****

# 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** | **12/07/2023** | **Jonathan Thomas** |  |

## 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

Jonathan Thomas

## Algorithm Cipher

Given that Artemis Financial wants to add a file verification step as part of their data transfer operations, I recommend choosing the SHA-256 hashing algorithm to generate a checksum which can be used to verify file integrity. SHA-256 is a good choice for generating a checksum given that the algorithm has an “avalanche effect” meaning that even small changes in input will result in large changes to the generated output. Additionally, SHA-256 is designed to be collision resistant, meaning that it is infeasible for two unique inputs to produce identical outputs. Combined, these factors make SHA-256 an excellent choice for generating checksums used for file verification.

The ”256” part of SHA-256 indicates that the hashing algorithm always outputs a 256-bit key, unique to the provided input. Having many bits in the hash output provides a significant degree of randomness and “security in sheer quantity” as the total number of possible unique combinations using 256-bit encryption is equivalent to 2256 possible combinations. The use of random numbers within a hashing algorithm makes it more difficult for malicious actors to reverse-engineer the process by which an output is generated. Even though most “random” numbers are generated using methods that are only pseudorandom, keeping the methods, by which these numbers are produced, obscured can hinder the efforts of bad actors. This randomness paired with the sheer quantity of 256-bit possible outputs ensures the security of the SHA-256 algorithm.

## Certificate Generation

Insert a screenshot below of the CER file.

A computer screen with white text

Description automatically generated

## Deploy Cipher

A black text on a white background

Description automatically generated

## Secure Communications

A screenshot of a computer

Description automatically generated

I had trouble getting Firefox or Chrome to “trust” my self-signed certificate but trying to navigate to the page using http:// results in an error. I took this as an indication that my refactor worked successfully.

## Secondary Testing

## A screenshot of a computer Description automatically generated

To reduce the number of vulnerable dependencies, I updated the dependency check plugin to the most recent available version (9.0.2) and updated the org.springframework.boot version to the highest version that was compatible with Java version 1.8 (2.7.18). A quick Google search indicated that only 2.\*.\* versions of springframework are compatible with Java version 1.8 and according to the release history of springframework’s GitHub page, 2.7.18 is the last version in the 2.\*.\* chain.

That said, 4 vulnerable dependencies still exist.

## Functional Testing

Insert a screenshot below of the refactored code executed without errors.

A screenshot of a computer program

Description automatically generated

## Summary

The first areas of security addressed in the refactored code come from the use of my self-signed certificate to enable use of the HTTPS protocol. Use of this protocol ensures that interactions with the API are secured end-to-end. This guaranteed secure communication also covers the “Client / Server” area of the VAPF as all communication between the client and server are secured using the HTTPS protocol. Finally, the refactored code implements Cryptography, specifically the SHA-256 hashing algorithm to produce a checksum which can be used for validating that users are accessing the file or data they intended to. In our application, the checksum has been generated from the string following the ”data: “ label when navigating to the “/hash” endpoint.

Beyond refactoring the SslServerApplication code, I attempted to add layers of security by updating the dependency check plugin to ensure the most up-to-date information regarding vulnerabilities as well as using the highest available version of the springboot framework that is compatible with the Java version in use for this application to reduce the number of vulnerabilities found in sub-dependencies.

## Industry Standard Best Practices

Best practices for maintaining the security of this software application include running regular dependency checks, keeping all dependencies updated to their most recent versions and possibly upgrading the version of Java used in this application as even the highest available version of springboot framework that is compatible with Java version 1.8 contains vulnerable dependencies.