****

# Practices for Secure Software Report

Table of Contents

[Document Revision History 3](#_Toc102040754)

[Client 3](#_Toc102040755)

[Instructions 3](#_Toc102040756)

[Developer 3](#_Toc102040757)

[1. Algorithm Cipher 4](#_Toc102040758)

[2. Certificate Generation 6](#_Toc102040759)

[3. Deploy Cipher 8](#_Toc102040760)

[4. Secure Communications 8](#_Toc102040761)

[5. Secondary Testing 9](#_Toc102040762)

[6. Functional Testing 10](#_Toc102040763)

[7. Summary 12](#_Toc102040764)

[8. Industry Standard Best Practices 13](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **12/14/2024** | **Sean Jetté** | **Second Release** |

## 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

Sean Jetté

## Algorithm Cipher

## ****High-Level Overview****

## The recommended form of encryption algorithm cipher for Artemis Financial is the Advanced Encryption Standard (AES), which is “a highly trusted encryption algorithm used to secure data by converting it into an unreadable format without the proper key” (GeeksforGeeks, 2024b). Developed by the National Institute of Science and Technology (NIST), the AES algorithm is widely regarded as the gold standard used for securing data due to the nature of its design, widespread usage, and its efficiency concerning securing data at rest (Crawford, 2019). Additionally, this symmetric algorithm cipher operates on fixed block sizes, which consist of 128 bits, and supports a few different key sizes that range from 128, 192, or 256 bits, respectively. Its implementation of symmetric encryption standards enables this algorithm cipher to secure data efficiently and is widely used in securing internet communications, protecting sensitive data, and encrypting files; making it a cornerstone of modern cryptography ((GeeksforGeeks, 2024). Moreover, within AES, there are various modes that can be used for block ciphers to encrypt arbitrary amounts of data while also implementing different security and performance characteristics (OWASP, 2024). An option to consider is the implementation of the Galois/Counter Mode (GCM). This could prove useful for its data confidentiality/integrity, high performance, and its widely supported usage of cryptographic libraries (Pake, 2024)—a key factor for securing financial transactions, customer data, and other sensitive information.

#### **Hash Functions and Bit Levels**

AES-128 is an encryption algorithm cipher and not considered a hash function. An example of a hash function would be the Secure Hashing Algorithm or SHA, which is a cryptographic algorithm used to generate a hash from input data (GeeksforGeeks, 2024a). Often, these can be used to verify data integrity but not for encryption. Bit levels for AES in this instance is 128, whereby the 128-bit key is calculated as 2^(128) possible key combinations and as such, it is known to be virtually secure against brute-force attacks; even at today’s computational capabilities (Beal, 2024). Moreover, within the AES standard, there are additional bit sizes at 192 and 256, which offer stronger security for potentially higher-risk applications. In consideration of usage with AES, hashing functions like SHA-256 may be used in conjunction for key derivation, authentication, and additional data integrity requirements (Manico & Detlefsen, 2015).

#### **Symmetric vs. Asymmetric Keys**

AES is a form of symmetric encryption whereby the same key encrypts or scrambles the data and in order to decrypt the information, you use the same key. The main function for this is that the key must remain secret for the integrity of the system (Manico & Detlefsen, 2015). While this might not be as secure of asymmetric encryption standards (Adetunji, 2024), this methodology is more efficient for large datasets. This makes it ideal for scenarios requiring fast and secure processing, such as encrypting bulk financial data or securing communications. Conversely, asymmetric cryptography, which is essential to signing, forms of encryption utilize two keys where one, a public key is known to everyone, and a private key that is only known to the sender (Manico & Detlefsen, 2015). In some instances, the public key can be used for encryption but must be decrypted only by the private key.

#### **Random Numbers**

In relation to the generation of random numbers, this step in the encryption process is critical for the overall algorithm’s security process through the use of key generation, initialization vectors in encryption modes like AES-CBC or AES-GCM, and salts in hashing. Specifically, the essence of random numbers helps to generate two factors: unpredictability and entropy, which is the representation of uncertainty or disorder in the system (Parker, 2023). As a key aspect in cryptography, key generation is applied to both symmetric and asymmetric algorithms. By ensuring the keys are not predictable, the algorithm helps to ensure confidentiality of the data (Parker, 2023). According to Adcyber (2023), the initialization vectors (IVs) used in encryption specific modes are there to ensure the plaintext does not product identical ciphertexts through a random or predetermined sequence of bytes that is added to the encryption process—increasing the overall security of any ciphered message. Random salts within hashing add additional random values to input data before hashing to prevent hash values that are identical (Parker, 2023). While attackers are learning more sophisticated methodologies to crack passwords through usage of programs like Hashcat, this additional measure in the hashing process helps to protect against precomputed attacks like rainbow tables, which is “a large database of hash value pairs linked to their plaintext counterparts” (Wolford, 2024). An example of Java’s functionality in random number generation is the *SecureRandom* class, which provides a reliable tool for generating secure cryptographic random numbers. Unlike standard random number generators, *SecureRandom* is specifically designed to ensure high-quality randomness (Manico & Detlefsen, 2015).

**History and Current State**

Algorithm ciphers have a long history dating back to ancient times through the seeding of secret messages. Dating back thousands of years, we can see evidence of encryption usage from 1900 BCE in Egypt, to enciphered writings with Mesopotamia, to the legendary Caesar Cipher where each letter in plaintext is replaced by a set of revolving number of letters either going forward or backward within the Latin alphabet (Schneider, 2024). While there has been countless eras where cryptography has been used, such as in WWI and WWII, where the father of modern computing, Alan Turing, the inventor of the Turing machine, developed a sophisticated machine that was used to crack the German Enigma cryptosystem (Schneider, 2024), our modern usage of data encryption standards stems from the early 1970s where the formerly known National Bureau of Standards (NBS), now known as NIST, initiated a program to develop the Data Encryption Standard (DES) to “protect computer data and to allow for large-scale commercial interoperability” (Chen & Scholl, 2022). As such, this newly developed encryption standard, DES, was the first public encryption for use created by the U.S. government (Chen & Scholl, 2022).

In the current state of cryptography, we have methodologies in use like the Ron Rivest, Adi Shamir and Leonard Adleman (RSA) public key cryptosystems—based on one of the oldest methods to encryption that uses prime number generation for its keys and thus, this asymmetric methodology is prohibitively difficult to factor without the known private and public keys; making it a must-have in our modern cryptographic landscape (Schneider, 2024). Within the world of AES, this type of symmetric algorithm cipher replaced older algorithms like DES and 3DES due to vulnerabilities and shorter key lengths. It is now the gold standard for encryption, endorsed by the U.S. National Institute of Standards and Technology (NIST) *(Crawford, 2019)*. Additional factors within the encryption standards that are highly necessary are hash functions, such as the Secure Hash Algorithm (SHA), ensuring a high degree of data integrity for digital signatures, blockchain, and password hashing. Moreover, modern encryption supports technologies like TLS/SSL for secure internet communications and is essential to Java web applications (Manico & Detlefsen, 2015).

While these forms of encryption prove ultimately useful, our modern computing landscape is on the brink of the adaptation toward quantum computing breakthroughs that might prove useful for nation states or advanced persistent threat (ATP) actors, such as any adversarial state-sponsored group with access to significant resources. As such, these quantum computing technologies, or post-quantum cryptography, can also prove to be useful where it can have “the potential to be far more secure than previous types of cryptographic algorithms, and, theoretically, even unhackable” (Schneider, 2024).

## Certificate Generation

Insert a screenshot below of the CER file.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Secure Communications

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

A screen shot of a computer

Description automatically generated

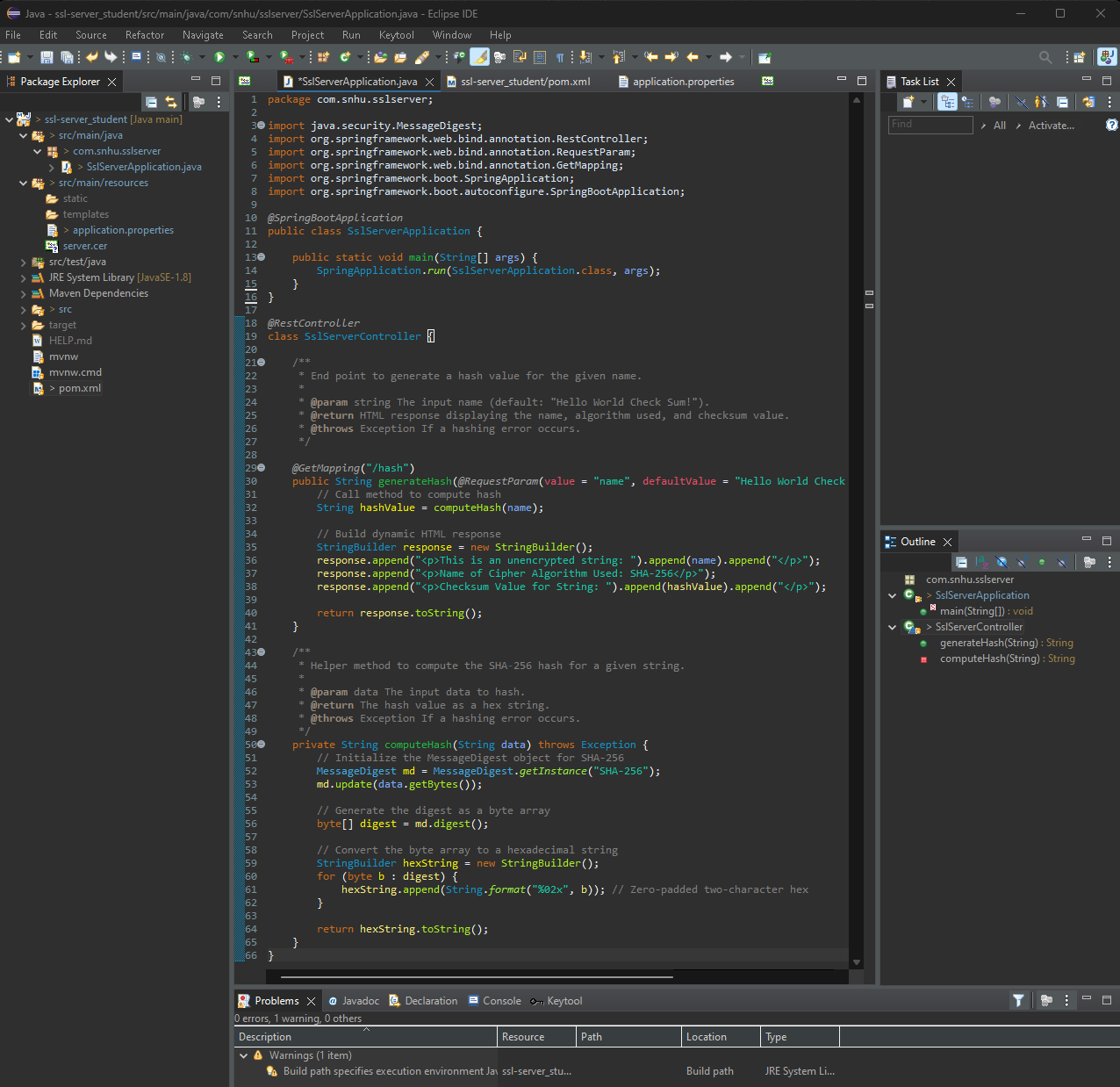
This screenshot shows the enabling of SSL (so HTTPS will be used) and configures the keystore for the SSL certificate

A screenshot of a computer screen

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

Description automatically generated

## Functional Testing

## Starting with syntactical vulnerabilities, the analysis of the software yields is syntactically correct while compiling without Any errors. The program does yield one warning concerning the use Java SE 1.8; an older version. Moreover, while the program outputs this warning, the project correctly compiles and runs within the given environment set up. There are repositories for downloading and configuring this older version of Java SE that could be utilized but the more important aspect would be to update the program to the latest version of Java SE for improved security features, implementation of new features, better performance gains, and long-term support of the Java web framework. Moreover, working with older versions of Java might have an effect on third-party libraries used throughout the development of the program and would require all third-party libraries to also be updated.

## With respect to logical vulnerabilities, the *generateHash* method builds an HTML response dynamically with string concatenation. This method, while technically working, might prove to be difficult for maintenance challenges. Additionally, the method is implemented with throwing a generic *Exception* that can ultimately obscure errors causes. It would be beneficial for the development team to implement specific exception handling to improve clarity of the code and for debugging purposes.

## Throughout the codebase, there are some potential security vulnerabilities that were identified. Starting with inputs: an input, the *name* parameter from the *@RequestParam* annotation, is directly used without sanitation. Due to this unsanitized input, there is the potential to expose the application to Cross-Site Scripting (XSS) attacks. This can be true if the out-putted HTML is executed within a web browser. To mitigate this, encoding or sanitizing the input to prevent malicious scripts from being executed would be vital. Implementation of the SHA-256 hash algorithm is basically the standard in this instance and while there has been an update to SHA3-256, collision resistance for this hashing algorithm is considered secure and collision-resistant. The code does not interact with any databases, and thus is not a risk for SQL injection attacks but could be if and when databases are implemented. Lastly, because the code relies on managed memory within the JVM, the code would not exhibit any buffer overflow risks.

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

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Summary

To begin, the code was refactored to improve its security posture by addressing vulnerabilities identified throughout the testing process. The initial step in the refactoring involved implementing input validation to ensure inputs were sanitized and validated effectively, mitigating the risk of injection attacks such as cross-site scripting (XSS) or remote code execution (RCE). This was achieved through the use of the Apache Commons Text library, which eliminated vulnerabilities in string interpolation mechanisms while also ensuring proper escaping of any untrusted data for the prevention of executing any malicious code. Next, API endpoints were also thoroughly reviewed and secured to enforce authentication requirements that reduced apparent risks for aspects like unauthorized access and data leaks.

Cryptographic practices were improved through the enforcing of HTTPS with TLS; ensuring secure client-server communication are met. The proper configuration of SSL/TLS certificates using the *Keytool* utility ensures that the data transmitted between our clients and the server will remain encrypted; preventing threats such as a man-in-the-middle (MITM) attack. Additionally, improving exception handling mechanisms for the prevention of sensitive data from being exposed whilst still displaying user-friendly messages to end users should prove useful. Ultimately, this approach helps to minimize the associated risk of adversaries gaining any insight into the application's internal processes while still preserving critical debugging information for backend development. The updates also provide the necessary adherence for secure coding standards and practices; reinforced by the use of the Maven dependency-check report for the identifying of CVEs, evaluating false positives or negatives, and to track overall dependency health throughout the development process.

The process of adding layers of security to the application started with input sanitization using the *StringEscapeUtils.escapeHtml4()* method to handle the *name* parameter; ensuring that malicious inputs like XSS payloads would not execute. The selection of the Apache Commons Text library supports the necessary requirements by addressing known interpolation vulnerabilities within the codebase. An additional security enhancement involved is the usage of cryptographic hashing practices—particularly the use of SHA-256, a collision-resistant algorithm that meets modern cryptographic standards. This was implemented via the *MessageDigest* API, which generates checksums. Moreover, exception handling in the *generateHash* method was designed to avoid the exposing of sensitive internal details to end users, whereby displaying only generic error messages and retaining detailed logs for any internal review and debugging purposes.

Through the refactoring process, the code effectively addressed areas of vulnerability, supporting the steps outlined in the vulnerability assessment process flow diagram. By addressing these areas systematically, the application meets modern security standards, mitigates identified risks, and provides a solid foundation for ongoing maintenance and security improvements.

## Industry Standard Best Practices

As a reiteration of previous practices mentioned above throughout the report, the maintaining of security requirements throughout the DevSecOps of the code base is paramount to an effective and secure program. For security requirements, implementing input validation and sanitation of the codebase to prevent attacks like XXS or RCE plays a huge role; especially if inputs that pose such a risk aren’t identified through the manual code review. For this project in particular and other web-based programs, ensuring the use of secure hashing practices, such as SHA-256—a collision-resistant hashing algorithm, via the *MessageDigest* API for the generation of checksums for data protection and integrity. This also leads to the implementation of exception handling to prevent any sensitive data from being exposed. This is especially important for debugging purposes but also providing end-users with vanilla messaging and not giving away important security aspects of the program’s code for exploitation purposes. For the implementation of secure communications, I enforced HTTPS and configured an SSL/TLS certificate for encrypted client-side communications to reduce any MITM attack vectors and thus, leading to potential data breaches. Lastly and arguably most important is leveraging open-source dependency management through the use of the Maven dependency-check within the Eclipse IDE. Maintaining up-to-date information on submitted and documented CVEs plays a vital role in preparing your program to work without known exploitation getting in the way. Using this plugin to regularly identify, address, and update third-party dependencies is vital to the security of the organization’s web application framework.

As a contractor for programming Artemis Financials’ web application, it’s important to not only exercise security and coding best-practices but ensure that we build trust and a good reputation for the organization as a whole. Applying the necessary industry standards throughout DevSecOps demonstrates commitment for protecting the clients, stakeholders, and the customers who utilize the program. Moreover, such secure coding practices and proactive vulnerability checks help to minimize the likelihood of incidents that might tarnish the organization’s reputation. A focus on regulatory compliance is also necessary due to potential legal or regulatory consequences if adherence to such practices is not adequately followed. An additional factor to consider throughout the development process that applies to the overall well-being of the company is long-term efficiencies. Mitigating any identified vulnerabilities throughout the development process can ultimately prevent additional costs towards post-release patches, security incidents, and any sort of damage control requirements if known vulnerabilities are not adequately addressed and mitigated. Lastly, the adoption of such secure coding standards helps to create a scalable foundation for any sort of future development through the software lifecycle.

References:

Adcyber. (2023, June 25). What is the initialization vector in AES encryption? - Cyber Insight. *Cyber Insight*. https://cyberinsight.co/what-is-the-initialization-vector-in-aes/

Adetunji, D. (2024, September 10). *Symmetric and asymmetric key encryption – explained in plain English*. freeCodeCamp.org. https://www.freecodecamp.org/news/encryption-explained-in-plain-english/

Beal, V. (2024, June 17). *128-Bit Encryption*. Techopedia. https://www.techopedia.com/definition/29708/128-bit-encryption

Chen, L., & Scholl, M. (2022, May 26). *The Cornerstone of Cybersecurity – Cryptographic Standards and a 50-Year evolution*. NIST. https://www.nist.gov/blogs/cybersecurity-insights/cornerstone-cybersecurity-cryptographic-standards-and-50-year-evolution

Crawford, D. (2019, February 4). *How does AES encryption work?* ProPrivacy.com. https://proprivacy.com/guides/aes-encryption

GeeksforGeeks. (2024a, June 14). *How Does a Secure Hash Algorithm work in Cryptography?* GeeksforGeeks. https://www.geeksforgeeks.org/how-does-a-secure-hash-algorithm-work-in-cryptography/

GeeksforGeeks. (2024b, July 16). *Advanced Encryption Standard (AES)*. GeeksforGeeks. https://www.geeksforgeeks.org/advanced-encryption-standard-aes/

Manico, J., & Detlefsen, A. (2015). *Iron-Clad Java: Building secure web Applications (Oracle Press)* (Version 1.0) [Webpage]. McGraw-Hill Education. https://learning.oreilly.com/library/view/iron-clad-java/9780071835886/?sso\_link=yes&sso\_link\_from=SNHU

OWASP. (2024). *Cryptographic Storage - OWASP Cheat Sheet Series*. https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic\_Storage\_Cheat\_Sheet.html

Pake, J. (2024). *Galois Counter Mode (GCM) - Complex Security*. Complex Security. https://knowledge.complexsecurity.io/cryptography/gcm/

Parker, J. W. S. (2023, December 30). *What is the importance of random number generation in cryptography?* James Parker. https://www.jamesparker.dev/what-is-the-importance-of-random-number-generation-in-cryptography/

Schneider, J. (2024, August 30). A brief history of cryptography: Sending secret messages throughout time. *IBM Think*. https://www.ibm.com/think/topics/cryptography-history

Wolford, B. (2024, May 31). *What is a rainbow table attack and how to prevent it?* Proton. https://proton.me/blog/what-is-rainbow-table-attack