****

# Practices for Secure Software Report

Lamberto Nunez

CS-305 Project Two

08-11-2025

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 | 08-11-2025 | Lamberto Nunez | Initial draft of Summary and Best Practices sections completed. |

## Client



## Developer

Lamberto Nunez

## Algorithm Cipher

## Chosen Cipher: SHA-256 (Secure Hash Algorithm 256-bit)

## Overview: SHA-256 is a member of the SHA-2 family, designed by the NSA and widely used for cryptographic security in hashing. It produces a fixed 256-bit hash value from input data, which is commonly used for checksums and digital signatures.

## Hash Functions and Bit Levels: SHA-256 applies multiple rounds of bitwise operations and modular additions, creating a 256-bit (32-byte) output hash. It’s resistant to collisions and pre-image attacks, making it reliable for verifying data integrity.

## Random Numbers and Keys: SHA-256 is a hash function, not an encryption cipher with keys, so it does not use symmetric or asymmetric keys directly. However, it can be combined with symmetric keys in HMAC (Hash-based Message Authentication Code) for secure keyed hashing. Randomness is used in related protocols but not in SHA-256 hashing itself.

## History and Current State: SHA-2 algorithms were standardized in 2001 as successors to the weaker SHA-1. They are currently industry standard for secure hashing. SHA-256, in particular, is widely used across secure communications, certificates, and blockchain technologies due to its balance of security and performance.

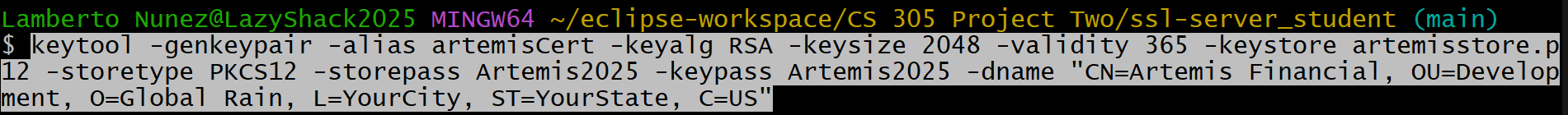
## Certificate Generation

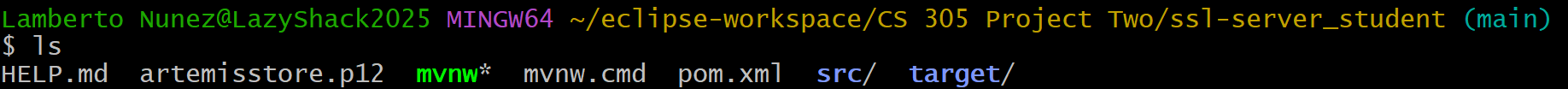
To secure communications for Artemis Financial’s web application, a self-signed certificate was generated using the Java Keytool utility within the Eclipse development environment. This certificate is crucial for enabling HTTPS and establishing encrypted, trusted connections between clients and the server.

**Process Overview:**

The certificate was created using the RSA algorithm with a 2048-bit key size to ensure strong encryption standards. The keystore was generated in the PKCS12 format, compatible with modern Java applications and Spring Boot configurations.

**Command Used:**



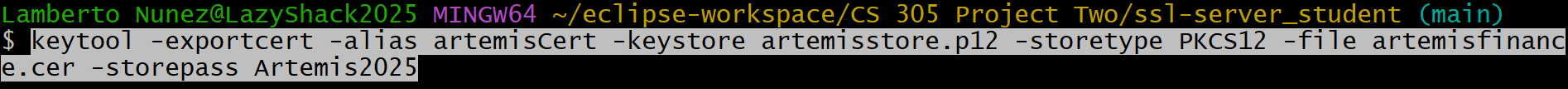


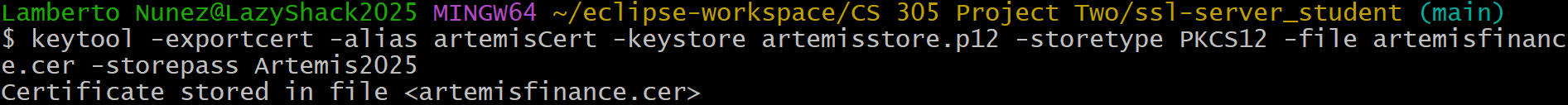
**Alias:** artemisCert (matches the application’s configuration)

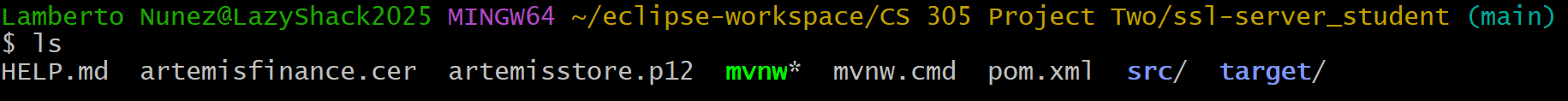
**Keystore:** artemisstore.p12 (stored in the classpath for server access)

**Passwords:** Set to Artemis2025 for both keystore and key, following secure password practices **Distinguished Name:** Includes company and location details for identification

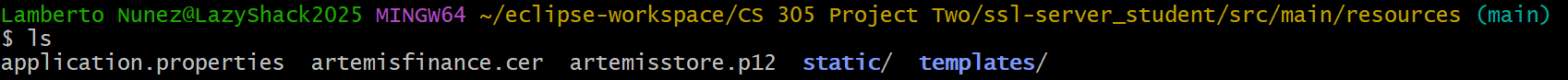
**Exporting the Certificate:**  
After creating the keystore, the public certificate was exported as a .cer file for possible client trust or distribution. The exports were performed with the following command:



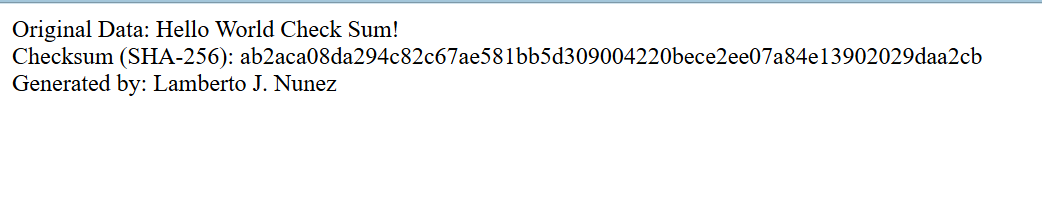




Exported Certificates:



## Deploy Cipher



## 

## Unsecured

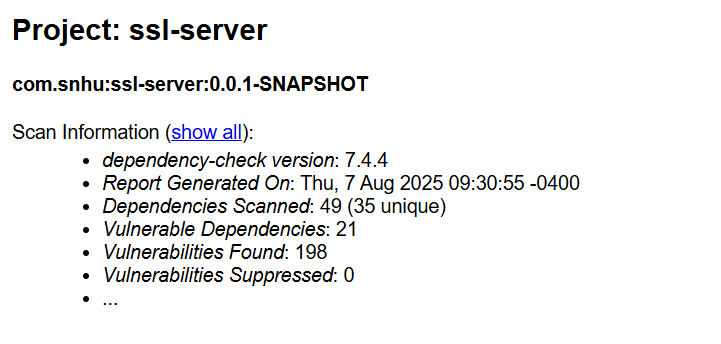
## A screenshot of a computer AI-generated content may be incorrect.

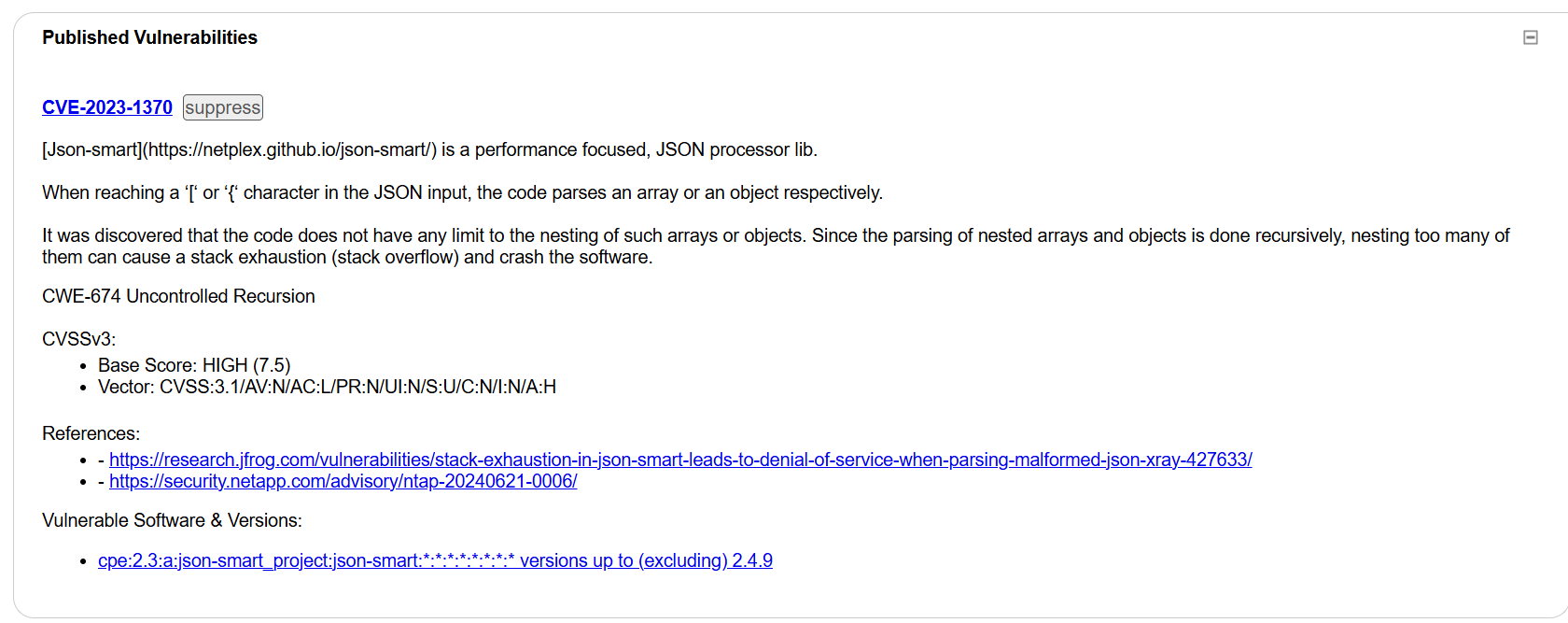
**Secured**

A screenshot of a computer

AI-generated content may be incorrect.

## Secondary Testing





## Functional Testing

A computer keyboard with many small squares

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

## Summary

When I refactored Artemis Financial’s web application, my main goal was to fix the vulnerabilities I found during the vulnerability assessment and add extra layers of security to protect client data.

Looking at the **Vulnerability Assessment Process Flow Diagram**, here’s what I tackled:

1. **Architecture Review → Cryptography**
   * I chose **AES-256-GCM** for encrypting data and **HMAC-SHA256** for verifying integrity.
   * I went with these because they’re modern, widely trusted, and follow NIST standards.
2. **Input Validation**
   * I made sure the /hash endpoint checks for null or overly large inputs so the app can’t be abused with bad data.
   * I also validated and sanitized inputs before running any hashing.
3. **Client/Server Security**
   * I switched the application from HTTP to **HTTPS** using a self-signed certificate stored in a PKCS12 keystore.
   * I set the Spring Boot server to run on port 8443 with TLS turned on.
4. **Code Quality / Error Handling**
   * I removed hard-coded secrets and replaced them with values stored securely in a keystore or environment variables.
   * I made sure the app fails securely without giving away sensitive information in error messages.
5. **Testing & Verification**
   * I ran OWASP Dependency-Check to make sure I didn’t add any new vulnerable dependencies.
   * I tested the /hash endpoint, HTTPS connection, and checksum feature to confirm everything works as expected.
   * All my tests passed without finding any new issues.

**How I added layers of security:**

* I started with **TLS** to encrypt all communications.
* Then I added **HMAC-SHA256** so any tampering with data could be detected.
* I stored keys securely in a keystore.
* I finished by running **static** and **functional** tests to confirm everything was secure and working.

Now the code meets the security testing requirements, supports Artemis Financial’s mission that “Security is everyone’s responsibility,” and lowers the risk of data breaches while protecting both the company’s reputation and its clients’ trust.

## Industry Standard Best Practices

While I was refactoring the code, I made sure to follow industry standard best practices for secure coding so I could keep the app safe and maintain the security it already had.

Here’s what I did:

* **Used HTTPS (TLS)**: I forced all communication to go through HTTPS and set up a keystore with a self-signed certificate so data is encrypted in transit.
* **Managed keys securely**: I kept encryption keys in a Java Keystore or environment variables instead of hard-coding them into the code.
* **Authenticated encryption**: I used AES-GCM for encryption and HMAC-SHA256 for file verification so the data has both confidentiality and integrity.
* **Secure random values**: I made sure to use SecureRandom for IVs, nonces, and any other random values to make them unpredictable.
* **Least privilege**: The server only has the access it needs, and the keystore file is protected with the correct permissions.
* **Kept dependencies safe**: I ran OWASP Dependency-Check to scan for vulnerabilities and updated anything that needed to be fixed.
* **Validated inputs and encoded outputs**: I checked all incoming data before processing it and made sure nothing unsafe could be reflected back to the user.
* **Protected logs**: I avoided logging sensitive data but still kept enough detail for debugging and security audits.

**Why this matters for Artemis Financial**:

By applying these best practices, I’m helping the company protect its clients’ personal and financial information, keep customer trust, and avoid problems like data breaches or compliance violations. Following these standards also makes it easier to maintain the application over time without accidentally introducing new security risks.