

# 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** | **4/19/25** | **Nathaniel White** |  |

## 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

Nathaniel White

## Algorithm Cipher

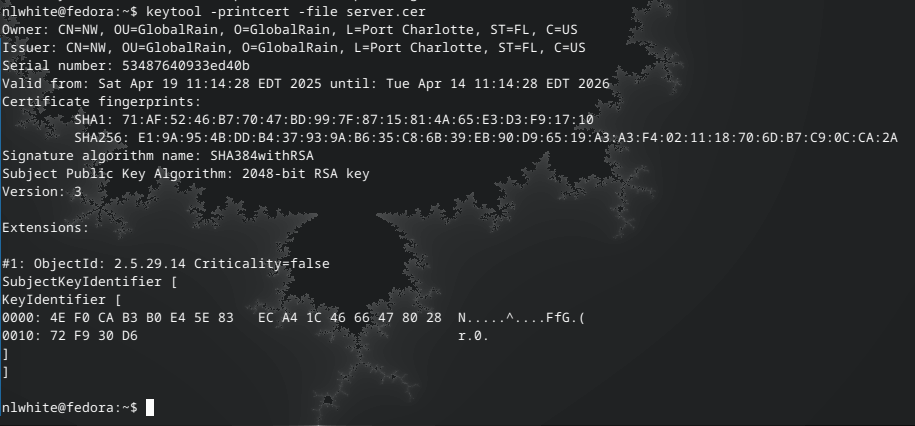
At the core of essentially every encryption algorithm is a hash function. A hash function is a special type of function that takes a value as input and returns a value that, while not appearing like the original value at all, can reproduce the original value without fail. The value returned by the function typically holds the same amount of data regardless of the input value, known as the bit-level of the algorithm.

There are two types of encryption algorithms: symmetric and asymmetric. Symmetric algorithms are the type of encryption that most people are familiar with. By taking some value (typically a string) as input, the algorithm obfuscates the data while maintaining its uniqueness. In other words, the algorithm turns the data into something no one can read/understand using a passphrase and can revert the data back to its legible form when given the same passphrase. The algorithm is symmetric because the same passphrase can be used to decrypt and encrypt the data. Non-symmetric algorithms, also known as public key cryptography, uses a keypairs to facilitate secure data transfers. Keypairs include a private key and a public key. Both keys are held by the user sending the data, but only the public key should be shared. The private key is used to sign the data, while the public key is used to validate that signature. The algorithm is asymmetric because one key is used to decrypt the information, while the other key is used to encrypt the information (Manico & Detlefsen, 2014). Let’s say two users were to communicate using this system; I’ll call them A and B. With both users owning a keypair (one public key and one private key), A would encrypt their data with B’s public key, while B encrypts their data with A’s public key. When A sends a message to B, B can decrypt that message using their private key even though they never shared it with A.

Over the years, encryption algorithms have gotten stronger and stronger to defend against bad actors. At the time of writing, there exists two main options which are considered the standard. One is Advanced Encryption Standard (AES) and the other is Secure Hash Algorithm 256-bit (SHA-256). It should be noted that both of these algorithms are also Payment Card Industry Data Security Standard (PCI DSS) compliant. However, AES is currently not possible using the MessageDigest class within java.security. There exists other cryptography classes which do support AES, but the resulting code from using both MessageDigest and the Cipher class (from javax.crypto) is more complex with little to gain. Meaning, using SHA-256 will lead to less lines of code to maintain. For these reasons, I’ll be using SHA-256.

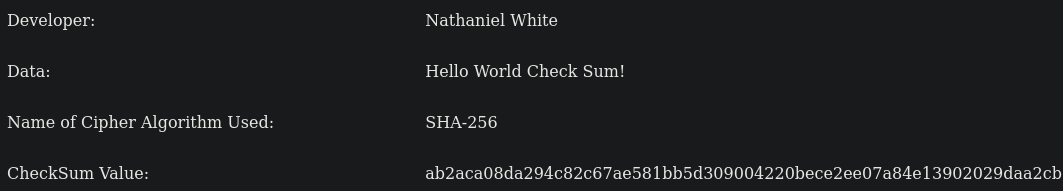
## Certificate Generation

Insert a screenshot below of the CER file.



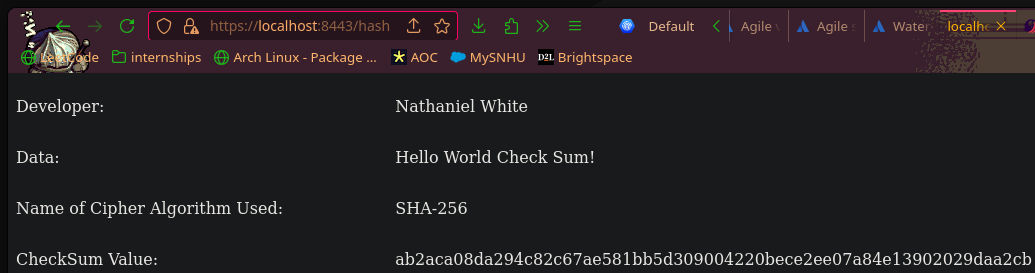
## Deploy Cipher

Insert a screenshot below of the checksum verification.



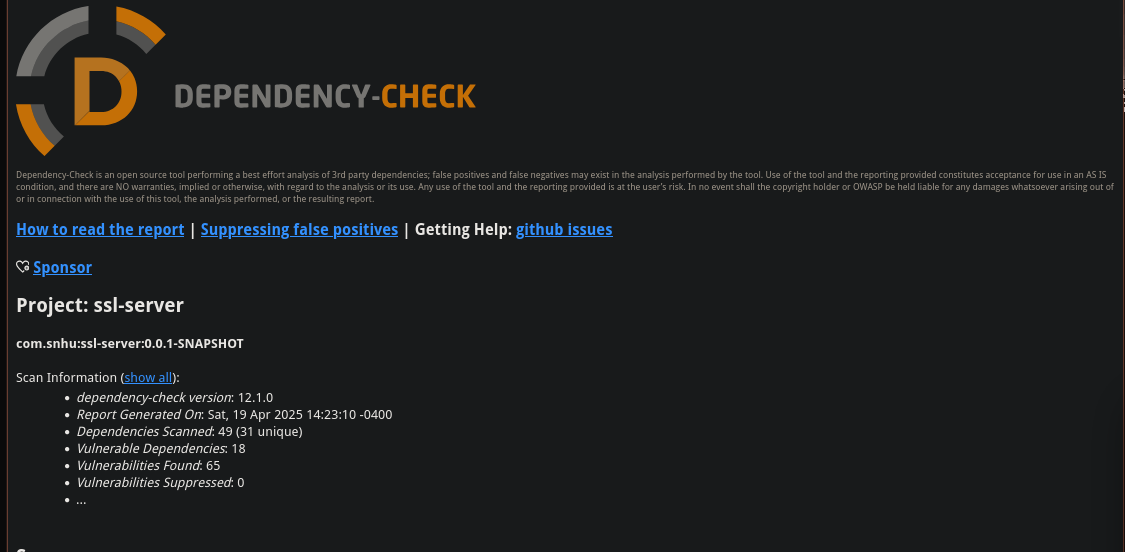
## Secure Communications

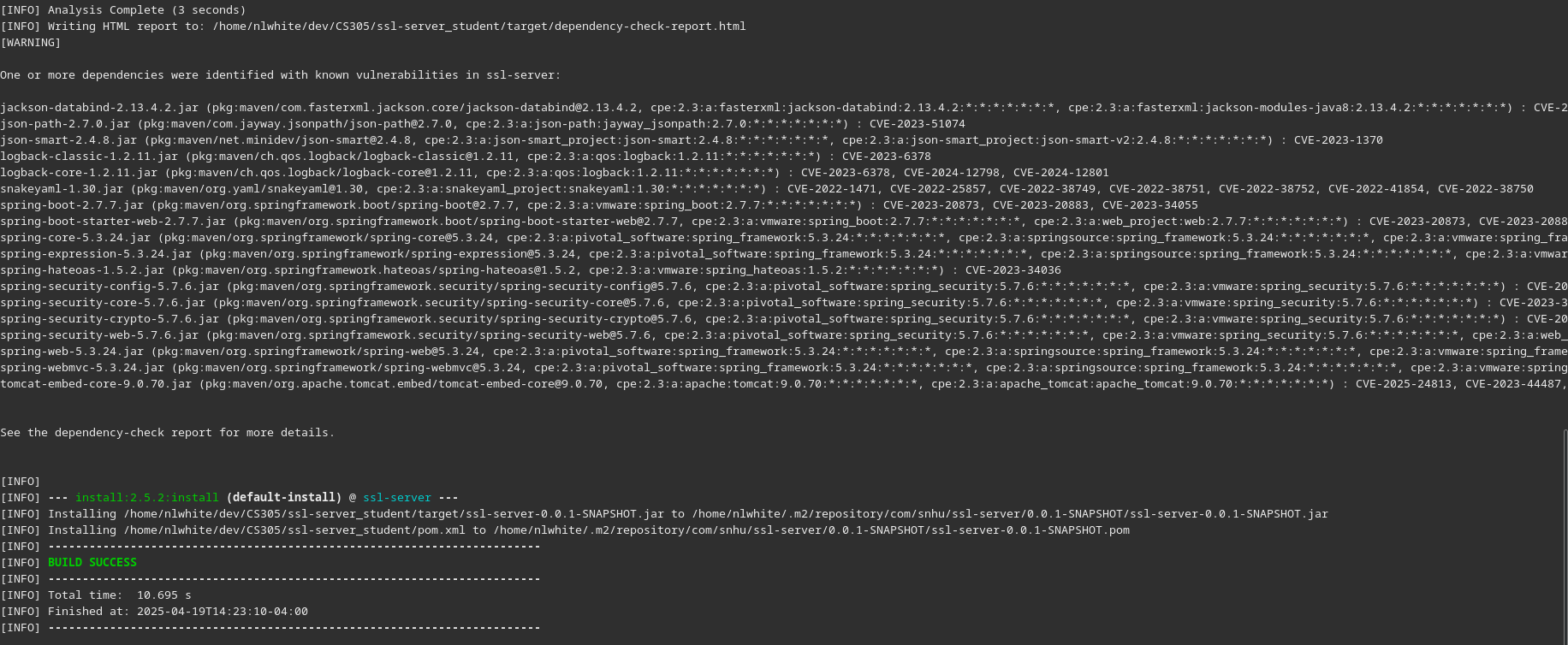
Screenshot of the web browser that shows a secure webpage:



## Secondary Testing

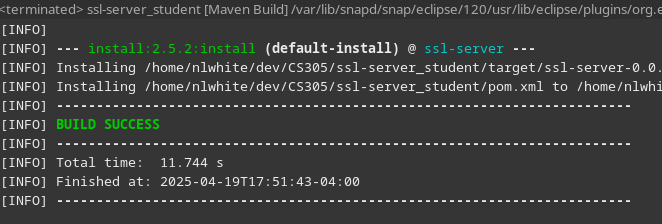
Refactored code executed without errors and the dependency-check report:





## Functional Testing

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



## Summary

When considering the Vulnerability Assessment Process Flow Diagram, there some items deserve more attention than others.

|  |  |
| --- | --- |
| **Vulnerability Process Diagram Node -** | **- Risk Factor** |
| Input Validation - | - LOW |
| APIs - | - MID |
| Cryptography - | - HIGH |
| Client/Server - | - HIGH |
| Code Error - | - MID |
| Code Quality - | - MID |
| Encapsulation - | - LOW |

The core areas I addressed while refactoring the code was cryptography and client/server connection. Of course, it is always important to maintain code quality and seek out errors, hence the medium risk factor. Although I am not necessarily permitted to be making big changes regarding the APIs in use for the application, I did have to update the parent pom within pom.xml in order for the application to build successfully, giving another medium risk. As for encapsulation, it is partly forced upon the code via annotations which are tied to either a class or method name. So long as the commonly known SOLID principles are used to ensure code quality, there’s not much risk here. Lastly, there is no input *to* validate, so no need to worry about input validation.

As previously discussed, I chose the encryption algorithm with care specifically to prevent security vulnerabilities. I also wrote a WebSecurityConfig class which tells the server what level of security a client must need in order to visit the URI(s) associated with the server’s domain (@RestController class methods). In this case, there is no delineation of roles, as anyone may perform the hash function without fear of a cyber threat. What is the point of the security class is if anyone is let through the door? Well, the explicit authorization using SecurityFilterChain is what allows the endpoint to use Hyper Text Transfer Protocol Secure (HTTPS). HTTPS adds a security layer on top of HTTP by requiring a certificate to be signed by an authority before delivering a payload. Meaning, in order to make a successful connection to server via HTTPS, the server must be ready to present a signed certificate to the client.

With the codebase refactored, it is now possible to generate a checksum verification value based on a static data value from any computer connected to the same network as the machine running the application.

## Industry Standard Best Practices

I followed industry standard best practices by choosing an encryption algorithm sufficiently advanced enough to comply with modern encryption standards, namely PCI DSS. Additionally, I also choose a sufficiently complex password for the encrypted keystore, Although, the password is still stored in a plain-text application.properties file which is what Spring Boot uses to configure the embedded tomcat server. Care should be taken with this file to not be shared within version control systems if applicable.

Aside from the security of the application, I also maintained a consistent code style in efforts to product higher code quality. This meant adhering to commonly used naming conventions and taking care to name functionality in an intuitive manner. For example, in order to convert bytes (raw memory) into a hex value (the resulting checksum), I created two functions, each one with a distinct job, rather than one large function that cannot be easily reused. It is important to follow industry standards such as these because they ensure that all software is treated with the same level of rigor. This would include but is not limited to the design, implementation, and testing of the software product.

Citations

Manico, J., & Detlefsen, A. (2014c). *Iron-Clad Java: Building secure web Applications (Oracle Press)*. http://dl.acm.org/citation.cfm?id=2826076