****

# 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 5](#_Toc33111307)

[3. Deploy Cipher 5](#_Toc33111308)

[4. Secure Communications 6](#_Toc33111309)

[5. Secondary Testing 7](#_Toc33111310)

[6. Functional Testing 9](#_Toc33111311)

[7. Summary 11](#_Toc33111312)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **02/19/2021** | **Jon Frodin** | **This report documents the security status of the application as well as the refactored code review. This includes dependency checks prior to refactoring as well as after refactoring. Also included is a dependency check performed after suppressing false positives. A broad overview of encryption and security is included. This report documents the changes and their impact on security.** |

## 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

Jon Frodin

## 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.

In developing this system, I chose the SHA-256 encryption algorithm cipher. This is because the SHA-256 is a 256-bit encryption algorithm. Without getting into too much detail, a 256-bit encryption algorithm provides protection against collisions (two different values mapping to the same encrypted value) and is too hard to crack, even for a supercomputer, in a human lifetime. In order to crack a 256-bit encryption it would take 3.31\*10^56 years.[[1]](#footnote-1) Hash functions are used in encryption algorithms, including the one I used. These functions transform your readable data into non-readable data and this process is one-way. Once data is hashed, it cannot really be retrieved without access to the key.[[2]](#footnote-2) The hash function in this cipher takes any input and outputs a 64-digit long string of basically random text. Modern computers can utilize these algorithms without too much computer power, but it would take something extremely advanced and fast to practically break the algorithm.

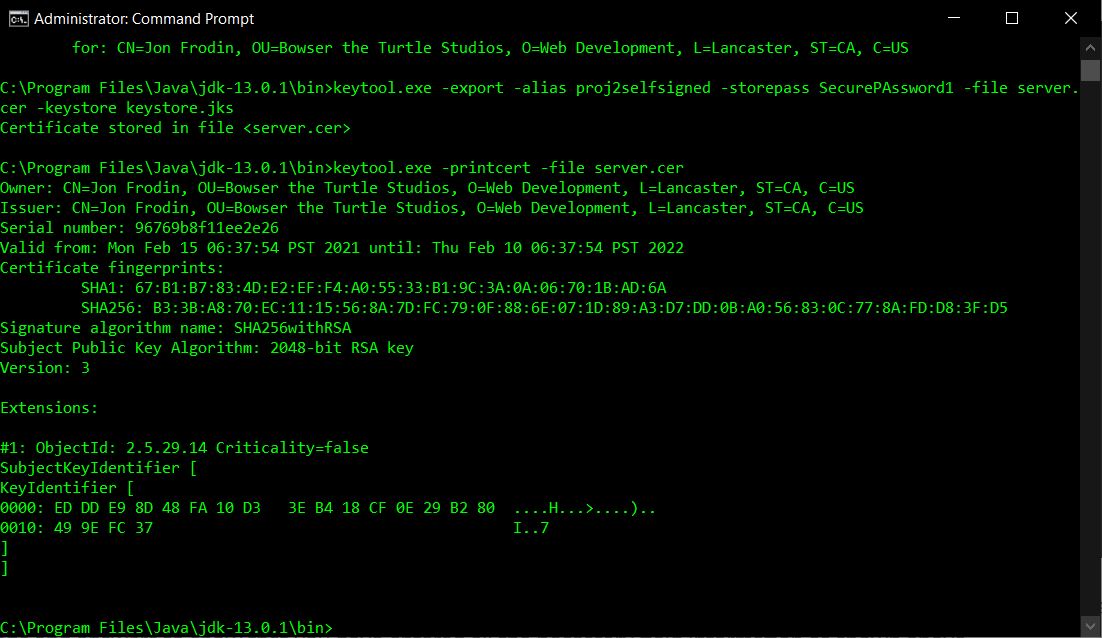
Two major types of keys exist in cryptography. The first is symmetric. Symmetric cryptography uses a single key to encrypt and decrypt information. That key must be kept in a secure location because if it were to be lost, any data encrypted with that key would be irretrievable. These work by using extremely large prime numbers to unlock and lock data.[[3]](#footnote-3) Then, there is also non-symmetric key cryptography which works in a different manner. Non-symmetric keys use one key to encrypt data, and that key is generally public-facing. That key is used to encrypt data, and then a private key is used to decrypt that same data. This also utilizes random number generation to ensure that the keys do not overlap, or at least are extremely unlikely to overlap. Furthermore, the keys have to be independent of one another. Meaning, someone having access to the public key should not give that person any way whatsoever to derive the private key.

Encryption has a very long history. People have been looking to hide the content of their messages from prying eyes for quite some time. While the methods for disguising the content of messages is very interesting, we are here to discuss modern cryptography. This has its roots in the Caesar Box and the Vigenere Cipher, followed by the Enigma machine from World War II.[[4]](#footnote-4) IBM created the “Lucifer” encryption available for civilian use all the way back in 1971.[[5]](#footnote-5) A rather ominous name due to the association with rebellion. The transportation of a secret may have been better represented by a project named Prometheus, in my opinion. Non-symmetric cryptography was also first developed in the 1970s and has exploded in complexity and private and public use since then. The days of 56-bit encryption are long gone for anyone who cares about security. Modern algorithms are far more robust having 256-bit as the current standard of the day, with options up to 2048-bits.[[6]](#footnote-6)

## 2. Certificate Generation

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

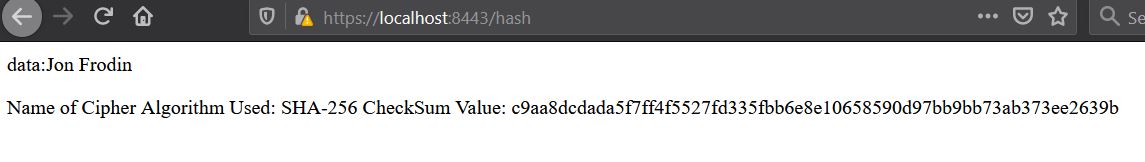
* To demonstrate that the keys were effectively generated, export your certificates (CER file) and submit a screenshot of the CER file below.



## 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.

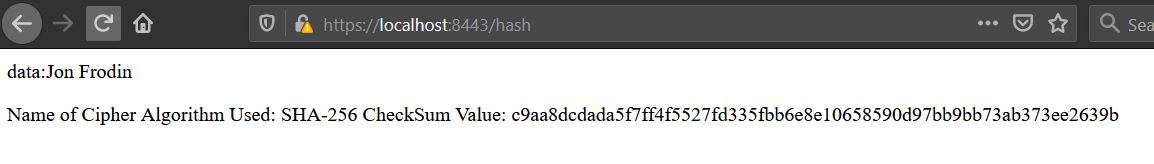
* Insert a screenshot below of the checksum verification. The screenshot must show your name and a unique data string that has been created.

As you can see, the data has been encrypted and the checksum value displays on the browser when the correct navigation path is entered.

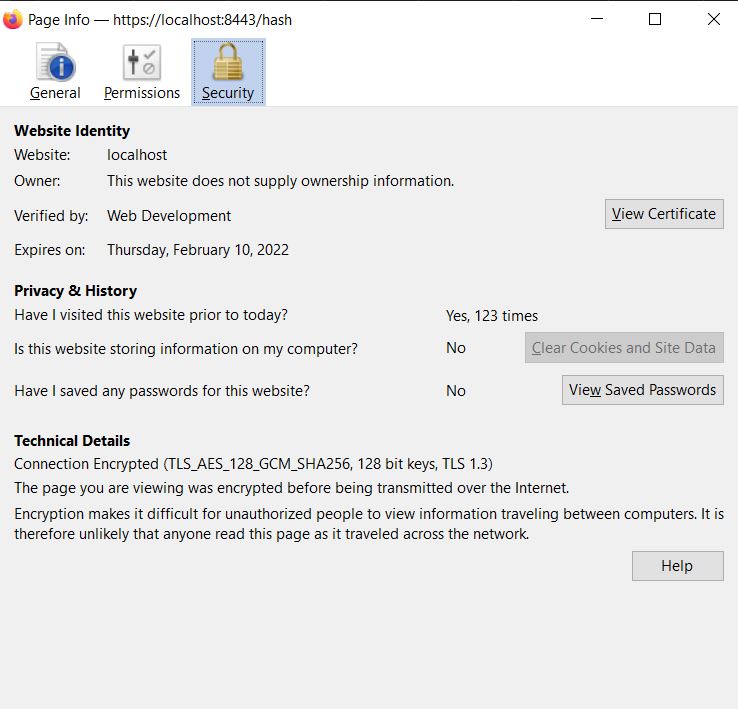
## 4. Secure Communications

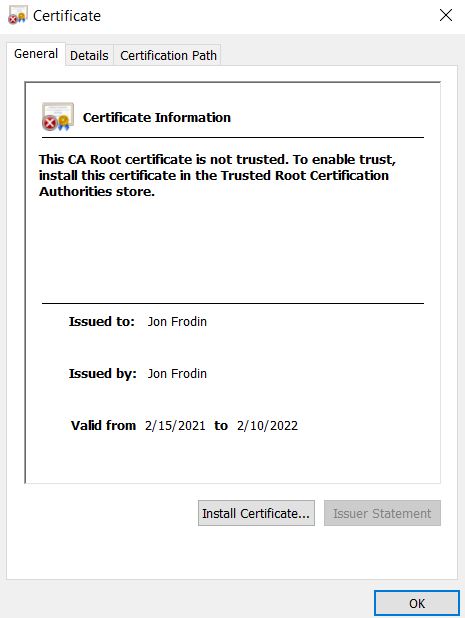
Refactor the code to convert HTTP to the HTTPS protocol. Compile and run the refactored code to verify secure communication by typing **https://localhost:8443/hash** in a new browser window to demonstrate that the secure communication works successfully.

* Insert a screenshot below of the web browser that shows a secure webpage.



As you can see, the padlock demonstrating a secure connection has a warning on it. Again, this is simply because I am using a self-signed certificate. When you look at the details, you can see that the connection was encrypted via the SHA256 standard with TLS 1.3.





## 5. Secondary Testing

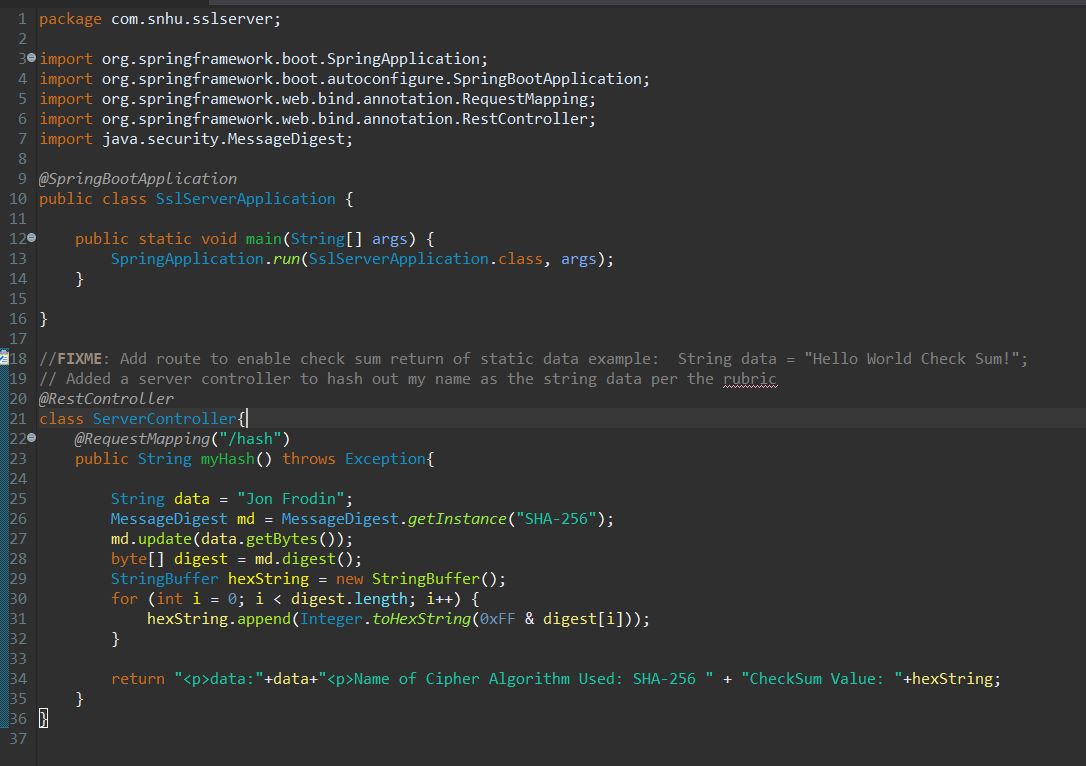
Complete a secondary static testing of the refactored code using the dependency check tool to ensure code complies with software security enhancements. You only need to focus on the code you have added as part of the refactoring. Complete the dependency check and review the output to ensure you did not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

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

Here is the dependency check before refactoring the code:



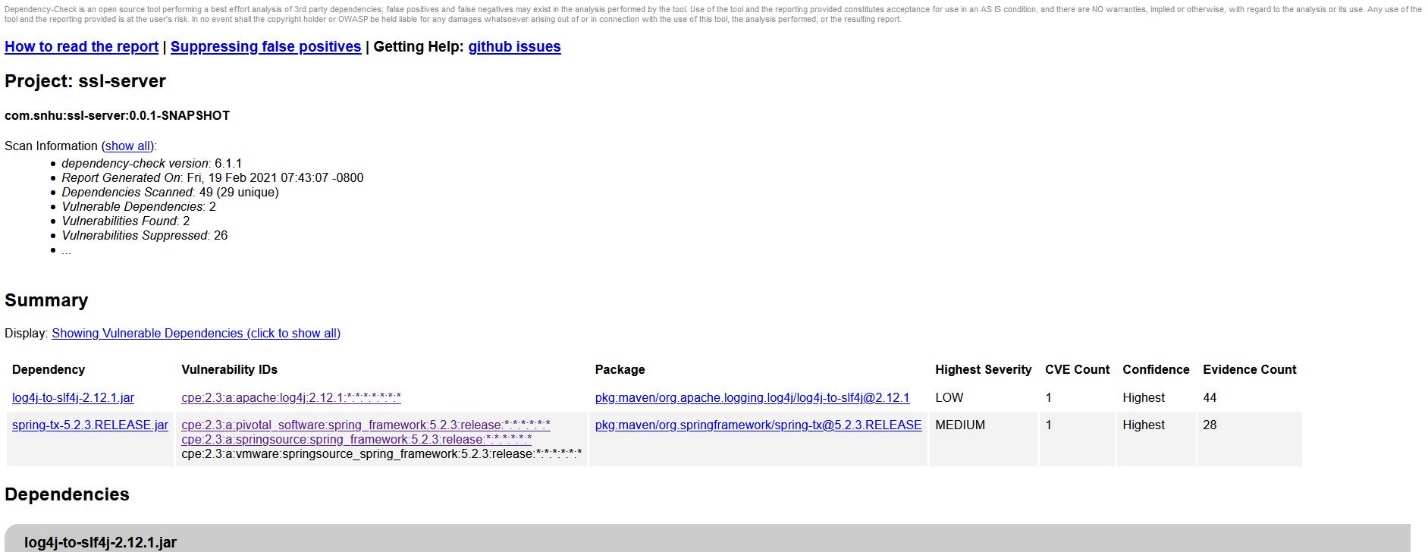
This is the refactored code:



This is the dependency check after refactoring the code:



As you can see, additional security issues were not found after refactoring the code, so my refactoring did not introduce any security vulnerabilities. Furthermore, though it was not necessarily required in this project, I went ahead and reviewed for false positives according to the methods outlined in Module Six of this course. After suppressing the false positives, the results showed the following.

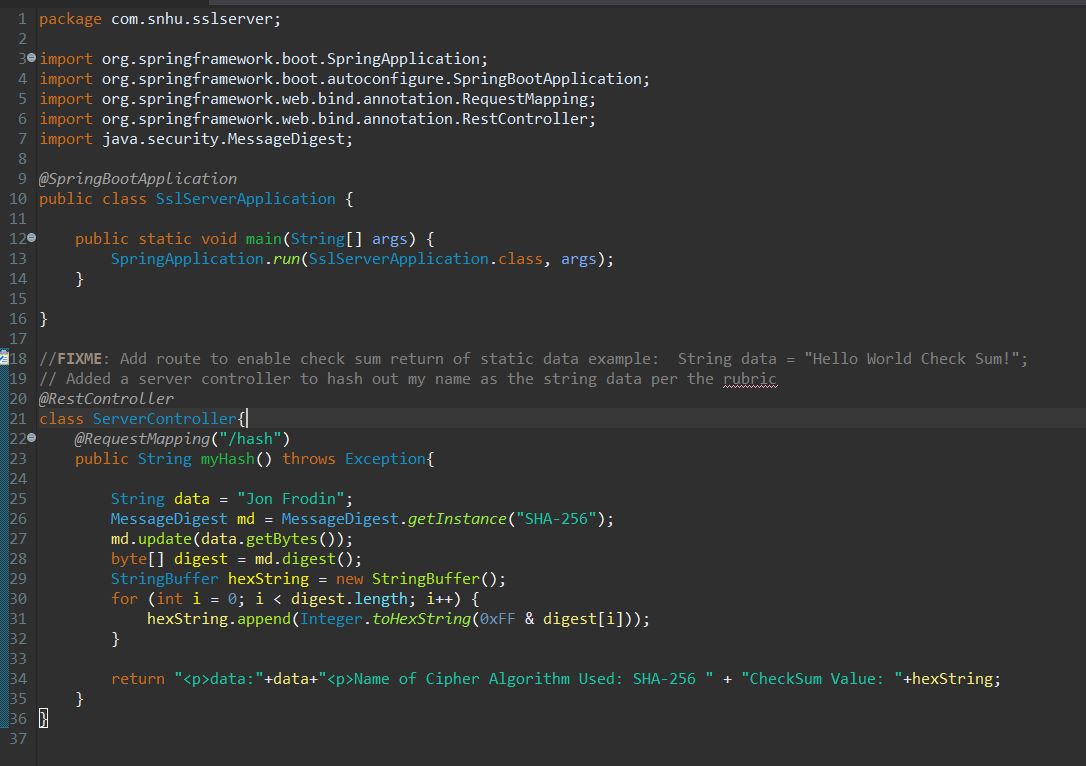


All three of these dependency checks are available as HTML files in the project submission for more detailed review if desired.

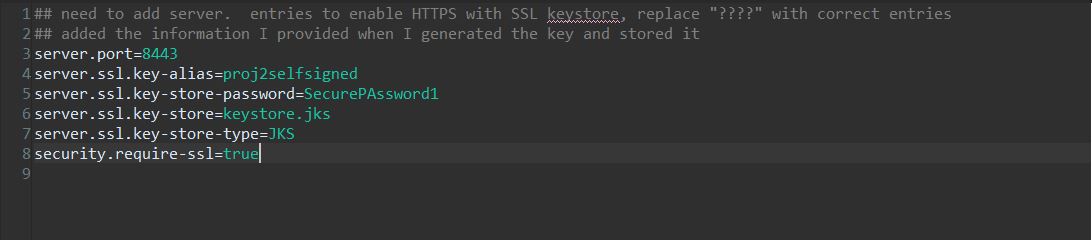
## 6. Functional Testing

Identify syntactical, logical, and security vulnerabilities for the software application by manually reviewing code.

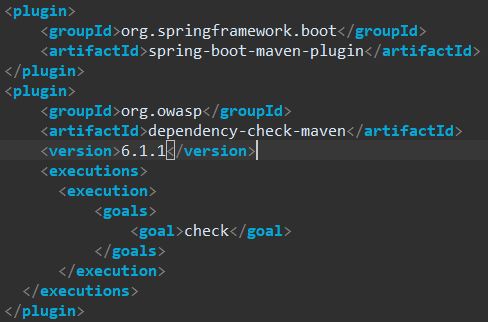
* Complete this functional testing and include a screenshot below of the refactored code executed without errors.



The code has been executed with no errors. Furthermore, I will attach a screenshot of the application.properties file for review. There are also no errors in that.



In the pom.xml file, I only updated the maven dependency check to the latest version, but also no errors there.



## 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.

In this project, I addressed cryptography, server/client, and code quality mostly. Code quality was addressed bye accounting for security in coding practices and patterns. I implemented a secure, 256-bit hashing algorithm in the code to protect sensitive data. The data, in this case, was just my name, but it could be anything. In the case of Artemis Financial, it could be financial plans, estimates, corporate documents, or client information. This data is secured by using encryption before being stored. Client/server was also addressed in this project by integrating a security certificate and enforcing a TLS connection to the web application. TLS secures the connection itself and the communication between the client and the server. Encryption was used in both the data and the TLS connection to provide secure communication. Finally, all code was reviewed and inspected for errors or vulnerabilities caused either by my code or by dependencies.

The first layer of security that I added for the client was an SSL certificate. This certificate ensures that the site is secure and that the users of the site know that their communication is encrypted. This ads value for Artemis Financial in the form of user trust as well as in data and communication security. The second layer of security that I added was the SHA-256 encryption algorithm included. It is not enough to encrypt a communication session using TLS. You also have to encrypt the data itself. This adds security because if someone were able to get access to the server where data is being stored, they would have access to very sensitive documents. When you encrypt the data then you ensure that if anyone does steal the information on the server, they will not be able to decipher what they have taken.

The final layer of security that I incorporated into this application was HTTPS enforcement. This ensures that the session cannot be performed over normal HTTP. If the user wants to interact with the site, it must be in an encrypted session to protect both the server and the client. This adds value because it does not give old browsers the option of a poorly secured connection which can add to our ability to prevent things like man-in-the-middle attacks.

Best practices for maintaining the current security of the software application involve regular dependency checks to look out for new vulnerabilities that are being discovered every day, secure coding for adding all the other modules needed, input validation for any inputs taken from any users, building a solid API that allows specific things to be done, but also makes clear to the client what to do and how to do it, and regular code reviews to ensure any changes made are checked for any errors or unforeseen interactions with currently existing code.

These can be achieved in many ways and to a degree commensurate with the needs of Artemis Financial. For example, if a code review in the future shows that SHA-256 is no longer a viable security option, then they can upgrade it to RSA-2048 if needed, or whatever standard is out at that time. Input validation can be performed by only allowing specific types of input and whitelisting available inputs rather than trying to blacklist every new threat that comes up. Regular dependency checks can highlight any new threats discovered by the NIST or other bodies that check code for vulnerabilities. If any new threats are found, they can be mitigated using recommendations on the reporting sites, or by switching to a module that does not need the dependency. Secure coding should always be practiced and going forward secure coding will have to be implemented. Creation of the API will have to be done carefully so that users cannot perform any actions not explicitly authorized for their level of access. Taking these best practices into account, the application should operate smoothly for years to come and provide Artemis Financial with all of the functionality they need.

1. <https://www.clickssl.net/blog/128-bit-ssl-encryption-vs-256-bit-ssl-encryption> [↑](#footnote-ref-1)
2. <https://sectigostore.com/blog/hash-function-in-cryptography-how-does-it-work/> [↑](#footnote-ref-2)
3. <https://csrc.nist.gov/csrc/media/events/random-number-generation-workshop-2004/documents/developmenthistory.pdf> [↑](#footnote-ref-3)
4. <https://ghostvolt.com/articles/cryptography_20th_centuary.html#:~:text=The%20RSA%20algorithm%20for%20asy> [↑](#footnote-ref-4)
5. *Ibid.* [↑](#footnote-ref-5)
6. <https://searchsecurity.techtarget.com/answer/From-1024-to-2048-bit-The-security-effect-of-encryption-key-length> [↑](#footnote-ref-6)