****

# 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** | **17 June 2026** | **Albert Waterman** |  |

## 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

Albert Waterman

## Algorithm Cipher

I used the SHA-256 algorithm to generate a secure checksum. SHA-256 stands for "Secure Hash Algorithm 256-bit" and is part of the SHA-2 family created by the NSA. It takes any input, like a name or string, and turns it into a 64-character hash that’s unique to that input. It’s a one-way function, so once it’s hashed, there’s no way to reverse it back. I used Java’s built-in MessageDigest class import to apply SHA-256. SHA-256 isn’t like typical encryption where you use symmetric or asymmetric keys. This is a hash function, so there’s no key pair or random number involved. It just gives you a consistent fingerprint of whatever data you give it. I picked it because it’s collision-resistant, meaning two different inputs won’t end up with the same hash. That’s important for making sure the data stays unique and tamper-proof. Hashing and encryption has come a long way. Earlier versions like SHA-1 and DES got replaced once they were found to be weak. It is still considered safe and is used for things like digital signatures and file integrity. It’s built into Java making it very easy to implement.

## Certificate Generation

Insert a screenshot below of the CER file.

A computer screen with white text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screenshot of a computer

AI-generated content may be incorrect.

## 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 program

AI-generated content may be incorrect.

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 error

AI-generated content may be incorrect.

## Summary

After refactoring the code, I added a SHA-256 hash function to protect the data and help secure the API. Since this is a RESTful application, it was important to make sure that any data being exchanged was protected. I also configured the application to use HTTPS by setting up a certificate, which helped encrypt the connection between the client and server. That way, sensitive info isn’t sent in plain text. I wrapped the hashing logic in a try-catch block to make sure any errors could be handled without crashing the app. I also made sure the Spring Boot and Tomcat versions were updated and compatible to avoid any issues tied to older versions. All of this helps make the application more secure and protects users from known threats.

## Industry Standard Best Practices

In refactoring the application, I followed industry best practices for secure coding to ensure the system remained stable and protected against known threats. I used trusted, built-in Java libraries for cryptographic functions instead of relying on third-party tools, which helps reduce the attack surface. Sensitive operations, like hashing, were handled securely with proper exception handling in place to catch unexpected issues without exposing the system. I also conducted a static analysis using OWASP Dependency-Check to verify that no vulnerabilities were introduced during the update. This tool compared the application’s dependencies against known CVEs, giving me confidence that the refactored code met security standards.