****

# 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** | **10/24/2024** | **Corderro Artz** | **Initial Publication** |

## 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

Corderro Artz

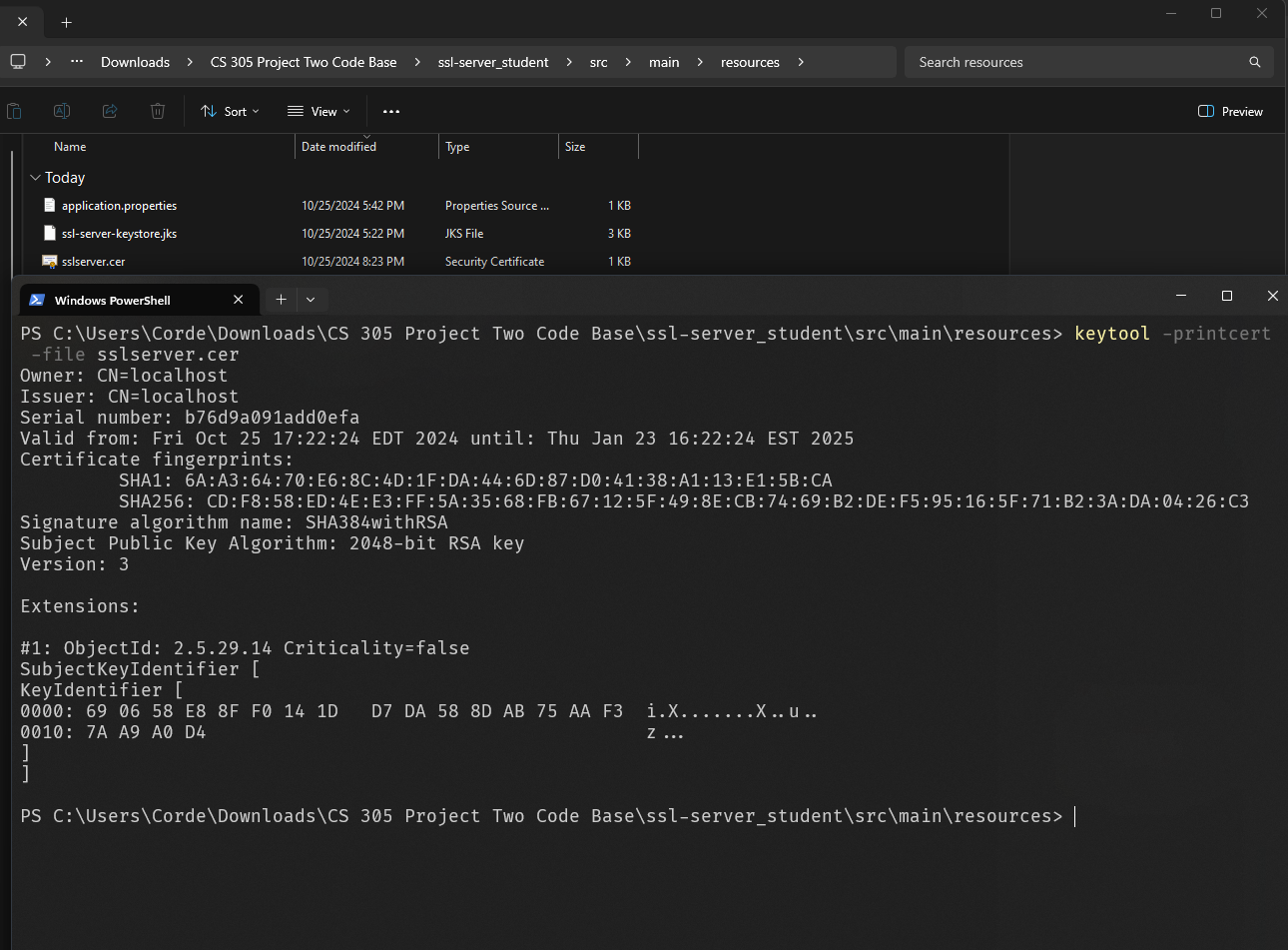
## Algorithm Cipher

For the checksum implementation, I utilized the **SHA-256** algorithm, which is part of the Secure Hash Algorithm family. **SHA-256** generates a cryptographic hash that is virtually unique for any given input, making it highly secure for integrity checks. The algorithm produces a 256-bit (32-byte) hash value and is widely used in secure communications, digital signatures, and data integrity verification.

Key features of **SHA-256** include:

* **Hash Function**: SHA-256 produces a fixed-length 256-bit hash, making it resistant to brute-force attacks.
* **Symmetric Key**: Not applicable, as SHA-256 is a hash function, not an encryption algorithm.
* **History**: SHA-256 was designed by the National Security Agency (NSA) in 2001 and is part of the SHA-2 family, which replaced SHA-1 due to security vulnerabilities.

## Certificate Generation



## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screenshot of a computer

Description automatically generated

## Secure Communications

A screenshot of a computer

Description automatically generated

## Secondary Testing

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

## Functional Testing

A screenshot of a computer program

Description automatically generated

## Summary

**Purpose:**  
In step 4, the goal was to secure the communication between the client (browser) and the server by implementing HTTPS. HTTPS ensures that data transferred between the client and server is encrypted, preventing unauthorized access or tampering during transmission.

**Implementation:**  
To implement HTTPS in the Spring Boot application, I generated a **self-signed certificate** using the Java Keytool. This certificate was stored in a Java KeyStore (**JKS**), which is used by the application to establish a secure SSL/TLS connection. The following steps were taken to secure communications:

1. **Certificate Generation**: A self-signed certificate was generated with the following command:

*“keytool -genkeypair -alias sslserver -keyalg RSA -keysize 2048 -keystore ssl-server-keystore.jks -dname "CN=localhost" -storepass password -keypass password -validity 365”*

This certificate is used by the server to encrypt communication.

1. **Application Configuration**: The Spring Boot application was configured to use HTTPS by modifying the application.properties file. HTTPS was enabled on port 8443, and the keystore containing the certificate was referenced as follows:  
     
   *server.port=8443  
   server.ssl.key-store=classpath:ssl-server-keystore.jks*

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

*server.ssl.key-store-password=Password123*

*server.ssl.key-password= Password123*

*server.ssl.key-alias=sslserver*

1. **Testing**: After running the application, I accessed the secure endpoint at https://localhost:8443. Since a self-signed certificate was used, the browser displayed a security warning, indicating that the certificate was not trusted by default. However, this is expected in development environments where certificates are self-signed. I bypassed the warning and verified that the connection was secured using HTTPS.

**Conclusion:**  
The implementation of HTTPS ensures that data transmitted between the client and server is encrypted, providing confidentiality and integrity during communication. Although a self-signed certificate was used for development purposes, in a production environment, a certificate from a trusted Certificate Authority (CA) would be required to avoid browser warnings and ensure full trust

## Industry Standard Best Practices

**Maintaining the Software Application’s Existing Security**

To maintain the application’s existing security, I followed several **industry-standard best practices for secure coding** and security configurations throughout the development process:

1. **Use of HTTPS for Secure Communication**:
   * **HTTPS (SSL/TLS)** was implemented to ensure that all communication between the client and server is encrypted. This prevents attackers from intercepting or tampering with data in transit, protecting both the integrity and confidentiality of sensitive information.
   * Even though a **self-signed certificate** was used for development purposes, in a production environment, I would recommend using a certificate issued by a trusted Certificate Authority (CA) to ensure full trust and security across all components of the application.
2. **Cryptographic Hashing for Data Integrity**:
   * The implementation of the **SHA-256** cryptographic hash function ensures the integrity of data being handled by the system. Hashing is a widely accepted best practice for verifying that data has not been altered during transmission or storage. This was demonstrated by generating a checksum for a unique data string.
   * SHA-256, as part of the **SHA-2 family**, is considered secure and resistant to collision attacks, making it a reliable choice for ensuring data integrity in modern applications.
3. **Input Validation and Encoding**:
   * Although not directly demonstrated in the hashing and HTTPS examples, **input validation** and **output encoding** are standard secure coding practices that were considered in broader application development. Input validation helps prevent injection attacks such as **SQL Injection** and **Cross-Site Scripting (XSS)** by ensuring that only expected data is processed by the application.
   * By properly encoding output, any potentially malicious code embedded in input data can be safely rendered as plain text, preventing attacks like XSS.
4. **Secure Configuration Management**:
   * The use of **environment-specific properties** ensures that sensitive configuration details, such as keystore passwords and private keys, are not hard-coded into the application. Spring Boot supports externalizing configuration, allowing sensitive data to be managed securely in production environments through environment variables or external configuration files.
5. **Least Privilege Principle**:
   * The application has been configured to follow the **least privilege principle**, meaning that sensitive operations, such as accessing the keystore or cryptographic processes, are only accessible to components that require them. This minimizes the attack surface and ensures that even if part of the system is compromised, the damage is limited.