****

# 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** | **15Aug2021** | **James Kraatz** |  |

## 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

James Kraatz

## 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.

Verifying that the intended parties are securely connected, and that the data shared between them is of paramount importance, especially dealing with financial information. The type of information available provides a tempting target and motivation to hackers. I recommend the TLSv1.2 TLS\_ECDHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256 cipher suite using the AES\_128\_GCM algorithm for bulk encryption. This is a minimum recommendation that will fulfill the security needs of this application. I strongly suggest implementing the five TLSv1.3’s as well on the server side. The minimum recommended TLSv1.2 cipher suite implements AES\_128\_GCM block encryption which has not been hacked to date. AES128 uses a 128-bit encryption that is faster to compute than the AES256 encryption and therefore consumes less resources of the server. The weak point of the AES128 encryption system is passing the decode key to the receiver.

The hash function assists in communications by conferring confidence on the integrity of the data passed to the receiver. The suggested suite implements the SHA256 hash function to generate a checksum value of the passed data. The receiver then verifies the data by calculating the checksum and comparing the calculated checksum with the sender’s checksum.

Three major elements of keeping the client’s information secure when transferring that data across the internet are authentication, confidentiality, and integrity. Authentication is the verification that the sender and receiver are the intended parties of communication and data transference. Confidentiality is the act of maintaining the secrecy of the communication and data from parties other than the intended receiver and sender. Lastly, integrity ensures the data is not secretly changed in transit without the receiver of the data knowing about it.

Random numbers are pattern less and consequently unpredictable. They provide was of authenticating systems by producing a unrepeatable and, hopefully, unguessable data to process and return. A server can send a client a new random number that is processed, by the client’s system, when the client enters their password. The processed output between the password and the random number can be sent back to the server where it is used to confirm, or authenticate, the client. The allure of this is that each time the client logs in, a different authentication value is sent across the network and if captured, is not likely to be useful to use in a replay attack.

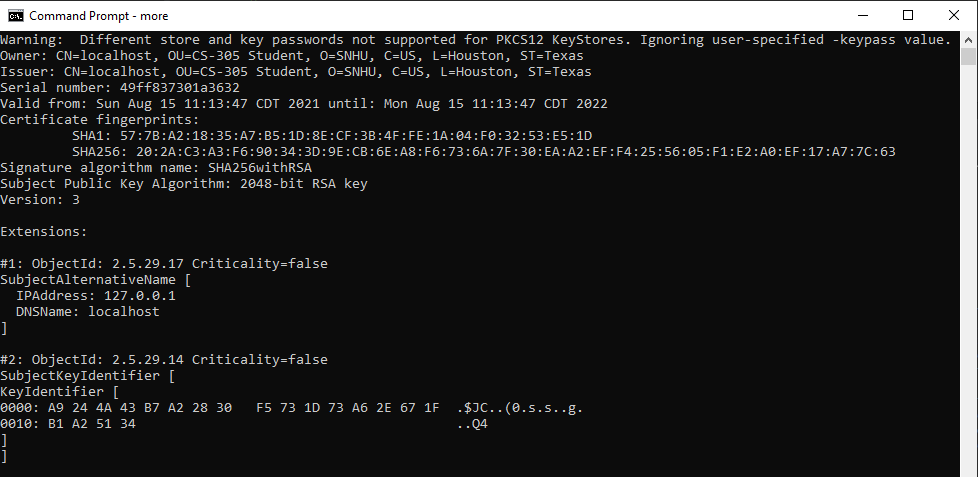
Keys are used in encrypt and decrypting data. This encryption is the attempt to keep the messages between the client and server confidential. There are two types of keys used, asymmetric and symmetric. Asymmetric keys have different values used for encrypting and decrypting data. Typically, one key is public, and an authenticated endpoint, could be either the client or server, uses the public key to encrypt the message to be sent to the other endpoint. The other key, the “private” key, is then used to decrypt the message. The use of the term public does not mean that the key is not tried to be kept secret. Random numbers once again become useful here in generating both public and private keys into changing and unpredictable keys. Symmetric keys are where the same key used to encrypt the key is used to decrypt the key. Symmetric keys require that the two endpoints have the same key to share encrypted data. These keys need to be shared between endpoints anytime they change. One way of doing this is to share the next sessions keys in the current session. Regardless how the keys are exchanged, random numbers can be used in the process of generating the “passwords” or keys for the encryption to prevent guessable data, such as birthdates and addresses, from being used as keys. Data integrity is knowing the same data sent from one endpoint is the same as the data received at the other endpoint. This is done by sending a “digital fingerprint” of the data packet. The sending endpoint will process the data being sent to create a long value, called a checksum, and sends it with the data to the receiver. The receiving endpoint recreates the checksum by reprocessing the received data and comparing it with the checksum sent by the sending endpoint. If the two checksums match, the data is presumed to be untainted.

While the recommend cipher suite is TLSv1.2 for compatibility reasons, the current standard is TLSv1.3. TLSv1.3 removes several older, more vulnerable, suites and no longer allows their usage, including all SSL and any TLS version prior to TLSv1.2. Authentication, in TLSv1.3 requires “perfect forward secrecy” which is an encryption system that changes the keys used for encryption and decryption frequently and automatically. TLSv1.2 does provide some forward secrecy suites and are recognized in the key agreement section of the cipher suite name. The forward secrecy key exchange algorithms all end in “E” for ephemeral, such as ECDHE for example. Since TLSv1.3 requires forward secrecy, there is no key exchange method listed in their cipher suite names.

## 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.

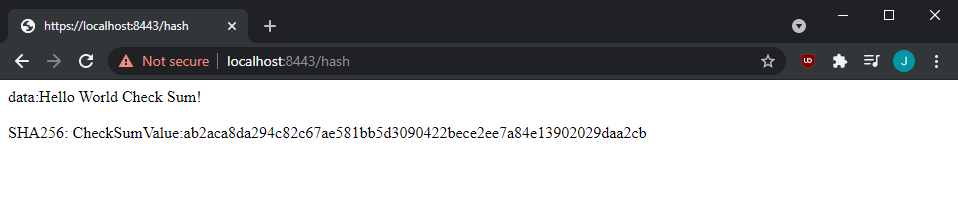


Certificate File

## 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.

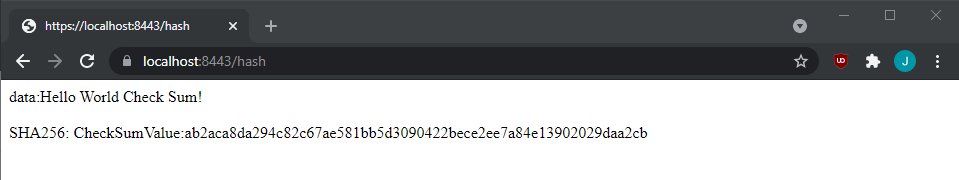


Checksum verification

## 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.

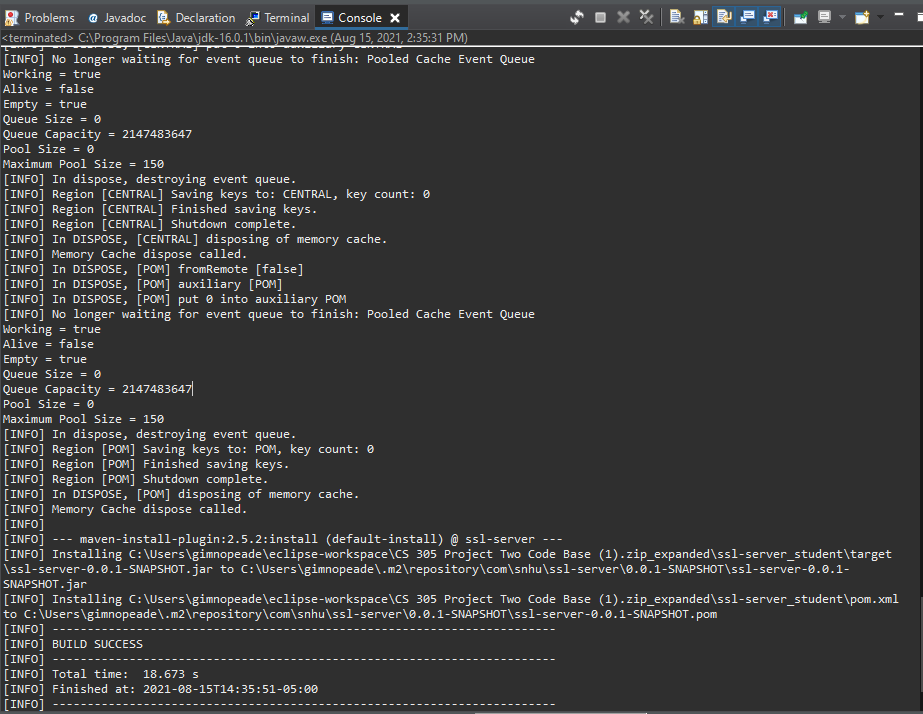


Secure webpage

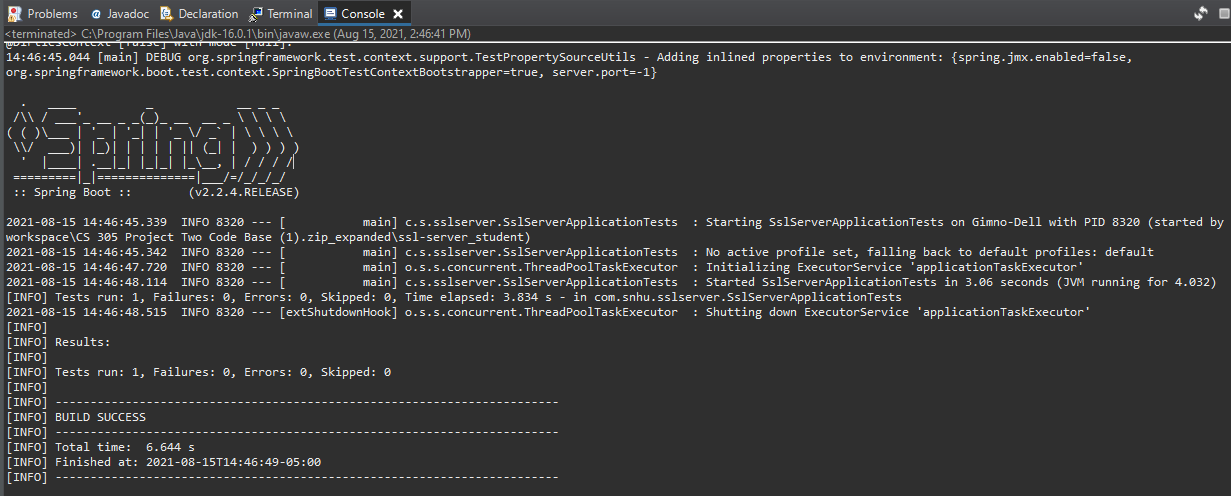
## 5. Secondary Testing

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.

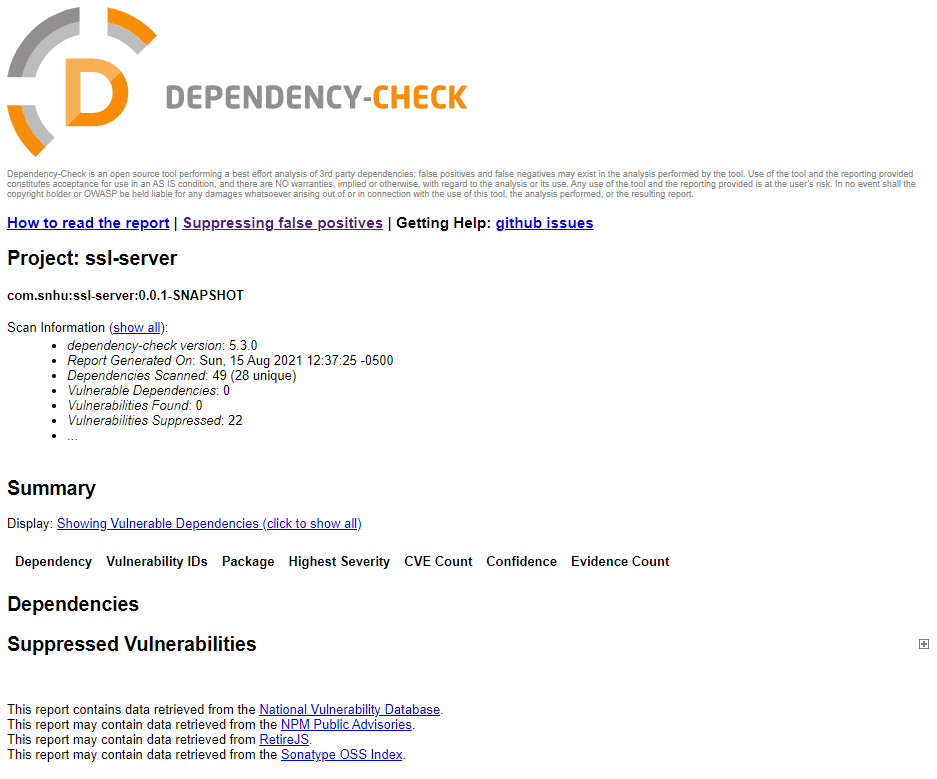
* Include the following below:
  + A screenshot of the refactored code executed without errors
  + A screenshot of the dependency check report



Maven "install " goal



Maven "test" goal

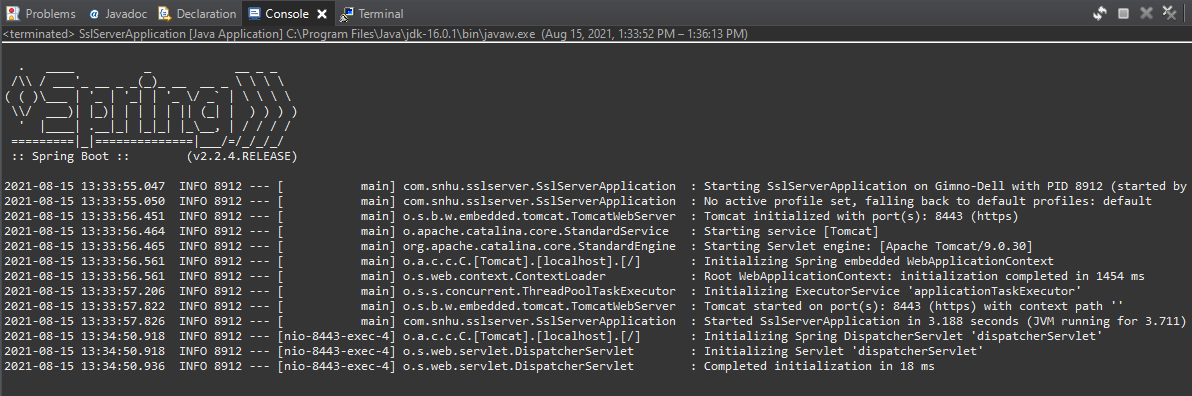


Dependency check report

## 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.



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.

Input Validation, Cryptography, Client / Server vulnerabilities are areas of security addressed by the code refactoring. Input Validation is addressed by confirming the data is from a reliable source using certificates and certificate authorities. Cryptography is used to generate keys and checksums used to validate certificates and message integrity. Client/Server vulnerabilities are reliant on the authentication between the endpoints and by confirming the applied certificates and verifying the data checksums the client can be confident it is communicating with a trusted server.

Input data should be confirmed safe, allowable lengths, and properly formatted. If possible, free-form input fields should be avoided. Input data should then be filtered and parsed before processing. This is first before authentication and authorization to prevent a hacker from using data input fields to gain unwanted access to the system. Authentication and authorization follow to provide valid access to the system services. The access, data flow between two endpoints, then needs to be encrypted to maintain its integrity. All system components are to be encapsulated to prevent revealing the systems structure to intruders.

Best practices to follow to keep the software secure are to keep the keystores on a separate server than the server application. Frequently reseed the random number generator and update the private key passwords. Use a reliable Certificate Authority and do not use certificates with unnecessarily long validity times. Lastly, update the software when new releases come out as the new releases may address software security vulnerabilities.

Attached with this report is the “ssl-server\_student” java project implementing a secure communication protocol. The source files along with the vulnerability reports are included in the compressed CS-305 Project Two\_JamesKraatz.zip file.