****

# 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** | **6/23/2024** | **James Furman** |  |

## Client



## Developer

James Furman

## Algorithm Cipher

This encryption cipher will be used to verify files transferred in Artemis’s web application. A hash function algorithmically converts data (in this case, a file being transferred in the webapp) to a smaller data value, so that the integrity of the original data can be verified without duplication. Hash collision, when multiple original values map to the same hashed value, nullifies this functionality in some cases. The frequency of collision can be reduced by good algorithm design, and the ability to crack hash-protected systems with brute force attacks can be mitigated with larger hash sizes. Symmetric keys are used to encrypt data; in such cases, the key is used to transform data into an unreadable and protected scrambled form, and then back into its original form. For this use, asymmetric encryption is appropriate, where a private key is used to encrypt the data and a public key is used to decrypt it; if the public key successfully decrypts a file in an expected fashion, this demonstrates that it was originally encrypted using the private key, verifying that the sender was the party with the private key. For this application, I recommend using Java’s built-in tools to hash with a SHA-256 algorithm. The application will use Java’s built-in random number generator to create keys, meaning that it will create hash keys using constantly changing system information, making the keys effectively impossible to predict or reproduce (i.e. they are practically random). This approach will maximize the hash function’s security against attackers seeking to tamper with the file transfer functionality.

## Certificate Generation

![A screenshot of a computer

Description automatically generated]()

## Deploy Cipher

In this screenshot, TLS has been disabled so that the application server will connect without a certificate; the screenshot shows that, given the string “James Furman”, the hashing function generates a SHA-256 hash. In the release version, instead of a string, the file being transferred would be fed into the hashing function, which would work identically. A screenshot of a computer

Description automatically generated

## Secure Communications

This screenshot, taken after TLS has been reenabled, shows that the self-generated certificate is being read by the browser. “James Furman” isn’t considered a trusted certificate authority by the browser (this could be reconfigured in the browser’s settings), but the screenshot nevertheless shows that TLS is enabled, and the certificate is being checked. ![A screenshot of a computer

Description automatically generated]()

## Secondary Testing

The first screenshot below shows the refactored portion of the application, including new import statements. The second screenshot shows the dependency report header and list of vulnerable dependencies, none of which is referenced in the refactored code (the dependency check html file is in the application’s target folder, alongside a higher-resolution pdf version of the screenshot).

![A screenshot of a computer program

Description automatically generated]()

A document with text and numbers

Description automatically generated

## Functional Testing

The below screenshots show the code after manual review and refactoring.

![A screenshot of a computer program

Description automatically generated]()

![A screenshot of a computer program

Description automatically generated]()

## Summary

The ServerController object has been abstracted to a separate class file, the NoSuchAlgorithm exception catch block in the hashing function has been slightly modified and a note has been added that it should be secured in production, and if the hash function can’t execute properly, it will now return null instead of returning the input value, preventing injection or other potential unsecure behavior downstream.

This refactoring addressed code quality and cryptography. While no vulnerable dependencies needed to be changed, the addition of the OWASP dependency check to the program’s pom file, assuming it is monitored going forward, also addresses API security.

At each step of the process of adding security layers through refactoring, I add working security functionality, create dummy functionality (e.g. making stubs, disabling TLS) so that the ongoing addition can be implemented and tested at a basic level, and make sure to use secure coding for whatever code is implemented and document any necessary future changes. Then, it is possible to return to this portion of the program at a later stage of development and, in reference to the code documentation, smooth out its integration with the more advanced stage the application has reached.

## Industry Standard Best Practices

I applied industry standard best practices in using a secure hashing algorithm, ensuring dependencies are up-to-date, coding securely (in this case, with abstraction/encapsulation), and documenting my code. Using secure tools like a SHA-256 algorithm reduces the likelihood they will fail, keeping dependencies updated reduces the likelihood that they will contain vulnerabilities, secure coding reduces the likelihood that the application code will contain vulnerabilities, and documentation enables developers to easily implement secure code later.

Using industry best practices improves the odds that attackers will fail to exploit vulnerabilities in Artemis’ software. This safeguards against potential financial, reputational, or legal consequences that could profoundly damage the company and its shareholders’ and employees’ wellbeing. Further, best practices reduce the accrual of technical debt, meaning that the software we create now will be easier and cheaper to maintain in the future, for purposes of improving functionality and of maintaining security.