****

# 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** | **[Date]** | **[Your Name]** |  |

## 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

Harrison Bergeron

## 1. Algorithm Cipher

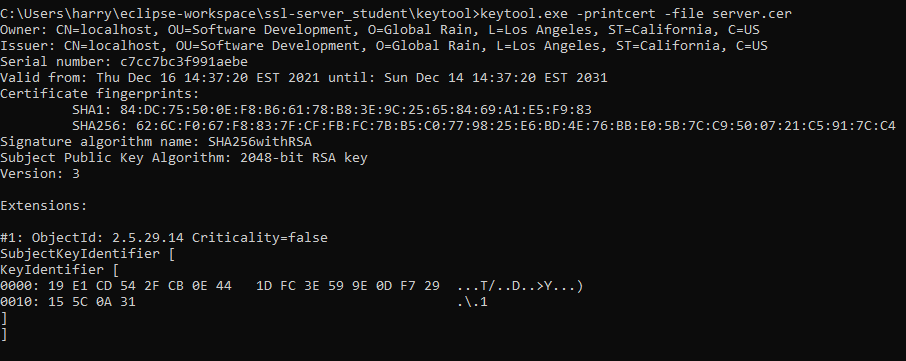
Determine an appropriate encryption algorithm cipher to deploy given the security vulnerabilities, justifying your reasoning. Be sure to address the following:

Out of all the encryption ciphers available for Artemis Financial, AES 256 might be the best option for checksum verification when combined with SHA 256 to generate checksums. AES has been officially established by the National Institute of Standards and Technology as a secure encryption algorithm. Since data may be arbitrary in size, it would be useful to choose a cipher designed for the task. Security attacks like brute force attacks need to be protected against because the data will exist in an encrypted state for a short period of time, which means attackers may try random key combinations to decrypt the data. A 256-bit AES encryption algorithm could make brute force attacks virtually impossible via modern computing capabilities, since iterating through all possible 256-bit keys would take inconceivable amounts of time. One downside to using AES for encryption is that it uses symmetric keys, meaning that “the same key is used for both encrypting and decrypting the data” (Manico & Detlefsen, 2015). Asymmetric encryption algorithms use a public key for encryption which allows anyone to encrypt data with the key, but use a private key for decryption, which is the case for RSA. An algorithm like RSA is by itself not efficient for encryption of large amounts of data, however, and requires that the size of the message being encrypted is smaller than the key size. Using the AES cipher for long term encryption might pose the risk that keys are revealed accidentally given enough time, so data would need to occasionally be encrypted with a new key, be encrypted more than once, or have its key be encrypted with another algorithm like RSA while storing these additional private keys elsewhere. Choosing anything more secure than AES might be overkill given current limitations of computing power, especially when considering the time required to encrypt and decrypt could increase greatly. DES and 3DES are not ideal for encryption of files and have small key sizes by default: “3DES provides no more than 112 bits of security” (Manico & Detlefsen, 2015). Overall, AES is a strong choice for the encryption of Artemis Financial’s data in the long term because of its time and space efficiency for encryption and decryption as well as a secure 256-bit key size.

## 2. Certificate Generation

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

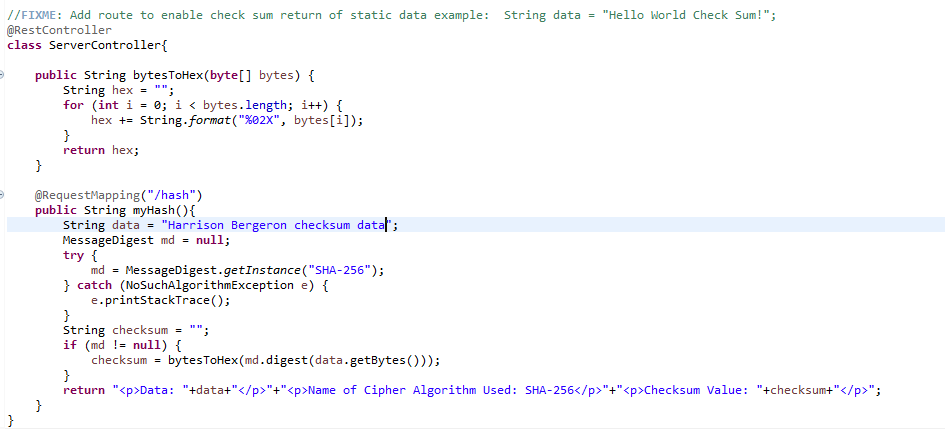
* To demonstrate that the keys were effectively generated, export your certificates (CER file) and submit a screenshot of the CER file below.



## 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.

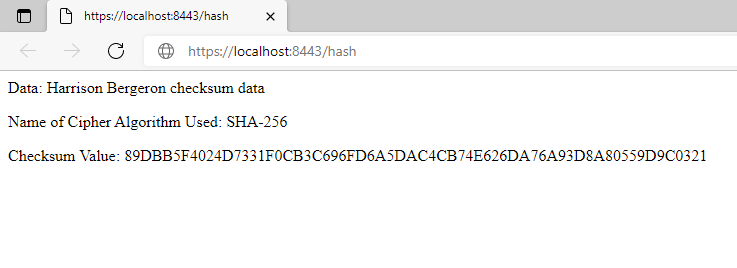
* Insert a screenshot below of the checksum verification. The screenshot must show your name and a unique data string that has been created.



## 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.

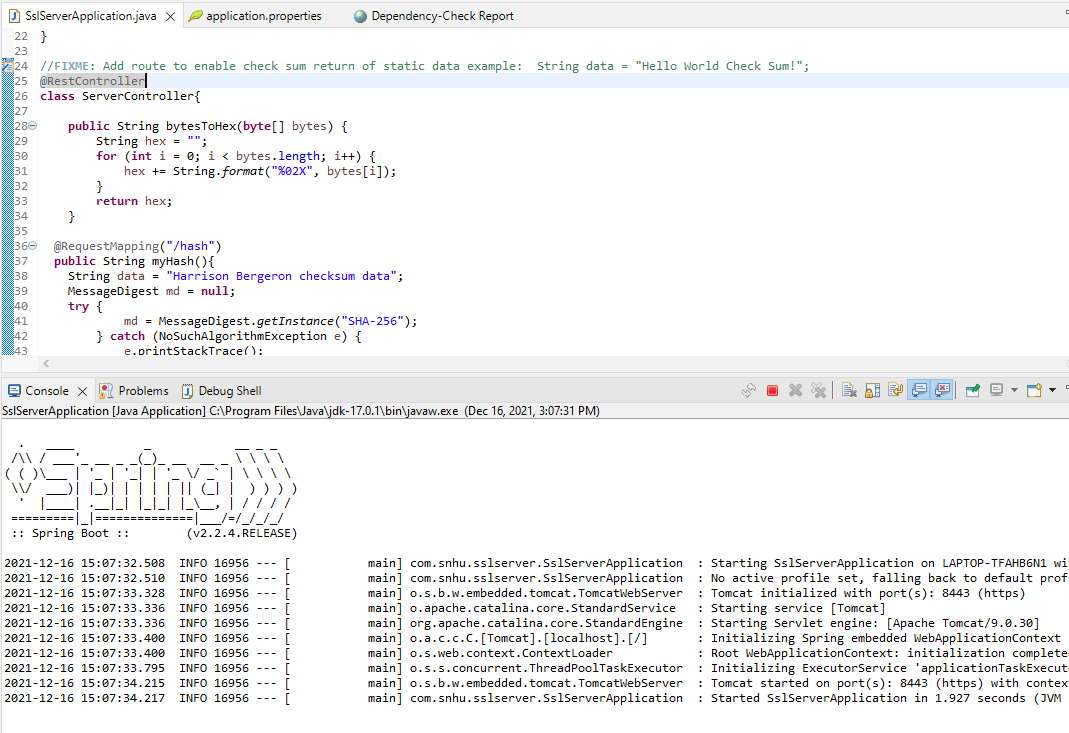
* Insert a screenshot below of the web browser that shows a secure webpage.



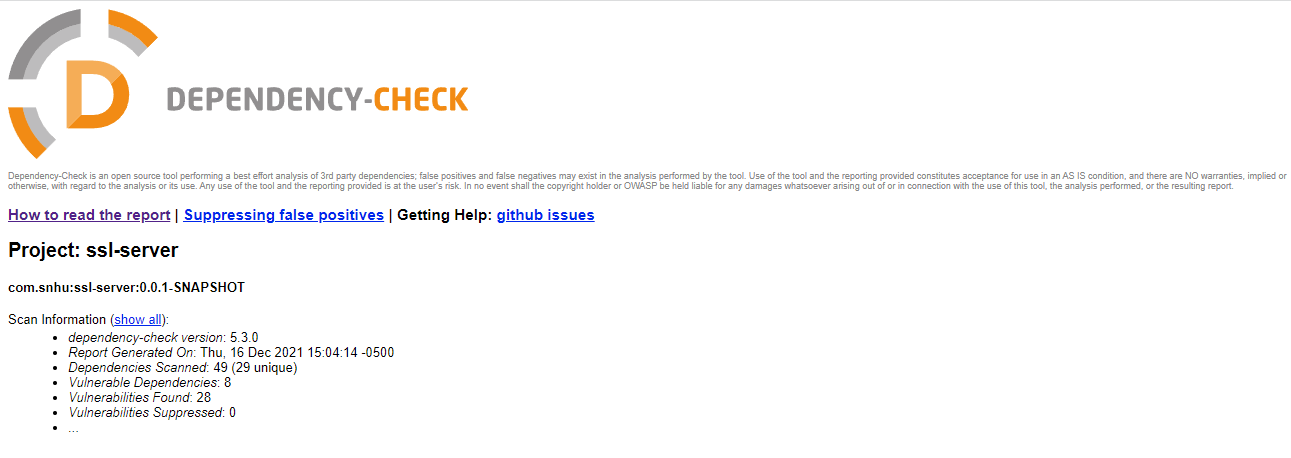
## 5. Secondary Testing

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.

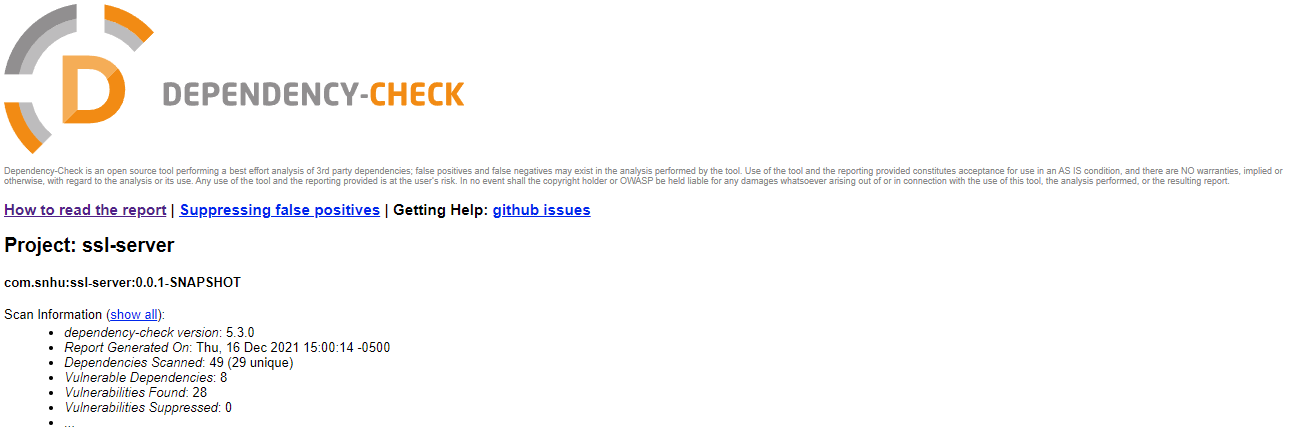
Executing the refactored code causes no new errors or security vulnerabilities to be introduced. The pre-existing security vulnerabilities introduced in the base code stayed the same before and after refactoring.



Dependency check before code refactor:



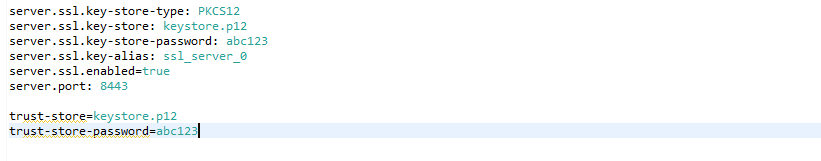
Dependency check after code refactor (java.security and org.springframework.web dependencies in use):

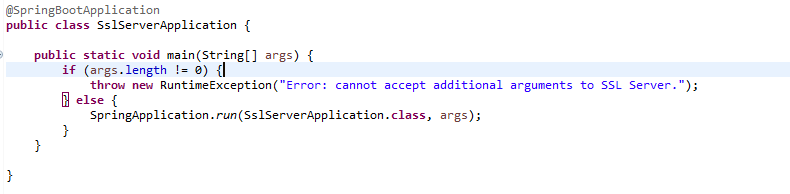


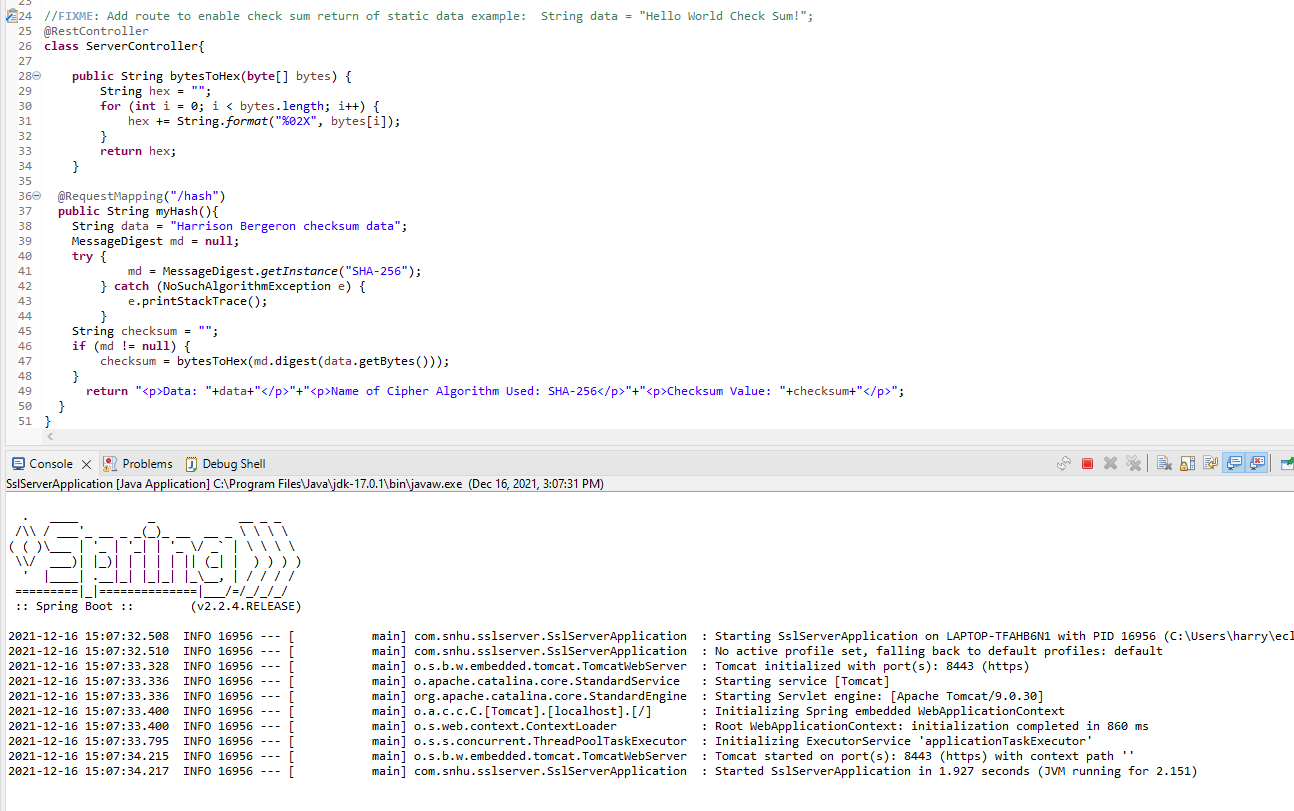
## 6. Functional Testing

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.







## 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.

In reviewing the code, I used these areas of the Vulnerability Assessment Diagram to determine which areas of the code needed to be fixed: Input Validation, Cryptography, Code Error, and Code Quality. Firstly, accepting any input for the SSL Server Application program in the form of an argument list is a potential security risk, since untrusted individuals running the program might insert any arbitrary data into the arguments that could be malicious. The refactored code involves checking to make sure that the command line argument list is empty or throwing an error otherwise, since no arguments are required to test this server. Next, I used SHA 256 to generate a checksum for Artemis Financial to allow for file verification. Refactoring the code to include this checksum is a requirement to allow Artemis Financial to transfer data without potential middle man interference. A checksum can guarantee that the data that was sent was not altered. Code Error was addressed by implementing proper error checking for obtaining an instance of the Java MessageDigest class, since in the event that a NoSuchAlgorithmException was thrown, the program would unexpectedly crash. By handling the event, security risk is minimized and the flow of the program behaves as expected. Code Quality was address by checking that the returned MessageDigest object was not null, so an exception would not be thrown if an instance could not be created. Additionally, the application.properties file was modified to use HTTPS with a self-signed certificate to reduce the risk of potential middle man attacks where another host could impersonate the original web application. Artemis Financial no longer has to worry about certain dangerous threats regarding intercepted files now that SSL and checksum verification are enabled.

References

Manico, J., & Detlefsen, A. (2015). *Iron-clad java: Building secure web applications ; best practices for secure Java Web Application Development*. Oracle Press.