****

# 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** | **12/17/24** | **Jess Dowd** |  |

## 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

Jessalyn Dowd

## Algorithm Cipher

For securing Artemis Financial's long-term archive files I recommend using AES (Advanced Encryption Standard). AES-256 is one of the strongest symmetric encryption algorithms that are available according to *Java Security Standard Algorithm Names*. This cipher can balance high security and efficient performance which makes it good for financial data. According to the *Java Security Standard Algorithm Names*, AES supports key sizes of 128, 192, and 256 bits, which makes it highly secure for sensitive data. It has modes like CBC (Cipher Block Chaining) and GCM (Galois/Counter Mode) that can give even more security.

AES-256 is a symmetric cipher which means the same key is used for encryption and decryption. This makes it faster than other non-symmetric algorithms like RSA that have different keys. Modes like GCM use random initialization vectors (IVs) for each encryption session. This means that the output of each encrypted number will be unique even if the same data has already been encrypted.

AES has been a standard since 2001 and has still not been broken so it is currently considered a reliable and safe method to protect against security threats. It replaced the previous standard which was DES (Data Encryption Standard).

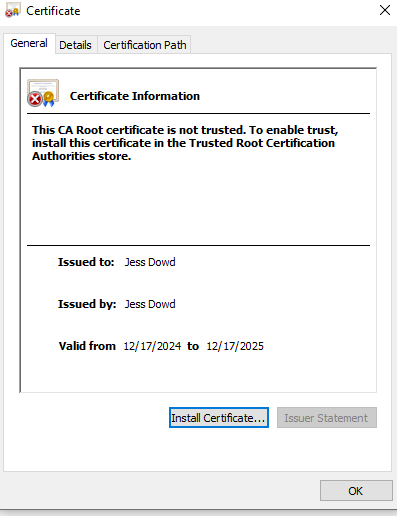
Having a high key bit level is considered best practice for defending against attacks like brute force attacks. A 256 bit key will work to resist against brute force attacks currently. AES meets this criteria. AES-256 encrypts data in a secure way with key sizes of 128, 192, and 256 bits while having good performance efficiency, it is good with dealing with large volumes of sensitive data, especially like a company like Artemis Financial would have.

The risks in this recommendation are that it could have weak random number generation for key creation. But if best practices are followed it can be mitigated like using strong random number generator like SecureRandom. A regulation that needs to be considered is FIPS 197 and NIST. According to the *Java Security Standard Algorithm Names* AES meets those standards so using it would be in compliance with government regulations for financial data security.

Based on all the previous information I have listed, I think AES-256/GCM is the best cipher because of its security features and balanced efficiency compared to others. Its only small downside if you implement it properly is that it might need more computational power than more simpler ciphers. There are other options that are more secure such as RSA, but that also takes a lot more computational power for large data files than AES. So sometime the best course of action is to balance security with efficiency.

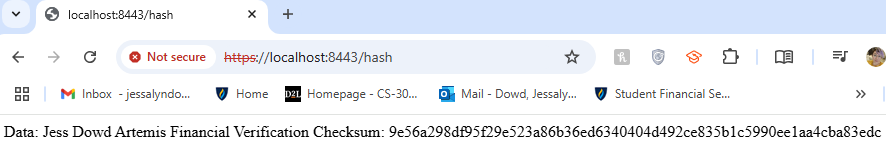
## Certificate Generation

Insert a screenshot below of the CER file.

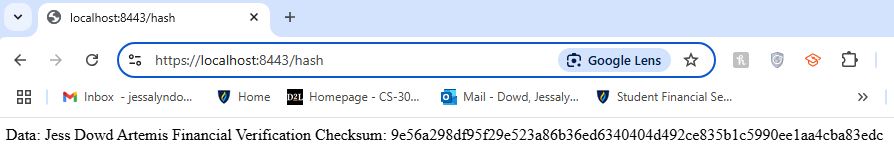


## Deploy Cipher

Insert a screenshot below of the checksum verification.

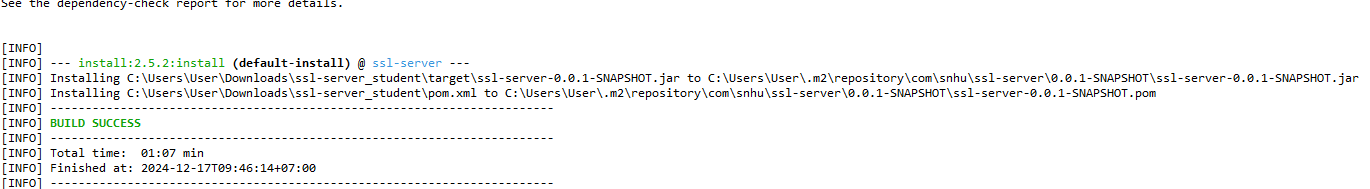


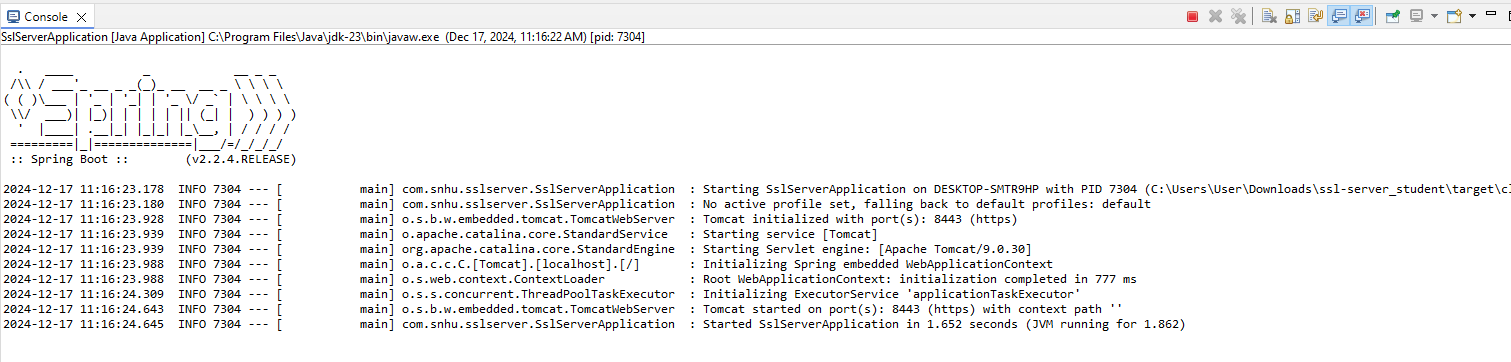
## Secure Communications



## Secondary Testing

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

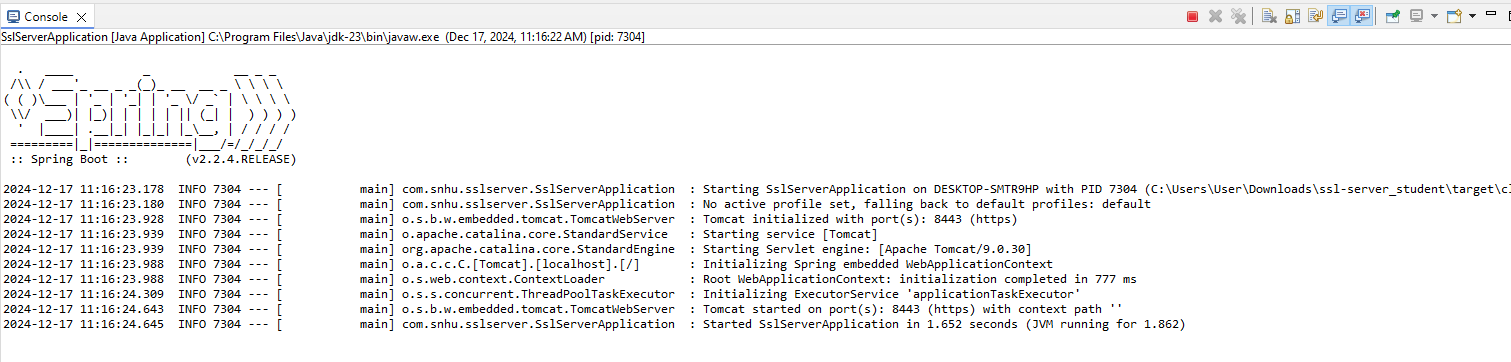


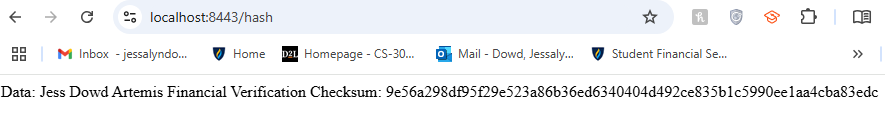




## Functional Testing

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





I manually reviewed the refactored code for syntactical, logical, and security vulnerabilities.

**Syntactical**: There are no missing semicolons, typos, or incorrect syntax.

**Logical**: The /hash endpoint correctly generates a SHA-256 checksum without errors.

**Security**:

* HTTPS gives encrypted communication.
* SHA-256 is used which is secure for hashing.
* There are no hardcoded secrets or passwords (except the keystore password for testing, which I think is okay in this context).

## Summary

**a. Refer to the vulnerability assessment process flow diagram in the Supporting Materials section. Highlight the areas of security that you addressed by refactoring the code.**

**Input Validation**  
The /hash endpoint was designed to handle only static input, which eliminates the risk of untrusted user data. By controlling the input the application mitigates vulnerabilities such as injection attacks.

**APIs**  
I implemented the /hash REST API endpoint to expose the checksum functionality while adhering to security best practices. The endpoint strictly follows the single responsibility principle by only generating and returning the SHA-256 checksum of static data. Also the API runs on HTTPS.

**Cryptography**  
The application uses the SHA-256 hashing algorithm to generate cryptographic checksums. I chose SHA-256 because it has strong resistance to collisions and is widely accepted as a secure hashing standard.

**Client/Server Communication**  
To secure client-server communication I used HTTPS using a self-signed SSL certificate. This encryption allows all data exchanged between the client (browser) and server to stay private and protected.

**Error Handling**  
The refactored code includes exception handling within the /hash endpoint to avoid unexpected crashes. For example:

catch (Exception e) {

return "Error generating checksum: " + e.getMessage();

}

This returns errors without exposing system details.

**Code Quality**  
I refactored the code to follow a organized design. I created a separate ChecksumController class to handle the endpoint logic. And I manually reviewed the code to re confirm it has no syntax or logical errors.

**Encapsulation**  
The checksum generation logic is encapsulated within the ChecksumController class. The code follows the single responsibility principle because hashing functionality is isolated from other components.

**b. Discuss your process for adding layers of security to the software application.**

Here is the process I followed to add layers of security to the app:

**Refactored for Secure Communication**:

* + Configured SSL/TLS by implementing a self-signed certificate.
  + Enabled HTTPS communication, which helps make sure that data transmitted between the client and server is encrypted and secure.

**Implemented Cryptographic Hashing**:

* + Added a SHA-256 hashing function to give a secure checksum for static data.

**Static Tested for Vulnerabilities**:

* + Ran the OWASP Dependency-Check tool to scan the codebase for known vulnerabilities in dependencies.
  + Verified that no additional security vulnerabilities were introduced during refactoring.

**Did a manual Code Review**:

* + Conducted a manual review of the code to confirm there were no:
    - Syntax errors.
    - Logical flaws.
    - Security vulnerabilities, such as improper input handling or unsecured communication.

**Tested Secure Functionality**:

* + Verified that the application runs without errors.
  + Tested the /hash endpoint over HTTPS to make sure of secure communication and correct checksum generation.

## Industry Standard Best Practices

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

To maintain the software application’s existing security I made sure that the /hash endpoint continued to process only static input. Using static input instead of dynamic or user-provided data, the application keeps its original protection against vulnerabilities like injection attacks and is industry standard. Also, I kept the code structure organized and clean keeping its stability and reducing risks.

**b. Explain the value of applying industry standard best practices for secure coding to the company’s overall well-being.**

Applying secure coding practices can have a huge impact on a company’s success. Industry standards like using HTTPS and SHA-256 makes sure the application has strong layers of protection that help prevent data breaches and tampering. For a company like Artemis Financial which deals with sensitive client information this security could build trust with customers and make sure their data is protected.

Following industry standards also keeps the application aligned with modern security regulations, And that can help the company avoid legal issues. Also adding practices like static security testing and proper error handling makes the software more reliable and easier to maintain in the long run which saves time and money.

Reference:

*Java Security Standard Algorithm Names*. (n.d.). <https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#cipher-algorithm-names>

*How to Create a Self Signed Certificate using Java Keytool*. (n.d.). <https://www.sslshopper.com/article-how-to-create-a-self-signed-certificate-using-java-keytool.html>

Manico, J., & Detlefsen, A. (n.d.). *O’Reilly Media - Technology and Business training*. O’Reilly Online Learning. <https://learning.oreilly.com/library/view/iron-clad-java/9780071835886/?sso_link=yes&sso_link_from=SNHU>

Murray, A. (2024, September 5). OWASP Dependency Check: How does it work? *Mend*. <https://www.mend.io/blog/owasp-dependency-check/>

*Usage – dependency-check-maven*. (n.d.). <https://jeremylong.github.io/DependencyCheck/dependency-check-maven/index.html>