

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:

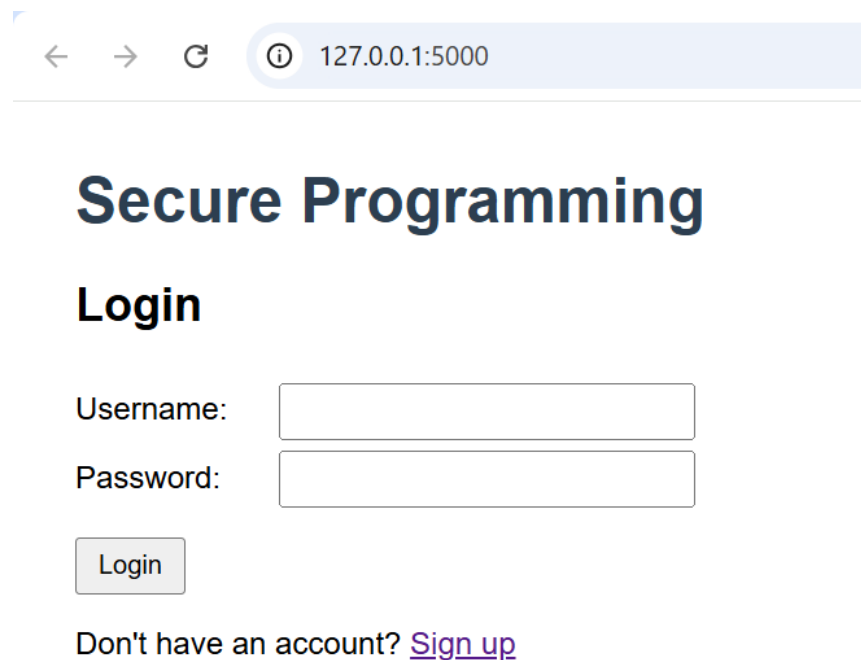
- Admin: Can create users and manage the system.
- Analyst: Can access encrypted datasets and analyze the results.
- Regular User: Can only access a basic user interface with no access to the dataset.

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:

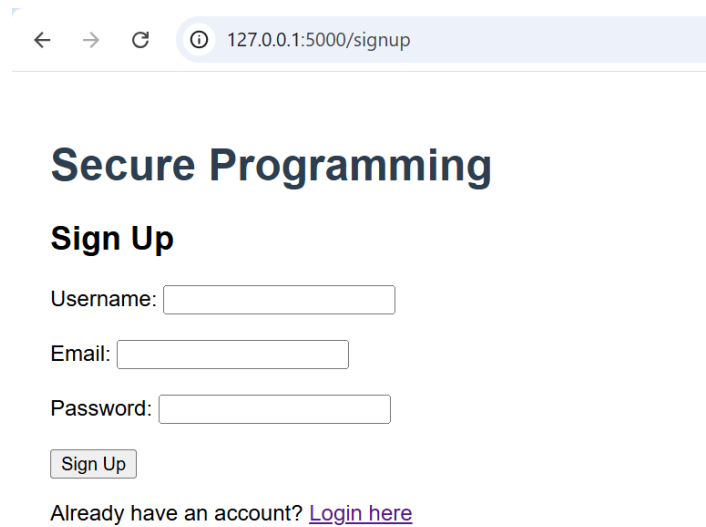
- **Login Screen:** The user can login from here and can go to sign up option as well.



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:5000'. The page title is 'Secure Programming'. Below the title is a 'Login' section. It contains two input fields: 'Username:' and 'Password:'. Below these fields is a 'Login' button. At the bottom of the login section, there is a link that says 'Don't have an account? [Sign up](#)'.

Figure 1: Login page

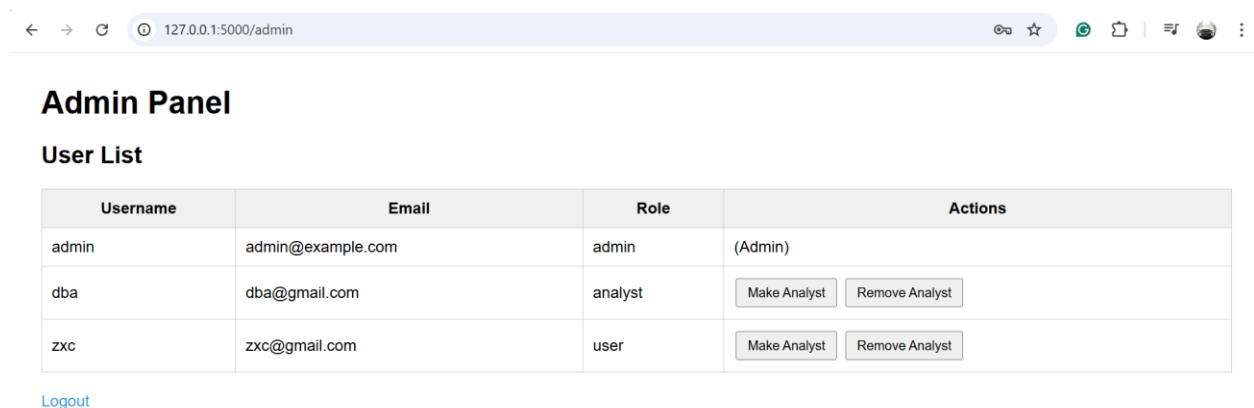
- **Sign up page:** The user can create an account from this page, normal account without an access to the data set.



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:5000/signup". The page title is "Secure Programming" and the section is "Sign Up". It contains three input fields for "Username:", "Email:", and "Password:". Below these fields is a "Sign Up" button. At the bottom, there is a link that says "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.



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:5000/admin". The page title is "Admin Panel" and the section is "User List". It contains a table with the following data:

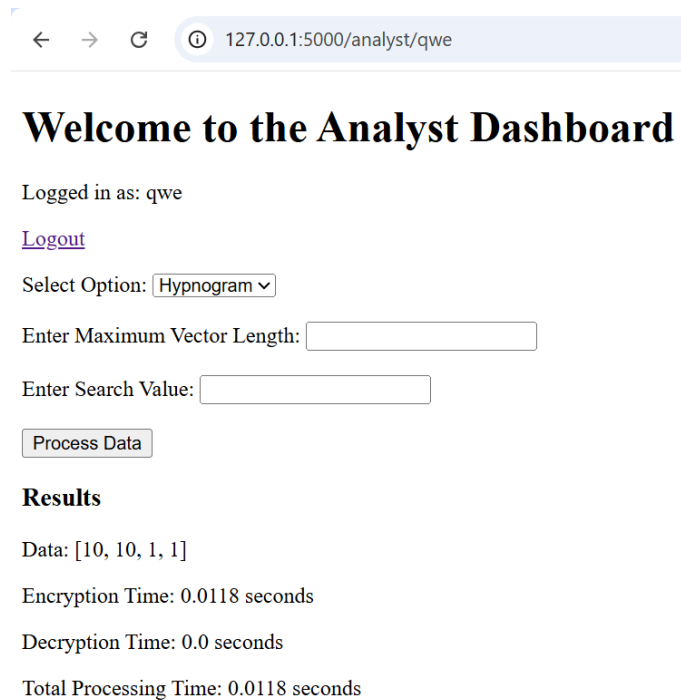
Username	Email	Role	Actions
admin	admin@example.com	admin	(Admin)
dba	dba@gmail.com	analyst	<button>Make Analyst</button> <button>Remove Analyst</button>
zxc	zxc@gmail.com	user	<button>Make Analyst</button> <button>Remove Analyst</button>

Below the table, there is a "Logout" link.

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

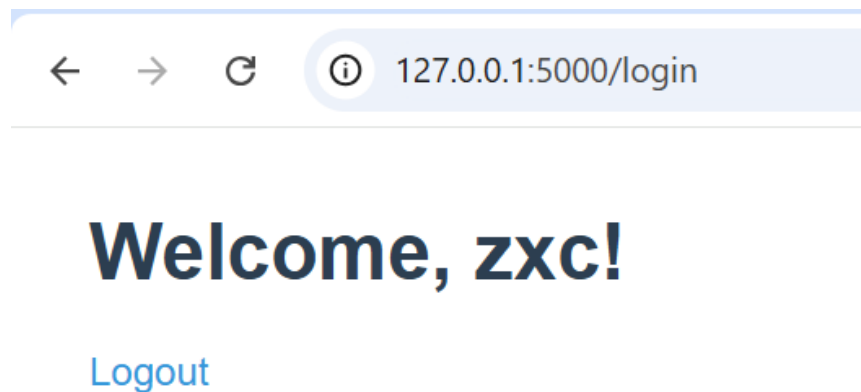
results, including the identified indexes, will then be displayed. The interface will also show the time taken for encryption and decryption. However, the integration is currently not functioning correctly and is not displaying accurate results.



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:5000/analyst/qwe". The main heading is "Welcome to the Analyst Dashboard". Below this, it says "Logged in as: qwe" with a "Logout" link. There is a "Select Option:" dropdown menu currently set to "Hypnogram". Below that are two input fields: "Enter Maximum Vector Length:" and "Enter Search Value:". A "Process Data" button is located below the search field. Under the heading "Results", the following information is displayed: "Data: [10, 10, 1, 1]", "Encryption Time: 0.0118 seconds", "Decryption Time: 0.0 seconds", and "Total Processing Time: 0.0118 seconds".

Figure 4: Analyst dashboard

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



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:5000/login". The main heading is "Welcome, zxc!". Below this is a "Logout" link.

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 <code>werkzeug.security</code> library is used for password hashing, which provides secure hash functions and salting mechanisms.
A03: Injection	SQL queries use parameterized statements (e.g., <code>cursor.execute("INSERT INTO ...", (value,))</code>).
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., <code>ValueError</code>), reducing the risk of insecure behavior.
A05: Security Misconfiguration	The Flask <code>SECRET_KEY</code> is set using an environment variable and not hardcoded. Secure cookie attributes (<code>HttpOnly</code> , <code>Secure</code> , <code>SameSite</code>) 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.

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.