Experiment No. 5

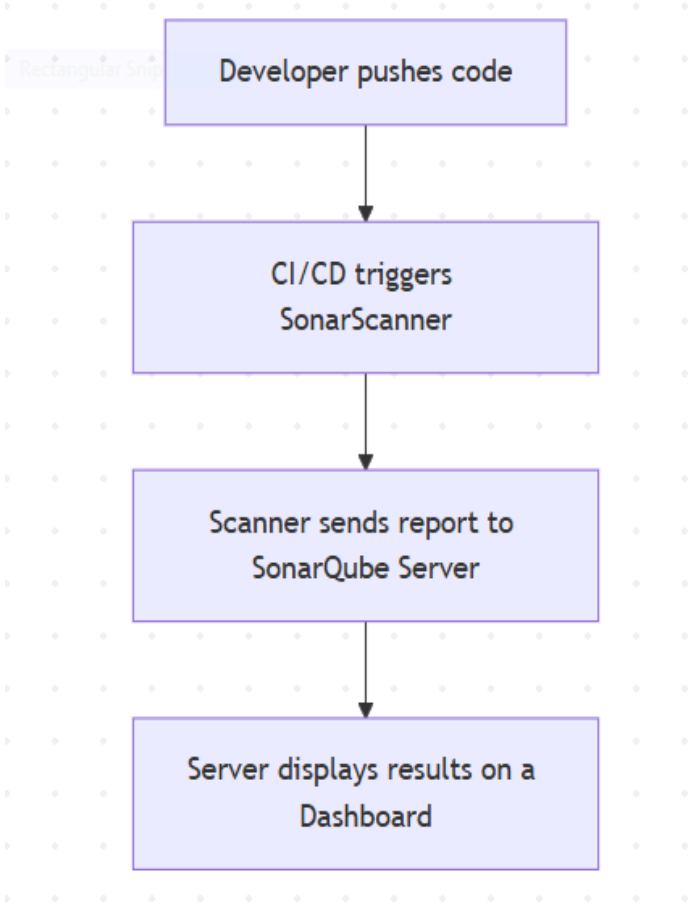
Aim: To perform static code analysis using tools like SonarQube to identify insecure code patterns.

Learning Objective: To perform static code analysis using tools like SonarQube to identify insecure code patterns to prevent vulnerabilities.

Theory:

Static Code Analysis, also known as Static Application Security Testing (SAST), is a security methodology that analyzes an application's source code for vulnerabilities **without executing it**. It's like having an automated expert proofread your code for security mistakes before you even try to run it. This is a "white-box" testing method because it requires access to the underlying source code.

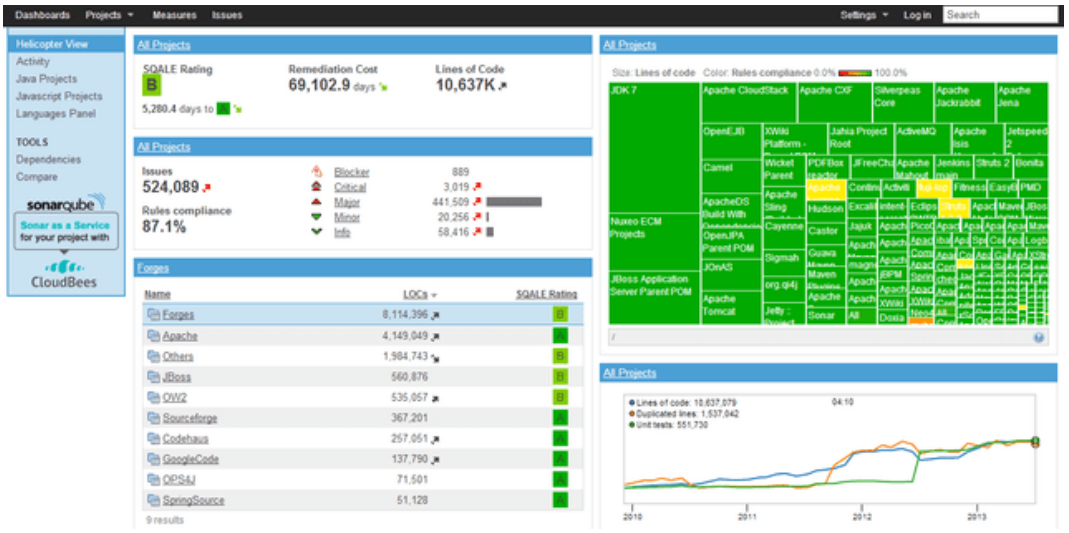
- **Why is SAST important?** It helps developers "shift left," which means finding and fixing security issues early in the development process. Catching a vulnerability during development is significantly cheaper and easier to fix than discovering it in a live production environment after a security breach.
- **What is SonarQube? SonarQube** is a leading open-source platform for continuous inspection of code quality and security. It automates code reviews by using static analysis to detect bugs, code smells (patterns that indicate deeper problems), and critical security vulnerabilities across more than 25 programming languages.
- **How SonarQube Works** The process involves two main components:
    1. **SonarScanner**: A command-line tool that analyzes your project's source code.
    2. **SonarQube Server**: A web server that receives the analysis report from the scanner, processes it, and displays the results on a detailed, interactive dashboard. It categorizes issues into **Bugs**, **Vulnerabilities** (e.g., SQL Injection), **Security Hotspots** (code that is security-sensitive and requires manual review), and **Code Smells**.



This procedure uses Docker to simplify the SonarQube setup, making it accessible for a local experiment.

1. **Set Up the SonarQube Environment** First, ensure you have Docker installed. Run a single command in your terminal to download and start the SonarQube Community Edition server. This will make the SonarQube dashboard available in your browser.
2. **Prepare the Vulnerable Code** Create a new project folder. Inside it, write a simple application file (e.g., a Python script) that contains several common and easily identifiable security vulnerabilities, such as SQL Injection, a hardcoded password, and command injection.
3. **Download and Configure SonarScanner** Download the SonarScanner command-line tool from the official SonarQube website and unzip it. In your project folder, create a configuration file named sonar-project.properties to tell the scanner about your project's key and where to find the source code.

4. **Run the Analysis** Open a terminal in your project's root directory and execute the sonar-scanner command. This will analyze the vulnerable code file and send the results to your local SonarQube server.
5. **Review the Results on the Dashboard** Navigate to the SonarQube dashboard in your web browser (typically http://localhost:9000). Log in with the default credentials, locate your newly created project, and explore the "Issues" tab to see the detailed list of vulnerabilities SonarQube has automatically detected.



Here are the setup commands and code needed for the experiment.

1. Start SonarQube with Docker

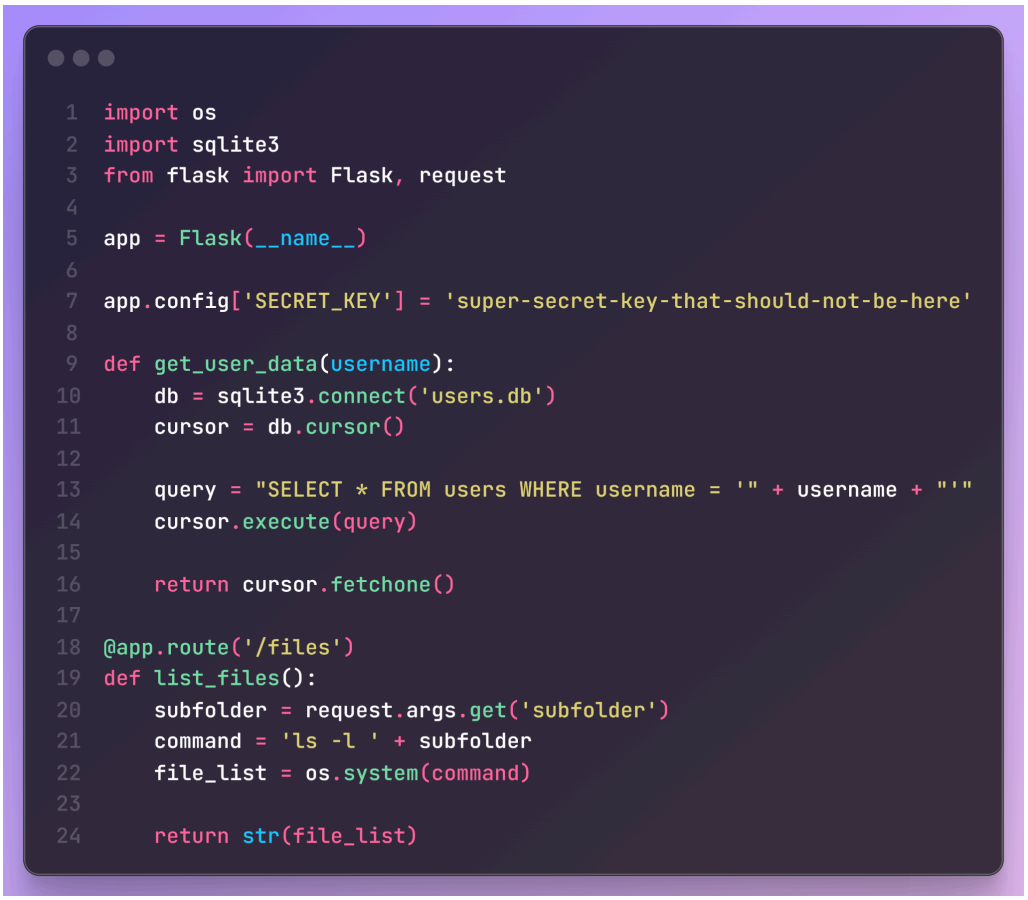
Open your terminal and run this command. It may take a few minutes for the server to be ready.

```
docker run -d --name sonarqube -p 9000:9000 -p 9092:9092 sonarqube:community
```

Once started, access the dashboard at http://localhost:9000 (login: admin, password: admin).

2. Create Vulnerable Code

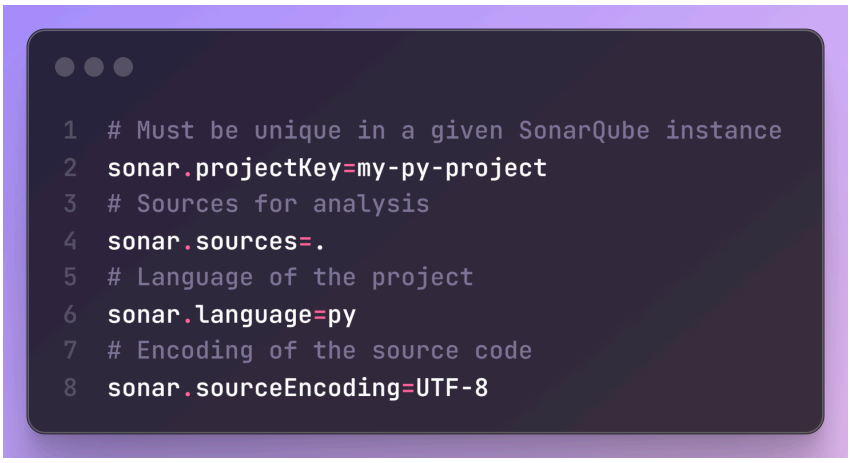
Create a folder named my-vulnerable-project and inside it, create a file named insecure_app.py:



```python
import os
import sqlite3
from flask import Flask, request

app = Flask(__name__)

app.config['SECRET_KEY'] = 'super-secret-key-that-should-not-be-here'

def get_user_data(username):
    db = sqlite3.connect('users.db')
    cursor = db.cursor()

    query = "SELECT * FROM users WHERE username = '" + username + "'"
    cursor.execute(query)

    return cursor.fetchone()

@app.route('/files')
def list_files():
    subfolder = request.args.get('subfolder')
    command = 'ls -l ' + subfolder
    file_list = os.system(command)

    return str(file_list)
```

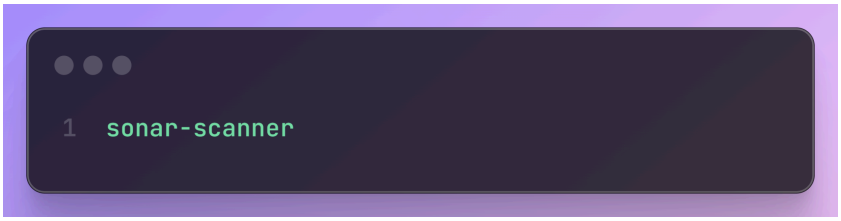
3. Configure SonarScanner

create this file: sonar-project.properties



```
# Must be unique in a given SonarQube instance
sonar.projectKey=my-py-project
# Sources for analysis
sonar.sources=.
# Language of the project
sonar.language=py
# Encoding of the source code
sonar.sourceEncoding=UTF-8
```

4. Run the Scan



Learning Outcome: Upon completing this experiment, you will understand the principles of Static Application Security Testing (SAST) and its role in identifying vulnerabilities early in the development lifecycle. You will gain practical experience using a SAST tool like SonarQube to scan code, interpret security reports, and identify common insecure patterns like SQL injection or hardcoded secrets.

Conclusion: This experiment successfully demonstrated the power and simplicity of using a SAST tool like SonarQube to automatically identify critical security vulnerabilities directly from source code. By scanning a deliberately insecure application, we saw how SonarQube pinpointed exact issues like SQL Injection and hardcoded secrets without ever running the program. Integrating static analysis into the development workflow provides developers with immediate, actionable feedback, allowing them to fix security flaws proactively and build more secure applications from the ground up.

Name:
Class: BE-CSE
Roll No.:

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				