****

# 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** | **6/22/2024** | **Joe Clancy** | **Initial release** |

## 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

Joe Clancy

6/22/2024

CS305

## Algorithm Cipher

For this application, I recommend using SHA-256 (Secure Hash Algorithm 256-bit) for the encryption algorithm cipher.

Hashing algorithms are one-directional mathematical formulae which attempt to generate a unique value for every possible input. (NCCIC, n.d.). When a message is input to a hash algorithm, the result is an output called a message digest. A message digest ranges in length from 160-512 bits, depending on the algorithm (Oracle, 2024).

The bit level of the algorithm (SHA-256, SHA-384, SHA-512, etc) indicates the number of bits used in the encryption key. Bit levels increase security by increasing the size of the security key. A 512-bit key, for example, is exponentially more difficult to defeat with a brute-force attack than a smaller 384 or 256-bit key. Large bit levels can also make the algorithm meaningfully slower, however, and thus are not always preferable.

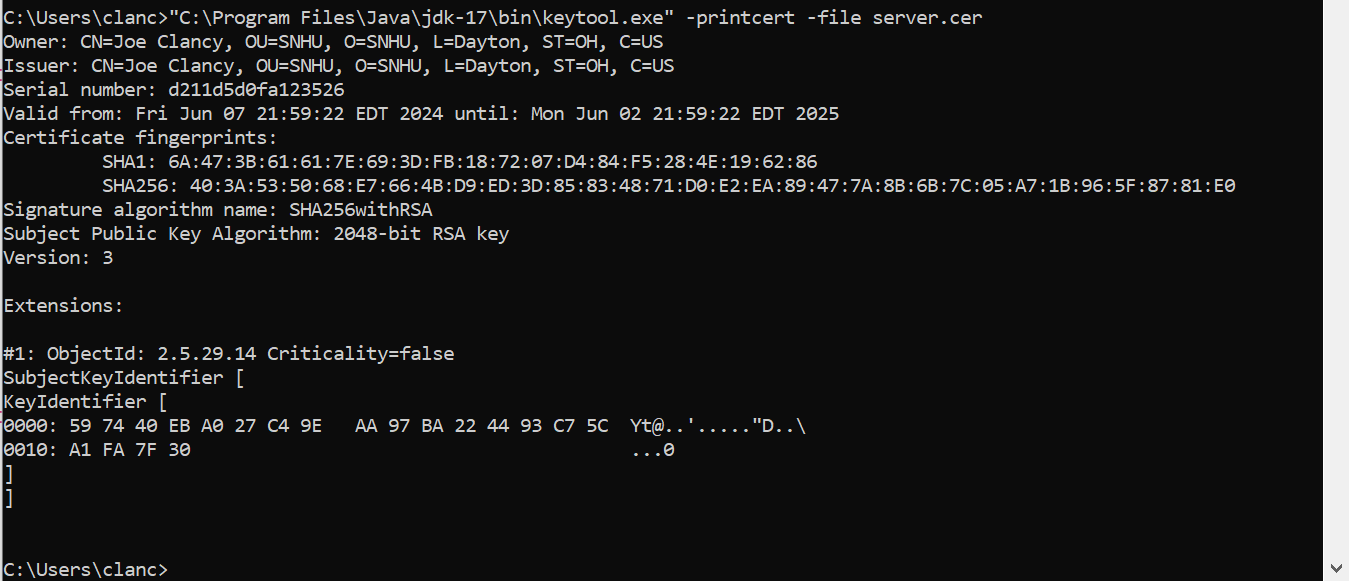
Encryption algorithm ciphers use hash functions to ensure data integrity by creating a unique hash value that represents the data. Any change in the data will result in a different hash value, indicating tampering. In conjunction, random numbers are used to ensure that even if the same plaintext is encrypted multiple times, the ciphertext will be different each time. This speaks to the importance of using secure libraries for random number generation, as predictability may lead to vulnerabilities.

A symmetric key algorithm uses the same key is used for both encryption and decryption. This is efficient for encrypting large amounts of data. Asymmetric algorithms use a pair of public and private keys and are often used to securely exchange a symmetric key. (Manico, J., & Detlefsen, A., 2014)

Encryption algorithms, as a concept, are very old, with examples like the Caesar Cipher dating back thousands of years. Modern encryption algorithms rely on advanced and computationally intensive math to mask information. While these algorithms can theoretically be broken, it would require so much computational power as to be functionally impossible with current technology. Technology continues to improve, however, and the need for algorithms that will remain secure with the advancement and proliferation of quantum computing is an ongoing challenge.

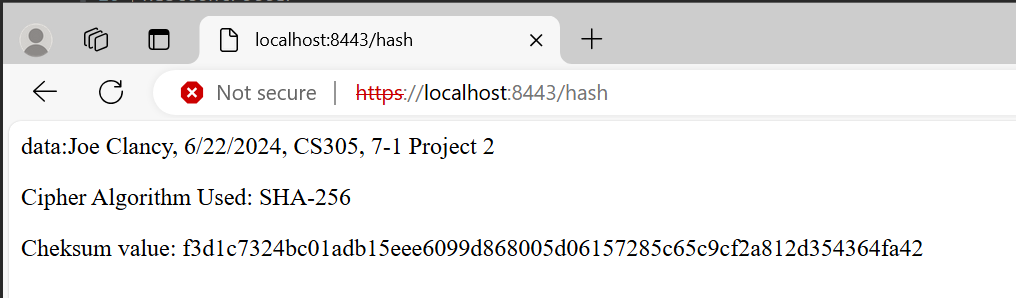
## Certificate Generation

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.

A screenshot of a computer

Description automatically generated

Attempting to access the webpage without security protocols results in a rejected request.

A screenshot of a computer

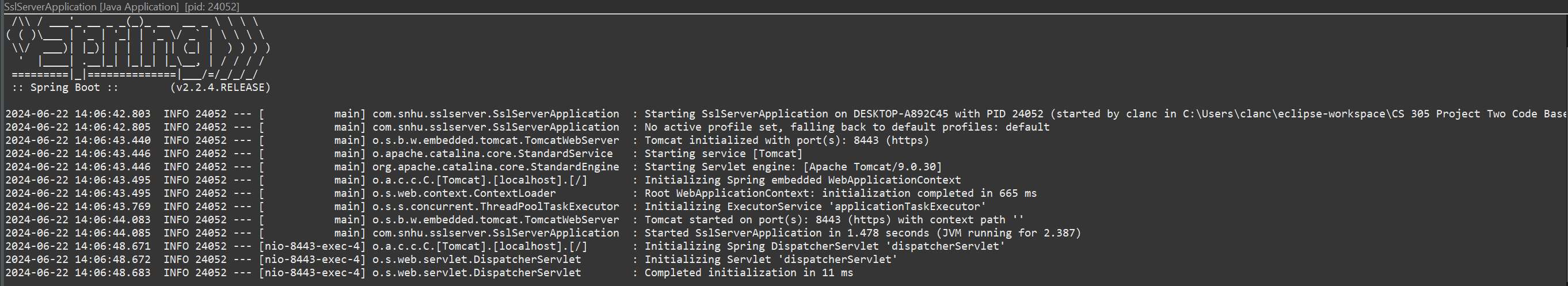
Description automatically generated

Since we are using a self-signed certificate, the browser does not intrinsically trust the certificate. Hence, it identifies the webpage as “Not Secure”. If we were to replace the self-signed certificate with one from a Certificate Authority, this error would resolve, but for development purposes it is acceptable.

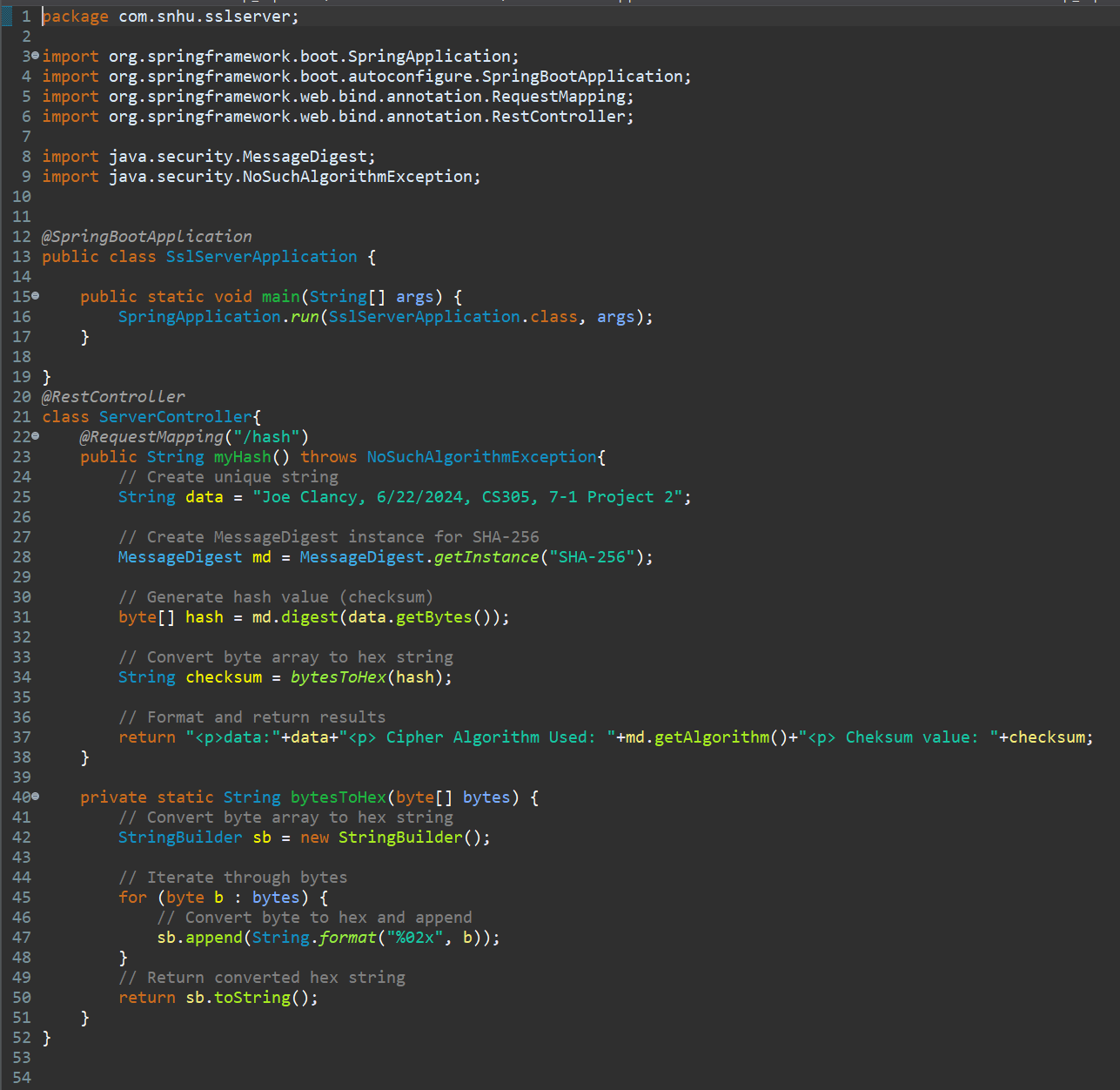
## Secondary Testing

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

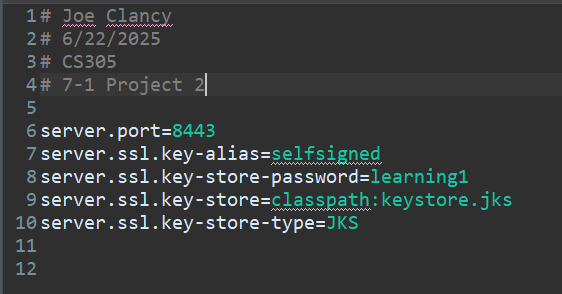
Successful execution:



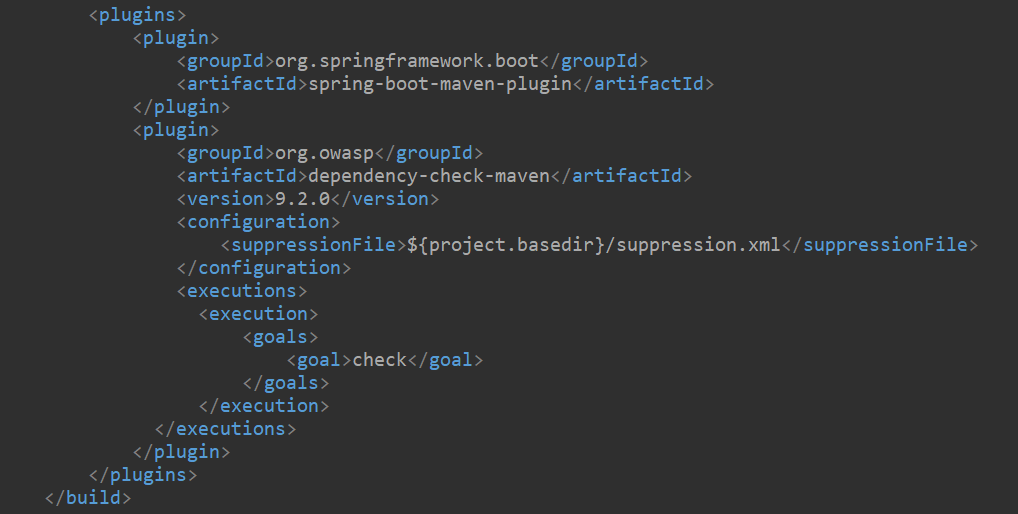
SslServerApplication.java:



application.properties:



pom.xml



suppression.xml

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generatedA screen shot of a computer code

Description automatically generated

Dependency Report

A screenshot of a web page

Description automatically generated

## Functional Testing

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

To confirm that the hashing function and checksum values are working correctly, we can compare the checksum returned by the server controller with one from a third-party tool provided by KeyCDN (KeyCDN Tools, n.d.).

Server provided checksum value for the user message:

A screenshot of a computer

Description automatically generated

Third-party tool checksum for same message:

A screenshot of a computer

Description automatically generated

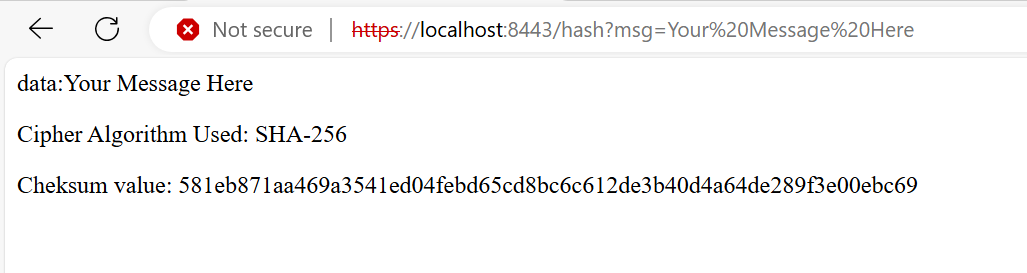
The identical checksum values confirm that the message was received without modification by the server controller.

In order to both send and receive data from the server controller, the /hash request mapping was expanded to accept a string value.

SslServerApplication.java

A screen shot of a computer program

Description automatically generated

This allows the server controller to accept parameters through the URL.

Because we are accepting string parameters, input validation becomes critical and a limitation on string length is added.



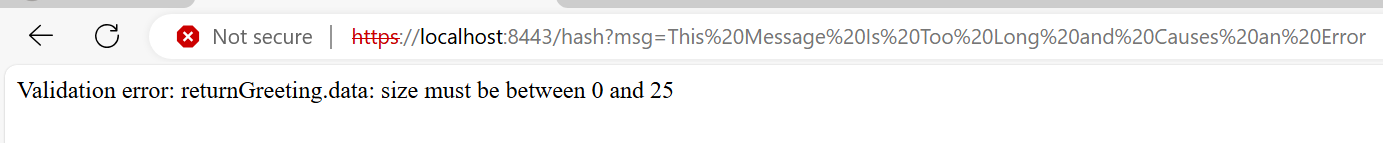
This identifies another problem, however, in that no specific behavior for error handling has been created. To address this, we add a GlobalExceptionHandler() class.

GlobalExceptionHandler.java:

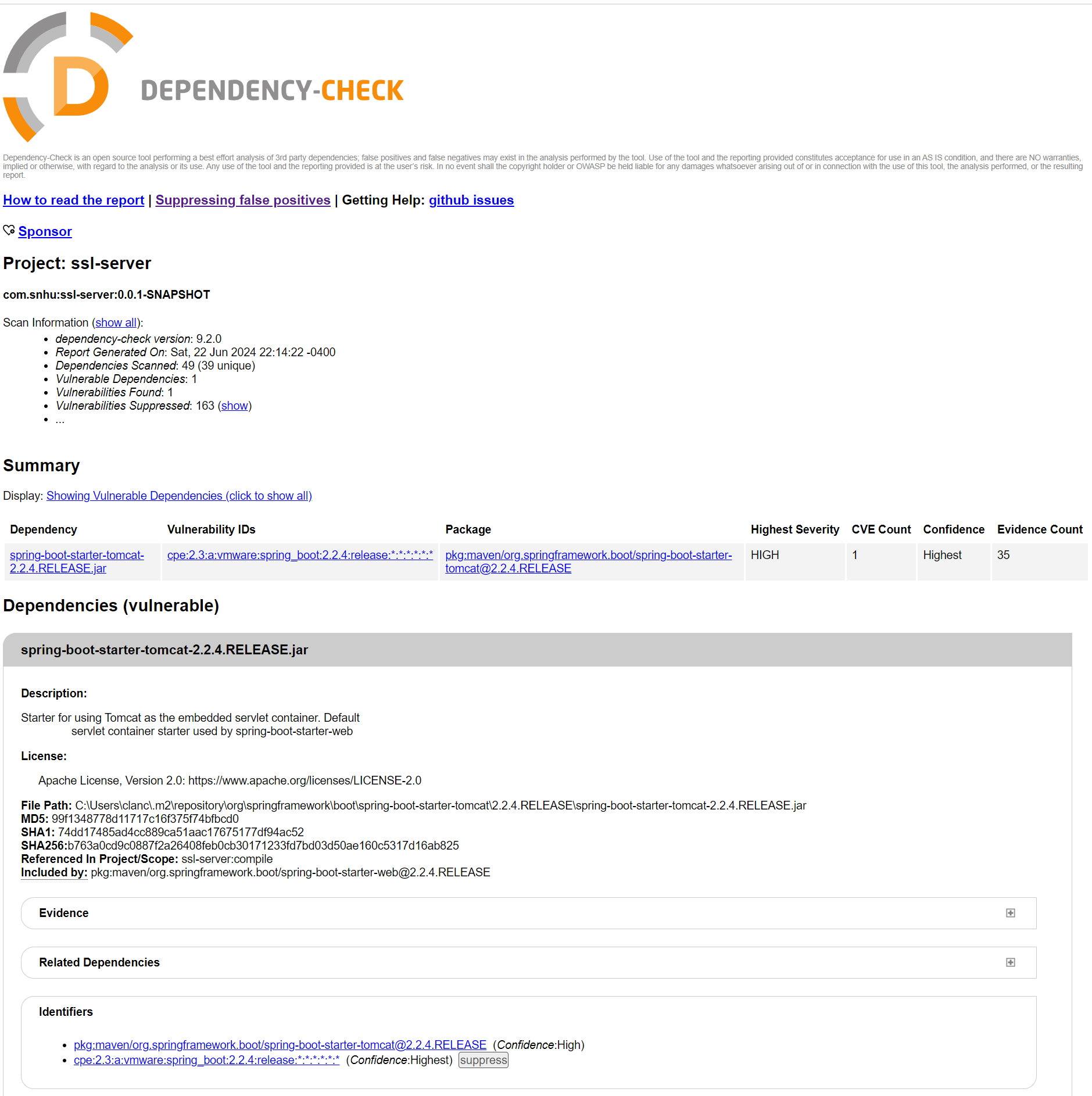
A screen shot of a computer program

Description automatically generated

Now, when invalid inputs are provided to the server controller, the exception is handled intentionally:



Unfortunately, the additional dependencies required to add this functionality have introduced a new vulnerability to the dependency report.



This vulnerability is a false positive, however, because the application does not rely on the affected method. The following suppression is added to the suppression.xml to hide it in future dependency reports:

A black background with white text

Description automatically generated

The dependency check once again shows no vulnerabilities:

A screenshot of a computer

Description automatically generated

## Summary

A yellow square with black text and white arrows

Description automatically generated

The first area of security that I addressed when refactoring the code was cryptography. The application was modified to enforce the use of HTTPS protocols, securing communication between the server controller and the end user. A self-signed certificate was used for development purposes but should be replaced with a certificate from a reputable CA before the application is deployed.

A hashing function was added to hash messages provided to the server controller through the API and provide a checksum for the hash. The SHA-256 algorithm was to create a message digest from the original message string. Users can verify that the server controller received their message without modification using the checksum value and any number of existing tools.

However, this is not particularly useful if the server controller cannot receive inputs from end users, so functionality was added to receive inputs through the API. This, in turn, introduced the need for the second area of security I addressed when refactoring: input validation. A constraint was added to reject string inputs longer than 25 characters. Additional constraints may be considered by the client before the application is deployed.

With the addition of input validation necessarily came constraint violation exceptions, introducing the need for the final area of security I addressed during the refactor: Code Error. The GlobalExceptionHandler() class was added to safely address errors created by bad API requests or other causes.

As I added each layer of functionality to the application, new security considerations presented themselves and the security posture of the entire application needed to be reconsidered. This highlights the difficulty of thoroughly securing applications and the need for thoughtful consideration of the security ramifications of any refactor.

## Industry Standard Best Practices

I used industry standard best practices to maintain the application’s existing security by enforcing the use of HTTPS protocols, thereby safeguarding data in transit between the server controller and the end user. I also provided checksum verification to confirm the integrity of transmitted data and input validation to protect against malicious API requests. I also implemented error handling to obfuscate the internal structure of the system from external view.

Additionally, I used a dependency management tool (Maven) to evaluate known vulnerabilities in both the application’s initial dependencies and the additional dependencies that were added to accommodate the new functionality and security features.

Applying industry standard best practices for secure coding is valuable to the company’s overall well-being in several ways. First, secure coding practices help to prevent data breaches, which may inflict significant financial loss to the company by exposing customer information, financial records, and intellectual property.

Securing customers’ personal data, financial information, and transaction history is also critical to maintaining the company’s reputation and the trusted relationship with its clients that it relies on to be successful. Lax software security has the potential to damage that trust and thereby the business.

Additionally, as a financial institution, the company is legally obligated by many jurisdictions to safeguard the sensitive information it handles. Regulations like the Payment Card Industry Data Security Standard (PCI-DSS) and the Gramm-Leach-Bliley Act explicitly require the company to implement software security measures including secure networks, strong access controls, and vulnerability management. (Federal Trade Commission, n.d.).

**References**

Federal Trade Commission. (n.d.). *Gramm-Leach-Bliley Act.* [*https://www.ftc.gov/business-guidance/privacy-security/gramm-leach-bliley-act*](https://www.ftc.gov/business-guidance/privacy-security/gramm-leach-bliley-act)

KeyCDN Tools (n.d.) *SHA256 Generator*. KeyCDN. <https://tools.keycdn.com/sha256-online-generator>

Manico, J., & Detlefsen, A. (2014). *Iron-Clad Java: Building Secure Web Applications*. McGraw-Hill Education.

National Cybersecurity and Communications Integration Center (NCCIC) (n.d.) *File Hashing Fact Sheet*. <https://www.cisa.gov/sites/default/files/FactSheets/NCCIC%20ICS_Factsheet_File_Hashing_S508C.pdf>

Oracle. (2024, April 1). *JDK 22 Documentation*. <https://docs.oracle.com/javase/9/docs/specs/security/standard-names.htm>

OWASP. (15 May, 2024). *Suppressing False Positives*. <https://jeremylong.github.io/DependencyCheck/general/suppression.html>