****

# 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** | **[Date]** | **Adam Kramez** |  |

## 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

Adam Kramez

## Algorithm Cipher

[Insert text.]

**Step 1: Algorithm Cipher Recommendation for Artemis Financial**

**A. High-Level Overview of the Encryption Algorithm Cipher**

For Artemis Financial's web application, I recommend the Advanced Encryption Standard (AES) cipher.

AES-256 is a symmetric encryption algorithm widely used by the most preeminent institutions (such as various US governmental agencies, large financial institutions like JP Morgan) for securing sensitive data (Shah, 2023). It’s a symmetric algorithm, utilizing the same key for both encryption and decryption: both the sender and receiver must share the secret key to access the data. The key itself is generated using random numbers to make sure it’s unpredictable, reinforcing security against attacks. AES operates on fixed-size blocks of data (128 bits) and uses a series of transformations (14 rounds for AES-256) to encrypt the original data, making it extremely difficult for unauthorized users to decrypt without the key. The combination of a long key length and the use of random numbers for key generation makes AES-256 extremely resistant to brute-force attacks.

**B. Hash Functions and Bit Levels of the Cipher**

Though AES is primarily an encryption algorithm, it is often used alongside cryptographic hash functions to ensure data integrity and verification, particularly SHA-256 (Rais Rabtsani et al., 2024). To briefly explain, a hash function is an algorithm that transforms an input into a fixed-length sting of bytes. In the case of SHA-256, it generates a 256-bit hash value, regardless of the size of the input. A hash function is deterministic: the same input will always produce the same output. Even a tiny change in the input (like modifying a single letter) will result in a completely different hash value, meaning it is extremely sensitive to input and thus very useful for verifying data’s integrity.

Regarding AES bit levels, the cipher supports three different key lengths: AES-128, AES-192, AES-256. AES-128 offers fairly strong security with the fastest encryption speed, AES-192 offers a balance between security and performance, and AES-256 provides the highest level of security but with slightly slower performance.

For Artemis Financial, it is recommended that AES-256 in combination with SHA-256 for checksum verification is used. This will ensure that both data encryption and integrity verification meet modern security standards, making it difficult for attackers to compromise or alter financial data during transmission. As explained, AES-256 provides a high level of encryption security that is resistant to most known cryptographic attacks. Furthermore, SHA-256 will be effective for creating checksums to verify file integrity, ensuring data has not been modified during transfer. Together, AES and SHA-256 will ensure that client data is both encrypted and verifiable, aligning with Artemis Financial's security requirements.

Random numbers, symmetrical vs asymmetrical algorithms and how it relates to the AES recommendation

Random numbers play an integral role in encryption (Stipcevic, 2012). They are essential for key generation, ensuring that encryption keys in symmetric algorithms like AES are unique and unpredictable, which protects against brute-force attacks. In asymmetric encryption, random numbers generate public-private key pairs, maintaining uniqueness across keys. Initialization vectors used to initialize algorithms also rely on randomness to add unpredictability to the encryption process. Additionally, there is the concept of salts: random numbers are used when hashing data, such as passwords, to prevent rainbow table attacks by ensuring that identical passwords produce different hashes. Furthermore, nonces, which are random values, help maintain uniqueness for transactions or messages, preventing attacks in protocols like TLS and various authentication systems.

Symmetric and asymmetric encryption are two different categories of cryptographic key usage. Symmetric encryption employs a single key for both encryption and decryption, requiring that both the sender and receiver share this key beforehand. Symmetric algorithms are popular for their speed and efficiency, making it ideal for encrypting large volumes of data, such as financial records (which suits Artimis’s needs well). However, symmetric encryption key consideration: its keys need to be distributed securely, as sharing the key between parties can compromise security if intercepted. As it can be imaged, managing keys in large organizations can become complex.

On the other hand, asymmetric encryption uses a pair of keys: a public key, which can be shared openly, and a private key which is kept secret. This method enhances secure key distribution and allows for authentication through digital signatures. Asymmetric encryption is slower and better suited for small data exchanges or key exchanges.

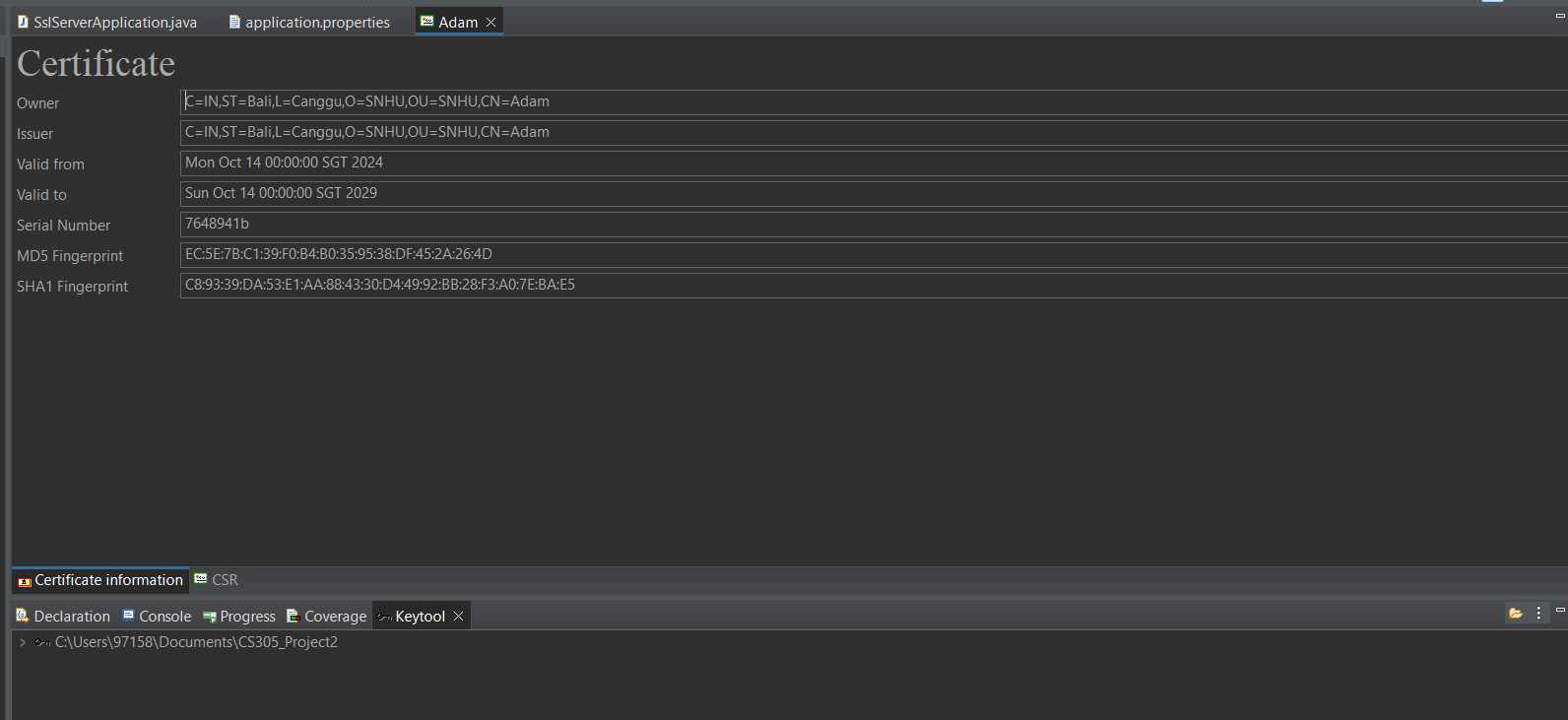
Brief history of algorithmic encryption

The history of encryption algorithms dates back thousands of years. A famous example is that of the Caesar cipher used by Julius Caesar in the 1st century BCE, where each letter was shifted by a fixed number of positions in the alphabet. As with all things, with time, the complexity of the algorithms evolved. An example of that evolving complexity is the Vigenère cipher, introduced in 1553, which used a keyword to apply multiple Caesar ciphers. Another milestone was the Enigma machine, used by Nazi Germany during World War II, which became infamous for a complex analogue rotor system that scrambled military communications. However, just as famously, it was cracked by Alan Turing and his team.

With the dawn of the digital age, the arms race between encryption and decryption continues. The development of the Data Encryption Standard in 1977 was one of the first milestones in modern key encryption standards. While it was a complex symmetric system, it was vulnerable to brute force attacks. By 2001, the Advanced Encryption Standard replaced the DES and has become the global standard, with enhanced security with key sizes of up to 256 bits (*Federal Information Processing Standards Publication, 2001)*. Looking forward, the rise of quantum computing poses significant and as-of-now unknown challenges to existing encryption methods, prompting research into post-quantum cryptographic algorithms.

## Certificate Generation

Insert a screenshot below of the CER file.





[Insert screenshots here.]

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screenshot of a computer

Description automatically generated

## Secure Communications

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

A screenshot of a computer

Description automatically generated

## Secondary Testing

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

A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

Figure 1: Dependency check before refactoring

A screenshot of a computer

Description automatically generated

Figure 2: Dependency check after refactoring

Refactoring introduced no new vulnerabilities

## Functional Testing

During this testing section I made some changes to the code to increase security.

Full code:   
 A screenshot of a computer

Description automatically generatedA screen shot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generated  
  
  
  
Changes made:

1. Removed hard coded data

A screenshot of a computer program

Description automatically generated

Moved data to the properties file so they can be injected into the application

A screen shot of text

Description automatically generated

Figure 3: used @value annotations to dynamically inject the data from application.properties

2. Added input validation

A computer screen shot of text

Description automatically generated

3. Improved error handling

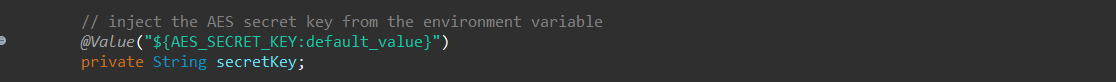
A screen shot of a computer

Description automatically generated

4. Created environment variable for AES key

A computer screen shot of a black screen

Description automatically generated



Application running without errors:

A screen shot of a computer

Description automatically generated

These changes will be expounded upon in the summery section.

## Summary

In refactoring the code to comply with software security measures, I adhered to the vulnerability assessment protocol and addressed each area of security that I deemed most critical to the applications security.

Input Validation:

In the refactored code, I implemented input validation for the name and the data fields. Initially, in the first draft of the code, the data fields were hardcoded into the application and there was no validation of these fields. However, after refactoring, the methods isValidName ensures that the data string only contains letters and spaces and is not unreasonably long. Method isValidData does a similar thing to the data field. Validating these fields ensires the integrity of the inputs and protects against potential tampering or accidental misconfigerations.   
  
Cryptography

The AES encryption addresses the need for securing the data, both at rest and in transit., with a method to encrypt the data with AES-256 encryption. Initially the encryption was hardcoded in the applicaition, but then was altered to be injected via an environment variable, reducing risk of potential exposure. Moreover, SHA-256 hashing was used to ensure data integrity with a checksum verification. Together, this provides a modern and robust encryption defense.

Client / Server

At first, the application ran with HTTP. With the creation of a CA certificate and altering the application.properties file, it now runs far more securely over HTTPS, increasing the integrity of the data exchanged between the server and clients, reducing the risk of man-in-the-middle attacks.

Code Error handing

**Error handling** was improved in the second refactoring by more appropriately catching and appropriately handling exceptions. This is important in order not to **could potentially leak sensitive implementation details to malicious users.**

Code Quality

The second refactoring followed secure, quality coding practices by ensuring that potentially sensitive data was injected rather than hardcoded into the source code, which could unnecessarily expose sensitive information. For the name and data fields, they were moved to the application.properties file and injected with @value parameters. For the AES key, a key was generated using command prompt and injected into the program as an environment variable.

Encapsulation

The above changes also helped to encapsulate the code, enforcing a separation of concerns that helps to prevent sensitive data being unnecessarily exposed. Furthermore, private methods were used where possible, such as the input validation method ands and checksum method, further encapsulating any sensitive business logic.

Process for adding layers of security to the software application

The process of adding layers of security to the software application began with a preliminary step of setting up the software environment and creating a CA certificate for later HTTPS using SSL configurations. The first real step focused on the implementation of AES-256 encryption for data confidentiality and SHA-256 hashing, as these cryptographic measures were the core of protecting both the integrity and confidentiality of the data transmitted and stored by the application. I had to ensure that the application could generate secure checksums using SHA-256, and then encrypt the data using AES-256. After the implementation was made, a dependency check was ran to ensure I had added no new vulnerabilities to be concerned about.

Once the encryption and hashing were in place, the next step was securing the communication channel. By utilizing the .cer certificate I had created earlier and configuring HTTPS using SSL, I ensured that the application could communicate securely over the web. Running the application on HTTPS added a crucial layer of transport security, protecting data in transit from potential interference.

In the subsequent refactoring phase, I focused on refining the code, enhancing code quality, input validation, and encapsulation. I removed hardcoded values such as the AES encryption key and data like the name and checksum string from the source code. These were externalized and injected via the application.properties file, significantly improving security by preventing sensitive data from being exposed in the code. Additionally, input validation was implemented to ensure that all inputs, such as the name and data, met strict validation rules to avoid malicious data input, improving the robustness of the application. Finally, these steps also enhanced encapsulation, ensuring that sensitive data remained controlled within secure structures.

By layering these security measures—cryptography, transport layer security, input validation, and secure coding practices—I attempted to follow security best practices at multiple levels.

## Industry Standard Best Practices

## Explain how you used industry standard best practices to maintain the software application’s existing security

To implement industry best practices, I attempted to utilize industry security standards when securing the software. Firstly. this philosophy educated my choice of encryption cipher, as AES-256, as previously noted, is the industry benchmark for strong encryption. AES-256 is highly resistant to cryptographic attacks, such as brute force, and is commonly recommended in compliance frameworks like NIST.

The use of **SHA-256 hashing** to generate checksums ensures data integrity, preventing tampering or unauthorized changes to data. Moreover, SHA-256 is widely regarded as a secure hashing algorithm used across industries to protect sensitive data, and thus was the reason I chose it implement to generate checksums to ensure the software’s data integrity.

Secondly, industry standard practises influenced how I implemented secure software communication: I configured the application to use HTTPS by generating and applying an SSL certificate, which encrypts communication between the client and server. HTTPS is a foundational security measure, ensuring that data in transit is protected from attacks, like man-in-the-middle (MITM) attacks. HTTPS considered an essential industry practice for any application dealing with sensitive information.

Thirdly, to prevent common web vulnerabilities, I applied input validation techniques to ensure that all incoming data follows expected formats. Input validation is absolutely critical and an essential industry standard. By validating user inputs, I attempted to protect the application for potentially malicious injection-based vulnerabilities.

Lastly, I followed best practices by removing hardcoded sensitive information, such as the AES encryption key and other static values, from the code and instead storing them in an external configuration file (application.properties) and via environment variables. This reduces the risk of accidental exposure and unauthorized access to sensitive data. Secure configuration management is a best practice in mitigating risks associated with code leaks.

Value of Applying Industry Standard Best Practices for Secure Coding

The most apparent and direct value of secure coding practices is protect the application from a wide range of vulnerabilities that could be exploited by attackers. By adhering to these practices, the company reduces the risk of data breaches, unauthorized access, and service disruptions, ensuring the safety of its data and that of its users. Perhaps less apparent but no less critical to a companies functionality in a real world market is compliance and regulation coherence with industry bodies. Following industry standards helps the company comply with regulatory requirements (such as GDPR, HIPAA, or PCI-DSS) that mandate the protection of customer data. Non-compliance can result in heavy fines, reputational damage, and legal liabilities, but by following best practices, the company mitigates these risks. Moreover**,** adhering to secure coding best practices demonstrates the company’s commitment to data protection and security, which is critical for building and maintaining trust with customers, partners, and stakeholders: a secure application reassures users that their data is handled responsibly and safely. Adhering to industry standards is also crucial for Long-Term Security and Scalability: by applying industry standards, the company ensures that its software can withstand evolving security threats. These practices lay the foundation for ongoing security, making it easier to scale securely as the application grows and develops new features.

Overall, following industry-standard best practices is essential to ensuring that the application is secure, maintainable, and compliant with industry regulations, which ultimately protects the company's reputation and financial stability.

References:

Announcing the ADVANCED ENCRYPTION STANDARD (AES) (PDF). (2001, November 26). Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). Archived (PDF) from the original on August 23, 2024. Retrieved August 26, 2024.

Shah, J. (2023, April 13). What is AES encryption? The Complete Guide. 1Kosmos. <https://www.1kosmos.com/authentication/aes-encryption/>

Rais Rabtsani, M., Triayudi, A., & Soepriyono, G. (2024). Combination of AES (advanced encryption standard) and SHA256 algorithms for data security in bill payment applications. *SAGA: Journal of Technology and Information System*, *2*(1), 175–189. https://doi.org/10.58905/saga.v2i1.250

Mario Stipcevic. (2012). Quantum random number generators and their applications in cryptography. Proc. SPIE 8375, Advanced Photon Counting Techniques VI, 837504. https://doi.org/10.1117/12.919920