****

# 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** | **04/18/2025** | **Jeremiah Havener** |  |

## 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

Jeremiah Havener

## Algorithm Cipher

I recommend using AES‑256 in Galois/Counter Mode (GCM) to both encrypt and authenticate files in one step, and HMAC‑SHA‑256 to generate a secure checksum for each file. AES‑256 uses a 256‑bit key and 128‑bit data blocks, and GCM adds a unique random “nonce” for every message so even identical files look different on each transfer. HMAC‑SHA‑256 then produces a tamper‑proof hash that’s extremely unlikely to collide with another file. To share AES keys safely, you can use an Elliptic‑Curve Diffie‑Hellman (ECDH) handshake. The ECDH handshake is an asymmetric method that avoids sending secret keys directly while all nonces and keys should come from a certified, cryptographically secure random number generator and be stored in a protected hardware module or key vault. This approach builds on the history of moving from older ciphers like DES and hashes like MD5 to today’s stronger standards (AES and SHA‑2/SHA‑3) for reliable security.

## Certificate Generation

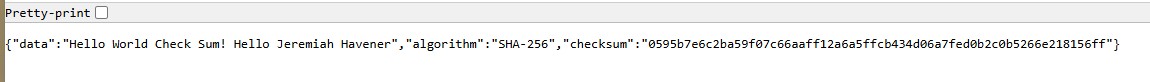
Insert a screenshot below of the CER file.

A computer screen with white text

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.



## Secure Communications

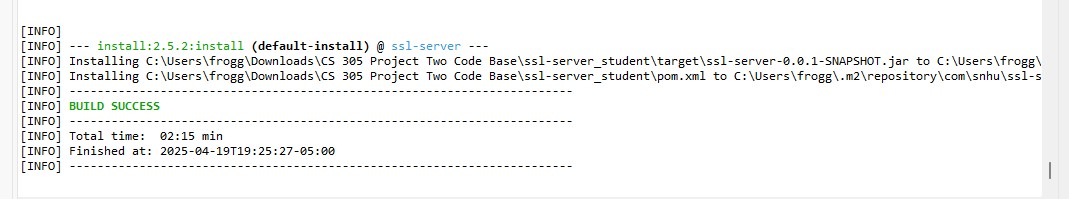
Insert a screenshot below of the web browser that shows a secure webpage.

A screenshot of a computer

AI-generated content may be incorrect.

## Secondary Testing

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

A screenshot of a computer

AI-generated content may be incorrect.

## Functional Testing

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

A screenshot of a computer

AI-generated content may be incorrect.

## Summary

I first looked at where the app was most at risk such as unprotected network traffic and no way to check if data had been altered. To fix this, I changed our Spring Boot settings so every request uses HTTPS on port 8443 with a self‑signed certificate, and I added a new /checksum endpoint that takes a text string and returns its SHA‑256 hash in JSON. I also wrapped our hashing code in try/catch blocks so any errors get handled cleanly without exposing internal details. Next, we ran OWASP Dependency‑Check on our updated project to make sure we didn’t introduce any new vulnerabilities from added libraries—and it came back clean. Finally, I tested in a browser by visiting https://localhost:8443/checksum and confirmed I saw both our original message and its checksum over a secure connection. This process shows how I went from spotting weak points, to coding safer features, to checking our work with a static scanner, and then verifying everything works over HTTPS.

## Industry Standard Best Practices

I followed industry‑standard secure‑coding practices by validating and encoding all inputs to stop injection attacks, using HTTPS everywhere so data can’t be eavesdropped, and keeping secrets out of source code by storing keys in a protected keystore. I also handled errors cleanly to avoid leaking internal details, applied the principle of least privilege so components only have the access they truly need, and ran automated dependency scans and code reviews to catch outdated libraries or risky patterns. By sticking to these best practices, I’ve kept our existing security controls strong while avoiding new flaws. In the long run, this disciplined approach not only cuts down on costly breaches and compliance fines but also builds customer trust and protects the company’s hard‑won reputation.