

# Practices for Secure Software Report

Table of Contents

[Document Revision History 3](#_Toc102040754)

[Client 3](#_Toc102040755)

[Instructions 3](#_Toc102040756)

[Developer 4](#_Toc102040757)

[1. Algorithm Cipher 4](#_Toc102040758)

[2. Certificate Generation 4](#_Toc102040759)

[3. Deploy Cipher 4](#_Toc102040760)

[4. Secure Communications 4](#_Toc102040761)

[5. Secondary Testing 4](#_Toc102040762)

[6. Functional Testing 4](#_Toc102040763)

[7. Summary 4](#_Toc102040764)

[8. Industry Standard Best Practices 4](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **06-19-2025** | **Kayne Kirk** |  |

## Client



## Instructions

Submit this completed practices for secure software report. Replace the bracketed text with the relevant information. You must document your process for writing secure communications and refactoring code that complies with software security testing protocols.

* Respond to the steps outlined below and include your findings.
* Respond using your own words. You may also choose to include images or supporting materials. If you include them, make certain to insert them in all the relevant locations in the document.
* Refer to the Project Two Guidelines and Rubric for more detailed instructions about each section of the template.

## Developer

Kayne Kirk

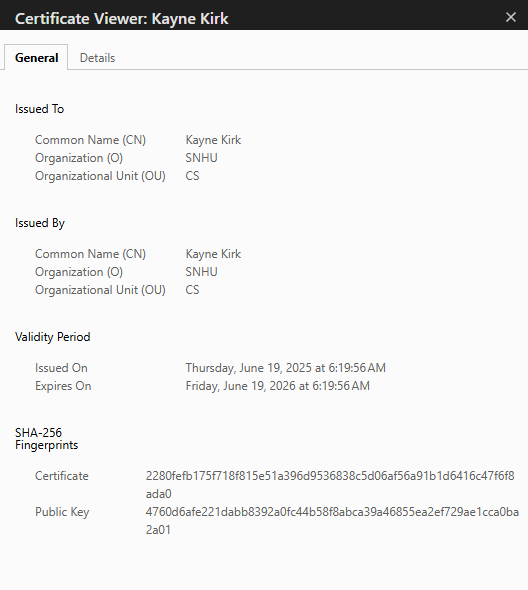
## Algorithm Cipher

The encryption algorithm I would recommend is AES-256, or the Advanced Encryption Standard with a 256-bit key. AES is largely considered both the most secure and most efficient encryption option for general usecases, becoming standardized by National Institute of Standards and Technology in 2001. The 256-bit key allows for high performance while still defending against all known malicious activity, down to brute-force hacking attempts. AES is a *symmetric key*, so the encryption and decryption process is facilitated by a single key, but its use of random numbers makes it so that even the same inputs produce different ciphers (important for deterring pattern recognition attacks). This combination of high-performance and high-security makes it a great fit for handling clients’ financial data, and even (as we see in the project) supports communication protocols like SSL which prioritize security and mesh well with our existing Spring Boot framework.

In our project, we employ SHA-256 for our checksum verification. SHA-256 produces 256-bit fixed-length output, which is great for generating checksums as we have done here.

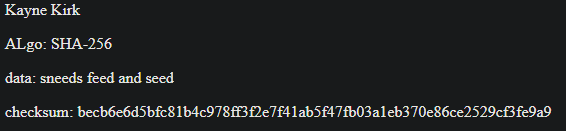
## Certificate Generation

Insert a screenshot below of the CER file.



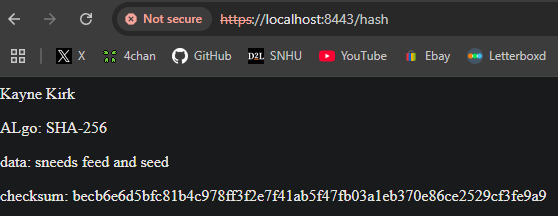
## Deploy Cipher

Insert a screenshot below of the checksum verification.



## Secure Communications

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

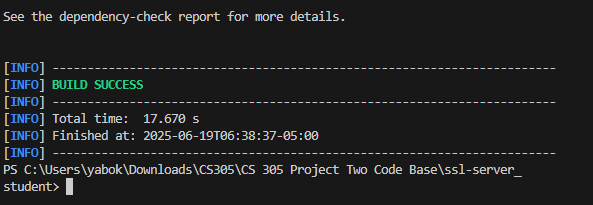


While this says *not secure*, we still rerouted the HTTP traffic from the default 8080 port to the HTTPS 8443 port and loaded SSL. Self-signed certificates are simply not secure as anyone can make them (such as malicious actors), per SSLShopper (2010)

**5**. **Secondary Testing**

Insert screenshots below of the refactored code executed without errors and the dependency-check report.

First, we built the project using *mvn verify*:



And found a number of vulnerabilities in the dependency check report:



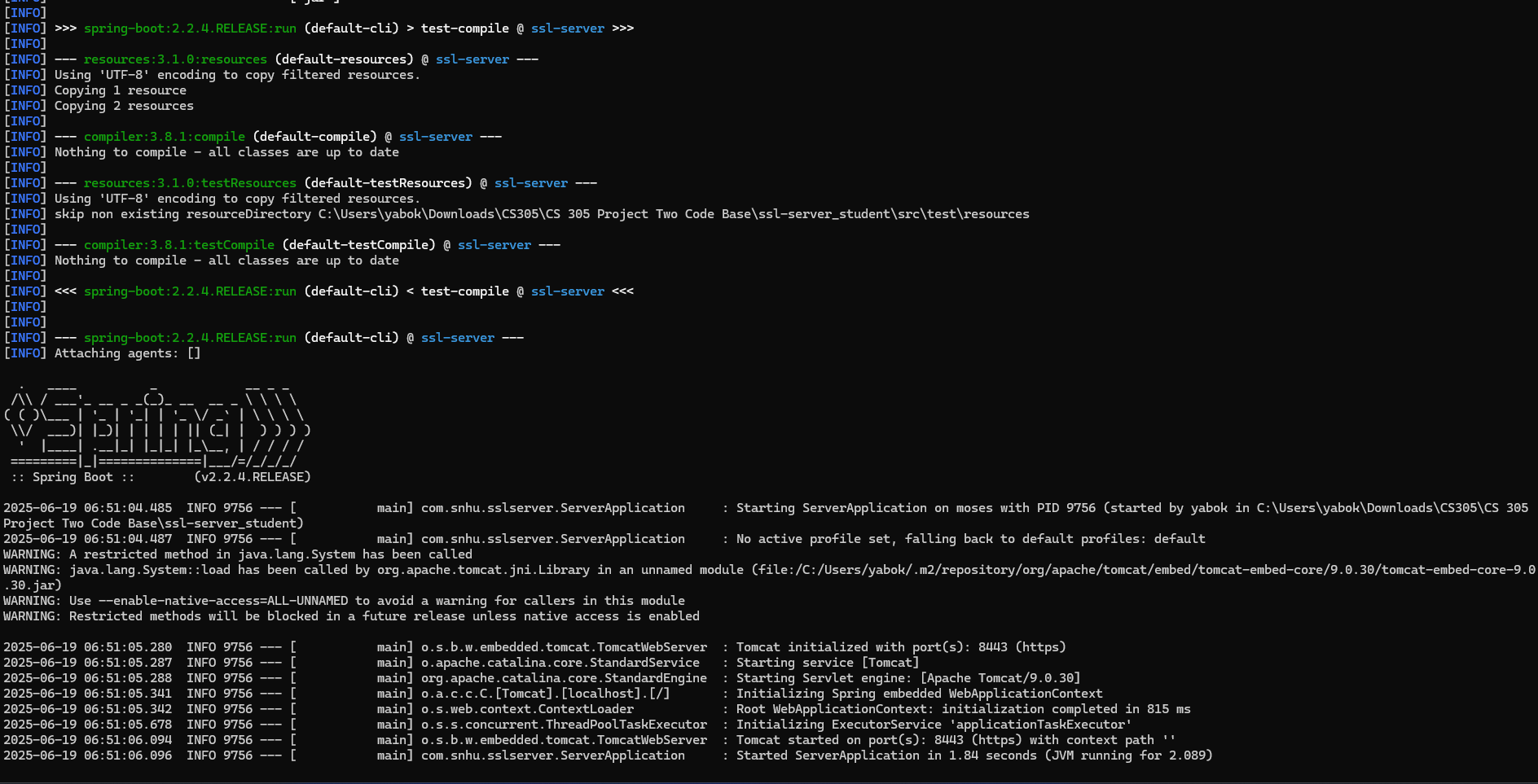
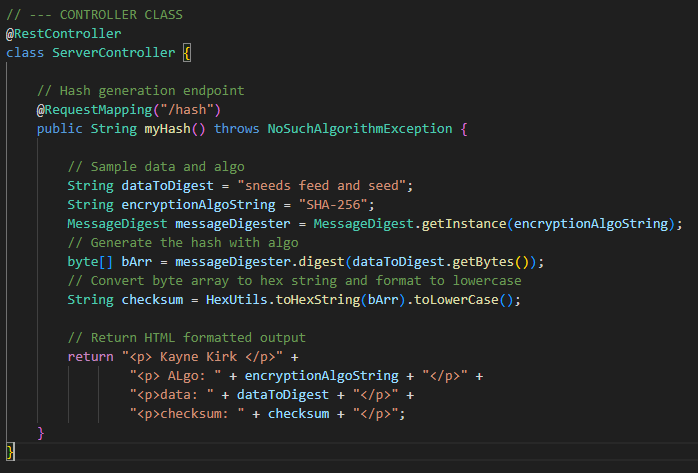
Compare this to the scan *prior* to putting in any new code at all (I re-downloaded the codebase and ran this on a fresh project):



(No new vulnerabilities!)

## Functional Testing

Insert a screenshot below of the refactored code executed without errors.



(I did this earlier, but did it again just to comply with the prompt. Hope this is what you mean.)

The code compiles and runs without any errors, so it is syntactically valid, as it would not compile if that were not the case. The */hash* endpoint uses MessageDigest on an algorithm string (for ease of switching out later, rather than hard-coding [I don’t know when that variable will be reused, if at all, but still]). This gets performed on the input string, which results the hash being represented as a hexadecimal string that we output (formatted to lowercase – stylistic preference) and returned as some HTML elements to show up in the browser window. Our project is about as basic as a Spring Boot application as you can get, adhering to all controller conventions with *@RestController* and *@RequestMapping* annotations.

Security-wise, we have no real vulnerabilities here outside of potentially upgrading Spring Boot. There is no user input that anybody can inject anything into, and our implementation of the secret keys does not reveal sensitive information from either errors (as only NoSuchAlgorithm can be thrown, and there is no user-input that can manipulate this message) or user data (as none is input). Furthermore, even if that *were* the case, we are not in any hot water just yet, as SHA-256 is non-reversible and there are no known vulnerabilities.

## Summary

Our refractored code has enhanced the software’s security by adding cryptographic hashing so that hard-coded data (and, in the future, secret keys, etc) are not stored and/or communicated to other parts of the system in plaintext, which can be easily intercepted for malicious purposes. We *Analyzed* and *Remediated* the insecure structure of the program and simply added a cryptographic industry standard. The checksum itself is important for ensuring data integrity and verification, as non-reversible outputs can expose sensitive architecture to bad actors.

We *Mitigated* and *Validated* both present and future potential vulnerabilities, like configuring the HTTPS through a self-signed certificate (that should be validated later to be trusted in production) that makes sure data is encrypted in client-server communication. While we still have no user input (which is a big plus to security, still), our HTTPS configuration confirms that any *future* endpoints will automatically inherit this data transit encryption.

7. Industry Standard Best Practices

Our complicance with best practices includes much of what has been discussed thus far – HTTPS configuration, incorporation of SHA-256, and aiming for minimal added dependencies. Our attack surface did not increase with the refractored code, as our OWASP Dependency-Check confirmed there were not new vulnerabilities that were introduced. We have poised the application to comply with regulation while setting a foundation for long-term security and integrity by being attentive to avoiding new attack vectors at such an early stage of development. .

References

SSLShopper. (2010, October 30). *How to create a self signed certificate using Java Keytool*. [h ttps://www.sslshopper.com/article-how-to-create-a-self-signed-certificate-using-java- keytool.html](https://www.sslshopper.com/article-how-to-create-a-self-signed-certificate-using-java-keytool.html)