****

Gino Murin

# 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** | **02/14/2022** | **Gino Murin** | **Software Security Assessment** |

## Client



## Developer

Gino Murin

## 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 highly secure nature of the data being used by Artemis Financial, I would recommend the AES encryption algorithm. It is highly regarded as one of the most secure encryption algorithms in existence, and when paired with a SHA-256 hash function, can be extremely difficult to crack.

SHA, which stands for Secure Hashing Algorithm is a family of hashing algorithms known for their ability to avoid collisions by producing unique, randomized outputs. In SHA256, there are 2256 or 1.1579209e77 possible hashes to be generated within SHA256. Even with the use of modern sophisticated computer hardware, it is extremely difficult to crack SHA256.

AES uses what is known as symmetric encryption, which is a method that utilizes only a single key. While this method is both secure and convenient, its weakness is that the key must be shared for other parties to decrypt. On the other hand, SHA-256 uses what is known as non-symmetric encryption. In non-symmetric encryption, each entity has two of their very own keys, a public key and a private key. To send an encrypted message to a target entity, that entity’s public key must be obtained to encrypt the message. Once encrypted, only the receiving entity will be able to decrypt the message using their private key. This method has the advantage of having one key that is never shared.

## 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.

Graphical user interface, text, application

Description automatically generatedText

Description automatically generated

## 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.

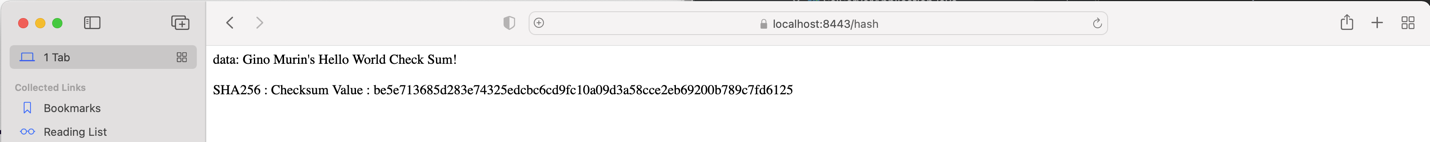


Figure - Output of Code

Text

Description automatically generated

Figure - Refactored Code

## 

## 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.

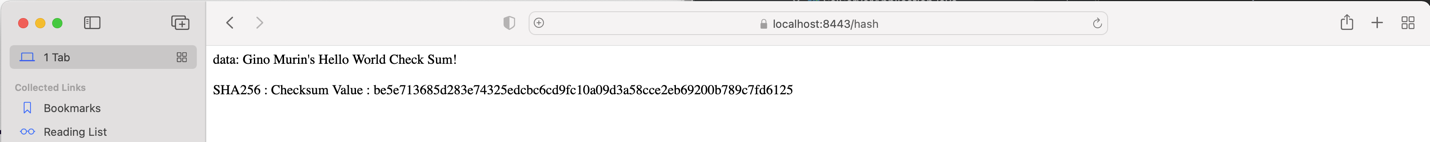


Figure - Output of code. The lock icon indicates the presence of a secure https connection with an ssl certificate

## 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

Text

Description automatically generated

Figure - A Successful Maven Build Installing Dependency Check

A picture containing text, indoor, monitor, electronics

Description automatically generated

Figure - Dependency Check Report HTML File

## 

## 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.

A screenshot of a computer

Description automatically generated with medium confidence

Figure -Refactored Code

## 

## 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 primary areas of security addressed in the refactoring of the code were API’s, Cryptography, Code Quality, and Encapsulation. These were all necessary changes that enhanced the functionality of the application, while enhancing the security and preventing potential threats.

In regard to API’s, we added a REST controller route at ‘/hash’. Routing to this address in the browser allows our back-end code to perform its necessary functionality, which brings us to the topic of cryptography. Our program encrypts a hard coded string. This could be easily adapted to encrypt other sensitive data as we see fit in later iterations of the application. Within our controller, we make use of encapsulation by relegating functionality to their respective methods. For example, our ‘/hash’ route’s myHash() function calls on our bytesToHex() function to generate and return a checksum value to be output in the browser. And of course, code quality is obtained through rigorous testing and refactoring of the code to ensure we achieve the intended functionality and do not have any errors.

Adding these additional layers of security to the application are in the best interest of all parties involved. Both the customer and their clients will be better protected from attacks and exploits that could otherwise leak sensitive data or impair the functionality of the customer’s application.

While secure design and implementation are important first steps on the path to a secure system, it is also important to always follow security best practices. The customer and their client alike should be aware of the importance of secure passwords and never sharing any sensitive information with unknown parties.