2025

Secure Programming COMP.SEC.300-2024-25-1

EXERCISE WORK

MUHAMMAD HASAN USAMA

Objective:

The objective is to create a secure web interface that protects sensitive data (based on access role) from unauthorized access while addressing common vulnerabilities identified by OWASP and other security standards.

The user interface will simulate a multi-role system with the following roles:

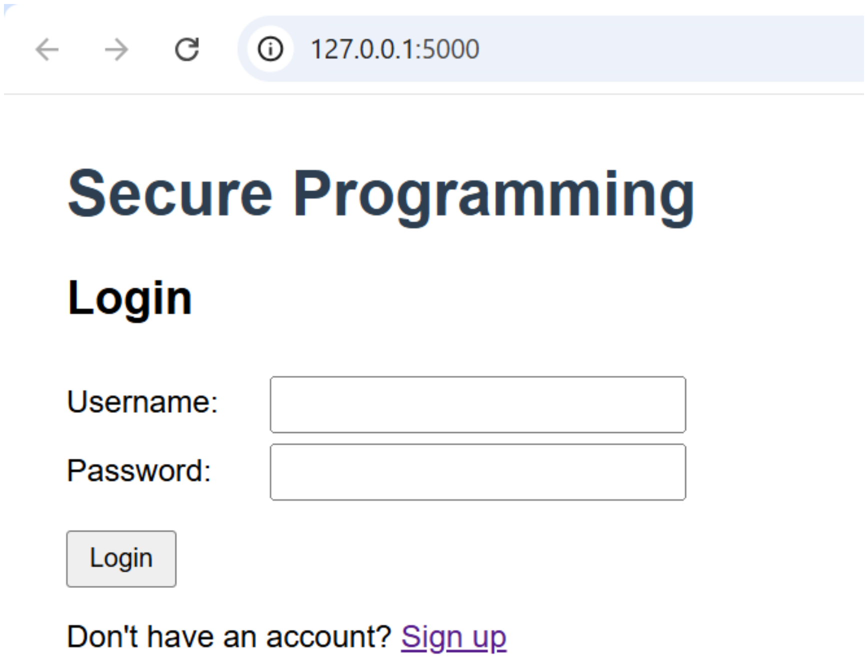
- Admin: Can create users and manage the system.
- Analyst: Can access encrypted datasets and analyze the results.
- Regular User: Can only access a basic user interface with no access to the dataset.

This web interface will be integrated with the previously developed spade crypto system.

User Interface Overview:

I have used a **Flask-based web app**, that can be accessed through a browser. The interface of the web interface includes:

- **Login Screen**: The user can login from here and can go to sign up option as well.


127.0.0.1:5000

Secure Programming

Login

Username:

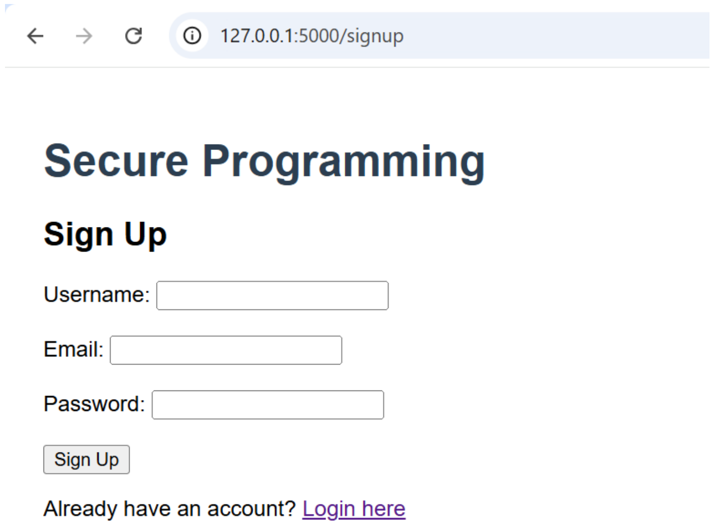
Password:

Login

Don't have an account? Sign up

Figure 1: Login page

- **Sign up page:** The user can create an account from this page, normal account without an access to the data set.


127.0.0.1:5000/signup

Secure Programming

Sign Up

Username:

Email:

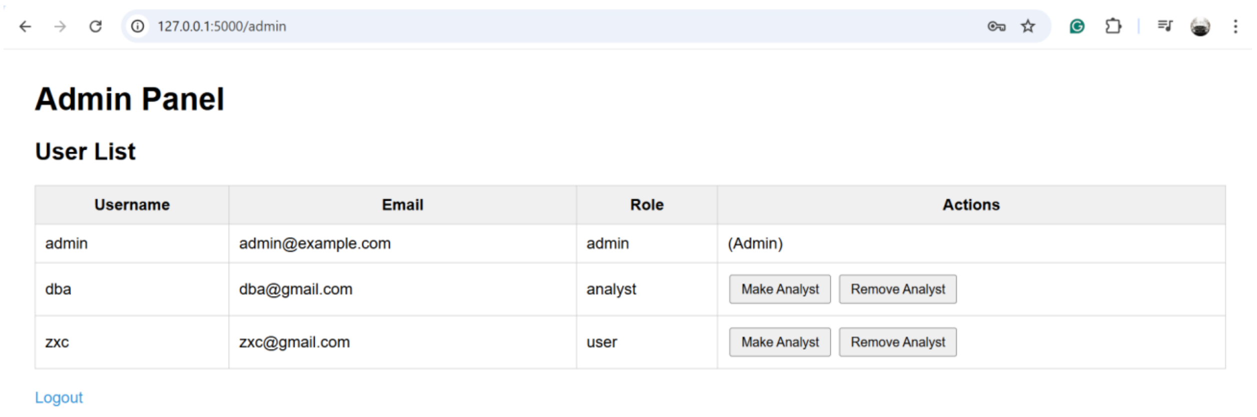
Password:

Sign Up

Already have an account? Login here

Figure 2: Sign up page

- **Admin Dashboard**: If admin user is login it will be shown an admin dashboard where all registered users will be visible along with the assigned roles, which can be modified here.


127.0.0.1:5000/admin

Admin Panel

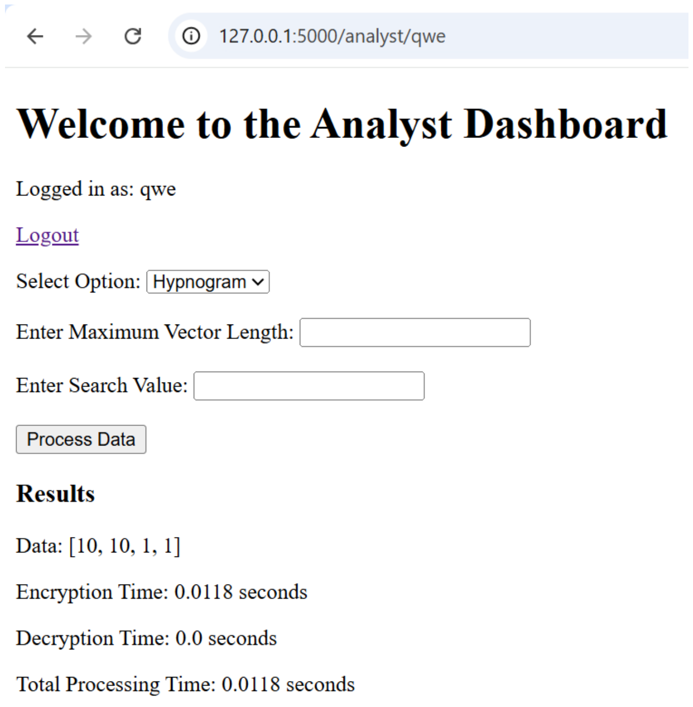
User List

Username	Email	Role	Actions
admin	admin@example.com	admin	(Admin)
dba	dba@gmail.com	analyst	Make Analyst Remove Analyst
zxc	zxc@gmail.com	user	Make Analyst Remove Analyst

Logout

Figure 3: Admin Dashboard

- **Analyst Panel**: The Analyst can access encrypted data for analysis. There are two datasets available, and the Analyst can choose between them. Based on the selected dataset and a chosen vector length, the data will be partially decrypted to perform the analysis. The results, including the identified indexes, will then be displayed. The interface will also show the time taken for encryption and decryption. However, the integration is currently not functioning correctly and is not displaying accurate results.



127.0.0.1:5000/analyst/qwe

Welcome to the Analyst Dashboard

Logged in as: qwe

Logout

Select Option: Hypnogram

Enter Maximum Vector Length:

Enter Search Value:

Process Data

Results

Data: [10, 10, 1, 1]

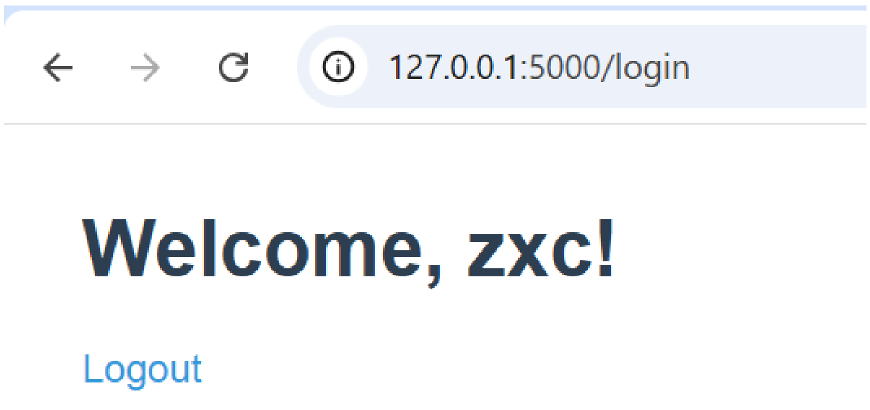
Encryption Time: 0.0118 seconds

Decryption Time: 0.0 seconds

Total Processing Time: 0.0118 seconds

Figure 4: Analyst dashboard

- **Normal user dashboard:** Simple welcome message, no access to the data.



127.0.0.1:5000/login

Welcome, zxc!

Logout

Figure 5: Normal User Dashboard

Structure of the Program

File	Description
admin.py	Script to create admin user in the starting
app.py	All web logic, role checks, encryption triggers, and searches
crypto_operations.py	Custom cryptography logic (SPADE)
templates/*.html	Login and role-based views for Admin, Analyst, and User
dataset/	Contains plaintext DNA and Hypnogram data
dB/	Encrypted databases created during each encryption

Secure Programming Solutions

OWASP Top 10 security checklist has been used to implement features for security:

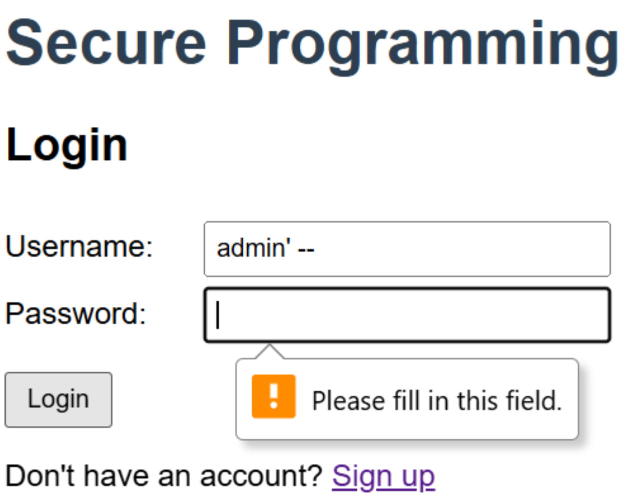
OWASP Risk	How It Was Handled
A01: Broken Access Control	In the Flask web app, route decorators are used to control who can access certain pages. For example, the **/admin** page can only be opened by users with the **Admin** role. This helps make sure that only the right people can see or use certain parts of the website.
A02: Cryptographic Failures	User passwords are hashed before being saved in the database to protect them from being exposed if the system is compromised. The **`werkzeug.security`** library is used for password hashing, which provides secure hash functions and salting mechanisms.
A03: Injection	SQL queries use parameterized statements (e.g., cursor.execute("INSERT INTO ...", (value,))).
A04: Insecure Design	All user inputs (search, signup forms) are validated. For example, during signup, passwords must be at least 6 characters long, and usernames/emails are validated using regex. Invalid or unknown inputs

OWASP Risk	How It Was Handled
	raise exceptions (e.g., `ValueError`), reducing the risk of insecure behavior.
A05: Security Misconfiguration	The Flask SECRET_KEY is set using an environment variable and not hardcoded. Secure cookie attributes (HttpOnly, Secure, SameSite) are properly configured. File and folder paths are created/checked safely before use.
A06: Vulnerable Components	Only well-maintained standard libraries and trusted third-party packages are used (Flask, Werkzeug, SQLite3, psutil). No untrusted or deprecated packages are included.
A07: Auth Failures	All datasets are encrypted before storage or analysis. Keys are never stored. Decryption is partial and tightly scoped, failing silently or with error messages if data is altered or access is unauthorized.
A08: Data Integrity	Encryption and decryption prevent tampering. Decryption fails silently if unauthorized.
A09: Logging/Monitoring	Basic logging to stdout. Can be expanded using Python logging for production.
A10: SSRF/CSRF	No external URLs are fetched by the server, reducing SSRF risk. Forms use secure methods (GET/POST) with server-side validation. CSRF protection is not implemented yet, but can be added using Flask-WTF or similar libraries.

Testing:

1. SQL Injection in Login

   Not moving forward with no password:



Secure Programming

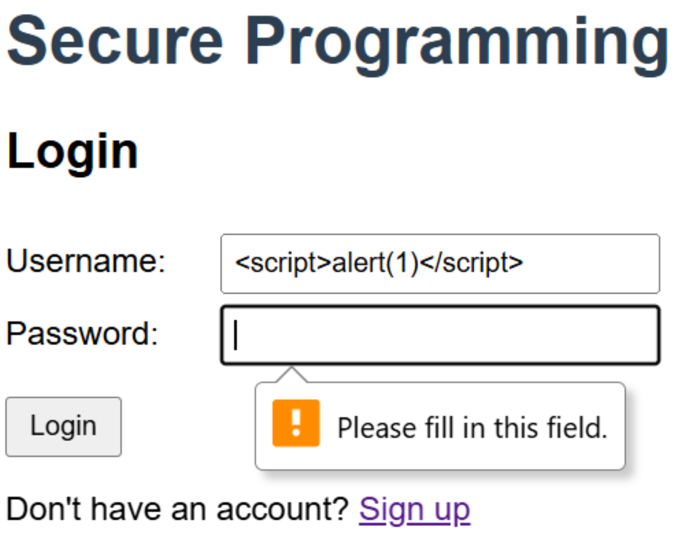
Login

Username: admin' --

Password:

2. XSS in Username

   Not moving forward with XSS script:



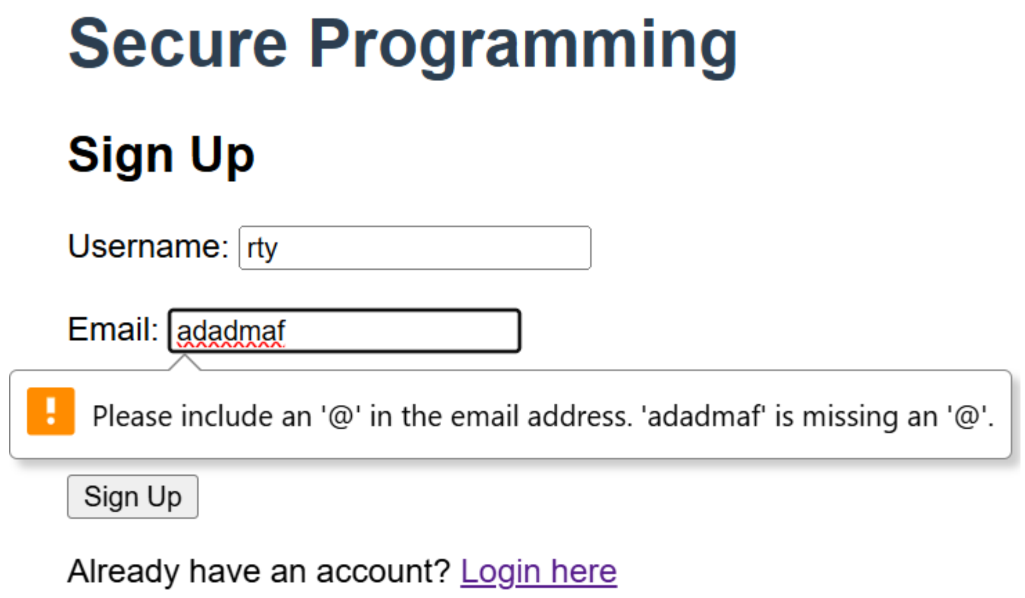
Secure Programming

Login

Username: <script>alert(1)</script>

Password:

3. Invalid Email Format
   Invalid email address will not be stored during sign up:



Secure Programming

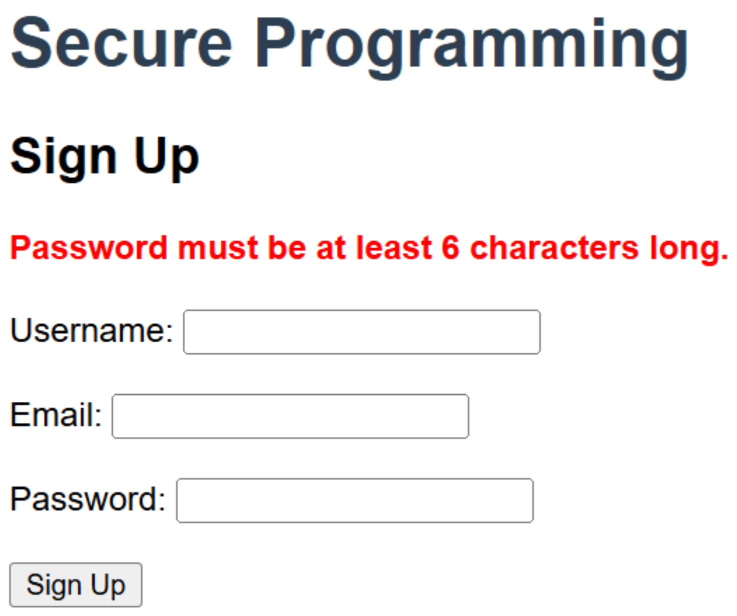
Sign Up

Username: rty

Email: adadmaf

Please include an '@' in the email address. 'adadmaf' is missing an '@'.

4. Weak Password
   Weak password will not be accepted:



Secure Programming

Sign Up

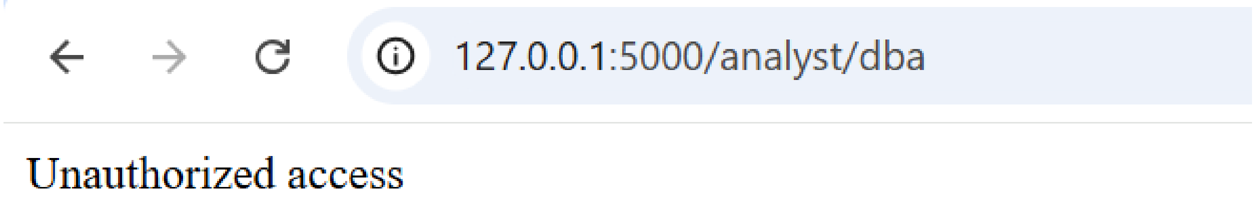
Password must be at least 6 characters long.

Username:

Email:

Password:

5. Access analyst panel as regular user

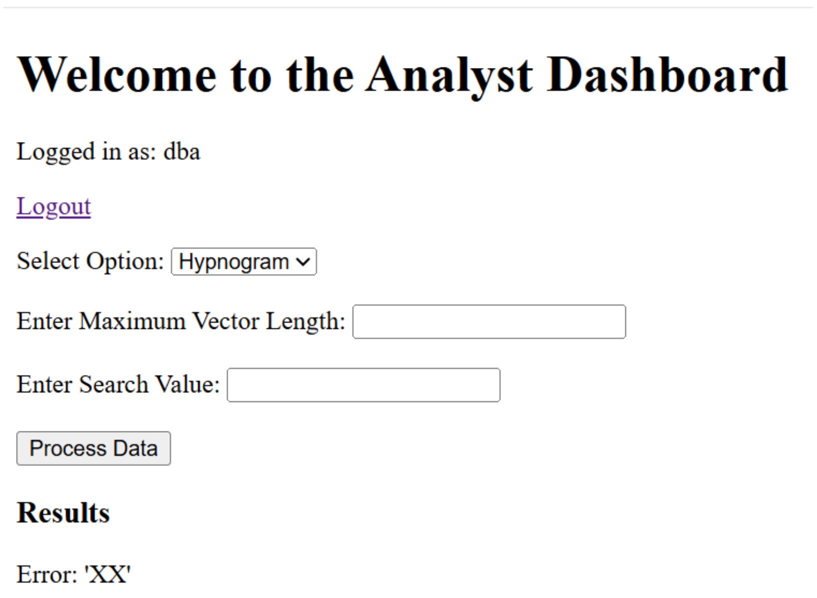


127.0.0.1:5000/analyst/dba

Unauthorized access

6. Invalid input

   Will show error:



Welcome to the Analyst Dashboard

Logged in as: dba

Logout

Select Option: Hypnogram

Enter Maximum Vector Length:

Enter Search Value:

Process Data

Results

Error: 'XX'

Secure Coding Techniques in spade program:

The core concept of the program I have made in previous course is that the dataset is encrypted, and the analyst is provided with a key to partially decrypt the data. This allows the analyst to perform operations—such as finding the indexes of a given value—without accessing the full original dataset. This approach can also be extended to support operations like calculating averages, sums, and other statistical analyses, enabling meaningful insights while preserving data privacy and anonymity.

Changes from Earlier Work:

The SPADE system was originally a command-line cryptographic demo. This project extended it to:

- Add Flask web interface with roles.
- Integrate SPADE cryptosystem with real datasets.
- Implement encryption logging and secure storage.
- Add database integration with encrypted tables.
- Handle user inputs and exceptions.

Suggestions for Improvement

- **CSRF Protection**: Could be added using Flask-WTF for form security.
- **Audit Logging**: User activity logs were not included.
- **JWT Tokens**: Future-proof authentication using token-based access.

Conclusion

This project successfully implements a secure, role-based encryption system for sensitive datasets using Python, Flask, and a custom cryptographic engine (SPADE). It adheres closely to **OWASP guidelines**, validating that secure programming principles can be effectively applied even in custom cryptosystem scenarios.

The project is modular, extensible, and provides a strong foundation for real-world applications that demand secure data storage, access control, and user accountability.

Declaration:

This was my first time creating a web interface. I already knew how to write basic Python code, but I needed help to understand how to build a web app. I used online forums and ChatGPT to learn the correct way to write the code. However, the main idea, logic, and how the program works were all my own. I made sure I understood everything I used and didn’t just copy without knowing what it does. Also, used AI to refine wording in this report and write conclusion.