****

# 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** | **10/18/2025** | **Sadman Anwar** |  |

## 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

Sadman Anwar

## Algorithm Cipher

Artemis Financial seeks a secure, scalable solution for encrypting long-term archive files. Based on current cryptographic standards and enterprise best practices, I recommend deploying the Advanced Encryption Standard with a 256-bit key in Galois/Counter Mode (AES-256-GCM). This cipher offers authenticated encryption, ensuring both confidentiality and integrity, and is widely supported across modern platforms and regulatory frameworks (Oracle, n.d.; RFC 5116, 2008).

**Security Protection Best Practices** To defend against brute-force attacks, ciphertext tampering, and replay attacks, AES-256-GCM should be implemented with the following best practices:

* Use a unique, cryptographically secure initialization vector (IV), which is a non-secret value used to initialize an encryption operation so that repeated encryptions of the same plaintext under the same key produce different ciphertexts, for each encryption operation (NIST, 2013).
* Validate the GCM authentication tag during decryption to detect tampering (Kampanakis et al., 2024).
* Store keys in a centralized Key Management Service (KMS) with strict access controls and audit logging (ShareArchiver, 2023).
* Rotate encryption keys periodically and enforce least-privilege access policies.
* Use envelope encryption to separate data keys from master keys, reducing exposure in case of compromise.

**How the Cipher Will Be Used** Each archive file will be encrypted using a unique AES-256 key in GCM mode, an authenticated encryption mode for block ciphers that turns a block cipher (typically AES) into an AEAD primitive that provides both confidentiality (encryption) and integrity/authenticity (an authentication tag) in one operation. The IV and authentication tag will be stored with the ciphertext. The AES key will be encrypted (wrapped) using a KMS-managed master key. This envelope encryption model ensures secure key distribution and simplifies key rotation (RFC 5116, 2008).

**Best Cipher and Why** AES-256-GCM is the best choice because it:

* Provides both confidentiality and integrity in one operation.
* Is hardware-accelerated on modern processors (AES-NI).
* Is widely supported and NIST-approved.
* Has a strong security margin due to its 256-bit key size (Security Stack Exchange, 2018).

**Justification and Technical Overview** AES is a symmetric block cipher, meaning the same key is used for encryption and decryption. Its 256-bit key size offers a vast keyspace, making brute-force attacks infeasible. GCM mode uses a nonce (IV) and produces an authentication tag, ensuring both confidentiality and integrity (NIST, 2013).

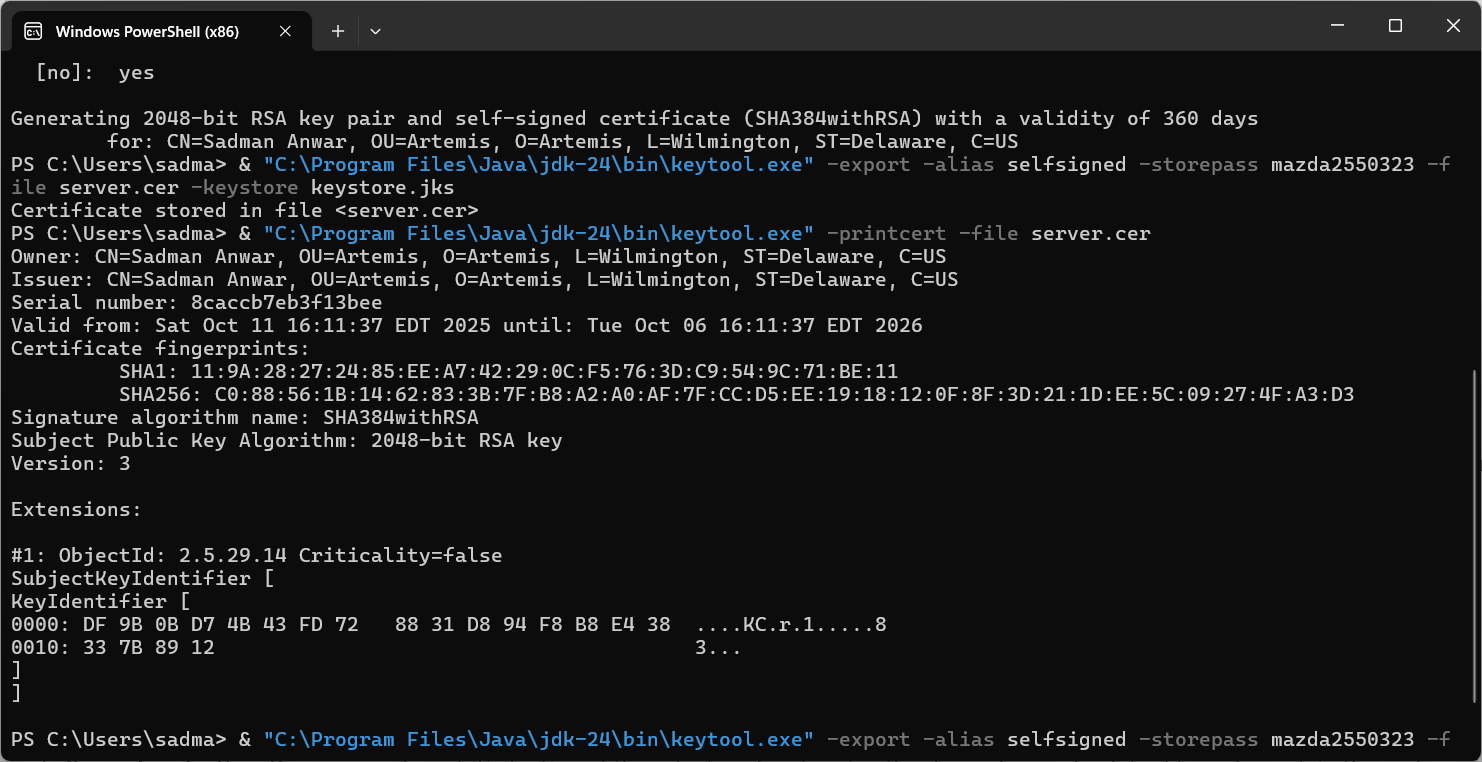
**Hash Functions and Bit Levels** Hash functions like SHA-256 are used for key derivation and integrity checks. Bit levels (e.g., 256-bit keys) indicate the strength of the cipher against brute-force attacks. Higher bit levels offer greater resistance to exhaustive key search.

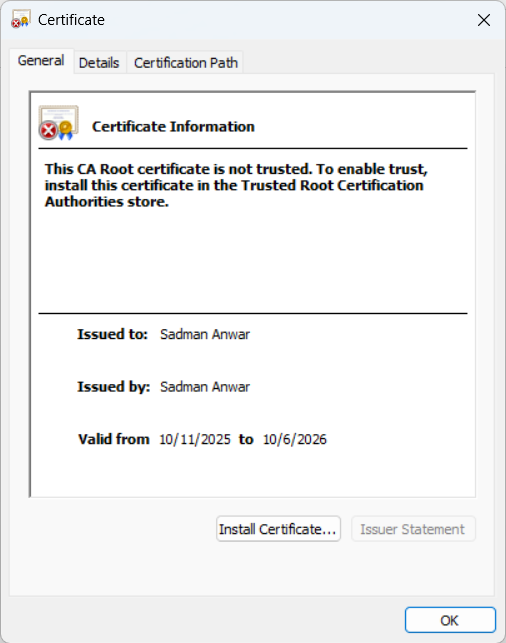
**Random Numbers and Key Types** Cryptographically secure random numbers are essential for generating IVs and keys. Symmetric keys (used in AES) are efficient for bulk data encryption. A symmetric key is a single shared secret key that encrypts and decrypts data. Symmetric algorithms are fast and suitable for bulk data encryption (AES). While Asymmetric keys use separate public and private keys, the public key can be widely distributed, and the private key remains secret. Use cases include certificate signing, key exchange, and digital signatures (RSA, ECDSA). Asymmetric keys (e.g., RSA, ECC) are used to encrypt symmetric keys or establish trust between parties (RFC 5116, 2008).

**History and Current State of Encryption Algorithms** Encryption has evolved from simple substitution ciphers to modern block ciphers. DES was once standard but was replaced by AES due to its vulnerability to brute-force attacks. AES remains unbroken and is the global standard for symmetric encryption. GCM mode was introduced to combine encryption and authentication efficiently (NIST, 2013; Oracle, n.d.).

## Certificate Generation

Insert a screenshot below of the CER file.





## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screenshot of a computer

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.

**Refactored Code Executed Without Errors**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Dependency Check Report**

**A screenshot of a computer

AI-generated content may be incorrect.**

## Functional Testing

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

**Server Started**

A screenshot of a computer

AI-generated content may be incorrect.

**Running on localhost:8443**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Manual Code Review**

High severity

* Hard-coded keystore password in application.properties: Need to remove and use an environment variable or externalized config to store the password.
* Application serves both HTTP and HTTPS, allowing mixed content: Need to disable HTTP or add a redirect and HSTS for production.

Medium severity

* Unhandled exceptions that reveal stack traces to clients: add global exception handling to hide internals. For instance, in the HashController file we should verify MessageDigest exceptions are handled, and it is preferable to convert checked exceptions to meaningful 500 response rather than throwing a raw Exception. These exceptions can be handled by using try-catch statements in Java.
* Insecure dependency: Need to upgrade the vulnerable dependencies.

Low severity

* Minor formatting, unused imports, or noncritical logging statements: Need to remove unused imports, add more meaningful logs, and get rid of non-critical logging statements

## Summary

This project refactored the Artemis Financial application to add transport-level security, data integrity verification, and dependency vulnerability management. I implemented a secure endpoint (/hash) that returns a SHA-256 checksum for a clearly labeled input string that includes my name and a unique data token for verification. I generated a self‑signed certificate (keystore.jks and server.cer) using Java Keytool and configured Spring Boot to serve HTTPS on port 8443 by adding SSL keystore settings to application.properties. . The application was compiled and run without errors. A browser request to https://localhost:8443/hash confirmed secure transport and successful checksum generation. Finally, I integrated the OWASP dependency-check into the Maven lifecycle and re-ran the static analysis to ensure the refactor did not introduce new vulnerabilities; any preexisting false positives were suppressed using suppression.xml with documented rationale. Limitations remain: the keystore is self-signed and appropriate only for development; production requires a CA-signed certificate and KMS/HSM-backed key management. The next steps are to automate dependency scanning in CI, replace the self-signed certificate with a CA-signed certificate, and rotate keys on a regular schedule.

## Industry Standard Best Practices

I applied the following industry-standard best practices during the refactor:

* Use of vetted cryptographic primitives: I used SHA‑256 from Java’s standard crypto provider for integrity verification and recommend AES‑GCM for confidentiality in production.
* Proper randomness and IV handling: any symmetric encryption should use a CSPRNG, (Cryptographically Secure Pseudo-Random Number Generator)for IV/nonce generation and must guarantee uniqueness per key. Self-signed certificate private keys must be generated with secure entropy and stored securely.
* Secure configuration: HTTPS was enforced in application.properties with the keystore on the application classpath for local testing; in production this keystore should be replaced by a certificate from a trusted CA and keys stored in a managed key store.
* Dependency hygiene: OWASP Dependency-Check was integrated into Maven to scan for known vulnerable components.
* Least privilege and separation of concerns: application configuration avoids storing plaintext secrets in source files for production; secrets should be externalized to environment variables or secret managers.
* Testing and automation: functional testing validated endpoint behavior and checksum output; dependency scanning is planned to be automated in CI with gating rules to prevent accidental inclusion of new high-risk dependencies.
* Documentation and auditability: all changes, including keystore generation commands, are documented in the report and the project ZIP for reproducibility and future audit.

**References**

Kampanakis, P., Campagna, M., Crocket, E., Petcher, A., & Gueron, S. (2024). *Practical challenges with AES-GCM and the need for a new cipher*. NIST Workshop on Block Cipher Modes of Operation. <https://csrc.nist.gov/csrc/media/Events/2023/third-workshop-on-block-cipher-modes-of-operation/documents/accepted-papers/Practical%20Challenges%20with%20AES-GCM.pdf>

National Institute of Standards and Technology. (2013). *Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC* (SP 800-38D). <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf>

Oracle. (n.d.). *Cipher algorithm names (Java Security Standard Algorithm Names)*. <https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#cipher-algorithm-names>

RFC 5116. (2008). *An interface and algorithms for authenticated encryption*. <https://datatracker.ietf.org/doc/html/rfc5116>

RFC 8439. (2018). *ChaCha20 and Poly1305 for IETF protocols*. <https://datatracker.ietf.org/doc/html/rfc8439>

Security Stack Exchange. (2018). *Why would I ever use AES-256-CBC if AES-256-GCM is more secure?* <https://security.stackexchange.com/questions/184305/why-would-i-ever-use-aes-256-cbc-if-aes-256-gcm-is-more-secure>

ShareArchiver. (2023, September 30). *Guarding your data: The power of AES-256 encryption in archiving*. <https://sharearchiver.com/blog/aes-256-encryption-archiving/>