

# Practices for Secure Software Report

**Table of Contents**

[**Document Revision History 3**](#_30j0zll)

[**Client 3**](#_1fob9te)

[**Instructions 3**](#_3znysh7)

[**Developer 4**](#_2et92p0)

[**1. Algorithm Cipher 4**](#_tyjcwt)

[**2. Certificate Generation 4**](#_3dy6vkm)

[**3. Deploy Cipher 4**](#_1t3h5sf)

[**4. Secure Communications 4**](#_4d34og8)

[**5. Secondary Testing 4**](#_2s8eyo1)

[**6. Functional Testing 4**](#_17dp8vu)

[**7. Summary 4**](#_3rdcrjn)

[**8. Industry Standard Best Practices 4**](#_26in1rg)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **2023 OCT 15** | **Dennis James Stelmach** | **Completion of Document** |

## Client



## Developer

Dennis James Stelmach

## Algorithm Cipher

* RSA for SSL Certificate:  
   RSA (Rivest-Shamir-Adleman) is a widely utilized asymmetric encryption algorithm that plays a vital role in modern cryptography. It operates based on the mathematical properties of large prime numbers and is renowned for its security and versatility in various cryptographic applications.  
   In RSA encryption, a pair of keys is generated: a public key used for encryption and a corresponding private key used for decryption. The security of RSA encryption is rooted in the practical difficulty of factoring the product of two large prime numbers, a task that becomes increasingly complex with larger prime numbers.  
   The process involves modular exponentiation and specific mathematical operations on large numbers, ensuring the confidentiality and integrity of the transmitted data. RSA is widely used in digital signatures, certificates, secure communication protocols like HTTPS, and various other security applications where data privacy and authenticity are paramount.  
   Key sizes in RSA, often measured in bits, significantly impact the algorithm's security. Common key sizes include 2048 bits and 4096 bits. As the key size increases, the computational effort required to break the encryption through brute-force methods escalates, making RSA a robust choice for secure data transmission.
* SHA-256 for hashing data:  
   SHA-256 (Secure Hash Algorithm 256-bit) is an appropriate hashing algorithm cipher that avoids collisions. It is part of the SHA-2 (Secure Hash Algorithm 2) family of cryptographic hash functions.  
   SHA-256 is a cryptographic hash function that takes an input (or 'message') and produces a fixed-size output hash value of 256 bits (32 bytes). It is designed using the Merkle-Damgard construction, involving several rounds of processing to transform the input data into the final hash value. SHA-256 is widely used in various security applications and protocols, including digital signatures, certificates, and integrity verification of data.  
   SHA-256 is designed to be collision-resistant, meaning that it is computationally infeasible to find two different inputs that produce the same hash value. This resistance to collisions is achieved through the design of the algorithm, including its bitwise operations and the avalanche effect, where a small change in the input results in a significantly different hash value.

## Certificate Generation

Keystore and certificate generation

## 

Generation, export, and display command:

PASSWORD=SUPER\_SECRET\_PASSWORD; \

echo "Gen:"; \

keytool -genkey -keyalg RSA -alias selfsigned -keypass "$PASSWORD" -keystore keystore.jks -storepass "$PASSWORD" -validity 360 -keysize 2048; \

echo -e "\n-----\n"; \

echo "Export:"; \

keytool -export -alias selfsigned -storepass "$PASSWORD" -file server.cer -keystore keystore.jks; \

echo -e "\n-----\n"; \

echo "Print:"; \

keytool -printcert -file server.cer;

## Deploy Cipher

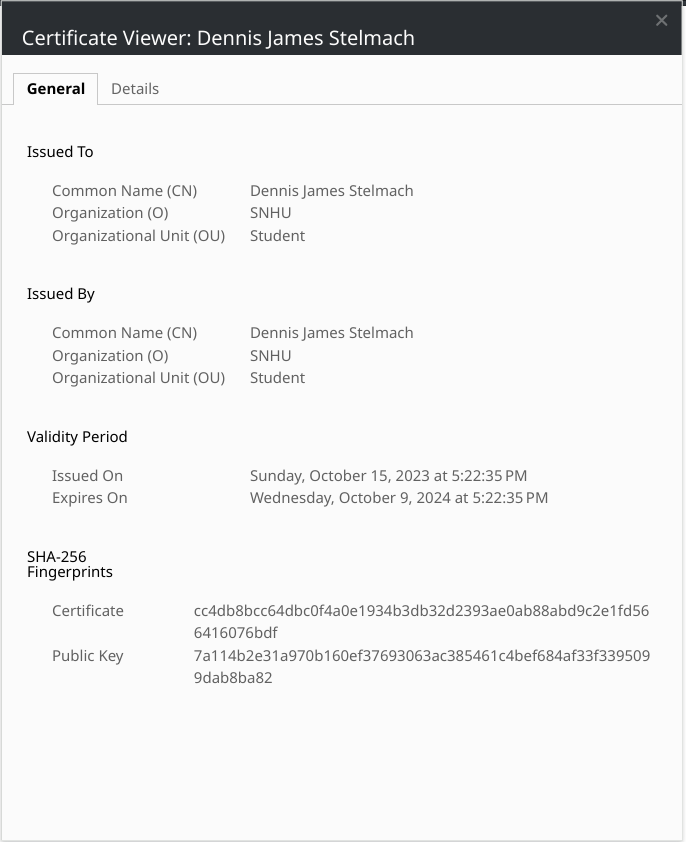
## 

## Secure Communications

Here we see that HTTPS is used, but modern browsers flag self-signed certificates as “Not Secure”:

## 

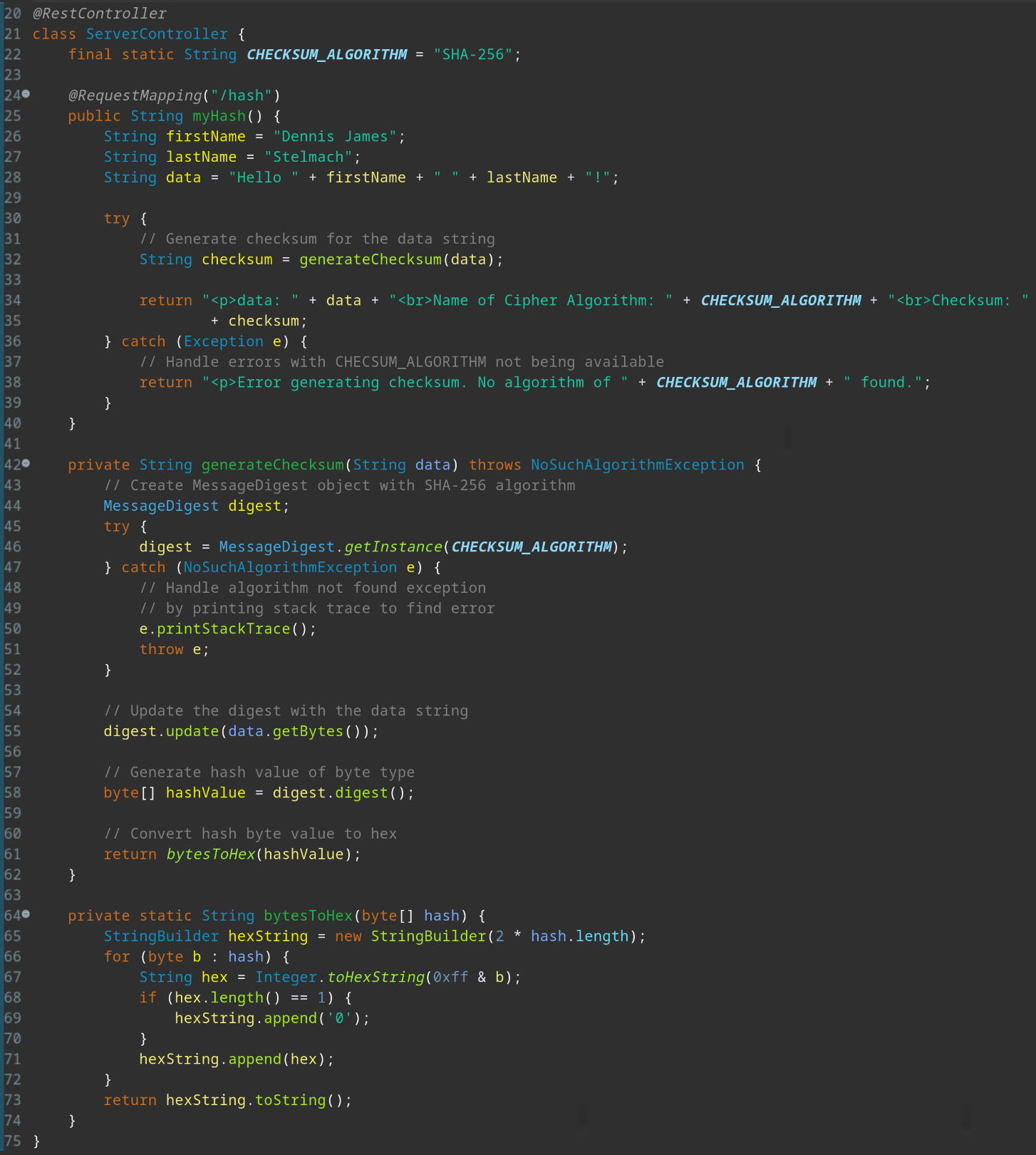
Here we see the SSL certificate is working properly:



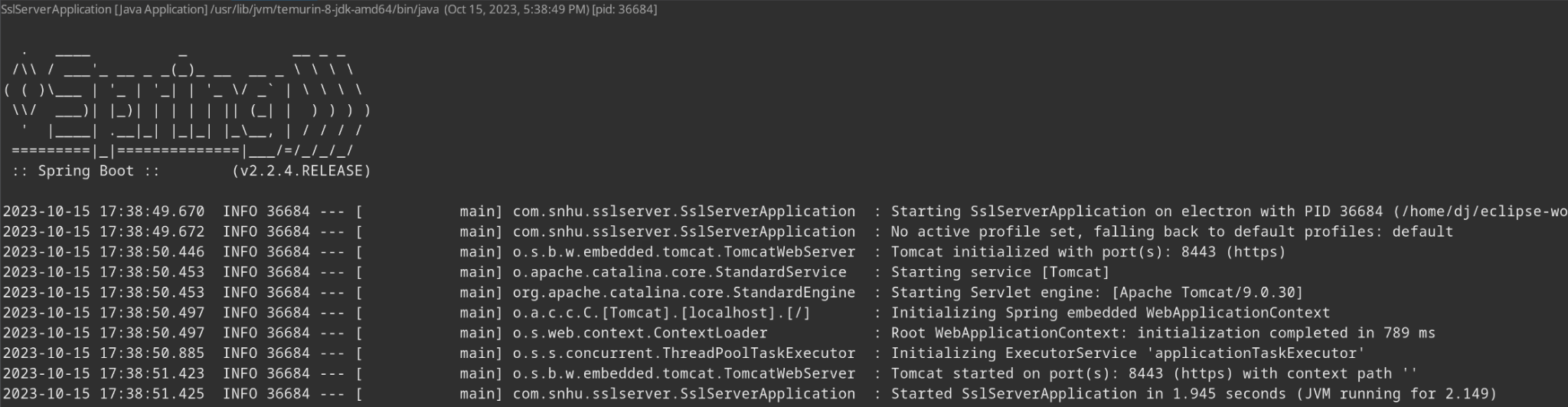
## Secondary Testing

Refactored codebase is also available as a ZIP attachment to this document.

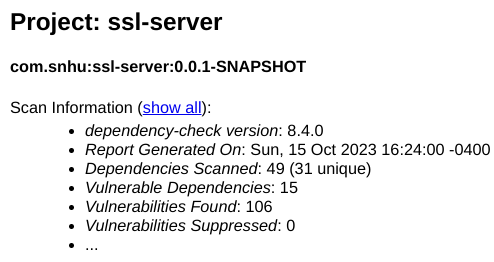
Refactored code:



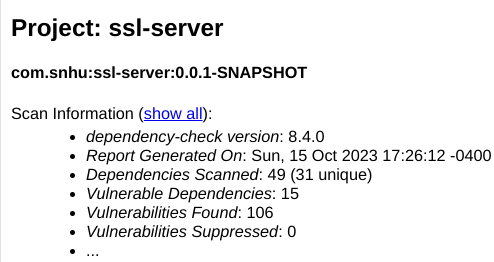
Code executes without errors:



dependency-check report before refactor:



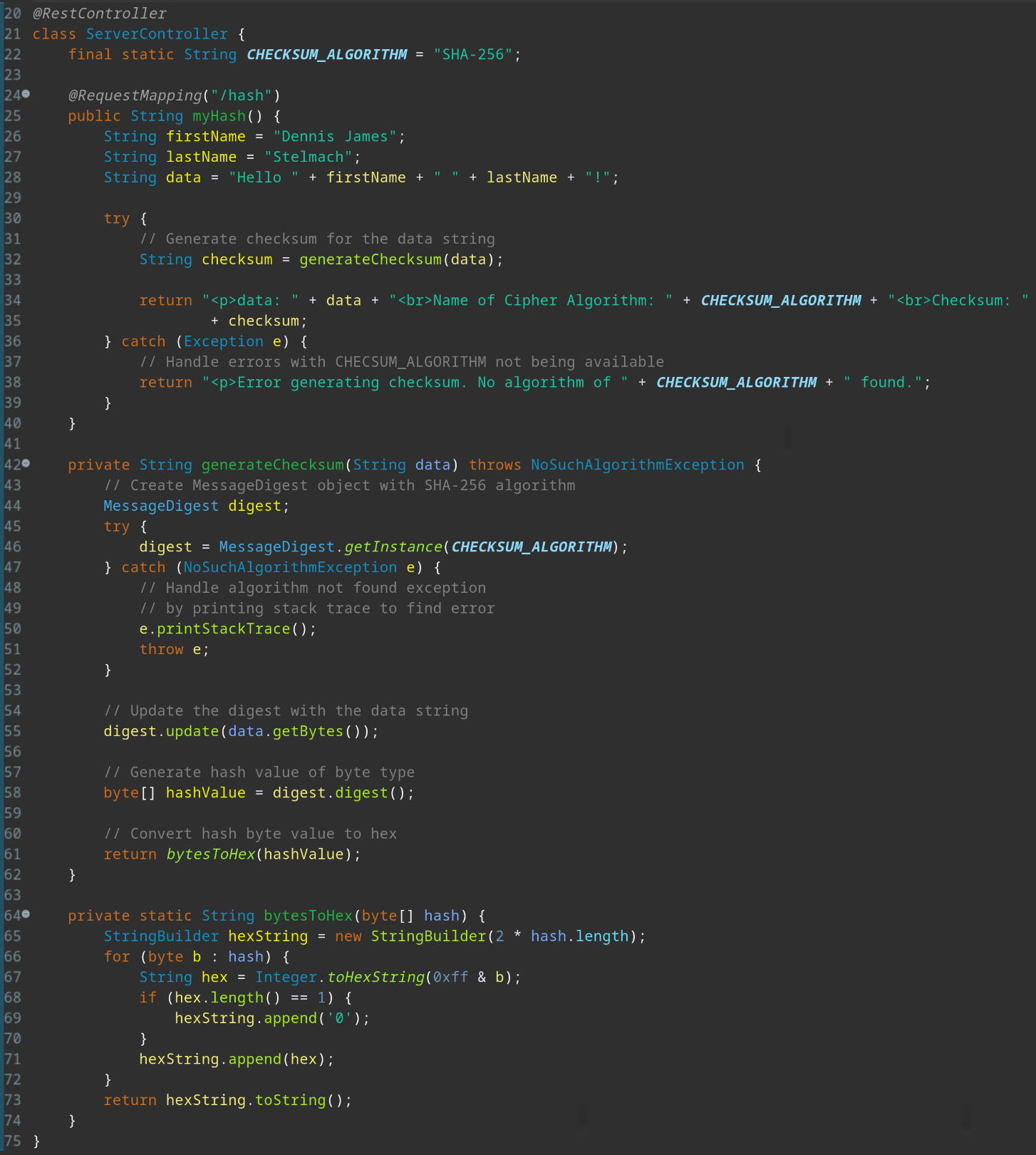
dependency-check report after refactor:



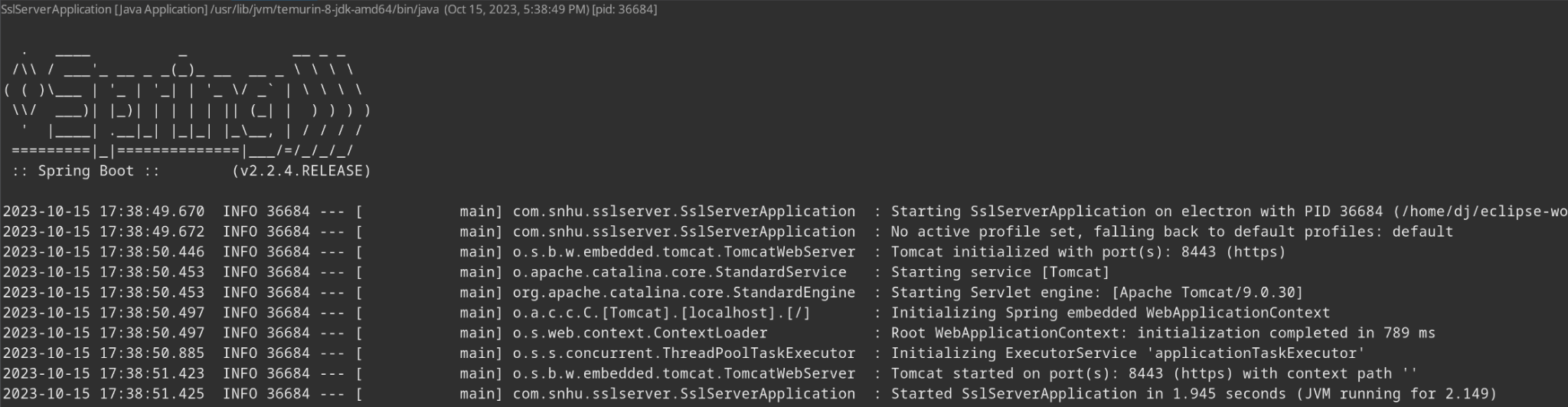
Report is unchanged by refactor.

## Functional Testing

Refactored code:



Code executes without errors:



## Summary

The ServerController class addresses several security aspects:

1. Input Validation: The data used for generating the checksum is hardcoded within the method. There's no direct user input here, which minimizes the risk of injection attacks.
2. Algorithm Selection: The class uses the SHA-256 algorithm for generating checksums, a widely accepted cryptographic hash function known for its security. It ensures data integrity and authenticity.
3. Error Handling: The class includes error handling mechanisms. If the specified algorithm (SHA-256) is not available, it handles the NoSuchAlgorithmException gracefully, preventing unexpected failures and enhancing the application's resilience.
4. Secure Communication: While not directly tied to the ServerController class, HTTPS is configured with the use of a self-signed certificate.

The following layers of security can be added

1. Secure Algorithm Usage: The use of SHA-256 demonstrates a commitment to using strong cryptographic algorithms, ensuring the integrity of the data.
2. Input Sanitization: Although not directly applicable in this context due to the hardcoded data, it’s crucial to validate and sanitize any user inputs in a real-world scenario. This step prevents common vulnerabilities like SQL injection or Cross-Site Scripting (XSS) attacks.
3. Secure Communication: While a self-signed certificate is used in this context, a certificate issued by a Certificate Authority will allow foreign browsers to communicate with the app without triggering a “Not Secure” error due to the self-signing of the certificate.
4. Authentication and Authorization: If this application extends its functionality to handle user-specific data or sensitive operations, implementing authentication (ensuring users are who they claim to be) and authorization (defining what authenticated users are allowed to do) mechanisms is essential.
5. Regular Updates: Keeping all software components, including Spring Boot and related libraries, up-to-date is critical. Security vulnerabilities are regularly discovered, and updates often include patches to address these vulnerabilities.
6. Logging and Monitoring: Implementing robust logging mechanisms allows for the detection of any suspicious activities. Coupled with monitoring tools, these can provide insights into potential security threats.
7. Static Code Analysis and Code Reviews: Regularly scanning the codebase for vulnerabilities using static code analysis tools and conducting code reviews can identify security issues early in the development process.
8. Dependency Scanning: Regularly check for vulnerabilities in third-party dependencies. Using tools like OWASP Dependency-Check can highlight if any used libraries have known security issues.
9. Compliance with Security Standards: Ensure the application complies with industry-standard security practices and, if applicable, regulatory standards like GDPR or HIPAA.

## Industry Standard Best Practices

The provided code demonstrates adherence to several industry-standard best practices to maintain security:

1. Secure Algorithm Usage: The code employs SHA-256, a widely accepted cryptographic hash function, ensuring data integrity. Using well-vetted algorithms is a fundamental best practice to prevent cryptographic vulnerabilities.
2. Error Handling: The code includes proper error handling for algorithm unavailability, preventing unexpected failures. Robust error handling ensures graceful degradation, enhancing the application's stability and user experience.
3. Code Modularity: The code exhibits modularity by separating concerns into distinct methods. Modularity enhances readability, maintainability, and reduces the likelihood of introducing security vulnerabilities during future modifications.
4. Input Validation: Although not applicable in the given scenario due to hardcoded data, input validation is crucial. Sanitizing and validating user inputs prevent common vulnerabilities like SQL injection and Cross-Site Scripting (XSS) attacks.
5. Avoiding Hardcoded Secrets: While the code doesn't involve user authentication, avoiding hardcoded secrets and sensitive information within the source code is a standard practice. Secrets should be stored securely, often in environment variables or secure vaults, reducing the risk of exposure.
6. Use of Frameworks: Leveraging established frameworks like Spring Boot ensures that many security concerns are handled transparently. Spring Security, for instance, provides robust authentication and authorization mechanisms.

There are several beneficial values to applying industry standard best practices in our codebase:

1. Mitigating Known Vulnerabilities: Following industry best practices helps mitigate known vulnerabilities. By adhering to established standards, the codebase is less likely to contain common pitfalls, reducing the likelihood of exploitation.
2. Compliance and Trust: Adhering to best practices ensures compliance with industry standards and regulations. For instance, GDPR mandates secure coding practices to protect user data. Compliance not only avoids legal repercussions but also builds trust with users and clients, enhancing the company's reputation.
3. Cost-Efficiency: Fixing security issues post-development is far more costly than implementing security from the start. Secure coding practices, when integrated early, prevent the need for extensive rework, saving both time and resources.
4. Customer Confidence: Secure applications inspire confidence in customers. Users are more likely to trust and continue using services that are secure and protect their data, leading to customer loyalty and positive reviews.
5. Preventing Data Breaches: Security breaches can be catastrophic. Best practices act as a robust defense, reducing the risk of data breaches that could lead to financial loss, legal actions, and reputational damage.
6. Competitive Advantage: Security-conscious companies have a competitive edge. In a world where data breaches are common, a company known for its secure practices attracts partners, clients, and customers who prioritize data protection.
7. Long-Term Sustainability: Secure coding practices ensure the longevity of software. As threats evolve, applications following best practices are easier to adapt and secure against emerging vulnerabilities, ensuring the software's sustainability over time.