****

# 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** | **[Date]** | **[Your Name]** |  |

## Client



## Developer

Shane Edge

## Algorithm Cipher

Overview:

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function from the SHA-2 family, designed by the NSA. It produces a fixed-length 256-bit (32-byte) output, regardless of input size. It’s deterministic, one-way, and collision-resistant, which means it’s computationally infeasible to reverse-engineer the input or find two inputs that hash to the same output.

Hash Functions and Bit Levels:

SHA-256 uses a 256-bit output, which makes it resilient against brute-force attacks and preimage attacks. Compared to older algorithms like MD5 (128-bit) or SHA-1 (160-bit), SHA-256 provides significantly stronger protection against collisions and is still widely trusted in financial, governmental, and commercial environments.

Use of Random Numbers and Key Types:

SHA-256 itself is a hash function, not an encryption algorithm, so it doesn’t use keys or random numbers in the same way symmetric/asymmetric encryption does. However, if encryption is later required, it can be paired with algorithms like AES (symmetric) for confidentiality, or RSA (asymmetric) for secure key exchange.

History and Current State:

Earlier hash algorithms like MD5 and SHA-1 have been deprecated due to vulnerabilities. SHA-256, introduced in 2001, is still considered secure and widely used in TLS/SSL, blockchain systems (e.g., Bitcoin), and file integrity checks. It’s part of the U.S. FIPS 180-4 standard and recommended by NIST, making it a trusted choice for financial data.

## Certificate Generation

Insert a screenshot below of the CER file.

A screenshot of a certificate

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

Shown in step 4 screenshot.

## Secure Communications

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

A yellow box with black numbers and a yellow circle

AI-generated content may be incorrect.

## Secondary Testing

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

A close-up of a text

AI-generated content may be incorrect.

A screenshot of a computer error

AI-generated content may be incorrect.

## Functional Testing

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

A close-up of a text

AI-generated content may be incorrect.

## Summary

Cryptography: Integrated SHA-256 as the cryptographic hashing algorithm for consistent, tamper-evident checksum generation.

Client/Server Communication: Implemented HTTPS with a self-signed SSL certificate to encrypt traffic between the client and the server, preventing eavesdropping and man-in-the-middle attacks.

Encapsulation: Created a dedicated controller class to handle hashing logic, keeping business logic separate from application startup code.

To harden the application:

I generated a secure keystore with the keytool utility and configured it for TLS encryption via the Spring Boot application.properties file.

I modularized the checksum feature using a controller class and used SHA-256 to ensure message integrity.

I ran a dependency vulnerability scan using the OWASP Dependency-Check Maven plugin to identify and address third-party library risks.

These steps added defense-in-depth to the system, ensuring both application-level and transport-level security.

## Industry Standard Best Practices

To ensure the SSL Server complies with industry standards for secure coding, I applied best practices recommended by organizations such as OWASP and NIST:

Maintaining Security Using Best Practices

SHA-256 hashing was used instead of weaker or obsolete algorithms (like MD5 or SHA-1).

HTTPS/TLS setup followed secure transport guidelines by enabling encrypted communication and requiring proper keystore configuration.

Spring Boot dependency management was evaluated with OWASP Dependency-Check, identifying any CVEs tied to third-party libraries.

No hardcoded sensitive information (e.g., passwords) was committed to source files, following secure credential handling practices.

Value to the Company

Applying these practices:

Reduces risk of breaches or data exposure, avoiding costly fines or customer distrust.

Supports regulatory compliance (like PCI DSS or HIPAA for encrypted communication).

Increases software resilience and long-term maintainability by reducing technical debt.

Builds trust with end users and stakeholders by demonstrating a commitment to security from the ground up.