**Practices for Secure Software Report  
Artemis Financial — Global Rain**

*Developer: Matthew Baerg  
Date: 2025-10-09  
Course: CS 305 — Software Security*

# Table of Contents

Document Revision History

Client

Instructions

Developer

1. Algorithm Cipher

2. Certificate Generation

3. Deploy Cipher

4. Secure Communications

5. Secondary Testing

6. Functional Testing

7. Summary

8. Industry Standard Best Practices

# Document Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comments |
| 1.0 | 2025-10-09 | Matthew Baerg | Initial submission |

# Client

Artemis Financial seeks to modernize its public-facing web application with robust, modern security controls to protect client financial data in transit and at rest. This report documents the security enhancements implemented and the accompanying test evidence.

# Instructions

This report follows the Project Two template: it documents an algorithm cipher recommendation, certificate generation, checksum deployment, HTTPS configuration, secondary static testing, functional testing, and a summary tied to the vulnerability assessment flow.

# Developer

Matthew Baerg

# 1. Algorithm Cipher

Recommendation: Use AES‑256‑GCM for symmetric encryption of data in transit/at rest within services, and SHA‑256 for one‑way checksums/verification. For digital signatures (if needed), prefer ECDSA with P‑256 (or Ed25519) over RSA for performance and security trade‑offs.

Overview of AES‑GCM and SHA‑256 (high level):

• AES‑GCM is an authenticated encryption mode that provides confidentiality, integrity, and authenticity in a single primitive. Galois/Counter Mode adds an authentication tag to detect tampering. AES‑256 uses a 256‑bit key.

• SHA‑256 is a cryptographic hash function that maps arbitrary input to a 256‑bit digest. It is preimage and collision‑resistant for current practical purposes and appropriate for file checksums and integrity verification.

Hash functions and bit levels:

• SHA‑256 → 256‑bit digest. For stronger margins, SHA‑384/512 are available. Avoid broken/weak hashes like MD5 or SHA‑1 for security‑critical use.

Randomness and keys:

• Use a CSPRNG (e.g., Java SecureRandom) for IVs/nonces and key material. For AES‑GCM, use a unique 96‑bit IV per encryption; never reuse IVs with the same key.

Symmetric vs. asymmetric:

• Symmetric (AES) is fast and used for bulk data. Asymmetric (RSA/ECDSA) is used for key exchange and identity. In TLS, asymmetric crypto authenticates the server and establishes session keys; symmetric ciphers encrypt the session.

Brief history/current state:

• The industry has moved from DES/3DES and MD5/SHA‑1 to AES (with GCM/CCM modes) and SHA‑2/SHA‑3 families. ECDH/ECDSA are preferred over RSA‑2048 in many contexts due to performance and forward‑secrecy with ephemeral key exchange (TLS 1.2/1.3).

# 2. Certificate Generation

Self‑signed certificate generated via Java Keytool (keystore PKCS12). Example commands used in Eclipse/terminal:

keytool -genkeypair -alias artemis-server -keyalg RSA -keysize 2048 -validity 365 -storetype PKCS12 -keystore keystore.p12 -storepass changeit -keypass changeit -dname "CN=localhost, OU=Engineering, O=Global Rain, L=Seattle, S=WA, C=US"  
# Export public cert for trust testing:  
keytool -exportcert -alias artemis-server -storetype PKCS12 -keystore keystore.p12 -storepass changeit -rfc -file artemis.cer

Evidence:

[Insert CER file listing (e.g., artemis.cer in project root) screenshot here]

*Tip: Include your name and a unique test string in the screenshot when requested.*

# 3. Deploy Cipher

Refactor the application to expose a checksum endpoint that returns a SHA‑256 hash of a client‑provided input. Include the developer’s name and a unique data string in the UI or request so the screenshot proves authorship.

Example Spring components:

Service (ChecksumService.java)

import java.nio.charset.StandardCharsets;  
import java.security.MessageDigest;  
import java.util.HexFormat;  
import org.springframework.stereotype.Service;  
  
@Service  
public class ChecksumService {  
 public String sha256(String input) {  
 try {  
 MessageDigest md = MessageDigest.getInstance("SHA-256");  
 byte[] digest = md.digest(input.getBytes(StandardCharsets.UTF\_8));  
 return HexFormat.of().formatHex(digest);  
 } catch (Exception e) {  
 throw new IllegalStateException("SHA-256 unavailable", e);  
 }  
 }  
}

Controller (HashController.java)

import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class HashController {  
 private final ChecksumService checksumService;  
 public HashController(ChecksumService checksumService) {  
 this.checksumService = checksumService;  
 }  
  
 // Example: https://localhost:8443/hash?data=MatthewBaerg-unique-12345  
 @GetMapping("/hash")  
 public String hash(@RequestParam String data) {  
 return checksumService.sha256(data);  
 }  
}

Evidence:

[Insert Checksum verification showing name + unique data string and resulting hash screenshot here]

*Tip: Include your name and a unique test string in the screenshot when requested.*

# 4. Secure Communications

HTTP→HTTPS: Configure Spring Boot to use the self‑signed cert and listen on 8443. Disable plain HTTP for the demo, or redirect it to HTTPS.

application.properties

server.port=8443  
server.ssl.enabled=true  
server.ssl.key-store=classpath:keystore.p12  
server.ssl.key-store-type=PKCS12  
server.ssl.key-store-password=changeit  
server.ssl.key-alias=artemis-server  
  
# (Optional) Strict transport security headers via a filter or security config if using Spring Security.

Place keystore.p12 under src/main/resources. Run the app, then open https://localhost:8443/hash in a browser. Because the certificate is self‑signed, the browser will warn; proceed to view the page for local testing.

Evidence:

[Insert Secure webpage at https://localhost:8443/hash (browser address bar + lock/warning) screenshot here]

*Tip: Include your name and a unique test string in the screenshot when requested.*

# 5. Secondary Testing

Run OWASP Dependency‑Check (Maven plug‑in) to verify that refactoring did not introduce new known‑vulnerable dependencies. Focus on newly added/updated components.

pom.xml (plugin snippet)

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.owasp</groupId>  
 <artifactId>dependency-check-maven</artifactId>  
 <version>12.1.0</version>  
 <executions>  
 <execution>  
 <goals>  
 <goal>check</goal>  
 </goals>  
 </execution>  
 </executions>  
 <!-- For air-gapped runs, you may need suppression files or cached data -->  
 </plugin>  
 </plugins>  
</build>

Commands:

# Update then run analysis  
mvn -U org.owasp:dependency-check-maven:check -Dformats=HTML,JSON  
# Or bind to the build lifecycle:  
mvn clean verify

Evidence:

[Insert Refactored code executed without errors (console build success) screenshot here]

*Tip: Include your name and a unique test string in the screenshot when requested.*

[Insert Dependency‑Check HTML report summary (no new vulnerabilities) screenshot here]

*Tip: Include your name and a unique test string in the screenshot when requested.*

# 6. Functional Testing

Manual review checklist (synthesized):

• Syntactical: Application builds with no compile errors; endpoints load; configuration resolves keystore.  
• Logical: /hash returns expected digests for known vectors; empty/invalid inputs are handled gracefully (400 Bad Request or equivalent).  
• Security: Only HTTPS is enabled; no sensitive data is logged; CSPRNG used where applicable; weak algorithms disabled; dependencies scanned clean.

Evidence:

[Insert Application running, /hash endpoint tested successfully (curl/browser) screenshot here]

*Tip: Include your name and a unique test string in the screenshot when requested.*

# 7. Summary

Mapping to the Vulnerability Assessment Process Flow:

• Architecture Review → Chosen AES‑GCM for confidentiality and SHA‑256 for integrity; planned TLS termination.  
• Input Validation → /hash validates presence of data param; consider length limits and canonicalization.  
• APIs → Minimal REST surface; returning only derived digests, no secrets.  
• Cryptography → Proper algorithms (AES‑GCM, SHA‑256); unique IVs and secure key storage emphasized.  
• Client/Server → HTTPS configured with self‑signed cert for local; ready for CA‑issued cert in production.  
• Code Error/Quality/Encapsulation → Clear service/controller separation, exceptions wrapped, secrets externalized.

Outcome: The refactoring adds transport‑layer security, a verifiable checksum feature, and passes static and functional checks without introducing new known vulnerabilities.

# 8. Industry Standard Best Practices

Maintaining security while refactoring:

• Follow least‑privilege for secrets; do not commit keystores/passwords to source control.  
• Prefer vetted libraries and keep them updated; scan with Dependency‑Check regularly.  
• Enforce HTTPS, modern TLS versions, and strong cipher suites; disable obsolete protocols.  
• Handle errors without leaking stack traces or secrets (centralized exception handling).

Organizational value:

• Applying standards like TLS with strong ciphers, SHA‑256 checks, and routine SCA (software composition analysis) reduces breach risk, improves compliance posture, and builds client trust while lowering long‑term maintenance costs.

**Screenshot checklist:**

1. CER file export

A screenshot of a computer program

AI-generated content may be incorrect.  
2) Checksum verification showing your name + unique data  
3) HTTPS browser page at <https://localhost:8443/hash>

A screenshot of a computer

AI-generated content may be incorrect.  
4) Successful build console output

A black screen with white text

AI-generated content may be incorrect.

A computer screen with text and images

AI-generated content may be incorrect.  
5) Dependency‑Check report summary

A screenshot of a computer

AI-generated content may be incorrect.  
6) App running without errors (functional test)