****

# 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** |
| --- | --- | --- | --- |
| **2.0** | **10/16/2020** | **Austin Autrey** | **Project 2** |

## 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

Austin Autrey

## 1. Algorithm Cipher

The encryption algorithm I chose to use was the SHA-256 algorithm. It is known as a secure hash algorithm, and it falls into the category of message digest algorithms. The SHA-256 message digest algorithm essentially is used to calculate a form of a message through electronic data. For SHA 256, this message’s length must be less than 2^64 bits, and then the message is put into a hash algorithm. This in turn produces a message digest. This algorithm is considered secure because it renders it almost impossible to find the same message that corresponds to the original message digest, or to find different instances of a message that will produce the same digest. This essentially means that it is highly unlikely for a collision to occur.

The hash functions of the SHA-256 algorithm consist of a cryptographic hash that is known to modify hashed data to render it unreadable. This in turn outputs verification failure when used with certain algorithms or the generation of random numbers. As far as bit levels of the cipher, this algorithm uses message blocks each containing 512 bits. These bits are then represented as a sequence of sixteen 32-bit words. The message is essentially padded, parsed, and sets the initial hash value. In this instance the SHA-256 algorithm will consist of eight 32-bit words in hex. The hash computation then occurs, and outputs the message digest.

Symmetric keys deal with encryption. In this assignment a symmetric or secret key was used to disguise information. In turn the information can’t be undisguised unless it has the secret key. In this situation the password to the keystore was the secret key. An asymmetric or non-symmetric key has to do with signing. It is a form of public key cryptography using both a public and private key. The private key remains secret and would only be known by the sender of the message. This key signs the data. The public key in contrast can be known by anyone and is used to confirm the signature. In some cases, the public key can also be used for encryption, but it depends on the specific algorithm. In conclusion, sometimes systems use random numbers. Random numbers essentially produce a pathway of numbers randomly. This is used to gather random data and can be done by using a cryptographic pseudorandom number generator. There are also methods in the java library that can be used to generate random numbers.

Cryptography has been around for a long time. This began with secret messages being encrypted in earlier times, so they could not be compromised in times of war or terror. In contrast, we now have digital encryption algorithms. One of the first instances of encryption being used electronically was by the Germans in world war two. This continued to progress into the 1970’s when a cipher called Lucifer was created. This eventually led to it becoming the data encryption standard until it was compromised. This was later replaced by the AES, and as technology continued to evolve more ciphers were created. Today there are ciphers such as AES, RSA, and several more. As technology continues to develop with things such as artificial intelligence, the need for new and stronger encryption algorithms will continue to be in demand.

## 2. Certificate Generation

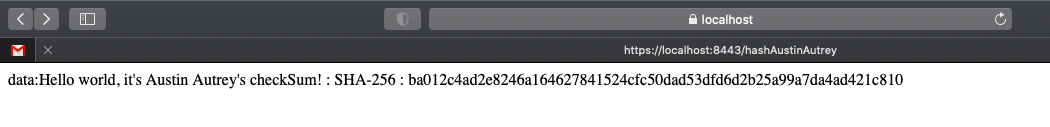
In order to generate an appropriate self-signed certificate, I used the mac terminal to create a public-key cryptography standard pk12 keystore. Attached below is the self-signed certificate that was generated from creating the keystore in the terminal.

Text

Description automatically generated

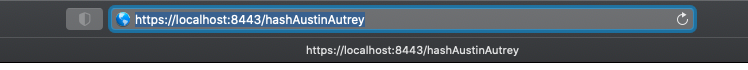
## 3. Deploy Cipher

Attached below is the screenshot of the produced checksum. It shows the generation of a checksum deploying the algorithm cipher SHA-256 that I chose as well as a short message. To generate this checksum, I simply created a new class called SslServerController. I then created a method called get Hash to create a message digest containing the algorithm cipher. Next, I created an additional method myHash in order to produce a message for the checksum.



## 4. Secure Communications

Attached below are two screenshots showing proof of secure communications. The first screenshot simply shows the URL using the hypertext transfer protocol secure port of 443. This verifies secure communications, and also shows the transition from HTTP to HTTPS. The second screenshot shows the result of clicking on the lock in order to show the certificate being used to allow for an encrypted connection to the local host. In order to provide this secure connection, I simply created a certificate and imported it into the same folders containing the pom file and application properties. I then refactored the code in the application properties providing the details of the keystore. This involved entering the alias, password, keystore, and keystore type of the keystore imported.



Graphical user interface, website

Description automatically generated

## 5. Secondary Testing

There are three screenshots listed below that show the use of secondary testing. The first screenshot shows the refactoring of the code in order to produce a checksum with the encryption cipher. The third screenshot shows the code refactored in order to match the keystore that was imported into the application, so a secure checksum could be generated. The second screenshot shows the dependency report produced after a successful verify, compile, and spring boot run test. I would like to note that this is the initial dependency check produced before any false positives could be suppressed. There were no new security vulnerabilities introduced in this phase.

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

A screenshot of a computer screen

Description automatically generated

[Insert screenshots here.]

## 6. Functional Testing

The screenshots below show how the code was refactored in order to suppress false positives in the initial dependency report. After the initial dependency report was produced, it was determined that there were false positives, as well as security vulnerabilities. The security vulnerabilities came in the form of CVE’s. I determined the potential false positives and suppressed them by the CVSS confidence score ratings. I determined false positives by any CVSS score of seven or below. I also determined that the potential vulnerabilities involving tomcat could possibly lead to denial of service attacks, which is why I did not suppress any vulnerabilities involving a greater score than seven for tomcat. All other CVE’s suppressed were also due to the CVSS score being seven or lower. In order to suppress the CVE’s, I simply created a suppression file listed below as a screenshot, and then refactored the pom file as shown below in order to include the suppressions. The other identified vulnerability excluding the two different varietals of tomcat was SnakeYAML. It had a score of 7.5 for CVSSv3, so I determined it to be a vulnerability, not a false positive.

Text

Description automatically generated

Text

Description automatically generated

## 7. Summary

Discuss how the code has been refactored and how it complies with security testing protocols. Be sure to address the following:

Throughout the process of working through this project, there were different areas specifically from the VAPFD that were addressed. First off input validation was used to secure input. This was done by cryptography through encryption of the checksum using the SHA 256 algorithm. This in turns also represents cryptography, as the algorithm cipher I used was included in the refactoring of the code. Code error, code quality, and code review through code services and plugins were all involved in the refactoring of the code. This involved using the dependency check to check for vulnerabilities in certain third-party services, and maven plug ins. After using the dependency check, I refactored my code accordingly through the suppression and pom files to suppress false positives. Secure API interactions were achieved by converting to HTTPS protocol from HTTP and importing my certificate for use by adding the keystore to the application. Encapsulation was used through the encryption of the message digest.

Adding layers of security for this application began by generating a certificate through the mac terminal. I chose to use the keystore of type PKCS. I then proceeded to add the keystore to the application through both the application properties and pom file. The next thing I did, was refactor the code in the SslServerApplication class. I did this by adding a new class in order to add my algorithm cipher and produce a checksum. Then I needed to convert HTTP to HTTPS. I did this by refactoring the code in the application properties file to include the information from my keystore. This included using the secure port 443 as well. I then was able to produce my message digest in the browser. Next, I needed to check out the dependency check for vulnerabilities and false positives. This was done by checking out the CVSS scores and determining if the confidence level was high enough to not be a false positive. I then created a suppression file, added the false positives, and then refactored the pom file. Lastly, I ran another dependency check, and kept repeating the process of removing false positives till there were none left. All of these layers helped add layers of security to the website in order to help prevent breaches as well as helping keep the website secure.

The best practices to maintain the security of the software application consists of few things. First, I would continue to run dependency checks of the software application regularly. New CVE’s come out every so often or are revised to help get rid of the risks. For example, the CVE could be revised where the version of the software being ran is no longer at risk. This would now make that vulnerability a false positive. It will be best to continue to use file verification in order to confirm secure communications. This is especially important as you are dealing with personal financial information of customers. Also, continue to always use the HTTPS protocol over the HTTP protocol. One is secure, and one is not. It would also be smart to run the application with the fewest privileges as possible, and these can be adjusted to enhance security as needed. Another idea is to implement the xss-protection security header in order to avoid cross-site scripting attacks. In conclusion, always conduct security awareness training, enable public key pins to prevent man in the middle attacks, and always use the current version of transport layer security.