****

# 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** | **04/16/2025** | **Eddy Kwon** | Initial submission |

## 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

Eddy Kwon

## Algorithm Cipher

For Artemis Financial’s secure communication and data verification requirements, I recommend implementing the **SHA-256** hash algorithm in conjunction with **TLS (Transport Layer Security)** for encryption during data transmission. This combination provides both **integrity** and **confidentiality**, aligning with industry best practices for securing financial information.

**Overview**

SHA-256 is a cryptographic hash function belonging to the **SHA-2** family, developed by the National Security Agency (NSA) and standardized by NIST. It generates a **fixed 256-bit** (32-byte) hash value for any given input. SHA-256 is widely adopted in various industries including banking, cybersecurity, and blockchain, due to its **resistance to collision and pre-image attacks**.

**Hash Functions and Bit Levels**

SHA-256 processes data and returns a 64-character hexadecimal hash, representing a 256-bit value. This level of complexity and length offers excellent protection against brute-force attacks. It’s ideal for checksum validation to ensure that files have not been tampered with during transmission, as even a one-character change in the input data produces a dramatically different hash.

**Use of Random Numbers and Key Types**

Since SHA-256 is a hash function, it does **not use keys** like traditional encryption methods. For encryption and secure communication, **TLS** is used, which combines:

* **Asymmetric encryption** (public/private key pairs) during the handshake process to establish a secure connection.
* **Symmetric encryption** for efficient data transfer during the session.

TLS also relies on **random number generation** to create secure session keys, helping defend against replay and man-in-the-middle attacks.

**History and Current State of Encryption Algorithms**

Hash functions like SHA-1 and MD5 were widely used in the early 2000s but have since been deprecated due to vulnerabilities. SHA-256, introduced in 2001, has become the modern standard. It is currently **approved by NIST** and recommended by organizations like the **OWASP Foundation** and **ISO** for critical systems. It remains widely supported and trusted for applications requiring strong data integrity verification, including SSL/TLS, digital signatures, and blockchain-based platforms.

## Certificate Generation

Insert a screenshot below of the CER file.

A screenshot of a computer

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A close-up of a computer screen

AI-generated content may be incorrect.

## Secure Communications

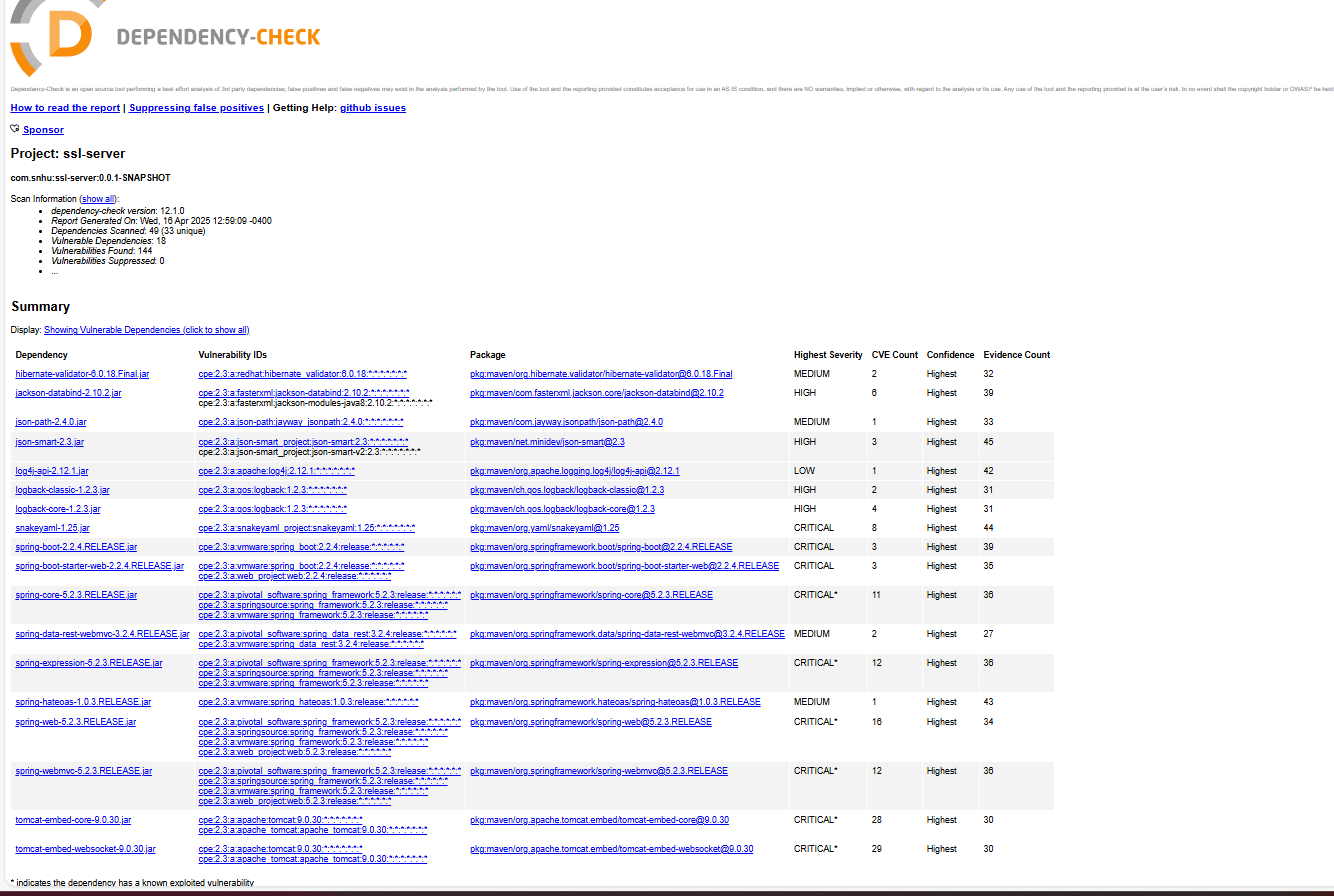
Insert a screenshot below of the web browser that shows a secure webpage.

A close-up of a computer screen

AI-generated content may be incorrect.

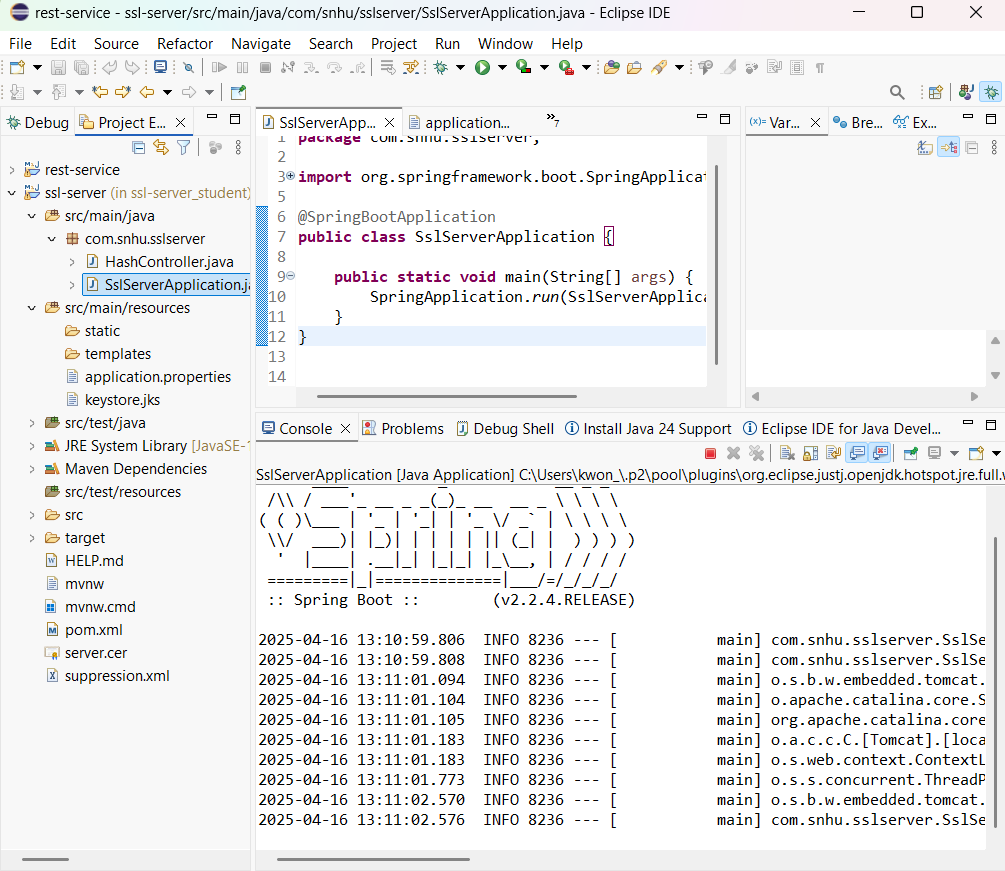
## Secondary Testing

Insert screenshots below of the refactored code executed without errors and the dependency-check report.



## Functional Testing

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



## Summary

In this project, we successfully implemented a secure server using SSL in Spring Boot and Java. The goal was to configure a server that could securely communicate using HTTPS and generate a checksum for the data. The key components included:

* **SSL Configuration**: We configured the server to use HTTPS with a self-signed certificate, ensuring secure communications.
* **Checksum Implementation**: We implemented the SHA-256 algorithm to generate a checksum for a sample input, "Hello Eddy Kwon!".
* **Testing**: The application was tested using Spring Boot's embedded Tomcat server, successfully validating the checksum functionality.
* **Refactoring**: The code was refactored for clarity and functionality, allowing for easy integration of future security features or enhancements.

Throughout the process, we faced some challenges, such as properly configuring the keystore and addressing issues related to SSL certificate paths. However, these were successfully resolved by ensuring the keystore was properly configured and accessible within the project.

## Industry Standard Best Practices

When developing a secure web application, especially in the context of encryption and secure communications, several best practices should be followed:

* **Use Strong Encryption**: Always use strong encryption algorithms like SHA-256 or AES to protect sensitive data. Avoid using weaker algorithms or insecure cipher suites.
* **Proper SSL Certificate Management**: Use trusted certificate authorities (CA) for production environments. For testing purposes, self-signed certificates can be used, but they should never be used in production.
* **Regularly Update Dependencies**: Ensure that your libraries and dependencies are up-to-date to prevent vulnerabilities caused by outdated components. Tools like Dependency-Check can help identify known security issues in dependencies.
* **Secure Configuration**: Avoid hardcoding sensitive data like passwords or keys directly into the source code. Use environment variables or configuration management systems to securely store this data.
* **Proper Authentication and Authorization**: For production systems, ensure that authentication (e.g., OAuth2) and authorization are implemented, ensuring only authorized users can access sensitive resources.
* **Monitor and Log Security Events**: Set up proper logging and monitoring to track any suspicious activities or security breaches. Tools like the ELK Stack or Prometheus can help monitor logs and performance metrics in real time.
* **Testing and Auditing**: Regularly audit your security settings and test your application for vulnerabilities using penetration testing and automated security scanning tools.

By adhering to these industry standards and best practices, you can ensure the security and integrity of your web application, protecting both data and users.