

# CS 305 Project Two

**Practices for Secure Software Report**

Table of Contents

[Document Revision History 3](#_Toc33111302)

[Client 3](#_Toc33111303)

[Instructions 3](#_Toc33111304)

[Developer 4](#_Toc33111305)

[1. Algorithm Cipher 4](#_Toc33111306)

[2. Certificate Generation 4](#_Toc33111307)

[3. Deploy Cipher 4](#_Toc33111308)

[4. Secure Communications 4](#_Toc33111309)

[5. Secondary Testing 4](#_Toc33111310)

[6. Functional Testing 5](#_Toc33111311)

[7. Summary 5](#_Toc33111312)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **October 18, 2020** | **Dylan Oldenburg** |  |

## Client



## Instructions

Deliver this completed Practices for Secure Software Report documenting your process for writing secure communications and refactoring code that complies with software security testing protocols.

Respond to the steps outlined below and replace the bracketed text with your findings in your own words. If you choose to include images or supporting materials, be sure to insert them throughout.

## Developer

Dylan Oldenburg

## 1. Algorithm Cipher

Determine an appropriate encryption algorithm cipher to deploy given the security vulnerabilities, justifying your reasoning. Be sure to address the following:

* Provide a brief, high-level overview of the encryption algorithm cipher.
* Discuss the hash functions and bit levels of the cipher.
* Explain the use of random numbers, symmetric vs non-symmetric keys, and so on.
* Describe the history and current state of encryption algorithms.

Given the client's stated goal of file/data verification via checksum, I recommend the hashing algorithm SHA-256. SHA-256 is a widely used and secure algorithm. It is commonly used in certificate generation for SSL/TLS (Rhodes, 2020). SHA-256 takes data of any length, and hashes it into a pseudorandom 64-byte value. SHA-256 cannot be decrypted; due to the use of modulo arithmetic in the hashing algorithm, it is impossible to retrieve the original data based on the hash value. SHA-256 offers an extremely low risk of collisions (two inputs producing the same hash output), and has the benefit of the so-called "avalanche effect": even a small change to the input will result in a large change to the resulting hash value (Callaghan, 2019). SHA-256 became a widely-used standard after security vulnerabilities were found in SHA-1 algorithms. SHA-2 algorithms, including SHA-256, were created to address those vulnerabilities.

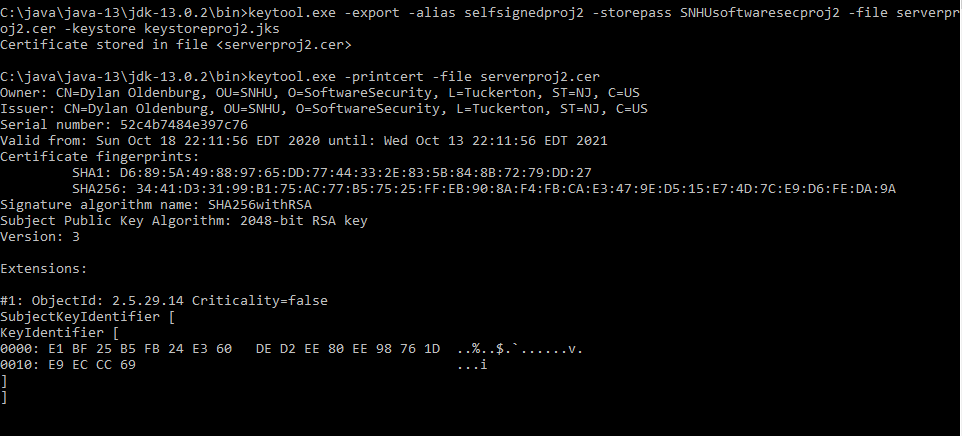
Generating truly random numbers is difficult for computers without an external source of entropy, which is not always available. So, encryption algorithms often rely on pseudorandom number generation, using extremely complex algorithms that produce outputs that are difficult to trace back to the original data. For example, the 2048-bit SHA-256 algorithm used in this application produces a pseudorandom value between 1 and 22048 -- a truly massive number. This is how the risk of collisions is kept to a minimum.

In encryption (not hashing) algorithms, the encrypted data can be encrypted with a key. This includes symmetric algorithms (where the same key is used for encryption and decryption) and asymmetric algorithms (where one key is used for encryption, and another for decryption). Asymmetric algorithms are slower than symmetric algorithms, but prove particularly useful for secure communications and digitally signing data transmissions.

## 2. Certificate Generation

Generate appropriate self-signed certificates using the Java Keytool, which is used through the command line.

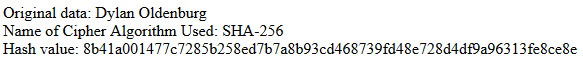
* To demonstrate that the keys were effectively generated, export your certificates (CER file) and submit a screenshot of the CER file below.

The above screenshot shows the printed certificate, including the algorithm used (SHA256withRSA), and the bit-length of the algorithm (2048-bit, or 64-byte).

## 3. Deploy Cipher

Refactor the code and use security libraries to deploy and implement the encryption algorithm cipher to the software application. Verify this additional functionality with a checksum.

* Insert a screenshot below of the checksum verification. The screenshot must show your name and a unique data string that has been created.

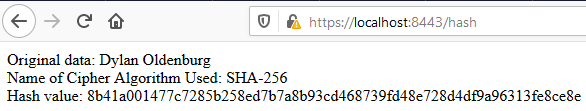


The above screenshot shows the original data (my name), the cipher algorithm used (SHA-256), and the resulting 2048-bit/64-byte hash value.

## 4. Secure Communications

Refactor the code to convert HTTP to the HTTPS protocol. Compile and run the refactored code to verify secure communication by typing **https://localhost:8443/hash** in a new browser window to demonstrate that the secure communication works successfully.

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

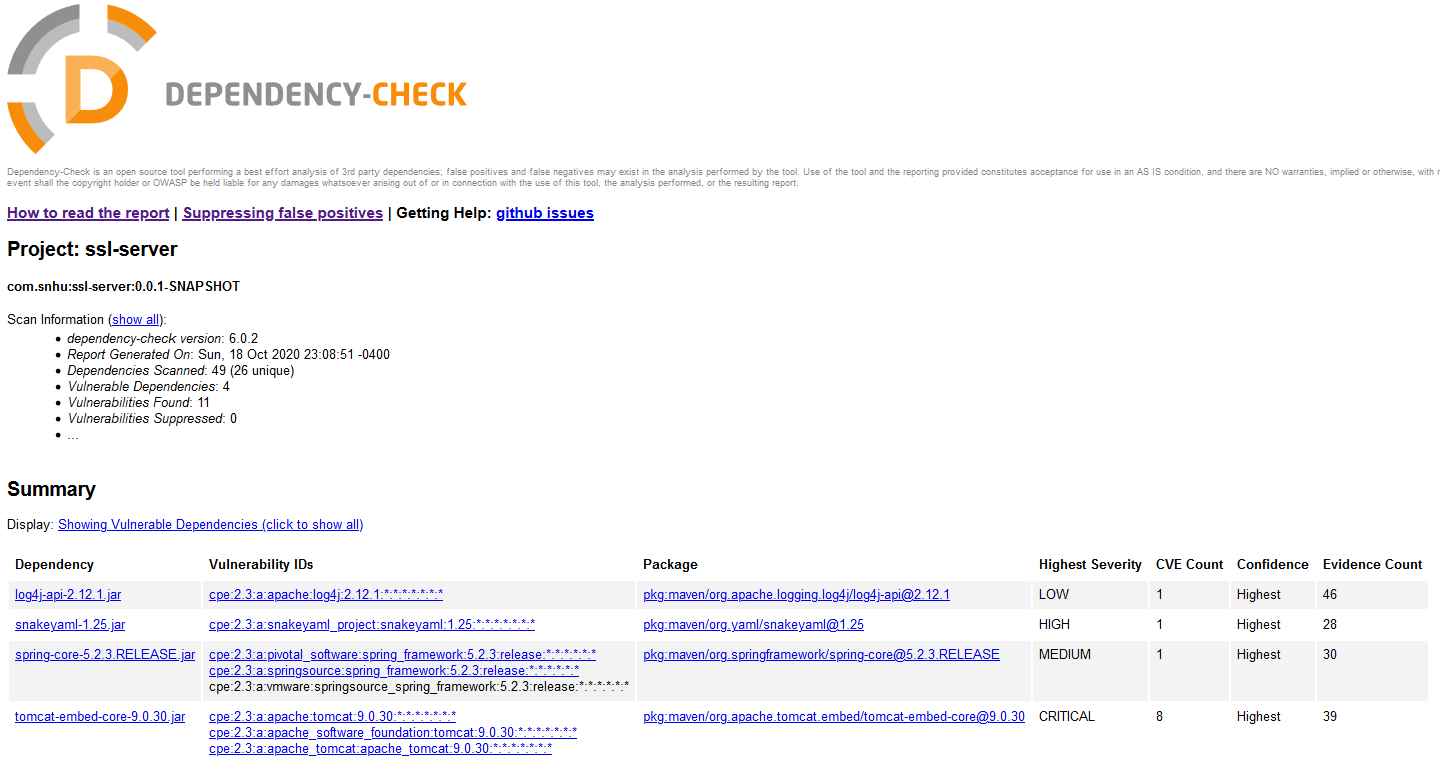
The above screenshot shows the server accessed via HTTPS through a web browser (Firefox). The yellow warning sign is due to the self-signed nature of my certificate.

## 5. Secondary Testing

Complete a secondary static testing of the refactored code using the dependency check tool to ensure code complies with software security enhancements. You only need to focus on the code you have added as part of the refactoring. Complete the dependency check and review the output to ensure you did not introduce additional security vulnerabilities.

* Include the following below:
  + A screenshot of the refactored code executed without errors
  + A screenshot of the dependency check report

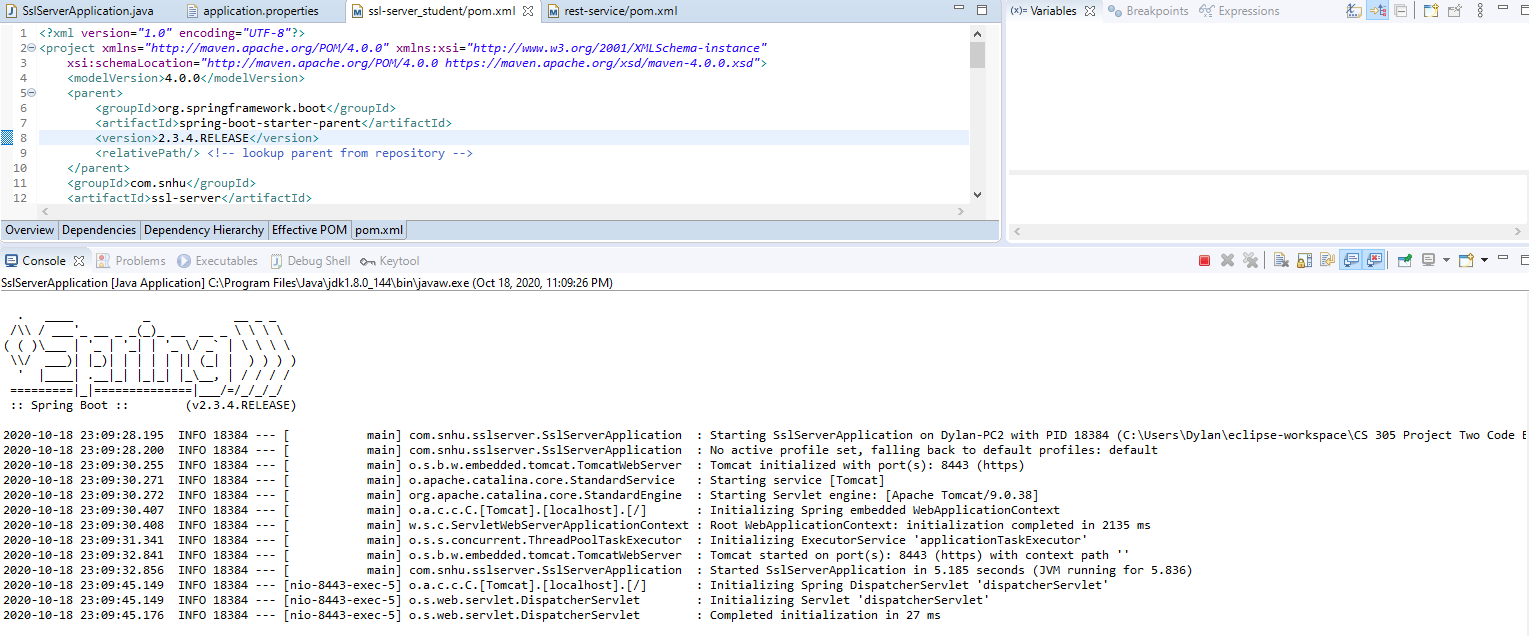
The above screenshot shows the console report of program execution with no errors displayed.

However, performing a Maven dependency check reveals 11 known vulnerabilities in the current code.

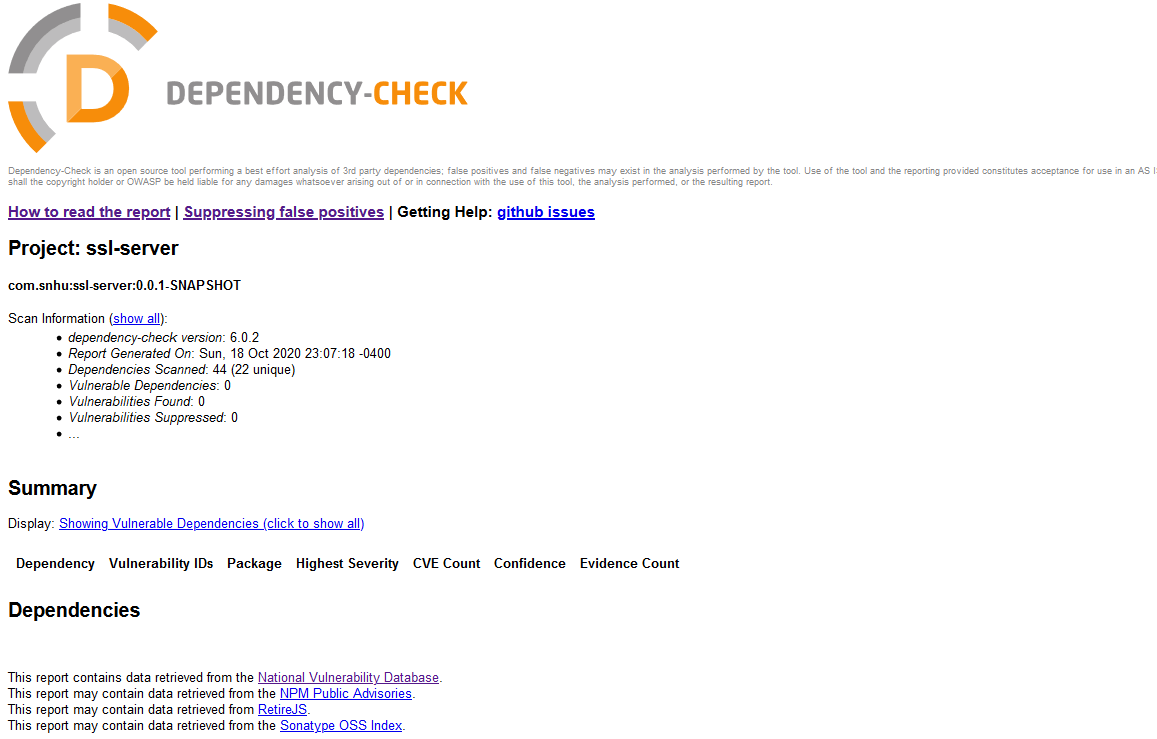
## 6. Functional Testing

Identify syntactical, logical, and security vulnerabilities for the software application by manually reviewing code.

* Complete this functional testing and include a screenshot below of the refactored code executed without errors.



I identified the source of the vulnerabilities as a dependency on an outdated version of the Spring Framework (2.2.4). I updated this dependency to the latest currently-available version (2.3.4), and tested the server again to ensure no errors had been introduced.



## A second dependency check indicates that the code base now suffers from zero known vulnerabilities.

## 7. Summary

Discuss how the code has been refactored and how it complies with security testing protocols. Be sure to address the following:

* Refer to the Vulnerability Assessment Process Flow Diagram and highlight the areas of security that you addressed by refactoring the code.
* Discuss your process for adding layers of security to the software application and the value that security adds to the company’s overall wellbeing.
* Point out best practices for maintaining the current security of the software application to your customer.

The original code base has been refactored to support the checksum validation of static data. This hashing algorithm implementation is suitable for validation of files of all types, using the industry-standard SHA-256 algorithm. This highlights the importance of three areas of security: Crytography, Input Validation, and Code Quality. In the area of Crytography, it is important to select a cryptographic (encryption or hashing) algorithm that is both appropriate for your application, and robustly secure against attackers. In the area of Input Validation, it is important to ensure any untrusted user inputs are secure and parameterized. Checksums are a common method for file and data validation, ensuring that any input data has not been tampered with. The input used in this code base is static and trusted, but future implementations may rely on untrusted user data which should be validated appropriately. Finally, in the area of Code Quality, use of secure coding patterns is necessary when working with cryptography to ensure that sensitive data is not exposed at any point during handling.

The project dependency base has also been refactored to address outstanding CVEs (Common Vulnerabilities and Exposures). The original application had 11 known vulnerabilities, including man-in-the-middle attacks, DoS (denial-of-service), RFD (reflected file download), HTTP Request Smuggling, or even remote code execution in some cases. The source of these CVEs was identified as an outdated Spring Framework dependency (version 2.2.4) which has since been updated to the lateset version (version 2.3.4). This update has addressed all known CVEs, making the code base significantly more secure.

In maintaining the security of the software application, it will be important to remain conscious and observant of dependency vulnerabilities. Secure API Interactions, a key area of security, requires that any third-party libraries and APIs (many of which are open source) should be continuously evaluated for security vulnerabilities, and any detected vulnerabilities should be mitigated as soon as possible. Routinely performing a Maven Dependency Check, either manually or via an automated system, will ensure that any known CVEs are identified as soon as possible, so that mitigation efforts can begin. It will also be important, as the software application grows and begins to accept user input, to ensure that the application is protected from XSS (cross-site-scripting) and injection attacks (SQL or otherwise). This is best handled by input parameterization, HTML entity encoding, and above all, treating all outsider/user input as unsafe and potentially malicious.

**References**

Walker, J., Kounavis, M., Gueron, S., & Graunke, G. (2009). Recent Contributions to Cryptographic Hash Functions. *Intel Technology Journal,* *13*(2), 80-95.

Manico, J., & Detlefsen, A. (2015). *Iron-clad Java: Building secure web applications*. New York, NY: McGraw-Hill Education. Retrieved September October 3, 2020, from https://learning.oreilly.com/library/view/iron-clad-java/9780071835886/

Hoffman, C. (2019, September 30). What Is a Checksum (and Why Should You Care)? Retrieved October 04, 2020, from https://www.howtogeek.com/363735/what-is-a-checksum-and- why-should-you-care/

SHA-256 Algorithm Overview. (2019, September 16). Retrieved October 04, 2020, from https:// www.solarwindsmsp.com/blog/sha-256-encryption

Rhodes, D. (2020, April 27). What's SHA-256? An Overview of the SHA256 Algorithm. Retrieved October 04, 2020, from https://komodoplatform.com/sha-256-algorithm/

Pound, M. (2017, April 11). *SHA: Secure Hashing Algorithm*. Lecture. Retrieved October 4, 2020, from https://www.youtube.com/watch?v=DMtFhACPnTY

Callaghan, P. (2019). Why You Should Use SHA-256 in Evidence Authentication. Retrieved October 04, 2020, from https://blog.pagefreezer.com/sha-256-benefits-evidence- authentication