****

# Practices for Secure Software Report

**Table of Contents**

[**Document Revision History 3**](#_4y6ax136a5m6)

[**Client 3**](#_wms8sy8s98jy)

[**Instructions 3**](#_k4147n144v87)

[**Developer 4**](#_gise7enpbzcn)

[**1. Algorithm Cipher 4**](#_ng16ovo0qksx)

[**2. Certificate Generation 4**](#_6spx33nblbdk)

[**3. Deploy Cipher 4**](#_ce5kuhmjld72)

[**4. Secure Communications 4**](#_1msctupy0d0)

[**5. Secondary Testing 4**](#_s091ansyfyee)

[**6. Functional Testing 4**](#_mx0ckr8zapds)

[**7. Summary 4**](#_iubj0ps9dob2)

[**8. Industry Standard Best Practices 4**](#_f0afnds4a5ty)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **4/18/2025** | **Christopher Ellis** |  |

## 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

[Insert your name here.]

## Algorithm Cipher

Artemis Financial has requested we recommend a cryptographic algorithm to use for file encryption in their application. Our recommendation is the AES-256 cipher algorithm, using the GCM mode to ensure data authenticity (OWASP n.d.). This algorithm was chosen from a list of available cipher algorithms in the Java platform standard list of cipher algorithms (Oracle 2017).

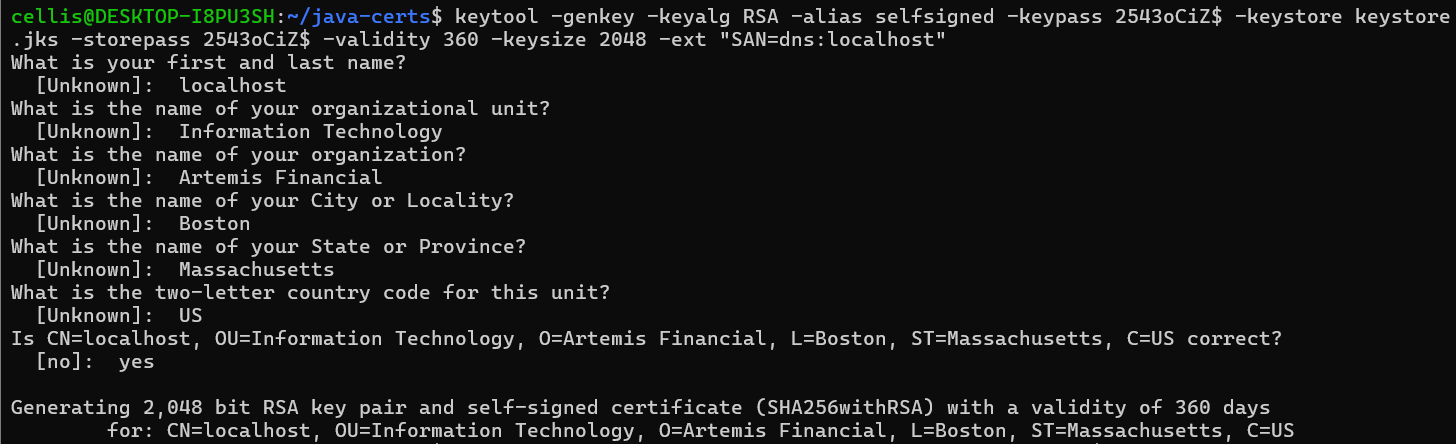
The OWASP recommendations for encryption algorithms was compared to the list given by the Java platform in order to select an algorithm that meets the needs of Artemis Financial. Specifically we first eliminated any algorithms not suitable for file encryption such as key wrapping algorithms like AESWrap, along with all of the algorithms that have been deemed insecure at this point in time by either OWASP or Microsoft as recommended in their assessment of weak ciphers (Microsoft n.d.). AES is the modern recommended standard by OWASP and was chosen as the best choice due to this recommendation from the top choices of AES and RSA. RSA, while meeting most of the needs, depends on the length of keys generated to ensure security, adding overhead to the administrative management of the encryption process.

Key length is optional with the AES algorithm, the choices being 128 and 256 bit. The 256 bit key length was chosen for greater security, with consideration that there will be some computational overhead to the encryption process for using longer keys, which was a tradeoff made for the added security as the overhead should be minimal in the scenario of encrypting files, which are only encrypted/decrypted when operated on which is infrequently. The larger key size specifically helps thwart attacks like brute force as the key’s length makes each prediction significantly longer along with adding many more necessary predictions, to the point the likelihood of it succeeding is negligible to non-existent (NIST 2022). AES is a symmetric encryption algorithm, meaning the same key is used to both encrypt and decrypt the data, which is perfectly suitable for something like file encryption as the keys do not need to be distributed as with public key cryptographic systems for things like certificates or SSH.

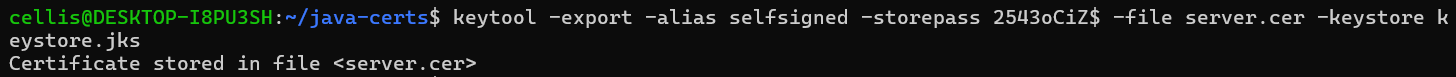
## Certificate Generation

Insert a screenshot below of the CER file.

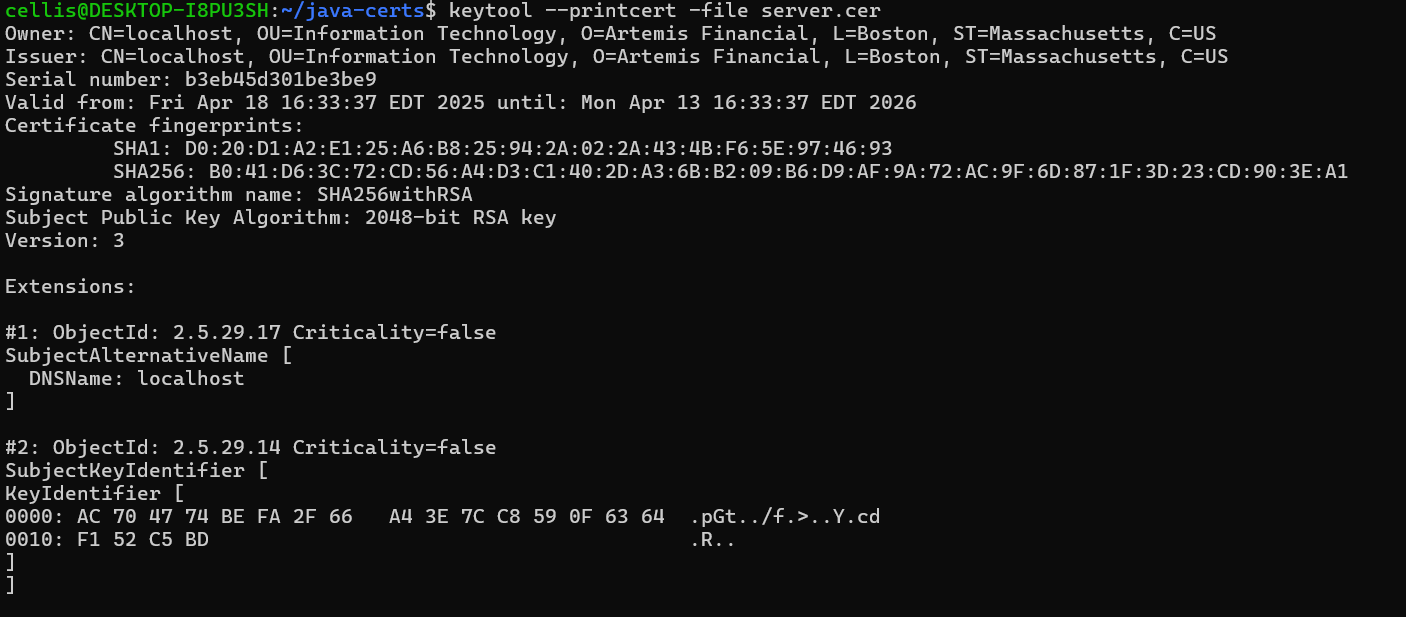
Key generation



Export

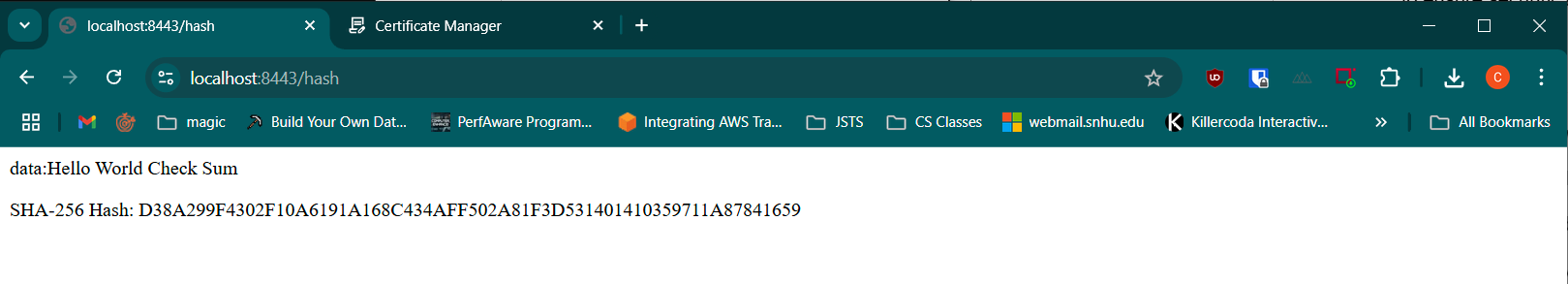


Cert output



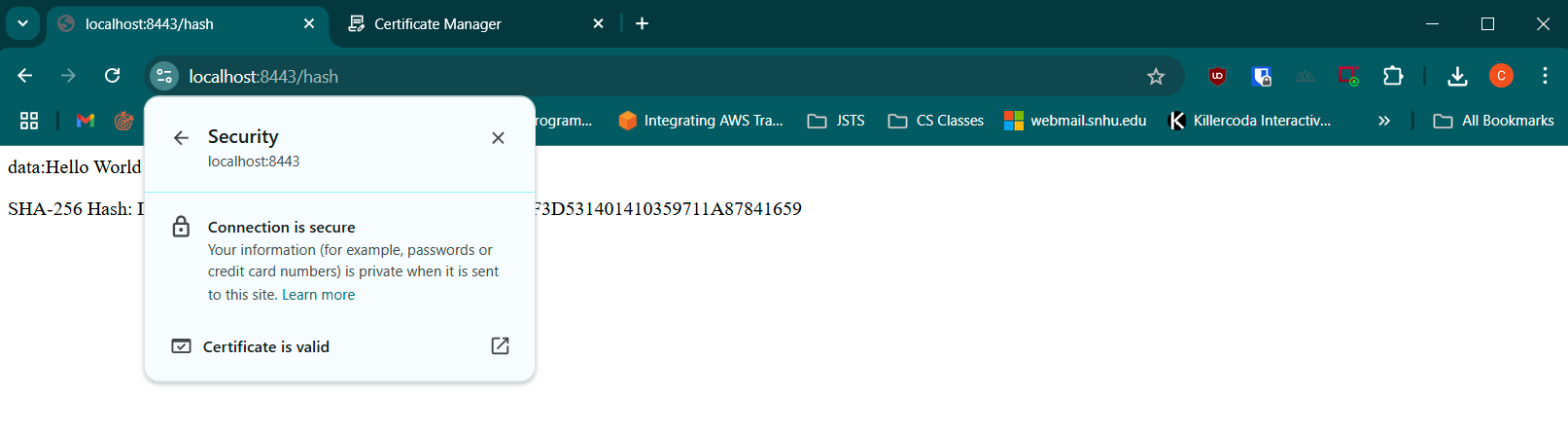
## Deploy Cipher

Insert a screenshot below of the checksum verification.

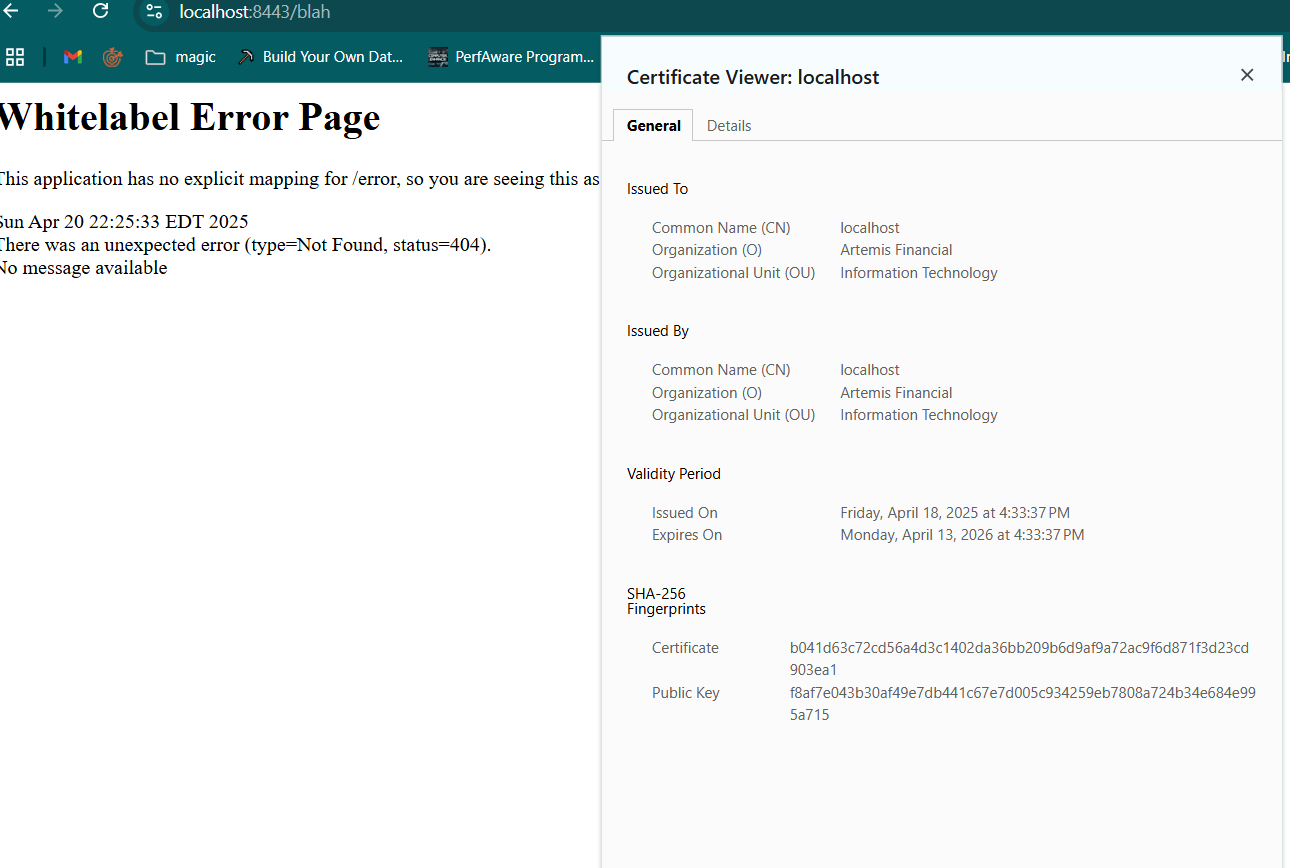
****

## Secure Communications

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

****

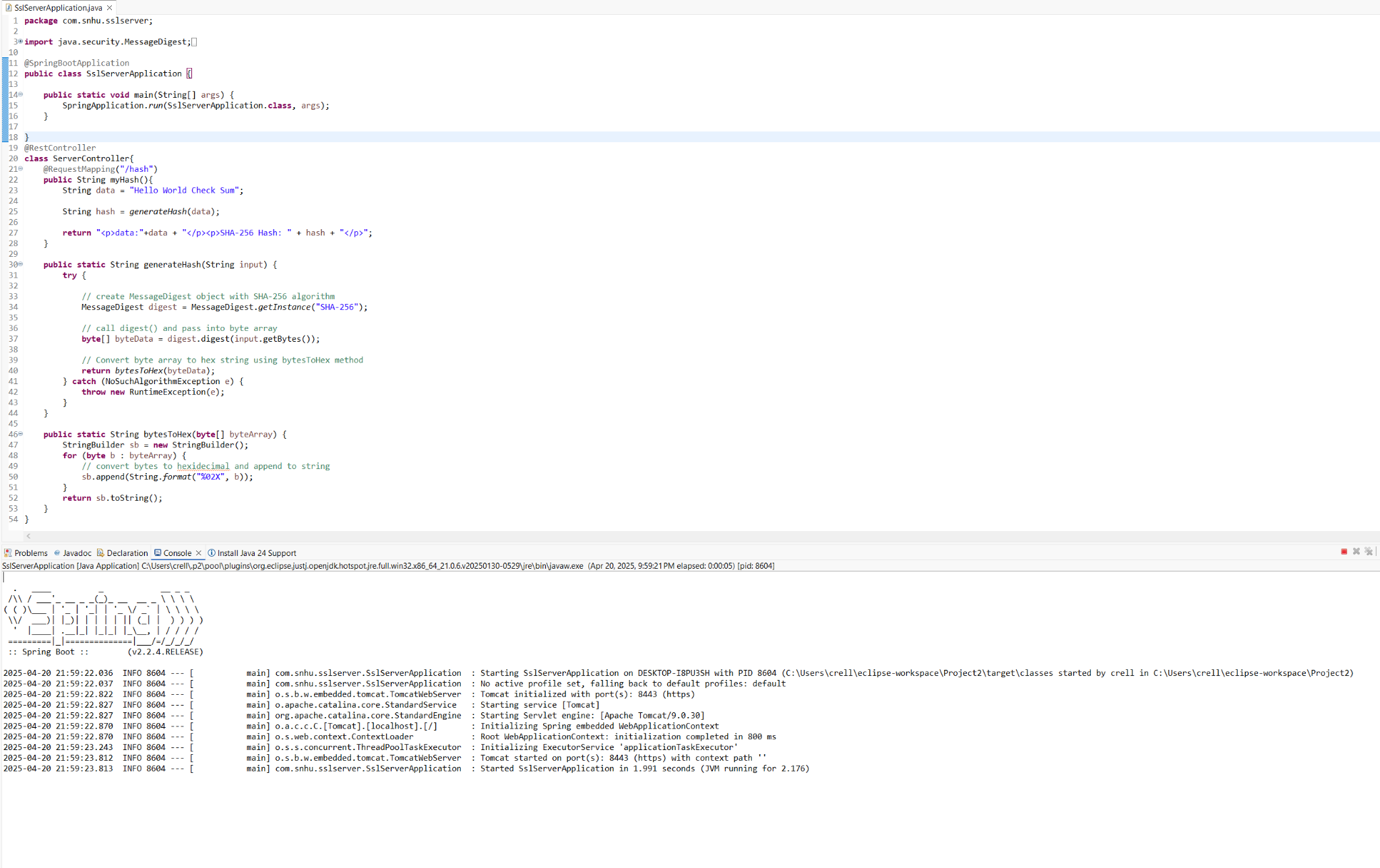
certificate data:

****

## Secondary Testing

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

Refactored code running without errors:



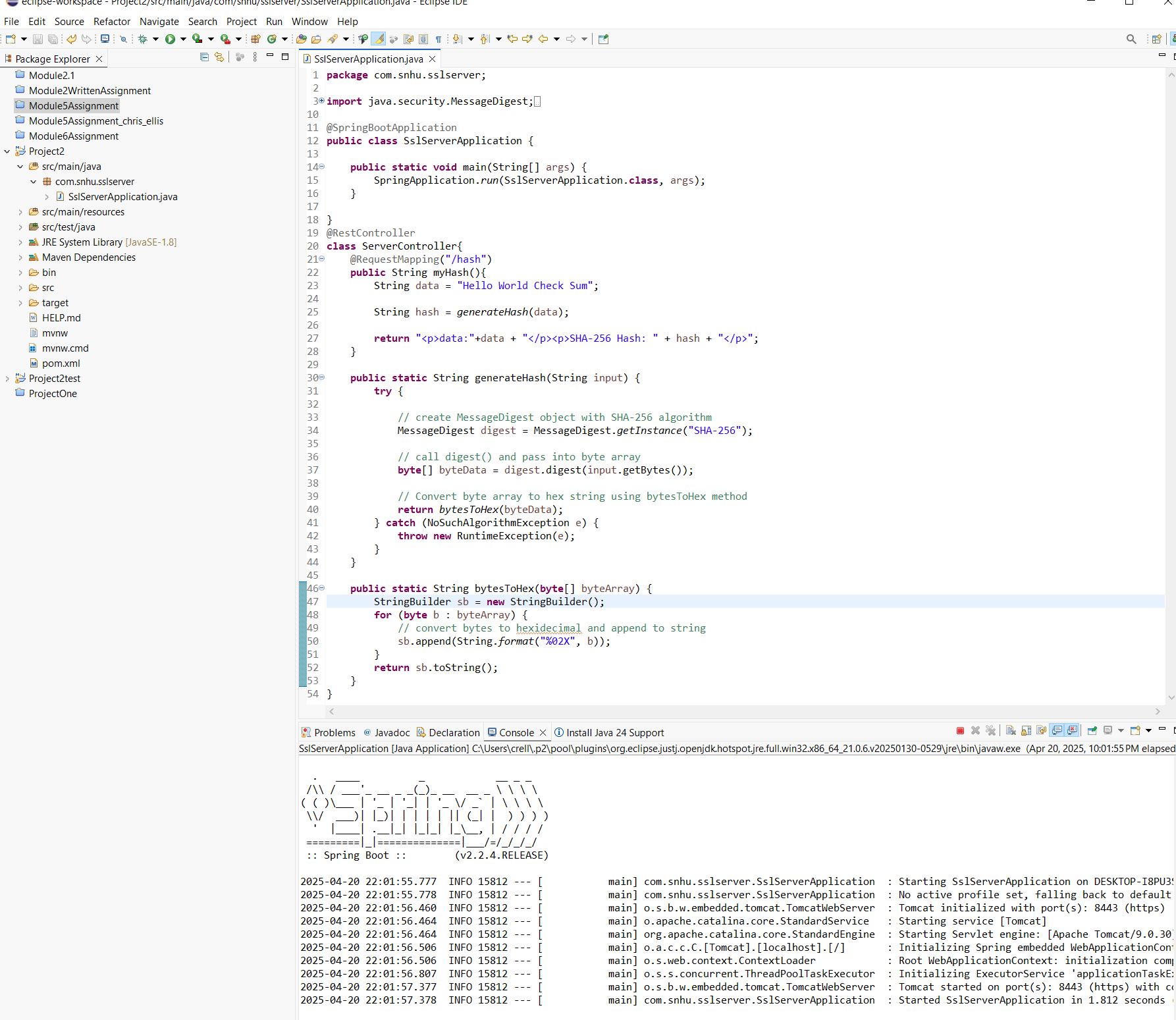
Dependency-Check report (this report shows vulnerabilities, but the number of dependencies and vulnerabilities is the same as before refactoring any code, just providing the SSL parameters to run the server)



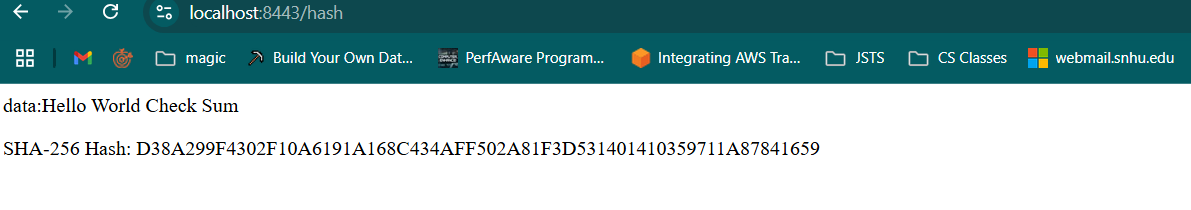
## Functional Testing

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

The below code is the refactored version of the code that provides a rest controller and an endpoint for hashing a static string using the previously specified algorithms and ciphers. The server serves this endpoint as a page /hash over a secure SslServerApplication using the generated keystore in order to secure the connection in transit in case user data were to be sent over the network to the application. There are some possible security vulnerabilities present in this code, however they are not deemed a major issue at this time as they are mainly handled by the environment we are running in and the fact that we are still in a testing phase. One of these issues is that the API endpoint for the hashing code does not perform any input validation, which is not deemed an issue as we don’t take in any user input and only use a static string passed from another function. However, this should be fixed now as leaving it in its current state means a refactoring of the code later could result in someone unintentionally using that function thinking it has input validation for user facing code that it does not. Another artifact of the IDE environment and testing is that the keystore password is directly set in the application.properties file. Since we are just running in the IDE and not on a server this is not really a vulnerability, but it is bad practice to store any credentials in a file that might be committed to a code repository, so we should refactor our code for production to read this value and any other sensitive values from environment variables or the code can pull them directly from a secrets management tool like Hashicorp’s Vault or Amazon Web Services Secrets Manager which both have an API applications can call to get a secret value. While errors are handled and we catch exceptions, only the whitelabel error page from Spring is setup, and our code has not been written to handle all errors at the top of the stack, meaning we could possibly leak data if an unknown/uncaught error occurred and the stack trace was dumped. Finally, our dependencies are out of date as shown in the static security scan, and we should update to the most recent dependencies before moving to production along with a plan to continuously evaluate dependencies on a specified basis and remediate any issues we find based on their criticality.



Below is the page running in the browser from this code:



## Summary

Referring to the vulnerability assessment process diagram, the likely areas we need to cover in this application are input validation, secure api interactions, encryption, secure error handling, and the general secure coding practice. As previously mentioned we are not explicitly validating input to the functions for hashing as the input is a static string in the code, but to ensure this is safe for users it should validate against an expected set of inputs. API transactions are secure as we do not have any erroneously public data, the methods on the server are public but we have no explicitly public variables, and all of the variables currently in the code are scoped inside the existing functions with no global variables.

Cryptography is important to check and get right, as shown by the previous evidence in this document the server is appropriately serving HTTPS on port 8443, and the certificate is valid and secure for the localhost hostname. We will need to generate production certificates for the domain and properly secure the credentials related to the certificate authority, or use a public certificate authority like Let’s Encrypt. Our hashing algorithm is SHA-256, which is considered secure and is recommended by OWASP for operations like file hashing (OWASP n.d.).

Our application is not distributed and has no database so we do not need to consider distributed composing.

Errors are caught in the application and it properly uses try/catch for operations that need it, ensuring we do not have our errors propagate to a stack dump when we know what the intended errors from a function are. There are additional practices here we should consider, like custom error pages, and having generic errors caught at the end of evaluation to ensure stack traces are not leaked to error pages at all.

Finally we have generally followed secure coding practices from the beginning, choosing secure algorithms, and building the application to not expose information or variables as public by default. We are not taking in user input through the API that needs to be sanitized, and we are serving our application over HTTPS with a valid certificate.

Secure data structures would be followed but currently we are not using any custom data structures to store data from the API that we would need to secure, but we would follow through here by continuing to preserve variable access to only pieces of data that need public access, as well as using methods to internally access the data with getters and setters where appropriate.

My process for adding layers of security was to begin at the bottom of the software stack and build up, considering the elements in the vulnerability assessment diagram as I went in order to maximize security. This also ensures that when working on a higher layer that we consider how that layer interacts with those below it in order to avoid missing pieces if we were to build things in a less systematic way. Beginning with the variables or data structures, and building our classes and methods on top of those is how I went through the actual development process. Knowing that I would need algorithms for file encryption I did pre-research to choose the most appropriate algorithms for each component (TLS, file encryption, etc.) and followed best practices to implement those. After having a running prototype of the software I ran the static security scanning tool Dependency-CHeck from OWASP against the codebase to see what possible vulnerabilities might exist in the dependencies. The last step was to perform functional tests on the code to make sure everything worked and things functioned correctly, including the serving of HTTPS with a valid certificate in my browser, which I also performed on both Chrome and Firefox to make sure the behavior was consistent across at least two popular browsers.

## Industry Standard Best Practices

The industry standard for secure development is continuous security applied at every level of development through the software development lifecycle, along with continuously checking the software after it is running in production to ensure it remains secure and keep up to date on any vulnerabilities that are newly discovered in dependencies, the runtime environment, or nowadays the cloud provider that it runs on (Hornbeek 2019).

This process was mostly followed for this project, minus a few things that would be present in a development environment with multiple developers like continuous integration, code was developed from a secure architecture, algorithms were researched and chosen for the their security attributes in the specific application that they would be deployed in, and static security checks were run against the codebase in order to determine possible existing vulnerabilities in the other libraries used to make the code function correctly.

Static security scans are an important part of this process, along with dynamic testing and automated tools for running unit and integration tests, all of which help ensure an application remains secure even as code is refactored. Static scans also provide a baseline for vulnerabilities to check against in the dependency tree and reviewing them ensures that we can make the best choices for dependencies to ensure things are secure for users.

Applying these practices is good for a company’s overall wellbeing as it helps developers feel like they are taking user security seriously. I have encountered this in my own career and my current job at a very security focused healthcare startup that takes security very seriously allows me to worry about my area of expertise and securing that knowing that everyone else in the organization is doing their part to ensure things remain secure for the users, which is a value that I consider important.

**References**

OWASP. (2016 a). *M5: Insufficient cryptography*. OWASP. Retrieved from https://owasp.org/www-project-mobile-top-10/2016-risks/m5-insufficient-cryptography

OWASP. (n.d. b). *Cryptographic storage cheat sheet*. OWASP Cheat Sheet Series. Retrieved March 28, 2025, from https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic\_Storage\_Cheat\_Sheet.html

Oracle. (2017). *Java platform, standard edition security standard names*. Oracle Documentation. Retrieved March 28, 2025, from https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html

Rhodes, D. (2025, January 11). *SHA-256 Cryptographic Hash Algorithm*. Komodo Platform. <https://komodoplatform.com/en/academy/sha-256-algorithm/>

Microsoft. (n.d.). *Security assessment: Weak cipher* (Defender for Identity). Microsoft Learn. Retrieved March 29, 2025, from https://learn.microsoft.com/en-us/defender-for-identity/security-assessment-weak-cipher

National Institute of Standards and Technology. (2022). *FIPS 197: Advanced encryption standard (AES)* (Updated). <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf>

Hornbeek, M. (2019, January 8). *9 pillars of continuous security best practices*. DevOps.com. https://devops.com/9-pillars-of-continuous-security-best-practices/​DevOps.com