****

# 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** | **October 19, 2025** | **Justin Hardin** |  |

## 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

Justin Hardin

## Algorithm Cipher

For the Artemis Financial application, my recommendation is to implement SHA-256 as the hash function for the checksum verification. SHA stands for Secure Hash Algorithm. SHA-256 belongs to a family of cryptographic hash functions that were created by the National Security Agency (NSA) in collaboration with the National Institute of Standards and Technology (NIST) in 2001 (Encryption Consulting, 2024). It was created as a successor and improvement upon the SHA-1 family, which faced brute force attack vulnerabilities. The “256” in the name represents the fact that SHA-256 produces a fixed hash size in a 256-bit value no matter what the input is. This ensures cryptographic hashing that is both consistent and secure. It is commonly used for creating and verifying certificates issued by Certificate Authorities as it helps ensure the authenticity and integrity of these certificates (Encryption Consulting, 2024), making it a good choice for our needs.

From a technical standpoint, SHA-256 works by converting our data string to binary, then initializes the initial hash values, which are eight 32-bit words. The message then gets divided into 512-bit blocks; each one it performs a series of bitwise operations, modular additions, and logical functions using the current hash value and the block (Encryption Consulting, 2024). Then a compression function is applied to each block, which involves mixing the bits of the hash value and the message block. The compression function is iterated for each block and uses the output of each iteration as input for the next one. After all blocks are processed, a final hash value is output as the SHA-256 hash of the original message.

Since we’re using a cryptographic hash function instead of an actual encryption algorithm, we don’t rely on keys or random numbers in the traditional sense. However, to ensure a secure communication channel, we use asymmetric keys with our SSL/TLS certificate, which is where a public key encrypts while a private key decrypts. This is in contrast to symmetric keys, where the same key encrypts and decrypts. Using asymmetric keys is slower but allows for a secure key exchange. The asymmetric keys ensure secure communication while the hashing is for verifying data integrity.

Historically, the SHA family started with SHA-0 in 1993 when NIST published “Secure Hash Standard” (Debnath et al., 2017). It only later became known as SHA-0. SHA-1 was released in 1995, improving on SHA-0, but there were many vulnerabilities discovered. It wasn’t until 2001 that the SHA-2 family was released, and after years of support and improvement, it has become an industry standard, especially in the financial field. The SHA-3 family released in 2015, however it hasn’t been adopted widely as of yet.

So, all in all, SHA-256 is a great choice for the needs of Artemis Financial due to being an industry standard in financial fields, being widely supported across platforms, and its fast performance, making it great for checksum verification and data integrity.

## Certificate Generation

Insert a screenshot below of the CER file.

Below is the generation of our key using Java’s keytool. It generates a key, exports it to a .cer file, and prints the certificate for use.

Generating the Key

A computer screen with text on it

AI-generated content may be incorrect.

Export to .cer file



Printing the certificate

A computer screen with white text

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

Below is a screenshot capturing the data string used as input for our hash algorithm and the SHA-256 checksum output.

A screenshot of a computer

AI-generated content may be incorrect.

## Secure Communications

Insert a screenshot below of the web browser that shows a secure webpage.

A screenshot of a computer

AI-generated content may be incorrect.

I’ve included a screenshot using the Firefox browser. I imported the self signed certificate as a trusted certificate for the purposes of showing a secure connection in the browser. For some reason, this tactic did not work when using chrome. This seems to be the reality of using a self-signed certificate instead of a public CA.

A screenshot of a computer

AI-generated content may be incorrect.

## Secondary Testing

Insert screenshots below of the refactored code executed without errors and the dependency-check report.

Below is a screenshot of the console output of the application. It shows a successful run, using Tomcat to establish a secure connection.

A screenshot of a computer

AI-generated content may be incorrect.

Below is a screenshot of the dependency check report. There were many vulnerabilities found due to using an outdated framework and dependencies. These should clear right up once updated. Fortunately, our refactored code didn’t add any new vulnerabilities.

A screenshot of a computer error

AI-generated content may be incorrect.

Below is a screenshot of the dependencies for which a vulnerability was found. These are due to an out of date Spring framework and out of date dependencies.

A screenshot of a computer

AI-generated content may be incorrect.

## Functional Testing

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

Below are the screenshots capturing the entirety of the refactored code for SslServerApplication.java.

A computer screen shot of a program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

Below is a screenshot of the console output of the running application. I included the “Problems” window to show that there are no errors when running the application.

A screenshot of a computer

AI-generated content may be incorrect.

## Summary

During the refactoring of the code, I tried to address different layers of security while providing the required functionality. I created a function, getChecksum(), that returns the checksum using a hard-coded data string. The returned checksum is returned by calling another function, generateHash(), that generates the hash, feeding in the data string as a parameter. To generate the hash, the function, generateHash() creates a message digest instance that turns the data string into bytes, then calls another function, bytesToHex() that converts the bytes into hex characters and is returned as a string. I used a try/catch format in generateHash() to ensure proper error handling. Once the checksum is returned, I call one final function, buildHtmlResponse() that uses the data string as one parameter, and the checksum as another. This functions returns html that is used to display the data string that was used, as well as the checksum that was created from that data string on the webpage.

Addressing the vulnerability process flow diagram:

Input validation – for the demonstration purposes of this project, I hard-coded the data string used for generating the checksum. This ensures data integrity. However, when in production and relying on user input for the data string, we’ll need to ensure proper data validation is in place for input to ensure that there can be no data injection vulnerabilities. This will include a limit to the number of characters that can be used.

APIs – We used the Java Security API, specifically our use of the message digest, Spring boot framework to handle http/https requests, and SSL/TLS APIs. All of these are well trusted. Our only vulnerabilities are due to them being outdated, but that can be resolved easy enough by upgrading each of them that are out of date.

Cryptography – We implemented SHA-256 cryptographic hash function to ensure data integrity, creating a checksum verification endpoint for data validation.

Client/Server – We converted http to https protocol for proper encrypted communication. We implemented SSL/TLS using a self-signed certificate. We configured our server to require secure connections on port 8443.

Code Error – Our use of try/catch in the generateHash() function is a direct step taken to ensure proper error handling.

Code Quality – We used secure coding practices by implementing exception handling, using Java’s built in MessageDigest and the code was functional, executing successfully and without error.

Encapsulation – Making the helper functions private ensures that only the relevant part of the application knows about these functions. This prevents external access which reduces attack opportunities.

The first layer of security was ensuring the secure transport of data by implementing https. The data transmitted between the client and the server is encrypted which prevents eavesdropping attacks. The self-signed certificate provided encryption for development purposes. Eventually a public CA-signed certificate should be used.

The next layer is data integrity. By using SHA-256 to implement checksum verification, it allows clients to verify that the data has not been tampered with during its transmission and storage.

Code security is the next layer. I followed secure coding practices, using standard library functions for cryptography. I implemented proper error handling to prevent unwanted leakage of information. I avoided some common vulnerabilities like data injection and data overflow. I tried to structure the code in an easy-to-read way to make maintaining it as painless as possible, as well as making security auditing easier.

Another layer of security I used was configuration of the app. I made sure the application.properties app was properly configured. I made sure communication is only possible by https and that the certificate is valid for a reasonable amount of time.

Each of these layers were verified by testing functionality, running the dependency check to identify vulnerabilities, did a manual code review to ensure that I didn’t overlook anything concerning, and verified a secure connection to the server.

## Industry Standard Best Practices

I tried to follow industry standard best practices by using layered security measures. I ensured that the transport of data was secure by using https and ssl. I made sure the application itself was secure by using checksum verification. And at the code level, I used error handling and input validation. This is important because if one layer gets compromised, there are other layers to provide protection. I used standard libraries in the code. These libraries have been vetted for years and are well trusted. I made sure the application was configured properly in the application.properties file. I used input validation by hardcoding the data string used for our checksum verification. I tried to use code that’s easy to read and maintainable.

Using these best practices should ensure that Artemis Financial feels confident in our approach. By using a layered approach to security, they know there are multiple approaches to security that work together. Their clients depend on secure financial planning services so they must be able to trust Artemis Financial. And by extension, Artemis Financial must be able to trust us, and I believe that our thorough use of best practices and industry standard approaches, this will build a foundation of trust between all parties.

**References:**

Encryption Consulting. (2024, March 4). *What is SHA- 256? | Encryption Consulting*.

Encryption Consulting.

<https://www.encryptionconsulting.com/education-center/sha-256/>

Debnath, S., Chattopadhyay, A., & Dutta, S. (2017). Brief review on journey of secured hash algorithms. *2017 4th International Conference on Opto-Electronics and Applied Optics (Optronix)*.

<https://doi.org/10.1109/optronix.2017.8349971>

Manico, J., & Detlefsen, A. (2015). *Iron-clad Java building secure web applications*. New York, Ny Mcgraw-Hill Education.

‌

‌

‌