****

# 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 5](#_Toc102040763)

[7. Summary 6](#_Toc102040764)

[8. Industry Standard Best Practices 6](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **8/24/2025** | **Matt Kostandin** |  |

## 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

Matt Kostandin

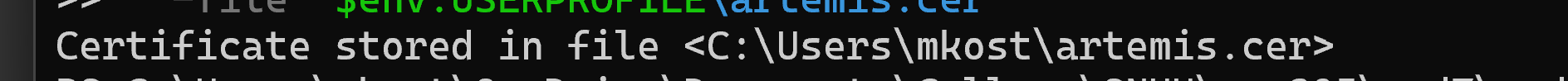
## Algorithm Cipher

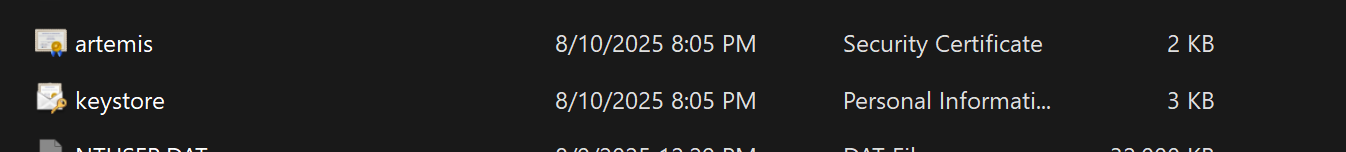
I recommend using SHA‑256 as Artemis Financial’s standard checksum for file and data verification, alongside HTTPS/TLS for secure transport. SHA‑256 is a one‑way cryptographic hash that turns any input into a short, fixed “fingerprint,” allowing us to quickly detect accidental corruption or deliberate tampering. It is a widely trusted, modern standard used across government and industry and is still considered strong today. This choice avoids deprecated options like MD5 and SHA‑1, keeps performance high, and aligns with common compliance expectations while fitting cleanly into our Spring Boot service and delivery pipeline.

SHA‑256 produces a 256‑bit 64‑character hex that is extremely hard to forge. For public, read‑only verification plain SHA‑256 is appropriate for example, users confirming a download. When we must prove authenticity as well as integrity like when preventing an attacker from swapping both a file and its checksum, we can use HMAC‑SHA‑2566 which adds a server‑held secret key. Hashing itself doesn’t require randomness, but our overall security continues to rely on strong randomness and modern TLS preferring TLS 1.3 for encryption in transit. Older algorithms and protocols like MD5, SHA‑1, SSLv3, and TLS less than 1.2 remain disabled to reflect current best practices.

## Certificate Generation

Insert a screenshot below of the CER file.





## Deploy Cipher

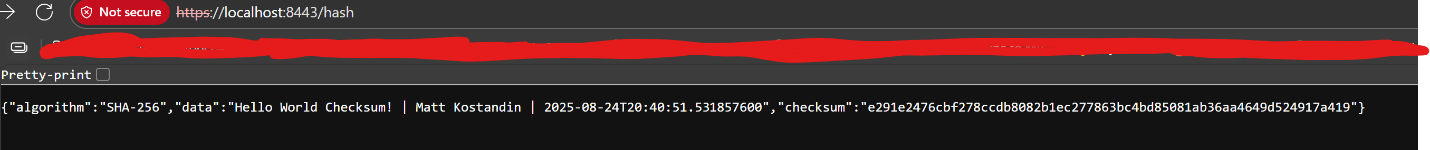
Insert a screenshot below of the checksum verification.

A screen shot of a computer

AI-generated content may be incorrect.

## 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.

A screenshot 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.

A screenshot of a computer program

AI-generated content may be incorrect.

## Summary

I strengthened Artemis Financial’s service by replacing a plain checksum with HMAC‑SHA‑256 and failing fast if the secret is missing to add integrity and authenticity, and by introducing conservative limits on untrusted query inputs to blunt denial‑of‑service and abuse. I framed these changes using the Vulnerability Assessment Process Flow Diagram: first identify the key risks being unauthenticated hashes and unbounded inputs, then assess their likelihood and impact, and finally mitigate them with targeted controls before verifying the results.

I moved sensitive information like secrets and certificate locations out of the code, built the app successfully, and checked that the secure page at https://localhost:8443/hash loads and returns the expected result. This maps to the diagram’s verify step showing the fix works end‑to‑end and prepares us for monitor and maintain, because configuration lives outside the code and can be rotated without redeploying.

## Industry Standard Best Practices

I followed simple, well‑known security basics. I keep connections private by using up‑to‑date HTTPS and turning off older, unsafe versions. For data checks, I use SHA‑256; when I also need to prove the data really came from this service, I use HMAC‑SHA‑256, which adds a secret key only the server knows. I don’t hard‑code secrets in the project. I pass them in as environment variables. I also set sensible limits on the text the endpoint accepts so a single request can’t tie up the server.

Beyond that, I set things up to be easy to run and audit. Settings are centralized, keys can be changed without touching the code, and automated scans review our open‑source libraries for known issues. Together these steps create layers of protection like encrypted transport, signed hashing, input checks, and supply‑chain checks and they support Global Rain’s “security is everyone’s responsibility” principle while keeping the app straightforward to maintai