****

# Practices for Secure Software Report

Table of Contents

[Practices for Secure Software Report 1](#_Toc201534675)

[Document Revision History 3](#_Toc201534676)

[Client 3](#_Toc201534677)

[Instructions 3](#_Toc201534678)

[Developer 4](#_Toc201534679)

[1. Algorithm Cipher 4](#_Toc201534680)

[2. Certificate Generation 4](#_Toc201534681)

[3. Deploy Cipher 5](#_Toc201534682)

[4. Secure Communications 5](#_Toc201534683)

[5. Secondary Testing 6](#_Toc201534684)

[6. Functional Testing 7](#_Toc201534685)

[7. Summary 8](#_Toc201534686)

[8. Industry Standard Best Practices 9](#_Toc201534687)

[9. Comprehensive List of Code Changes 10](#_Toc201534688)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **06/22/2025** | **Amauri Hopewell** |  |

## 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

Amauri Hopewell

## Algorithm Cipher

The first encryption algorithms were probably used by the Roman or Greek military. Between WW1 and WW2, military encryption became very sophisticated. As computers became more prominent, algorithms continued to develop, particularly in the context of internet communications, and for a while, elliptic curve cryptography provided even more secure algorithms. However, with the rise of quantum computing, many of these algorithms might soon be breakable, and the rise of quantum-resistant encryption seems to be the next big frontier. This shows that cryptography is always evolving, and Artemis must be ready to face current and future threats.[[1]](#footnote-2)

In making its choice, Artemis must first distinguish between true encryption algorithms and cryptographic hash functions.

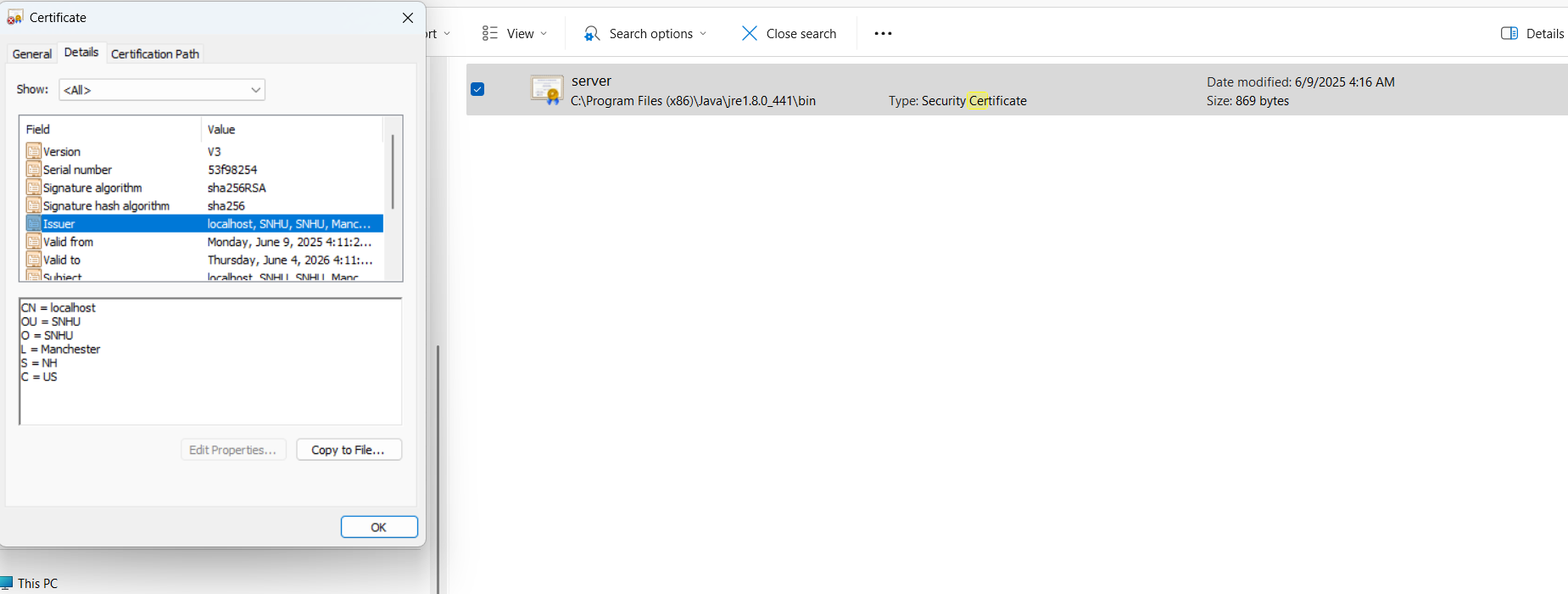
Because the main task here involves generating checksums, I chose the SHA-512, a cryptographic hash function which stands for “secure hash algorithm” with 512 bit (64 byte) digest, for my cypher function.

Similar to other SHA algorithms the SHA-512 algorithm breaks the input message into 8 blocks (adding padding when needed), and shifts the 8 blocks around each cycle of the algorithm. It then uses a series of predetermined binary sequences which function as pseudorandom, but not truly random, numbers, to make outputs consistent yet difficult to reverse engineer. The data is combined with itself and these pseudorandom sequences using a complex mix of bitwise shifting within blocks, and using negation, logical and, and exclusive or, and addition modulo 264, to not only make the end outcome tough to predict, but also make it tough to reverse engineer the results of any given round of encryption. To guarantee security, this goes on for 80 rounds, well beyond what the best collision attacks so far can replicate.

This is an excellent example of a hash function, rather than a reversible encryption function. If it were a reversible encryption function, there would need to be a choice between “symmetric key” algorithms and “asymmetric key” algorithms. Symmetric key algorithms, where the same key is used to encrypt and decrypt the data, are useful for transferring data when two trusted parties have the key but interception is a threat. Non-symmetric key algorithms are more useful for initial authentication. For example, in website connections, the non-symmetric RSA algorithm might be used to establish the connection, with AES used for actually transferring data.

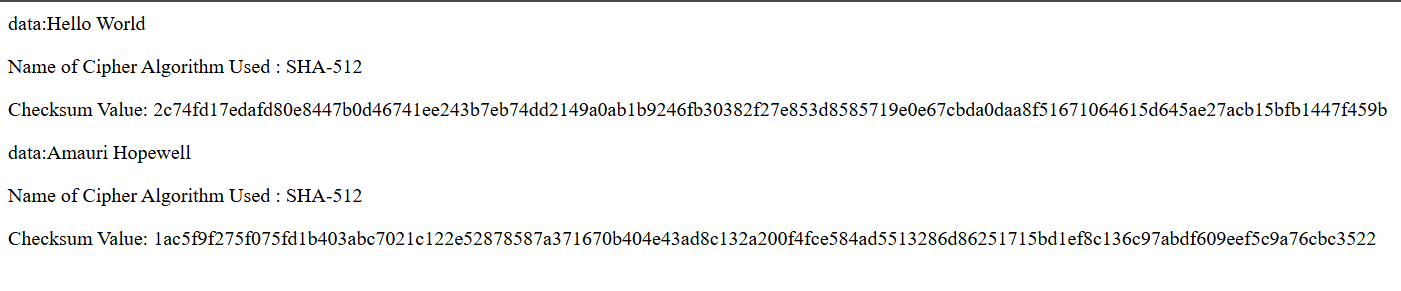
## Certificate Generation

A security certificate that a web browser can use to show what website the browser is connecting to, and verify it using private key encryption. While third-party security certificates are best, because they are independently verified, sometimes things like cost savings require using a privately generated security certificate. I did so using Java’s Keytool command here, resulting in the following .cer file, server.cer:



## Deploy Cipher

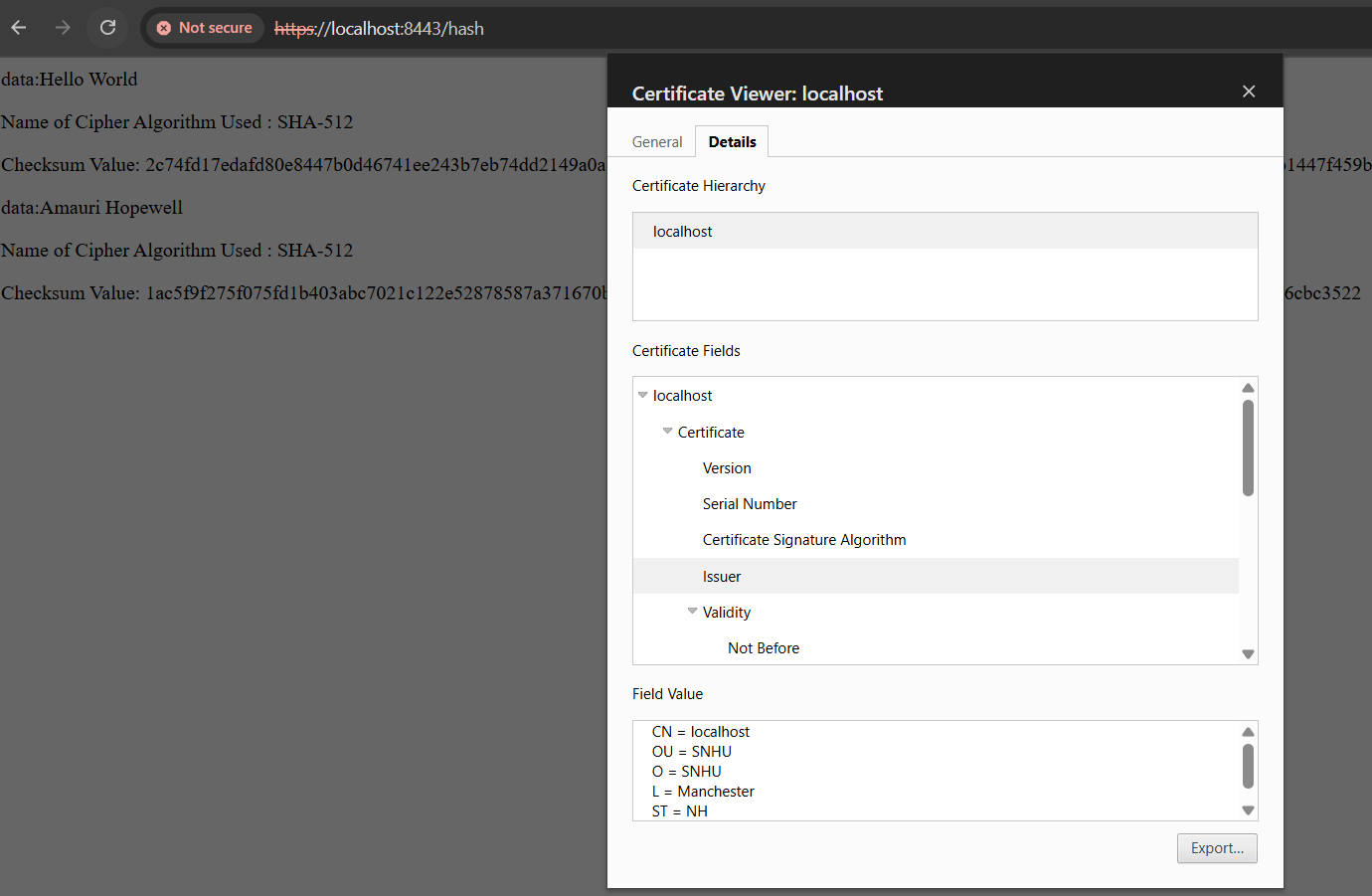
The following display of the webpage shows how SHA-512 can be used to generate checksums for strings, here one for “Hello World” and one for my name:



## Secure Communications

The following screenshot shows how the refactored website uses this .cer file to create a secure, HTTPS website with a security certificate. While Chrome displays an unsecured warning, this is a known issue that is difficult to fix even after importing the certificate into the list of trusted authorities, as explained here: <https://superuser.com/questions/1655188/chrome-still-displays-website-as-not-secure-despite-root-ca-being-added-to-tru>

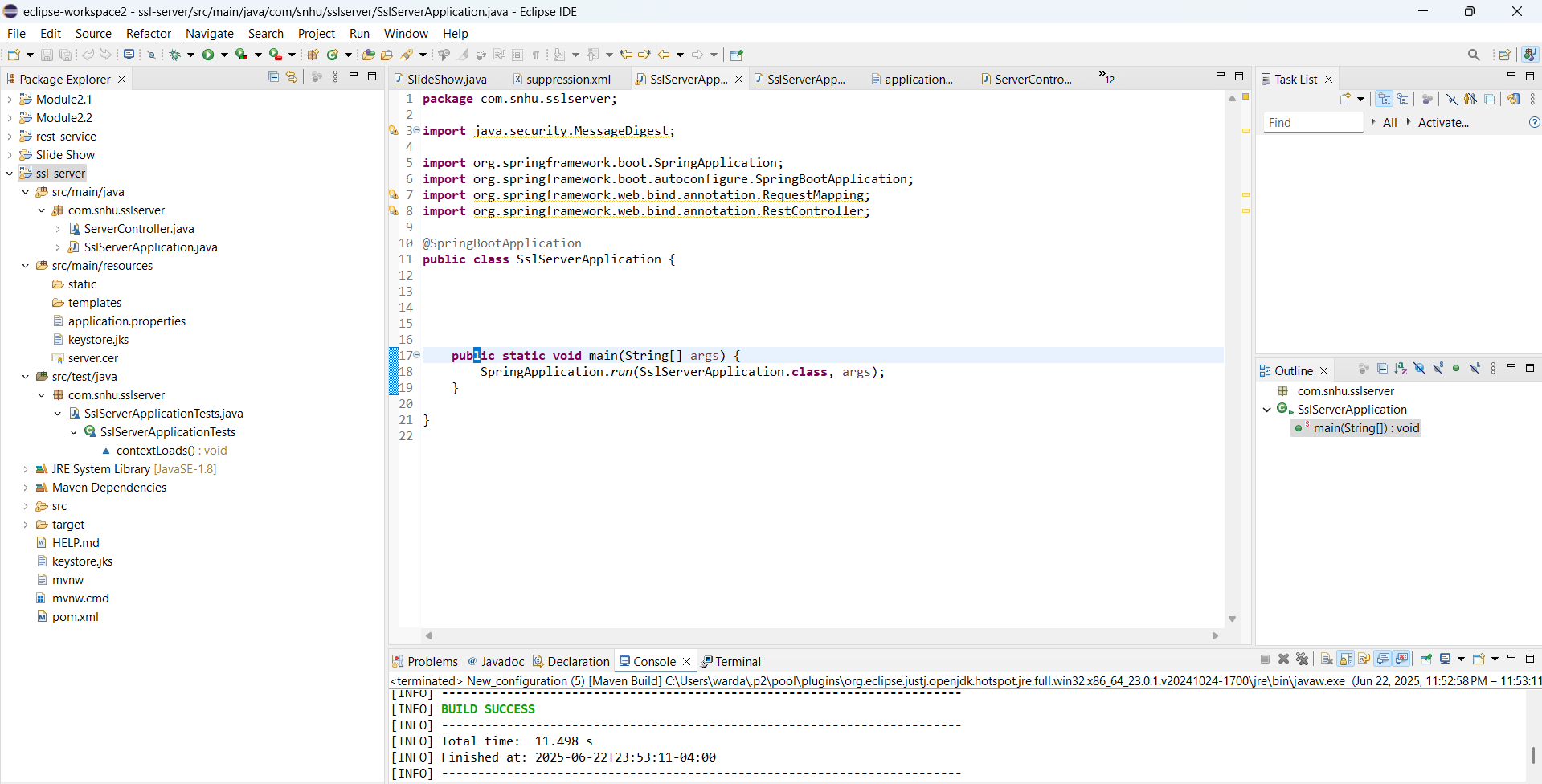
Nonetheless, Chrome clearly shows that the website is using HTTPS and the created certificate:



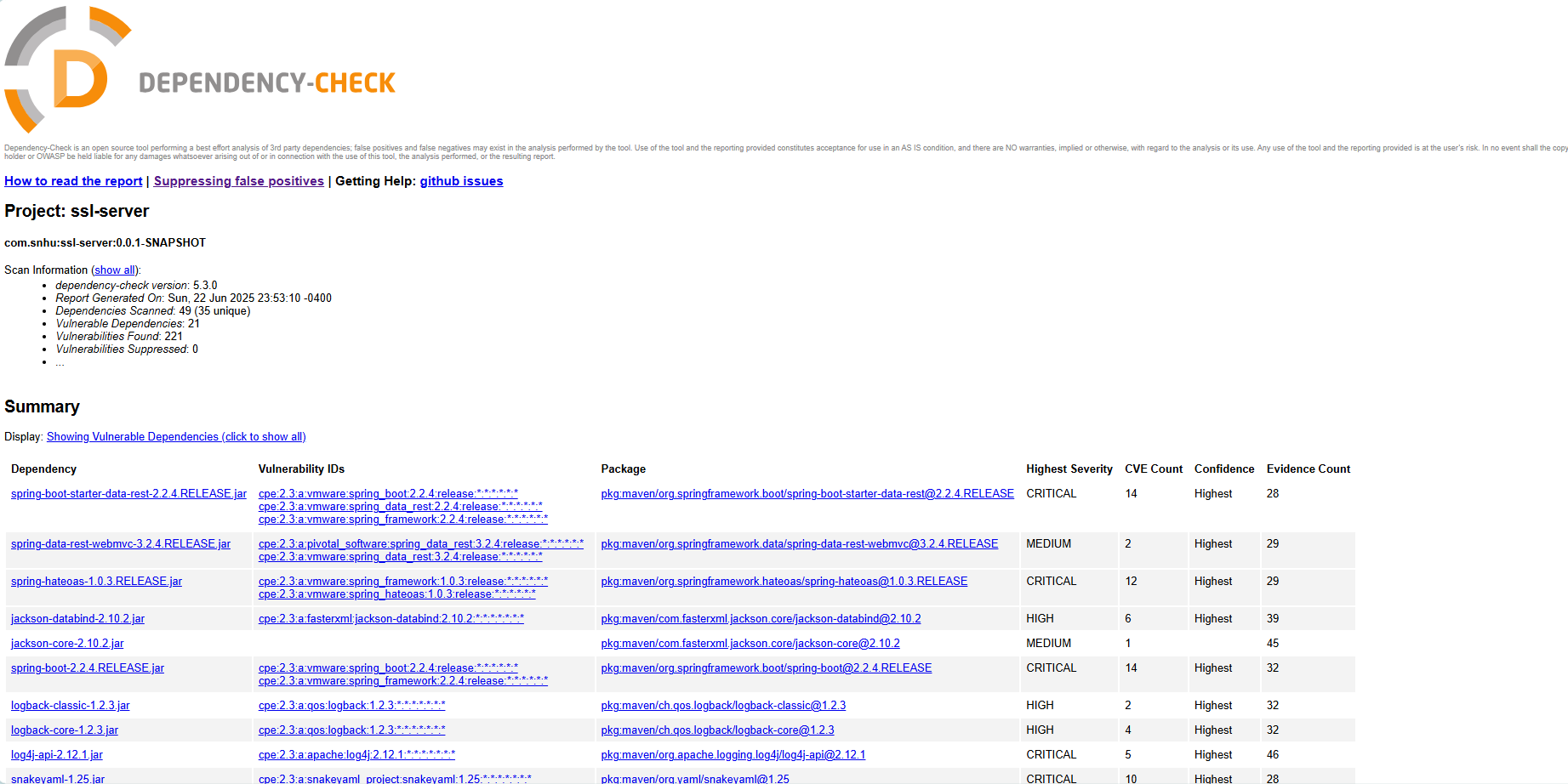
## Secondary Testing

To refactor the code to make it more secure and add functionality, I added the required checksum algorithm, required the use of HTTPS, and separated out server application functionality from the server controller, which handles generating checksum of strings. This will be discussed in more detail below.

The following screenshot shows the refactored code, broken into more secure classes, successfully being built as a Maven project to use dependency-check



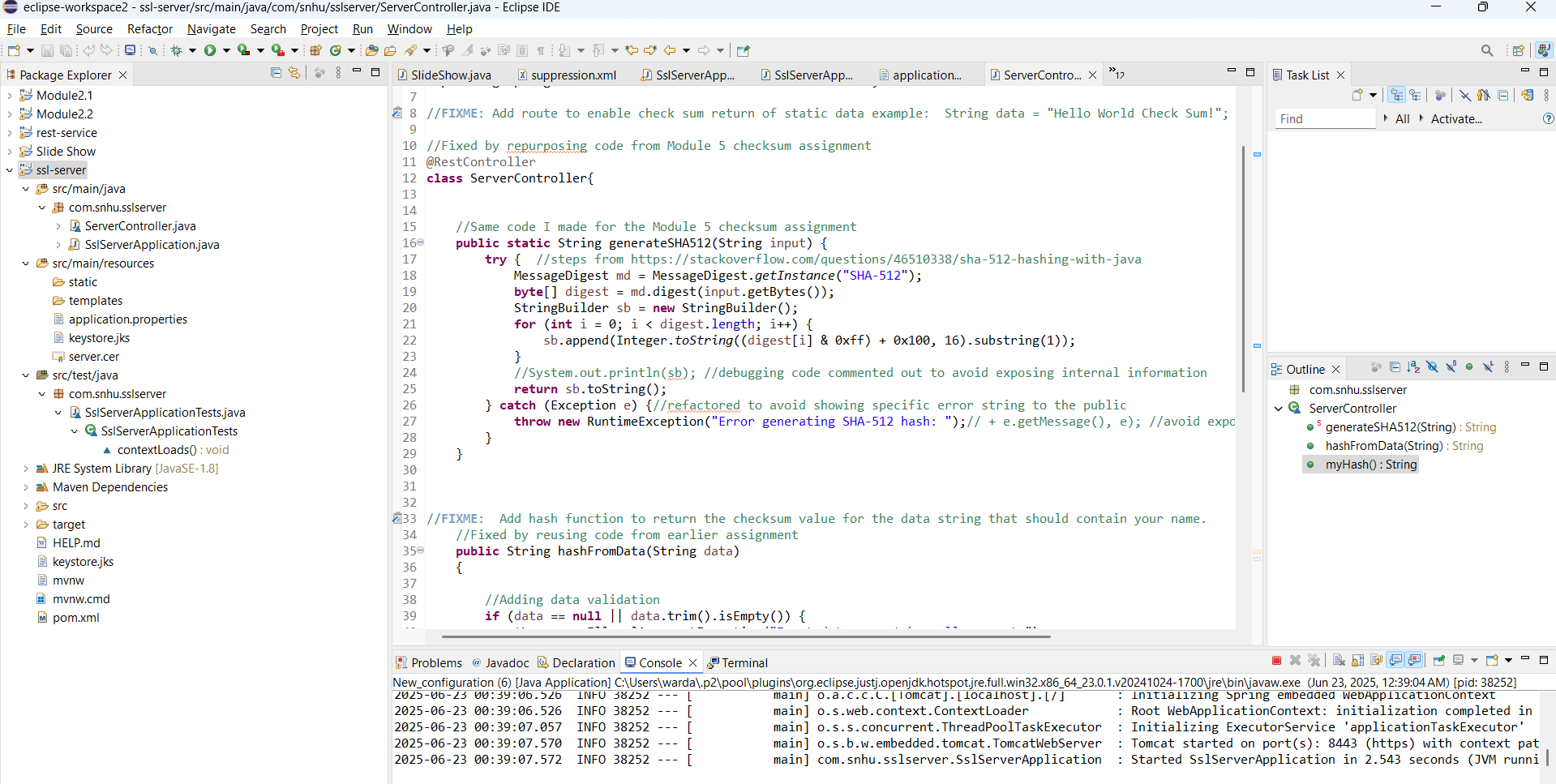
The following screenshot shows the results of the dependency check. The refactoring did not introduce any new errors.



## Functional Testing

To refactor the code to make it more secure and add functionality, I added the required checksum algorithm, required the use of HTTPS, and separated out server application functionality from the server controller, which handles generating checksum of strings. This will be discussed in more detail below.

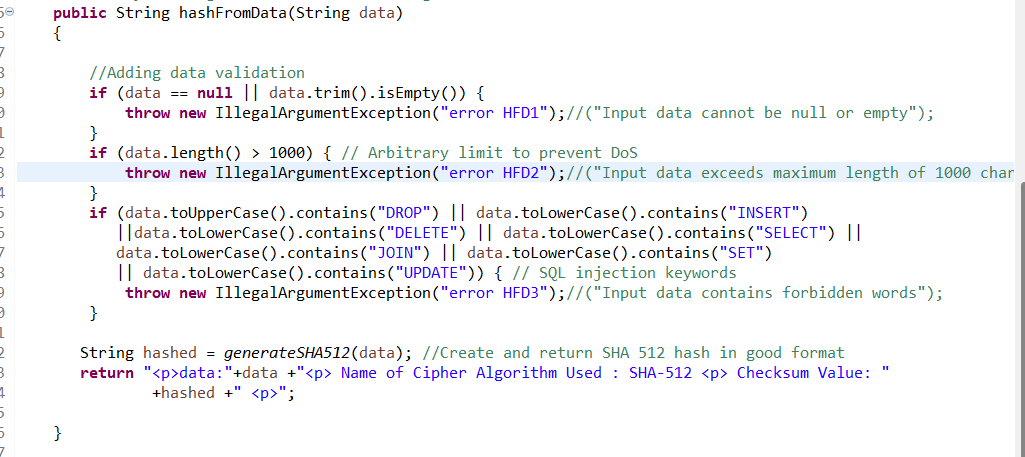
The following screenshot shows this refactored code, broken into more secure classes and with added SHA-512 functionality, successfully running as a Java application:



## Summary

To refactor the code to make it more secure, I focused on a few key areas from the SNHU vulnerability assessment process flow chart. In particular, I focused on the following areas

* Input validation
  + Because the ServerController class used functions involving string input and creating subsequent output to create hashed output, I created an input sanitization feature for the function that takes a string a calls the hashing function.
  + In particular, in the “hashFromData” function, I ensured that an error would be thrown if the data was empty, or if the data was excessively long, or if the data contained the 7 most common SQL keywords (to prevent against injection attacks), and I also ensured that non-descriptive numerical errors were thrown so attackers could not figure out how the system was reacting to their attacks:



* Client/Server
  + This is a website, and so there are significant concerns about clients sending malicious requests to the servers. To address this type of vulnerability, I focused on the possibility of SQL injection attacks, as discussed in the input validation bullet.
  + I also changed the application.properties file and added the line “server.http.port=8765” to ensure that, by default, the connection to port 8443 will *only* allow HTTPS, and forcing a redirect to this HTTPS port to achieve functionality.
* Code error
  + To ensure that code errors were unlikely to occur and be exploitable, I used numerous try/catch blocks, to help thwart malicious input. I also ensured that these did not print revealing messages to the website (as shown in the screenshot above), and I deleted all “system.println” statements that might have been used for debugging, because a hacker could use them to fine tune his attacks.
* Encapsulation
  + I ensured that different classes handled different key aspects of the server, and also encapsulated the underlying hash checksum functionality, using a separate function within a controller class.
* Controllers
  + The initial code had the ServerController and SpringBootApplication in the same class. To ensure that each controller handles only one thing, I moved these into separate classes, making it more difficult to find a single point of attack that would allow seriously compromising the application.

My personal process for adding layers of security to the application involves first scanning for obvious errors, looking for things like unhandled exceptions or unsanitized inputs. Next, I go through the flowchart, looking at each area and comparing it to each class in the project to ensure I have considered every aspect of it. Next, I use tools like dependency check to automatically look for potential vulnerabilities, and assess whether there is a need to upgrade software. Finally, I use the application or website, and try to think of ways I might break it. I then refactor the code using what I have found, and repeat the process over multiple times.

## Industry Standard Best Practices

In refactoring the application, I found it was important to use industry best practices. Many of these are handled by the flowchart referenced in Section 7, but it is also important to consider the overall security workflow, as the “DevSecOps” movement stresses. Industry best practices that I found important were the use of automated testers like OWASP’s dependency check, the use of procedural aids like flowcharts, and the use of up-to-date and standardized tools and libraries.

Using up-to-date and standardized tools was perhaps the most important one. These standardized tools range from Maven for integrating plugins, well-known libraries like Spring and Tomcat for servers, and algorithms like SHA-512 or SHA-256 for hashing. Using these tools and best practices allows benefiting from other dedicated and intelligent security professionals, rather than having to constantly reinvent the wheel and keep up with a seemingly endless amount of hackers. And keeping them up-to-date keeps the application secure against new attacks and bugs. This might not result in perfect security, but it can prevent a great deal of risk at a relatively low cost.

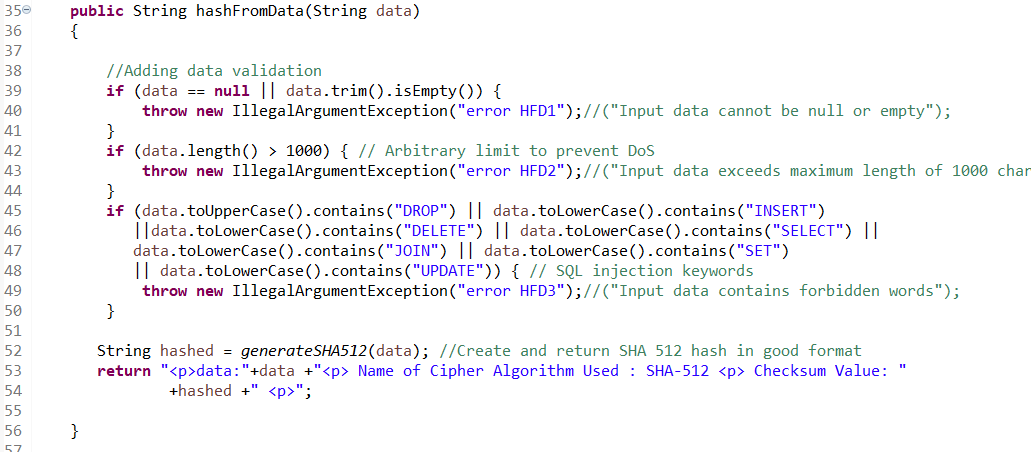
## Comprehensive List of Code Changes

As a reference, here is the full list of changes I made to the code:

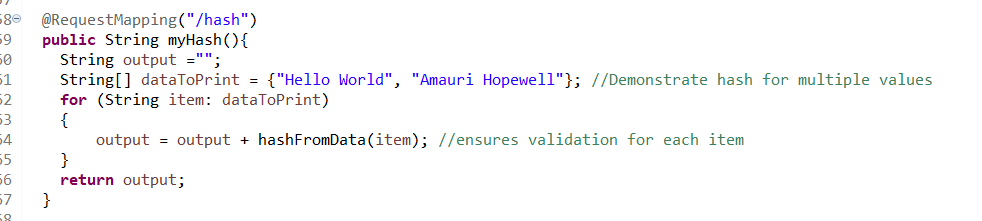
* Moved all ServerController code out of SslServerApplication and into its own class, ServerController.Java



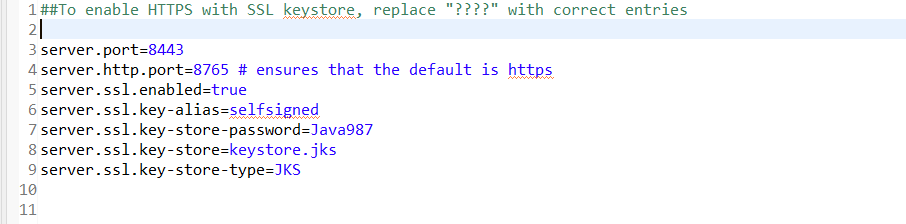
* Added a function generateSHA512 to the ServerController class, for creating checksums from a given input text
* Changed the exception handling in generateSHA512 to avoid printing revealing information to the webpage
* Created a hashFromData function in the ServerController class to sanitize input before applying the SHA-512 hashing.
* Changed my initial error messages in this function to more cryptic ones to avoid revealing sensitive information



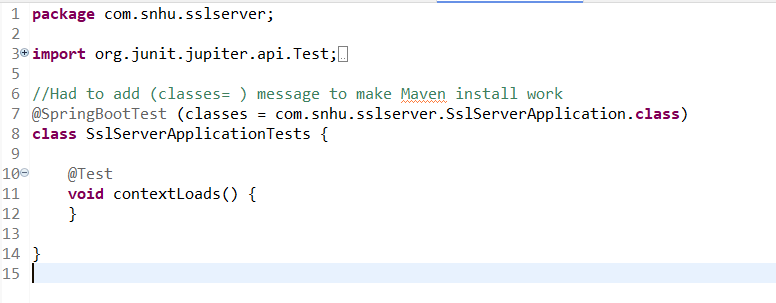
* Called this sanitizing function in the hash request mapping to avoid unsafe input (though the input to the function is fixed for now)



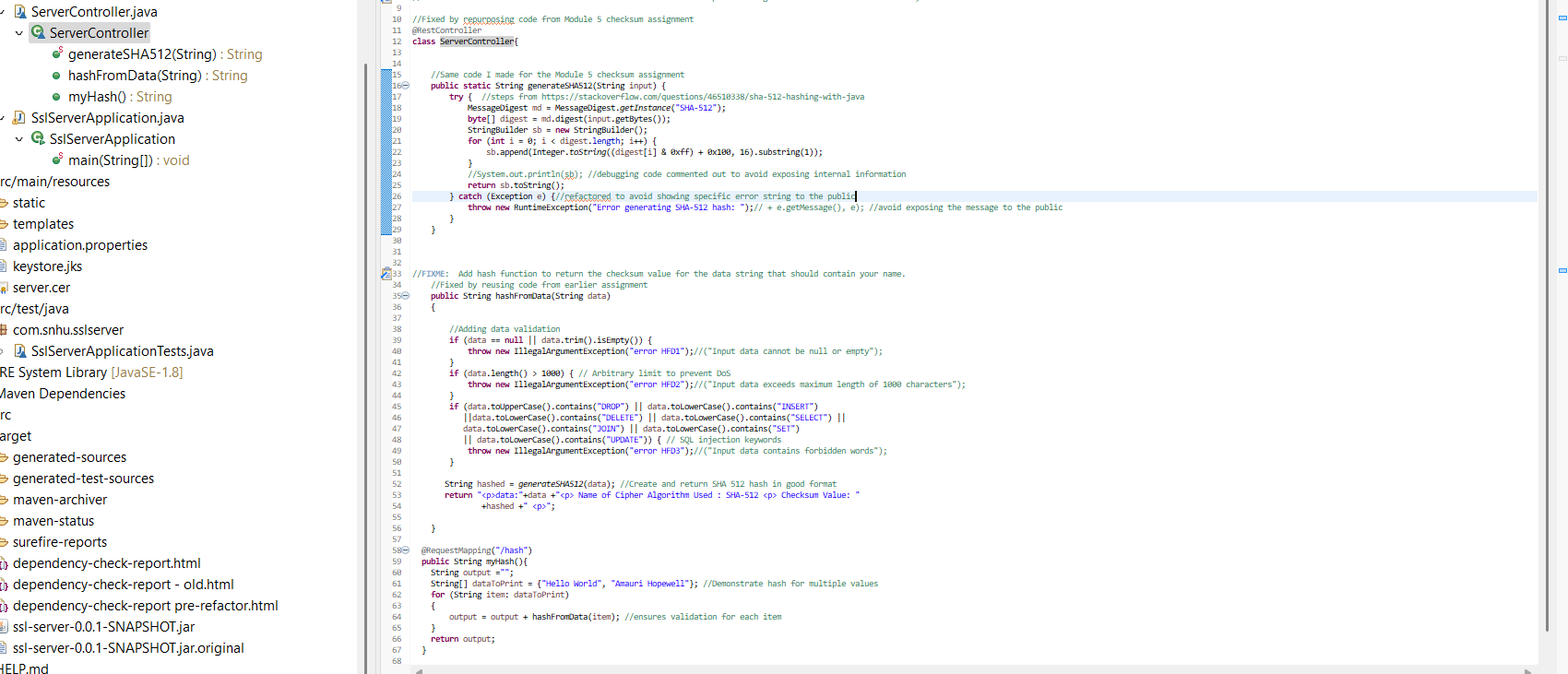
* I changed the application.properties file to use HTTPS with a signed certificate, and restricted use of regular HTTP on the normal port:

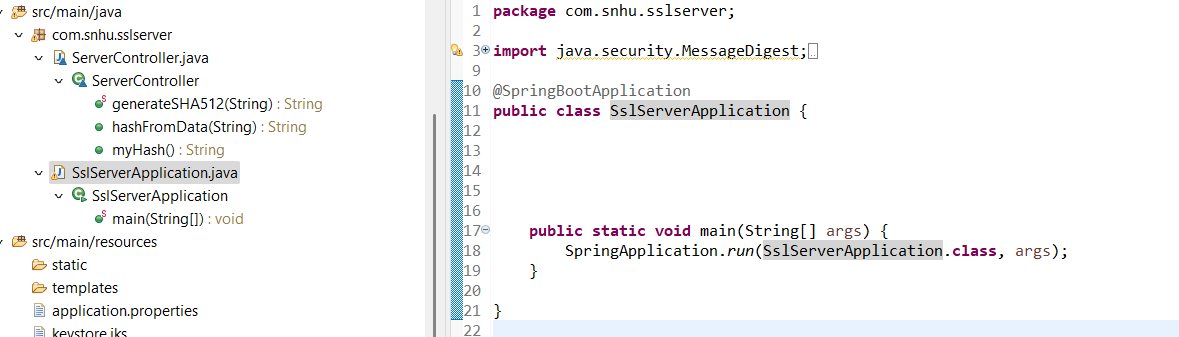


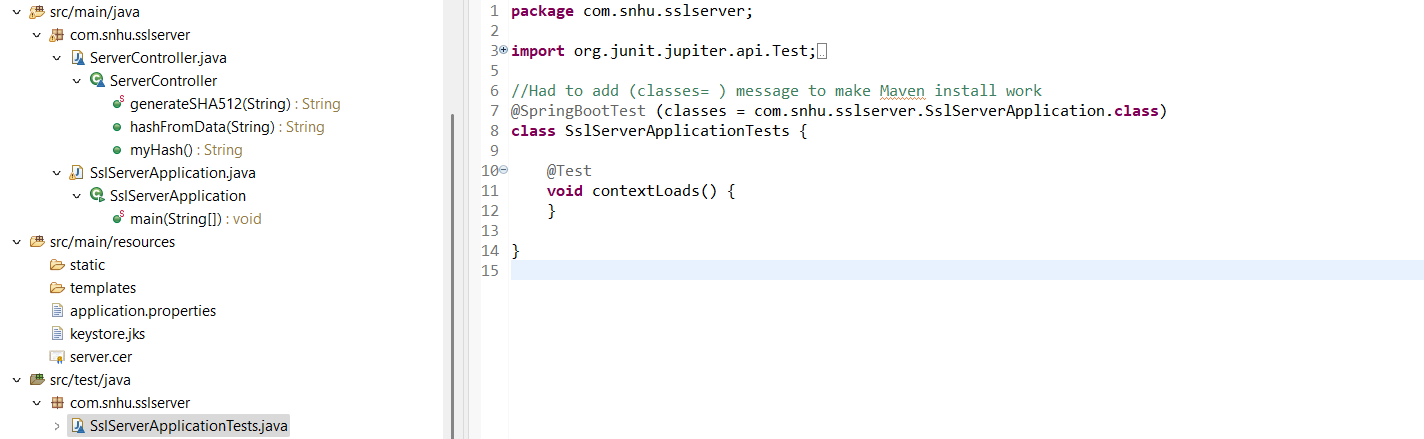
* Added parameters to the springBootTest function to ensure compatibility with Maven and OWASP’s dependency check



* The full view of each class is now as follows:







* After this, I reran dependency-check and ensured there were no new vulnerabilities.

1. Some wording in this report is reused from my earlier reports in this course, combined together into a coherent whole here. See those reports for additional sources. I have chosen not to include additional sources here for brevity because most of them stand for readily verifiable facts that are commonly known among security professionals. [↑](#footnote-ref-2)