****

# 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** | **10/19/2025** | **Reice Morgan** | **Project 7-1** |

## 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

Reice Morgan

## Algorithm Cipher

For this project, I implemented a secure algorithm cipher to protect the integrity and confidentiality of transmitted data. The cipher works by converting readable text (plaintext) into an unreadable format (ciphertext) using a specific encryption algorithm and a secret key. This ensures that only authorized users with the correct key can decrypt and access the original data.

I chose to use a **SHA-256 hashing algorithm** because it is widely trusted in the security community and provides strong protection against attacks. SHA-256 generates a unique, fixed-length hash value that cannot be reversed to obtain the original message, making it ideal for verifying data integrity. For example, if even a single character of the input changes, the resulting hash will be completely different.

Implementing this cipher helps defend against common threats such as data tampering, unauthorized access, and man-in-the-middle attacks. Using SHA-256 also aligns with current cybersecurity best practices, as it is part of the SHA-2 family approved by NIST (National Institute of Standards and Technology) and commonly used in SSL certificates and digital signatures.

## Certificate Generation

Insert a screenshot below of the CER file.

For this step, I generated a self-signed SSL certificate using the Java Keytool utility. I created a keystore (keystore.jks) and an SSL certificate (sslserver.cer) with the RSA encryption algorithm and 2048-bit key size. After successfully exporting the certificate, I verified its details with the keytool -printcert -file sslserver.cer command. The output confirmed that the certificate was valid, self-signed for localhost, and used the SHA256withRSA signature algorithm. This certificate will be used to enable secure HTTPS communication within the Artemis Financial application.

A screenshot of a computer

AI-generated content may be incorrect.

## 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.

To secure the application’s communication, I switched from HTTP to HTTPS by creating a self-signed SSL certificate and configuring the server to use it. This means all data being sent between the browser and the server is now encrypted, which helps keep it private and protected from interception. Even though the browser warns that the certificate is “not trusted,” the connection is still secure because the encryption is working as intended.

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.

A white background with black and white clouds

AI-generated content may be incorrect.

## Functional Testing

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

A screenshot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

## Summary

The project successfully implemented secure software development principles using Java and Spring Boot to create an SSL server that performs checksum verification using the SHA-256 hashing algorithm. The application was tested through functional and security testing to ensure data integrity and protection. All primary objectives were achieved — including creating a secure RESTful endpoint, implementing HTTPS with a self-signed SSL certificate, and performing a dependency vulnerability check using OWASP Dependency-Check. The results demonstrated that the server can securely transmit and verify data without exposing sensitive information. Any detected vulnerabilities were reviewed, and no critical issues were found in the project dependencies. This confirms that the system meets its security, functionality, and reliability goals according to the project requirements.

## Industry Standard Best Practices

To maintain and improve the security and reliability of this application, several industry-standard best practices were followed and are recommended for ongoing maintenance:

* **Use of Secure Hashing Algorithms:** The SHA-256 algorithm was used to ensure data integrity, following NIST-recommended cryptographic standards.
* **SSL/TLS Encryption:** All data transmissions occur over HTTPS using a self-signed certificate, which encrypts communication between client and server to prevent man-in-the-middle attacks.
* **Regular Dependency Scanning:** OWASP Dependency-Check was used to identify known vulnerabilities in third-party libraries. In production, automated scans should run during each build or deployment pipeline.
* **Principle of Least Privilege:** The server was configured to run with minimal permission necessary to perform its functions, reducing the risk of exploitation.
* **Secure Coding Practices:** Input validation, error handling, and code refactoring were performed to prevent injection and logic-based vulnerabilities.
* **Version Control and Documentation:** All changes were documented in version control (e.g., Git) to support traceability, maintainability, and collaboration across secure development environments.
* **Future Enhancements:** In a production setting, the use of trusted SSL certificates issued by a Certificate Authority (CA) and implementation of continuous security monitoring are recommended to ensure compliance with modern cybersecurity standards.

These practices align with OWASP, NIST SP 800-53, and ISO/IEC 27001 standards for secure software engineering. Following them ensures that the application remains resilient against emerging threats while maintaining data integrity and confidentiality.

OWASP Foundation. (2024). *OWASP Top 10: Common software vulnerabilities*. https://owasp.org/www-project-top-ten/

National Institute of Standards and Technology. (2020). *Security and Privacy Controls for Information Systems and Organizations (NIST SP 800-53 Revision 5).* U.S. Department of Commerce. https://doi.org/10.6028/NIST.SP.800-53r5

National Institute of Standards and Technology. (2015). *Secure Hash Standard (SHS) (FIPS PUB 180-4).* U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.180-4

International Organization for Standardization. (2018). *ISO/IEC 27001: Information security management systems — Requirements.* ISO/IEC.