National College of Ireland

Bachelor of Science (Honours) In Computing

Cybersecurity

2023/2024

Sean Carr

x20508479

x20508479@student.ncirl.ie

SimpleScan

Technical Report

Contents

Executive Summary

This Report shows how I developed and deployed my web-based web app and network scanning tool that can be used to increase security of websites or networks for business owners or individuals, this project aims to make a simple user-friendly experience for people looking to perform security scans such as spidering or active scans on websites using ZAP, Aswell as performing networks scans on networks using Nmap.

The Key functionalities of my tools is that you can monitor different types of scans you have made, view the scan results, and generate reports in XML.All functionality is managed by sessions and secure user authentication which also allows you to find scan results you have made in the past. OWASP Zap uses its API to perform the security assessments while Nmap is used to perform network scans.

This project demonstrates how to perform network and web security scans through a web app in simple way while also being secure and making results easily accessible. I would like to expand the project in the future to make it more comprehensive and add more features.

1.0 Introduction

1.1. Background

I undertook this project so I could develop an accessible tool that can be used by a business or individual to perform detailed scans of a web app or a network, in my internship and part time work I see how useful these tools are at finding vulnerabilities so I made a way anyone can use them withing their application.

1.2. Aims

The project aims to provide a web application that allows the user to perform automated web scans by spidering then actively scanning the website, It allows network

scans to be perform to identify open ports and services, it can then parse these into the website from xml and allows the user to see the results in a more digestible way.

1.3. Technology

The application uses numerous technologies to work:

Python and flask: Used for server-side operations and requests.

HTML, CSS, Bootstrap: Used to design a nice-looking user-friendly frontend.

JavaScript: Enhances how to frontend interacts with the backend and updates.

PostgreSQL and SQL alchemy: The database system and ORM for managing the scan results and data from the web and network scans.

OWASP Zap API: performs the web scanning via a local zap API in daemon mode.

Nmap: performs network scans via subprocess in python code.

Jinja2: Uses “blocks” to display content from the server side.

Amazon EC2 : Hosts and serves all apps on the web

1.4. Structure

Executive summary: summarises key points of the project.

Introduction: Outlines backgrounds, aims, technology and structure of the document

System Design: Shows architecture of the system, how data flows and how components interact.

Implementation: Details Implementation of the system features, shows code snippets and how the methodologies work.

Testing and results: Explains testing approaches used and the outcomes.

Future Development: Suggest improvements for the project in the future.

Conclusion: Summarises what the project achieved and how it affects the users

Appendices and references: Additional info such as reports and ethics applications or references etc.

2.0 System

2.1. Requirements

2.1.1. Functional Requirements

2.1.1.1. Use Case Diagram

2.1.1.2. Requirement 1 User Registration

2.1.1.3. Description & Priority

This requirement allows new users to register for the web application, it ii used for user management and access control It is extremely high priority to keep to app secure.

2.1.1.4. Use Case

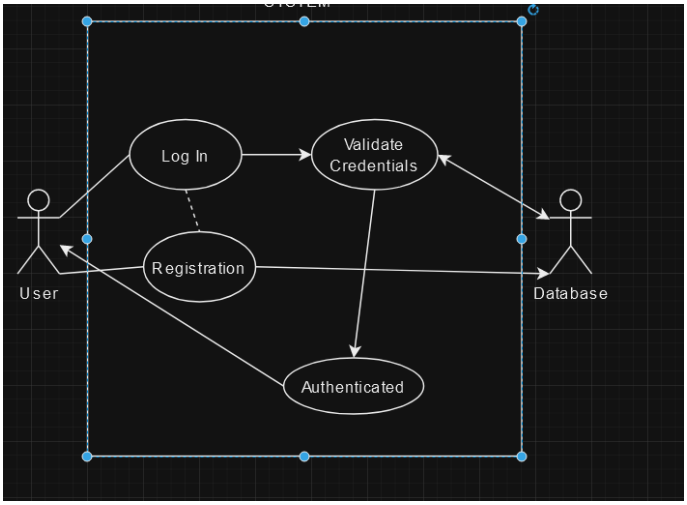
Scope

The scope of this use case is to register a new user to the system

Description

This use case describes the way which a new user can create an account within the system

Use Case Diagram



Flow Description

Precondition

The system is operational and accessible from a web browser

Activation

This use case starts when a visitor accesses the registration page

Main flow

1. The System displays the registration page.
2. The Visitor enters their desired username and password and clicks register

3. The system validates the credentials and then creates a new account
4. User is redirected to the login page

Alternate flow

None

Exceptional flow

E1 : Username already exists

1. The system detects the username is taken.
2. System displays an error message.
3. Use case restarts at step 1 main flow.

Termination

The User is redirected to the login page after successful registration or stays on registration if error

Post condition

System is ready for new requests

2.1.1.5. Requirement 2 User Login

2.1.1.6. Description & Priority

This requirement allows existing users to log in to the system , it is used to access features of the app, It is very high priority to keep to app secure.

2.1.1.7. Use Case

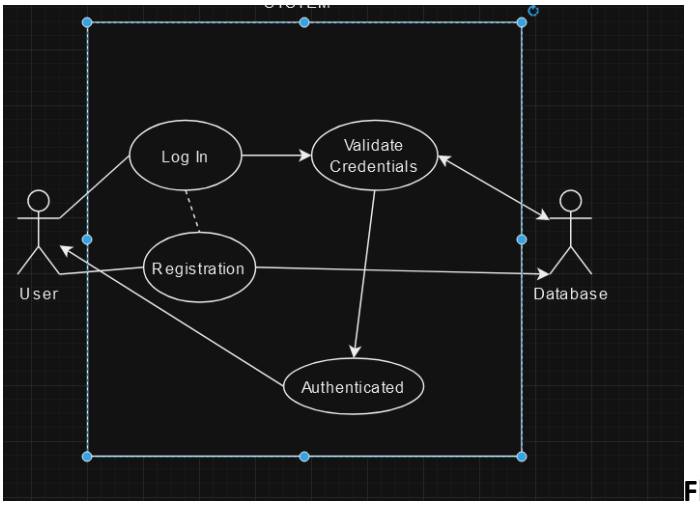
Scope

The scope of this use case is to authenticate a user then authenticate them to the system

Description

This use case describes the way which a new user can create an account within the system

Use Case Diagram



Flow Description

Precondition

The User must be registered in the system.

Activation

This use case starts when a visitor accesses the login page.

Main flow

1. The System displays the login page.
2. The Visitor enters their registered username and password and clicks login.
3. The system validates the credentials.
4. User is redirected to the Dashboard.

Alternate flow

None

Exceptional flow

E1 : Username already exists

1. The system detects the credential hashes do not match.
2. System displays an error message.
3. Use case restarts at step 1 main flow.

Termination

The User is redirected to the dashboard after successful login or remains on page after failing to login

Post condition

System displays dashboard with past user scans

2.1.1.8. Requirement 3 Network Scanning

2.1.1.9. Description & Priority

This requirement allows authenticated users to perform network scan on selected ips,it is a high priority functionality as it one of the core features.

2.1.1.10. Use Case

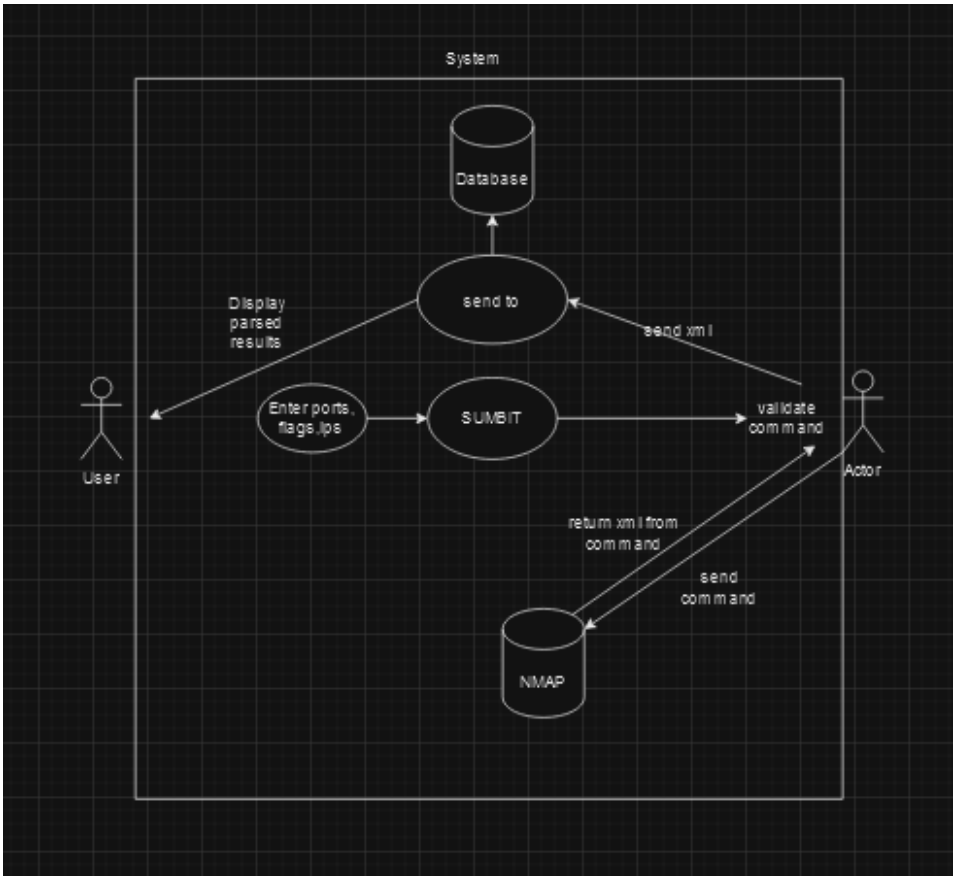
Scope

The scope of this use case is to perform network scans and display parsed results

Description

This use case describes the way which a user can initiate a network scan and view the results

Use Case Diagram



Flow Description

Precondition

The User must be logged in to the system.

Activation

This use case starts when a visitor accesses the network page.

Main flow

1. The system displays the page
2. User enters the Ip addresses ,ports and flags(A1)
3. System executes the scan with the parameters(E1)
4. System displays the scan results parsed

Alternate flow

A1:No inputs for flags or ports

1. User does not enter any ports or flags
2. System uses defaults flag and ports
3. System proceeds at step 3 of main flow

Exceptional flow

E1 : Scan Failure

1. The system encounters an error during scan
2. System displays an error message.
3. Use case restarts at step 1 main flow.

Termination

Scan results are displayed or an error message is shown

Post condition

The system is ready for another scan

2.1.1.11. Requirement 4 Start Web spider scan

2.1.1.12. Description & Priority

This requirement allows existing users to start a web spider scan on selected urls.It is a core functionality so high priority

2.1.1.13. Use Case

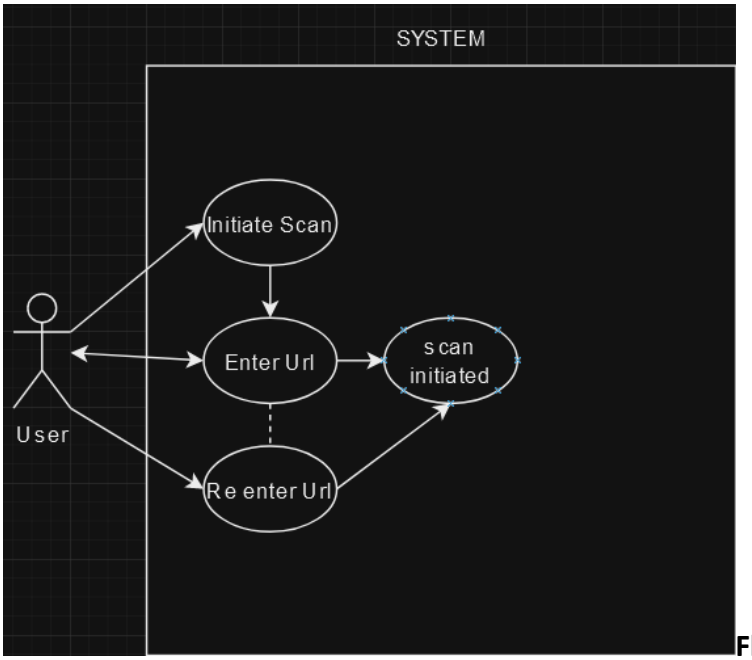
Scope

The scope of this use case is to perform a spider scan on a specified url

Description

This use case describes how a user can initiate a spider scan to map out the sitemap and links of a site

Use Case Diagram



Flow Description

Precondition

The User must be logged into the system.

Activation

This use case starts when a user access the web scan page and wants to perform a spider scan.

Main flow

1. The user enters their specified url into the input field
2. They click the spider scan button
3. System sends the url to the backend and zap api initiates the scan(E1)
4. System monitors when the scan is finished and displays when finished

Alternate flow

None

Exceptional flow

E1 : Url not supplied

1. The url is not provided by the user
2. Systems displays error message
3. Restart at main flow 1

Termination

The Spider scan successfully finishes or an error message is thrown

Post condition

User can initiate other scans or review results

2.1.1.14. Requirement 5 Start Web active scan

2.1.1.15. Description & Priority

This requirement allows existing users to start a web active scan on selected urls.It is a core functionality so high priority

2.1.1.16. Use Case

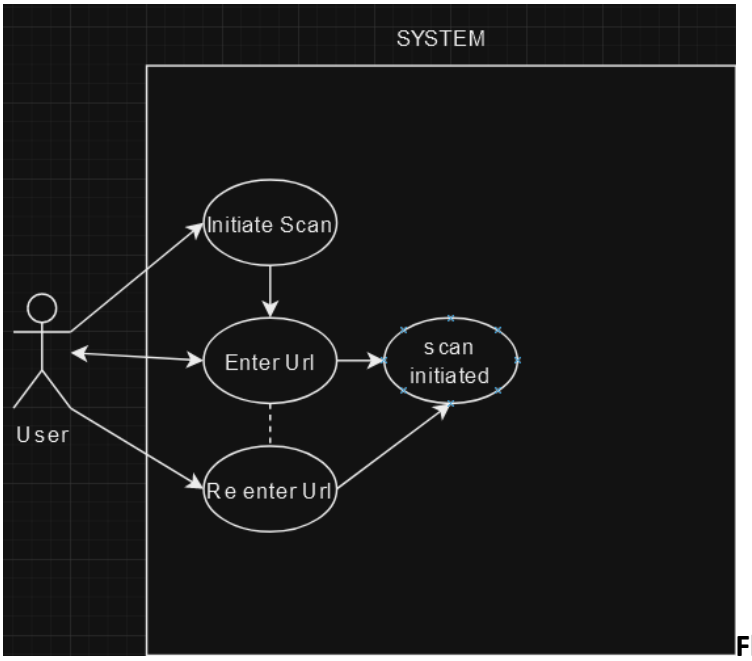
Scope

The scope of this use case is to perform active scan on a specified url

Description

This use case describes how a user can initiate a active scan to find and detect vulnerabilities within the application

Use Case Diagram



Flow Description

Precondition

The User must be logged into the system.

Activation

This use case starts when a user access the web scan page and wants to perform a active scan.

Main flow

1. The user enters their specified url into the input field
2. They click the active scan button
3. System sends the url to the backend and zap api initiates the scan if a spider scan has been performed(E1)
4. System monitors when the scan is finished and saves results when finished

Alternate flow

None

Exceptional flow

E1 : Url not supplied

4. The url is not provided by the user
5. Systems displays error message
6. Restart at main flow 1

E2:spider scan not performed

Spider scan has not been performed

System displays that spider scan needs to be performed

Termination

The Spider scan successfully finishes or an error message is thrown

Post condition

User can initiate other scans or review results

2.1.1.17. Requirement 6 View scan Results

2.1.1.18. Description & Priority

This requirement allows users to view results of previous scans they have made ,It is good to help track past scans ,medium priority.

2.1.1.19. Use Case

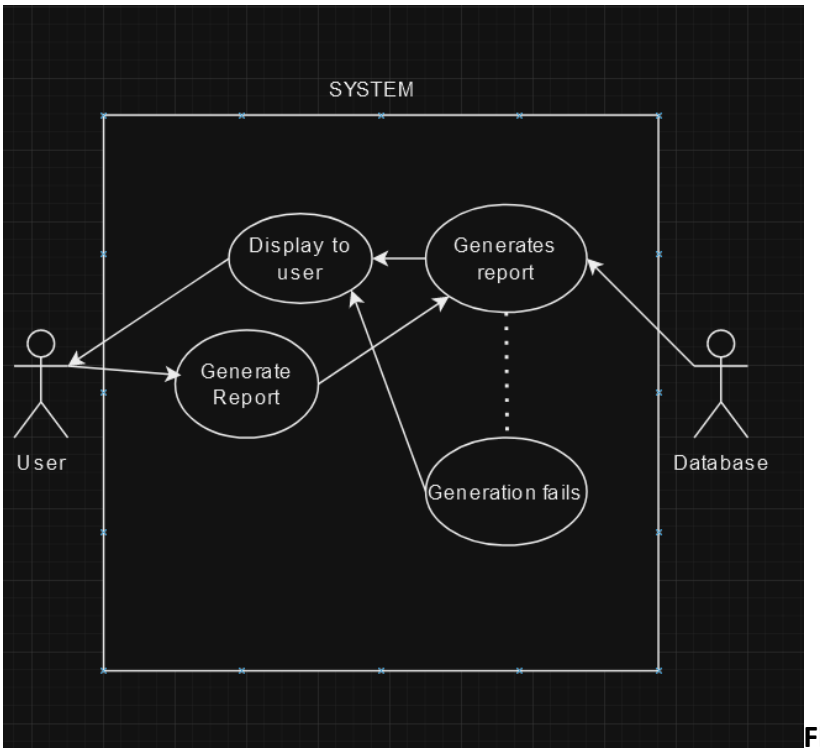
Scope

The scope of this use case is to retrieve and display scan results

Description

This use case describes how a user can view detailed scan results from previous network or web scans

Use Case Diagram



Flow Description

Precondition

The User must be logged in and have made previous scans on the system.

Activation

This use case starts when a user navigates to a scan results page

Main flow

1. The System displays the list of scanned ips or scanned webpages

2. User selects a specific scan to view the results
3. The system retrieves and displays the scan details(E1)

Alternate flow

None

Exceptional flow

E1 : No scans made/found

1. The system finds no scans have been made or found
2. System displays that no scan is found

Termination

Scan results are displayed or message is thrown

Post condition

User can view other results or log out

2.1.2. Data Requirements

Data Requirement 1: User Data

Description: The system must be able to securely store username and passwords

Fields:

Username: String

Password hash: String

Relationships: Users might have associated scan results

Storage: Data should be stored hashed in a database securely

Data Requirement 2: Scan Data

Description: System must store details of network and web scans made by users

Fields:
Scan id: String, unique identifier for each scan.
User id: Integer, foreign key linked to the User table, used as a session id.
Ip address: String (for network scans).
url: String (for web scans).
scan_type: String, type of scan ('network', 'web spider', 'active scan').
status: String, status of the scan ('started', 'in progress', 'completed', 'failed').
results: Text, JSON formatted string of scan results.

Xml report: Text, XML formatted report of the scan

Relationship: each scan is associated with a user

Storage: Scan data should be stored in a database with relationships to user ids and indexed to be quickly accessed when requested.

2.1.3. User Requirements

User Requirement 1: Ease of Use

Description: The system should be user friendly users should be able to use the system without full knowledge of the way the system works

How To:

Intuitive UI

Small number of steps to start scans / view results.

User Requirement 2: Performance

Description: The system should perform well under normal use, while able to perform multiple scans without lag

How To:

Response time should not exceed 5 seconds per request.

System should be scalable to 5 + scans performable at same time.

User Requirement 3: Security

Description: The system should be able to securely handle user data and not be vulnerable to common vulnerabilities

How to:

Secure authentication mechanisms

Encryption of sensitive data

Regular testing

2.1.4. Environmental Requirements

Web app should be running on a server with a dual core processor,have at least 8gb ram to perform multiple scans at same time,and then at least 100Gb of storage to store user scans.

The software must be comptabile to run on any popular operating system and then be accessible from all popular browsers such as chrome edge and firefox.Database must be running on Postgre SQL12 or later and must use python 3.8 with flask for backend.

For Hosting it must be ran on a hosting service that allows web scanning and networking scanning on it such as amazon web services ,hosting services like Heroku will not allow hosting of these tools as I have encountered before that it is not possible to host nmap and zap on services like this.Logs must be monitored to see if any malicious activity is taking place or if there is any outages etc.

Ther must be compliance with all standards ,we must comply with GDPR when handling personal data, security standards must also be followed for handling info about security of other systems.

All these requirements make sure that the hosting of the app go smooth and you do not run into any errors and then you comply with any security or compliance standards while being a reliable platform.

2.1.5. Usability Requirements

The systems should be running efficiently so that users can do what they want,it use shoulda s little resources ap possible while still being able to complete tasks,users should be able to learn the system easily and perform tasks quickly and system should be usable by anyone.

2.2. Design & Architecture

The system is a web app that is designed for web and network scanning,it uses flask and python for the backend as thee framework and language,and integrates ZAP API for security scans.The architecture is a Web server(nginx),Flask server,(Backend) database(PostgreSQL),and ZAP Api for web scanning.

The Web server(nginx and Flask)

- Handles HTTP Requests and responses
- serves my app,web pages and APIs
- handles user sessions and authentication.

My Database(PostgreSQL and SQLAlchemy)

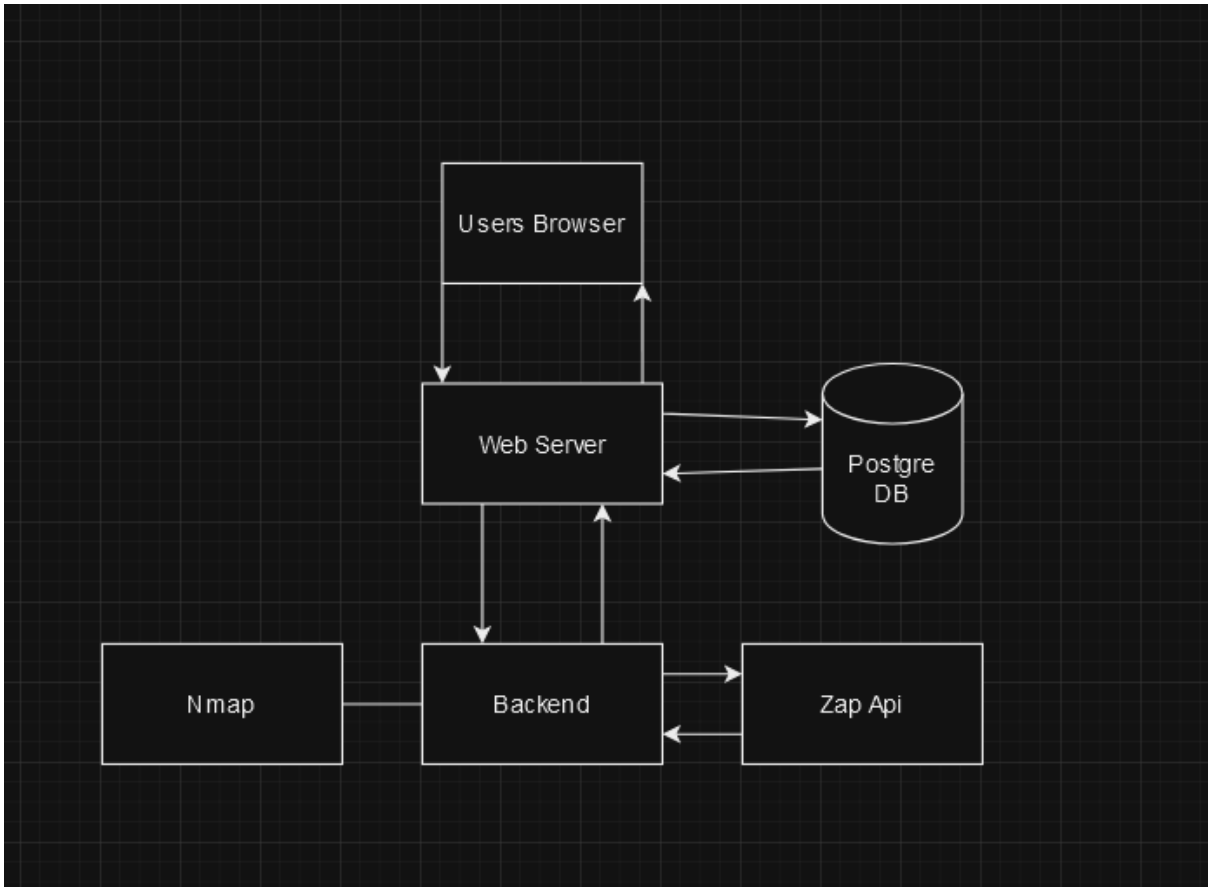
- Stores User Data,network scans and web scans
- Uses Models User,NetworkScan and ScanResult

ZAP API

- Used to initiate scans and give back results
- Interacts with frontend to perform scans when they are requested

Frontend(HTML/CSS/Javascript)

- Designs the User interface
- Uses ajax to display scan queries live for web scanning



2.3. Implementation

First of all I will go over how the file gets everything when the file is ran it sets up the initial configuration in __init__.py

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

import os
db = SQLAlchemy()
#import db and create app then configure routes and serve app to wsgi
def create_app():
    app = Flask(__name__)
    app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://postgres:Welcome181@database-2.cdco2ayo4wwh.eu-north-1.rds.amazonaws.com:5432'
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False


    db.init_app(app)


    from .routes import configure_routes
    configure_routes(app)

    return app
```

Here it imports flask sqlalchemy and then sets database then initialses the app then configures the routes

Wsgi.py then runs the app through the command

```python
from app import create_app

application = create_app()

if __name__ == "__main__":
    application.run()
```

gunicorn --timeout 120 --bind 0.0.0.0:8000 wsgi:application which serves the flask app to my amazon ec2 instance on the cloud.

The first thing the user sees is the login screen as they are not authenticated to the app yet

There is a jinja 2 block that gets all alerts on these pages and updates them when there is an alert through flasks alert feature.

When a user registers in the routes it takes the supplied username and password in the form

```python
@app.route('/register', methods=['GET', 'POST'])
def register_route():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        return register(username, password)
    else:
        return render_template('register.html')
```

```python
def register(username, password):
    # Check if user already exists
    if User.query.filter_by(username=username).first():
        flash('Username already exists.', 'error')
        return redirect(url_for('register_route'))

    # Create new user with hashed password
    new_user = User(username=username)
    new_user.set_password(password)
    db.session.add(new_user)
    db.session.commit()

    flash('User successfully registered!', 'success')
    return redirect(url_for('login_route'))
```

```python
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password_hash = db.Column(db.String(255))
    #sets and rreturns hash of password in db
    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

It then checks for an existing user then if none exists it send the password to the set password function which then hashes the password using werkzeug security hashing and stores it in the database then displays a success message and redirects to the login page so a user can log in

```python
from werkzeug.security import generate_password_hash, check_password_hash
```

Login works very similar but instead it just compares the password hashes to the ones in the database for security reasons.

Once the user is on the home page they are presented with a bootsrap navbar that is made using a default.html which is extended to make everything the same across all webpages as seen below

```
{% extends 'default.html' %}

{% block title %}Dashboard Home{% endblock %}

{% block content %}
<div class="container mt-4">
  <h2 class="text-center">Welcome to Simple Scan Dashboard</h2>
  <h4 class="mb-3">Scanned IPs:</h4>
  <div class="list-group">
    {% for ip in unique_ips %}
      <a href="{{ url_for('show_scans', ip_address=ip.ip_address) }}" class="list-group-item list-group-item-action list-group-item-dark">{{ ip.ip_address }}</a>
    {% endfor %}
  </div>
</div>
{% endblock %}
```

In the home page a user can see any previous network scans that have been made in the past as seen in the gui ,these scans are displayed in a separate page and are defined in the routes as seen

```
@app.route('/scans/<ip_address>')
def show_scans(ip_address):
    if 'user_id' not in session:
        return redirect(url_for('login_route'))

    scans = NetworkScan.query.filter_by(ip_address=ip_address, user_id=session['user_id']).all()
    if not scans:
        abort(404)  # Or handle the case where no scans are found differently
    return render_template('scans.html', scans=scans, ip_address=ip_address)
```

It uses jinja 2 to again loop through all of these scans from the user model based off of the userid session tracker then returns them as scans and ip addresses.

These scans are associated via a foreign key in the NetworkScan table that stores all scans from nmap and displays them on the scan page when requested through the functionality.

```
@app.route('/')
def home():
    if 'user_id' in session:
        unique_ips = NetworkScan.query.filter_by(user_id=session['user_id']).with_entities(NetworkScan.ip_address).distinct()
        return render_template('index.html', unique_ips=unique_ips)
    return redirect(url_for('login_route'))
```

```
class NetworkScan(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)  # Reference to the User model
    ip_address = db.Column(db.String(15), nullable=False)
    xml_data = db.Column(db.Text, nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)

    def __repr__(self):
        return f'<NetworkScan {self.ip_address}>'
```

Now we need to see how the actual network scan functionality is made.on the network scanning page we can input ports flags and ips on the form which is then sent to a subprocess of nmap which checks if there is any ports fields missing and if so replaces them with default settings as seen here

```
def run_nmap():
    if request.method == 'POST':
        ip_address = request.form['ip']
        ports = request.form['ports']
        flags = request.form['flags']
        if not ports:
            ports = '22,80,443'
        if not flags:
            flags = '-sV'
```

It will then input all of these into the nmap command with an xml output and then run the command via the subprocess as a result and will then parse these xml via our parse_nmap_xml()FUNCTION

```
command = f"nmap {flags} -p {ports} {ip_address} -oX -"
result = subprocess.run(command, shell=True, stdout=subprocess.PIPE, text=True)
scan_results = parse_nmap_xml(result.stdout)
```

The parse function we have in our network.py file will take the xml output and then find all the info we have defind it to find via etree

```
def parse_nmap_xml(xml_output):
    root = ET.fromstring(xml_output)
    scan_data = []
    for host in root.findall('host'):
```

,it will store address info such as ip and ports for the address section of the xml output and append them to the scan data array we have made

```
for host in root.findall('host'):
    for address in host.findall('address'):
        ip = address.get('addr')
        scan_data.append({'IP': ip, 'Ports': []})
```

then for the port sections it will take states and then info about what is happening on that port just like this

```
        for port in host.find('ports').findall('port'):
            portid = port.get('portid')
            state = port.find('state').get('state')
            service=port.find('service').get('name')
            product=port.find('service').get('product')
            version=port.find('service').get('version')
            scan_data[-1]['Ports'].append((portid, state, service,product, version)
```

it will then append it to the next part of the array and then return it to the runnmap function

```
def parse_nmap_xml(xml_output):
    root = ET.fromstring(xml_output)
    scan_data = []
    for host in root.findall('host'):
        for address in host.findall('address'):
            ip = address.get('addr')
            scan_data.append({'IP': ip, 'Ports': []})
        for port in host.find('ports').findall('port'):
            portid = port.get('portid')
            state = port.find('state').get('state')
            service=port.find('service').get('name')
            product=port.find('service').get('product')
            version=port.find('service').get('version')
            scan_data[-1]['Ports'].append((portid, state, service,product, version)
    return scan_data
```

It then displays this on the network page when we first complete the scan

After it will save to the NetworkScan model table with the raw xml data and associate it to the userid of the user logged in with the ip address also

```
def run_nmap():
    if request.method == 'POST':
        ip_address = request.form['ip']
        ports = request.form['ports']
        flags = request.form['flags']
        if not ports:
            ports = '22,80,443'
        if not flags:
            flags = '-sV'
        command = f"nmap {flags} -p {ports} {ip_address} -oX -"
        result = subprocess.run(command, shell=True, stdout=subprocess.PIPE, text=True)
        scan_results = parse_nmap_xml(result.stdout)

        # Save the XML output to the database
        new_scan = NetworkScan(user_id=session['user_id'], ip_address=ip_address, xml_data=result.stdout)
        db.session.add(new_scan)
        db.session.commit()

        return render_template('network.html', scan_results=scan_results)
    return render_template('network.html')
```

It all comes together to work like this.

Now for the Web scanning part

First of all it displays all the forms and buttons and hidden output

```
<div class="container">
  <!-- Form for starting scans -->
  <div class="form-custom">
    <input type="text" id="targetUrl" class="form-control" placeholder="Enter target URL">
    <button onclick="startSpider()" class="btn btn-primary">Start Spider Scan</button>
    <button onclick="startActiveScan()" class="btn btn-primary">Start Active Scan</button>
    <button onclick="generateXmlReport()" class="btn btn-primary">Generate Report</button>
    <pre id="output" class="result-box"></pre>


  <!-- Form for checking scan results -->

    <input type="text" id="scanId" class="form-control" placeholder="Enter scan Id">
    <button onclick="viewScanStatus()" class="btn btn-info">Check Scan Status</button>
    <button onclick="viewXmlReport()" class="btn btn-info">View Report</button>
    <div id="scanStatusDisplay" ></div>
    <pre id="xmlDisplay" class="result-box"></pre>
  </div>
</div>
```

The buttons start tasks in the backend

In the backend we have our zap api configured where it is running

```
from flask import request, jsonify,render_template,session
import requests
import json
from .models import db, ScanResult

ZAP_API_URL = "http://ec2-51-20-35-101.eu-north-1.compute.amazonaws.com:9090"
ZAP_API_KEY = "AMWFOWAIOFNA"
```

We boot the zap api by a command in our ec2 instance

./zap.sh -daemon -host 0.0.0.0 -port 9090 -config api.disablekey=false -config api.addrs.addr.name=.* -config api.addrs.addr.regex=true -config api.key=AMWFOWAIOFNA

It uses the zap.sh file to start in daemon mode on port 9090 with our api key set

The buttons we have set kick off certain functionality in the backend start spider and start active scan have the same functionality just different types of scans it send them to the backend like this

```
//starts spider function when the button is clicked in our backend
function startSpider() {
  startScan('/start_spider');
}
//starts active sscan in the backend when button clicked
function startActiveScan() {
  startScan('/start_active_scan');
}
```

Frontend to routes to web.py

```python
@app.route('/start_spider', methods=['POST'])
def start_spider_route():
    return start_spider()

@app.route('/start_active_scan', methods=['POST'])
def start_active_scan_route():
    return start_active_scan()
```

```python
def start_spider():
    return start_zap_scan(request, 'spider')

def start_active_scan():
    return start_zap_scan(request, 'ascan')

def start_zap_scan(request, scan_type):
    data = request.get_json()
    target_url = data.get('url')
    if not target_url:
        return jsonify({"error": "URL is required"}), 400
    zap_endpoint = f"{ZAP_API_URL}/JSON/{scan_type}/action/scan?apikey={ZAP_API_KEY}&url={target_url}"
    response = requests.get(zap_endpoint)
    scan_id = response.json().get('scan') if response.status_code == 200 else None
    return jsonify({"message": f"{scan_type.capitalize()} scan started", "scan_id": scan_id})
```

It will then get the data from the target url form through json and if it is not a url it will return an error

It will then set the endpoint as the scan type and send it using the target url and api key as seen

```python
zap_endpoint = f"{ZAP_API_URL}/JSON/{scan_type}/action/scan?apikey={ZAP_API_KEY}&url={target_url}"
response = requests.get(zap_endpoint)
```

And then use the request library to send a request to the api with the endpoint we made with the supplied information it will then get the response from zap and take the information that zap gives to us and then return it as seen in this snippet

```python
scan_id = response.json().get('scan') if response.status_code == 200 else None
return jsonify({"message": f"{scan_type.capitalize()} scan started", "scan_id": scan_id})
```

This then displays the response in the output field as seen.

```javascript
function startScan(apiEndpoint) {
  //gets the target url from the input field
  var targetUrl = document.getElementById('targetUrl').value;
  //sends the target url to the backend which then sends it to zap api with the required content header and method type in json
  fetch(apiEndpoint, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ url: targetUrl })
  })

    //gets response from backend
    .then(response => response.json())
    //gets output from backend and converts from json to string
    .then(data => document.getElementById('output').textContent = JSON.stringify(data, null, 2))
    //catches errors in console
    .catch(error => console.error('Error:', error));

}
```

This allows updating of the page with json and no need to refresh the page.

Scan Status is the same but just hits a different api endpoint and displays the scan status with inline html whenj it gets a json response

```
zap_endpoint = f"{ZAP_API_URL}/JSON/ascan/view/status/?apikey={ZAP_API_KEY}&scanId={scan_id}"
response = requests.get(zap_endpoint)
```

```
} else {
    document.getElementById('scanStatusDisplay').innerHTML = `<div class="alert alert-dark ">Scan Status: ${data.status}%</div>`;
}
```

Finally we have generation and viewing of xml reports for the scan

```
function generateXmlReport() {
    var scanId = document.getElementById('scanId').value;
    fetch('/generate_xml_report', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ scanId: scanId })
    })
    .then(response => response.json())
    .then(data => {
        if (data.message) {
            alert('XML Report Generated Successfully');
        } else {
            alert('Error: ' + data.error);
        }
    })
    .catch(error => console.error('Error:', error));
}

function viewXmlReport() {
    var scanId = document.getElementById('scanId').value;
    if (!scanId) {
        alert("Please enter a scan ID.");
        return;
    }
    fetch(`/view_scan_result?scanId=${scanId}`)
        .then(response => response.json())
        .then(data => {
            if (data.error) {
                alert("Error: " + data.error);
            } else {
                document.getElementById('xmlDisplay').textContent = data.xml_report;
            }
        })
        .catch(error => console.error('Error:', error));
}
```

When the user clicks the generate it send a post request to the backend that creates a zap api request similar to the previous ones

```python
def generate_xml_report(scan_id):
    data = request.get_json()
    scan_id = data.get('scanId')


    report_endpoint = f"{ZAP_API_URL}/OTHER/core/other/xmlreport/?apikey={ZAP_API_KEY}&scanId={scan_id}

    # Attempt to fetch the XML report from the ZAP API
    try:
        response = requests.get(report_endpoint)
        response.raise_for_status()  # Raises an HTTPError for bad responses
        xml_report = response.content
    except requests.RequestException as e:
        return jsonify({"error": str(e)}), 400

    # Parse the XML report
    import xml.etree.ElementTree as ET
    try:
        root = ET.fromstring(xml_report)
        # Process the XML data as needed
        processed_xml = ET.tostring(root, encoding='unicode')
    except ET.ParseError as e:
        return jsonify({"error": str(e)}), 400

    # Query the database for an existing scan result
    scan_result = ScanResult.query.filter_by(scan_id=scan_id).first()

    if scan_result:
        # If a scan result exists, update the xml_report
        scan_result.xml_report = processed_xml
    else:
        # If no scan result exists, create a new one
        scan_result = ScanResult(scan_id=scan_id, xml_report=processed_xml)
        db.session.add(scan_result)
```

It will send the report then save the content as xml_report it will then decode the xml as it is not readable if so,then it will check if there is a scan already and update it if so and if not then it will create the new scan and ad it to the db,it will then try to commit and return the json response whether it is committed if not to the frontend

```javascript
.then(response => response.json())
.then(data => {
    if (data.message) {
        alert('XML Report Generated Successfully');
    } else {
        alert('Error: ' + data.error);
    }
})
.catch(error => console.error('Error:', error));
```

all of these scans are then acceesible via the view scan report button which will kick off this functionality in the backend

```
function viewXmlReport() {
    var scanId = document.getElementById('scanId').value;
    if (!scanId) {
        alert("Please enter a scan ID.");
        return;
    }
    fetch(`/view_scan_result?scanId=${scanId}`)
        .then(response => response.json())
        .then(data => {
            if (data.error) {
                alert("Error: " + data.error);
            } else {
                document.getElementById('xmlDisplay').textContent = data.xml_report;
            }
        })
        .catch(error => console.error('Error:', error));
}
```

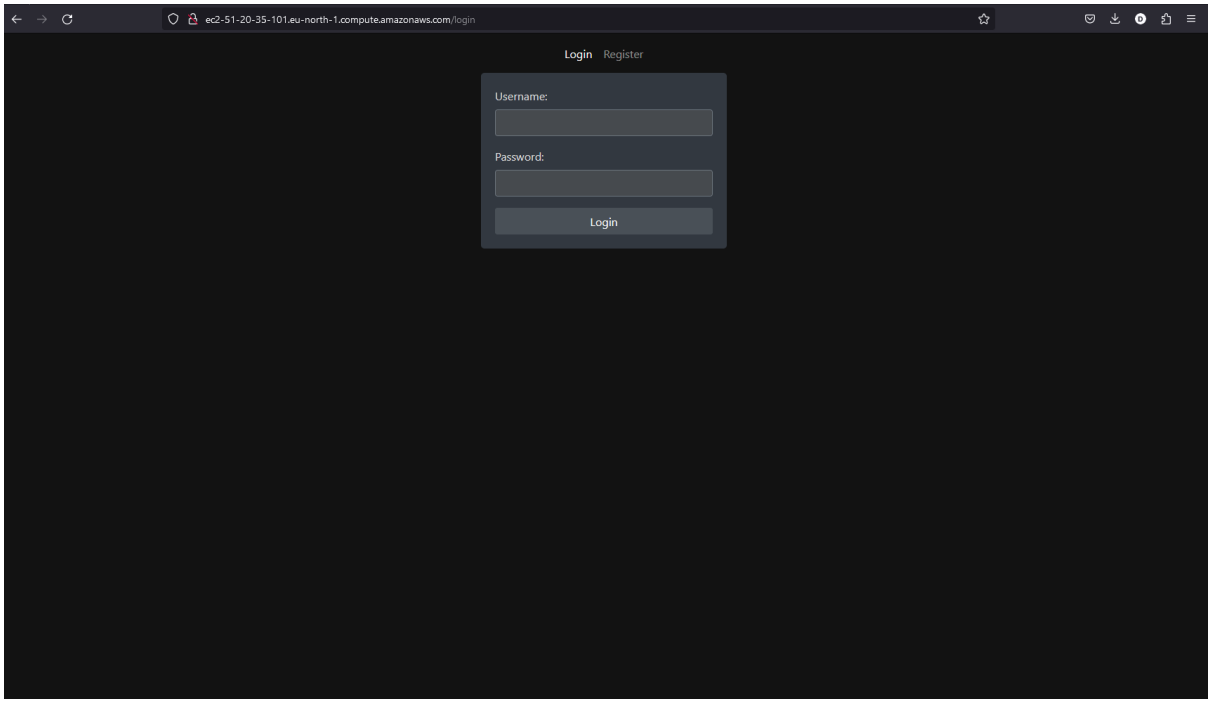
It works very similar to the generate result but instead will query the database instead of something from zap

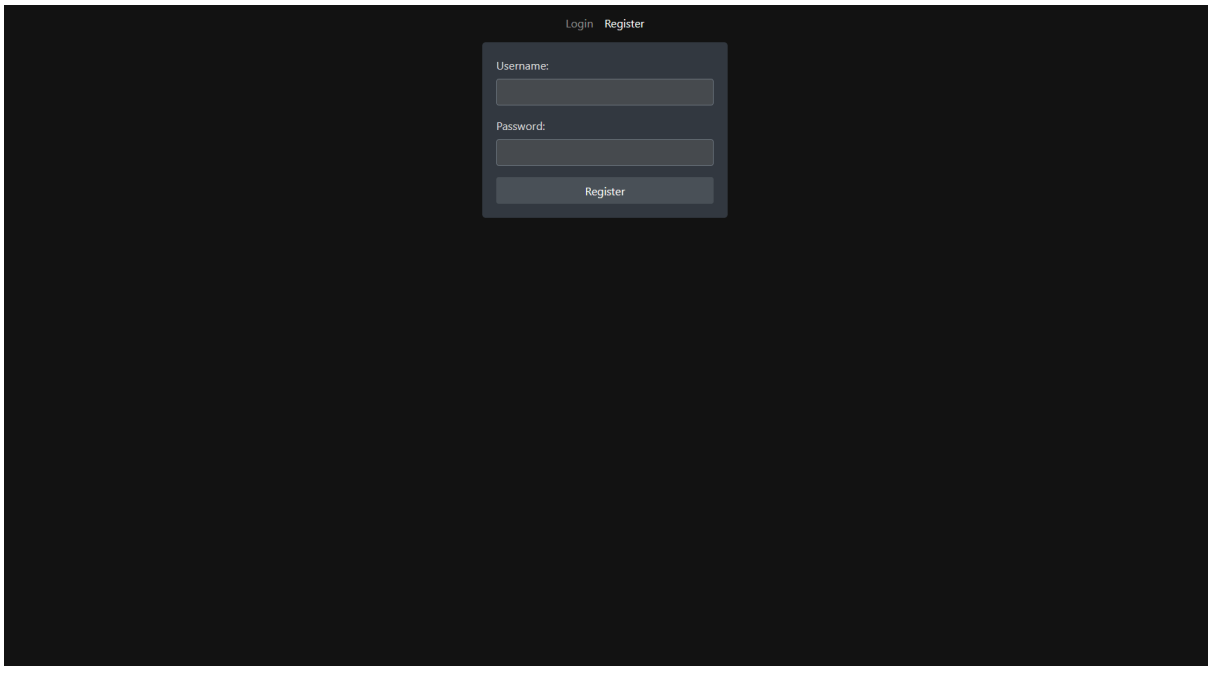
```
def view_scan_result():
    scan_id = request.args.get('scanId')
    if not scan_id:
        return jsonify({"error": "Scan ID is required"}), 400

    scan_result = ScanResult.query.filter_by(scan_id=scan_id,xml_report=ScanResult.xml_report).first()
    if scan_result is None:
        return jsonify({"error": "Scan result not found"}), 404


    return jsonify({"xml_report": scan_result.xml_report or "No report available."})
```

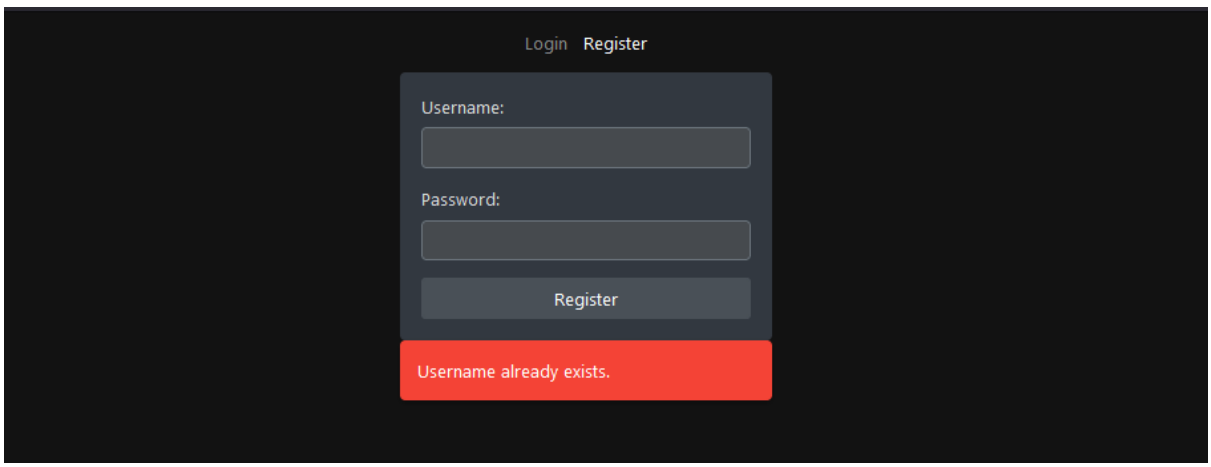
It will check for the scan id then if it finds it it will query the model and pull the scan id and xml report then return the xml report to the frontend as seen in the 1st screenshot.

2.4. Graphical User Interface (GUI)

Login/Register Screen


ec2-51-20-35-101.eu-north-1.compute.amazonaws.com/login
Login
Register
Username:
Password:
Login

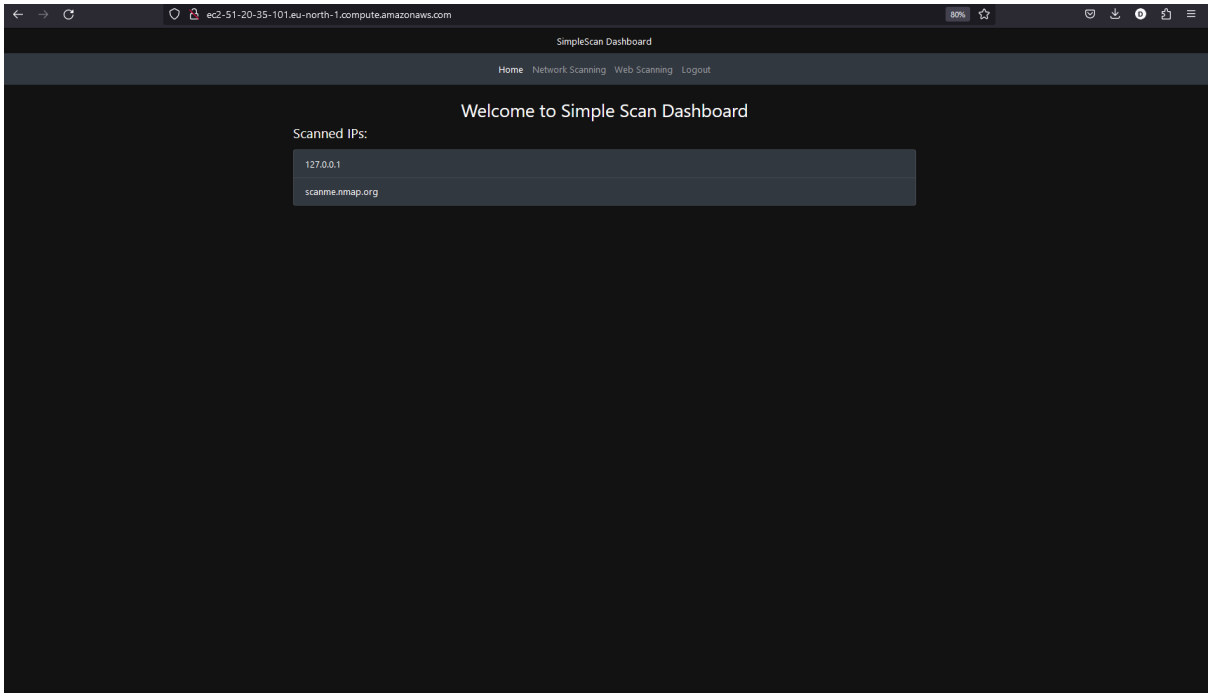

Login
Register
Username:
Password:
Register


Login
Register
Username:
Password:
Register
Username already exists.

Each page contains error display

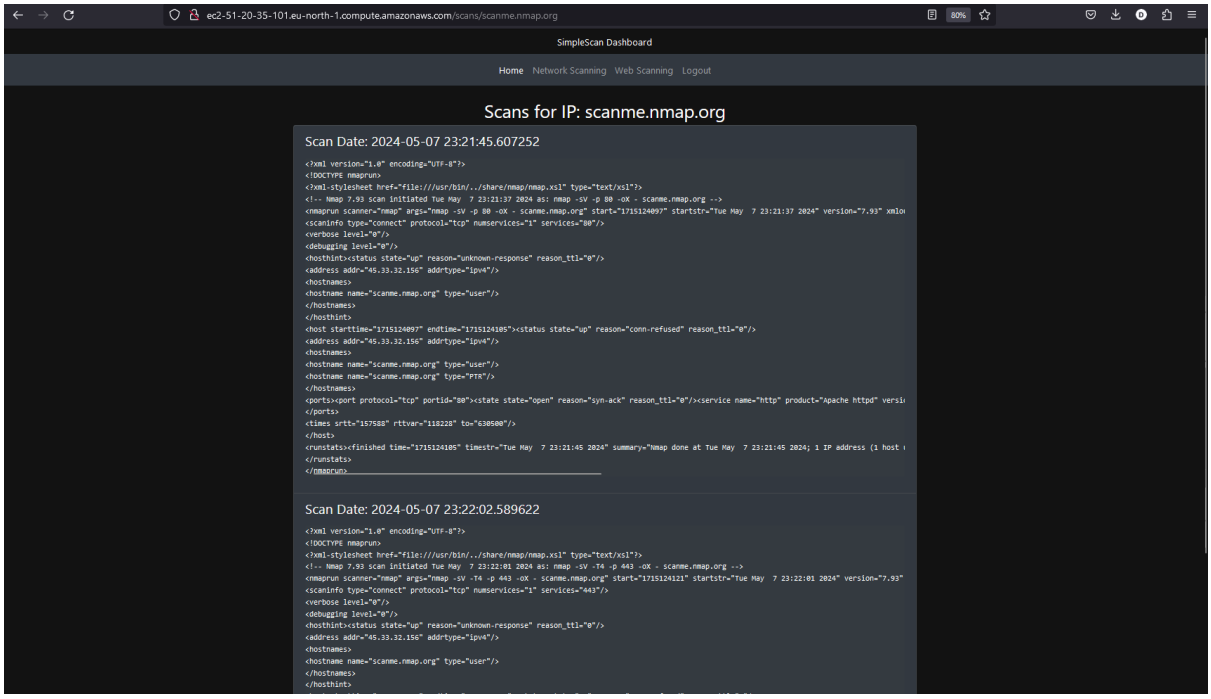
These screens show the forms for logging/registering into the system and are accessible from each through the navbar at the top

Dashboard



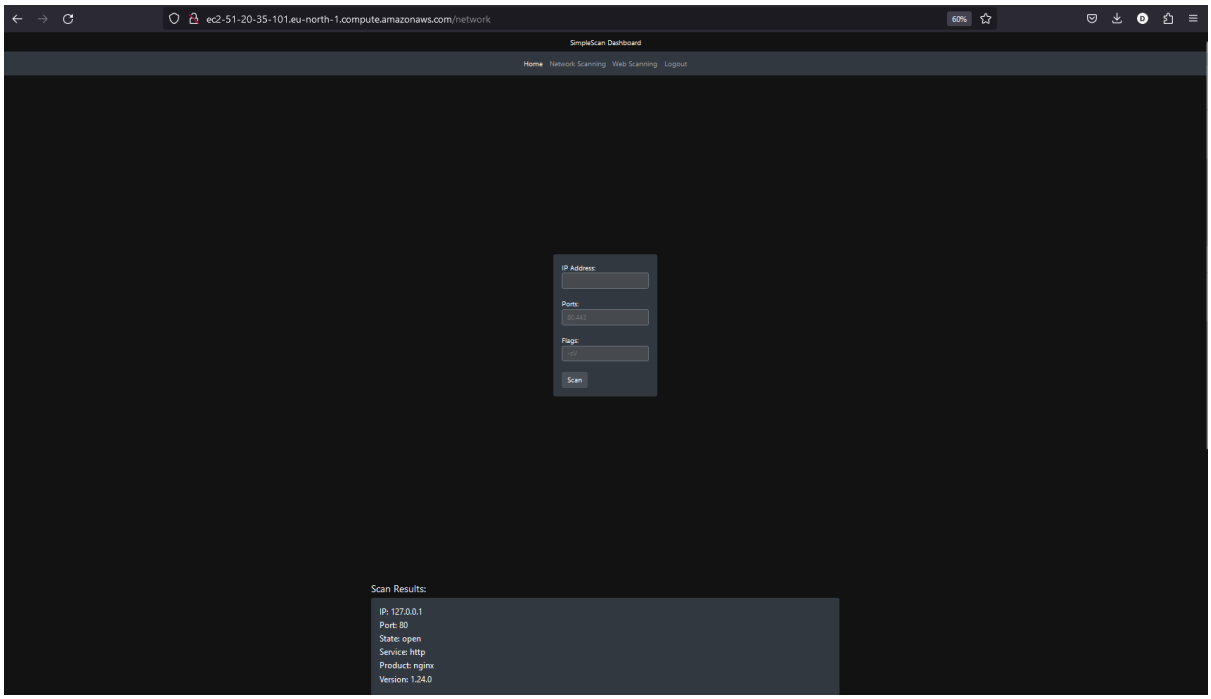
This is the landing page once we log in ,we can see previously scanned results for the currently logged in user,we can click on these and get the results from the previous scan,you can navigate to other pages and even logout if you want to.This navbar is present in all pages from now on

Scan Info



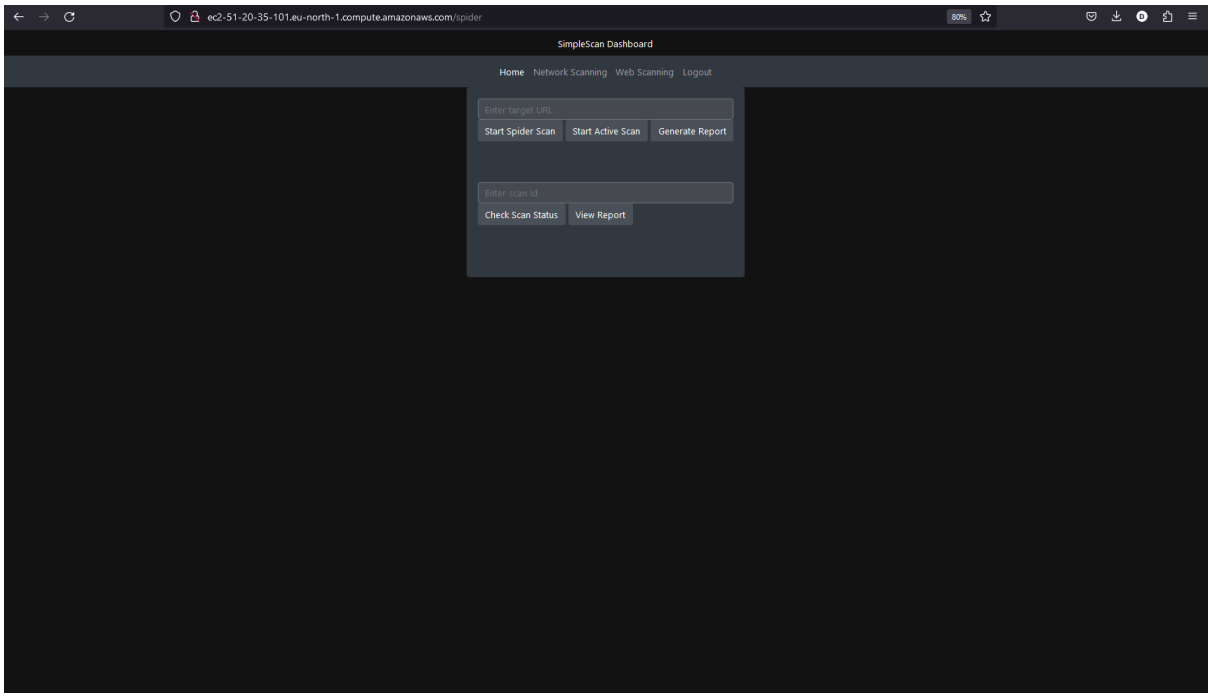
When clicking scanning results it will display them here and you can read into the details of the scan performed

Network Scanning

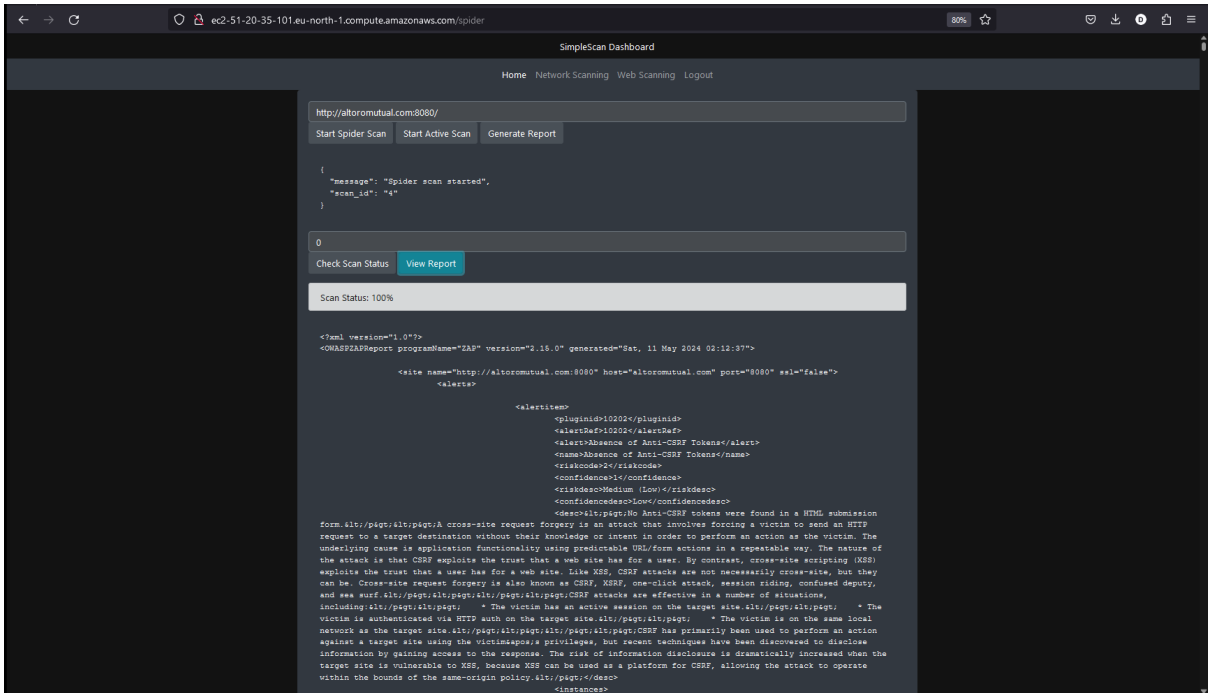


This page is accessible by clicking the network scanning page and will display a from to input a ip,ports and flags once entered and scanned it will display what nmap has found

Web Scanning



Landing page that contains functionality for web scanning with target url you want to scan and then able to check status and view report of scans



Here is all functionality activated and how it looks when done so.

2.5. Testing

For testing I did some manual testing by inserting payloads for xss ,sql injection and did not receive any errors I also scanned my site and it did not give any critical vulnerabilities

2.6. Evaluation

3.0 Conclusions

I think I have made a very good tool it has well designed modular codebase that I can make further modifications to if I want to ,I can maintain it easily and do not have a lot of technical debt if any,it is integrated with powerful tools like nmap and zap which allows me to make network and web scans.It Successfully meets all functional requirements and has good integration of all the tools I wanted to use ,in the future I would like to remove the dependency on these tools but for now it is fine.It has great security features such as session management hashing of passwords and secure systems.I am overall happy with this project and it can stand on its own as a good project.

4.0 Further Development or Research

If I hadfurther time to work on this project I would like to make it a bit more scalable with other tools , iwould also like to remove the need for other tools such as nmap or zap and maybe develop my own tools but these tools are fine for now/the near future.I would then also wish to improve some of the functionality of the website such as associating risks with websites or parsing the xml of web scans to be a bit more readable and then also I would like to test my application some more.

5.0 References

www.zaproxy.org. (n.d.). *API Reference*. [online] Available at: https://www.zaproxy.org/docs/api/.

nmap.org. (n.d.). *XML Output (-oX) | Nmap Network Scanning*. [online] Available at: https://nmap.org/book/output-formats-xml-output.html.

Flask (2010). *Welcome to Flask — Flask Documentation (3.0.x)*. [online] flask.palletsprojects.com. Available at: https://flask.palletsprojects.com/en/3.0.x/.

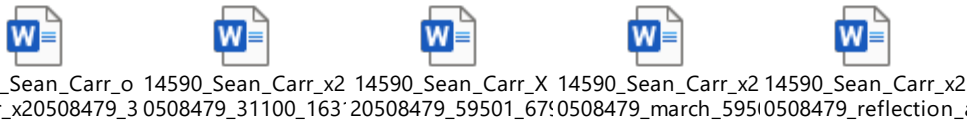
Amazon Web Services, Inc. (n.d.). *Amazon EC2 Resources - Amazon Web Services*. [online] Available at: https://aws.amazon.com/ec2/resources/.

6.0 Appendices

6.1. Project Proposal

National College

6.2. Reflective Journals



6.3. Other materials used

https://github.com/seancarr717/VulnScanner

https://drive.google.com/file/d/1HwFReRAVBz3XRVRQaYiKAfXCAtsokMem/view?usp=sharing