****

# 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** | **10/19/2024** | **Alexander Flood** |  |

## 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

Alexander Flood

## Algorithm Cipher

Artemis Financial develops different financial products for their clients, including savings, retirement, and investment products as well as insurance. Encryption is crucial to protecting sensitive financial data, ensuring secure communication between clients and the company’s web application.

The use of encryption to protect communications dates back hundreds of years, but modern cryptography began in the 1970’s. RSA (Rivest-Shamir-Adleman) was introduced in 1977 and was revolutionary for encryption by allowing the use of public-key cryptography. Before RSA, symmetric encryption (where the same key was used for encryption and decryption) was the only method available, which posed many challenges for secure key distribution (Kelly, 2009).

Today, RSA remains a cornerstone of modern encryption. RSA supports variable key lengths, with 2048-bit keys commonly used for strong security (*Java Security Standard Algorithm Names*, n.d.). RSA is an asymmetric encryption algorithm which means it uses two keys, public and private. The public key is used to encrypt the data, while the private key is used to decrypt the data. The larger the key size, the more secure the encryption is, but this also comes with a trade-off in terms of computer processing overhead (RSA Encryption: Definition, Architecture, Benefits & Use | Okta, n.d.). This makes it very difficult, almost impossible, for an attacker to decrypt the data without the private key.

This form of encryption is essential in the modern digital landscape, where data integrity and security are critical, especially for financial institutions such as Artemis Financial. RSA provides a foundation for various protocols which secure internet communications and are integral to the modern web.

RSA relies on the generation of large, random prime numbers to create the key pair (RSA Encryption: Definition, Architecture, Benefits & Use | Okta, n.d.). The public key can be shared openly, while the private key must remain confidential. This key system allows for secure communication over potentially insecure networks, such as the internet.

This type of encryption will help Artemis Financial ensure that client data is securely transmitted and cannot be hijacked or accessed by attackers. By using RSA for encrypting sensitive client communications, Artemis Financial can guarantee the integrity of financial information being shared through their web application, reassuring their clients and providing confidence in the Artemis Financial brand.

## Certificate Generation

I generated the certificate using the following command and answering the following prompt.

A screenshot of a computer program

Description automatically generated

Then we can export the certificate to a CER file using the following command.

A screenshot of a computer

Description automatically generated

This confirms that our self-signed certificate was correctly generated and exported to our CER file. We can now use this certificate for our Artemis Financial web application to secure communications over HTTPS and ensure the protection of sensitive client data.

## Deploy Cipher

## Using SpringBoot Security the RSA cipher can be implemented to secure and protect data in transmission from the web application.

Below you will see a verification of the cipher in action through a checksum I implemented in the web application project. We will review the code used to implement the cipher and checksum later in this document as part of a code review.

A screenshot of a computer

Description automatically generated

You will notice that https has not yet been implemented. It will be implemented in the next stage.

## Secure Communications

## To enable https, we will need to configure the web application to utilize the self-signed certificate we generated earlier in this process. Open the application.properties file inside the resources folder and update the fields to point towards your keystore with the appropriate password and alias.

## A close-up of a computer screen Description automatically generated

## To run the server with these new settings, click the run button in the menu bar of Eclipse.

## A screenshot of a computer Description automatically generated

## After a few moments, you will see the following output in your integrated terminal.

## A close-up of a screen Description automatically generated

## Using a webserver to navigate to <https://localhost:8443/hash>, we will now see that the connection is secured, and our cipher is functional.

A screenshot of a computer

Description automatically generated

## Secondary Testing

The next step is to run a vulnerability assessment using the OWASP Dependency Check tool, which will analyze the project dependencies and report known vulnerabilities (OWASP Dependency-Check | OWASP Foundation, n.d.).

Right-click the pom.xml file, hovering over the ‘run as’ sub menu, and selection ‘6 Maven verify’.

A screenshot of a computer

Description automatically generated

After running this command, a dependency-check-report.html file will be created in the /target subdirectory. This will show us the vulnerabilities that currently exist for the dependencies that make up our project pulls during the maven build process.

A screenshot of a computer

Description automatically generated

Because the project does not currently make use of snakeyaml-2.2.jar or springwebmvc-6.1.13.jar, we can create a suppression.xml file to tell the Dependency Check tool to suppress these vulnerabilities, as they are not applicable to the project. This will allow developers to focus on the relevant vulnerabilities that might appear later in the project.

To do this we can right click on the suppress vulnerability button next to a vulnerability we wish to suppress.

A screenshot of a computer

Description automatically generated

Then for the first suppression, click “Complete XML Doc”.

A screenshot of a computer code

Description automatically generated

After suppressing the vulnerabilities that are not relevant to the project, we can run the tool again and see that we are left with 0 vulnerabilities.

Past this into a suppression.xml file in the project folder for the project.

A screenshot of a computer

Description automatically generated

Repeat this process for every vulnerability you wish to suppress, adding the highlighted XML code to the suppression.xml document. By the time you are finished suppressing the vulnerabilities that do not apply, it should look something like this.

A screenshot of a computer program

Description automatically generated

The file can grow to be quite long but is a smart investment to save and focus developer time on vulnerabilities that impact the project. After completing the suppression.xml file, we can run the Dependency Check tool again to see that the project no longer has any notable vulnerabilities.

A screenshot of a computer

Description automatically generated

Below is the clean vulnerability report, giving the project a solid base to build out future functionality for the Artemis Financial web application.

A screenshot of a computer

Description automatically generated

## Functional Testing

The refactored code included an additional Java class, NewRestController. This controller handles requests to the /hash endpoint. It generates a SHA-256 checksum for a static string that returns my name along with the computed checksum, demonstrating the secure processing of data.

A screenshot of a computer code

Description automatically generated

The generateChecksum function uses the SHA-256 algorithm I selected to compute a cryptographic hash of the input data. The byteArrayToHex function converts the resulting byte array into a readable hexadecimal string, making it easier to display the checksum. Together, these helper functions ensure secure data processing and output.

The last step I did was update the project to a more recent version of Java and SpringBoot. I updated the Java version from 1.5 to 15. This provides a more secure and performant Java run time which supports newer libraries with less vulnerabilities. To update the Java version, I changed the <java.version>1.5</java.version> tag to <java.version>15<java.version> in the pom.xml file.

A text on a white background

Description automatically generated

To update SpringBoot I updated the <version>2.2.4 RELEASE</version> tag to <version>3.3.4</version>, which provides a more recent and secure version to build the project on while retaining the same functionality.

A screen shot of a computer code

Description automatically generated

The refactored code executes with no errors.

A close-up of a screen

Description automatically generated

## Summary

In this project, the code was refactored to comply with secure coding protocols by incorporating a range of security measures, including the use of RSA encryption for secure data transmission and SHA-256 for checksum verification. During the refactoring process, areas of vulnerability were identified and addressed to mitigate potential security risks.

Using the vulnerability assessment flow, we implemented cryptography by enabling RSA encryption to secure the data provided by the web application endpoints. A manual code review was performed to ensure that no memory errors were present, and the code compiled to satisfaction.

The OWASP Dependency Check tool was used to identify and analyze known vulnerabilities in the project’s dependencies. It was found that two libraries contained a total of four vulnerabilities not relevant to the project, snakeyaml-2.2.jar and spring-webmvc-6.1.13.jar. These vulnerabilities were suppressed using a suppression.xml file to allow more development and security focus on relevant vulnerabilities that might appear in the future.

The SHA-256 checksum was implemented to ensure data integrity, verifying no unauthorized modifications could be made during data transmission.

This refactoring process added additional layers of security through encryption, checksum verification, and secure communication, all which serve a secure foundation for serving financial information over the internet. By using these techniques, we addressed security concerns in the existing system and enhanced the overall security of the application.

## Industry Standard Best Practices

Throughout this project, industry standard best practices for secure coding were used to mitigate security vulnerabilities and enhance the project’s existing security. This included adding RSA encryption which follows the best practice of using public-key cryptography to protect sensitive information. RSA is widely regarded as one of the most secure asymmetric encryption algorithms.

Secure communication (HTTPS) was implemented via SSL/TLS, which ensures encrypted communication between the client and the server (Detlefsen & Manico, 2014). This is a critical best practice for any web application that handles sensitive data, especially inside the financial industry.

Dependency management was implemented by utilizing the OWASP Dependency Check which ensures any vulnerability in an external library is detected and analyzed properly. By suppressing non-applicable vulnerabilities and focusing on critical issues, the developers can better maintain a clean and secure codebase, following best practices for dependency management.

The implementation of the checksum validation using SHA-256 adds a layer of integrity verification, ensuring data has not been tampered with in transit. This follows a best practice for ensuring data integrity between the client and server, providing users trust their information is not being altered by an attacker.

Applying these industry standard best practices not only improves security of the web application, but also contributes to the overall trusted reputation of Artemis Financial, which is essential in the financial industry. Adhering to secure coding practices helps Artemis Financial protect its clients’ sensitive data from cyber attacks while ensuring compliance with government regulations and industry standards such as the PCI DSS (Payment Card Industry Data Security Standard) which helps protect payment card account data from data breaches (Understanding Payment Card Industry Data Security Standard (PCI DSS) | Controller’s Office, n.d.). In the competitive financial industry, demonstrating commitment to security is essential for a company’s long-term success.

**References**

*dependency-check – Suppressing False Positives. (n.d.).* [*https://jeremylong.github.io/DependencyCheck/general/suppression.html*](https://jeremylong.github.io/DependencyCheck/general/suppression.html)

*Detlefsen, A., & Manico, J. (2014). Iron-Clad java: Building Secure Web Applications. McGraw-Hill Education.*

*Java Security Standard Algorithm Names*. (n.d.). <https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#cipher-algorithm-names>

*Kelly, M. D. (2009, December 7). The RSA algorithm: A mathematical history of the ubiquitous cryptological algorithm. Swarthmore College.* [*https://www.sccs.swarthmore.edu/users/10/mkelly1/rsa.pdf*](https://www.sccs.swarthmore.edu/users/10/mkelly1/rsa.pdf)

*OWASP Dependency-Check | OWASP Foundation. (n.d.).* [*https://owasp.org/www-project-dependency-check/*](https://owasp.org/www-project-dependency-check/)

*RSA Encryption: Definition, Architecture, Benefits & Use | Okta*. (n.d.). Okta, Inc. <https://www.okta.com/identity-101/rsa-encryption/>

Understanding Payment Card Industry Data Security Standard (PCI DSS) | Controller’s Office. (n.d.). https://controller.ucsf.edu/how-to-guides/accounting-reporting/understanding-payment-card-industry-data-security-standard-pci