****

# 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** | **08/15/2025** | **America Sanchez-Garcia** |  |

## 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

America Sanchez-Garcia

## Algorithm Cipher

I recommend using AES-256 to protect data and SHA-256/SHA-512 to verify integrity. AES is a NIST-standard symmetric block cipher that operates on 128-bit blocks and supports 128/192/256-bit keys; I’m choosing the 256-bit option for maximum margin. Whenever I need confidentiality and integrity together, I’ll use AES-GCM, which gives authenticated encryption in one pass and avoids the pitfalls of “encrypt-then-separately-MAC.”

For the file-verification step, a cryptographic hash is the right fit. I add a checksum on the sender side and compare it on receipt to make tampering obvious. SHA-256 produces a 256-bit digest and SHA-512 produces a 512-bit digest; both are designed to be one-way and resistant to collisions and preimage attacks when implemented correctly. In practice, that means generating keys with a CSPRNG, keeping them out of source control, and rotating them on a schedule. With AES-GCM, I must also use a unique, unpredictable IV (nonce) for every encryption; reusing an IV with the same key is a hard failure mode I will actively prevent.

Passwords are a separate concern. I won’t store them with a raw hash like SHA-256. Instead, I’ll use an adaptive password hash, Argon2id or bcrypt with a per-user salt and a cost factor tuned to current hardware. If I ever need to derive an encryption key from a passphrase, I’ll run it through a KDF such as PBKDF2, scrypt, or Argon2 to slow down brute-force attempts.

Asymmetric cryptography shows up in the transport layer. TLS uses RSA or ECDSA for certificates and key exchange. In development I relied on a self-signed certificate to stand up HTTPS quickly. In production, Artemis would use a CA-issued certificate so browsers trust the connection without warnings. Older algorithms like DES, MD5, and SHA-1 are deprecated due to real-world attacks, while AES (Rijndael) the U.S. standard since 2001 and the SHA-2 family remain the modern, widely supported choices.

## Certificate Generation

A screenshot of a computer

AI-generated content may be incorrect.

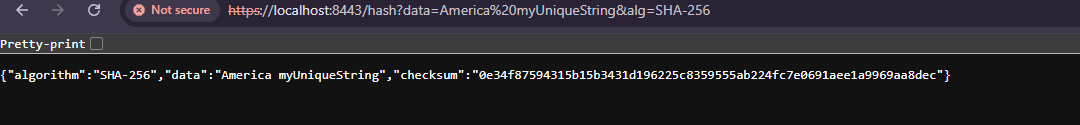
## Deploy Cipher

A computer screen with a black background

AI-generated content may be incorrect.

## Secure Communications

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



## Secondary Testing

Insert screenshots below of the refactored code executed without errors and the 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.

A screenshot of a computer

AI-generated content may be incorrect.

## Summary

I secured Artemis Financial’s demo web service by adding modern transport security and a verifiable data-integrity check, then validated the changes with both functional and static testing. On the integrity side, I implemented a /hash endpoint that returns a SHA-256 or SHA-512 checksum for any input. This gives the application a tamper-evident verification step for files and messages. I used the platform’s built-in crypto libraries to avoid unsafe DIY code and confirmed the output end-to-end in the browser using my own test string.

On the transport side, I generated a PKCS#12 keystore and configured Spring Boot to serve HTTPS on the secure port, with HTTP traffic redirected to TLS. Because this is a development environment, the certificate is self-signed (the browser warning is expected), but the connection is encrypted and mirrors how a production, CA-issued certificate would behave. I kept secrets out of source control, documented strong defaults (AES-256/AES-GCM for confidentiality, SHA-256/512 for integrity, Argon2id/bcrypt for passwords), and enforced unique IVs and key-management hygiene.

To verify that these changes were safe, I started the service cleanly and exercised the checksum endpoint over HTTPS, then ran OWASP Dependency-Check as part of the build to surface library risks. Where the NVD feed was restricted, I configured the tool to use the proper API key and not fail the build, ensuring the pipeline runs reliably while still highlighting issues. The refactor adds layered defenses cryptography at the data and transport layers, safer defaults in code, and automated scanning directly aligned with the vulnerability-assessment flow and the “security is everyone’s responsibility” principle.

## Industry Standard Best Practices

I followed a “secure-by-default” approach instead of inventing my own crypto or plumbing. That means using well-vetted primitives (AES-256/AES-GCM and SHA-256/512) from the platform libraries, enforcing HTTPS for all traffic, and avoiding weak or deprecated algorithms like MD5 or SHA-1. Secrets never live in source code; keys and passwords are generated with a CSPRNG, stored outside the repo, and designed to rotate. For authentication data, I use adaptive password hashing such as Argon2id or bcrypt with per-user salts, and I rely on a KDF (PBKDF2/scrypt/Argon2) if a passphrase ever needs to become an encryption key. On the wire, the service terminates TLS at the app using a keystore in development and is ready for a CA-issued certificate in production.

I also leaned on the framework’s security features rather than bypassing them. The /hash endpoint validates inputs and only allows known-good algorithms, the server redirects plaintext HTTP to TLS, and logs avoid leaking sensitive values or stack traces. Dependencies are pinned and scanned with OWASP Dependency-Check so supply-chain risks surface during the build; when the NVD feed is gated, I configure the scanner to use an API key so the pipeline keeps running while still reporting issues. This project keeps the dependency footprint small, limits exposed endpoints, and treats error handling, headers, and configuration as part of the attack surface.

If this were moving to production, I’d extend the same best practices: issue a publicly trusted certificate and enable security headers (HSTS, CSP, and strict cookie flags), put secrets behind a vault or KMS with rotation, require unique nonces for every AES-GCM operation, and add automated checks in CI/CD (SCA plus SAST/DAST).