****

# 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** | **19 OCT 2025** | **Ryan Ward** |  |

## 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

Ryan Ward

## Algorithm Cipher

I recommend using the SHA-256 hashing algorithm for Artemis Financial’s file verification. The SHA-256 cipher scheme provides a secure method for generating a checksum hash to verify transmitted data, and it is one of the most used hashing algorithms. When creating a hash of a block of data, a new, ideally unique hash is formed based on that data. The hash is generated by taking the data and performing a series of operations on it that produces a unique value with a fixed length. This hash can be used to check that the original data was unchanged and thus successfully transmitted by performing the same hashing function on the data before and after transmission and comparing the hashes. Therefore, the generated hash must be a unique value that cannot be reproduced unless using the exact same input and similarly cannot be reversed. It should be impossible for the original data to be reproduced from the hash. If another data set were used to create an identical hash, that would mean a collision occurred, which would make it possible for one data set to be exchanged with another. In this case, a check of the hashes would incorrectly indicate the data sets were the same, even though they are actually distinct.

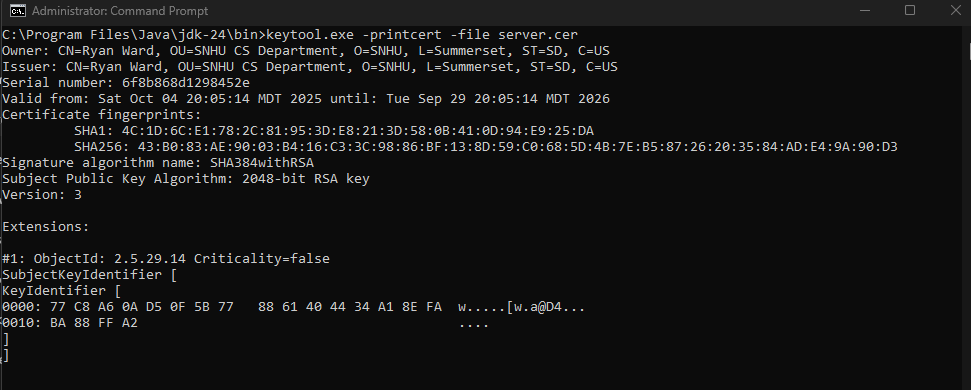
Hashing and bit levels are important components of any good cipher. Hashing, as described above, is the process of taking a piece of information, such as a password, and encoding it via various transformations. This can include shifting values, adding or removing values, and sorting the information. The intent of hashing is to generate an unreadable, but standardized, form of the information that can only be obtained via the proper encryption method. This makes the reversal of the encryption impossible. The only way to determine if the hashed data is correct is to perform the same encryption methods against it and compare the hashed value to a stored hash value. The number of bits used to encrypt data is also important, as an increase in the length of a key increases the time it takes to crack a cipher, particularly via a brute force attack. However, this also increases the time it takes to process a key when encrypting and decrypting data.

Random numbers are used quite heavily in encryption methods, as they provide a means of generating secure keys that are nearly impossible to recreate. A system will use various inputs to seed a random number generator. This generator can then produce a key that is unique and because of the random nature of the inputs used to create the random number, there is very little chance the same key would be generated twice. When generating such keys, there are two options for using them in ciphers: symmetric and asymmetric. Symmetric keys are created for use when the encryption and decryption functions are controlled by a single entity. This allows a single key to be used for both processes, simplifying the building of an application. On the other hand, there are asymmetric keys which can be used when dealing with scenarios where the sender of the data is unknown or there is a risk of a third party interfering with data transmission. The nature of asymmetric keys allows for a public key to be shared and used to encrypt data, but the data can then only be decrypted using a private key that is kept secret. This scheme allows anyone with the public key to send sensitive information to a recipient without the fear of the data being decrypted by a third party. Asymmetric keys work based on the factorization of prime numbers. Two very large prime numbers can be multiplied together to generate a new number that is very difficult to factor back to the original prime numbers. This new number is then used to generate both the public and private keys, such that they are distinct but mathematically related.

SHA-256 was developed by the NSA as a successor to the less secure SHA-1 and published in 2001. The NSA released a royalty-free patent for SHA-256 in 2004, making it widely available for public use. It is part of the SHA-2 family of hash functions and creates a fixed, 256-bit output from an input of any size. Unlike other hashing functions, SHA-256 protects against certain known vulnerabilities that plagued algorithms such as SHA-1 and MD5. This makes it ideal to use for checking the validity and integrity of data, such as a financial document.

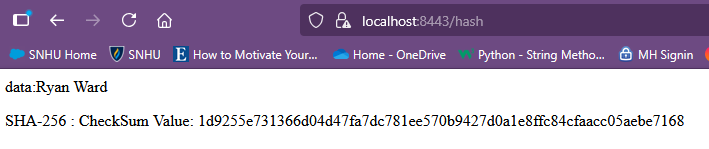
## Certificate Generation

Insert a screenshot below of the CER file.



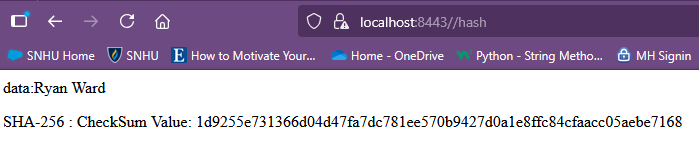
## Deploy Cipher

Insert a screenshot below of the checksum verification.



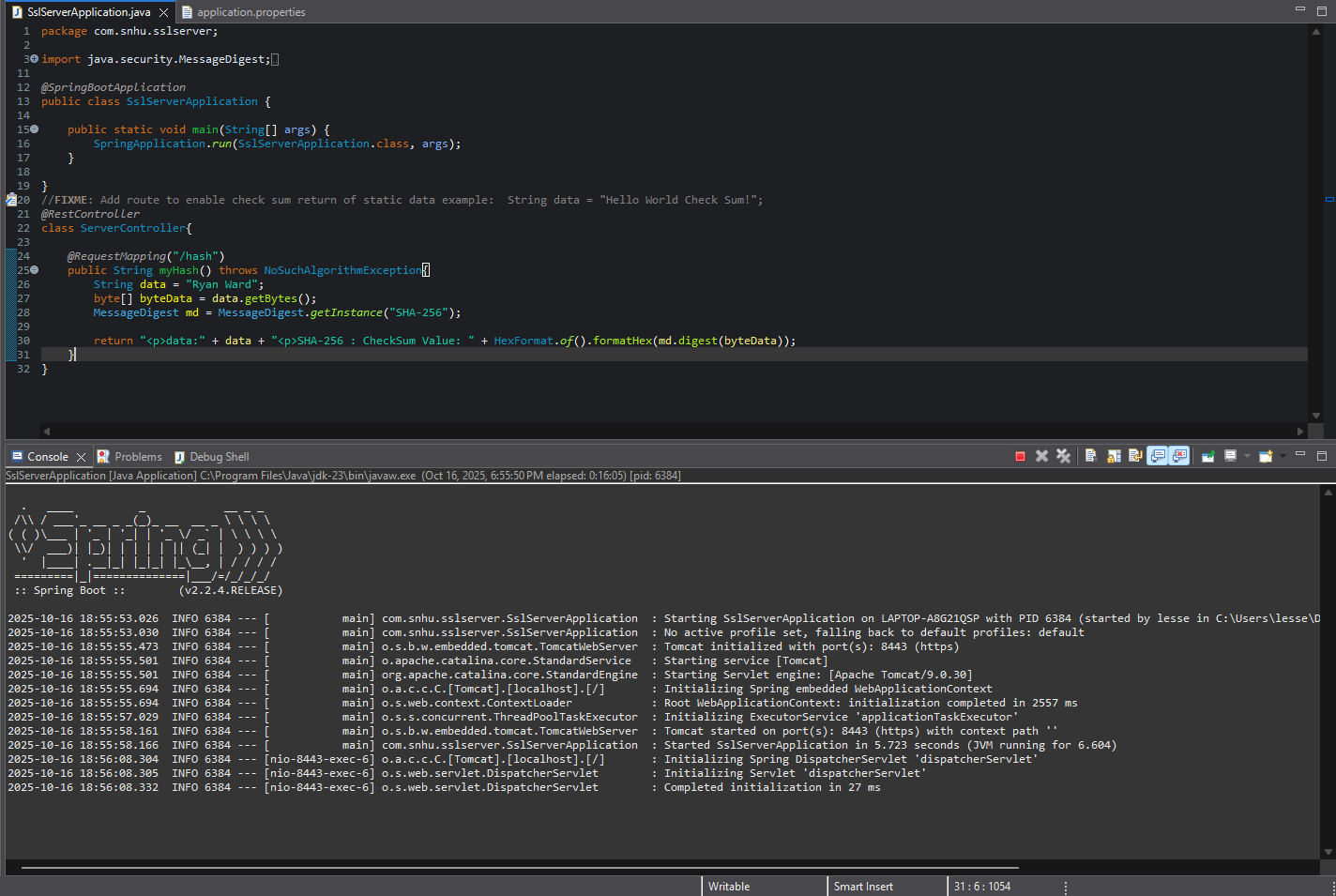
## Secure Communications

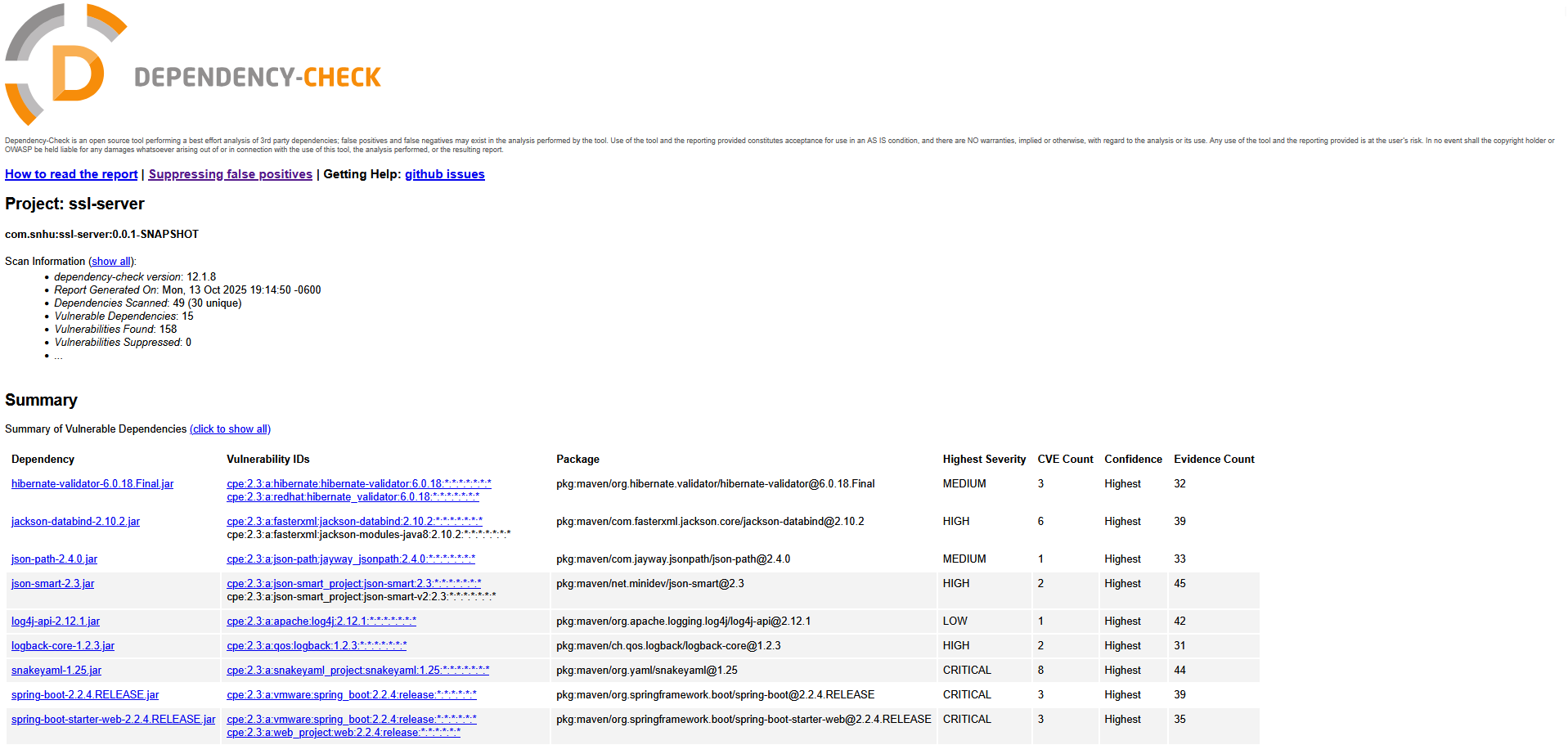
Insert a screenshot below of the web browser that shows a secure webpage.



## Secondary Testing

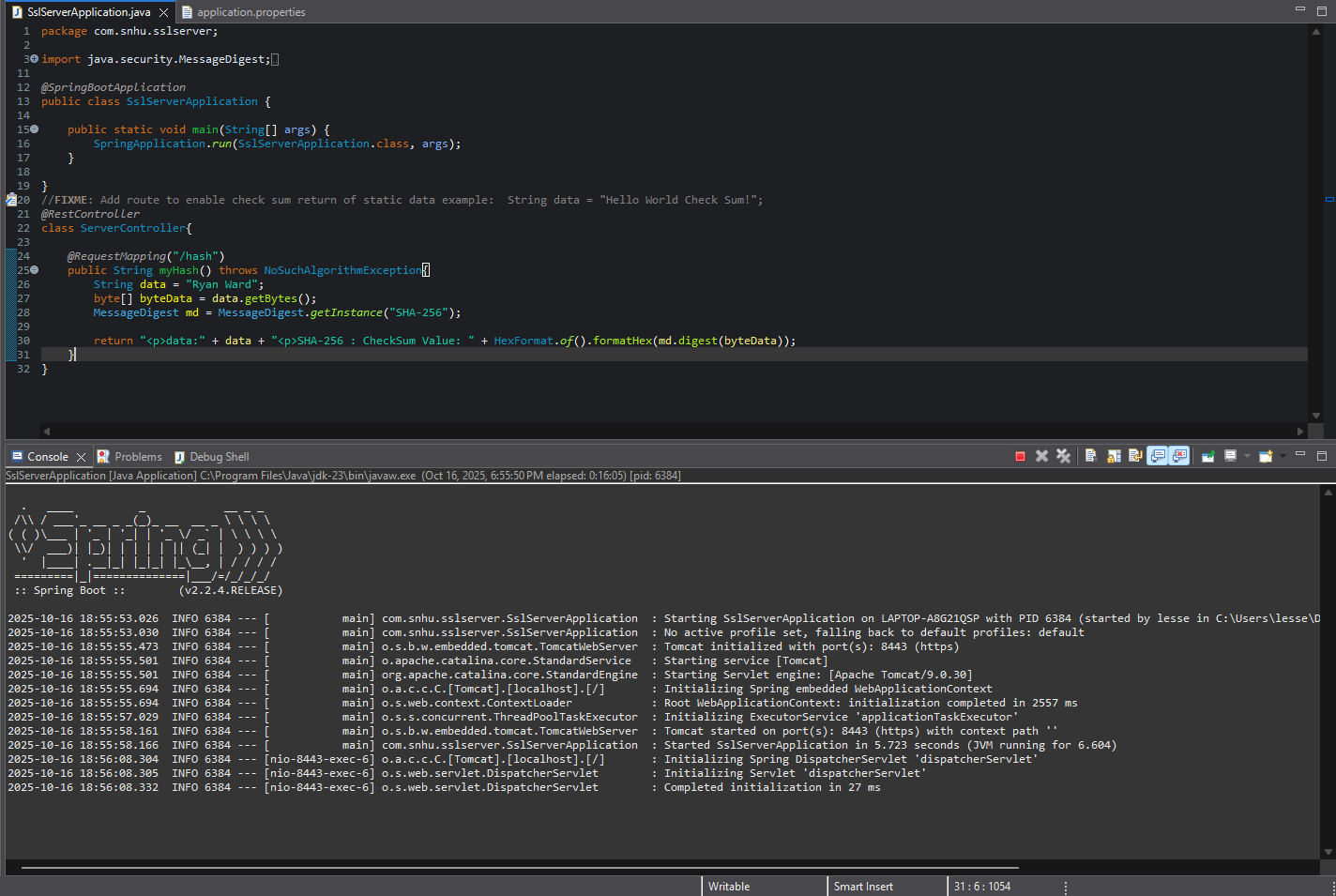
Insert screenshots below of the refactored code executed without errors and the dependency-check report.





## Functional Testing

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



## Summary

The following summarizes the vulnerability assessment process flow diagram regarding the various areas of security addressed by refactoring the code. To start, we’ll look at input validation. Validation of input is not a concern in this particular project, as the input used for hashing is hard-coded. No user input is needed and therefore input validation is skipped. Next, there’s secure APIs. Again, this is not an area of concern for this project, as no APIs are used. Moving on, we have cryptography. This is the main purpose of this code. We encrypt a string of data using and produce a hash value from it. For this application, I chose to use the SHA-256 hashing algorithm. Client/server issues are addressed by ensuring the program uses HTTPS communications. The code also addresses secure error handling by having the ability to throw an exception if an unknown hashing algorithm is used. This would prevent the code from running entirely. Secure coding practices are used throughout the program and address the concerns noted above. Additionally, static testing was conducted to ensure there are no dependency vulnerabilities that need to be looked at. Lastly, we have secure data structures. The only data structures used are a string to house the data to be hashed, and the hashed data itself. As we do not create any new data structures, there is no concern about the security of the data structures used. Also, there is no input to these data structures outside of the hard coded values, which alleviated the need for input validation and other secure coding concerns.

To add layers of security to the software, I first looked at creating the hash for the string data. By creating a message digest, I was able to use the SHA-256 algorithm to generate a hash that could be used as a verification tool to ensure the integrity and validity of the data. Next, I set about creating a secure communication environment, such that the server used HTTPS communication instead of the less secure HTTP. This was accomplished via certificate generation, which is used to verify the identity of the server, and by using port 8443 for secure transmission of data.

## Industry Standard Best Practices

To maintain the application’s existing security, I ran a static dependency check to look for any dependencies in the Spring Boot framework that would pose a security risk. From this list of vulnerabilities, I was able to deduce false positives and suppress them so that I could focus on the true vulnerabilities. Many of the vulnerabilities are mitigated via updates to the dependencies, or by following secure coding practices such as data throttling.

By applying industry standard best practices for secure coding, many of the concerns over a vulnerable application are reduced or eliminated. These practices help to ensure the integrity of the customer’s data, instilling a greater sense of trust for the company. This trust serves as the foundation of the company and must be protected at all costs. Also, by employing secure coding practices, we are able to maintain the integrity of our applications and data systems. This will prevent bad actors from gaining access to sensitive information while also making sure that customer data is available to authorized users when needed. By having their data protected and available, customers will continue to have faith in the company and use its services for years to come.