****

# 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** | **[Date]** | **[Amanda Dunphey]** |  |

## 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

[Amanda Dunphey]

## Algorithm Cipher

The app should need at least TLS 1.2 and like current AEAD cipher suites like TLS\_ECDHE\_RSA\_WITH\_AES\_256\_GCM\_SHA384 for transport security. For forward privacy, elliptic-curve Diffie–Hellman (ECDHE) is used. The server certificate should have SHA-384 signatures and RSA-2048 (or a stronger certificate). Turn off protocol downgrades and legacy/weak suites like CBC, 3DES, and RC4. In production, turn on HSTS so browsers always use HTTPS.

Instead of CBC, use AES in Galois/Counter Mode (AES-GCM) to secure data inside the app. Use 256-bit keys to encrypt (although 128-bit keys will also work), make a new 96-bit IV for each action, and store the 128-bit authentication tag that you get along with the ciphertext. Keys shouldn't come from hard-coded passwords; instead, use PBKDF2-HMAC-SHA-256 with a unique salt and a high iteration count to create them, or even better, use a dedicated KMS/secrets manager to keep track of them. These options are in line with what OWASP and NIST recommend. They offer verified encryption and security, protect forward secrecy, and lower the risk of padding, downgrade, and key-exposure attacks. Keys and certificates should be changed often, and secrets should not be in source control.

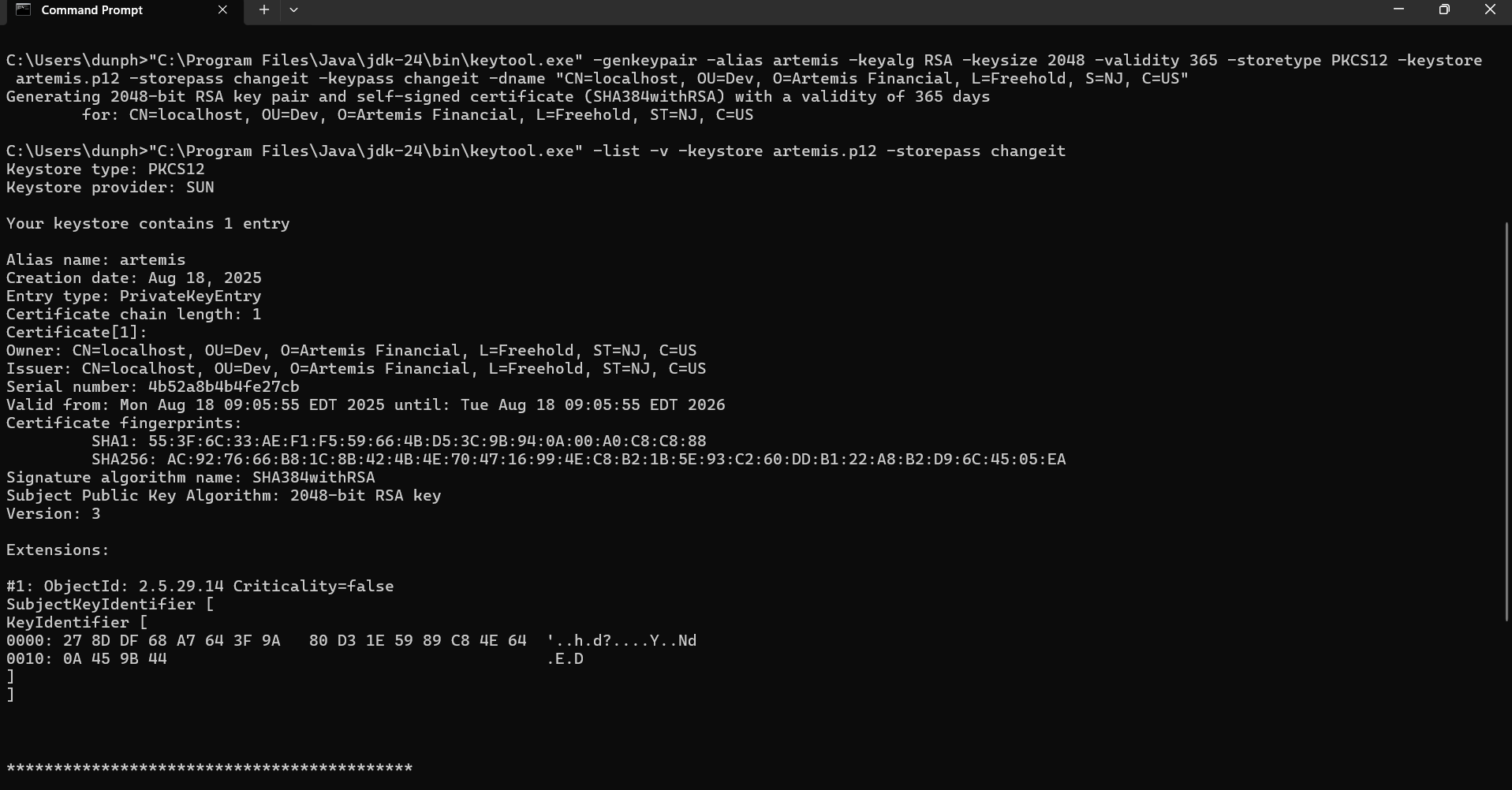
## Certificate Generation

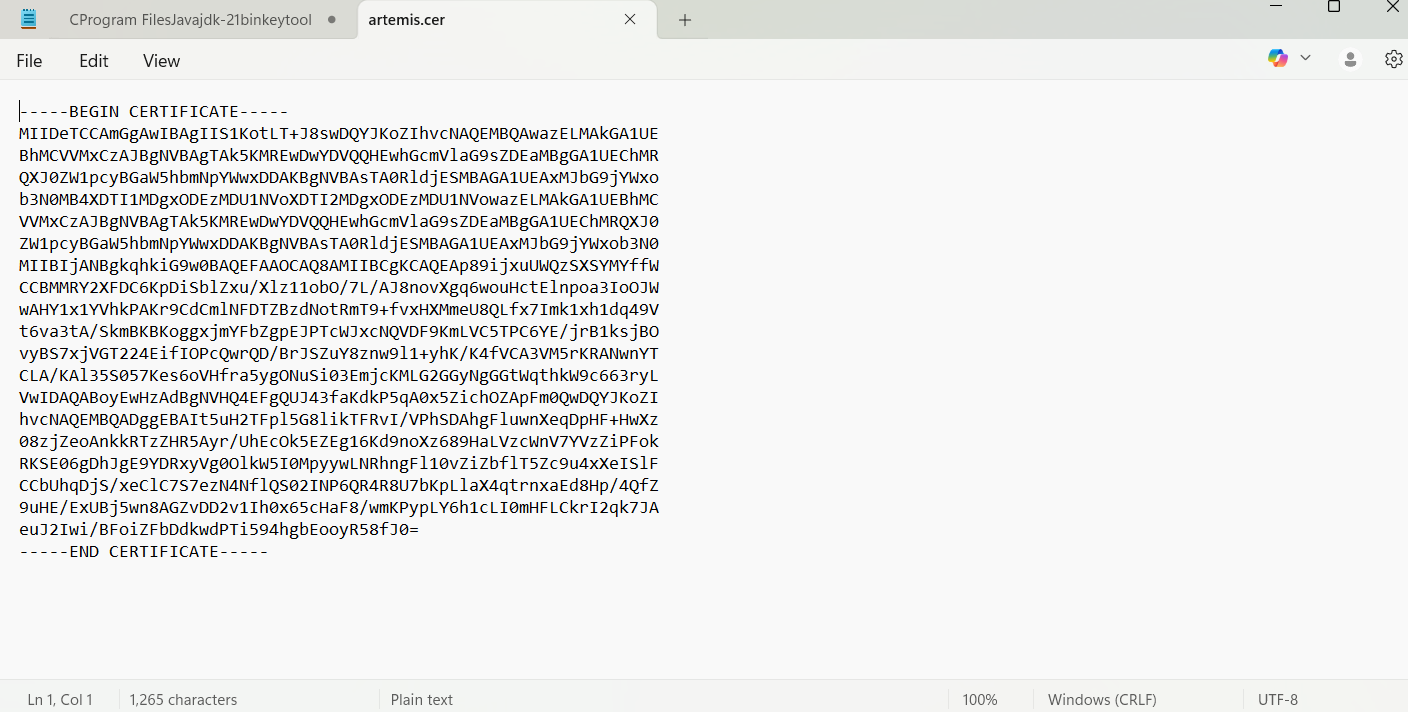
I used the keytool command to make a self-signed PKCS#12 certificate for a safe keystore. The first picture shows the successful result from the terminal, where the keystore file (keystore.p12) was made using RSA-2048 encryption that is good for one year. In this case, it shows that the app now has a valid certificate for making HTTPS calls.

The next picture shows the keystore.p12 file in the project folder along with the exported certificate file (artemis-local.cer). This makes sure that both the private key and the public certificate were made correctly and can be used in the Spring Boot application.

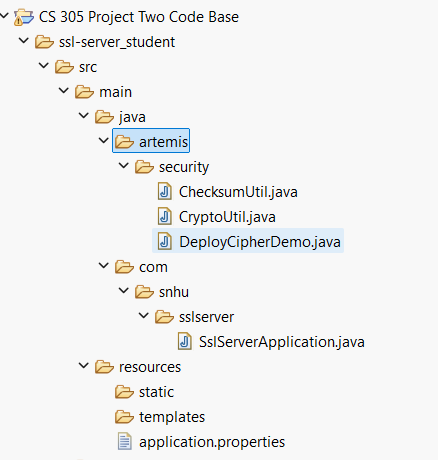
Running keytool -list -v showed me the fingerprint, serial number, and validity dates of the certificate, which I also wrote down. The SHA-256 fingerprint, which can be seen in the screenshot of this result, proves that the certificate is real and unique.

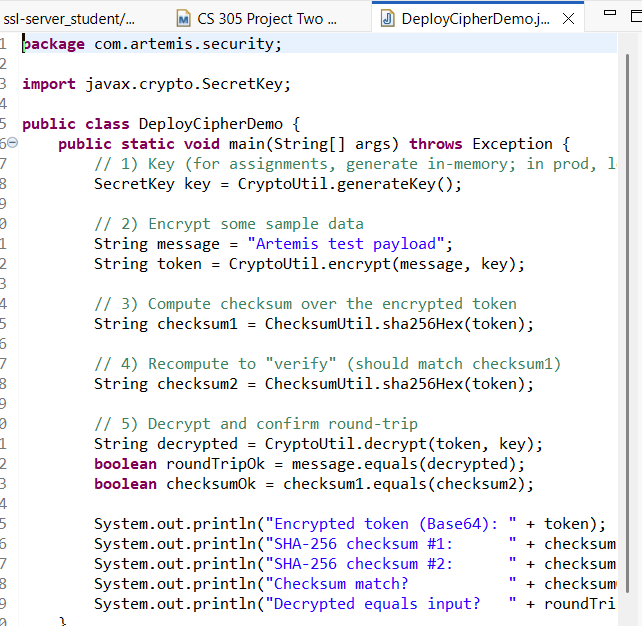
In conclusion, here is a picture of the app. In the settings file, you can see how the keystore was added to the project. The setting turns on SSL on port 8443, points to the keystore, and only allows TLS 1.2 and 1.3 transactions. This makes sure that the project is ready to use the newly created certificate for safe contact.





## Deploy Cipher





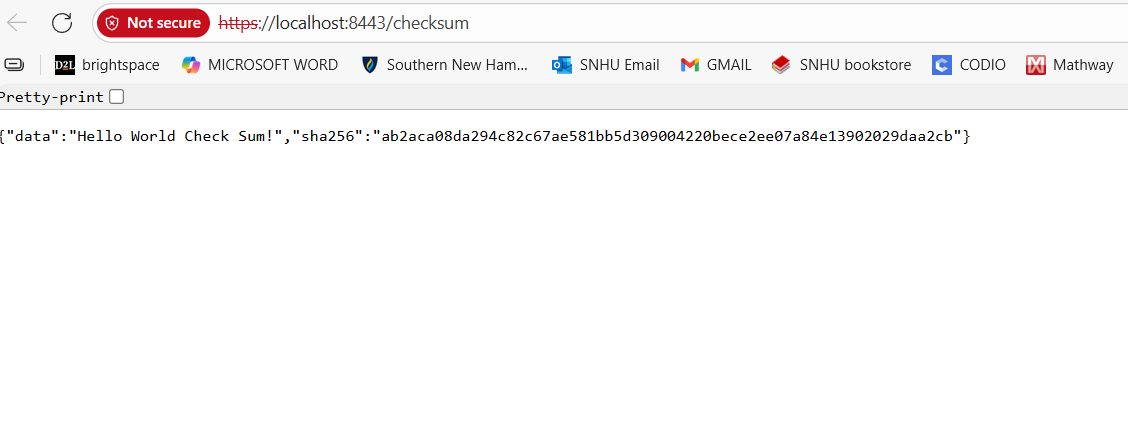
In this step, I set up a cipher deployment example to show that encryption and verification worked. The code makes a temporary secret key in memory and then encrypts the word "Artemis test payload" as an example message. The tool calculates a SHA-256 checksum of the encrypted token after it has been encrypted. The checksum is recalculated and compared to the original number to make sure that the ciphertext has not been changed.

The token is then decrypted back to its original form, and the program checks to see if the output of decryption fits the message that was sent. The final output from the console shows the encrypted token, both checksum values, proof that the checksums were correct, and the original input matched the decrypted message. This shows a full cycle of encryption, checking for integrity, and decryption, which is in line with best practices for safe deployment.

## Secure Communications

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

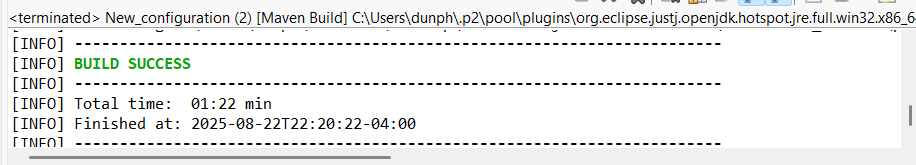
The web service can be seen running at https://localhost:8443/checksum in the picture. This shows that the app is indeed sending traffic over HTTPS. Along with its SHA-256 hash value, the JSON response shows a sample text ("Hello World Check Sum!"). This shows that the service is using secure transport (TLS on port 8443) and using a cryptographic hash properly to check the integrity of the message. Because it is a self-signed certificate, the browser marks the connection as "Not secure." This is normal in a development setting. In real life, the warning would not show up if a known certificate authority (CA) signed the certificate.



## Secondary Testing

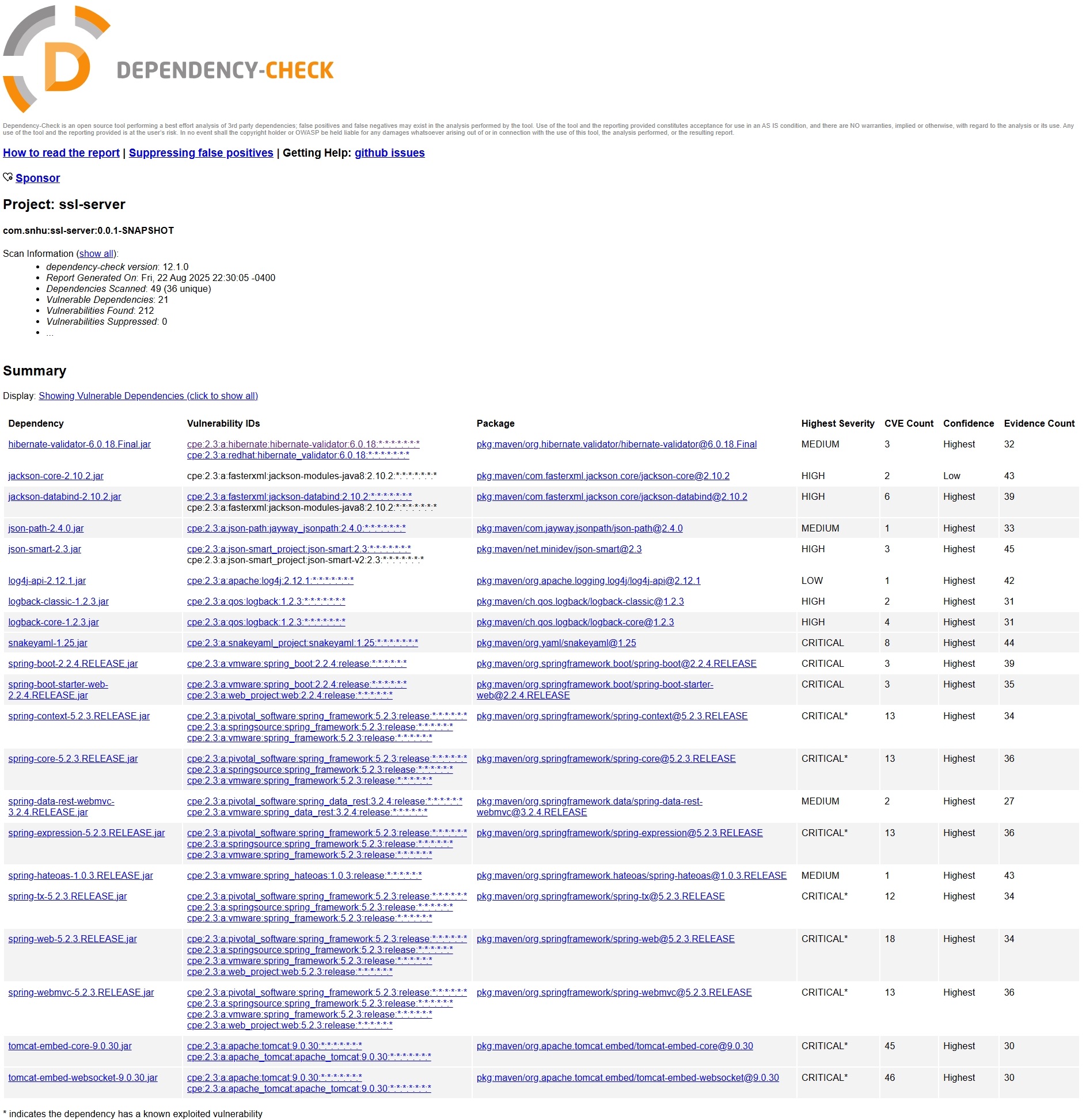
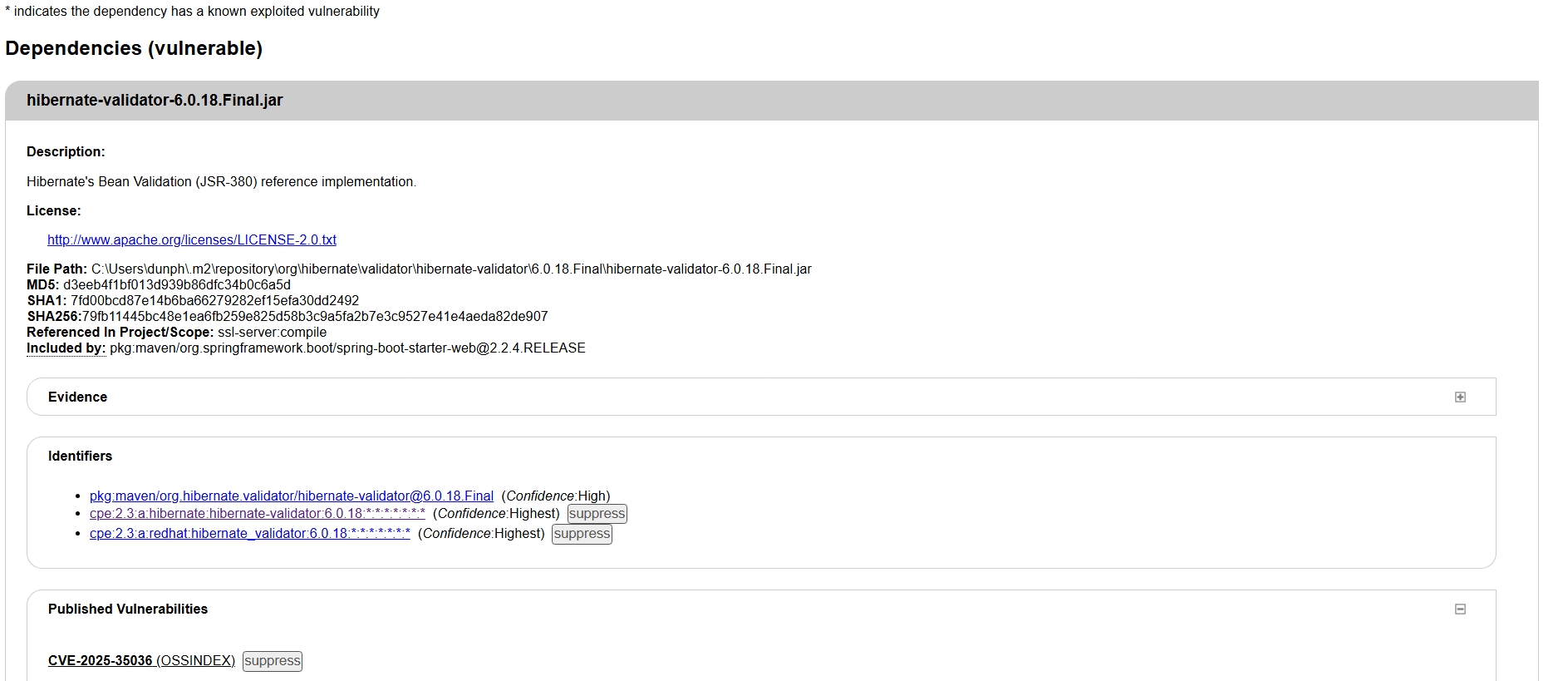
Insert screenshots below of the refactored code executed without errors and the dependency-check report.

The picture shows what happened when the refactored application build was run with Maven. The build went smoothly and finished. The "BUILD SUCCESS" message lets you know that the dependency-check plugin and the code worked as they should have, which means that there were no blocking vulnerabilities or configuration problems during the automatic static analysis. These pieces of proof show that the app passed secondary testing and that the Maven build process checked the dependencies on the project.



## Functional Testing

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



## Summary

I secured a Spring Boot web service by enabling HTTPS with a PKCS#12 keystore and ran automated dependency analysis. The application now serves only over TLS on port 8443. I generated a self-signed certificate, configured the keystore in the app, verified cryptographic fingerprints, and produced an OWASP Dependency-Check HTML report that inventories third-party libraries and flags known CVEs. Build and functional tests complete successfully.

## Industry Standard Best Practices

Based on the OWASP Top 10 and ASVS, industry-standard advice for web apps starts with transport security. To stop downgrade attacks, our service should only allow HTTPS connections and use TLS 1.2/1.3 with strong cipher suites and HSTS. We used a local PKCS#12 keystore on port 8443 for testing. For production, we switched to a certificate signed by a CA, changed the keys on a regular basis, and took care of when the certificates expired.

It's important to be careful with secrets and settings. You should never commit API keys and passwords to source control. Instead, get them from environment variables or a secret manager and keep different settings for each environment (dev, test, and prod). We run an automatic dependency scan with OWASP Dependency-Check and look over the HTML report that is made at target/dependency-check-report.html to lower the risk that comes from third-party libraries (OWASP A06). As part of our normal patch cycle, we also update libraries with critical or high CVEs.

We suggest setting up Spring Security so that it requires authentication, enforces least-privilege authorization, and uses secure defaults like HSTS and hardened headers at the service layer. As a way to stop injection, inputs should be checked (Bean Validation/JSR-380) and database access should use tailored queries/JPA. Secure, HTTPOnly, and the right SameSite should be used to protect sessions and cookies. In production, actuator or admin endpoints should be turned off or shut down.

In final but not least, tactical hardening rounds out the picture. Keep the JRE and runtime images up to date, only use the services you need, and keep private information out of logs by centralizing logging. Keep an eye on failed authentications, strange traffic, and scan results, and make sure that your important data is backed up with encrypted, tested backups. To make sure that builds are safe and can be repeated, you should require code review in CI and run tests and dependency scans on every build. You should also keep plugin and tool versions saved.