

# 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** | **4/13/2023** | **Scott Mansfield** |  |

## 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

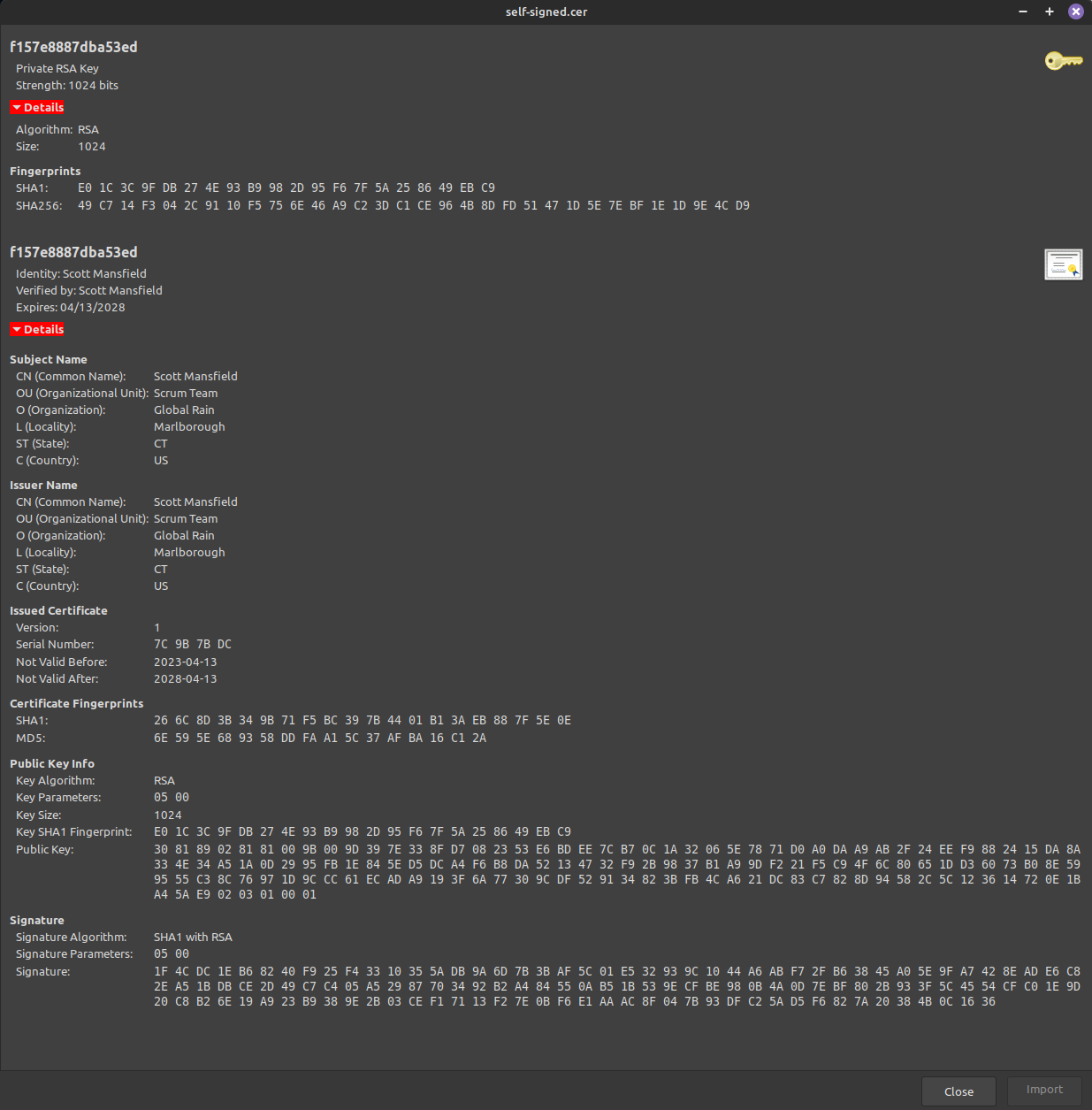
Scott Mansfield

## Algorithm Cipher

The Algorithm I have chose to use for generating these checksums is SHA3-512. The SHA3-512 algorithm is one of, if not, the most secure hashing algorithm out today. It is also the most secure algorithm that can be used with the MessageDigest class. Although SHA-2 algorithms are still considered secure enough for government applications and has almost identical performance to the new SHA-3 algorithms according to NIST documentation (<https://csrc.nist.gov/projects/hash-functions> , accessed 3/31/2023), I assume that eventually SHA-3 will become the standard. If there is a question whether or not a 512 bit hash is overkill, when testing it using a 1024 bit input data the program only took 3-4 seconds to execute. Most public keys are around 1024-2048 bits in length and that is what we will be hashing with this program.

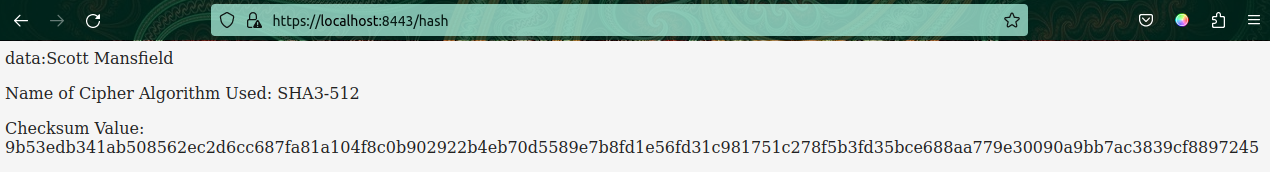
## Certificate Generation

Insert a screenshot below of the CER file.



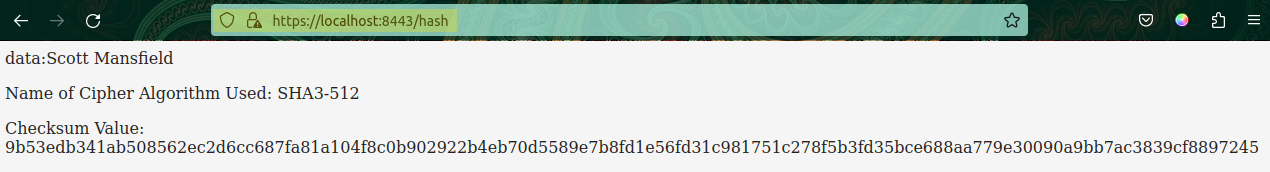
## Deploy Cipher

Insert a screenshot below of the checksum verification.



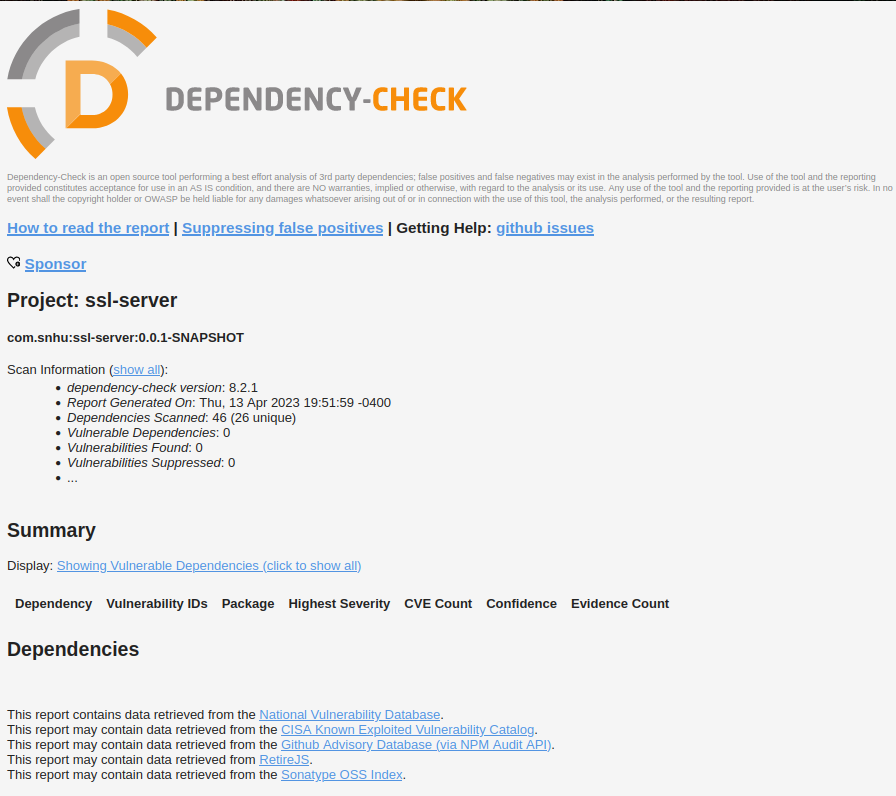
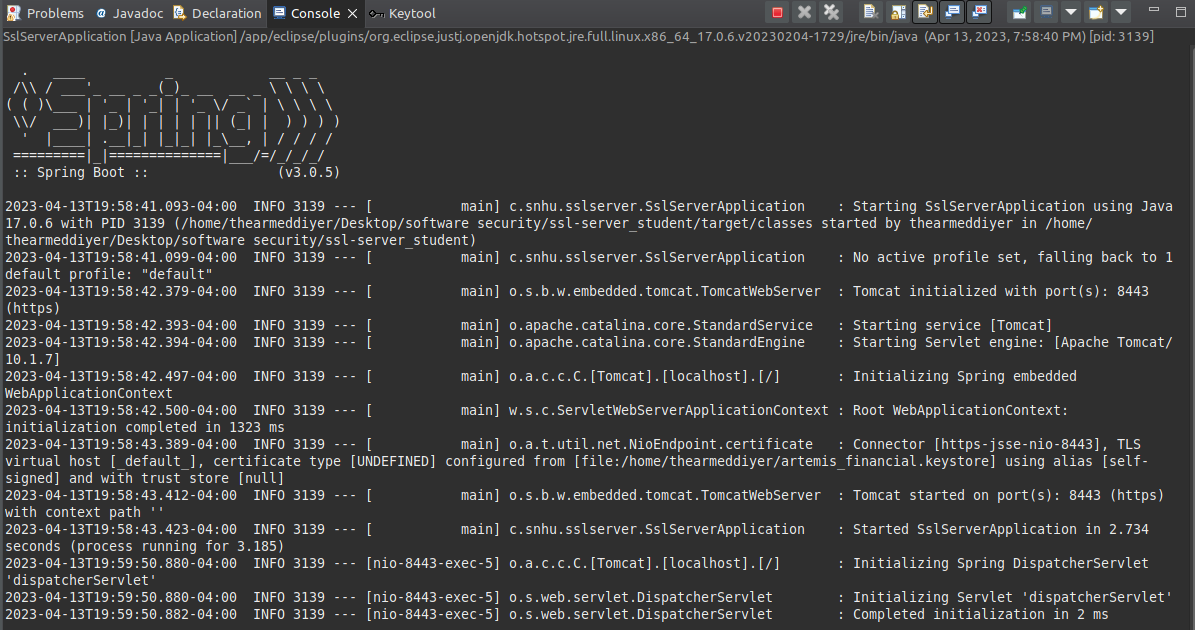
## Secure Communications

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



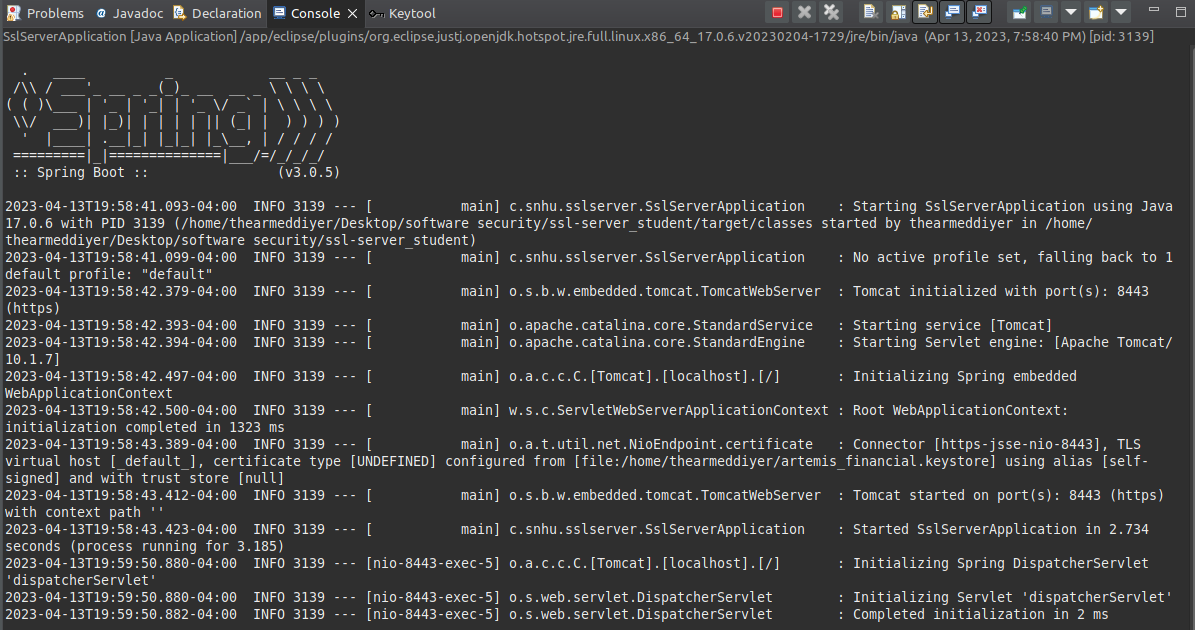
## Secondary Testing

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

Before I added any code, the first order of business was to update the pom.xml file to the current version of Maven and run a dependency check. This verified that the current code was accurate and gave me a baseline to start with. I installed the keytool plugin and proceeded to create a self signed CA to be used for development purposes. Once I had this information I could update the application.properties file with the new CA and keystore information to be used to implement a secure server communication later. I wont discuss the cypher I used to generate a checksum here as it was discussed that the beginning of the report. What I did was to add a new class called ServerController and used messageDigest and bigInt to create a hash with my chosen cipher. This code I had previously developed and is known to function well. From here I ran the code to verify it was working properly and then conducted a new dependency check. As I had not made any changes to the dependencies, this report was unchanged from before. I noted that the Spring Boot framework was utilizing an old version so I changed the pom.xml file to reflect the current version. This cleared up all but one of the dependency vulnerabilities which turned out to be snakeyaml. After doing research, Spring Boot was calling on snakeyaml version 1.33 which had known vulnerabilities whereas version 2.0 is currently secure. I had to add a dependency to the pom.xml file to force the application to use snakeyaml v. 2.0 in order to solve this vulnerability. As you can see above, the resulting dependency report shows no vulnerabilities with 0 dependencies suppressed. Then I ran the application again one last time and verified all was working well with no errors.

I have included all of the refactored code as an attachment to this report. The refactored code also includes the screenshots provided here as well as the before and after dependency check reports.

## Industry Standard Best Practices

Upgrading dependencies to the most current versions is one of the easiest and effective ways of avoiding vulnerabilities in applications. Dependencies are updated regularly and usually to patch security vulnerabilities, so this becomes ever the more important.

Securing communication with the application by using a Certificate Authority (CA) helps verify to the user that what they are accessing is correct and safe.

However, self-signed CA’s should only be used for development purposes and not out in the wild. Attackers can create their own self-signed CA’s and implement a man-in-the-middle attack due to lack of trusted third party signed CA.

Encrypting data and using checksum verification is an industry standard best practice as well which we have implemented here using a secure/unbroken cipher with an acceptable key length for the application.