CS763

Lab 1

11/12/23

Alessandro Allegranzi

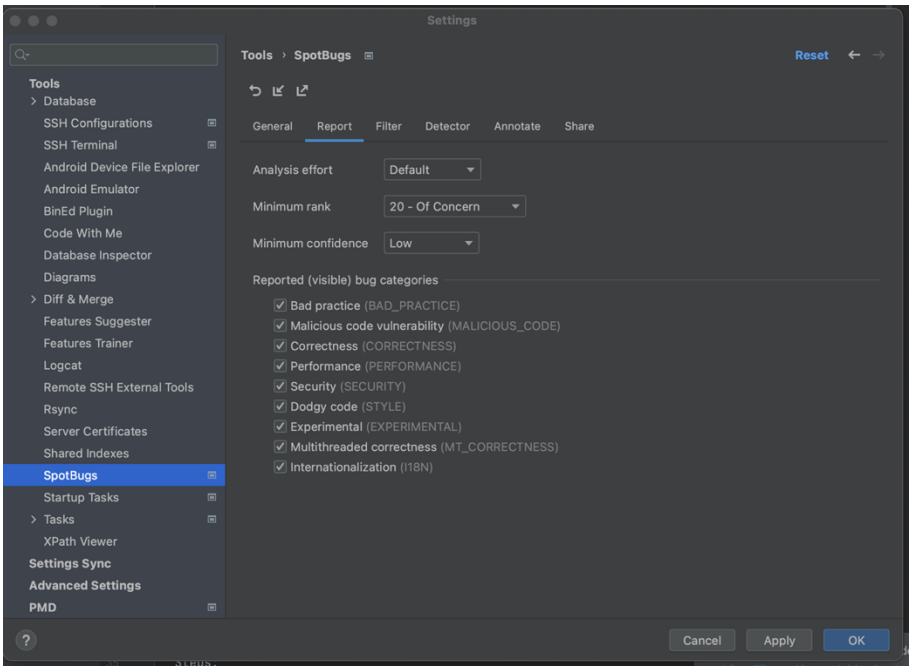
Table of Contents

Lab Steps

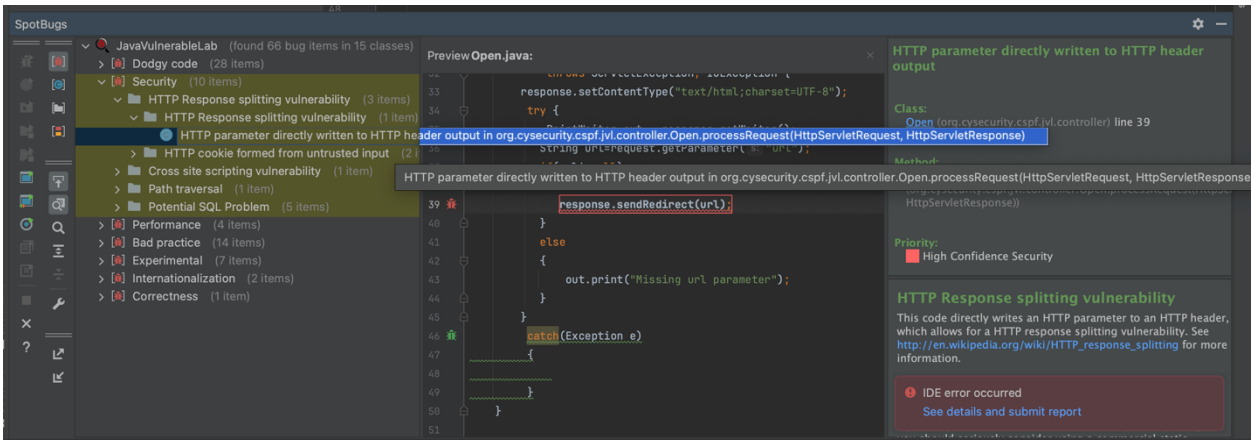
Below I am including some screenshots to show how I went through the lab steps as instructed in the lab document.

Spotbugs

I downloaded the spotbugs plugin for IntelliJ and configured the plugin as per the instructions.



I ran the Spotbugs Analysis through the IDE options on all files including test sources.

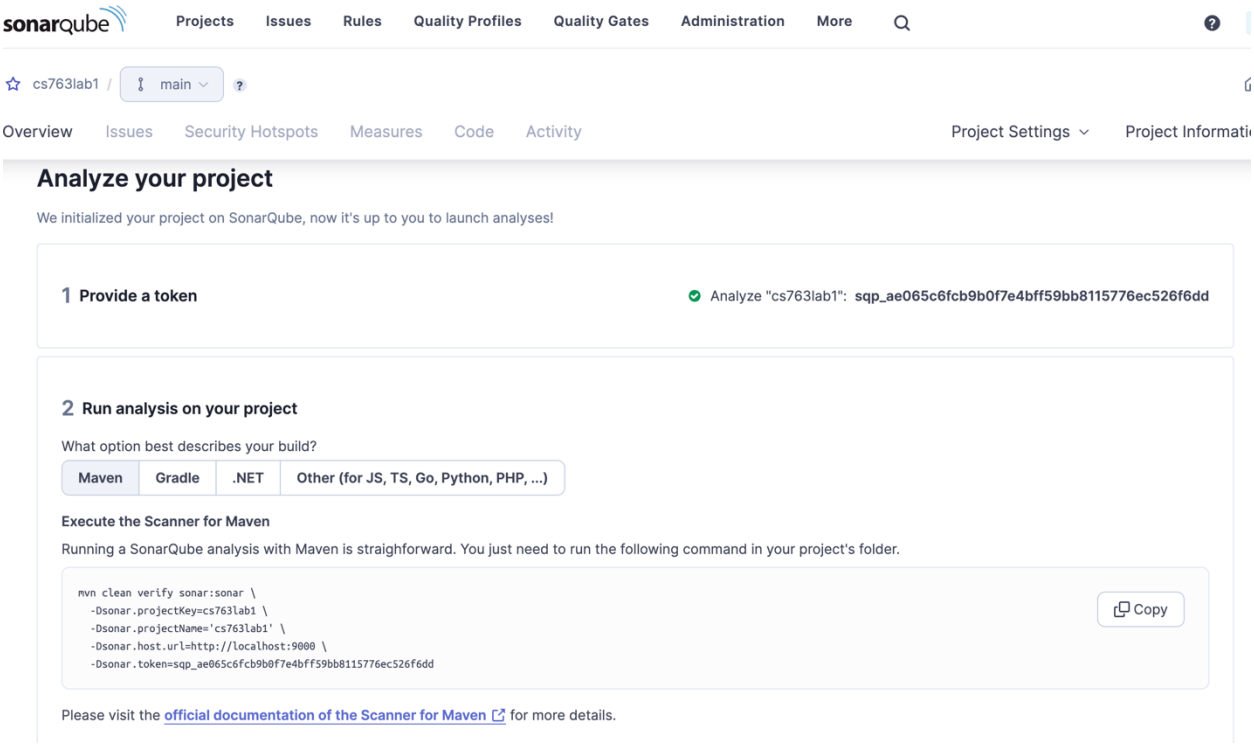


Sonarqube

I ran Sonarqube via docker rather than downloading and installing locally.

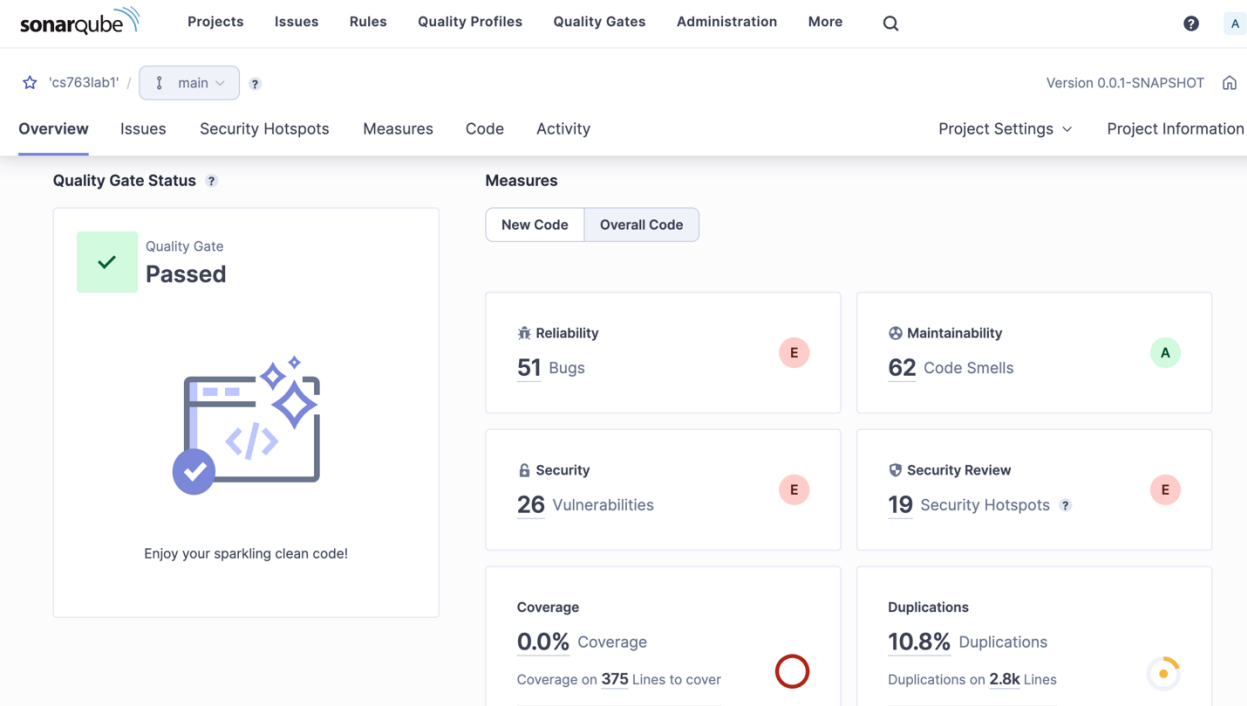
docker run -d --name sonarqube -e SONAR_ES_BOOTSTRAP_CHECKS_DISABLE=true -p 9000:9000 sonarqube:latest

I followed instructions to create a project token.

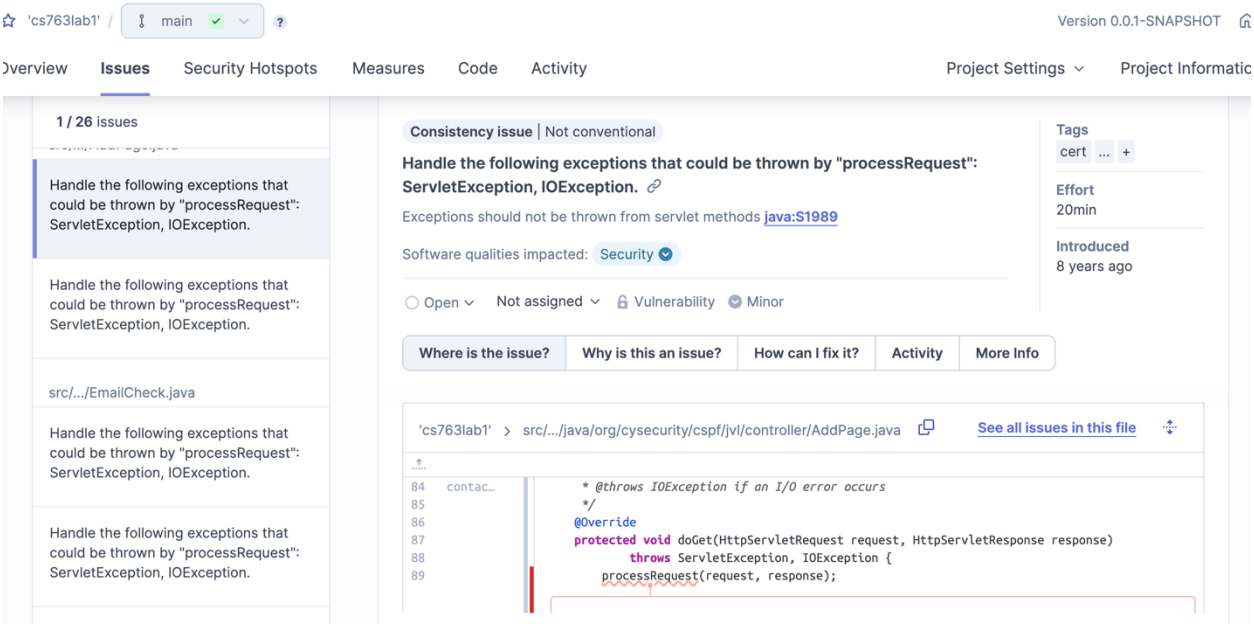


I ran the Sonarqube scan locally using the provided mvn command. Here is a screenshot of the results overview.

```
mvn clean verify sonar:sonar \
  -Dsonar.projectKey=test \
  -Dsonar.projectName='test' \
  -Dsonar.host.url=http://localhost:9000 \
  -Dsonar.token=sqp_47dbb44e41fb86ef7d47a22aeb3f238d8b871d2e
```



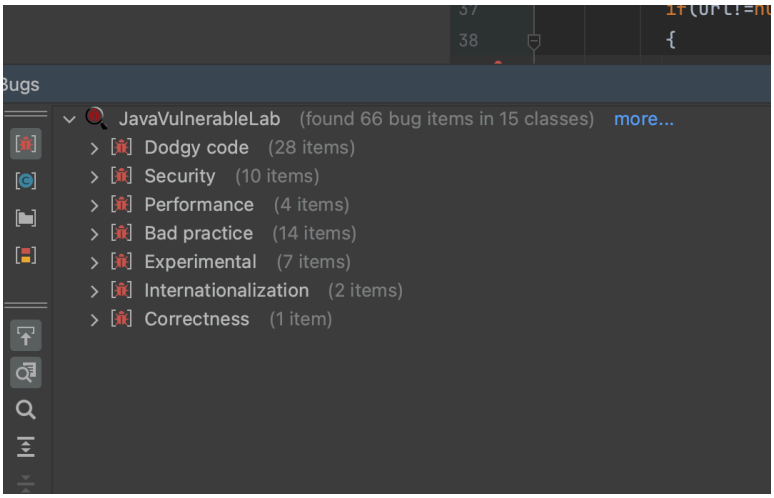
An example highlighted security vulnerability.



Questions

1. (20 points) How many bug items are identified by SpotBugs? What are the categories of these bugs? What are security bugs identified? Modify the configuration to only report the scary bugs. How many are they? Can you find any false positives?

   I ran the Spotbugs scan on all files including test sources, and it identified 66 bugs total.



   The breakdown is:
   Dodgy code: 28
   Security: 10
   Performance: 4
   Bad practice: 14
   Experimental: 7
   Internationalization: 2
   Correctness: 1

   The security bugs identified were (10 total):

   HTTP parameter directly written to HTTP header output

   > "This code directly writes an HTTP parameter to an HTTP header, which allows for a HTTP response splitting vulnerability. See http://en.wikipedia.org/wiki/HTTP_response_splitting for more information."

   HTTP cookie formed from untrusted input (x2)

"This code constructs an HTTP Cookie using an untrusted HTTP parameter. If this cookie is added to an HTTP response, it will allow a HTTP response splitting vulnerability. See http://en.wikipedia.org/wiki/HTTP_response_splitting for more information."

HTTP parameter written to Servlet output

"This code directly writes an HTTP parameter to Servlet output, which allows for a reflected cross site scripting vulnerability. See http://en.wikipedia.org/wiki/Cross-site_scripting for more information."

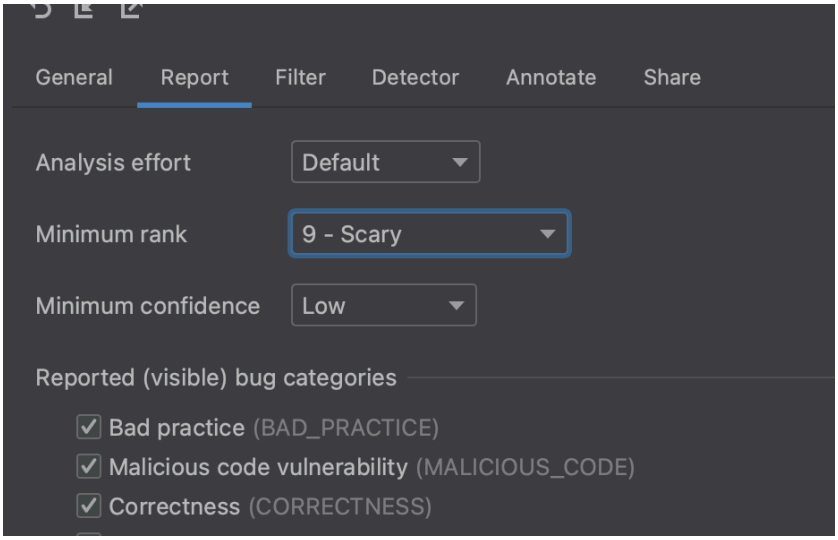
Relative path traversal

"The software uses an HTTP request parameter to construct a pathname that should be within a restricted directory, but it does not properly neutralize sequences such as ".." that can resolve to a location that is outside of that directory. See http://cwe.mitre.org/data/definitions/23.html for more information."

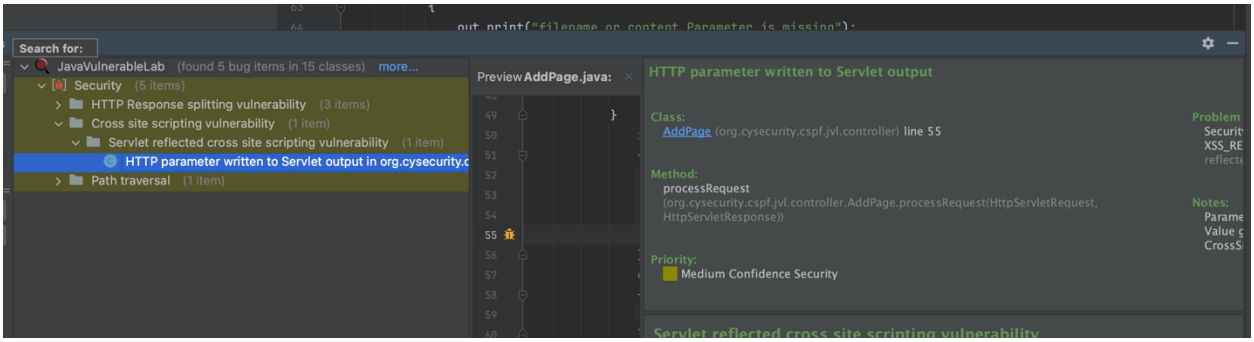
Non constant string passed to execute or addBatch method on a SQL statement (x5)

"The method invokes the execute or addBatch method on an SQL statement with a String that seems to be dynamically generated. Consider using a prepared statement instead. It is more efficient and less vulnerable to SQL injection attacks."

I altered the configs to only find scary bugs:



The report now showed only 5 security bugs, and no other bug type:



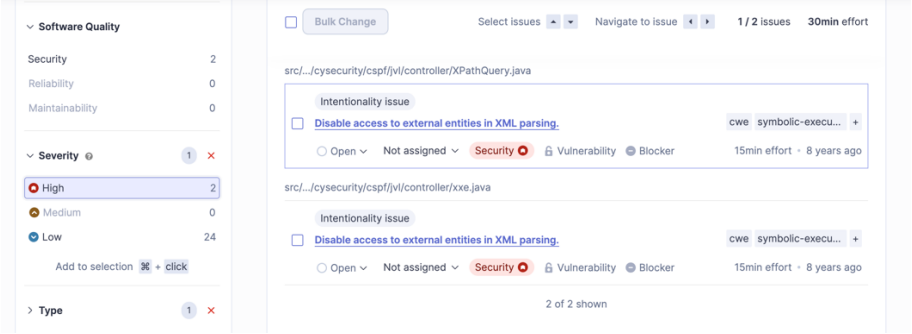
I did not spot any false positives in the report, but I have little experience on the subject of security, so there may have been some.

2. (20 points) How many issues are identified by SonarQube? What are the categories of these issues? What is the difference between vulnerability and hotspots? How many critical security issues are identified by SonarQube? Can you find any false positives?

   Sonarqube identified
   51 reliability bugs
   62 code smells
   26 security vulnerabilities
   19 security hotspots

   Security vulnerabilities are verified security flaws in code that need to be fixed immediately. Hot spots are security-sensitive pieces of code that should be reviewed but may not impact application security [1]. Hot spots are an example of defense in depth computing, where redundant layers of security are implemented as extra protection in the case certain vulnerabilities are exploited [2].

   Sonarqube found 2 high severity security vulnerabilities, but no critical issues.



Smilar to question 2, I did not spot any false positives in the report, but I have little experience on the subject of security. I'm sure there must have been some given the high number of flaws called out in the report.

3. (20 points) Compare the results by SpotBugs and SonarQube and state your findings.

Spotbugs and Sonarqube both showed overlap on security vulnerabilities. The main difference seemed to be how the two scans organized and classified the vulnerabilities. Sonarqube found many more security vulnerabilities than Spotbugs. Including Security hotspots, Sonarqube found 45 issues to Spotbugs' 10 security vulnerabilities. The reports also had different vulnerability category and naming conventions, but mostly caught the same issues, like HTTP Response Splitting and path traversal vulnerabilities. For example, both scans highlighted possible SQL injection vulnerabilities due to unsanitized user input, like email addresses. Both reports also called out potential SQL injection; those warnings were grouped under "Security Hotspots" in the Sonarqube UI.

Interestingly, the Sonarqube scanned categorized a high severity bug in the XML parser that left it open to XXE attacks in both the Xxe.java and XPathQuery.java controller files. The Spotbugs report did not point out either of those vulnerabilities that Sonarqube defined as "high" security threats. Similarly, Spotbugs called out a "HTTP cookie formed from untrusted input" vulnerability was not included in the Sonarqube report.

The most significant difference between the reports is that Sonarqube also provides a lot more guidance on code quality and readability suggestions, beyond security vulnerabilities. For example, Sonarqube suggested many actions like changing methods to be static, or adding "@Override" annotations to certain methods, for readability and maintainability. It also called duplicate lines of code. Sonarqube's categories also feature bug overlap, so for example 51 reliability bugs and 26 security bugs do not point to 77 separate bugs, but bugs that fall into both categories. Overall,

the Sonarqube report seemed more detailed and comprehensive than the Spotbugs scan. Spotbugs' main strong point, in comparison, is the ease of running the plugin inside the IntelliJ IDE while actively developing.

It seems like running both scans is a good idea and the best way to catch as many security vulnerabilities as possible.

4. (40 points) Choose 5 security bugs (or security issues) from any report and explain them in more detail. Do you know how to fix them? You may need to do additional research on them.

The first 3 security bugs I took from the Spotbugs report, and the last 2 are from Sonarqube.

1.

HTTP Response Splitting Vulnerability – HTTP Parameter directly written to HTTP header output

Location: Open.java, ProcessRequest() method line 39.
Description: "This code directly writes an HTTP parameter to an HTTP header, which allows for a HTTP response splitting vulnerability. See http://en.wikipedia.org/wiki/HTTP_response_splitting for more information."

HTTP response splitting is a security vulnerability that occurs when an attacker injects malicious content the output of an HTTP response [3]. This manipulation can lead to the splitting of the response into multiple parts, enabling the attacker to insert arbitrary content between these parts. The consequences of successful HTTP response splitting attacks include the potential for browser-based attacks such as cross-site scripting (XSS) or the manipulation of headers, leading to unauthorized access or information disclosure [3].

Some generic solutions to mitigate are: url-encode strings into the HTTP headers [3]. Do not let users control the value in the 'Location' header. Define the Content Security Policy in the HTML [3].

2.

Relative Path Traversal

Location: AddPage.java, processRequest() method line 45
Description: "The software uses an HTTP request parameter to construct a pathname that should be within a restricted directory, but it does not properly neutralize sequences such as ".." that can resolve to a location that is outside of

that directory. See http://cwe.mitre.org/data/definitions/23.html for more information."

A relative path traversal security vulnerability occurs when an application does not properly validate or sanitize user-supplied input, allowing an attacker to navigate through the file system using relative paths and access sensitive files or directories [4]. In such a scenario, an attacker may manipulate input parameters to traverse up or down the directory structure, gaining unauthorized access to files or data outside the intended scope of the application, which is a confidentiality vulnerability [4]. The attacker may also negatively affect application integrity by creating or altering critical files. Additionally, availability could also be compromised if the attacker deletes or corrupts unexpected critical files [4].

Mitigation strategies include input validation, enforcing proper access controls, and utilizing absolute paths instead of relying solely on relative paths to access resources within a web application [4].

3.

Servlet Reflected Cross-site Scripting Vulnerability
Location: AddPage.java, processRequest() method line 55
Description: "This code directly writes an HTTP parameter to Servlet output, which allows for a reflected cross site scripting vulnerability. See http://en.wikipedia.org/wiki/Cross-site_scripting for more information."

A Servlet Reflected Cross-site Scripting (XSS) Vulnerability occurs when a web application using Java Servlets allows user-supplied input to be included in the response without proper validation or output encoding [5]. In this scenario, an attacker can inject malicious scripts into input fields or parameters, and the server reflects this input back to the user within the generated HTML content. When unsuspecting users visit the affected page, the injected scripts are executed in their browsers within the context of the vulnerable application, potentially leading to the theft of sensitive information or unauthorized actions on behalf of the user [5].

Mitigation strategies involve validating and encoding user input before including it in responses, implementing secure coding practices, and utilizing frameworks or libraries that provide protection against XSS attacks [5].

4.

Disable access to external entities in XML parsing

Location: xxe.java, processRequest() method, lines 40-66

Description: “XML parsers should not be vulnerable to XXE attacks”

External Entity Injection (XXE) attacks occur when untrusted XML input containing a reference to an external entity is processed by a XML parser. [6]. This type if attack could lead to the disclosure of confidential data, denial of service, server side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts [6]. Java applications are particularly vulnerable to XXE attacks since the default setting on most JAVA XML parsers is to have XXE enabled [6].

A general mitigation strategy for XXE attacks is disabling DTD’s (External Entities) completely [6].

5.

Handle the following exceptions that could be thrown by "processRequest": ServletException, IOException.

Location: AddPage.java, line 89

Description: Servlets should not throw exceptions

Servlets are Java components used in web development to process HTTP requests and generate responses. Allowing servlets to throw exceptions will propagate them to the servlet container, where the default error-handling mechanism could impact the overall security and stability of the server [7]. Uncaught exceptions could leave the server in an unstable state, leaving it vulnerable to denial-of-service attacks. Additionally, exceptions by default include detailed error messages that may include sensitive data like stack traces or system configurations [7].

A general mitigation strategy is to surround all methods that could throw an exception with a try/catch block [7]. The exception can be handled by the system more elegantly and logged securely.

5. (Extra 5 points) If you are familiar with Java and web development, you may review the code manually, and see if you can find any security issues that are not identified by Spotbugs or SonarQube?

I scanned the code but was not able to spot anything that stuck out that Spotbugs or Sonarqube hadn't caught.

Summary and Reflection

1. What is the purpose of the lab in your own words?

   The purpose of this lab is to gain experience running SAST scans on code using automated frameworks and reading the resulting security bug reports. SAST tools are very useful to increase software security and overall code quality, so the experience from this lab should be very useful for future projects.

2. What did you learn? Did you achieve the objectives?

   I learned how to configure two static code scanners and run scans on code. I also learned how to read scan reports and filter bugs by type and severity. I did achieve the objectives.

3. Was this lab hard or easy? Are the lab instructions clear?

   I would rate the lab as low-medium difficulty, as it mostly involved running scans with 3rd party tools on code. The instructions were clear and I did not need to look for additional instructions online to get the tools to work. The research around security vulnerabilities was also very easy to do and I was able to find many resources. The results were overwhelming, and parsing real, pressing vulnerabilities from false positives was difficult, even given the SAST tools' vulnerability categories.

4. What do you think about the tools used? What worked? What didn't? Are there other better alternatives?

   Both tools seemed effective. Running Spotbugs inside the IDE was very convenient, and using an IDE plugin was much easier than running Sonarqube through Docker and working through the necessary configurations.

Any other feedback?

None

References

[1] "Security Hot Spots." *Sonarqube Docs*. https://docs.sonarsource.com/sonarqube/10.2/user-guide/security-hotspots/

[2] “defense-in-depth”. *NIST*. https://csrc.nist.gov/glossary/term/defense_in_depth

[3] “HTTP response splitting exploitations and mitigations”. Detectify. https://blog.detectify.com/industry-insights/http-response-splitting-exploitations-and-mitigations/

[4] “CWE-3 Relative Path Traversal.” *MITRE*. https://cwe.mitre.org/data/definitions/23.html

[5] “Cross Site Scripting (XSS)”. *OWASP*. https://owasp.org/www-community/attacks/xss/

[6] “XML External Entity Prevention Cheat Sheet”. *OWASP Cheat Sheets*. https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html

[7] “Exceptions should not be thrown from servlet methods” *Sonarsource Rules*. https://rules.sonarsource.com/java/RSPEC-1989/