****

# CS 305 Project Two

**Practices for Secure Software Report**

Table of Contents

[CS 305 Project Two 1](#_Toc111603424)

[Document Revision History 3](#_Toc111603425)

[Client 3](#_Toc111603426)

[Instructions 3](#_Toc111603427)

[Developer 4](#_Toc111603428)

[1. Algorithm Cipher 4](#_Toc111603429)

[2. Certificate Generation 6](#_Toc111603430)

[3. Deploy Cipher 8](#_Toc111603431)

[4. Secure Communications 9](#_Toc111603432)

[5. Secondary Testing 11](#_Toc111603433)

[6. Functional Testing 13](#_Toc111603434)

[7. Summary 14](#_Toc111603435)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **08/13/2022** | **Elorha Newcomb** | **New Project** |

## 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

Elorha Newcomb

## 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.

The purpose of encryption ciphers is to establish communication between two points while ensuring that malicious activities are not committed by third parties. Artemis currently has many options for encryption algorithm ciphers that can meet their security requirements. However, for this specific job performed for Artemis, *AES* is the preferred cipher to use. Currently, *AES* is adopted by many systems because of its flexibility, unbreakability, and fast decoding speed. In the beginning, *AES* was developed to replace *DES* because it was becoming increasingly easier to break into it as the internet and technology developed (Patel, 2014). In order to secure customer information properly, financial institutions should avoid DES encryption altogether.

Despite the fact that *AES* is symmetrical—meaning it uses the same key for both encryption and decryption—and therefore not always flawless, it's been adopted worldwide for its high level of security. Hackers haven't succeeded in breaking in yet, and even the US Federal Government uses it as a standard for encryption in the public sector recognizing its outstanding capabilities. *AES* does, however, have a disadvantage in its total key lengths of 128, 192, or 256 bits, which don't even come close to highly safe encryption schemes like *RSA*—which can go up to 2048 bits. In spite of this, brute force attacks still cannot be performed on *AES* thanks to the modes in which it operates, and compared to *RSA*, *AES* is much faster and efficient during execution time.

This happens because the longer the key, the longer the authentication process. In the case that a client like Artemis Financial relies on time-sensitive communication with their clients or stakeholders, we should always keep this concept in mind. *RSA* cipher differs from *AES* in that it is asymmetrical, relying on both a public and private key instead of only one (Kaufman, 2018), so anyone can communicate with the company from the outside. *RSA* protects the system against possible attacks by never revealing to the receiver the sender's real key, and since the *RSA* cipher does not require senders to know the recipient's information, it suffers significant delays when sending data.

Fortunately for us, *AES* and *RSA* can be combined in a Cipher Suite to minimize the security breaches that may arise from relying on only process during a project. In this research developed for Artemis Financial, I used *RSA* 2048 encryption to encapsulate a temporary key generated by Eclipse Keytools in order to perform the merging, and with this, I was able to preserve performance without sacrificing security. Artemis Financial's research provides an excellent example of how hash functions can be utilized to create keys from passwords, and the combination of security components, including hash functions or algorithms, helps ensure a secure connection. It is possible to use public key cryptography, secret key generation, and security protocols—such as *RSA*, *AES* with its *GCM* mode, and *SSL*—to manage the entire connection.

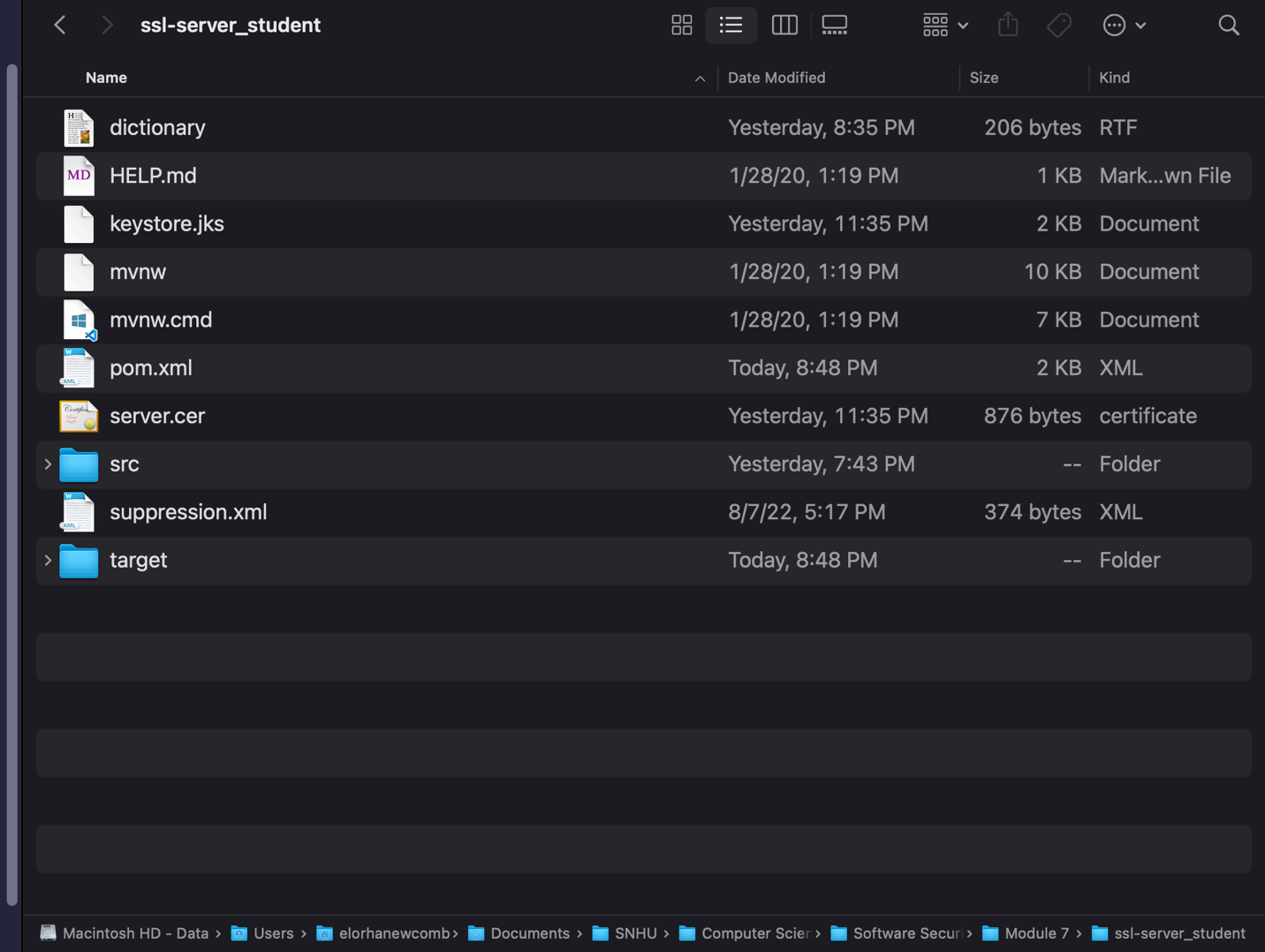
A Galois Counter Mode, or *GCM*, was chosen for this last project, as it prevents data corruption during authentication in addition to performing block encryption and employing the traditional counter mode. (Wang, 2019) The block mode of data encryption consists of several rounds of encryption using a substitution permutation. It takes 14 rounds to encrypt an *AES* 256, for example. It involves creating ciphertext from plain text through a process known as key expansion. To ensure that malicious software would not be able to easily decode the whole content based on one of the blocks, the ciphertext is broken up into several blocks of information and XOR’d in each round, to ensure each block of information has a unique value (Ferguson, 2018). Rows are shifted and columns are merged. A new block of data then undergoes another round of processing after this process is complete and so forth. Using *AES256* to secure the text would keep it safe from unauthorized eyes even if the network file was compromised. However, to decrypt the message successfully, both the sender and the receiver would need the same private key. For Artemis Financial, the adoption of this mode is critical to avoid coordinated attacks like Man in the Middle, which intercepts messages and tampers them in order to perform systems attacks and remain unnoticed. The process of hacking such files could take as long as an individual’s lifetime to be broken into using brute force.

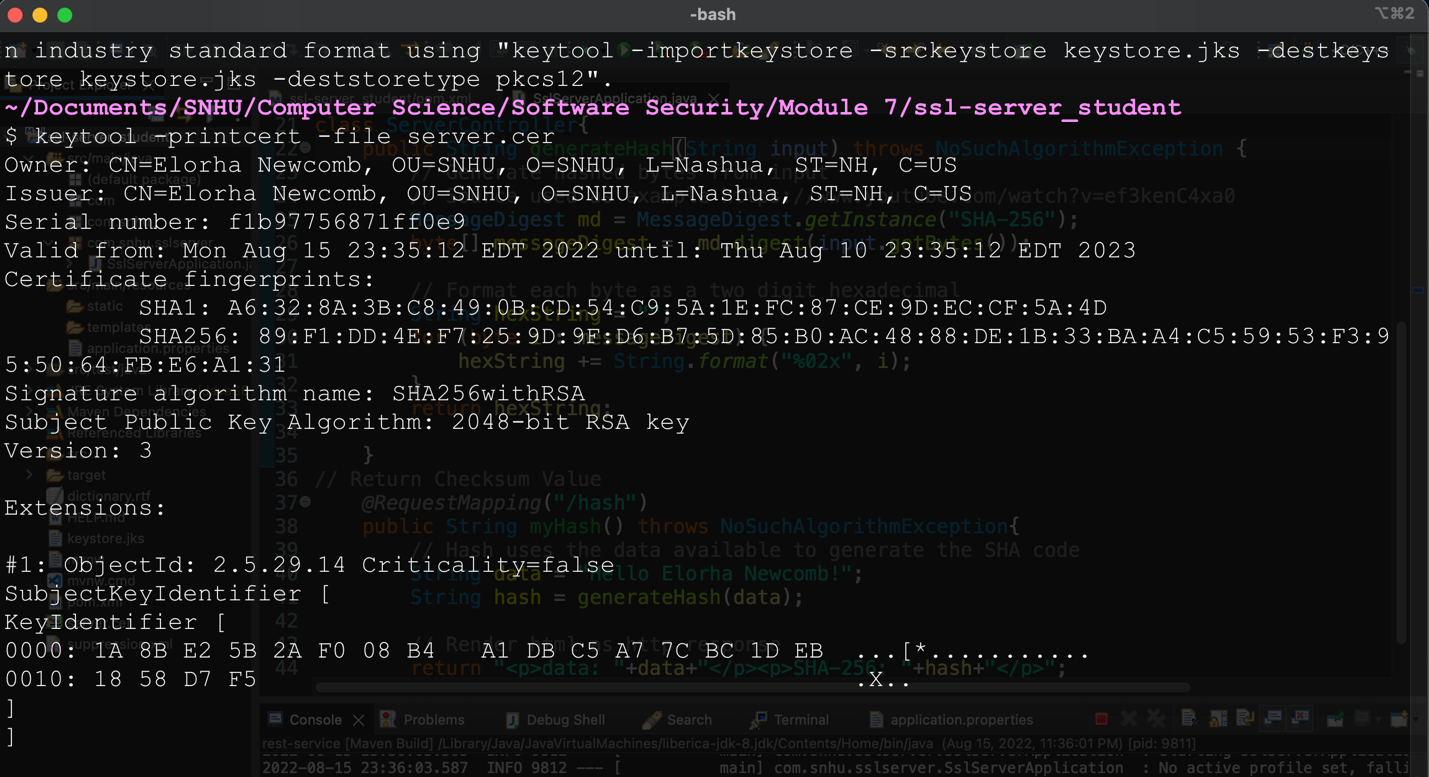
In today's world, encryption algorithms have many different testing and acceptance levels. Financial services companies are responsible for developing and implementing cybersecurity, managing incidents and risks associated with cybersecurity, and complying with local regulations. Security requirements have increased among regulatory agencies and the federal government. The importance and expectations surrounding cybersecurity are paramount when considering the most common threats facing financial institutions. Several techniques are being used daily to change or take control of relevant user data. Understanding the crimes most likely to be used against Artemis Financial is essential. There has been an enormous increase in these crimes in recent years of credential and identity thefts, which are a constant concern in this industry. In order to remedy this problem, it is critical to secure systems and educate those who have access to internal data. In addition, third-party vendors, employees, as well as our cloud providers should be verified as being aware of the issues and working with our security team to ensure Artemis Financial goals are properly met.

## 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.



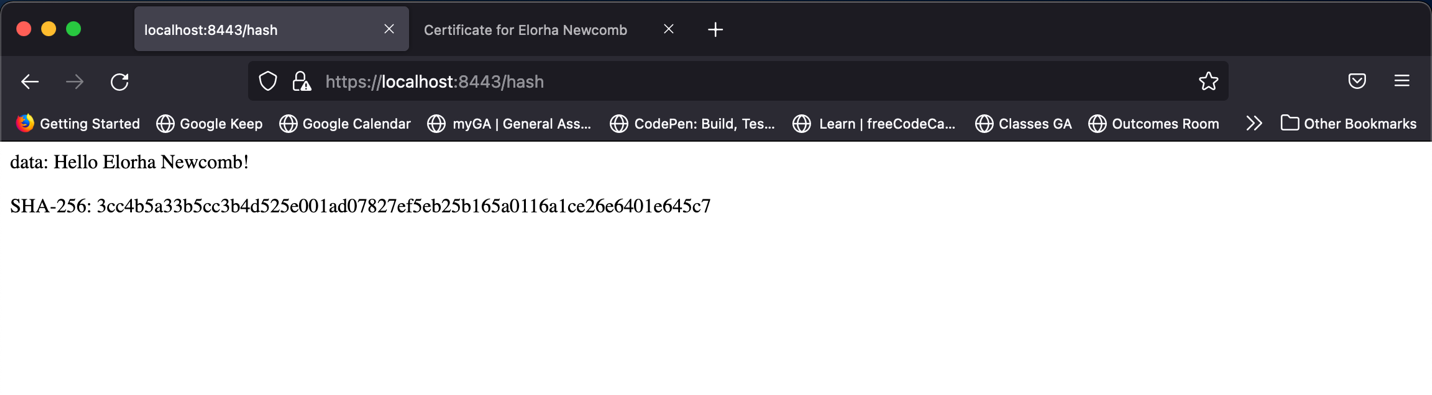


## 3. Deploy Cipher

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.

SSH With Key

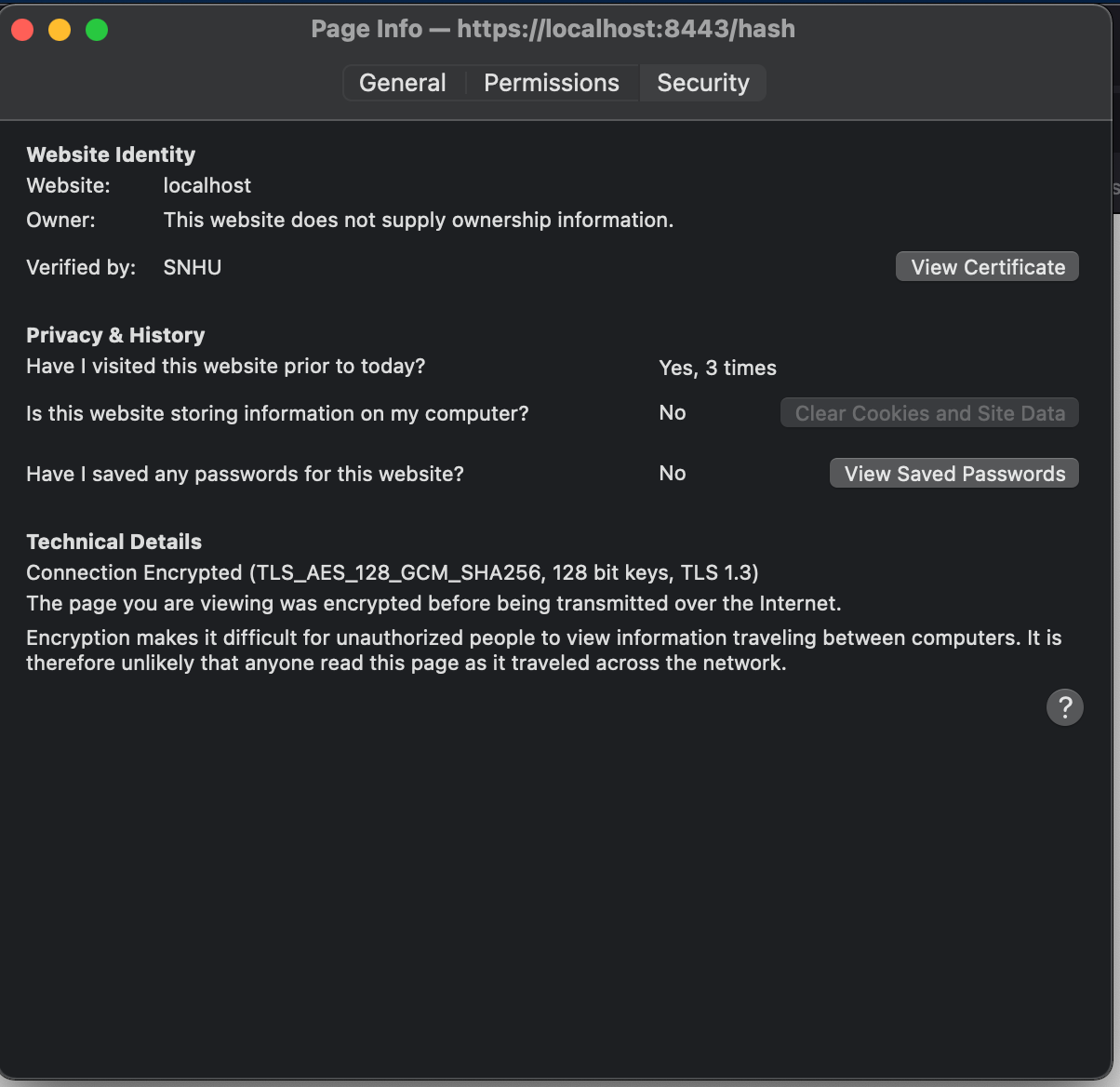


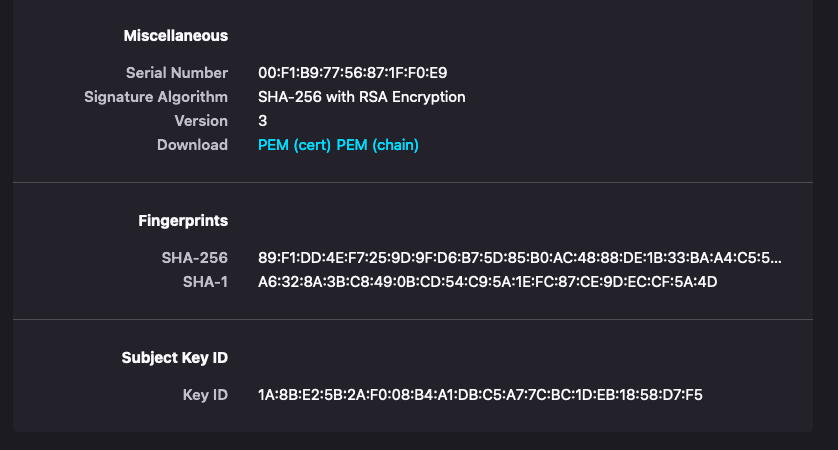
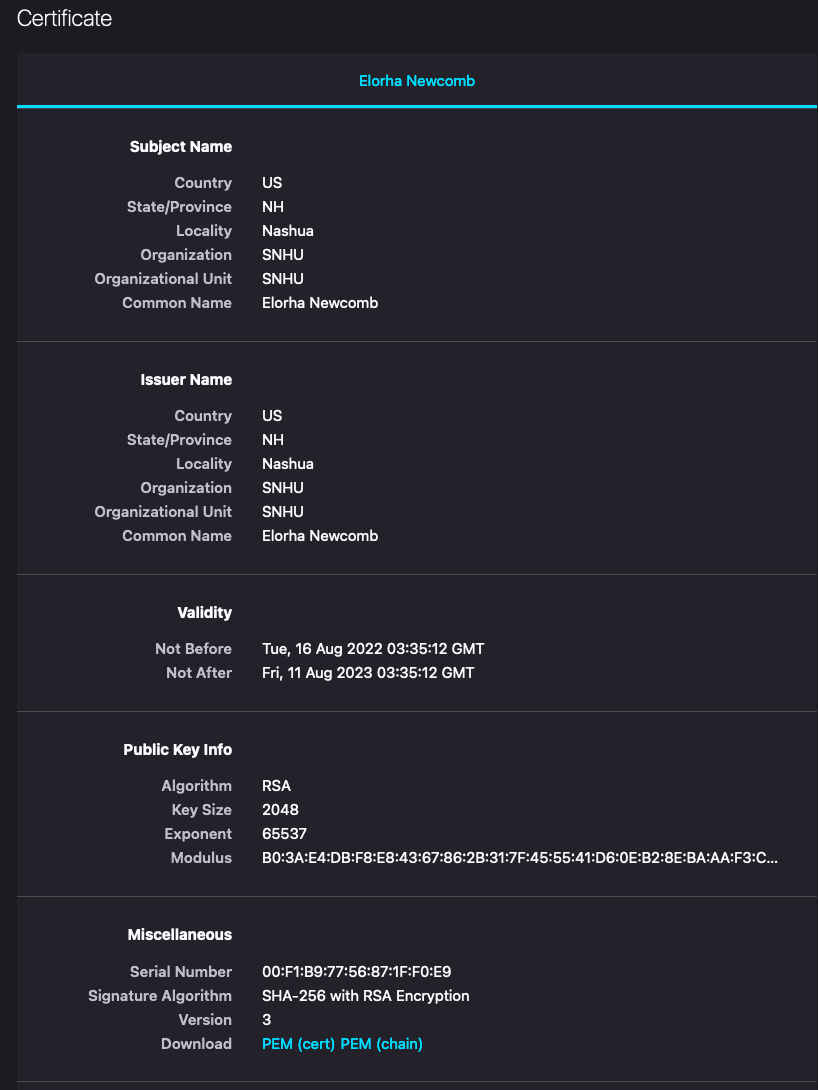
## 4. Secure Communications

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.

Localhost Information

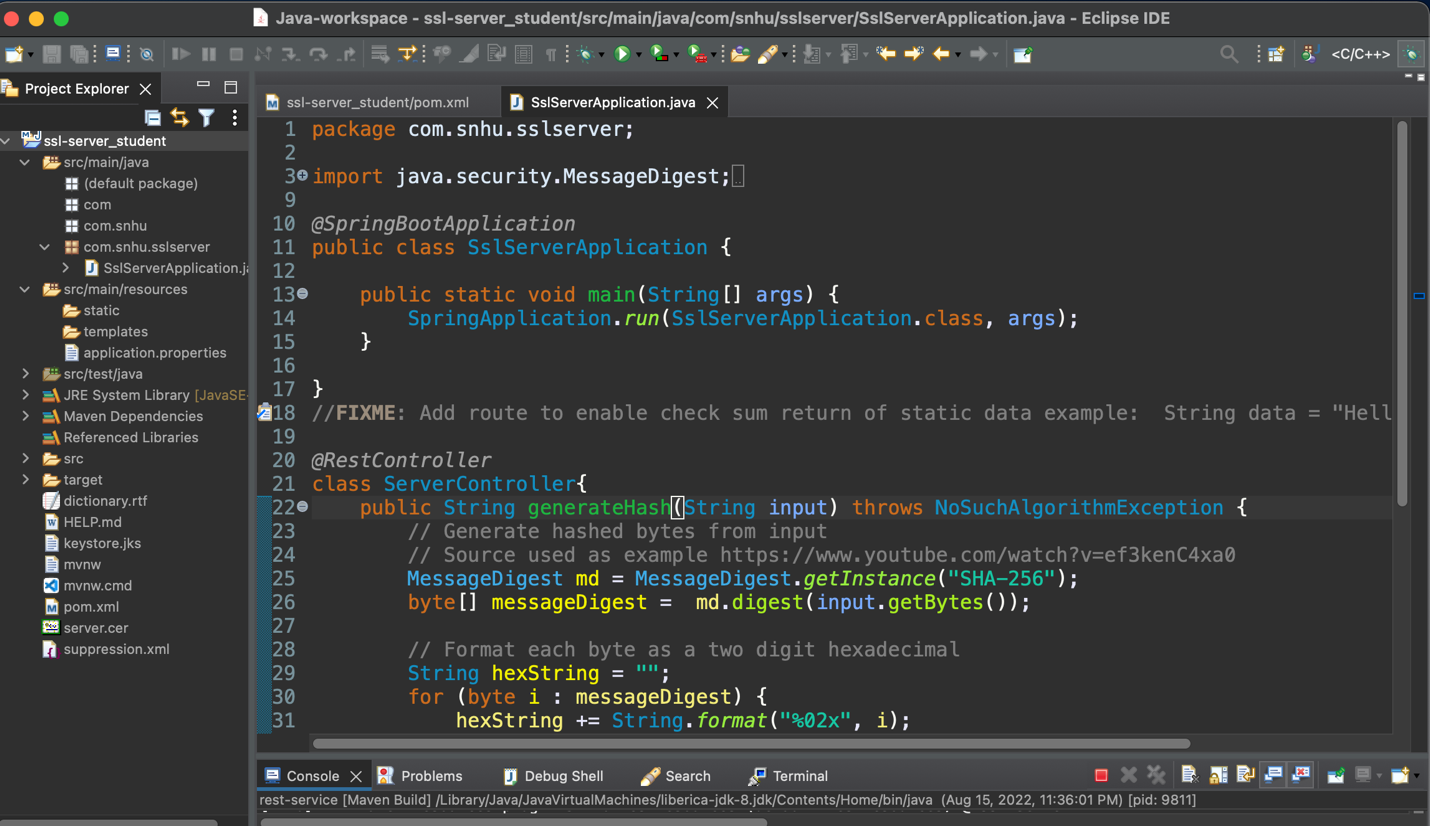


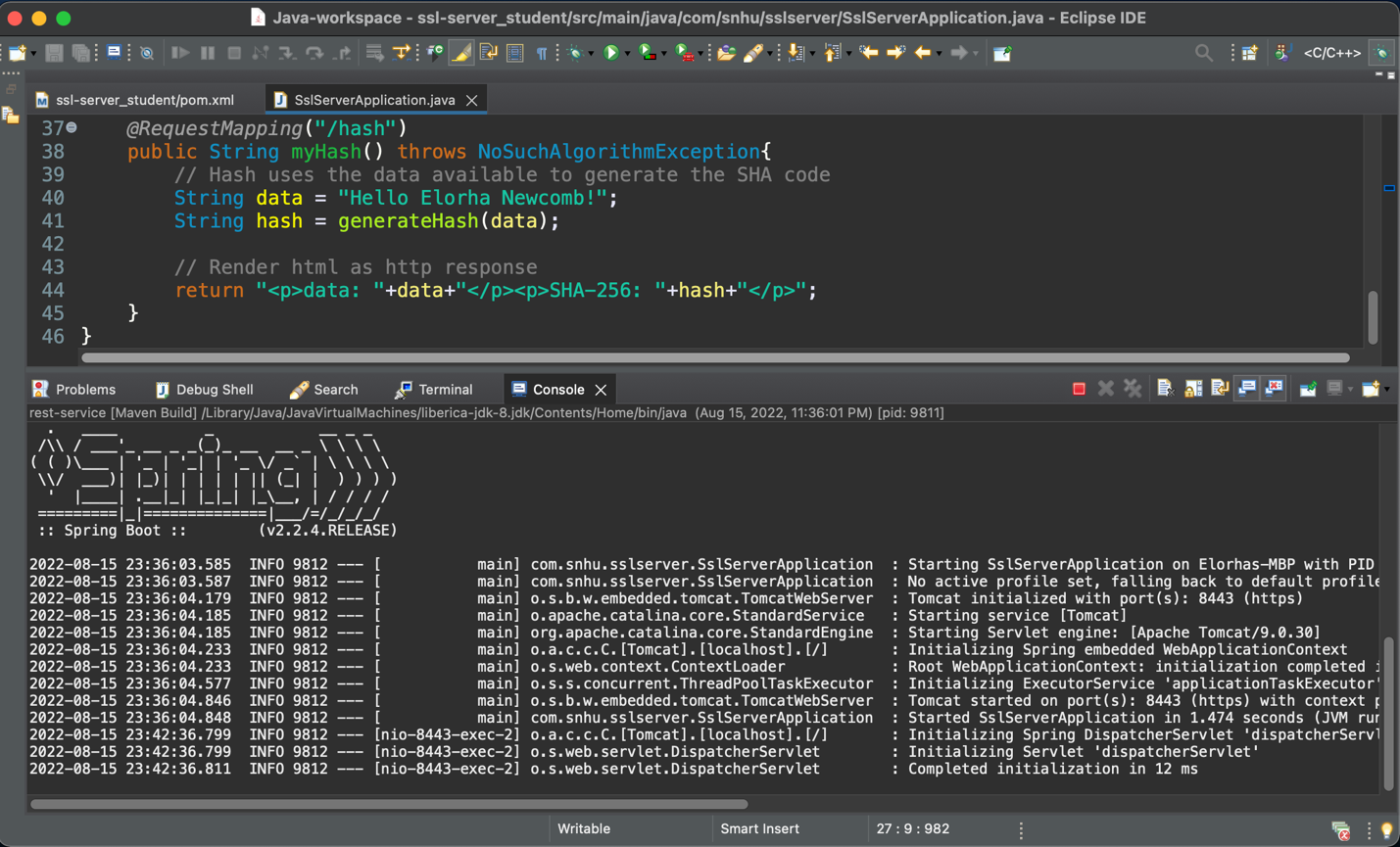
Certificate Generation

## 5. Secondary Testing

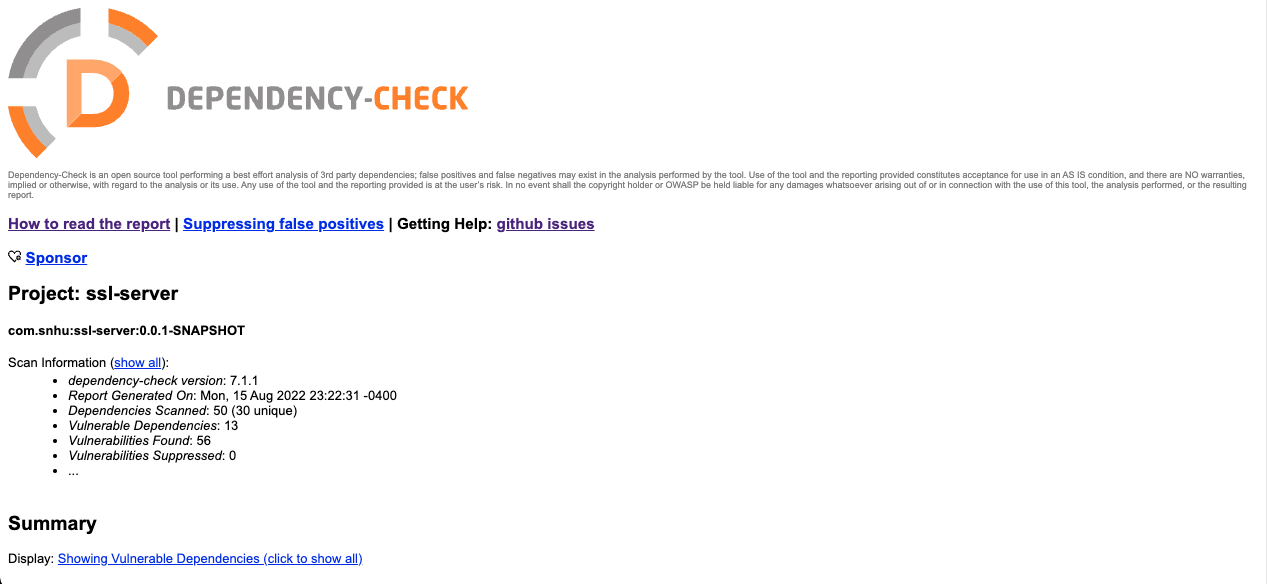
Complete a secondary static testing of the refactored code using the dependency check tool to ensure code complies with software security enhancements. You only need to focus on the code you have added as part of the refactoring. Complete the dependency check and review the output to ensure you did not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

* Include the following below:   
    
  A screenshot of the refactored code executed without errors

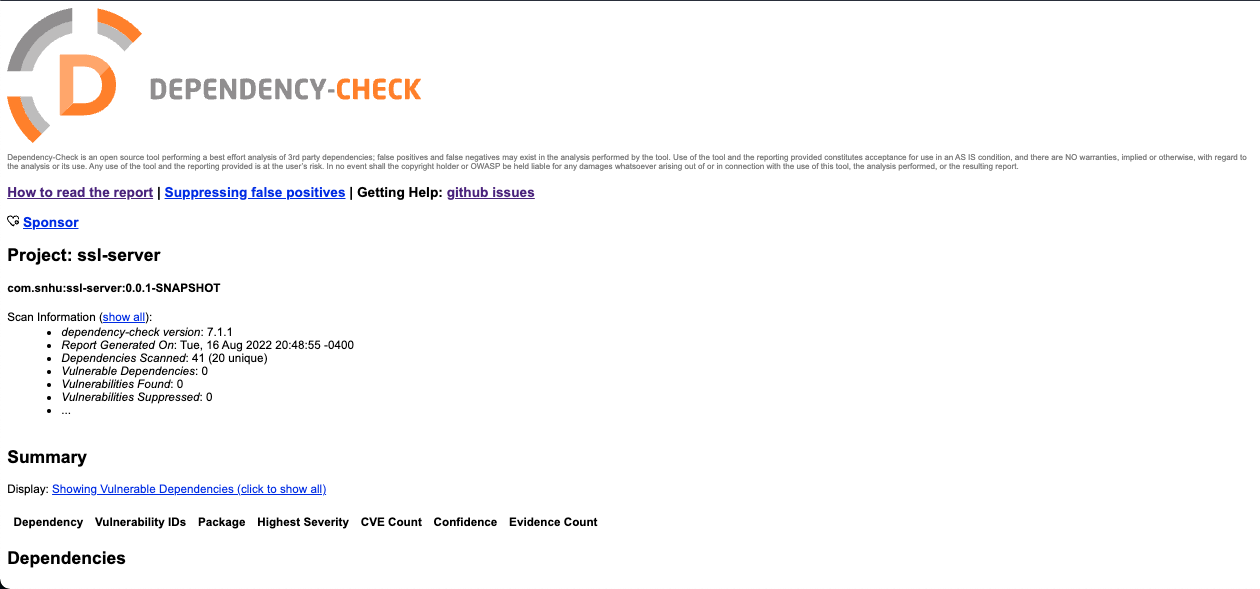




A screenshot of the dependency check report



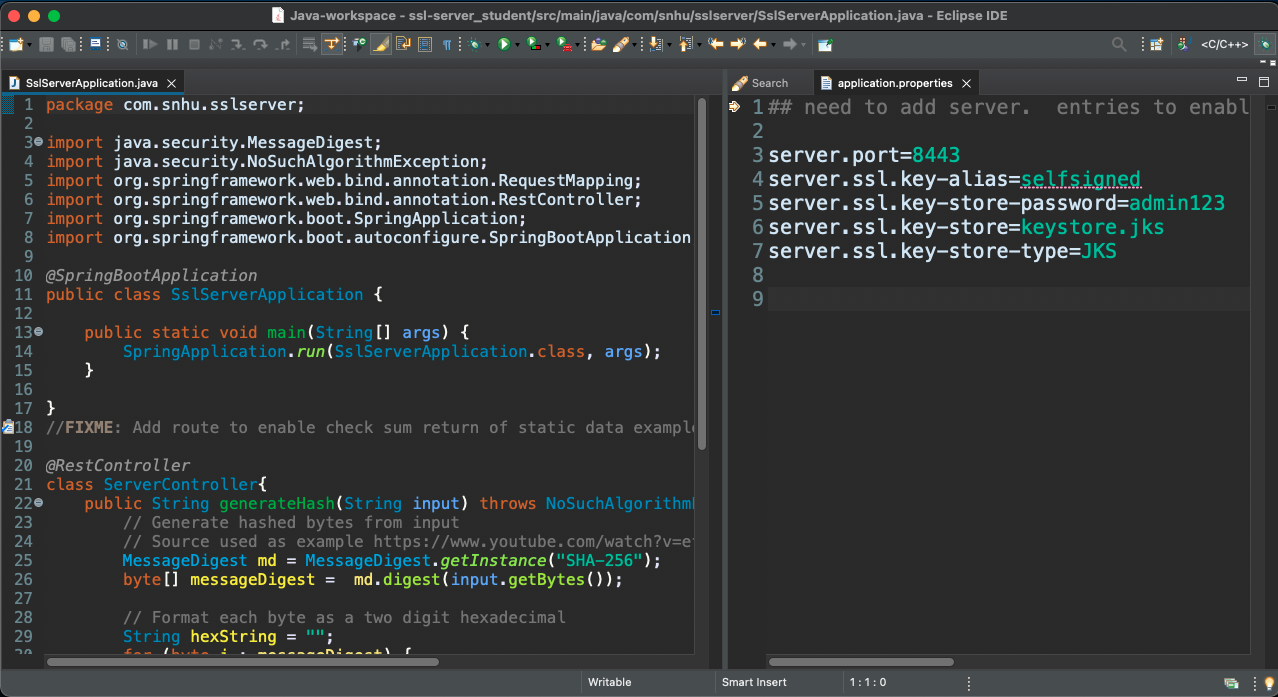
Dependency check after refactory

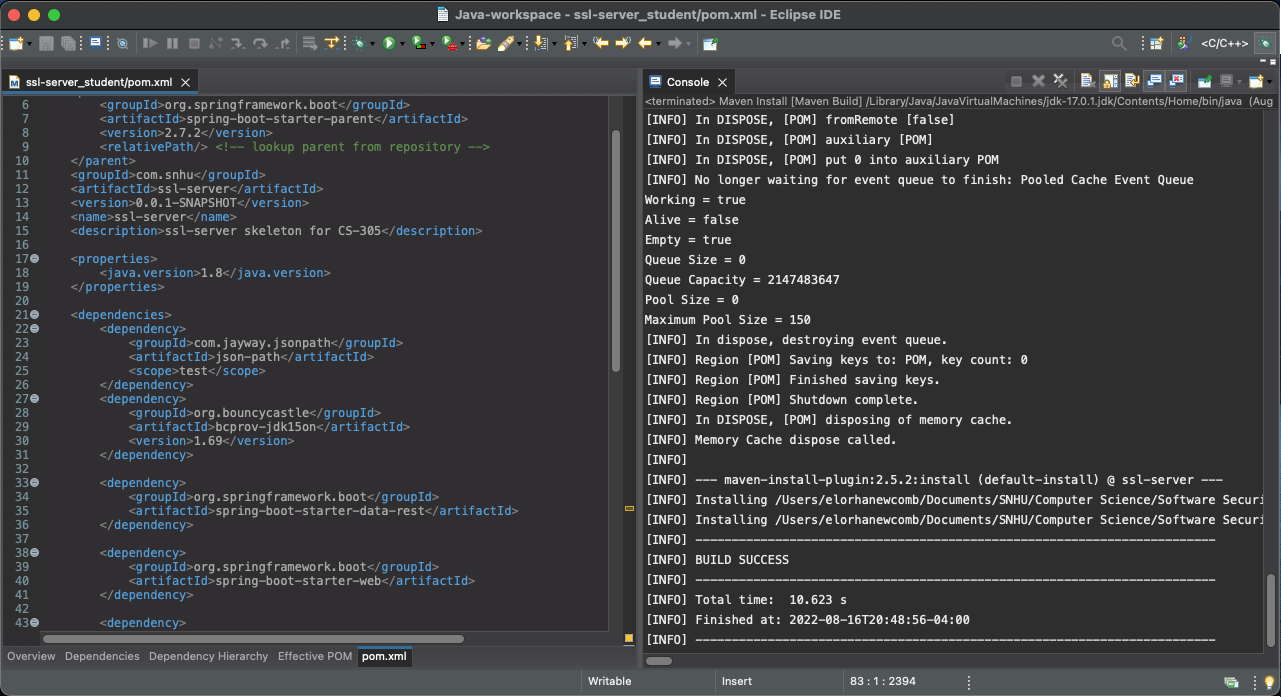


## 6. Functional Testing

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.

The main changes performed in my code aiming to strengthen security were made on the server and in the pom.xml file. In the beginning, the code was thoroughly checked for errors and a dependency check was performed to verify all the current vulnerabilities that needed fixing. The third step was to follow *OWASP* guidance and address all the dependency issues on the code until they were all up to date and no more errors could be found. For secure encryption and to create my self-signed certificate, I generated a temporary *.jks* key and used it to fill out the data for the hash function located in *application.properties*. Then, I implemented the checksum verification and ensured the server was working properly. Finally, I used my certificate to convert *HTTP* into a secure *HTTPS* connection, ensuring the whole communication is now encrypted and protected from opportunist attacks listening to the communication between sender and receiver. To effectively prevent further unwanted attacks, the most valuable tool is ensuring that a client/server communication is properly established and is free of breaches, which is why we have learned to set up a R*EST API* that follows the industry standard for encryption, and a dependency check to spot vulnerabilities more easily. Many of the errors from the pom.xml could be fixed with a simple update to the dependency they were using, which is extremely convenient to look at and update manually, especially for smaller teams.

If all dependencies are up-to-date and all containing code is functioning correctly, a significant number of hackers won't be able to gain access to Artemis Financial's application and steal sensitive data. To ensure that the newly implemented security mechanisms continue working properly with the application, it is recommended to run the *OWASP* dependency check at least once a week. The recommendation for the client is that our team runs a weekly run of *OWASP* on their servers, searching for any unusual activity. And also perform code runs often, at least on a bimonthly basis. When routine maintenance is scheduled at the same time and on the same day, it is easier to pinpoint issues. Refactoring and keeping frameworks updated is equally important, as they are extremely targeted by wrongdoers, and one of the easiest ways to invade a server without being spotted.

References

Ferguson, N., Kohno, T., & Schneier, B. (2010). 3. In *Cryptography engineering design principles and practical applications* (pp. 46–56). essay, Wiley.

Kaufman, C., Perlman, R., & Speciner, M. (2018). 2. In *Network security: private communication in a public world* (pp. 46–50). essay, Pearson.

Patel, M., & Prabu, I. (2014, January 1). *Figure 1 from an optimized cryptography algorithm and key exchange method for small scale devices: Semantic scholar*. An Optimized Cryptography Algorithm and Key Exchange Method for Small Scale Devices. Retrieved July 28, 2022, from https://www.semanticscholar.org/paper/An-Optimized-Cryptography-Algorithm-And-Key-Method-Patel/f1a38bb0b3cb2f20fb3c1b6216486a655742cb7a/figure/0

Wang, S. (2019, August 8). *The difference in five modes in the AES encryption algorithm*. Highgo Software Inc. Retrieved July 30, 2022, from https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/