CS 763 Project Assignment 4

Security Testing

Alessandro Allegranzi

Table of Contents

Github Repository

https://github.com/1-8192/CS763_project

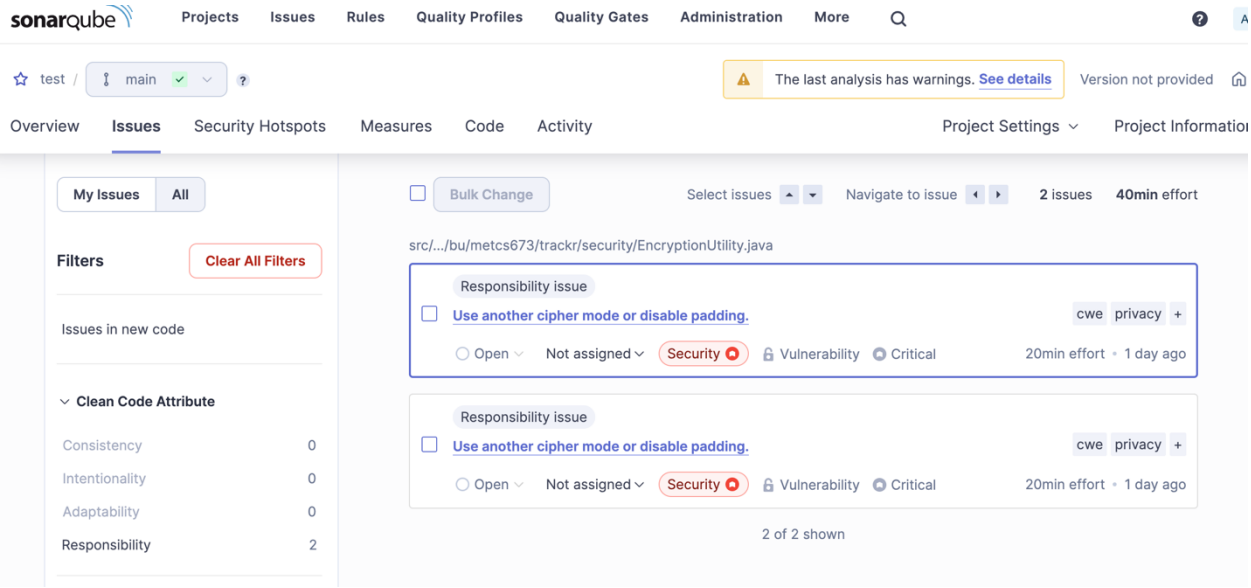
PR for SAST suggested changes: https://github.com/1-8192/CS763_project/pull/2

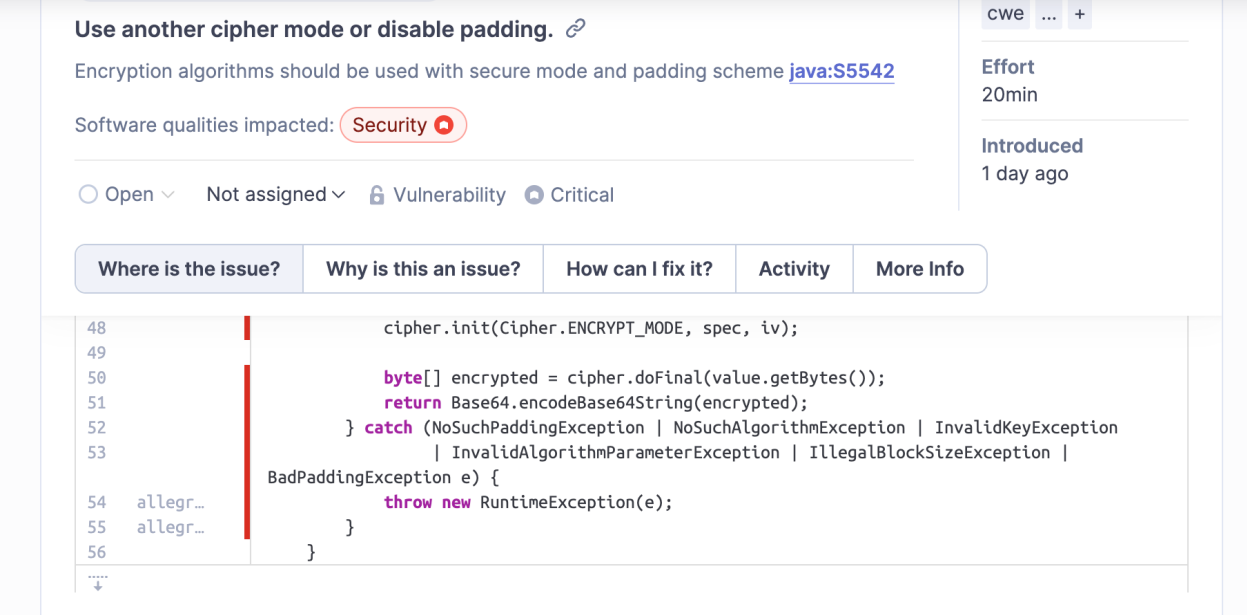
Project Report

First, find a SAST tool to scan the code and report if there are any issues. You may have already done this in assignment 1. If any issues are found, explain them in more detail, and try to fix them or at least propose possible solutions to fix them.

I have run a Sonarqube scan on the project in past weeks. Reports have shown some issues. I will include one example from last week's project, when I implemented database encryption.

I ran a Sonarqube SAST scan locally, which showed a vulnerability related to the original encryption algorithm I was using.





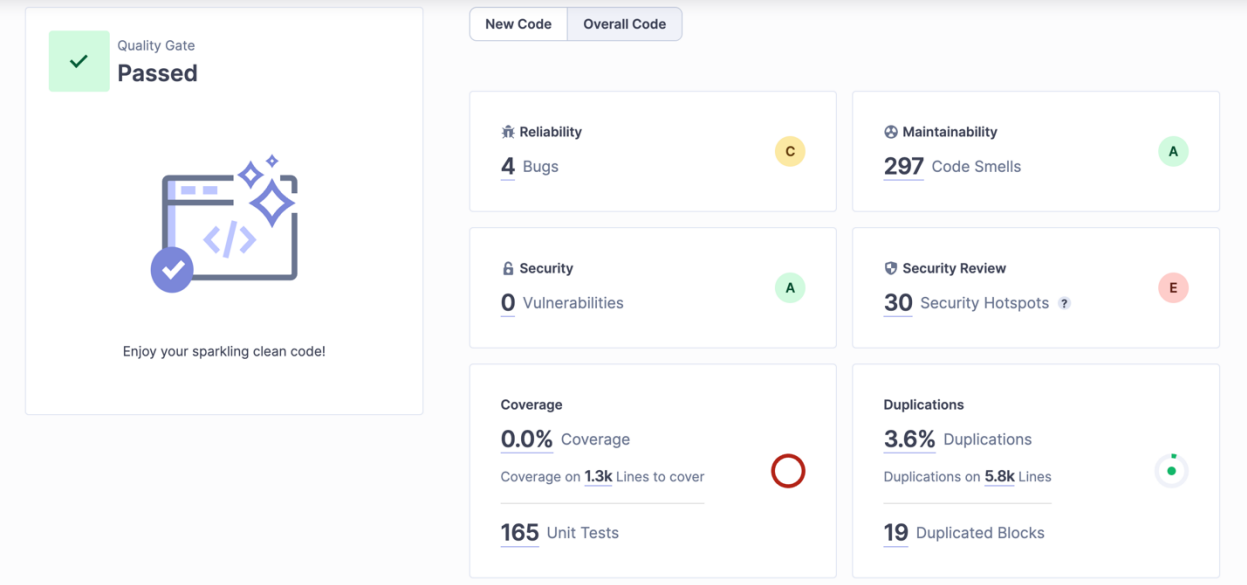
Sonarqube flagged the use of Cipher Block Chaining (CBC) mode and padding as a security vulnerability. CBC with padding is one of the weaker AES modes, and uses an initialization vector to add padding [1]. The initialization vector is vulnerable to attack, and if it breaks all succseive decryption breaks, as well. Galois/Counter Mode (GCM) combines encryption with authentication and integrity checks to provide both confidentiality and authenticity of data, so it is the recommended AES algo to use [1]. I followed Sonarqube's suggestion and swapped out the "AES/CBC/PKCS5PADDING" algorithm for "AES/GCM/NoPadding" crypto algorithm for Cipher.

One other change I missed in the previous assignment (Assignment 3) and that I needed to make was to change out the IvParameterSpec class for the GCMParameterSpec inside the encryption and decryption methods like so:

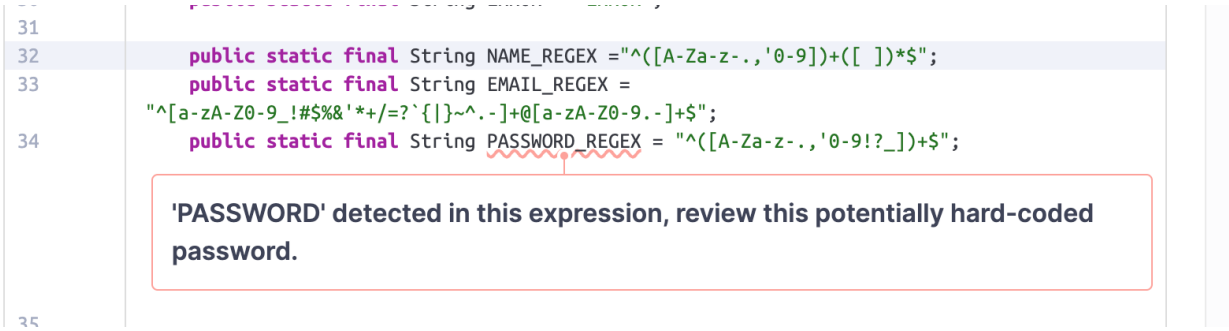
```
/**
 * Method to decrypt string.
 * @param encrypted encrypted string
 * @return decrypted String
 * @throws RuntimeException
 */
public String decryptString(String encrypted) throws RuntimeException {
    try {
        GCMParameterSpec gcmSpec = new GCMParameterSpec(128,
(vector.getBytes(StandardCharsets.UTF_8)));
        SecretKeySpec spec = new
SecretKeySpec(key.getBytes(StandardCharsets.UTF_8), algo);

        Cipher cipher = Cipher.getInstance(cryptoAlgo);
        cipher.init(Cipher.DECRYPT_MODE, spec, gcmSpec);

        byte[] original = cipher.doFinal(Base64.decodeBase64(encrypted));
```

```
        return new String(original);
    } catch (NoSuchPaddingException | NoSuchAlgorithmException |
InvalidKeyException
             | InvalidAlgorithmParameterException | IllegalBlockSizeException
| BadPaddingException e) {
        throw new RuntimeException(e);
    }
}
```

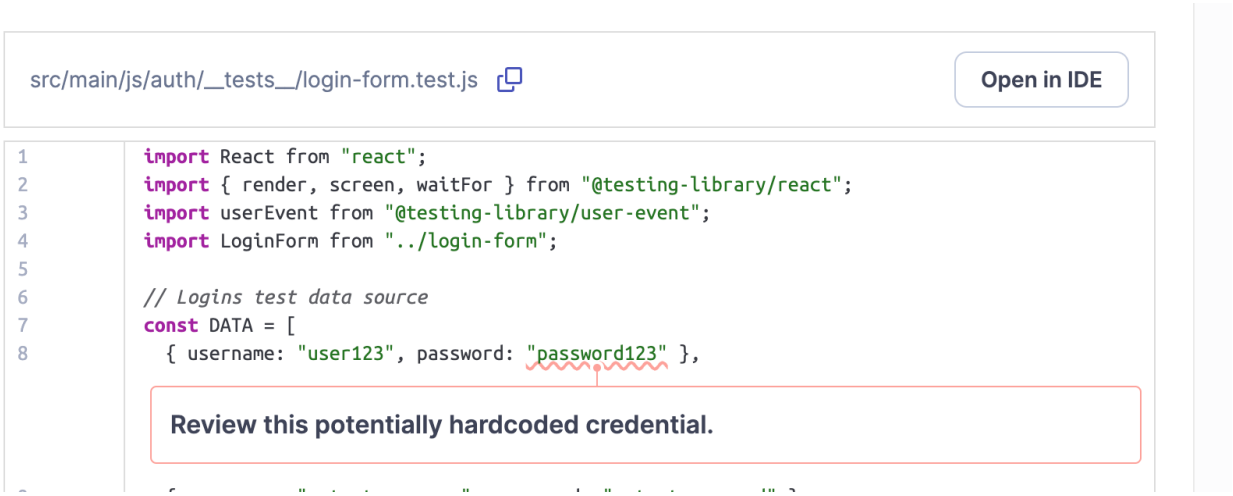
The successive scan showed the vulnerabilities were resolved. However, the scan showed several Security Hotspots for review.



Some of the hotspots I reviewed and deemed “safe.” For example, a constant named PASSWORD_REGEX set off a hotspot for review because SQ thought it may have been a hardcoded password saved to a variable. In this case, it was just a regex pattern needed to validate passwords, so I marked it safe. The warning was then removed from the report.



Another example was the report found ‘password’ in a test file, and flagged that a security concern. It seems like SQ just flags any instance of password as problematic. SQ does have settings to exclude test files from analysis, but it does not look like there are setting to exclude test files from ONLY security analysis. In terms of code coverage and other metrics, I think it is a good idea to keep including test files in the report.



The report did show several hotspots related to the dockerfile. The two issues were copying recursively and using the root user as default. The issues showed up multiple times on different commands, but those were the two main security concerns.

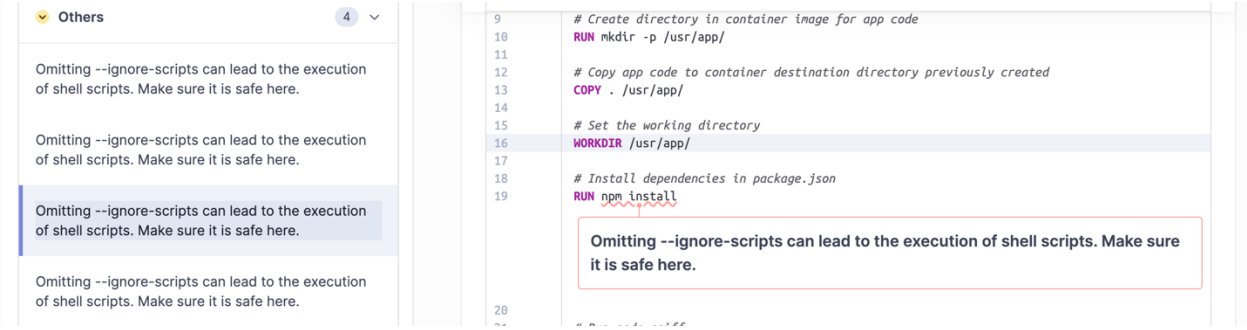
According to the SQ report, copying recursively can be an issue because
When ‘COPY’ or ‘ADD’ are used to recursively copy entire top-level directories at build-time, unexpected files might get copied to the image filesystem and affect their confidentiality. Additionally, running containers as privileged users is a vulnerability for runtime security. Any user running code on the container could perform administrative actions. The report gave this example: “Suppose the container is not hardened to prevent using a shell, interpreter, or **Linux capabilities**. In this case, the malicious user can read and exfiltrate any file (including Docker volumes), open new network

connections, install malicious software, or, worse, break out of the container's isolation context by exploiting other components."

I ended up leaving the docker file and these errors as is, as well. The docker file already creates a new nonroot user deeper in the file (https://github.com/1-8192/CS763_project/blob/773df467873b9831f84e10b151630533458d837e/Dockerfile#L44), and creating the new user early would require giving the nonroot user sudo permissions to run those commands, which defeats the purpose either way.

In terms of the copy command, I don't have enough knowledge of docker builds to know which specific files I should copy instead of grabbing the whole app, so I left that code as is as well. I think for most uses the docker build is still fairly safe for the scope of the application.

SQ also revealed some smaller configuration issues. For example, it recommended I add a '–ignore-scripts' flag to the npm install commands to block the execution of shell scripts. Shell scripts running inside the docker environment would be a problem since the attacker could run whatever code they wanted. That was a quick change and I added it.



Code example:

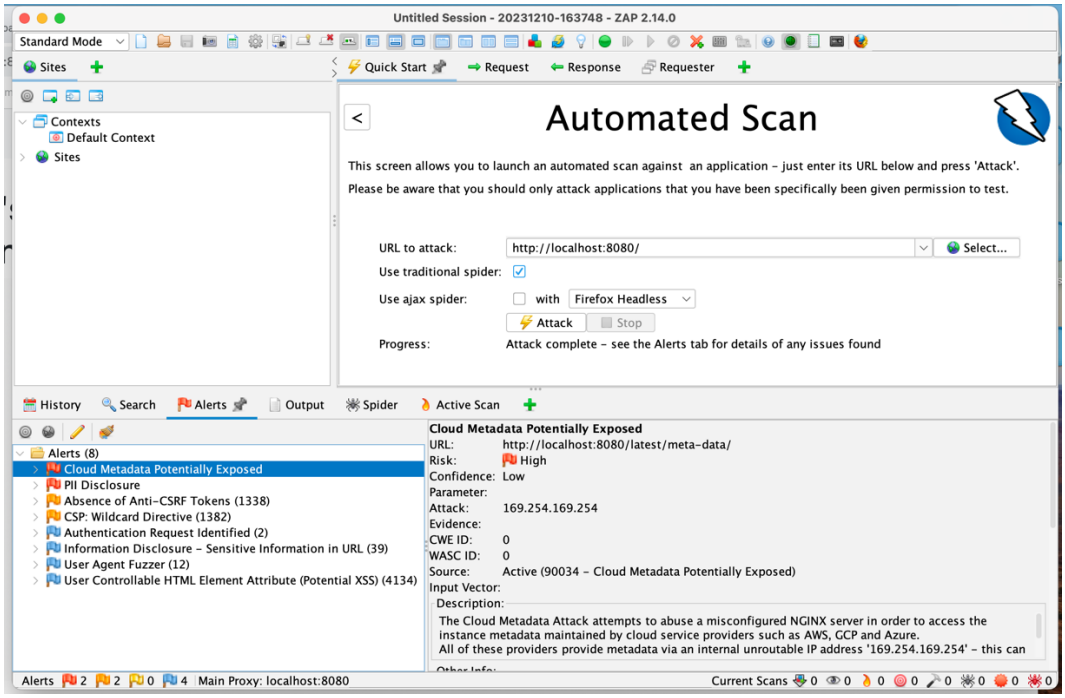
```
# Install Webpack, Webpack CLI global then all the dependencies in
package.json
RUN npm install -g webpack webpack-cli --ignore-scripts
RUN npm install --ignore-scripts
```

Then find a DAST tool to scan your app and report if there are any issues. The recommendation is to use OWASP ZAP (https://www.zaproxy.org/) for web applications. You can also use other tools. If any issues are found, explain them in more detail. Most tools such as ZAP provide automated scanning and attack. Mostly, you will need to

configure an authenticated scanning in order to scan pages that are only accessible by authenticated users.

Similar to the SAST section, I have run a ZAP DAST scan already in previous weeks but as automated Github actions. This week I downloaded the ZAP desktop application and ran scans locally.

I started by running the Trackr application locally via docker and starting an automated scan against the localhost url index page (localhost/8080).



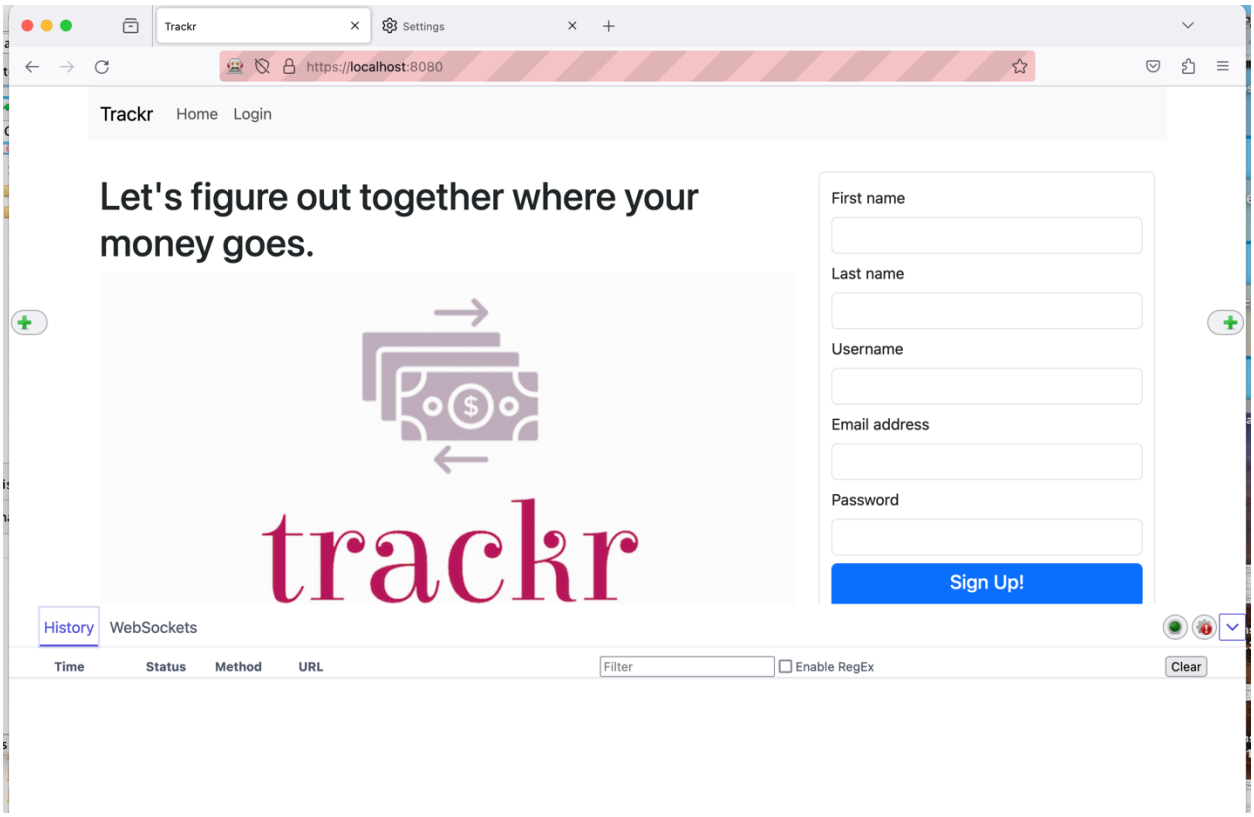
The two high risk flags are for Cloud Metadata being exposed and PII exposure. The Cloud Metadata risk is related to NGINX, which is a web server and load balancer used in many applications deployed on AWS, GCP, or other cloud services. The project is not currently deployed via cloud services, so I believe this risk is safe enough to leave unmitigated for the moment. PII is personally identifiable information, and the PII risk is about personal sensitive data being exposed. I'm not entirely sure which path set off this error. I examined the Spider tab, and it looks like it just crawled standard, random paths like /sitemaps.xml or /robots.txt, without really understanding the application itself. The Trackr application does have many dependencies, especially related to Bootstrap styling and other libraries, so the scan may have been following those links.

GET	http://localhost:8080/	Seed
GET	http://localhost:8080/robots.txt	Seed
GET	http://localhost:8080/sitemap.xml	Seed
GET	http://localhost:8080/OTHER/network/other/proxy.pac/?apin...	
GET	http://localhost:8080/OTHER/network/other/rootCaCert/?api...	
GET	https://www.zaproxy.org/docs/desktop/addons/network/opti...	Out of Scope
GET	http://localhost:8080/UI	
GET	https://www.zaproxy.org/	Out of Scope
GET	https://groups.google.com/group/zaproxy-users	Out of Scope
GET	https://groups.google.com/group/zaproxy-develop	Out of Scope
GET	https://github.com/zaproxy/zaproxy/issues	Out of Scope
GET	http://localhost:8080/UI/	
GET	http://localhost:8080/UI/acsrf/	
GET	http://localhost:8080/UI/ajaxSpider/	

There was a medium alert related to the absence of anti-CSRF tokens. We have discussed CSRF attacks in the course, so that alert did seem familiar and valid to the application.

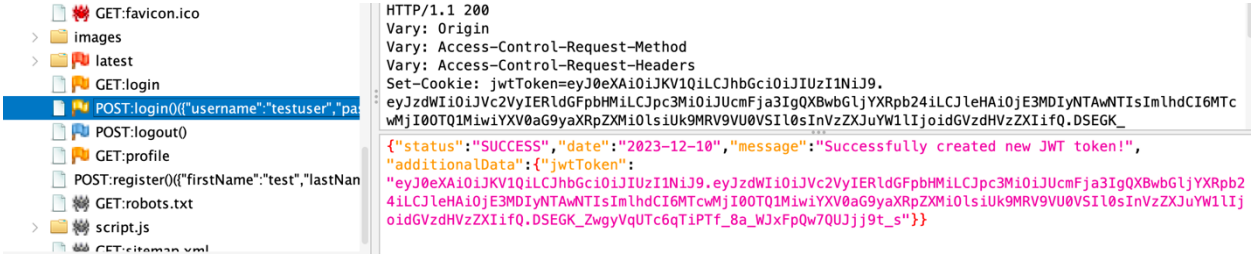
Next, I ran a scan using the Ajax Spider with the Chrome headless setting. The spider started crawling thousands of localhost URL's and I eventually stopped it manually. It did not reveal any additional risks and didn't seem like the best scan to use.

I then ran a manual scan through the Firefox browser. I tried with Chrome originally but had a hard time setting up the ZAP proxy and adding certificates, so I switched to Firefox. After setting the ZAP desktop proxy to port 9000 and adding the certificate to the Firefox browser, I launched the manual crawl, which brought up the Trackr application page as well as the ZAP HUD.

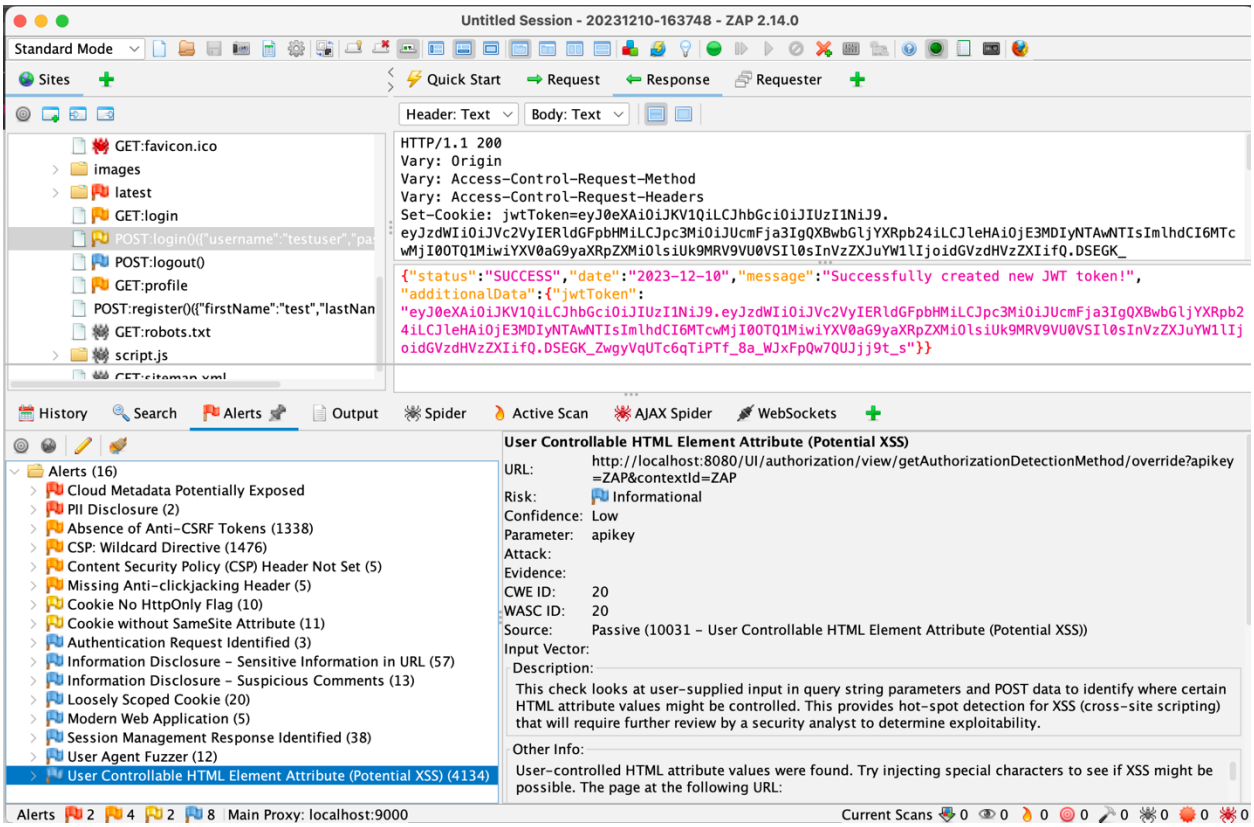


I interacted with the application in several ways by creating a profile, adding bank accounts, editing them, and adding transactions. I also manually logged out. The ZAP desktop showed the

url paths I was accessing and recorded the requests being made to the server. It also showed some new risks it had identified as I used the application. The ZAP Desktop application showed a lot of information for each request, including full headers, full response information, and payloads. It is really a great tool.



Below is a screenshot of the alerts panel, as well:



Most of the flagged medium risks stemmed from header and cookie settings. For example, ZAP called out a new vulnerability that the CSP header policy was not set. Content Security Policies help protect web applications against XSS and injections attacks by specifying allowed sources that can be loaded. Setting an appropriate CSP for the Trackr application would mitigate the main high-risk vulnerabilities that the ZAP DAST scan specified.

For this week's assignment, I attempted to implement a CSP in the Trackr application, but I ran into issues with the bootstap styling library, which Trackr uses. I was not able to set up a CSP successfully that also allowed bootstrap to work in time.

I checked the front end code, and the index.html file indeed did not specify a CSP policy in the head tag (the security configuration class also did not set a CSP policy. (More on that below)

```
<!DOCTYPE html>
<html xmlns:th="https://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title>Trackr</title>
</head>
<body>

<div id="react"></div>

<script src="built/bundle.js"></script>

</body>
</html>
```

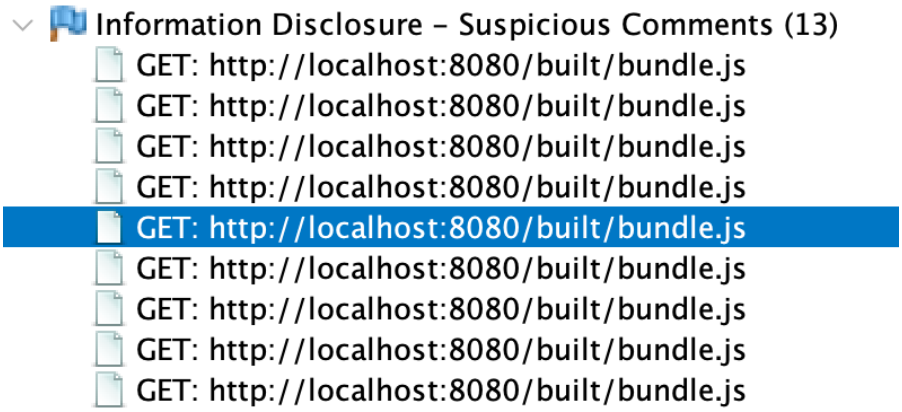
Another missing header that Zap highlighted, still related to CSP, is the 'frame-ancestors' directive. Anti-clickjacking headers are important to mitigate clickjacking, or "UI redress" attacks. Those attacks happen when an attacker tricks a user into clicking a link to another page when they meant to click to a top level page [2].

ZAP also found vulnerabilities related to the cookies used by the application, which did not specify the HttpOnly flag or SameSiteAttribute. Without the HttpOnly flag, a cookie can be accessed via Javascript, which is a serious vulnerability because it is a vector by which malicious code can be run. The missing SameSiteAttribute leaves the application code vulnerable to cross site script inclusion and attacks. These cookies-related errors were categorized as low risk.

I checked out the WebSecurityConfig class in the repository, and the configuration does define a CSRF protection scheme, but some options are set that seem insecure. For example, it chains a withHttpOnlyFalse() method that does not include an HttpOnly flag. It also does not

define a CORS policy. Like the ZAP scan specified, it looks like the Trackr application cookie settings are vulnerable and could be strengthened against attack.

The ZAP alerts also specify some informational level vulnerabilities. These vulnerabilities were also related to loosely scoped cookies, and possible revealing information in urls. The Trackr application is fairly simple and was built using bulky frameworks like REACT and Spring Boot, which automate a lot of the application creation and controller pathing. For example, many of the urls that ZAP flagged as having security vulnerabilities were bundle.js paths:



I imagine many of those vulnerabilities come from framework or template code and not the Trackr application-specific front end code. The ZAP report shows that a lot of additional work is necessary to ensure applications built using this technology handle sessions, authorization, and header management in a secure fashion.

(Optional) You can try to integrate ZAP into Github workflow through github actions. This can help automation but it is optional.

I have set up ZAP scans as a Github action in the project repository. The scans run on any push to main. Each scan creates a zip file that can be downloaded, and that contains scan results as HTML or JSON and some other formats.

ZAP scans: https://github.com/1-8192/CS763_project/actions/workflows/zap.yml

(Optional) Perform additional security testing. Manually explore the project

One weakness I am surprised Sonarqube (SQ) did not catch: I have a hardcoded encryption key in the Github repository. Due to time constraints, I wasn't able to look into storing the key in Hashicorp Vault, or another secret manager. Storing the key in plaintext in the Github repository is a very big issue and would not be a production-ready implementation of the feature. I included it in the application.properties file to inject into context for now, but ideally I

would be loading that from a key manager rather than hard coding the string. Long-term, I would look into storing the key in vault and injecting it into application configs.

(Optional) Use any other security testing tools and report findings.

Did not do this section due to time constraints.

References

[1] “Java Rules.” *Sonarsource*. https://rules.sonarsource.com/java/RSPEC-5542/

[2] “Clickjacking” *OWASP docs.* https://owasp.org/www-community/attacks/Clickjacking