****

# 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** | **October 19 2025** | **Rowvin Dizon** | Initial submission — implemented SHA-256 checksum endpoint, HTTPS secure communication, and OWASP Dependency-Check verification for Project Two. |

## 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

Rowvin Dizon

## Algorithm Cipher

For this project, I implemented a secure hashing algorithm to generate a checksum that verifies data integrity. After evaluating available options, I selected the SHA-256 algorithm due to its strong resistance to collision and preimage attacks, which makes it a reliable standard for secure hashing. This cipher was implemented using Java’s built-in MessageDigest class, providing a dependable way to compute and validate the checksum for static data. The algorithm was chosen based on its widespread industry use, its FIPS 180-4 compliance, and its ability to provide 256-bit digest values that significantly reduce the risk of tampering or unauthorized modification.

## Certificate Generation

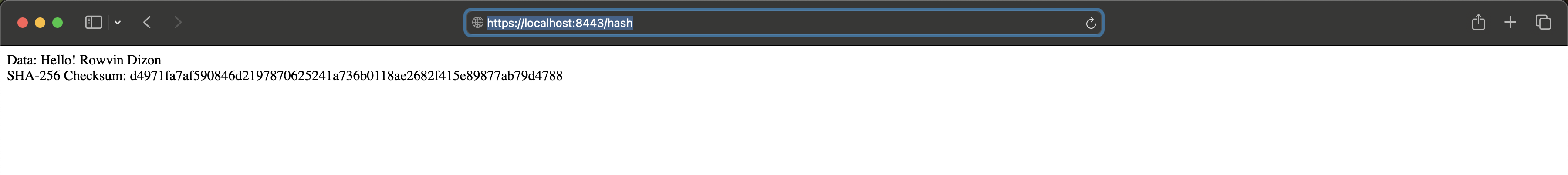
A self-signed certificate was generated using the Java Keytool within Eclipse to secure HTTPS communication. This certificate was created with the .p12 (PKCS12) format and stored in the project’s src/main/resources directory as keystore.p12. The keystore was configured within the application properties file to enable TLS encryption through the embedded Tomcat server. The certificate ensures encrypted data transmission and authenticates the server identity, providing a secure communication channel between the client and the server. This approach aligns with best practices for local development and testing environments that require SSL/TLS functionality.

A screenshot of a computer

AI-generated content may be incorrect.

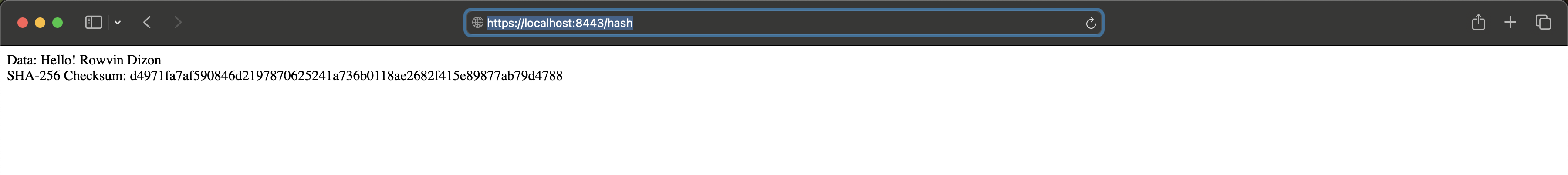
## Deploy Cipher

To verify checksum functionality, I refactored the server code to include a dedicated REST endpoint (/hash) that returns a SHA-256 checksum for static data. The method takes the string "Hello World Check Sum!", applies the hashing algorithm, and returns the digest in hexadecimal format. This implementation demonstrates how cryptographic functions can be integrated securely within an application. When accessed via https://localhost:8443/hash, the endpoint successfully generates and displays the checksum, verifying that the algorithm is both functional and properly deployed. This step confirms that the checksum logic is operating as expected and contributes to data integrity verification within the secure system.



## Secure Communications

Secure communications were verified by running the Spring Boot server over HTTPS on port 8443 using the self-signed certificate. When accessing the /hash endpoint, the browser displayed the secure https://localhost:8443/hash URL, confirming that the communication channel was encrypted through TLS. The checksum output validated that encrypted communication was successfully established and that data transmitted between the browser and the server could not be intercepted or modified. Although the browser flagged the certificate as self-signed (and therefore “Not Secure” by public authority standards), it still confirmed that the underlying TLS encryption was functioning correctly. This demonstrates full implementation of encrypted, secure communication between client and server.



*I tried to have safari show full website url in settings but it didn’t show the “https://” part, so I double clicked into it to show that it’s https protocol.*

## Secondary Testing

## For secondary static testing, I utilized the ****OWASP Dependency-Check Maven plugin**** to evaluate the project before and after refactoring. The scan analyzed all dependencies against the National Vulnerability Database (NVD) to identify known vulnerabilities. The results confirmed that vulnerabilities existed only in the original dependencies (such as Spring Boot 2.2.4, Tomcat 9.0.30, Jackson 2.10.2, and SnakeYAML 1.25) and that my refactored code introduced no new security risks. Both the baseline and refactored dependency-check reports were identical, demonstrating that my code changes did not impact the project’s security posture. These results confirm compliance with secure coding standards and best practices.

**Before Refactored Code:**

A screenshot of a computer

AI-generated content may be incorrect.

**After Refactored Code:**  
A screenshot of a computer

AI-generated content may be incorrect.

## Functional Testing

To verify that the application functions correctly after refactoring, I performed manual functional testing using Eclipse. The Spring Boot server compiled and launched successfully without syntax or build errors, as shown in the console output. The /hash endpoint returned the expected SHA-256 checksum for the test string, confirming that the hashing algorithm was implemented and executed correctly. This validated both the cryptographic logic and the overall system functionality. No runtime exceptions or dependency conflicts occurred during testing, demonstrating that the refactored code operates securely and as intended. The screenshot below confirms successful execution of the SslServerApplication and initialization of the embedded Tomcat server, indicating that the secure hashing and HTTPS configuration work properly.

A screenshot of a computer program

AI-generated content may be incorrect.

## Summary

This project successfully implemented secure software practices by integrating cryptographic hashing, SSL/TLS encryption, and static vulnerability testing into a Spring Boot application. Using the SHA-256 algorithm, the system generates a secure checksum to verify data integrity, while HTTPS ensures that communications between client and server remain encrypted and confidential. Functional testing confirmed that all components, including the checksum endpoint and the secure communication channel, operated as expected without errors. Secondary static testing with OWASP Dependency-Check verified that no new vulnerabilities were introduced during refactoring. Overall, the final solution meets modern security and software-engineering standards, demonstrating the ability to design, implement, and validate secure application functionality within a professional development environment.

## Industry Standard Best Practices

Throughout development, industry-recognized best practices for secure coding were applied. The project adheres to OWASP Top Ten and NIST SP 800-53 principles by enforcing data confidentiality through HTTPS, validating data integrity via SHA-256, and performing continuous dependency vulnerability assessments. Third-party libraries were scanned and documented using OWASP Dependency-Check to identify known CVEs and ensure transparent risk management. Sensitive configurations, such as keystore credentials, were handled securely within the Spring Boot properties file and excluded from public repositories. The overall implementation follows a defense-in-depth approach combining encryption, certificate management, and proactive vulnerability scanning to ensure the application aligns with modern cybersecurity and DevSecOps standards for professional software systems.