****

# Practices for Secure Software Report

Table of Contents

[Practices for Secure Software Report 1](#_Toc212318550)

[Document Revision History 3](#_Toc212318551)

[Client 3](#_Toc212318552)

[Developer 3](#_Toc212318553)

[1. Algorithm Cipher 3](#_Toc212318554)

[2. Certificate Generation 4](#_Toc212318555)

[3. Deploy Cipher 5](#_Toc212318556)

[4. Secure Communications 5](#_Toc212318557)

[5. Secondary Testing 6](#_Toc212318558)

[6. Functional Testing 13](#_Toc212318559)

[7. Summary 15](#_Toc212318560)

[8. Industry Standard Best Practices 16](#_Toc212318561)

[9. Citations 17](#_Toc212318562)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **10/25/2025** | **Alexander J. King** | **Initial release** |

## Client



## Developer

Alexander J. King

## Algorithm Cipher

For Artemis Financial’s web application, I have opted to use the **SHA-256** hash algorithm, mainly because it is widely used, highly secure, and an excellent cipher for their specific web application requirement: file verification through the use of checksum (White, M., 2025).

Like all hash functions, SHA-256 has the following attributes:

* **Fixed output.** The function takes input of an arbitrary length and always produces an output of a fixed size. More specifically, the SHA-256 will always produce a 256-bit hash value regardless of the original input size.
* **Deterministic results.** The same input will always produce the same output.

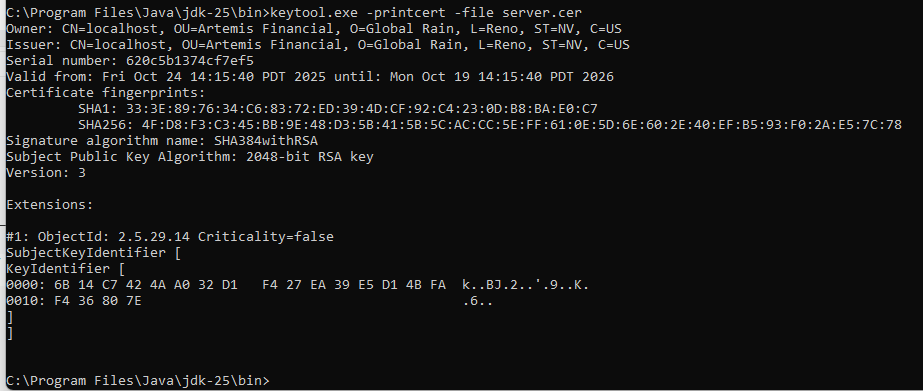
However, SHA-256 also has extra attributes that make it a *good* hash function:

* **One-way function.** The generated hash value cannot be reversed to reveal the original input.
* **High collision-resistance.** Collisions introduce vulnerabilities into systems because they allow hackers to break into protected areas of a website or application using a hash key that is the same as one used for authentication. Specifically for SHA-256, the total number of possible hash values is equal to 2256, which makes brute-forcing a collision nigh-impossible. Grant Sanderson, better known as 3Blue1Brown on YouTube, summarizes how difficult this would be as follows: “… even if you were to have your GPU-packed, kilo-Google-per-person, multiplanetary, giga-galactic computer guessing numbers for 37 times the age of the universe, it would still only have a 1-in-4 billion chance of finding the correct guess” (Sanderson, G., 2017).
* **Avalanche effect.** A small change to the input drastically changes the output, making it difficult to predict inputs based on small changes to the output.
* **Fast Performance.** Applications using SHA-256 do not sacrifice speed for security (White, M., 2025).

Generating a hash key through the use of checksums ensures any data stored or transmitted by Artemis Financial has not been corrupted, but it is just one piece of the security puzzle. We will also implement a digital certificate using asymmetric key cryptography to show users that this application is legitimate and to ensure connections are encrypted. I will explore these ideas further in my summary at the end of this document.

## Certificate Generation

Screenshots of the self-signed CER file below:



A screenshot of a computer screen

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

AI-generated content may be incorrect.

## Deploy Cipher

Screenshot of the checksum verification below:

A screenshot of a chat

AI-generated content may be incorrect.

## Secure Communications

Screenshot of a web browser that shows a secure webpage below:

Although my OS detects the certificate for the website is self-signed, this proof of concept shows the website will be secure through the use of HTTPS. All Artemis Financial needs is a third-party certificate authority (CA) to issue them a valid certificate for their web application, and we can refactor the code once more to use that certificate instead of the self-signed certificate.

A screenshot of a computer

AI-generated content may be incorrect.

## Secondary Testing

Screenshots of the refactored code executed without errors and the dependency-check report below:

**SslServerApplication.java Script**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**SslServerApplication.java Execution**

**A close up of text

AI-generated content may be incorrect.**

**ssl-server\_student/pom.xml Script**

A screenshot of a computer program

AI-generated content may be incorrect.

**ssl-server\_student/pom.xml Execution**

**A white background with red and blue lines

AI-generated content may be incorrect.**

**ssl-server\_student/pom.xml Execution** (continued)

**A white background with text

AI-generated content may be incorrect.**

**ssl-server\_student/pom.xml Execution** (continued)

**A close-up of a white background

AI-generated content may be incorrect.**

**Dependency Check Report**

A screenshot of a computer

AI-generated content may be incorrect.

## Functional Testing

Screenshot of the refactored code executed without errors below (now with private class methods):

**SslServerApplication.java Script**

A screenshot of a computer program

AI-generated content may be incorrect.

**SslServerApplication.java Execution**

A close up of text

AI-generated content may be incorrect.

## Summary

The areas of security we addressed through refactoring the application code were Input Validation, APIs, Cryptography, Client/Server, Code Error, Code Quality, and Encapsulation. The specific ways in which we addressed these areas are listed below:

* **Input Validation:** Checksums provide a mechanism to verify the integrity and authenticity of any data transmitted and stored by a web application. The SHA-256 hash algorithm allows Artemis Financial to apply a digital fingerprint to any data being transmitted to and from their application, including files containing sensitive financial information, passwords, etc. If a bad actor attempts to input something malicious, the checksum can flag that incoming data as malicious.
* **APIs:** Digital certificates and checksums work together to create secure API interactions. An API call starts with a TLS handshake in which the application server’s digital certificate is shown to the client to verify the application’s identity. If the certificate is legitimate, an encrypted channel is established to ensure any intercepted communications are not readable without the proper key. Once the channel is open, the checksum ensures any data passed within the API request is not corrupted or tampered with (even if the encrypted channel is somehow compromised).
* **Cryptography:** I touched on this in my explanation for APIs, but digital certificates are what make HTTPS possible, as SSL certificates allow data to be encrypted through the use of public and private keys. The certificate (normally issued by a third party certificate authority) is hosted on the application’s server and sent to any devices that visit the web application. Data sent by anyone accessing the website is encrypted using the public key included in the SSL certificate, which can only be decrypted using the private key. This helps ensure any data exchanges are protected from hackers attempting to intercept that data, as that data is essentially nonsense until it is decrypted.
* **Client/Server:** I also touched on this in my previous explanations, but secure distributed composing between the Client/Server is implemented for this application through the use of HTTPS protocols, which are enabled by the use of digital certificates and checksums. The client and server must verify each other’s identities through the use of digital certificates before establishing an encrypted channel, and data integrity between the client and server is maintained through the use of checksums.
* **Code Error:** Maven is a widely used build automation tool, and the Spring framework is a highly trusted Java platform. When used in tandem, they make managing large enterprise applications and their dependencies much easier. This application relies heavily on both, so the bulk of the code for the application is managed through them and their dependencies. The code we have added to the project does not introduce any new errors and should allow us to detect any file corruption without giving too much away.
* **Code Quality:** All new code introduced to the application follows secure coding practices and patterns by using tried-and-true algorithms and principles. SHA-256 is very widely used for checksums as well as password security (White, M., 2025), and the ServerController class we created to implement our cipher follows several OOP principles, including encapsulation and abstraction.
* **Encapsulation:** To be more specific, the ServerController class methods cannot be accessed outside of itself, leaving no entry points that a hacker could use to tamper with our cipher algorithm. Again, our refactored code does not introduce any new vulnerabilities into the application, but instead adds several layers of security.

To summarize, we added the SHA-256 hash algorithm to the application to create a checksum that can be used to validate the integrity of data stored and transmitted by the application. If a hash key on a file, password, or other piece of data does not match the expected output, the data can be flagged as malicious within the application. We also generated a digital certificate to enable HTTPS protocols on our website, which will ensure all lines of communication between the client and server for all users will be encrypted. Note, the self-signed certificate we used is for demonstration purposes only; Artemis Financial will need to contact a verified third-party certificate authority to generate a digital certificate for their application that will be trusted by all operating systems.

These layers of security working together should make it more difficult for hackers to upload malicious files and/or steal customer data.

## Industry Standard Best Practices

We covered how our refactored code adds layers of security, but our static testing *did* reveal some pre-existing vulnerabilities within the application. For context, the tool we used for static testing was the OWASP dependency check: an industry-standard static testing tool that is widely used for checking Java applications for known exploits. It compares the dependency files within the application with CPEs listed in the National Vulnerability Database and generates a report that includes several key pieces of information: CVE entries for each file, the severity of each vulnerability, how confident the checker is in its findings, etc. (Long, J., 2025). When we ran our static test using the OWASP dependency checker, it detected **15** vulnerable dependencies with **158** total vulnerabilities.

At first glance, that is quite an alarming number, but looking closely at the report reveals quite an easy fix for most of these vulnerabilities: update each dependency to their latest versions. The original application code is using very outdated versions of both Maven and the Spring Framework, both of which have been updated very recently to account for vulnerabilities. In addition, some of the reported vulnerabilities have lower than “CRITICAL” severities, as well as very few documented CVEs tied to them. These lower-rated vulnerabilities are also not marked as having known exploit histories, meaning that while the dependency has been documented as potentially vulnerable, no known attacks have occurred. The dependencies marked as critical vulnerabilities with known exploits are all tied to outdated versions of the Spring Framework and the Apache Tomcat Maven plugin, which can be updated to fix the problem.

This static testing, analysis of the application, and refactoring the code, are all part of industry standard best practices for secure coding. Sanitizing/validating inputs, creating secure API interactions, encrypting data, handling errors securely, and all of the other areas we addressed in our summary help ensure an application is designed with security in mind. When used in tandem with regular testing, review, and updates (especially for third-party libraries and dependencies), an application can launch secure and *remain* secure. By implementing our checksum, deploying a digital certificate through a verified certificate authority, regularly performing analysis and static testing, and making updates where necessary, Artemis Financial can give their customers peace of mind that their sensitive data won’t ever be compromised. This, in turn, will guarantee customer loyalty, as well as safeguard the company against possible legal action if there is a data breach as long as they remain compliant with industry standards for handling user’s financial data.

## Citations

GeeksForGeeks. (2022, April 29). *SHA-256 Hash in Java.* geeksforgeeks.org.

<https://www.geeksforgeeks.org/java/sha-256-hash-in-java/>

Long, Jeremy. (2025) *OWASP Dependency-Check.* jeremylong.github.io.

<https://jeremylong.github.io/DependencyCheck/general/thereport.html>

Sanderson, Grant [3Blue1Brown]. (2017, July 8). *How secure is 256 bit security?* [Video]. YouTube.

<https://youtu.be/S9JGmA5_unY?si=anDFvrPHzLfmIdKQ>

Tian, Beau & Piwowarek, Grzegorz. (2025, February 20). *SHA-256 and SHA3-256 Hashing in*

*Java.* Baeldung. <https://www.baeldung.com/sha-256-hashing-java>

White, Marcus. (2025, February 20). *[New research] How well does SHA256 protect against modern*

*password cracking.* SpecOpsSoft.com.

<https://specopssoft.com/blog/sha256-hashing-password-cracking/#:~:text=Digital%20Signatures:%20It%20is%20used,like%20salting%20and%20key%20stretching>.