****

# 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** | **16 August 2025** | **Dylan Dunagan** |  |

## 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

Dylan Dunagan

## Algorithm Cipher

* I recommend the AES (Advanced Encryption Standard) as the encryption algorithm for Artemis Financial. AES uses the same key for both encryption and decryption, which makes it a symmetric encryption cipher. Since Artemis Financial deals with customers’ financial and personal information, it is important to use an encryption cipher that provides confidentiality and maintains its integrity. AES is used in a vast array of applications because of its reliability and its resistance to modern attacks.
* AES has various sizes which allows it to maintain compatibility as well as balance its performance. AES’s main sizes are 128, 192, and 256 bits. I recommend SHA-256 for checksum and data verification. SHA-256 produces a 256-bit cryptographic hash, ensuring that even minor changes in data will create a dramatically different hash. Combining SHA-256 with AES allows for the verification of data integrity and confidentiality.
* AES is a symmetric cipher. So, it uses the same confidential key for both encrypting and decrypting. This makes the cipher incredibly secure, making it perfect for encrypting large datasets, especially financial records. Although it’s secure, key distribution can have its own problems. The sender and receiver must have access to the same key, but to prevent unauthorized access, the key must be stored and transmitted in a secure way. To do this, AES uses random number generating to create initialization vectors. This makes each transaction generate its own unique initialization vector, meaning that the same plaintext encrypted twice will produce two completely different vectors. Since I’ve recommended the 256-bit AES encryption, that ensures that each key is significantly longer than others, providing exponentially stronger protection against brute force attacks. The 256-bit key combined with the randomness of the initialization vectors protect against attacks that exploit patterns and data repetition. Another way that AES can securely distribute keys is through non-symmetric encryption. This means that a public key encrypts the symmetric key and only a private key can decrypt it. Essentially the data gets encrypted with a symmetric key, that gets further encrypted with a non-symmetric public key, the receiver can then decrypt with the non-symmetric private key, and then finally finish the decryption with the same symmetric key that it was originally encrypted with. This combination ensures that maximum amount of security for distribution.
* AES is currently the industry standard when it comes to encrypting sensitive data, whether it’s financial, healthcare related, or government data. AES’ cipher key length and strength allowed for different levels of security that proved to be much better than its predecessor DES (data encryption standard) which only had 56-bit keys. The best practices today are to use AES-256, use random number generating, combine AES with asymmetric encryption, and review dependencies.

## Certificate Generation

Insert a screenshot below of the CER file.

A screenshot of a computer

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.

From researching, I realized that my browser is using https, but it says “not secure” because I am using a self-signed certificate, not because it isn’t using https. I attempted to make the certificate trusted locally but it still would not get Google Chrome to trust the self-signed certificate.

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 screen shot of a computer code

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 screen shot of a computer program

AI-generated content may be incorrect.

## Summary

Following the vulnerability flow diagram:

* Input Validation & APIs: Using “NoSuchAlgorithmException” prevents potential runtime errors even though the checksum endpoint uses a static string.
* Cryptography: SHA-256 was chosen as the algorithm for data verification that is resistant to collisions. Base64 encoding was used for consistent output.
* Client/Server: HTTPS was enabled using a self-signed certificate, ensuring secure communication between client and server.
* Code Error & Code Quality: The refactored code runs without compilation or runtime errors.
* Encapsulation & Controllers: All new code was encapsulated in the ChecksumController class.
* Static Testing & Manual Review: No new vulnerabilities were added after the addition of the refactored code.

As for adding layers of security, by using the AES algorithm along with the SHA-256, it creates a strong encryption for all data so that transmitted data is verified that it hasn’t been altered. Using HTTPS further encrypts communications. All operations are using try/catch methods, preventing runtime and handling errors. Encapsulating everything under the ChecksumController class maintains a modular design and reduces risks of errors being introduced elsewhere in the future.

## Industry Standard Best Practices

While I added new code to the application, I focused on simplicity and functionality while ensuring the application stayed secure. I maintained modular coding by encapsulating all checksum logic under @RestController. I implemented SHA-256 for checksum, which follows NIST’s recommendation for secure cryptography. I used proper error handling so that exceptions could not expose any information. And additional static testing with the OWASP dependency-check confirmed that there were no new vulnerabilities. Applying proper secure coding practices reduces risks of breaches from attacks and protects the clients’ trust. Applying proper coding practices ensures that your code remains scalable for the future. Overall, following the industry best practices adds layers of security and maintains a culture and a standard for building reliable and safe software.