****

# Practices for Secure Software Report

**Table of Contents**

[**Document Revision History 3**](#_heading=h.p4qp96uf4mvu)

[**Client 3**](#_heading=h.ijp6ggi3uz5x)

[**Instructions 3**](#_heading=h.60cvn4xxp6hw)

[**Developer 4**](#_heading=h.wrf36eq818uj)

[**1. Algorithm Cipher 4**](#_heading=h.v85bfjlx4jhc)

[**2. Certificate Generation 4**](#_heading=h.og7aoxu86m0z)

[**3. Deploy Cipher 4**](#_heading=h.dh1pq7typmpr)

[**4. Secure Communications 4**](#_heading=h.72w751iwpeq2)

[**5. Secondary Testing 4**](#_heading=h.mdm979i3buen)

[**6. Functional Testing 4**](#_heading=h.oow862r3xucq)

[**7. Summary 4**](#_heading=h.cvk0d6t4lw4)

[**8. Industry Standard Best Practices 4**](#_heading=h.6cyxu9kqw13v)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **14 August, 2025** | **Taylor Colton** | **Initial Report** |

## 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

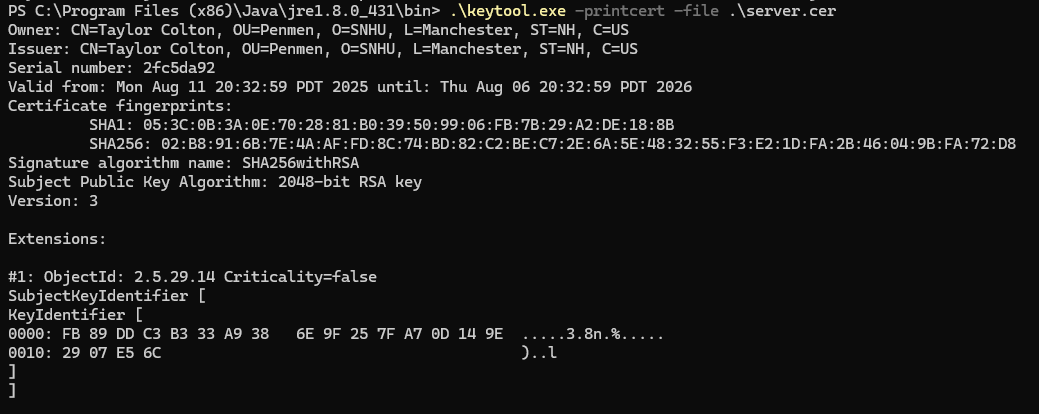
Taylor Colton

## Algorithm Cipher

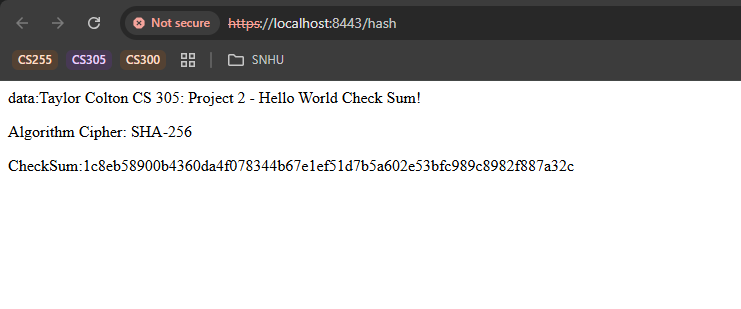
I suggest HTTPS/TLS for securing data in transit and AES-256 for sensitive data at rest for this project. AES-256 is a popular, NIST-recognized symmetric cipher with a 256-bit key that satisfies standard regulatory requirements for financial data. It is suitable for long-term archival security, quick, and extensively supported in Java. I use SHA-256 in conjunction with encryption for integrity checks as it generates a 256-bit digest and is highly effective at recognizing corruption or manipulation, which is precisely what my checksum endpoint depicts.

Simply put, AES-256 is effective for large quantities of data because it employs the same key for both encryption and decryption (symmetric); asymmetric cryptography, such as RSA, can be used in conjunction with AES-256 to exchange or secure the symmetric key as necessary. Best practice also entails employing random initialization vectors (IVs) and securely maintaining keys (rotation and protected storage) to ensure that ciphertexts stay unique even when encrypting identical material. AES is still the most prevalent option when it comes to file protection today, having surpassed DES and 3DES as the standard for robust, effective encryption. Meanwhile, SHA-256 continues to be a collision-resistant hash for integrity verification.

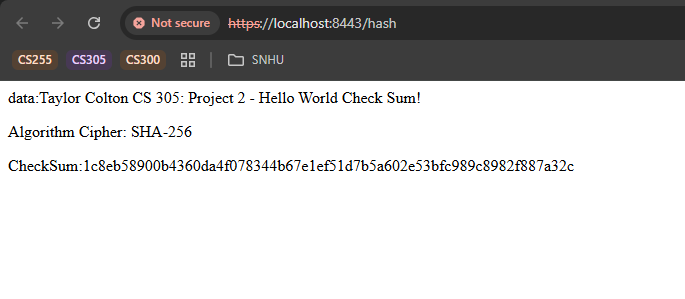
## Certificate Generation



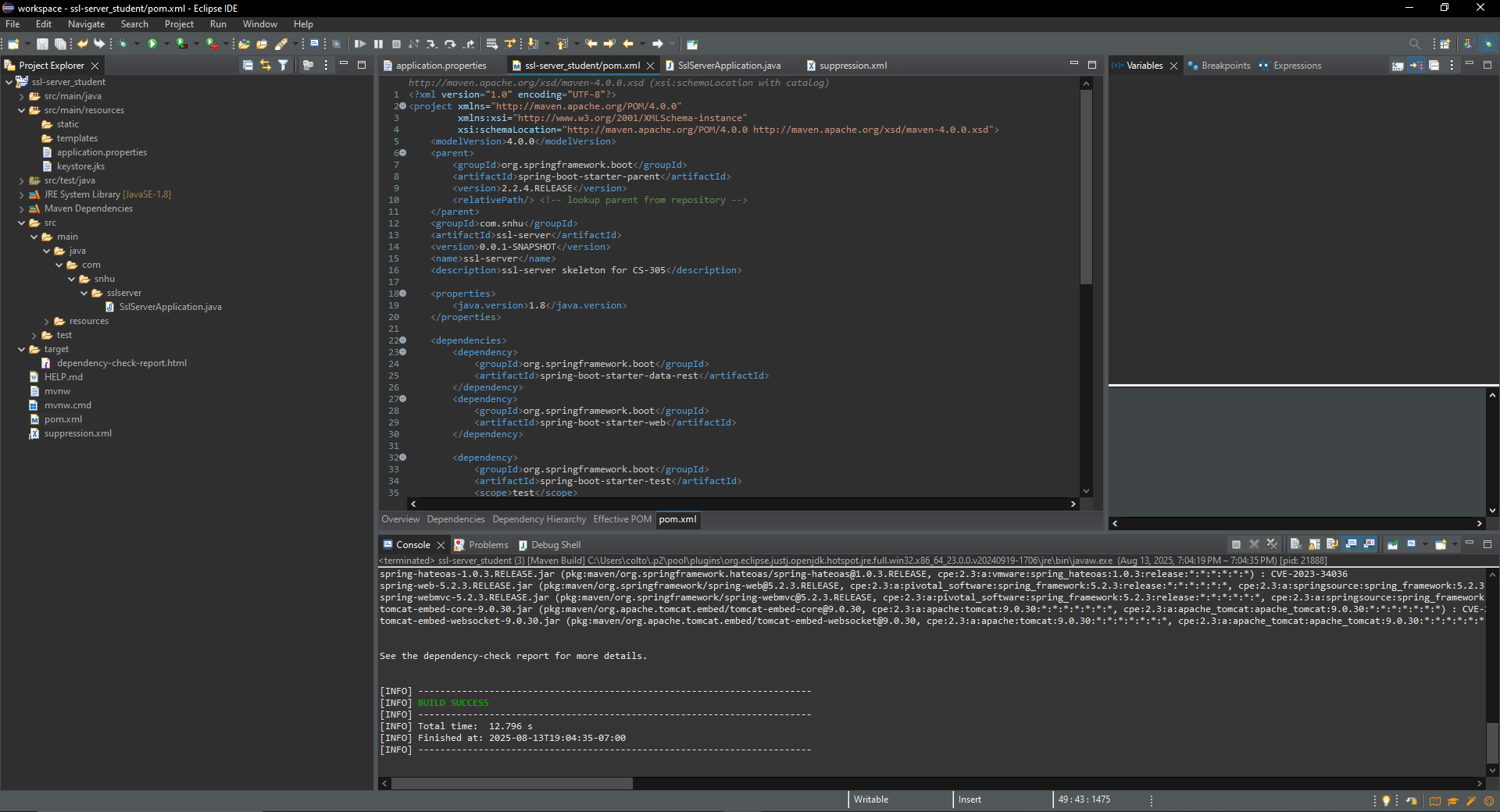
## Deploy Cipher

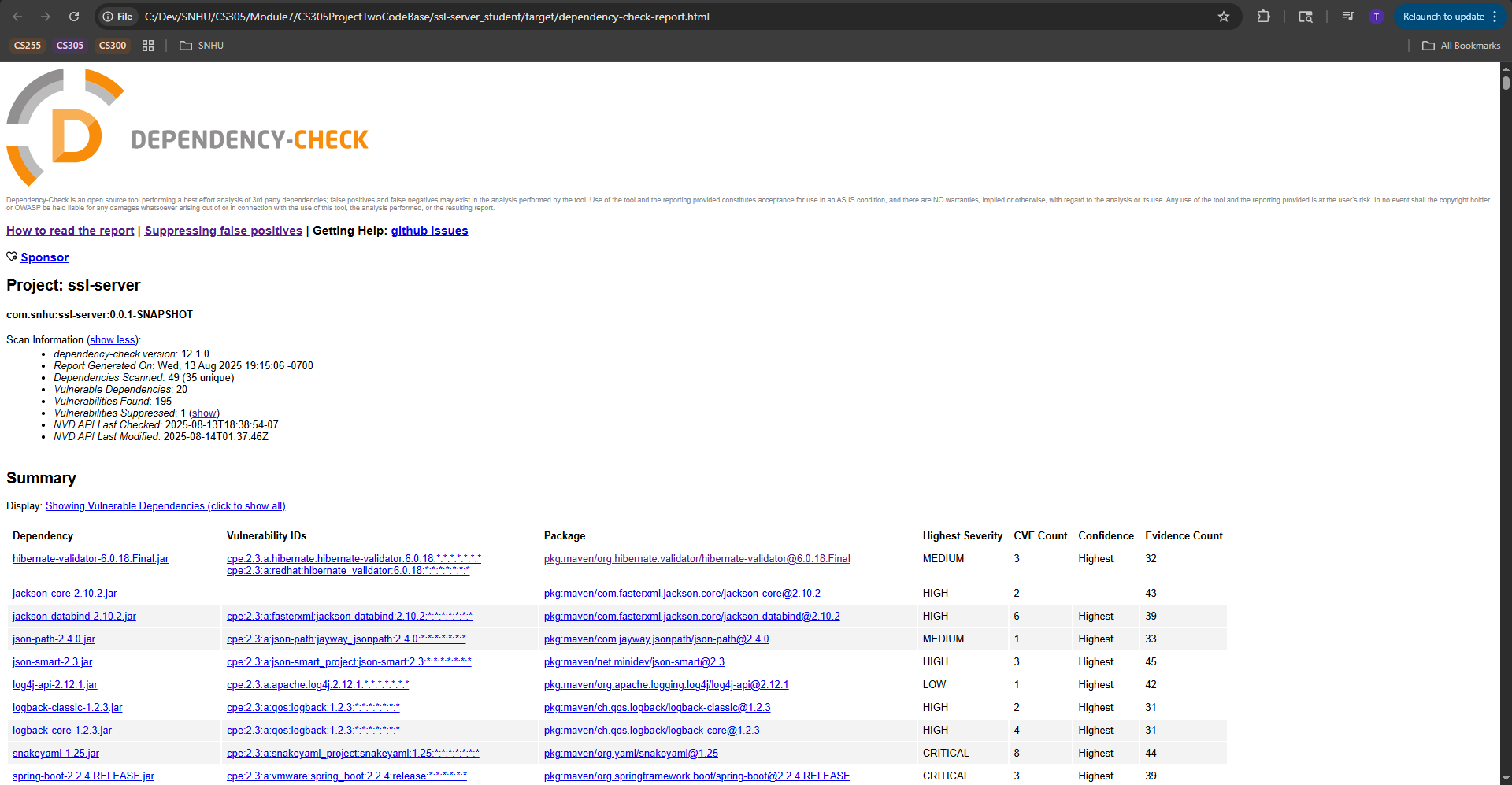


## Secure Communications

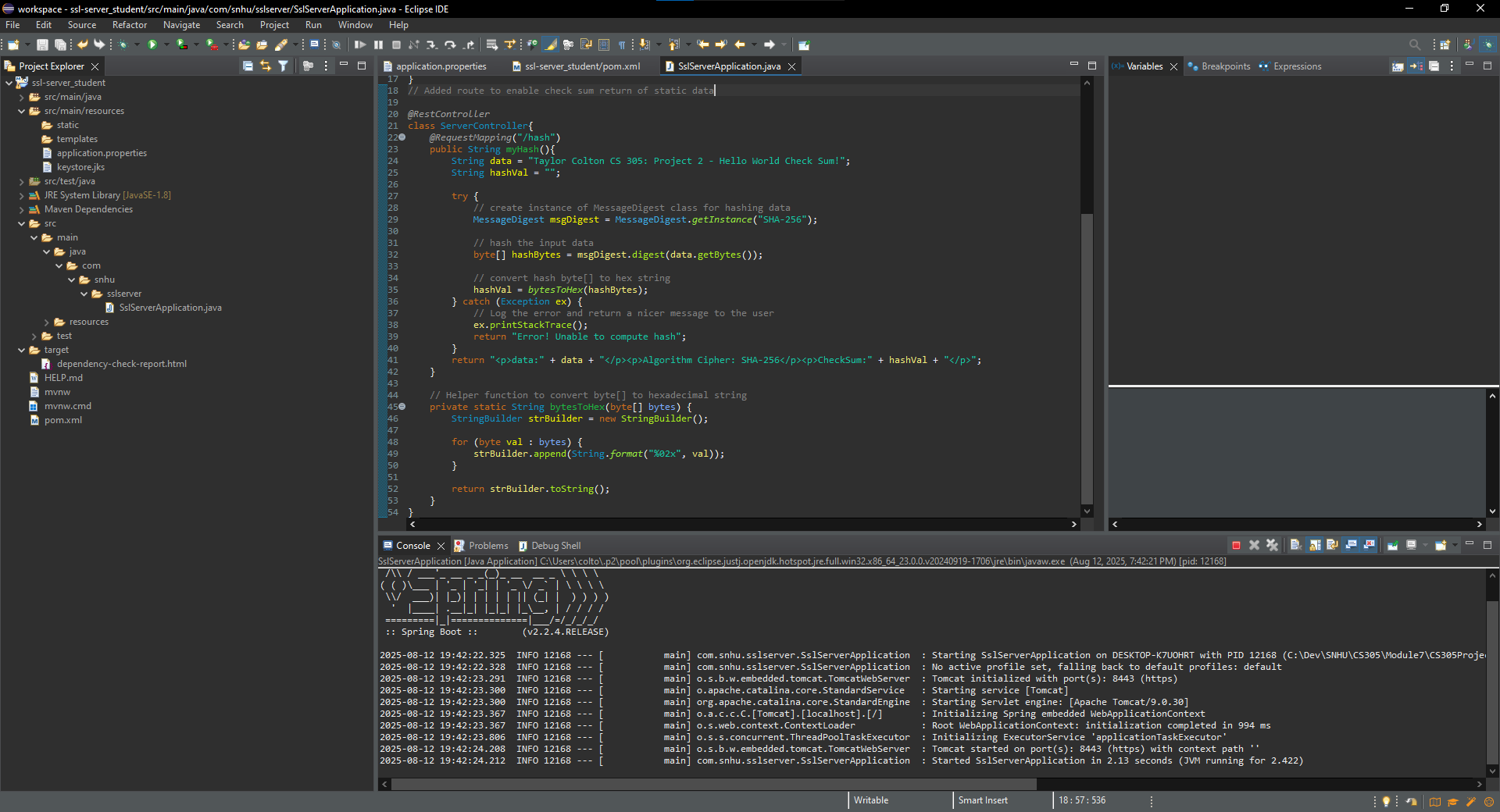


## Secondary Testing





## Functional Testing



## Summary

In order to automate vulnerability assessments and ensure transport security, I refactored the service. To ensure that all traffic to the /hash endpoint is secure while in transit, I first enabled HTTPS on port 8443 and pointed Spring Boot to a keystore (for more information, check the SSL properties for port, keystore location, type, alias, and password). In order to conduct security testing as part of the lifecycle, I also included the OWASP Dependency-Check plugin into the Maven build. Additionally, I supplied a suppression file that filters out obvious false-positive CPE mappings while maintaining the visibility of valid findings. By integrating automated dependency scanning, integrity verification (the SHA-256 checksum endpoint), and encrypted transport into a single repeatable build, these modifications jointly satisfy the project's security testing requirements.

In accordance with the vulnerability-assessment process, I: used automated scanning to find security problems; examined the results and isolated false-positives with targeted suppressions; fixed the issue by implementing HTTPS/TLS with a managed keystore and planned library updates; and confirmed by repeating the scan and testing the application over HTTPS on port 8443. Transport encryption (HTTPS/TLS), integrity checks (SHA-256), keystore-based secret handling, and continuous dependency management via the Maven plugin comprise this simple approach. By using a defense-in-depth strategy, the refactored code is guaranteed to adhere to security testing standards and be maintainable as the codebase changes.

## Industry Standard Best Practices

By automating dependency housekeeping and ensuring encrypted transport, I implemented industry-standard best practices, which I then confirmed in both the code and the build. In particular, I set up the SSL properties (port, keystore path/type, alias, and password) to maintain clear reliable defaults and enabled HTTPS with a maintained keystore in order to protect all traffic to the service, including the /hash route, in transit. In order to continuously surface known CVEs, I incorporated OWASP Dependency-Check into the Maven lifecycle. I then combined it with a suppression file that has a tight scope, filtering out only obvious false-positive CPE mappings, retaining actual vulnerabilities for maintenance. By guaranteeing confidentiality, integrity, and preventing vulnerable libraries from entering builds, these practices preserve the application's current security. They also improve the company's overall health by lowering the risk of breaches and unscheduled downtime, enhancing auditability and compliance posture, and establishing a developer-friendly, repeatable process for patching and verifying issues prior to release.