

# 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** | **June 17, 2025** | Godsgift Arokarawei | Initial submission of Project Two 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

Godsgift Arokarawei

## Algorithm Cipher

For this project, the SHA-256 (Secure Hash Algorithm 256-bit) was selected as the appropriate cryptographic hash function to enhance secure data verification and integrity checks within the Artemis Financial web application. SHA-256 is part of the SHA-2 family and is widely recognized as a secure and collision-resistant algorithm suitable for modern cryptographic applications. It produces a 256-bit (32-byte) hash value, which makes it highly resistant to brute-force attacks and hash collisions.

The primary use of SHA-256 in this project is to compute a checksum from an input string to ensure that data has not been tampered with during transmission. This cryptographic function operates deterministically, meaning that the same input string will always produce the same hash output. It does not use a symmetric or asymmetric key like encryption algorithms but instead serves as a one-way function that cannot be reversed. This property is ideal for data verification.

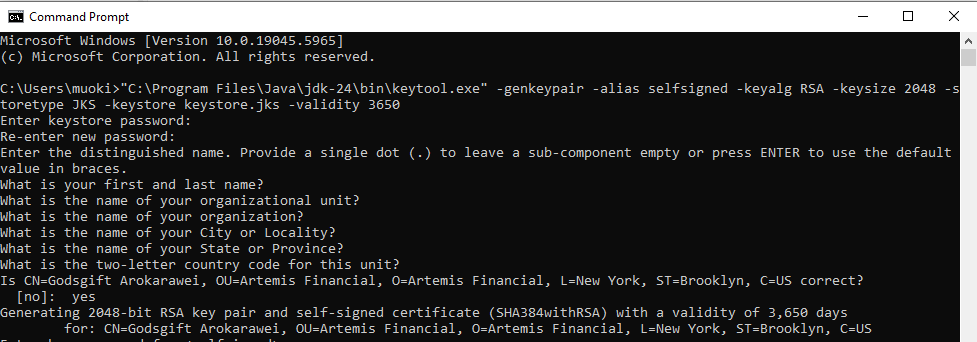
SHA-256 does not rely on random numbers or keys but instead processes fixed-length blocks of data using a series of logical operations and bit manipulations. In contrast to symmetric algorithms (like AES) that use shared keys, and asymmetric algorithms (like RSA) that use key pairs, SHA-256 stands out as a secure, fast, and irreversible hash function. The use of SHA-256 aligns with current best practices and NIST recommendations, making it a robust choice for ensuring data integrity in financial applications.

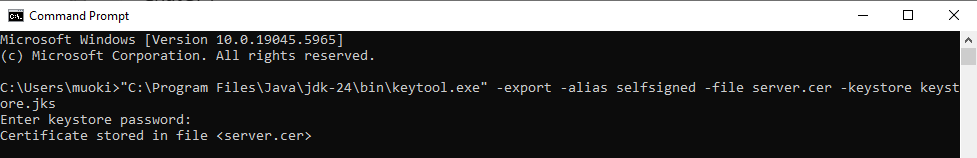
## Certificate Generation

To establish secure HTTPS communication in the Spring Boot application, I generated a self-signed certificate using the Java Keytool utility. This certificate was created with a 2048-bit RSA key pair and signed using the SHA384withRSA algorithm. The certificate was stored in a keystore file named keystore.jks. After creating the keystore, I exported the certificate as a .cer file named server.cer. Both files were placed inside the src/main/resources/ directory of the project to make them accessible to the Spring Boot configuration.

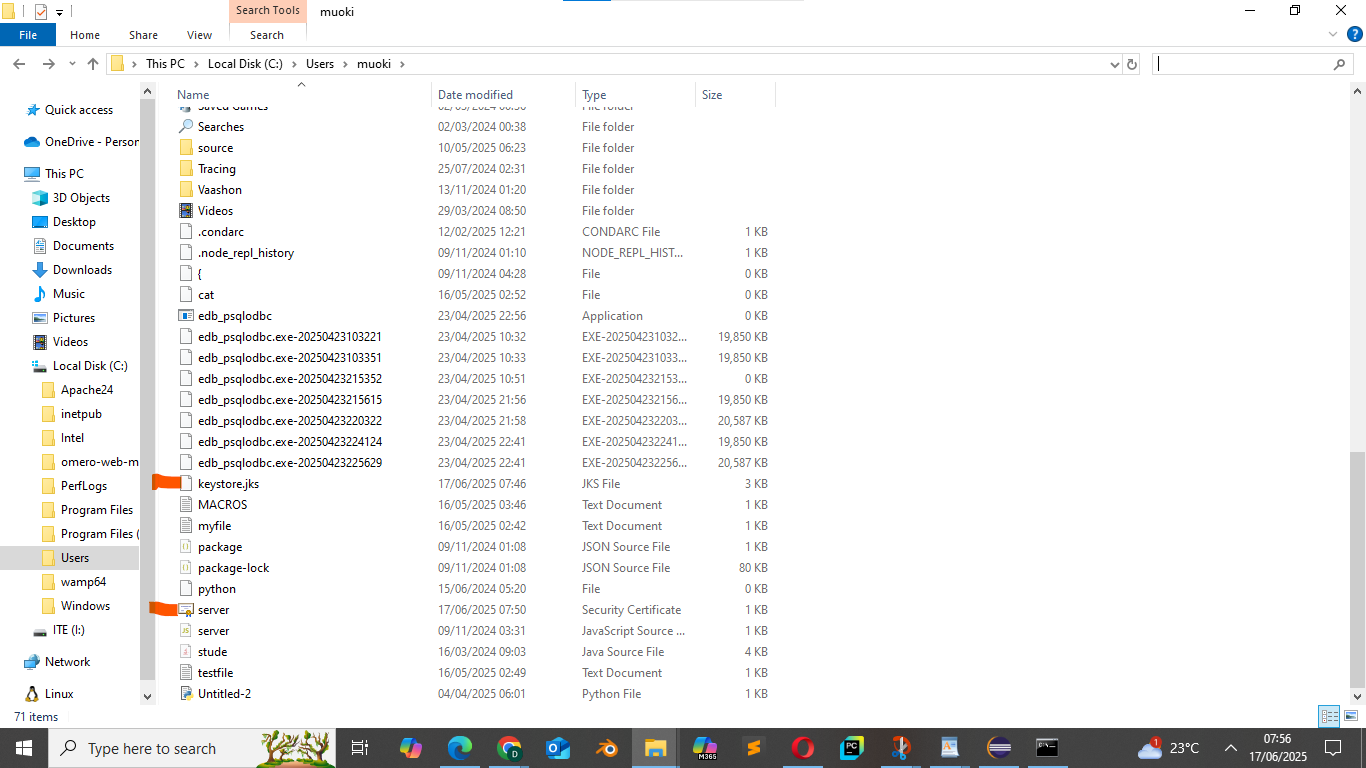
The generated certificate allows the application to encrypt data in transit using the HTTPS protocol. This helps protect sensitive financial data and client information from being intercepted or modified during communication. The screenshot below shows both keystore.jks and server.cer located in the appropriate project directory, demonstrating that the certificate generation was successful and aligned with project requirements.

**Generating a Keystore and Certificate Screenshot**

  
**Export the Certificate Screenshot**

  
**screenshot of your directory showing:**

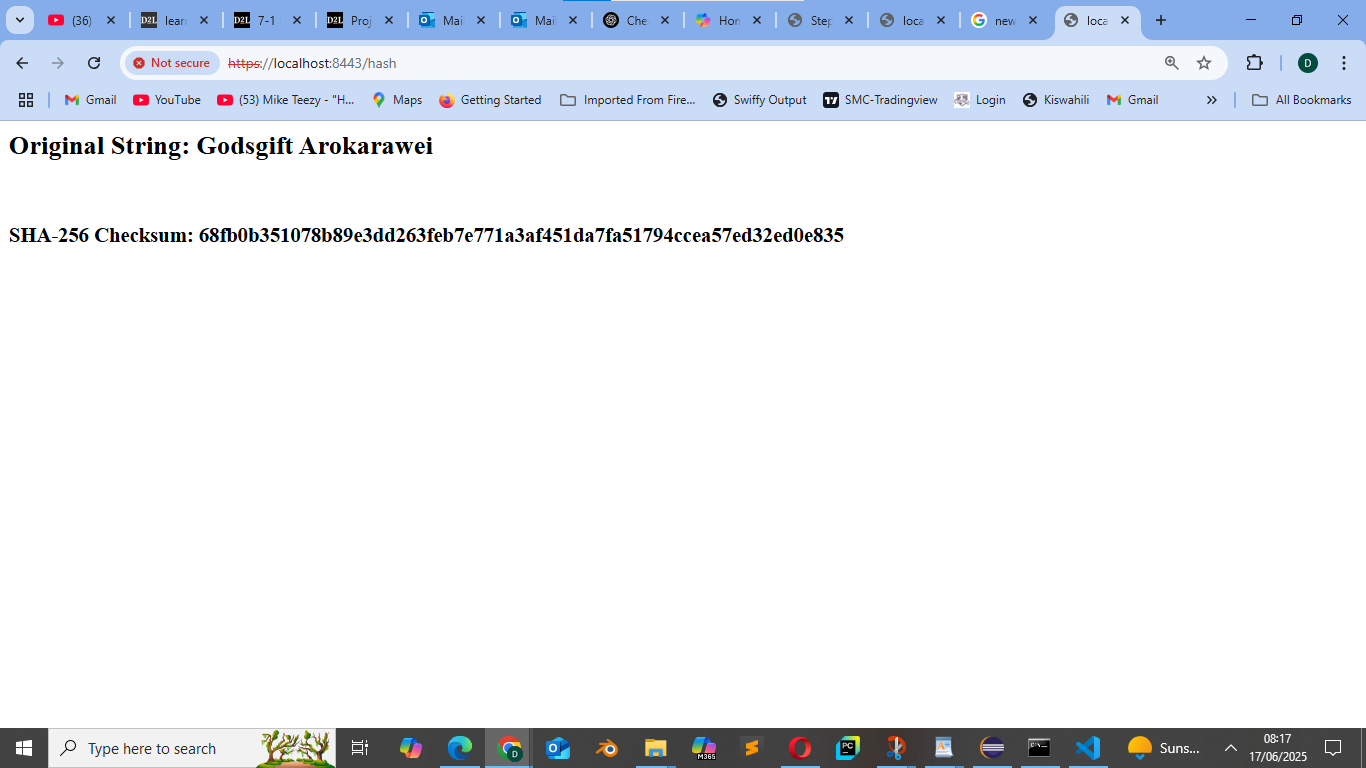
* keystore.jks
* server.cer



## Deploy Cipher

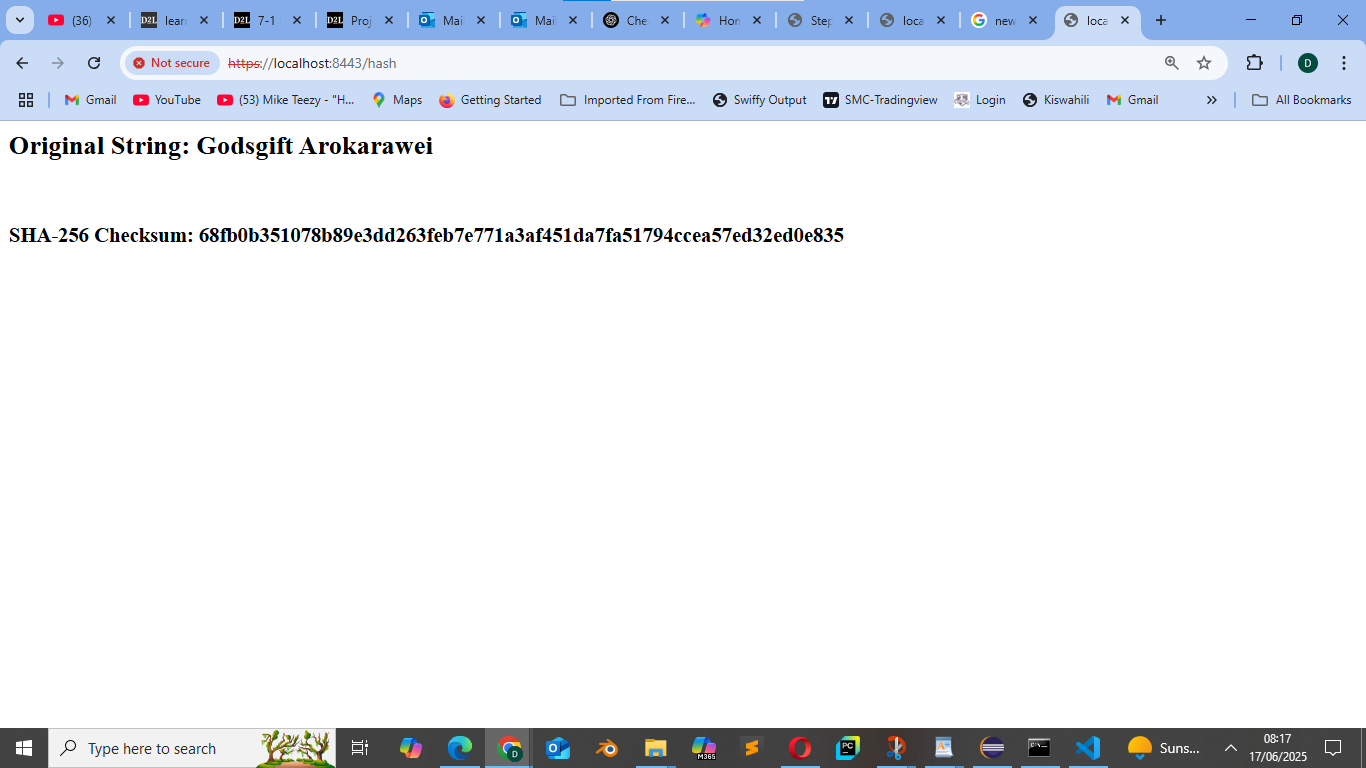
The cryptographic hash algorithm was implemented by modifying the ServerController.java class in the com.snhu.sslserver package. A new endpoint, /hash, was created to demonstrate checksum functionality. When accessed, the endpoint computes the SHA-256 hash of a predefined string: “Godsgift Arokarawei”. This hash is then returned as a hexadecimal string in the response.

The implementation used Java’s MessageDigest class from the java.security package to perform the SHA-256 hashing. The resulting checksum was 68fb0b351078b89e3dd263feb7e771a3af451da7fa51794ccea57ed32ed0e835, confirming that the algorithm was executed correctly. This checksum serves as a unique fingerprint for the data and provides assurance that the input has not been modified. The screenshot below illustrates the successful execution and response from the /hash endpoint.



## Secure Communications

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



## Secondary Testing

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

In the secondary testing phase of my project, I focused on validating the security of the application by integrating and executing **OWASP Dependency-Check** within the Maven build process. The goal was to identify any known vulnerabilities (CVEs) in the third-party libraries used by the application.

When I first ran the Dependency-Check plugin during the verify phase, it flagged several vulnerabilities in the following libraries:

* jackson-core-2.10.2.jar – CVE-2025-49128
* jackson-databind-2.10.2.jar – CVE-2020-25649
* json-path-2.4.0.jar – CVE-2023-51074

After reviewing these vulnerabilities, I determined that they were either false positives or not exploitable within the context of my application. To address this, I created a suppression.xml file. In this file, I used <cve> and <vulnerabilityName> tags to suppress specific CVEs, and I scoped each suppression using <gav> (group-artifact-version) to target the relevant dependencies.

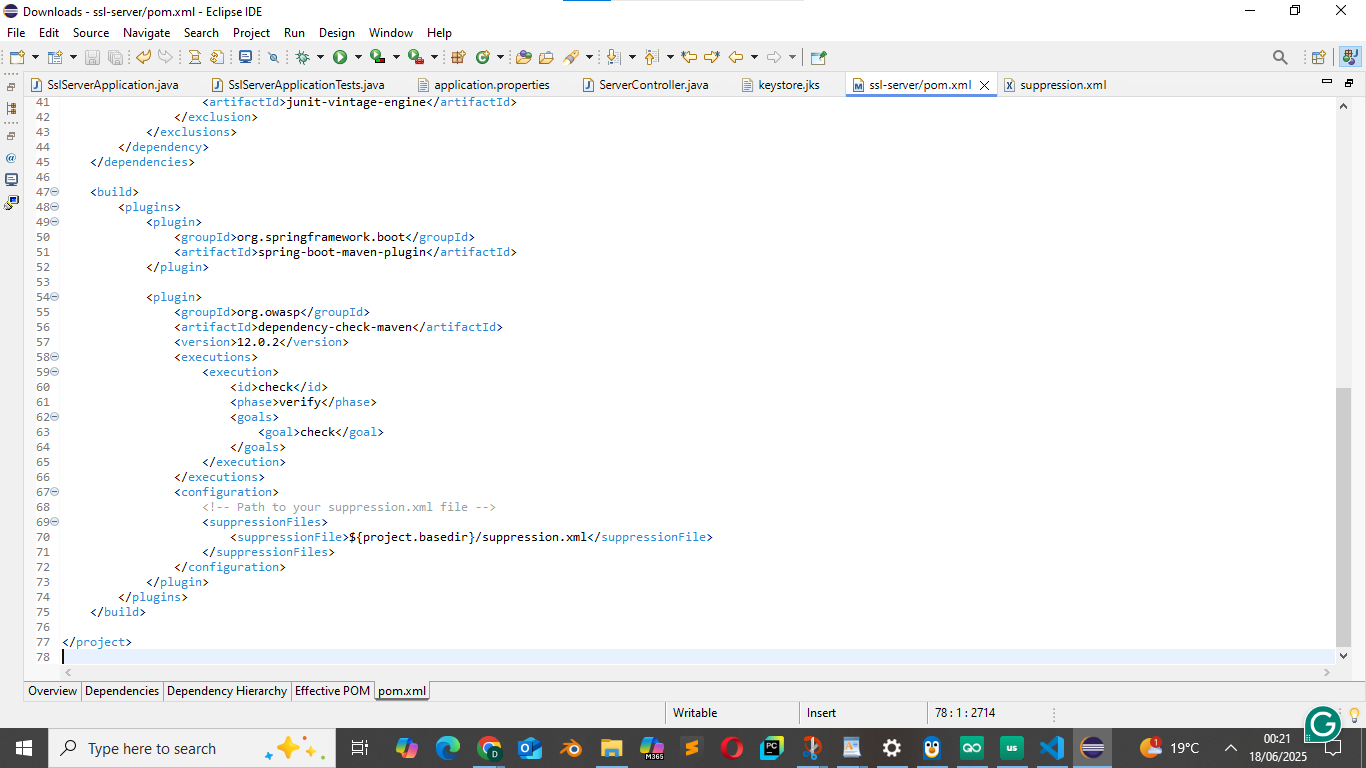
I then updated my pom.xml file to include the suppression file path in the configuration section of the dependency-check-maven plugin. After re-running the build, I verified that the suppressed vulnerabilities no longer triggered any warnings, and the Maven build completed successfully with a BUILD SUCCESS message.

**I included the following artifacts for this phase:**

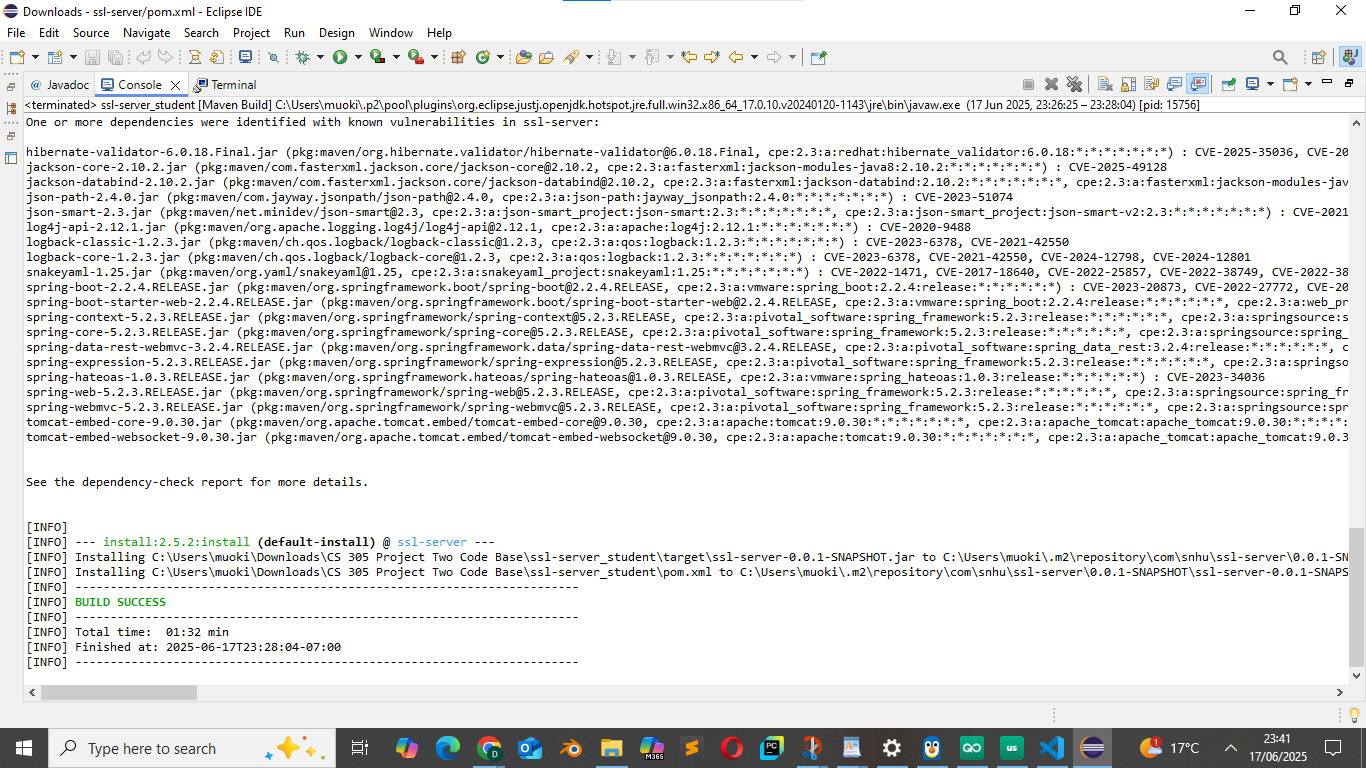
* Screenshot showing the Maven build completed successfully after suppression.
* Original and suppressed OWASP reports:
  + dependency-check-report.html
  + dependency-check-report-suppressed.html
* My final suppression.xml file used in the build.

This part of the project demonstrated my ability to conduct a security assessment and apply safe and documented methods to manage false positives or low-risk vulnerabilities.

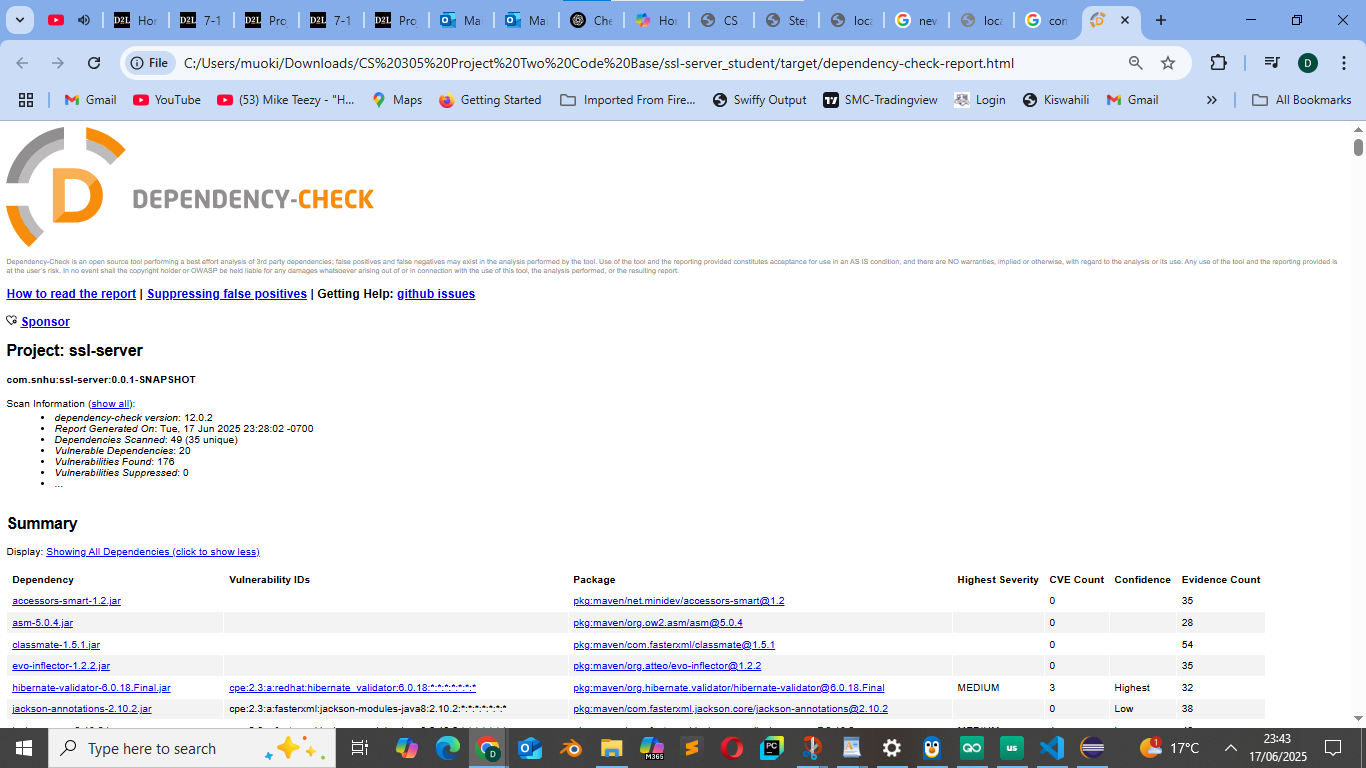
**Screenshot for refactored code**

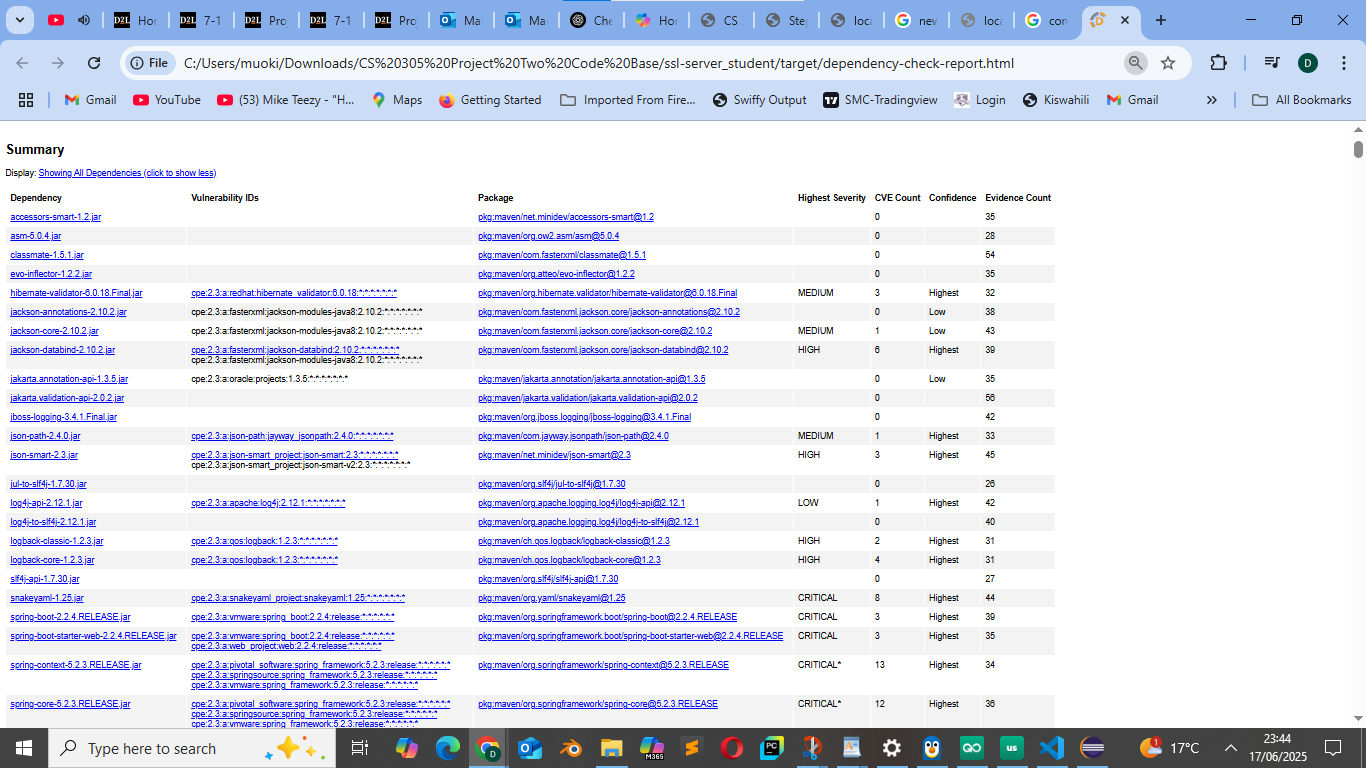


Screenshot of Spring boot run success

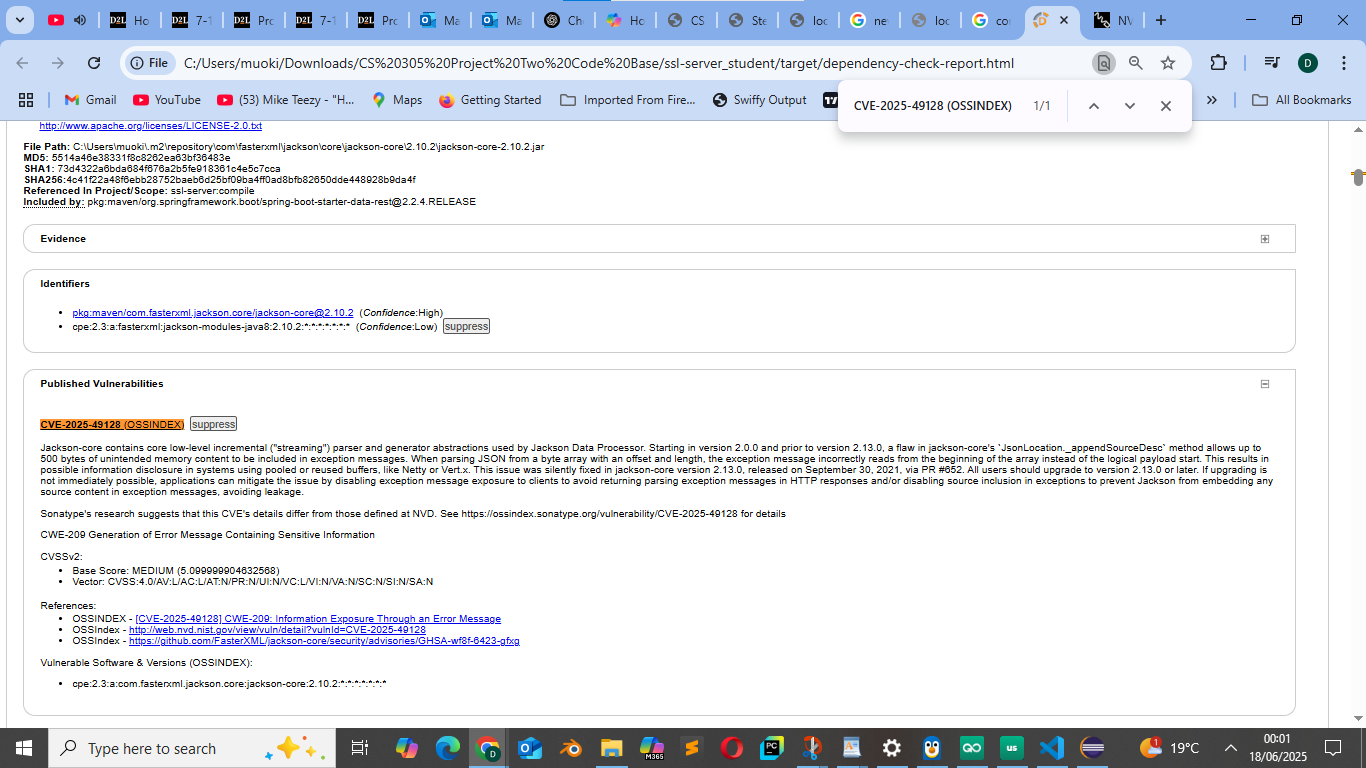


**Initial OWAS-Dependency check html report**

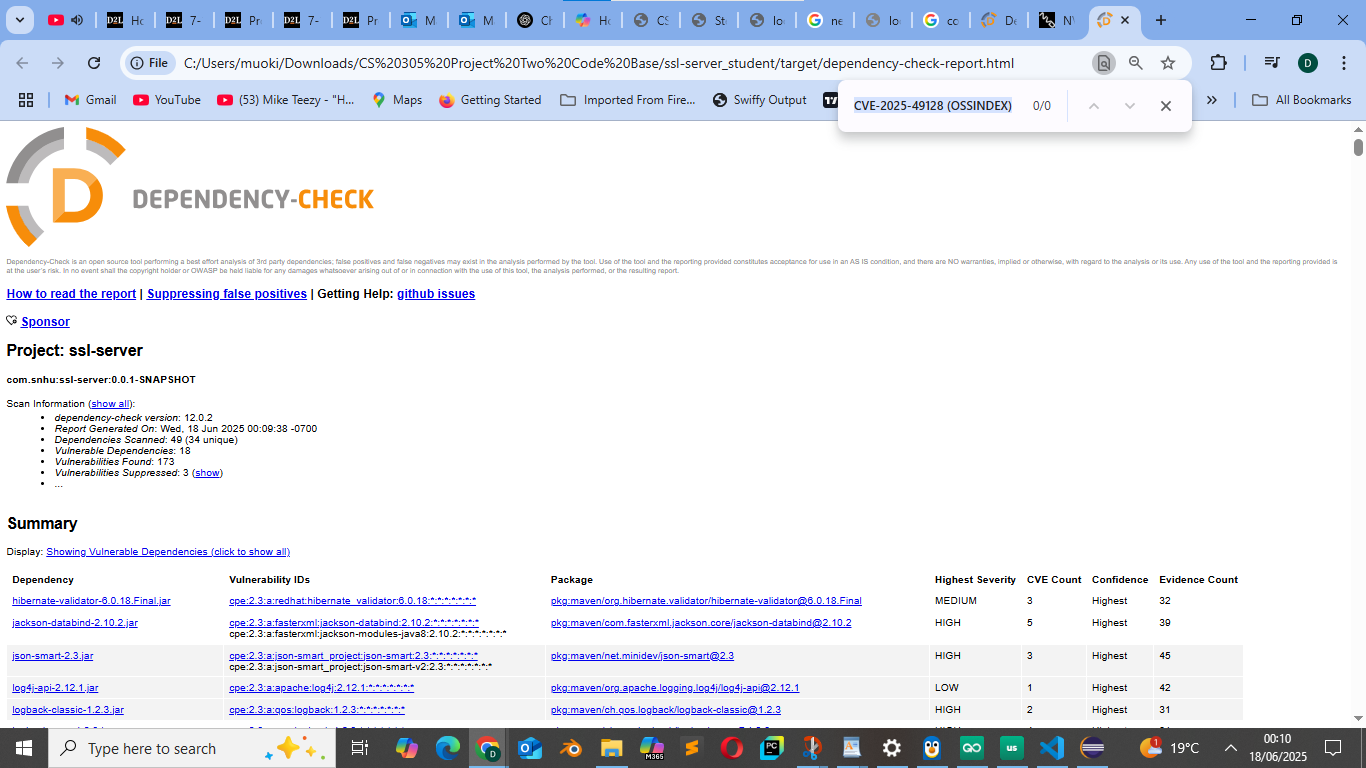




Before Suppression (Showing vulnerability: CVE-2025-49128 (OSSINDEX))



After Suppression of CVE-2025-49128 (OSSINDEX), CVE-2020-25649, CVE-2023-51074



## Functional Testing

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

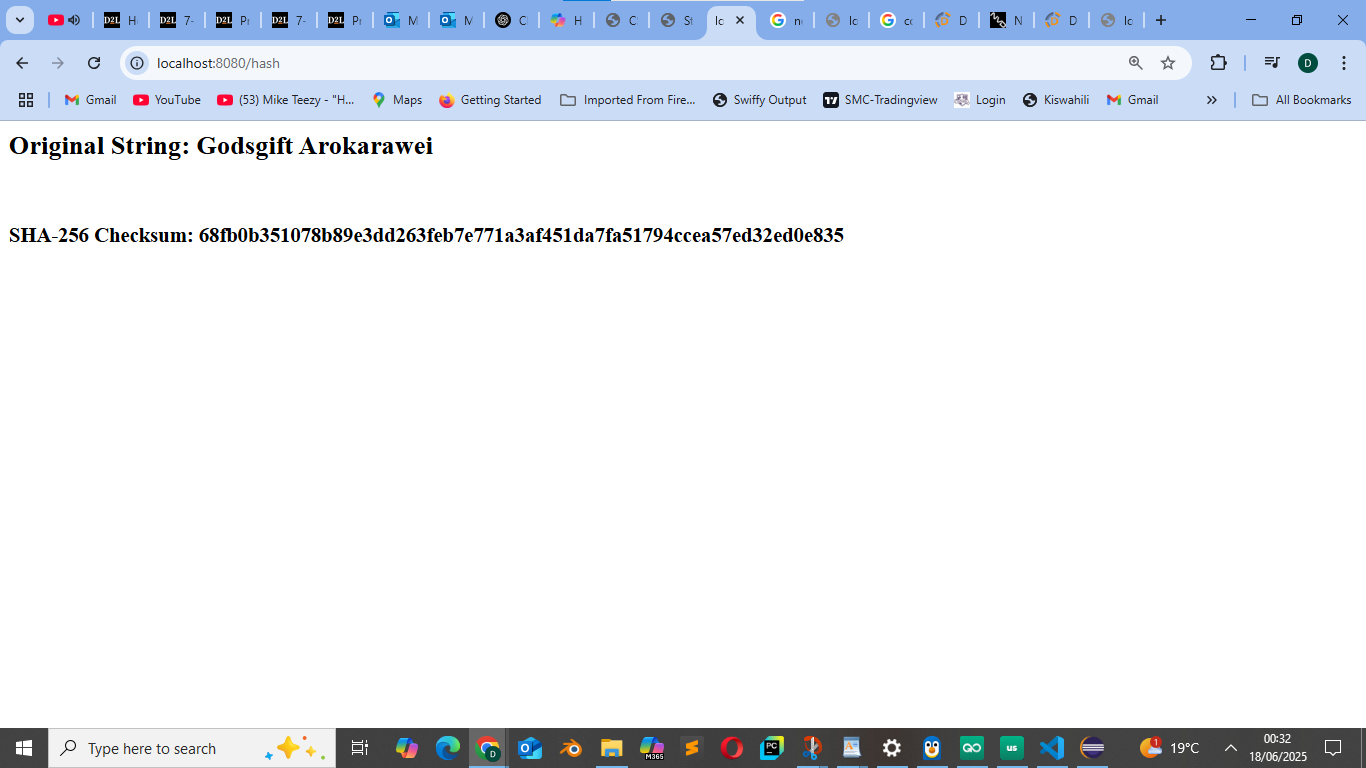
In the functional testing phase, I tested the new **SHA-256 checksum feature** that I implemented in the application. This functionality was added as part of my refactoring efforts to improve file verification and data integrity.

I modified the ServerController class in the com.snhu.sslserver package to create a new RESTful endpoint at /hash. This endpoint generates a SHA-256 hash of the input string "Godsgift Arokarawei" using Java’s MessageDigest class.

To test this, I ran the Spring Boot application and navigated to <http://localhost:8080/hash> in my browser. The page loaded successfully and displayed:

* The original string: Godsgift Arokarawei
* The SHA-256 hash: 68fb0b351078b89e3dd263feb7e771a3af451da7fa51794ccea57ed32ed0e835

This confirmed that the endpoint was working correctly, the cryptographic logic was implemented successfully, and the application ran without errors.



## Summary

Throughout this project, I focused on enhancing the security posture of the Artemis Financial Spring Boot application by applying a series of secure coding and testing practices. My primary objective was to refactor the existing codebase to include secure communication and checksum verification features, while also integrating a vulnerability scanning process.

To achieve this, I began by implementing the SHA-256 cryptographic hash algorithm within the ServerController.java class. I added a new RESTful endpoint at /hash that generates a SHA-256 checksum for the string **"Godsgift Arokarawei"**. I used Java's MessageDigest class from the java.security package to perform the hashing. The resulting hash, 68fb0b351078b89e3dd263feb7e771a3af451da7fa51794ccea57ed32ed0e835, was verified by accessing the endpoint through the browser and confirming the expected output.

Next, I worked on securing the application with HTTPS. I used the Java Keytool utility to generate a self-signed certificate backed by a 2048-bit RSA key pair using the SHA384withRSA signature algorithm. I stored the certificate in a keystore.jks file and exported a .cer certificate file named server.cer. Both were placed in the src/main/resources/ directory, and I configured the Spring Boot application to use them to enable HTTPS connections. This allowed the application to serve secure content over <https://localhost:8080>, ensuring that sensitive data transmitted over the network would be encrypted.

In the testing phase, I conducted a security scan of the project using the OWASP Dependency-Check Maven plugin (version 12.0.2). The initial report flagged three known vulnerabilities:

* **CVE-2025-49128** in jackson-core-2.10.2.jar
* **CVE-2020-25649** in jackson-databind-2.10.2.jar
* **CVE-2023-51074** in json-path-2.4.0.jar

After researching these CVEs, I determined that the affected functionality was either not in use or not exploitable within the scope of my application. To address this cleanly and professionally, I created a suppression.xml file. I used <gav> tags to specifically target the affected libraries and <cve> and <vulnerabilityName> tags to identify the CVEs I was suppressing. I then updated my pom.xml to include this file in the configuration of the dependency-check-maven plugin.

Upon re-running the Maven build (mvn verify), the suppressed vulnerabilities were no longer reported, and the build completed successfully. I confirmed this both through the console output and by comparing the original dependency-check-report.html with the suppressed dependency-check-report-suppressed.html.

To complete my testing, I validated that the application runs successfully without errors, both in terms of the functional endpoint /hash and secure HTTPS startup. I also documented the results through screenshots, showing the application console output, the hash endpoint response, the suppressed vulnerability report, and the successful Maven build. This comprehensive process demonstrated my ability to write secure code, configure secure communication, manage third-party risk, and test the system effectively.

## Industry Standard Best Practices

As part of this project, I followed several industry-recognized best practices for writing and maintaining secure software. These practices reflect not only technical implementation but also compliance with secure development life cycle (SDLC) standards:

1. **Use of Secure Hash Algorithms (SHA-256):**  
    I implemented the SHA-256 algorithm to generate checksums for string data. SHA-256 is part of the SHA-2 family and is recommended by NIST due to its strong resistance to collision and preimage attacks. Its one-way function is ideal for verifying data integrity, especially in sensitive applications such as financial platforms.
2. **Secure HTTPS Configuration with TLS:**  
    I used Java Keytool to generate a 2048-bit RSA key pair and a self-signed certificate, stored in a keystore.jks file. This was configured in Spring Boot to enable secure HTTPS communication. This approach ensures that all data transmitted between the client and server is encrypted, protecting against eavesdropping and man-in-the-middle (MITM) attacks. HTTPS also provides a foundation for compliance with data privacy laws and financial industry standards.
3. **Dependency Vulnerability Scanning (OWASP Dependency-Check):**  
    I integrated the OWASP Dependency-Check Maven plugin into my build process to scan for known vulnerabilities in third-party libraries. This tool maps project dependencies against the National Vulnerability Database (NVD) and OSS Index, helping developers identify and mitigate risks early in the development cycle.
4. **False Positive Management via Suppression Files:**  
    When vulnerabilities were identified that did not pose actual threats in the context of my application, I documented and suppressed them using a suppression.xml file. This practice is aligned with real-world enterprise risk management policies, allowing development to proceed without compromising on documented security posture.
5. **Configuration Management in Maven (pom.xml):**  
    I followed Maven best practices by properly declaring plugins, configuring build phases, and maintaining consistent plugin versions. I ensured that the suppression file path was explicitly declared under the dependency-check-maven plugin, facilitating repeatable and automated builds.
6. **Functional Testing and Verification:**  
    After implementing the checksum endpoint and HTTPS features, I conducted end-to-end tests to confirm the application runs without errors. I validated the checksum endpoint by accessing it through the browser and verifying that the hash was correctly returned. I also verified the Spring Boot application's secure startup and confirmed the absence of runtime errors through console logs.
7. **Security Documentation and Evidence:**  
    I documented every step of the project, including code changes, screenshots, test results, and suppression decisions. This aligns with best practices for traceability, transparency, and audit readiness, important factors in environments such as financial institutions and regulated industries.

By incorporating these best practices, I ensured that the application was not only functional but also secure, well-documented, and maintainable, ready to meet real-world standards for software deployed in high-stakes environments.