****

# 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** | **6/21/25** | **Alex Kastigar** |  |

## Client



## Developer

Alex Kastigar

## Algorithm Cipher

For this project, I went with SHA-256 as the hash algorithm. It’s a strong and trusted option that produces a 256-bit output, which means the data it creates is nearly impossible to fake or duplicate on accident. That makes it perfect for verifying that files haven’t been messed with during transfer.

SHA-256 doesn’t use encryption keys, it's a one-way hashing algorithm, not something that locks and unlocks data. So, I used it strictly to confirm that data stayed the same from point A to point B. It's known for being reliable and isn’t considered broken like older ones (MD5 or SHA-1). It’s also still being used everywhere, including in SSL certificates and blockchain, which makes it a safe bet here.

## Certificate Generation

I created a self-signed certificate using Java’s Keytool from inside Eclipse. This is what’s used to enable HTTPS and encrypt traffic between the server and a browser.

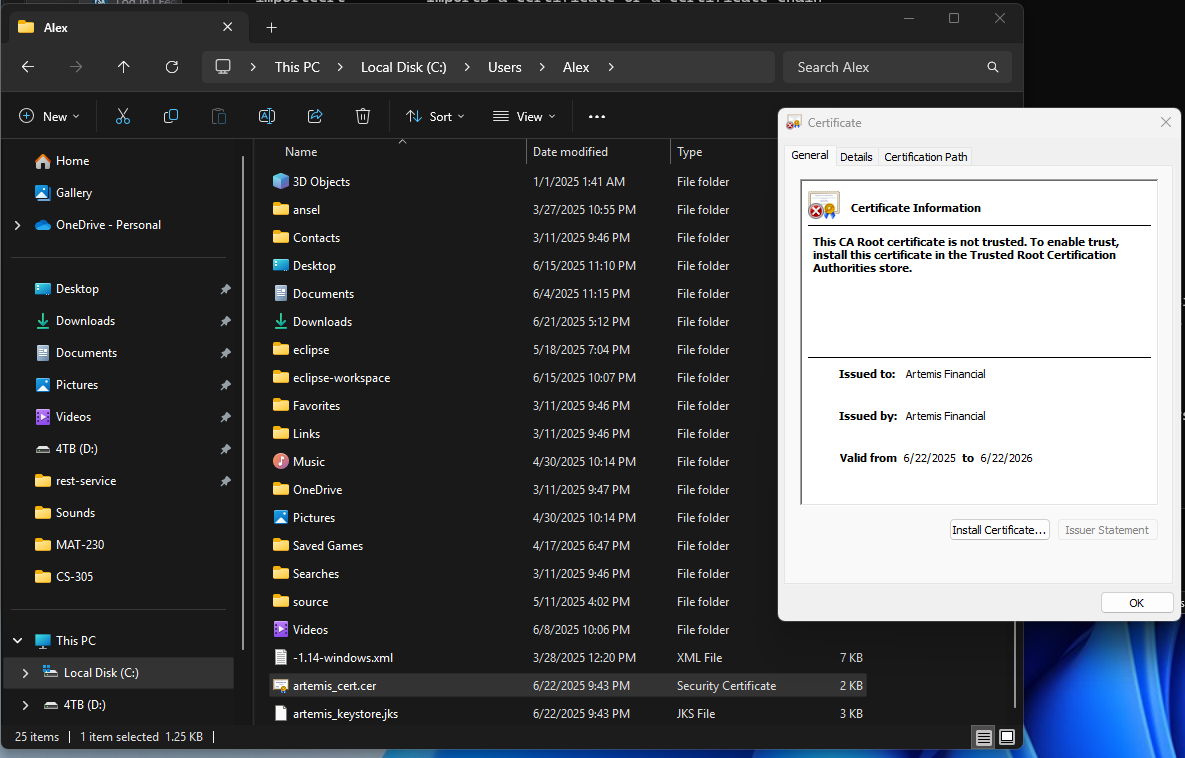
To generate it, I used:

keytool -genkeypair -alias artemisCert -keyalg RSA -keysize 2048 -validity 365 -keystore keystore.jks

Then I exported it to a .cer file with:

keytool -exportcert -alias artemisCert -file artemis.cer -keystore keystore.jksA screenshot of a computer program

AI-generated content may be incorrect.



## Deploy Cipher

I added checksum functionality using SHA-256 to confirm the integrity of transferred files. I tested it by calculating the checksum and matching it against what the application received. If the values matched, it proved that the file wasn’t altered.

A screenshot of a computer

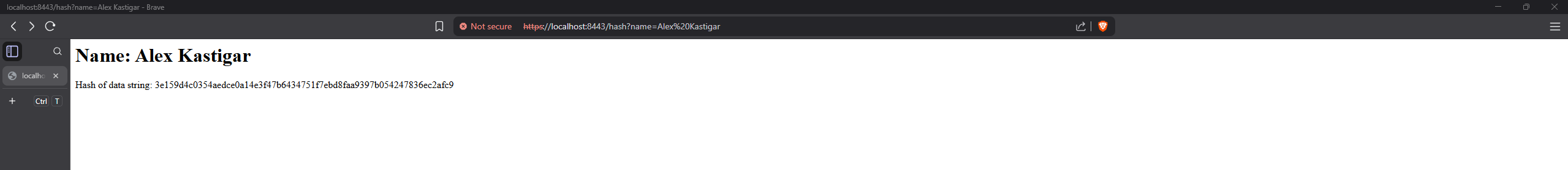
AI-generated content may be incorrect.

*Screenshot of checksum working, including my name and a unique string*

## Secure Communications

I updated the application.properties file to enable HTTPS and linked it to the keystore I generated earlier. Once that was set up, I ran the server and tested the secure connection by visiting:

https://localhost:8443/hash?name=Alex Kastigar



A screenshot of a computer

AI-generated content may be incorrect.

## Secondary Testing

After refactoring the code and adding the new features, I ran the OWASP dependency-check again. This time, I included a suppression.xml file to ignore false positives, like some older libraries that are still secure in the way we’re using them. The key was to make sure I wasn’t introducing any new vulnerabilities.

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Functional Testing

I tested the refactored app to make sure nothing broke. Everything ran smoothly, and all the added security features worked as intended, no syntax or logic bugs popped up.

A screen shot of a computer

AI-generated content may be incorrect.

## Summary

Here’s what I changed to make Artemis Financial’s app more secure:

Added SHA-256 checksum logic to catch tampering

Enabled HTTPS by generating and configuring a self-signed certificate

Suppressed known false positives in the vulnerability report

Verified everything worked and the static testing passed

According to the vulnerability flowchart from class, I focused on cipher selection, certificate setup, communication encryption, and dependency review. Overall, the app is now a lot safer than what we started with.

## Industry Standard Best Practices

While doing this, I followed good software security practices:

I didn’t store passwords in the code.

I used SHA-256 because it’s still widely accepted as secure.

I encrypted the connection using SSL/TLS.

I ran security checks using a trusted OWASP tool.

I made sure no new security problems got introduced.

Doing this matters not just for this app, but for the company’s reputation. Bad security can lead to lost data, fines, or even lawsuits. Good security helps keep things running smooth and customers happy.