DDSS Assignment 2

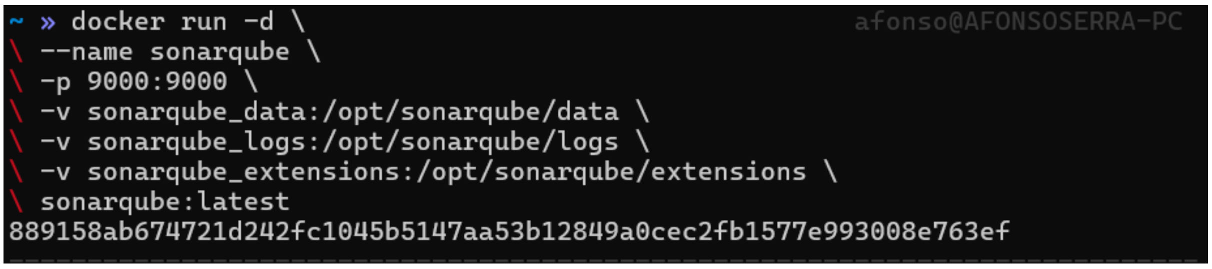
Afonso Serra – 2018279164

For this assignment I have used two *apache2* webservers, each corresponding to the correct and insecure versions of the same web app. The backend was coded with *php* as its easy to integrate it with this setup, and the vulnerabilities were introduced with it. Shipped with both web servers comes a shared, single PostgreSQL database with three tables: one for correct user data, one for incorrect user data and one for user posts. The key difference between the user tables is the password field – it is expected to be hashed in the former and plaintext in the latter, and this will be present in php code. These three instances that I've introduced are pulled as images from the Docker registry, with the apache servers being further built into adequately customized containers for the purpose of this assignment; they are all bundled within a docker-compose file that makes things easier.

Static Code Analysis

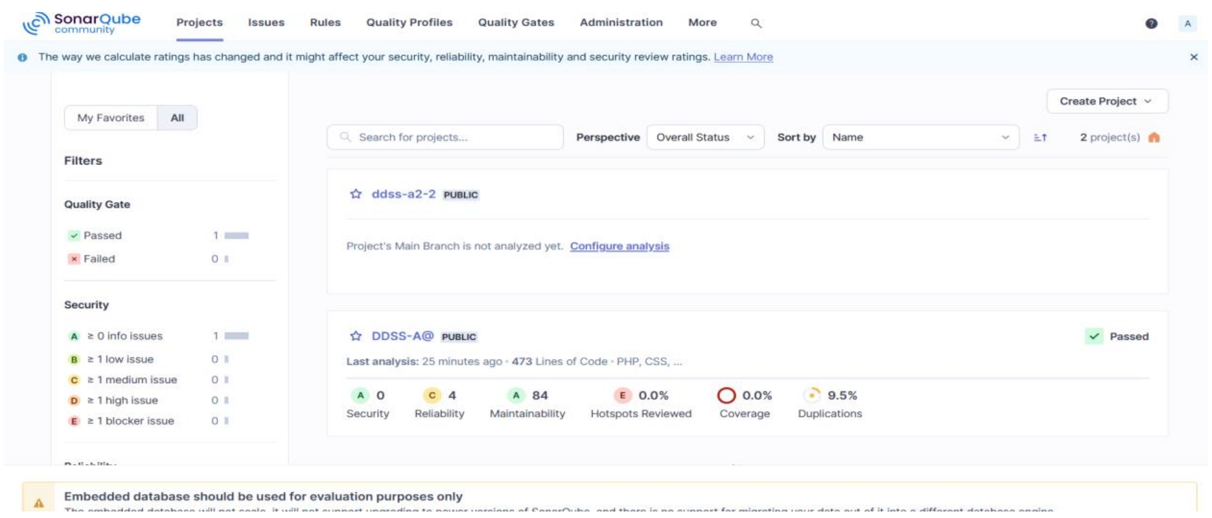
In this step I shall perform static code analysis to discover vulnerabilities that are evident from the source code alone. This can lead to false positives, but since there shall be more analysis these will likely be detected and handled. For this step, I have used *SonarQube* community edition which produced some results. These are the steps that I followed:

1. Pull and configure sonarqube:latest image



```
~ » docker run -d \                                   afonso@AFONSOSERRA-PC
\ --name sonarqube \
\ -p 9000:9000 \
\ -v sonarqube_data:/opt/sonarqube/data \
\ -v sonarqube_logs:/opt/sonarqube/logs \
\ -v sonarqube_extensions:/opt/sonarqube/extensions \
\ sonarqube:latest
889158ab674721d242fc1045b5147aa53b12849a0cec2fb1577e993008e763ef
```

2. Run it and access the provisioned web service at localhost:9000



The way we calculate ratings has changed and it might affect your security, reliability, maintainability and security review ratings. Learn More

Filters

Quality Gate

✓ Passed	1
✗ Failed	0

Security

A ≥ 0 info issues	1
B ≥ 1 low issue	0
C ≥ 1 medium issue	0
D ≥ 1 high issue	0
E ≥ 1 blocker issue	0

ddss-a2-2 PUBLIC

Project's Main Branch is not analyzed yet. Configure analysis

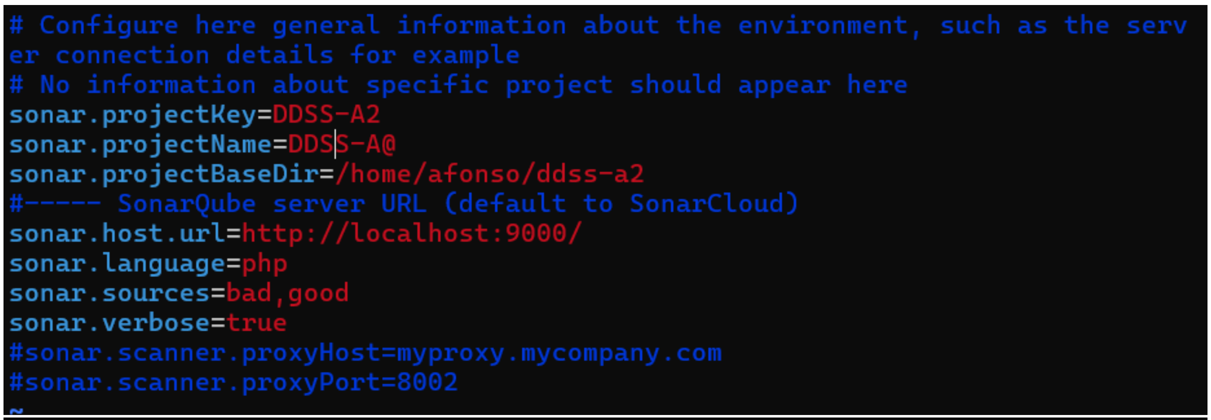
DDSS-A@ PUBLIC — ✓ Passed

Last analysis: 25 minutes ago · 473 Lines of Code · PHP, CSS, ...

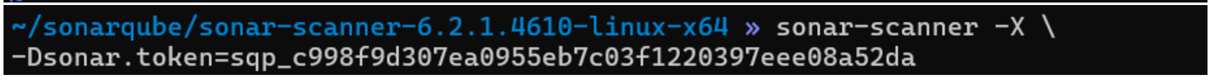
Security	Reliability	Maintainability	Hotspots Reviewed	Coverage	Duplications
A 0	C 4	A 84	E 0.0%	0.0%	9.5%

Embedded database should be used for evaluation purposes only

3. Create a SonarQube project and refer source code from local path
4. Unzip the SonarQube-Scanner CLI tool, configure it and run it. The configuration enables it to communicate with the server from the steps above, so the results from this scanner are fed to it



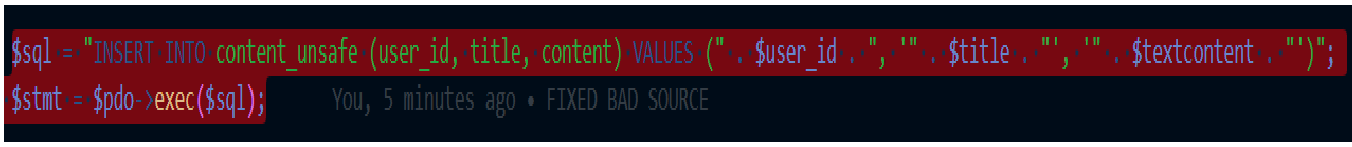
```
# Configure here general information about the environment, such as the serv
er connection details for example
# No information about specific project should appear here
sonar.projectKey=DDSS-A2
sonar.projectName=DDSS-A@
sonar.projectBaseDir=/home/afonso/ddss-a2
#----- SonarQube server URL (default to SonarCloud)
sonar.host.url=http://localhost:9000/
sonar.language=php
sonar.sources=bad,good
sonar.verbose=true
#sonar.scanner.proxyHost=myproxy.mycompany.com
#sonar.scanner.proxyPort=8002
~
```

5. 

```
~/sonarqube/sonar-scanner-6.2.1.4610-linux-x64 » sonar-scanner -X \
-Dsonar.token=sqp_c998f9d307ea0955eb7c03f1220397eee08a52da
```

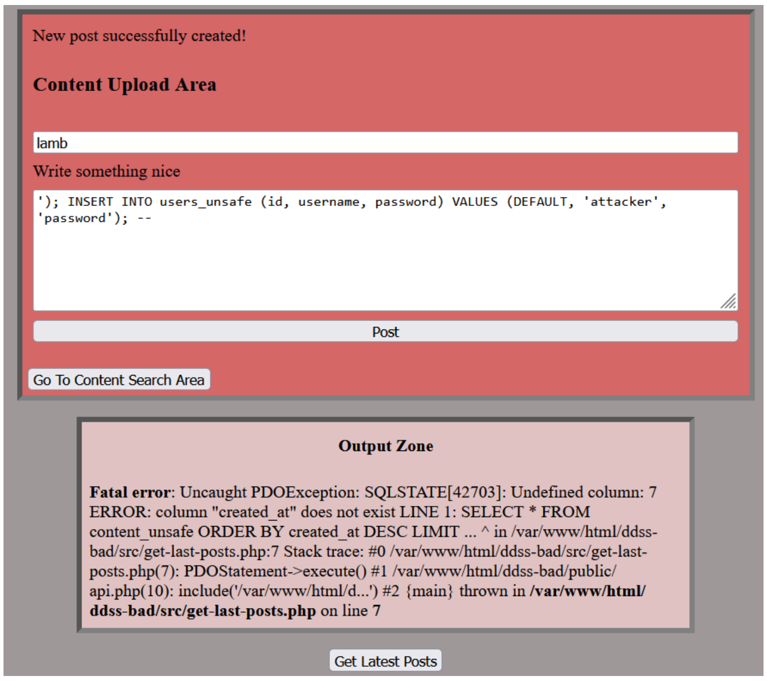
SonarQube did not properly identify all errors I expected it to even though it analyzed both good/ and bad/ sources. I know this isn't supposed to happen, so I took my analysis elsewhere since I know for a fact there are serious vulnerabilities in there. I suspect that it failed to identify such vulnerabilities because I'm using a less common approach to php + postgres integration with a PDO. I performed a manual SQL injection which allowed me to create a new user arbitrarily:

From:



```
$sql = "INSERT INTO content_unsafe (user_id, title, content) VALUES (" . $user_id . ", '" . $title . "', '" . $textcontent . "')";
$stmt = $pdo->exec($sql);        You, 5 minutes ago • FIXED BAD SOURCE
```

I wrote:



New post successfully created!

Content Upload Area

lamb

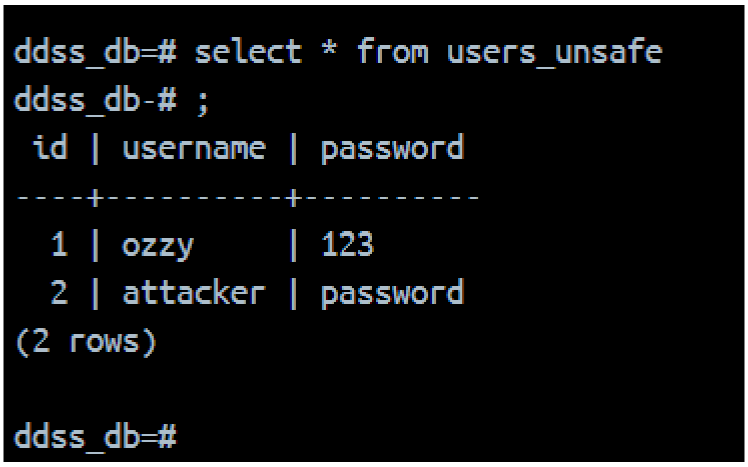
Write something nice

```
'); INSERT INTO users_unsafe (id, username, password) VALUES (DEFAULT, 'attacker', 'password'); --
```

Output Zone

Fatal error: Uncaught PDOException: SQLSTATE[42703]: Undefined column: 7 ERROR: column "created_at" does not exist LINE 1: SELECT * FROM content_unsafe ORDER BY created_at DESC LIMIT ... ^ in /var/www/html/ddss-bad/src/get-last-posts.php:7 Stack trace: #0 /var/www/html/ddss-bad/src/get-last-posts.php(7): PDOStatement->execute() #1 /var/www/html/ddss-bad/public/api.php(10): include('/var/www/html/d...') #2 {main} thrown in **/var/www/html/ddss-bad/src/get-last-posts.php** on line 7

Which resulted in:

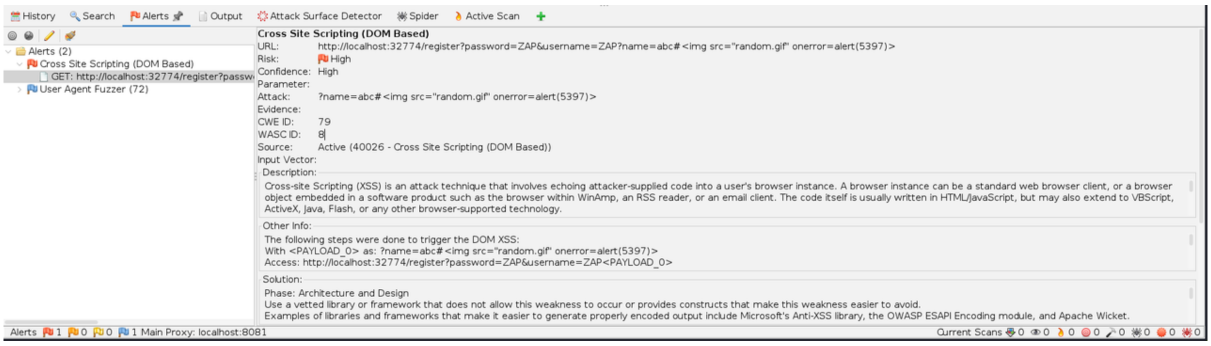


```
ddss_db=# select * from users_unsafe
ddss_db-# ;
 id | username | password
----+----------+----------
  1 | ozzy     | 123
  2 | attacker | password
(2 rows)

ddss_db=#
```

Dynamic Analysis

Performing an active scan with ZAP produced a few results; a serious XSS vulnerability was found:

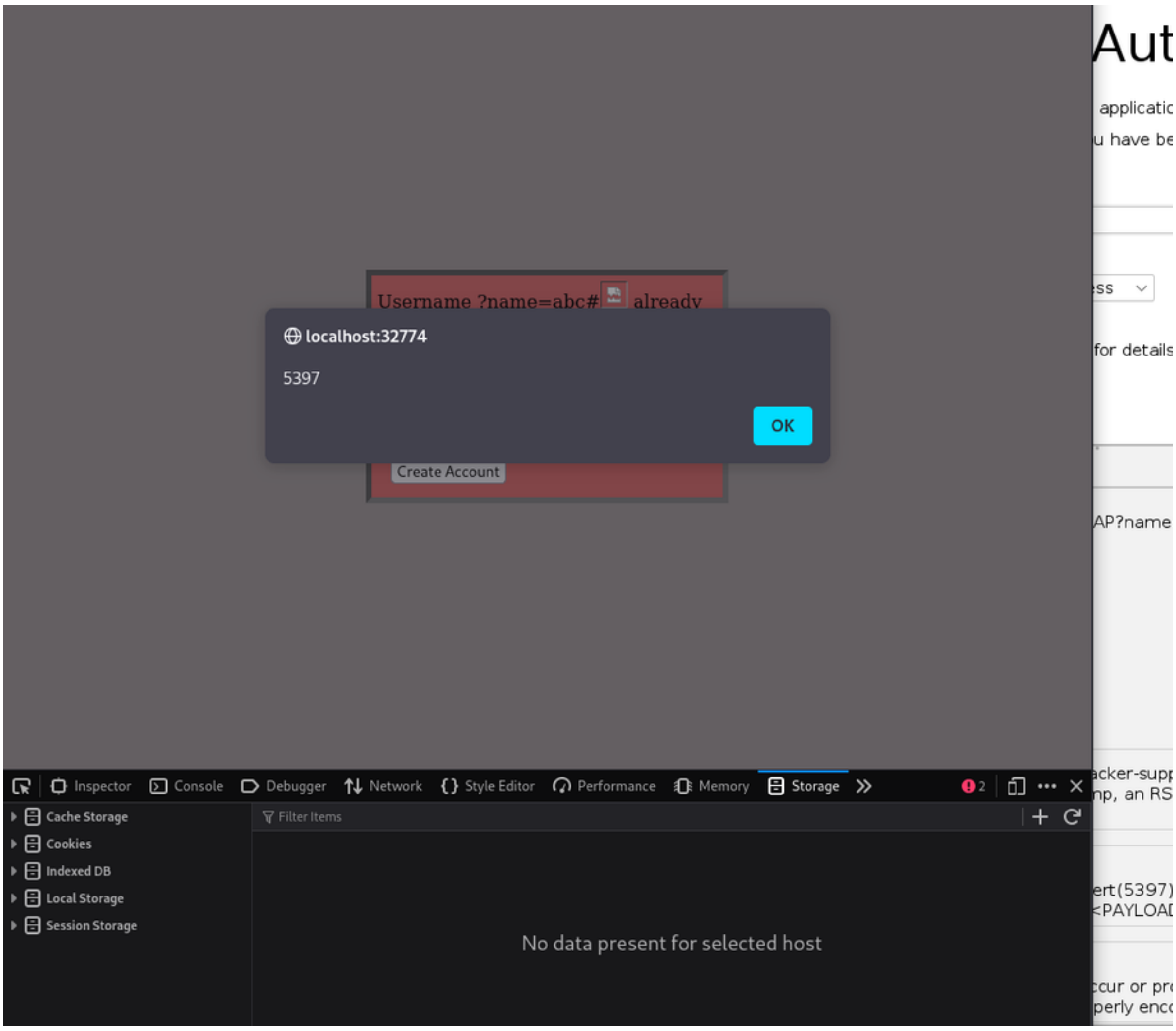


```
Alerts (2)
  Cross Site Scripting (DOM Based)
    GET: http://localhost:32774/register?passw
  User Agent Fuzzer (72)
```

```
Cross Site Scripting (DOM Based)
URL:          http://localhost:32774/register?password=ZAP&username=ZAP?name=abc#<img src="random.gif" onerror=alert(5397)>
Risk:         High
Confidence:   High
Parameter:
Attack:       ?name=abc#<img src="random.gif" onerror=alert(5397)>
Evidence:
CWE ID:       79
WASC ID:      8
Source:       Active (40026 - Cross Site Scripting (DOM Based))
Input Vector:
Description:
Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.
Other Info:
The following steps were done to trigger the DOM XSS:
With <PAYLOAD_0> as: ?name=abc#<img src="random.gif" onerror=alert(5397)>
Access: http://localhost:32774/register?password=ZAP&username=ZAP<PAYLOAD_0>
Solution:
Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.
```

```
Alerts 1 0 0 1 Main Proxy: localhost:8081                Current Scans 0 0 0 0 0 0 0 0
```

After running it myself, manually, I can confirm it:



I set the password as "ZAP" and the username as abc#<img src="random.gif" onerror=alert(5397)> (as per zap's automated scan)

Why SonarQube?

It has a free version and offers a very complete service. It also offers a wide array of options to integrate with, such as Git, GitLabs, Bitbucket, etc. In order to grasp this, I configured Git Actions' CI/CD which wasn't very useful because the free edition of this program only allows for local hosting – if they offered hosting for free then my .yml file would consist only of two steps; because of this limitation I would have to come up with several more steps which I opted out of, working offline instead.

Regardless, the output is very clean and not limited to security flaws (which it failed to report anyway) – it also gives insight about maintainability and redundant code, which are also important. The documentation is great. It's probably not the best option for a PHP based project, since there's a lot of hints of it being better suited for Java code bases. Were I to do this again, I would set up PHP Composer and use one of its many code analyzers.

Why ZAP?

I had already used ZAProxy in another class and enjoyed it very much. It is a complete set of tools to do this type of analysis, with lots of automated work taking place too. It also has a static code analyzer which unfortunately doesn't include PHP in its targets – but if I worked with Java instead, for example, then ZAP would have everything I need. Anyway, with ZAP I was able to draw the most conclusions including some that I was not expecting, such as XSS ones. The bad version of the app revealed 3 different XSS vulnerabilities while the good one only revealed a single one. Not only that, but ZAP was capable of finding another type of vulnerability of the type *Parameter Tampering*, in which a particular parameter was found to make the server return internal information about its runtime environment.

I tried to run some *sqlmap* scans to confirm that my sql vulnerabilities were *adequately placed* but I couldn't get this to work, and I believe it's due to reasons I don't completely understand yet. A good starting point to begin understanding could be the fact that I chose an odd structure for both applications; another point could be that I minimally used a frontend framework that could have confused this tool by introducing custom headers and somewhat protecting the input fields (htmx) – I'm not entirely sure though, as I believe this tool should have been impeccable at finding these vulnerabilities.