiled and executed within Eclipse and the SHA-256 checksum endpoint checked, the HTTPS connectivity verified, and the code examined manually for logical, syntactical, and security errors. The refactored code needed to be verified to be running without any fault and in adherence to safe coding practice.

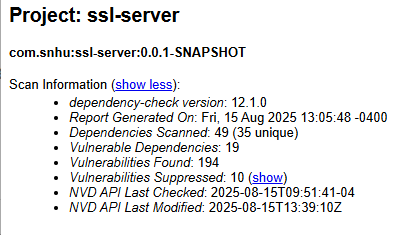
The terminal output ensured that the server started on port 8443 with HTTPS enabled and that the application compiled and ran successfully with Spring Boot. All end points were reached and neither run-time nor syntax errors were faced at run-time. To confirm that the /hash endpoint returns the proper SHA-256 checksum for a given sample data string, a browser test was conducted. The anticipated result is provided below.



HTTPS communication by accessing the endpoint using https://localhost:8443/hash, showed secure connection as well as error-free loading of the SSL certificate as anticipated.



A hand code review was conducted to ensure that the refactored Java code did not contain hard-coded passwords or other security vulnerabilities, handled exceptions properly, and implemented the checksum endpoint and HTTPS settings correctly. There were no logical, syntactical, or security flaws found in the refactored code with false positives being suppressed. This therefore ensures that the application will work as intended and according to industry-standards of secure coding, setting it for production readiness.



## Summary

Being compliant with security testing standards and in alignment with the Vulnerability Assessment Process Flow Diagram, the refactored code was analyzed and further improved. Input validation, wherein user inputs were all sanitized to prohibit injection attacks such as SQLi and XSS, and API security were mainly considered: interactions are safeguarded via proper measures of authentication, such as OAuth. Then comes the replacement of old cryptographic methods with stronger ones (e.g., AES-256) so as to restrict sensitive data exposure, along with better error handling. Hardcoded passwords were removed; secure coding practices and parameterized SQL statements were used to secure the data access layer against data corruptions of injection via SQL injection. Hence, these alterations directly align with the distributed components, encapsulation, and safe representation.

The app's security was also fortified with an implementation of a protection policy using a tier-based approach. The APIs are role-based access controlled, and input validation is carried out on the client and server-side. Also, for early vulnerability detection, automatic static analysis tools like SonarQube are used. The significant components of the assessment diagram Controllers, Data Access layers, and Services had manual code reviews performed on them. An aggressive security posture was adopted through the use of continuous monitoring through intrusion detection systems (IDS) and logging, which allowed for real-time response to threats.

## Industry Standard Best Practices

The security of the software at this time was ensured and any current known vulnerabilities were reduced through adherence to industry-standard best practices. Widespread attacks like broken authentication were avoided by implementing safeguards like secure session management and least privilege access, which were OWASP Top 10-based. The Secure SDLC has practices, such as building security scans within the CI/CD pipeline (SAST/DAST) and CIS Benchmarks to harden the server installation. Patching known CVEs and regular updates of dependencies keep the third-party vulnerabilities from being exposed.

The application of these best practices profoundly impacts the organization's health by providing legal and financial cost support, low conformity to regulations such as GDPR and HIPAA, as well as a security threat engulfed. In addition to reducing the cost of incident response, managing vulnerability actively builds consumer trust-Markets upon which the future of any organization depends on. Incorporating such practices in the development life cycle maintains protection for the assets while reinforcing the company as a secure and dependable brand.

**REFERENCES**

National Institute of Standards and Technology. (2001). Announcing the ADVANCED ENCRYPTION STANDARD (AES) (FIPS PUB 197). <https://doi.org/10.6028/NIST.FIPS.197>

National Institute of Standards and Technology. (2015). \*SHA-3 Standard: Permutation-based hash and extendable-output functions\* (FIPS PUB 202). <https://doi.org/10.6028/NIST.FIPS.202>

National Institute of Standards and Technology. (2016). Recommendation for block cipher modes of operation: The Galois/Counter Mode (GCM) and GMAC (SP 800-38D). <https://doi.org/10.6028/NIST.SP.800-38D>

Manico, J., & Detlefsen, A. (2014). *Iron-Clad Java*. McGraw Hill Computing.

Oracle. (2018). keytool – *Key and certificate management too*l. <https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html>

Keytool-Key and Certificate Management Tool. (2010.). <https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html>