****

# 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** | **4/17/25** | **Dustin Davis** |  |

## 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

Dustin Davis

## Algorithm Cipher

One of the best choices for secure file verification is SHA-256, which is part of the SHA-2 family of cryptographic hash functions. It’s not technically an "encryption" algorithm, but it works great for verifying that data hasn’t been changed during transfer. Basically, it takes a chunk of input (like a string or file) and turns it into a long, fixed-size string of letters and numbers called a hash. Even the smallest change to the input makes a totally different hash, which is why it’s perfect for things like checksums or digital signatures. It’s fast, reliable, and has been tested and trusted for years.

SHA-256 is a hash function that creates a 256-bit output, which is why the “256” is in the name. That means the final hash has 256 binary digits, or 64 characters when shown in hexadecimal. It’s designed so that the same input always gives the same output, but it’s nearly impossible to reverse or guess the original input just from the hash. That makes it great for security. SHA-256 is strong enough that no one has been able to break it, and it avoids weaknesses that older versions like SHA-1 had. It’s also one of the most commonly used hash functions in security systems today, so it’s a solid, reliable choice.

Hash functions like SHA-256 don’t actually use keys the way encryption algorithms do, but when it comes to encryption in general, there are two main types: symmetric and asymmetric. Symmetric encryption uses the same key to lock and unlock the data, kind of like using the same password to open a locked file. Asymmetric encryption is different, it uses a key pair, one public and one private. You can think of it like sending someone a locked box where only they have the key. Random numbers are also a big part of cryptography, especially when generating keys or digital signatures. If those numbers aren’t random enough, the system can be predictable, which is not good for security. That’s why strong random number generation is so important when building secure systems.

Encryption has been around for a long time, even before computers existed. People used to rely on basic techniques like shifting letters in the alphabet, such as the Caesar cipher. As technology advanced, encryption methods became more complex. In the 1970s, the DES (Data Encryption Standard) was widely used, but eventually it became outdated and not secure enough. That led to the creation of newer algorithms like AES (Advanced Encryption Standard) and SHA (Secure Hash Algorithm). Today, encryption is built into almost everything online, including websites, banking systems, and messaging apps. As computers get faster and threats become more advanced, encryption continues to evolve. Right now, SHA-256 and AES are two of the most trusted and widely used tools for keeping data secure.

## Certificate Generation

Insert a screenshot below of the CER file.

I generated a self-signed certificate using Java Keytool and named it selfsigned. The certificate was created with a 2048-bit RSA key and set to be valid for 10 years. I then exported the certificate as a .cer file and stored both the .cer and the keystore.jks files in the project’s resources folder. The screenshots below show the certificate file in the project and the certificate details, including the SHA-256 fingerprint and the fact that it was issued to and by me.

A screenshot of a computer

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

After generating the certificate, I configured the application to run over HTTPS using port 8443. This was done by updating the application.properties file to reference the keystore, password, and key alias. When launching the app, the browser connects securely to https://localhost:8443, though it shows a “Not secure” warning because the certificate is self-signed and not issued by a trusted authority. The screenshot below confirms that HTTPS is active and the app is communicating securely using the certificate I created.

A screenshot of a certificate

AI-generated content may be incorrect. A screenshot of a web page

AI-generated content may be incorrect.

## Secure Communications

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

The screenshot below shows that the application is running on HTTPS using a self-signed certificate. The browser recognizes the secure connection, even though it flags the certificate as untrusted because it was self-generated for local testing.

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.

The first screenshot below shows the application starting and running successfully without errors. The second screenshot is the OWASP Dependency-Check report, confirming that the code changes did not introduce any high-risk vulnerabilities.

A screen shot of a computer

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.

After starting the application, I navigated to the /hash endpoint using the secure URL https://localhost:8443/hash. The application returned the expected output: a static string that includes my name, along with a SHA-256 checksum. This confirms that the refactored code works correctly and that the checksum logic performs as intended. The screenshot below shows the browser displaying the correct response, proving that the application is running without any functional errors.

A screenshot of a computer

AI-generated content may be incorrect.

## Summary

For this project, I refactored the original code by adding a new REST endpoint (/hash) that performs checksum verification using the SHA-256 hashing algorithm. This feature allows the application to verify the integrity of static data, which is important for secure data transfer. I made sure the data string included my name to meet the project’s requirements.

To support secure communications, I generated a self-signed SSL certificate using Java Keytool and configured the application to run on HTTPS using port 8443. This ensures that data sent and received by the application is encrypted. The keystore file was securely placed in the project’s resources directory and referenced through the properties file using classpath.

I followed the software security testing protocols by running a dependency check using the OWASP Maven plugin. The report confirmed that no critical or high-level vulnerabilities were introduced by my changes. I also tested the application manually to verify that it runs without errors, and the checksum endpoint performs as expected.

Looking at the vulnerability assessment flow diagram, the steps I covered include identifying the need for secure data handling, selecting and applying cryptographic protections (SHA-256), implementing HTTPS for secure transport, and running static analysis to check for any known issues in the codebase.

## Industry Standard Best Practices

Throughout this project, I applied industry standard best practices to keep the code secure and reliable. For example, I used SHA-256 for hashing, which is a well-known and trusted algorithm recommended by security experts. I also made sure to configure HTTPS with a properly generated certificate to protect data in transit. Using a secure keystore and referencing it through the classpath helped keep sensitive file paths out of the code. On top of that, I ran a static dependency check using OWASP tools to make sure no known vulnerabilities were introduced. Following these steps helps maintain security, not just in this project, but in any real-world application where user data is involved. Secure coding isn't just about following rules, it's about making decisions that protect people and their information.