****

# 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

Yuriy Kuptsov

## 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.

I recommend SHA-256.

High-Level Overview of the SHA-256 Cipher

The SHA-256 (Secure Hash Algorithm – 256 bits) algorithm belongs to a family of functions designed by the United States National Security Agency. An extremely high-level description of the algorithm follows:

The input is broken into fixed length blocks of bits. Next, a compression function is applied to the first block with an initialization vector comprised of random or pseudorandom bits. The result of previous compression is fed into the next compression function along with the next block from the message. Likewise, the output of the compression is fed into the next compression function along with the next block from the message. This process is continued, until all of the blocks of the message have been consumed by a repeating sequence of six distinct compression function, each taking its input from the previous output combined with the next block in the message. Finally, the result of this chain of compressions is fed into a final compression which produces the final value – the “checksum”.

The SHA-256 Hash Function

The SHA-256 hash function is a successor of the SHA-1 hash function. It is among the strongest hash functions available and has not been compromised. Defined in the FIPS 180-4 NIST (National Institute of Standards and Technology) document, it accepts a 256 bit key and is widely trusted for financial transactions.

Bit Levels of the SHA-256 Cipher

The SHA family of ciphers include a range of bit levels. Common key sizes in bits are 128, 224, 256, 384, and 512. The primary advantage of a larger key size is a reduction in the probability of hash collisions. However, the probability of a hash collision with SHA-256 is astronomically low and successful hacking of a file to produce an intentional collision in order to deceive a system into accepting an invalid data input is vanishingly small. The improvement offered by a higher number of bits is of no real consequence under the circumstances that our application will be subjected to. Conversely, using a key of 385 or 512 bits requires a significantly longer computation time which does not justify the preference of a larger key size than 256. Finally, SHA-256 claims a 128-bit collision resistance. This is sufficiently secure for existing computer speeds in terms of the practicality of breaching its security. Should quantum computers programmed to crack SHA-256 become available, then another algorithm might be preferable. However, we are currently far away from this possibility in time.

Use of Random Numbers in Encryption

Keys are generated using random numbers. Keys generated with non-random numbers are less resistant to cracking. Most systems use pseudo-random number generators seeded with some environmental initial value (e.g. a function of the system time at which is was generated, monitoring of a mouse’s movements, random keystrokes, or some other physical process, etc.). Encrypted data should appear to be random to defy detection of patterns that could be used to infer a method to decrypt the data.

Symmetric vs Asymmetric Keys

Symmetric encryption uses a single key, possibly shared among multiple entities, to encrypt and decrypt messages. Asymmetric encryption used a public and private key pair to encrypt and decrypt messages. Symmetric keys are less secure in that the disclosure of a key can compromise the privacy of encrypted messages. Asymmetric public keys can only be used when combined with a private key. So long as the private key is not compromised, the public key’s utility is diminished. This provides better security.

History of Encryption Algorithms

The first known use of cryptography dates back to 1900 BC, in ancient Egypt, when a secret set of hieroglyphics were used as substitutes for known hieroglyphics. Such substitution methods continued to be used over the thousands of years that followed. The advent of computers made cracking substitution cyphers easy and such methods have become obsolete.

The next advancement is credited to Julius Caesar. A different method of substitution was employed where specific letters of the alphabet were swapped with others in order to scramble communications. They key to such a cipher was the mapping of individual letters in the encrypted message to their actual letters in the alphabet.

In the 16th Century, Vigenere designed a cipher using a key. The key was a sequence of letters. Each letter was assigned a numerical value based on its order in the alphabet (e.g. A = 0 and Z = 25). In a repeating fashion, the key was applied, letter by letter, to each letter in the message, their numerical values were summed, and modulus of the sum and 26 were used to look up the letter to substitute the original letter in the message.

The first electro-mechanical encryption method was created by Hebern in the 19th Century. His system employed rotating disks and a physical key embedded in one of the disks. This was followed by Arthur Scherbius’ Enigma machines used by the Germans during the first and second world wars. It was comprised of multiple rotors that rotated at different rates as the keys of a typewriter were pressed.

In the early 1970s, Horst-Feistel led a crypto-group at IBM and created a digital cipher called Lucifer. In 1973, the US National Bureau of Standards put out a request for proposals for a new block cipher which would become a national standard.

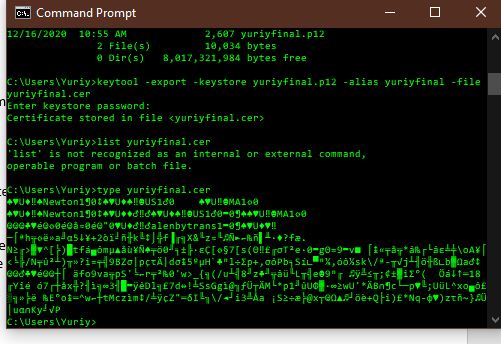
Current State of Encryption Algorithms

The first block cipher endorsed by NIST was Lucifer under the name DES. In the following years, this cipher was broken by exhaustive search. In 1997, NIST put out a new request for proposals. In 2000, it accepted the Rijndael algorithm and renamed it AES. Since that time, SHA-256 has been found to be more reliable than AES.

## 2. Certificate Generation

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

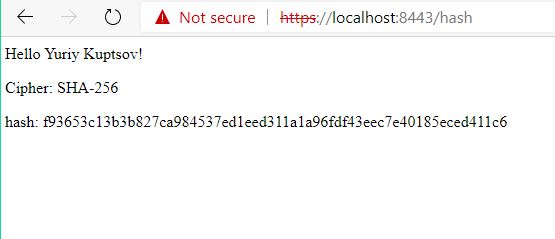
* To demonstrate that the keys were effectively generated, export your certificates (CER file) and submit a screenshot of the CER file below.



## 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.

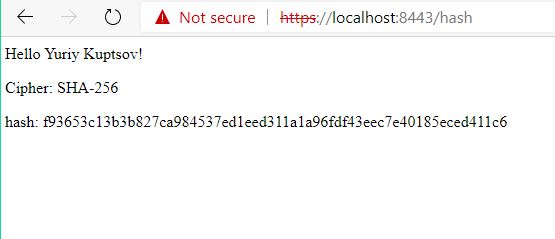


## 4. Secure Communications

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.

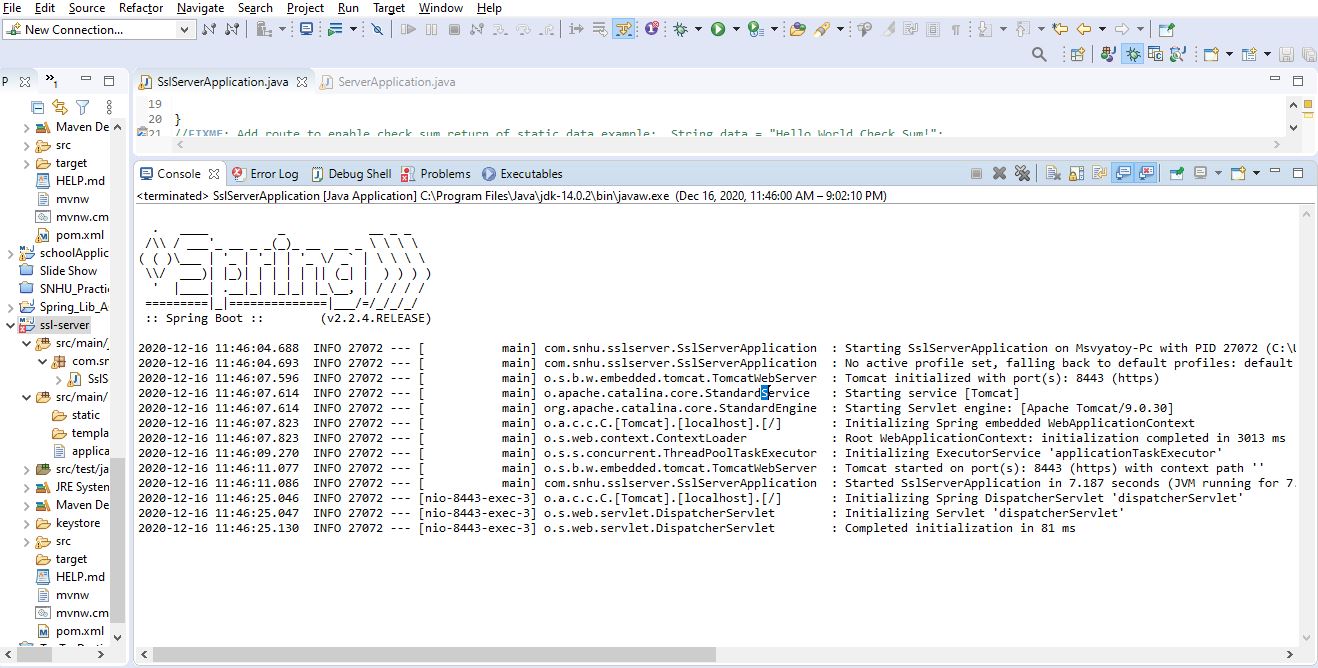
*Note: I am using a private certificate that has not been endorsed by the third-party, thus the “not secure” message. I am using SSL.*

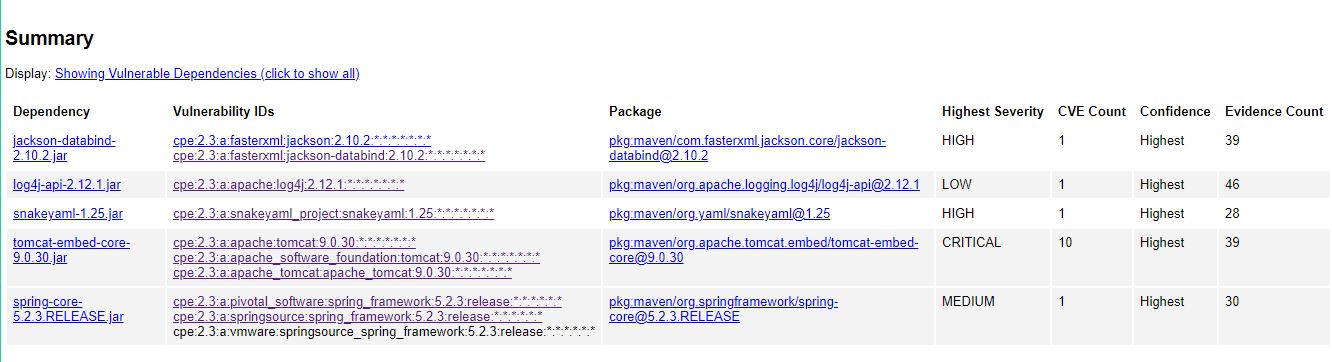


## 5. Secondary Testing

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



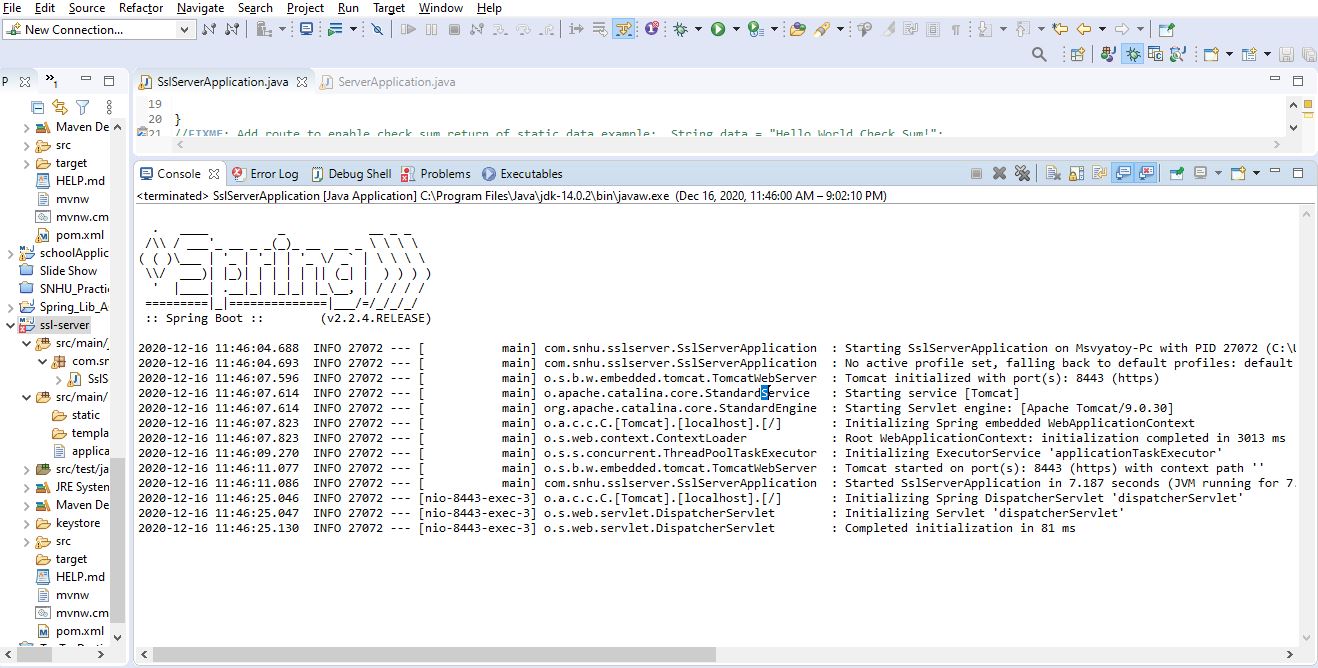


## 6. Functional Testing

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.

The code I added has no syntactical, logical, or security vulnerabilities. There is no need to further modify the code. Consequently, the screen shot remains the same.



## 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.
  + I reviewed the overall architecture.
  + Cryptography: Added SHA-256 cypher for generating checksum.
  + Cryptography: Added SSL certificate and used HTTPS for communication.
  + Code Quality: Reviewed my code and determined that it met best practices.
  + Code Review: Reviewed all aspect of the code and architecture.
* Discuss your process for adding layers of security to the software application and the value that security adds to the company’s overall wellbeing.
  + Generated and deployed an SSL certificate. Used SLL server. Employed checksum on data to ensure validity.
  + To generate the SSL certificate, I used keytool. I moved the certificate into the application’s directory structure and changed the properties file to inform the system of its location, password, and other information.
  + I employed the MessageDigest class to implement the SHA-256 encryption.
* Point out best practices for maintaining the current security of the software application to your customer.
  + I recommend that the client purchase a copy of Iron-Clad Java: Building Secure Web Applications from the webstore on our company website. Additionally, I recommend that the client hire a member of our team to periodically review their application and any changes that they make to the application before pushing those changes to production. If their in-house team wishes to develop or enhance the application without our assistance, I recommend that they review the Vulnerability Assessment Report upon each build, investigate all of the found vulnerabilities, and address them.