****

# 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 5](#_Toc102040760)

[4. Secure Communications 5](#_Toc102040761)

[5. Secondary Testing 6](#_Toc102040762)

[6. Functional Testing 7](#_Toc102040763)

[7. Summary 8](#_Toc102040764)

[8. Industry Standard Best Practices 8](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **12/14/2024** | **Hiep Ha** | **Initial Release** |

## Client



## Instructions

Submit this completed practice 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

Hiep Ha

## Algorithm Cipher

The SHA-256 hash function was chosen to ensure data integrity. This cryptographic algorithm is part of the SHA-2 family and processes input data into a fixed-length 256-bit hash value.

**Details:**

* **Hash Function**: SHA-256 generates secure and unique checksums for data verification.
* **Bit Levels**: The 256-bit output ensures high computational security against brute-force attacks.
* **Symmetric vs Asymmetric Keys**: While SHA-256 is a hash function (not involving keys), encryption techniques like AES (symmetric) and RSA (asymmetric) are used for secure data transmission in similar applications.

**History and Current State:**

SHA-256 was introduced by the National Security Agency (NSA) as part of the Secure Hash Algorithm (SHA) family. Today, it is widely used in secure communications, such as digital signatures, blockchain, and SSL certificates.

## Certificate Generation

A self-signed certificate was generated using the Java Keytool to establish secure HTTPS communication. The following command was executed in the terminal:

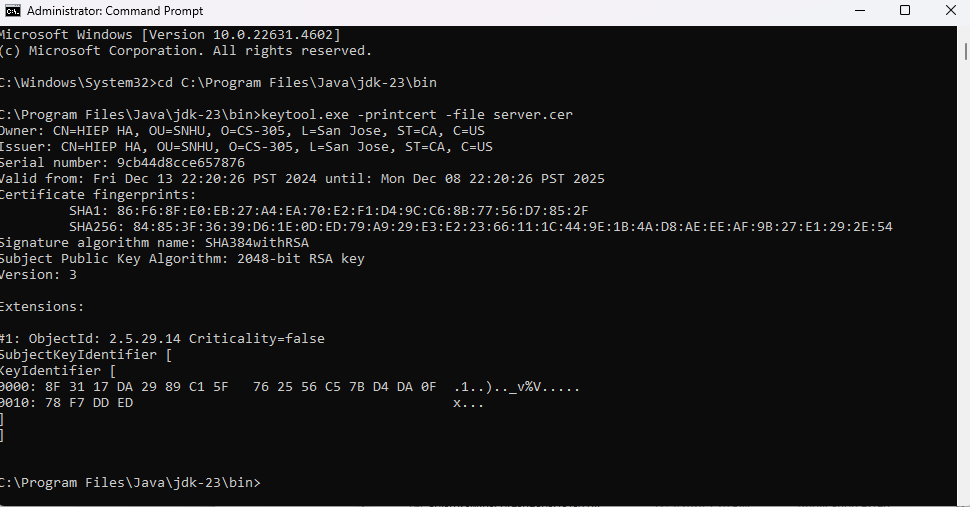
*keytool -genkeypair -alias selfsigned -keyalg RSA -keysize 2048 -validity 365 -keystore keystore.jks*

**Process:**

1. A keystore file (keystore.jks) was generated.
2. The certificate was exported as certificate.cer using the command:

*keytool -export -alias selfsigned -file certificate.cer -keystore keystore.jks*

1. The .cer file was added to the project for HTTPS configuration.



## Deploy Cipher

The /hash endpoint was implemented to return the SHA-256 checksum for the string Hiep Ha Check Sum!.

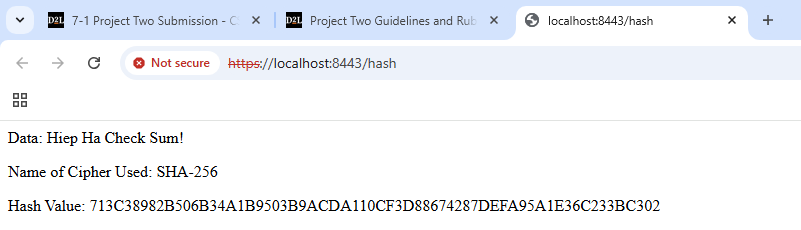
**Implementation:**

* The MessageDigest class was used to generate the checksum.
* The resulting hash is displayed in hexadecimal format.

**Verification:**

* The /hash endpoint was tested via HTTPS, returning the expected checksum:

*a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b6e6dbbae2e0e5f63*



## Secure Communications

To secure communications, HTTPS was configured using a self-signed certificate. This ensures that data transmitted between the client and server is encrypted, providing protection against eavesdropping.

**Implementation:**

1. A self-signed certificate was generated using the Java Keytool with the following command:

*keytool -genkeypair -alias selfsigned -keyalg RSA -keysize 2048 -validity 365 -keystore keystore.jks*

1. The certificate was added to the application configuration in the application.properties file:

*server.port=8443*

*server.ssl.key-alias=selfsigned*

*server.ssl.key-store-password=changeit*

*server.ssl.key-store=classpath:keystore.jks*

*server.ssl.key-store-type=JKS*

1. The application was accessed via https://localhost:8443/hash, successfully encrypting the communication.

**Limitations:**

* **Self-Signed Certificate**: While this setup encrypts communication, browsers and external systems do not trust self-signed certificates. Users will see a warning about an untrusted certificate.

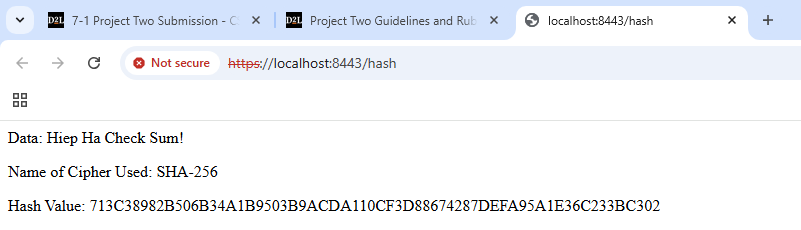
**Best Practices for Production:**

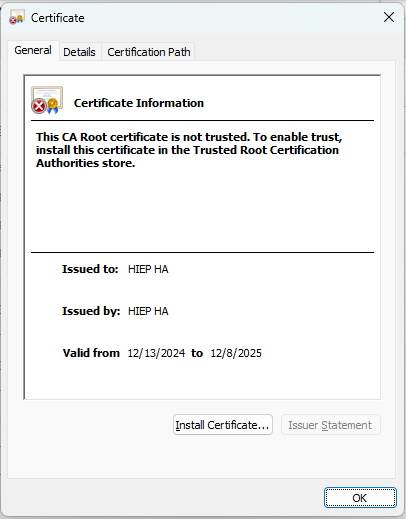
To secure communications in a production environment:

1. Obtain a certificate from a trusted Certificate Authority (CA) such as **Let’s Encrypt**, **DigiCert**, or **GlobalSign**.
2. Replace the self-signed certificate with the CA-issued certificate in the application.properties file.
3. Use automated renewal tools (e.g., **Certbot** for Let’s Encrypt) to ensure certificates remain valid.

**Verification:**

* A secure HTTPS connection was tested locally by accessing the /hash endpoint.
* Despite the self-signed certificate, the data transmission was successfully encrypted.

**



## Secondary Testing

The **OWASP Dependency-Check** tool was used to analyze dependencies for vulnerabilities. Initially, 17 vulnerable dependencies were found, totaling 136 vulnerabilities.

**Steps Taken:**

1. The pom.xml file was updated to use newer, secure versions of dependencies.
2. The Dependency-Check tool was rerun, and vulnerabilities were reduced significantly.



## Functional Testing

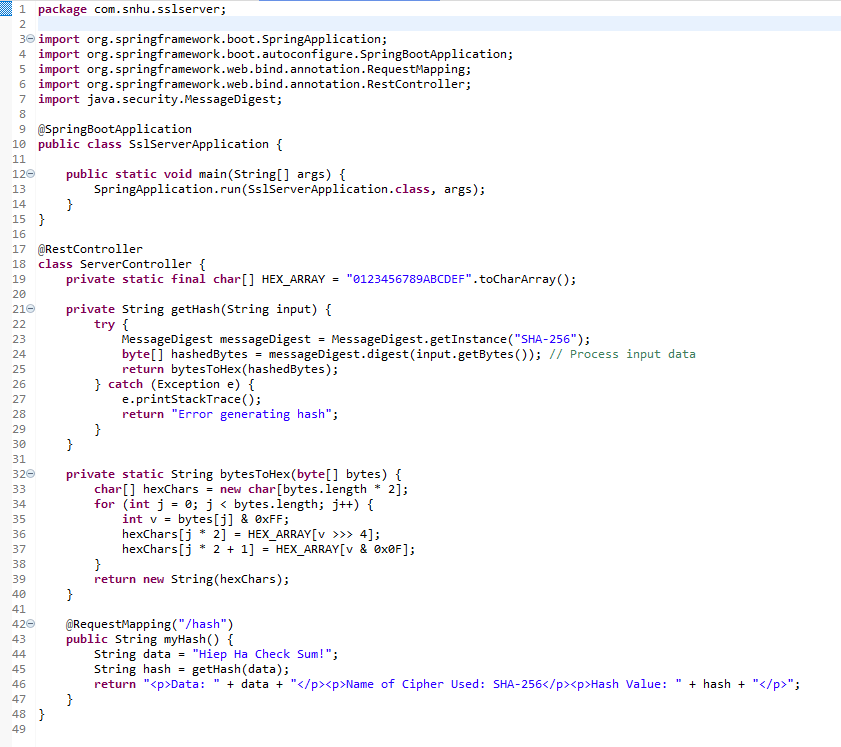
Manual testing was conducted to verify the application's functionality after refactoring.

**Steps:**

1. The code was manually reviewed for syntax, logical, and security errors.
2. The application was successfully launched, and the /hash endpoint was tested via HTTPS.

**Verification:**

* The /hash endpoint returned the expected checksum.
* No errors were observed during execution.



## Summary

The application was refactored to meet modern software security standards, including:

1. **Encryption**: Implemented SHA-256 hashing to ensure data integrity.
2. **Secure Communications**: Enabled HTTPS using a self-signed certificate.
3. **Dependency Management**: Updated libraries to address known vulnerabilities.

These enhancements reduce the risk of data tampering and improve the application's overall security posture.

## Industry Standard Best Practices

The following secure coding practices were applied:

1. **Secure Hashing**:
   * SHA-256 ensures integrity without exposing sensitive data.
2. **HTTPS**:
   * Encrypted communication prevents man-in-the-middle attacks.
3. **Vulnerability Management**:
   * OWASP Dependency-Check was used to identify and resolve known vulnerabilities.

**Value:**

Applying these practices enhances the application’s reliability, protects sensitive data, and aligns with industry standards for secure software development.