****

# Practices for Secure Software Report

**Harsh**

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** | **04/20/2025** | **Harsh Patel** | **Project 2** |

## 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

Harsh Patel

## Algorithm Cipher

Recommended Cipher: SHA-256.

SHA-256 (Secure Hash Algorithm 256-bit) is a widely used cryptographic hash function and a member of the SHA-2 family, developed by the National Security Agency (NSA) and standardized by NIST. While not an encryption algorithm in the traditional sense, it’s often referred to as a cipher because of its role in ensuring data security. Unlike encryption, which transforms data to keep it confidential, SHA-256 is designed to validate data integrity by generating a unique fixed-length digital fingerprint for any input. This makes it especially useful for applications like checksum verification, digital signatures, and ensuring that files haven’t been tampered with during transmission—an ideal fit for Artemis Financial’s need to verify files during secure communication.

As a hash function, SHA-256 always produces a 256-bit (32-byte) output, regardless of the input size. Even a minor change in the input will produce a drastically different output, a property known as the avalanche effect. It’s also a one-way function—once data has been hashed, it’s practically impossible to reverse the process and retrieve the original input. This is critical for security, as it means that sensitive data like passwords or confidential records can be hashed and compared without ever needing to expose the original values.

SHA-256 is a deterministic, keyless algorithm, meaning it does not use keys or random numbers in its operation. However, understanding how random numbers and keys are used in other types of encryption helps clarify SHA-256’s role. In cryptography, random numbers (often generated by secure random number generators) are essential for producing unpredictable keys, initialization vectors, and nonces. These elements strengthen encryption by making patterns harder to detect. Symmetric encryption algorithms—such as AES—use a single shared secret key to both encrypt and decrypt data. Asymmetric encryption, on the other hand, involves a pair of keys: a public key for encryption and a private key for decryption, with RSA being a well-known example. While SHA-256 doesn’t use keys in this way, it’s frequently used alongside these encryption methods to validate the integrity of encrypted data.

Historically, hash functions have evolved to counter growing threats. SHA-1, once the standard, became vulnerable to collision attacks, prompting the development and adoption of SHA-256. Since its introduction in 2001, SHA-256 has become a cornerstone of modern cybersecurity. It's used in technologies ranging from SSL/TLS and secure file storage to blockchain networks like Bitcoin. Today, SHA-256 is considered robust, with no practical vulnerabilities discovered that undermine its security for most applications. Its strong resistance to collision and pre-image attacks makes it a trusted choice for securing communications and validating file integrity—particularly useful in scenarios like Artemis Financial’s file verification step.

## Certificate Generation

Insert a screenshot below of the CER file.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a certificate

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

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.

## Secondary Testing

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

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Functional Testing

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

A screenshot of a computer screen

AI-generated content may be incorrect.

## Summary

The primary goal of the refactoring was to address potential vulnerabilities in the application and implement best practices for secure software development.

One of the first major changes I made was to **enable HTTPS** for secure communication between the client and server. This was crucial in ensuring that sensitive data would be encrypted during transmission, thus preventing man-in-the-middle (MitM) attacks. I updated the application’s application.properties file to include the necessary SSL configurations, such as specifying the key store and password, and setting the server port to 8443 to enforce the HTTPS protocol. By doing so, I ensured that all data exchanged with the server would be securely encrypted, adding a robust layer of protection against unauthorized interception.

I also focused on improving the security of the data being transmitted by implementing a secure cryptographic hash algorithm. For this, I used the **SHA-256** hashing algorithm, which is a secure, one-way hash function that transforms the original data into a fixed-length, irreversible hash. The hash is sent instead of the actual data, making it computationally difficult for any attacker to reverse-engineer the original content, even if they manage to intercept the transmission. This further strengthens the security of the application by ensuring that no sensitive data is exposed.

Additionally, I made sure that the application’s **API endpoints** were properly secured by enforcing HTTPS on the /checksum route. This means that the endpoint is only accessible via a secure connection, further reducing the risk of data being exposed over an insecure HTTP connection. By refactoring the code in this way, I ensured that the application adhered to secure communication protocols and minimized the potential for security breaches.

To ensure that the application was free from common vulnerabilities, I followed a thorough vulnerability assessment process. This process focused on areas like injection attacks, data exposure, and proper input validation. Since the application doesn’t directly interact with databases, the primary concern was ensuring that data was never exposed in plaintext. By using SHA-256 for hashing, I made certain that even if an attacker intercepts the data, they would only be able to see a hashed value, which is difficult to reverse.

In conclusion, through these refactorings, I not only improved the security of the application by securing the communication channels and ensuring proper data protection, but I also adhered to industry-standard protocols for secure software development. The changes I made help to mitigate risks associated with common vulnerabilities and ensure that the application complies with best practices for secure software development.

## Industry Standard Best Practices

In refactoring the SSL server application, I focused on applying industry-standard best practices for secure coding to maintain and enhance the application's security. One key area of focus was ensuring **secure communication** between the client and server. To do this, I enforced the use of **HTTPS** by configuring SSL (Secure Socket Layer) for the server. This ensured that any data transmitted between the client and the server was encrypted using strong cryptographic protocols, which is a critical security best practice. This helps protect against man-in-the-middle (MitM) attacks, where an attacker could intercept or manipulate the data being exchanged between the server and client.

Another standard security practice I incorporated was the use of **strong cryptographic algorithms** for data protection. Specifically, I utilized **SHA-256**, a secure hashing algorithm, to ensure that sensitive data, such as checksum values, was never transmitted in plain text. This hashing approach follows the best practices for protecting data from unauthorized access and ensures data integrity by making it impossible to reverse the hash back to its original form. This helps mitigate risks like data exposure or tampering during transmission.

Additionally, I ensured that the application’s API endpoints were secured by enforcing **HTTPS** for sensitive routes. This minimizes the risk of attackers exploiting vulnerabilities in the application by preventing communication over unencrypted HTTP, a common target for exploitation. By implementing these best practices, I adhered to the principle of **least privilege** by ensuring that only authorized users could access secure resources over encrypted connections.

Applying industry-standard best practices for secure coding has significant value for a company’s overall well-being. First, by ensuring that the software is secure from the outset, companies can reduce the likelihood of security breaches, which can lead to costly data leaks, financial losses, and damage to reputation. Security vulnerabilities in software can serve as entry points for cyberattacks, and applying best practices proactively helps close those doors. This, in turn, builds trust with customers and stakeholders, as they are more likely to engage with a company that prioritizes secure development practices.

Furthermore, following established secure coding practices ensures **compliance with industry regulations** and **standards**, such as GDPR, HIPAA, or PCI DSS. Non-compliance can result in legal penalties and loss of business, so maintaining security throughout the development process is crucial to staying on the right side of the law. It also reduces the risk of negative publicity associated with security incidents, which can have long-term impacts on customer loyalty and the company's public image.

Ultimately, applying these best practices not only protects the software but also safeguards the company’s reputation, financial standing, and regulatory compliance, contributing to its long-term success. It also ensures that the company is prepared to respond effectively to emerging security threats and that it can build more resilient systems for the future.