

YENEPOYA INSTITUTE OF ARTS,SCIENCE, COMMERCE AND MANAGEMENT



FINAL PROJECT REPORT

ON

VULNERABILITY ASSESMENT ON A WEB APPLICATION

TEAM MEMBERS:

Name	Register Number	Email Id
Mohammed Sayyan	22BCIECS072	22047@yenepoya.edu.in
Irfan TM	22BCIECS044	21153@yenepoya.edu.in
Chethan Manoj	22BCIECS029	22050@yenepoya.edu.in
Muhammed Shaan K	22BCIECS096	21112@yenepoya.edu.in
Mohammed Niham N.P	22BCIECS068	22088@yenepoya.edu.in

GUIDED BY: Mr. Shashank

TABLE OF CONTENTS

EXECUTIVE SUMMARY	3
Key Project Highlights	3
1.1 Project Aim.....	4
1.2 Technologies Used.....	4
1.3 Frontend Technologies	4
1.4 Backend Technologies	5
2.SYSTEM REQUIREMENTS	6
2.1 Functional Requirements.....	6
2.2 User Requirements	7
2.3 Non-Functional Requirements	7
3.IMPLEMENTATION.....	8
3.1 Project Setup and Structure	8
3.2 User Authentication and Session Management	8
3.3 Database Models and ORM.....	9
3.4 Route Implementations.....	9
3.5 HTML Templates and Jinja2 Rendering	10
3.6 Search and SQL Injection Design	10
3.7 Admin Panel Access Vulnerability	11
3.8 Logging and Monitoring.....	11
3.10 Security Focus in Implementation.....	11
4.TESTING	12
4.1 Test Plan Objectives.....	12
4.2 Data Entry Testing	12
4.3 Security Testing	12
4.4 Test Strategy	15
4.5 System Test Results (Summary)	15
5.SNAPSHOT OF THE PROJECT.....	16
Screenshot 1: XSS Vulnerability Demonstration via Malicious HTML Upload.....	16
Screenshot 2: PHP Remote Code Execution (RCE) Web Shell	17
Screenshot 3: SQL Injection Error via Search Function	18
Screenshot 5: Insecure Admin Panel Access	20
Screenshot 6: Post Creation Page	21
6.CONCLUSION	22
7.REFERENCES	23
8.APPENDIX	24
A. Screenshots of Exploits	24
B. Sample Payloads	24
C. Tools Used	24

EXECUTIVE SUMMARY

This project presents the analysis of a vulnerable Flask-based forum web application designed for educational purposes in the field of cybersecurity and ethical hacking. The application simulates a typical blogging platform, where users can register, log in, post content, and reply to discussions. The primary goal of this project is to identify, exploit, and understand real-world web application vulnerabilities in a controlled environment.

The application contains several critical security flaws that are intentionally left unpatched to demonstrate how poor coding practices can be exploited. The following vulnerabilities were identified and tested:

1. **Cross-Site Scripting (XSS)** – Malicious JavaScript can be injected into post and reply content, leading to client-side attacks such as cookie theft or session hijacking.
2. **SQL Injection (Search)** – The search feature directly incorporates user input into SQL queries, allowing attackers to extract sensitive data like usernames and hashed passwords.
3. **Insecure Direct Object References (IDOR)** – Authenticated users can delete posts they do not own due to the lack of ownership verification.
4. **Lack of Rate Limiting** – No limits on login or registration attempts make the application vulnerable to brute-force attacks and spam registrations.
5. **PHP Remote Code Execution (RCE)** - The application allows unrestricted file uploads through the `/upload` route. Due to the lack of proper validation and web server misconfiguration, attackers can upload PHP files that the server then executes as code. This enables remote code execution, allowing an attacker to run arbitrary system commands..
6. **Lack of Authorization** – Any logged-in user can perform actions without proper role-based access control, leading to privilege misuse (e.g., unrestricted post creation).
7. **Admin Panel Access Control Vulnerability** – The admin panel is protected only by a session variable `user_role`, which can be tampered with or abused due to insecure session handling.

Each vulnerability has been analyzed with proof-of-concept examples, and recommended mitigations are provided. These include input validation, the use of ORM-safe queries, secure session management, ownership checks, and proper role-based access control.

This project serves as a practical tool for understanding web vulnerabilities and highlights the importance of secure coding, especially for developers, testers, and security analysts working with Flask or similar web frameworks.

Core Functionality

The core functionality of the project revolves around a basic forum application built using the Flask framework. Users can register, log in, create new discussion posts, and reply to existing ones. Each post is stored in an SQLite database and displayed dynamically on the homepage. The application also includes a search feature to find posts and a basic admin panel. These features simulate real-world blog/forum behavior, making it ideal for security testing and vulnerability analysis.

Key Project Highlights

- Built a Flask-based forum application with deliberate security flaws for testing.
- Identified and demonstrated 7 critical vulnerabilities including XSS, SQLi, and IDOR.
- Performed practical exploitation using real attack vectors and documented outcomes..

1.BACKGROUND

1.1 Project Aim

The primary aim of this project is to explore and demonstrate the presence of security vulnerabilities in a web application using a forum/blog-like structure developed with Flask. By building a deliberately insecure application, this project helps cybersecurity learners and developers understand how common attacks such as Cross-Site Scripting (XSS), SQL Injection, Insecure Direct Object References (IDOR), and poor access control mechanisms can compromise a system. The goal is to simulate real-world vulnerabilities, perform ethical hacking techniques, and apply defensive coding strategies. This project ultimately serves both as a learning platform and a testing ground for understanding how to build more secure applications.

1.2 Technologies Used

The project is developed using a combination of backend and frontend technologies, forming a complete full-stack web application. It also incorporates basic database and templating functionality. For demonstration and testing purposes, only lightweight and easy-to-set-up technologies were selected, making the application easy to deploy and analyze on local systems. The stack is well-suited for educational environments focused on cybersecurity and ethical hacking.

1.3 Frontend Technologies

1. HTML5 (HyperText Markup Language)

HTML5 is used to define the structure and layout of each page in the application. It provides the skeleton for forms (registration, login, post creation), content display areas (posts, replies), and navigation elements (links and buttons). It ensures semantic structure and accessibility for users.

2. CSS3 (Cascading Style Sheets)

CSS3 is responsible for the styling and visual design of the web application. It enhances user experience by applying colors, fonts, spacing, and layout formatting. Features such as styled buttons, form input fields, and post cards help in making the interface clean and user-friendly.

3. Jinja2 (Flask Templating Engine)

Jinja2 is the templating engine integrated into Flask. It allows Python variables and logic to be injected into HTML templates securely and dynamically. For example, user session data is used to show or hide options like "Create Post" or "Admin Panel" based on login status or user roles.

4. Minimal JavaScript (Optional Client-side Logic)

JavaScript usage in this project is intentionally minimal to keep the focus on backend vulnerabilities. However, small scripts (e.g., alert pop-ups from XSS) may appear in the context of testing attacks. The project does not rely on advanced JavaScript frameworks, keeping the frontend simple and readable.

5. Form-based Interaction Design

All user inputs—such as creating an account, logging in, posting, replying, and searching—are handled through HTML forms, making it easy to demonstrate how frontend elements interact with backend logic.

1.4 Backend Technologies

1. Python (Flask Framework)

Flask is a lightweight Python web framework used to build the application's server-side logic. It defines URL routes, processes form submissions, handles HTTP requests and responses, and manages sessions. Flask's simplicity and modularity make it ideal for building testable and understandable applications.

2. SQLite (Database Engine)

SQLite is used as the database management system. It is file-based, requires no separate server, and integrates easily with Flask. It stores all persistent data such as registered users, forum posts, and replies. Since it is lightweight, it's perfect for running and testing on local machines.

3. SQLAlchemy (Object Relational Mapper)

SQLAlchemy provides an abstraction layer between Python and SQL. It lets developers define database models (`User`, `Post`, `Reply`) as Python classes. However, to intentionally demonstrate SQL Injection, raw SQL queries are used in some places (e.g., in the `/search` route), bypassing the ORM's safety features.

4. Werkzeug (Password Hashing and Session Support)

Werkzeug is a Flask dependency used for secure password hashing (`generate_password_hash`) and verification (`check_password_hash`). This ensures that plain text passwords are never stored in the database, following basic security best practices.

5. Python Logging Module

The built-in logging module is used for tracking key events like login attempts, session activity, and post deletions. Logs help in monitoring the application's behavior and are useful during security analysis and debugging.

6. Session Management (Flask's Secure Cookie-Based Sessions)

Flask stores session data on the client side using secure cookies. While convenient for small apps, this method is vulnerable if the session is not properly protected. The project highlights how insecure handling (like trusting the `user_role` from session directly) can be abused in access control bypass.

2.SYSTEM REQUIREMENTS

This section describes the functional, user, and environmental requirements necessary for the successful development, deployment, and testing of the Flask-based vulnerable forum application. These requirements ensure the application behaves as expected and operates efficiently under normal usage conditions.

2.1 Functional Requirements

Functional requirements describe the specific behavior and functions the application must support to be considered complete and functional.

- 1. User Registration**

The system must provide a registration form that allows new users to create an account using a unique username and password.

- 2. User Login and Session Management**

Registered users must be able to log in with valid credentials. Upon successful authentication, a session should be created and maintained until the user logs out.

- 3. Post Creation**

Authenticated users must be able to create forum posts by submitting a title and content. These posts are saved to the database and displayed on the home page.

- 4. Reply to Posts**

Users should be able to add replies to existing posts. These replies must be linked to both the post and the replying user.

- 5. Search Functionality**

The system must allow users to search for posts or user data using a search box. The results are to be dynamically fetched from the database.

- 6. Post Deletion**

Users must have the ability to delete their own posts. Admins should be allowed to delete any post.

- 7. Admin Panel Access**

Admin users must have access to a special dashboard that displays all registered users along with their roles.

- 8. Error Handling**

The system should handle invalid inputs, failed logins, and missing pages gracefully, showing appropriate error messages to the user

2.2 User Requirements

User requirements focus on what the end-user expects the system to do, along with usability aspects of the application.

1. **Ease of Use**

The interface should be simple and intuitive, even for users with minimal technical knowledge. Features such as navigation buttons, form inputs, and error messages must be clear and understandable.

2. **Accessibility**

The system should be accessible from any modern web browser without requiring additional plugins or extensions.

3. **Responsiveness**

The web application must display properly on various screen sizes such as desktops, laptops, and tablets.

4. **Feedback and Alerts**

Users must receive visible alerts or messages for key actions like registration success, login failure, post creation, etc.

5. **Security Awareness**

Since the project demonstrates vulnerabilities, users participating in testing should be aware that data and actions are simulated for educational purposes.

2.3 Non-Functional Requirements

Non-functional requirements define the quality attributes and operational constraints of the system, focusing on how the system performs rather than what it does.

1. **Performance**

The application should load forum posts and render pages within 2 seconds under normal conditions. The system should be responsive even with multiple users accessing posts and replies simultaneously.

2. **Usability**

The user interface must be simple, intuitive, and user-friendly. Basic operations such as registration, login, post creation, and search should be accessible without requiring prior technical knowledge.

3. **Maintainability**

The application code should be modular and well-commented, enabling future developers to update or extend functionalities with minimal effort. The use of Flask and SQLAlchemy helps separate concerns cleanly.

4. **Portability**

The system should run on major operating systems (Windows, Linux, macOS) with minimal setup. All dependencies are Python-based and can be installed via a `requirements.txt` file.

5. **Scalability (Limited)**

While the application is intended for local and small-scale use, it should be capable of handling dozens of users and hundreds of posts without crashing or slowing down significantly.

Security (Deliberately Weakened)

The application is intentionally built with security vulnerabilities for educational purposes. However,

6. this section highlights the importance of normally applying proper security practices such as input validation, secure sessions, and access control.

3.IMPLEMENTATION

The implementation phase is the most critical aspect of this project, as it not only involves constructing the actual functionalities of the web application but also deliberately integrating common security flaws for educational analysis. The goal was to simulate real-world programming errors and insecure design patterns to highlight how easily an attacker can exploit them if proper measures are not taken.

3.1 Project Setup and Structure

The project was implemented using **Python 3.x** along with the **Flask web framework**, which offers lightweight and modular design for building web applications. The structure of the application follows a typical Flask project format, including:

- **app.py** – The main application file containing routes, logic, database models, and Flask configuration.
- **/templates directory** – Contains HTML templates rendered using Jinja2 for all front-end views.
- **/static directory** – Contains static assets like `styles.css` for frontend design.
- **forum.db** – SQLite database file used to persist user data, posts, and replies.

The project was designed to be easily executable on a local system without requiring internet access or third-party APIs.

3.2 User Authentication and Session Management

- **Registration:**
New users can register by providing a unique username and password. Passwords are hashed using Werkzeug's `generate_password_hash()` function before storing them in the database, to prevent direct exposure of plaintext passwords.
- **Login:**
Users log in using their credentials. The system verifies them using `check_password_hash()`. On successful login, a session is created and stored in Flask's signed cookie. The session contains the `user_id` and `user_role`.
- **Sessions:**
Flask's built-in `session` dictionary is used to manage login state. A hardcoded `SECRET_KEY` ("VERY_SECRET_KEY_FOR_TESTING") signs the session. While sufficient for demo use, this design is insecure for production environments.
- **Logout:**
The logout function clears the user session and redirects to the homepage.

3.3 Database Models and ORM

Using **SQLAlchemy**, three core models are defined in `app.py`:

1. **User**: Contains fields for ID, username, hashed password, and role (user or admin).
2. **Post**: Stores user-generated forum posts with title, content, and foreign key linking to the author.
3. **Reply**: Contains reply content linked to both a post and a user.

A function `init_db()` automatically creates these tables and seeds them with test users and sample posts (some intentionally malicious for testing XSS and SQL Injection).

3.4 Route Implementations

The application includes multiple Flask routes that represent different functionalities:

1. Home Page (`/`)

- Lists all forum posts.
- Displays links to login, register, create post, and admin panel based on session data.
- Includes a JavaScript alert as a welcome message (optional).

2. Registration Page (`/register`)

- Accepts user input for username and password.
- Performs uniqueness checks on username.
- On success, redirects to login.

3. Login Page (`/login`)

- Validates username and password using secure hash check.
- Sets session variables on success (`user_id`, `user_role`).
- Logs session activity using `logging.debug`.

4. Create Post (`/post`)

- Authenticated users can submit a new post with a title and content.
- Post is linked to their user ID and saved in the database.

5. View Post (`/post/<post_id>`)

- Displays full content of the post.
- Shows replies associated with the post.
- Allows users to reply to the post.

6. Reply to Post (`/reply/<post_id>`)

- Saves the user's reply content to the database.
- Vulnerable to XSS if JavaScript is entered in reply form

7. Search (/search)

- Allows users to enter a keyword.
- Uses raw SQL with user input directly injected (vulnerable to SQL Injection).
- Example attack: ' OR 1=1 -- returns all posts and user credentials.

8. Delete Post (/delete_post/<post_id>)

- Allows users to delete posts, but lacks ownership checks (IDOR).
- Any logged-in user can delete any post by modifying the `post_id` in the URL.

9. Admin Panel (/admin_panel)

- Displays all users and their roles.
- Access control is based solely on `session['user_role'] == 'user'`, allowing unauthorized access through session manipulation.

3.5 HTML Templates and Jinja2 Rendering

Each route renders an HTML template using Jinja2. The templates use dynamic placeholders like `{{ post.title }}` to insert content.

However, in `view_post.html`, the following line introduces a **Cross-Site Scripting (XSS)** vulnerability:

```
html
CopyEdit
<p>{{ reply.content | safe }}</p>
```

Using `| safe` prevents Jinja from escaping HTML characters, allowing JavaScript payloads to be executed when rendered.

3.6 Search and SQL Injection Design

The `/search` route executes this vulnerable raw SQL query:

```
python
CopyEdit
text(f"SELECT ... WHERE title LIKE '%{query}%'")
```

This makes it possible to inject raw SQL such as:

```
sql
CopyEdit
' UNION SELECT id, username, password, NULL, 'user' FROM user -
```

The response includes sensitive data (like usernames and password hashes), clearly demonstrating an SQL Injection attack.

3.7 Admin Panel Access Vulnerability

The admin panel is meant for administrative users, but access control is flawed:

```
python
CopyEdit
if session.get('user_role') == 'user':
```

This logic ironically grants access if the role is "user", not "admin". This is a deliberate flaw that highlights poor session-based access control.

3.8 Logging and Monitoring

The logging module logs critical events like:

- Login attempts (success and failure)
- Post deletions
- Session values at different points

This helps during vulnerability testing and supports the evaluation of attack traces.

3.9 User Interface and Experience

- A minimal but clean UI is implemented using HTML and CSS.
- Forms are used for all input activities (login, registration, posting).
- CSS styles provide visual consistency across all pages.
- No frontend validation is applied to simulate a real-world scenario where backend validation is critical.

3.10 Security Focus in Implementation

Each part of the app is intentionally implemented in an insecure way to demonstrate:

- Lack of input validation
 - Use of unsafe Jinja filters (`| safe`)
 - Absence of authorization checks
 - Use of raw SQL queries
 - Poor session-based access control
- These decisions enable students and researchers to replicate, exploit, and understand common web vulnerabilities in a safe environment.

4. TESTING

The testing phase plays a pivotal role in ensuring that the application behaves as intended while also exposing its security flaws. For this project, testing was approached from two perspectives:

- **Functional Testing:** To verify the application performs basic user operations correctly.
- **Security Testing:** To identify and exploit known vulnerabilities such as XSS, SQL Injection, IDOR, and more.

Both manual and semi-automated testing methods were employed using developer tools, crafted payloads, and browser-based observations.

4.1 Test Plan Objectives

The objectives of the test plan were as follows:

1. To validate that core functionalities like registration, login, posting, replying, searching, and deleting are fully operational.
2. To test the robustness of the application against malicious inputs and identify how it handles unexpected or crafted data.
3. To exploit known vulnerabilities intentionally introduced into the application to study their behavior.
4. To evaluate the application's resistance to brute force and session tampering attempts.
5. To document all observed flaws and recommend appropriate mitigation strategies for future enhancement.

4.2 Data Entry Testing

This test focused on the input handling mechanisms of the application.

- **Test Cases:**
 - Entering valid usernames and passwords during registration.
 - Inputting very long strings (e.g., 10,000 characters) into the post content field.
 - Entering malformed data such as special characters, HTML tags, and SQL snippets.
- **Observations:**
 - The application accepted all inputs without restriction.
 - There was no length limit on text fields.
 - HTML and script tags were rendered directly in the browser, which enabled XSS.
 - No server-side validation was implemented.

4.3 Security Testing

Security testing was conducted by simulating various attack scenarios against the application to evaluate its vulnerability level. The following are the primary security tests carried out:

A. Cross-Site Scripting (XSS)

- **Objective:** To check whether the application escapes or filters malicious script input.
- **Payload Used:**

```
html
CopyEdit
<script>alert('XSS');</script>
```
- **Test Area:** Post content and reply fields.
- **Execution:**
 - A user created a post with the above payload as the content.
 - Upon visiting the post from another account, the browser executed the script.
- **Result:** The JavaScript alert popped up, confirming a stored XSS vulnerability.
- **Root Cause:** The `| safe` Jinja2 filter used in templates renders HTML without escaping it.
- **Risk:** Attackers can hijack sessions, deface the UI, or run malicious scripts in victim browsers.

B. SQL Injection (Search Function)

- **Objective:** To test for direct SQL injection vulnerabilities.
- **Payload Used:**

```
sql
CopyEdit
' UNION SELECT id, username, password, NULL, 'user' FROM user --
```
- **Test Area:** Search input box (`/search` route).
- **Execution:**
 - Entered the payload into the search bar.
 - The page rendered database records including usernames and hashed passwords.
- **Result:** SQL Injection confirmed. Sensitive data was exposed.
- **Root Cause:** The search query used raw SQL with string formatting:

```
python
CopyEdit
text(f"SELECT ... WHERE title LIKE '%{query}%'")
```

- **Risk:** Attackers can extract, modify, or delete data, and possibly escalate access.

C. Insecure Direct Object Reference (IDOR)

- **Objective:** To check if users can manipulate URLs to access or modify unauthorized resources.
- **Test Area:** Post deletion (`/delete_post/<post_id>`).
- **Execution:**
 - Logged in as a normal user.
 - Accessed the delete URL of a post created by another user.
 - The post was successfully deleted without any authorization check

- **Result:** IDOR confirmed.
- **Root Cause:** No ownership verification before deleting a post.
- **Risk:** Any authenticated user can delete any post, leading to data loss and misuse.

D. Lack of Rate Limiting

- **Objective:** To test if the application restricts repeated login attempts or rapid registrations.
- **Test Area:** /login and /register.
- **Execution:**
 - Repeated login attempts were made with different password guesses.
 - Multiple fake user accounts were registered in quick succession.
- **Result:** No rate limiting or CAPTCHA was present.
- **Root Cause:** No throttle mechanism using tools like Flask-Limiter or request counters.
- **Risk:** The application is vulnerable to brute-force attacks and mass account creation.

E. Lack of Input Validation

- **Objective:** To verify whether the system validates input formats and lengths.
- **Test Area:** All user inputs (username, password, title, content).
- **Execution:**
 - Entered extremely long strings, SQL queries, HTML tags, and scripts.
 - None were blocked or sanitized.
- **Result:** Inputs were directly processed and rendered.
- **Risk:** This creates a gateway for XSS, SQL Injection, DoS (via oversized inputs), and malformed data storage.

F. Lack of Authorization

- **Objective:** To determine whether actions are restricted based on user roles.
- **Test Area:** /admin_panel, /post, and delete operations.
- **Execution:**
 - Normal users could access the admin panel due to flawed session role logic.
 - Users could perform unrestricted post creations.
- **Result:** No role-based authorization enforced.
- **Risk:** Privilege escalation, spamming, and access to sensitive admin features.

G. Admin Panel Access Control Vulnerability

- **Objective:** To test if admin-only routes are truly protected.
- **Execution:**

- Manually set `session['user_role'] = 'admin'` using browser dev tools or code modification.
 - Accessed `/admin_panel`.
- **Result:** Admin panel opened successfully, even for non-admin users.
- **Root Cause:** Insecure and misconfigured role checking.
- **Risk:** Unauthorized access to user data, leading to a full compromise.

4.4 Test Strategy

A combination of **manual penetration testing**, **developer console observation**, and **code review** was used.

The following strategy was adopted:

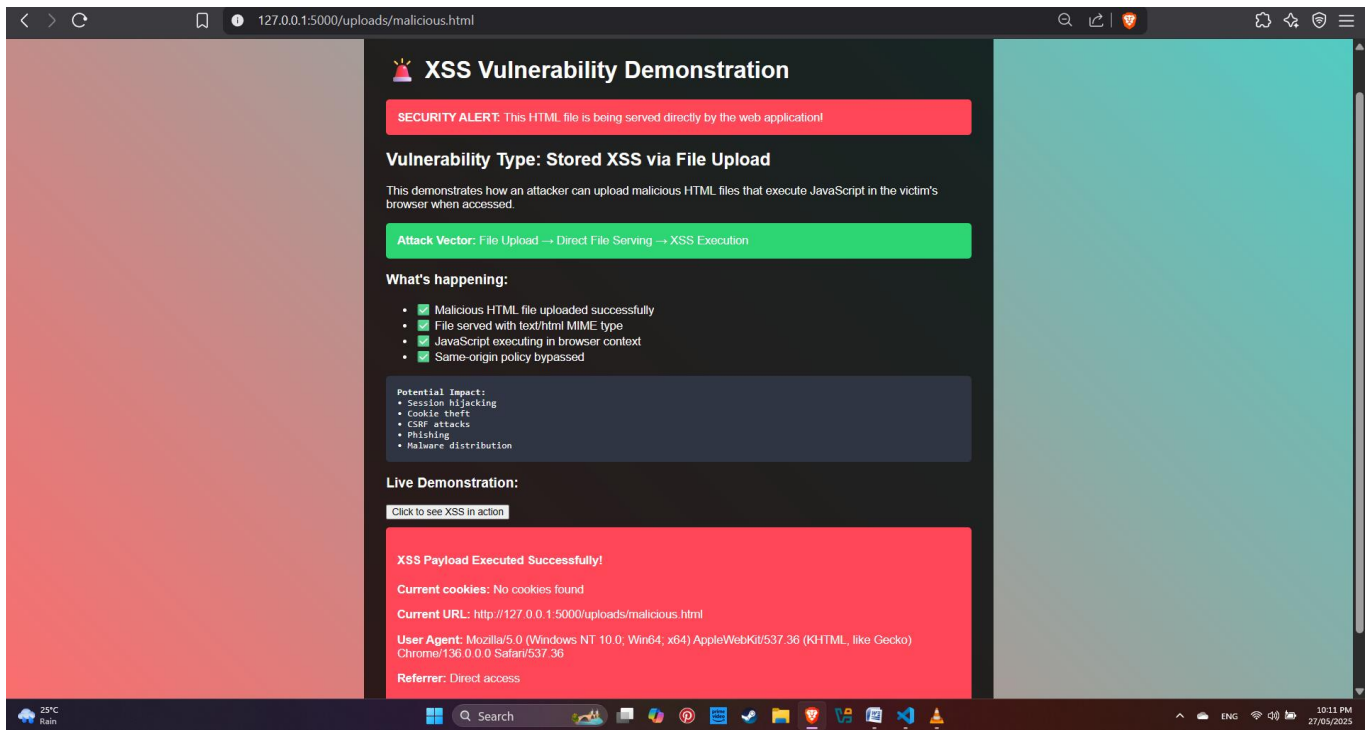
- **White-box Testing:** As source code was available, test cases were derived from analyzing logic flaws.
- **Exploratory Testing:** Testers tried unpredictable inputs and behaviors without predefined scripts.
- **Payload Injection:** Security payloads were inserted into forms, URLs, and session variables.

4.5 System Test Results (Summary)

Test Area	Test Type	Result	Vulnerability Found
Registration	Functional	Pass	No
Login	Functional	Pass	No
Post Creation	Functional	Pass	XSS Possible
Search	Functional	Pass	SQL Injection
Delete Post	Functional	Pass	IDOR
Admin Panel	Functional	Pass	Access Control Flaw
Input Forms	Boundary Testing	Pass	No Validation
Login Attempts	Brute Force Test	No Block	No Rate Limiting

5.SNAPSHOT OF PROJECT

Screenshot 1: XSS Vulnerability Demonstration via Malicious HTML Upload

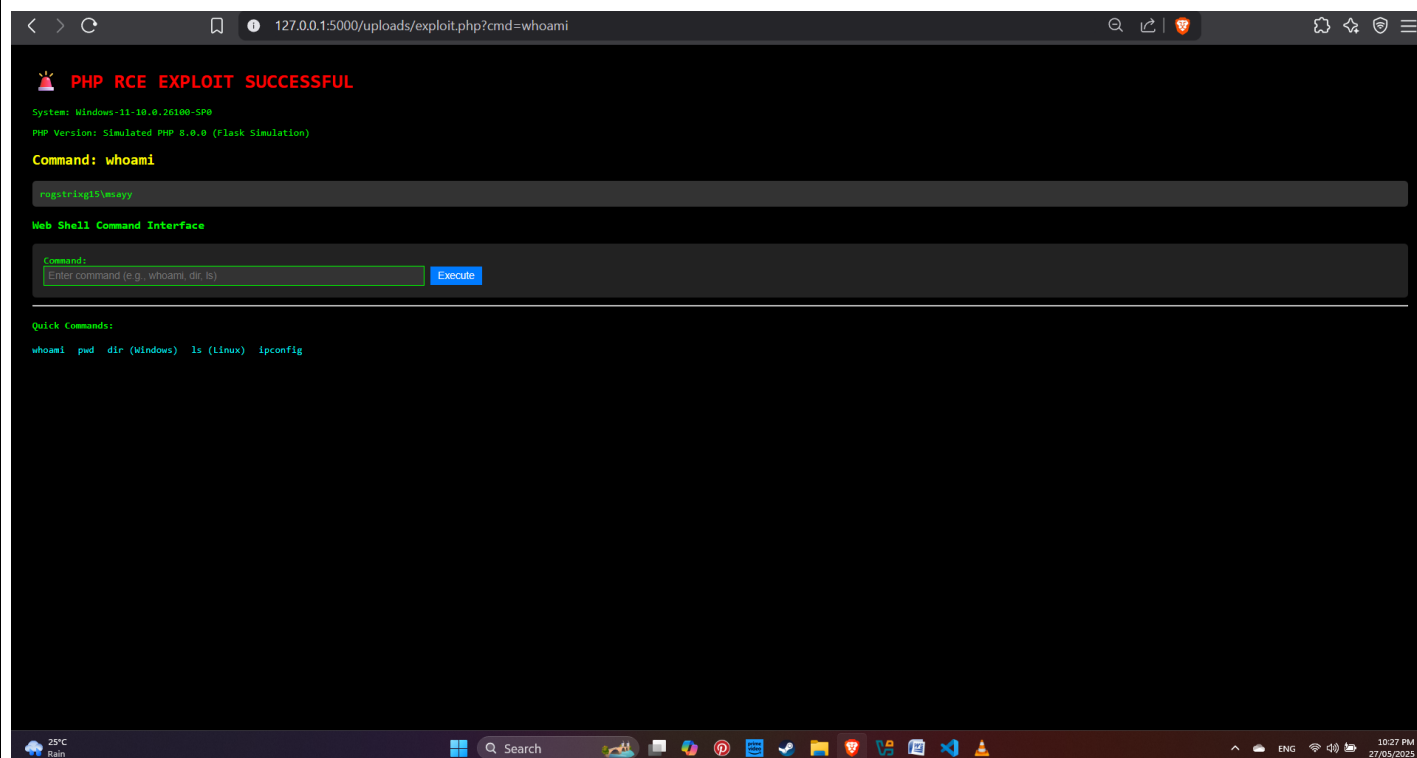


File: malicious.html

Description:

This screenshot demonstrates a **stored Cross-Site Scripting (XSS)** attack where a malicious HTML file is uploaded and served directly from the `/uploads` directory. The file is rendered with `text/html` MIME type, allowing embedded JavaScript to execute in the victim's browser. The demo includes session hijacking, cookie theft, and phishing potential. This confirms a high-risk vulnerability due to lack of input validation and secure upload handling.

Screenshot 2: PHP Remote Code Execution (RCE) Web Shell

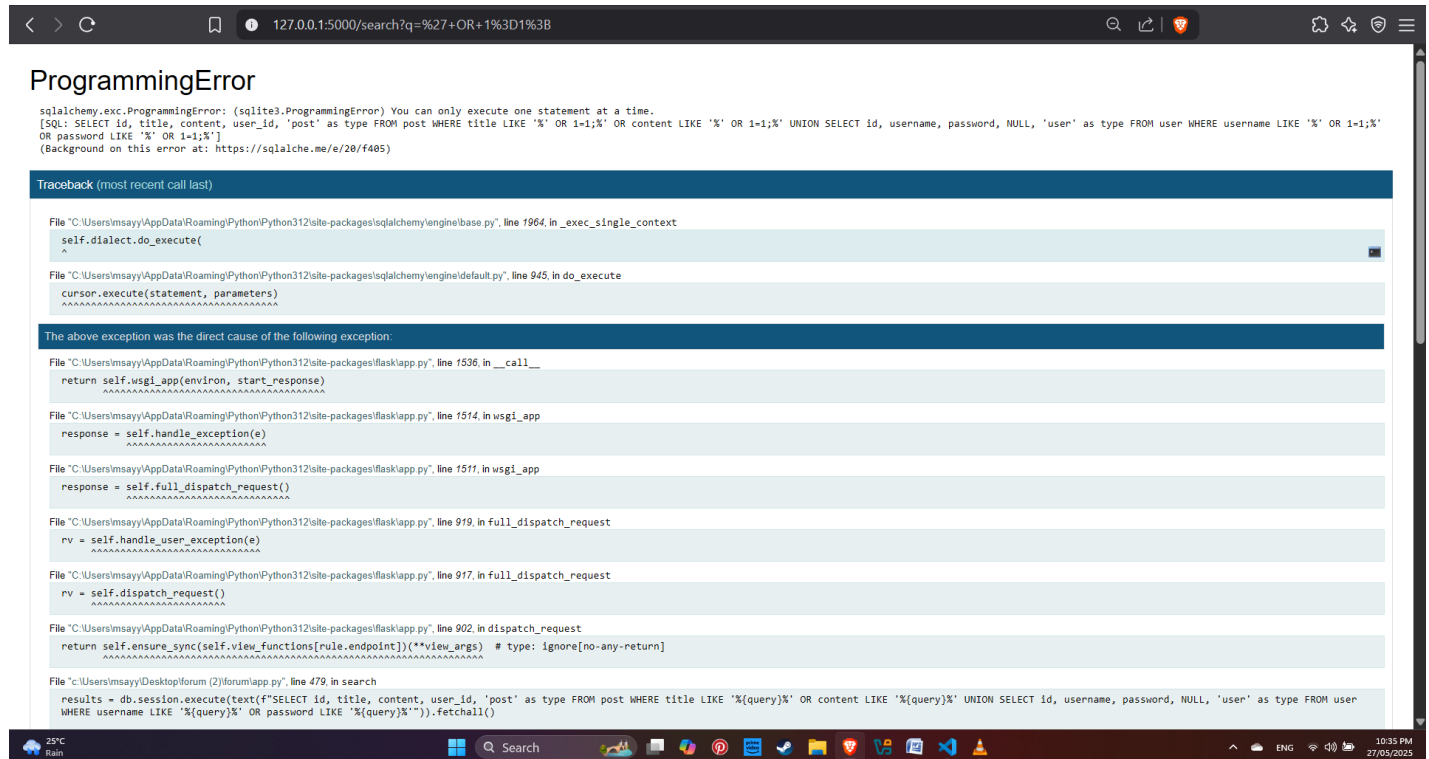


File: exploit.php

Description:

This shows successful **Remote Code Execution** by uploading a PHP web shell through the /upload route. The command `whoami` is executed remotely via the browser using `?cmd=whoami`, returning the system username. The web interface allows executing arbitrary system commands, proving complete server compromise. This highlights a critical vulnerability due to unrestricted file uploads and PHP execution in the uploads directory.

Screenshot 3: SQL Injection Error via Search Function

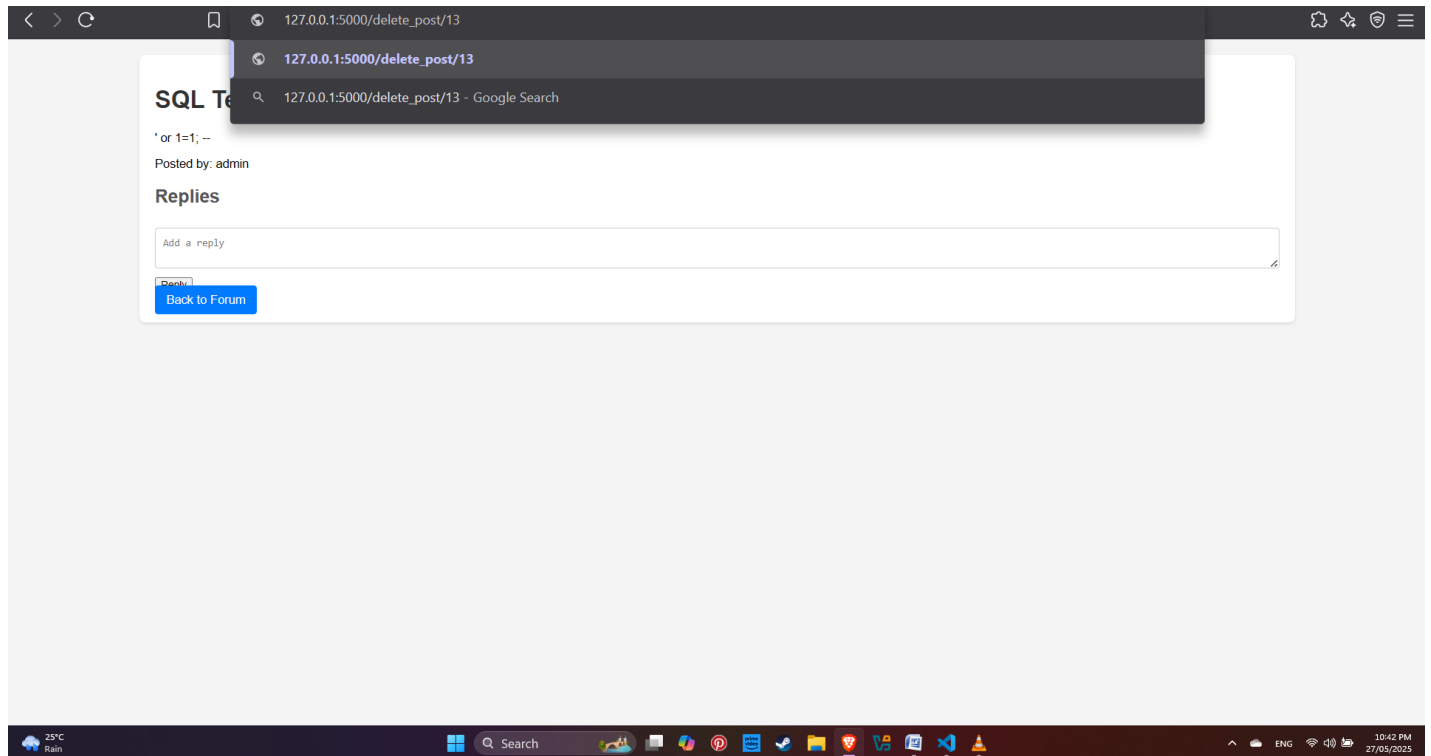


Query: ' OR 1=1;

Description:

This image displays an SQL Injection attempt through the `/search` route. The malformed query triggers a `ProgrammingError` from `SQLAlchemy`, indicating that multiple statements are being processed improperly. The vulnerability arises from directly embedding user input into raw SQL strings without parameterization. This shows that an attacker could manipulate queries to extract or modify sensitive data if the syntax is corrected.

Screenshot 4: Insecure Direct Object Reference (IDOR) Exploit

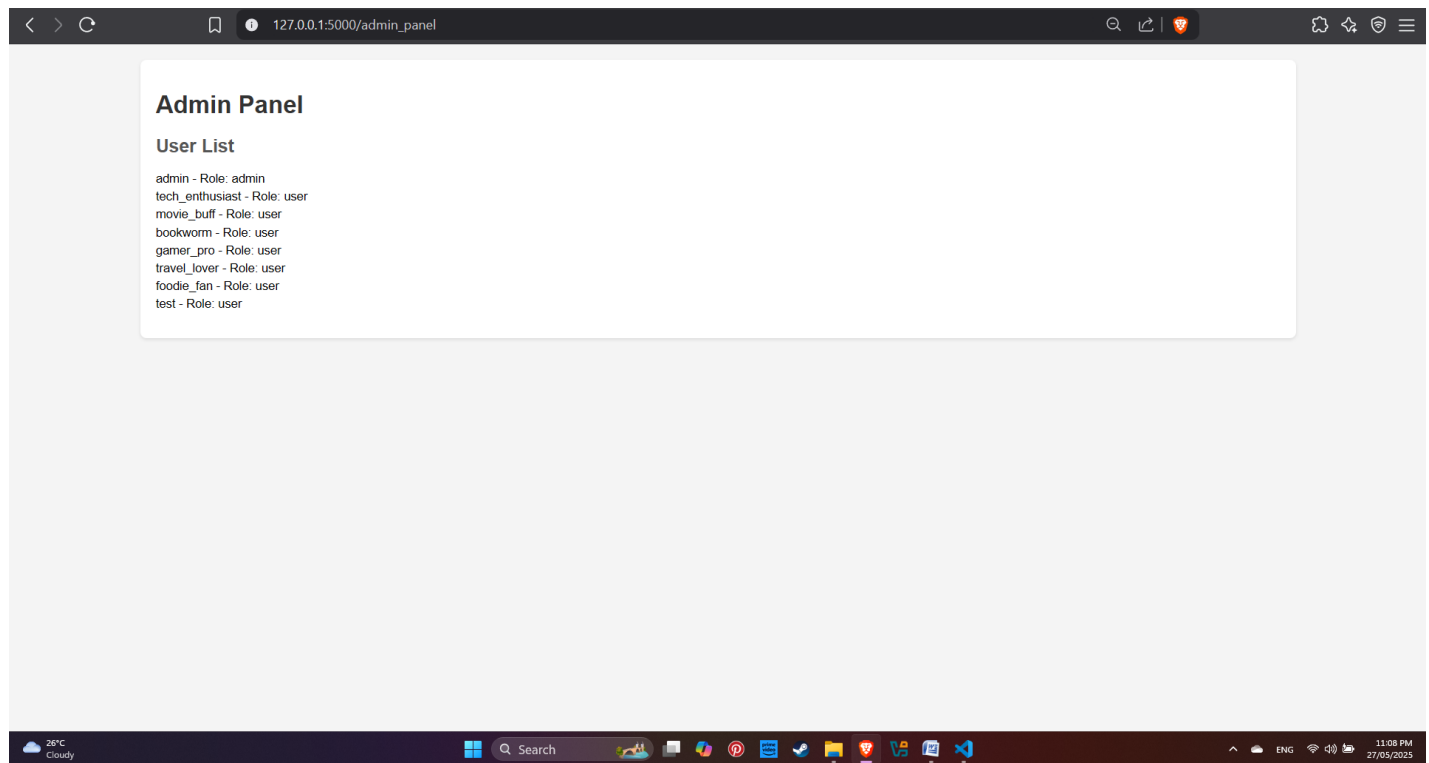


URL Accessed: /delete_post/13

Description:

In this snapshot, a logged-in user manually accesses a delete URL for a post (`post_id=13`) that they do not own. The post is deleted without ownership verification, demonstrating an **IDOR vulnerability**. This allows any authenticated user to delete content created by other users, leading to unauthorized actions and potential data loss.

Screenshot 5: Insecure Admin Panel Access

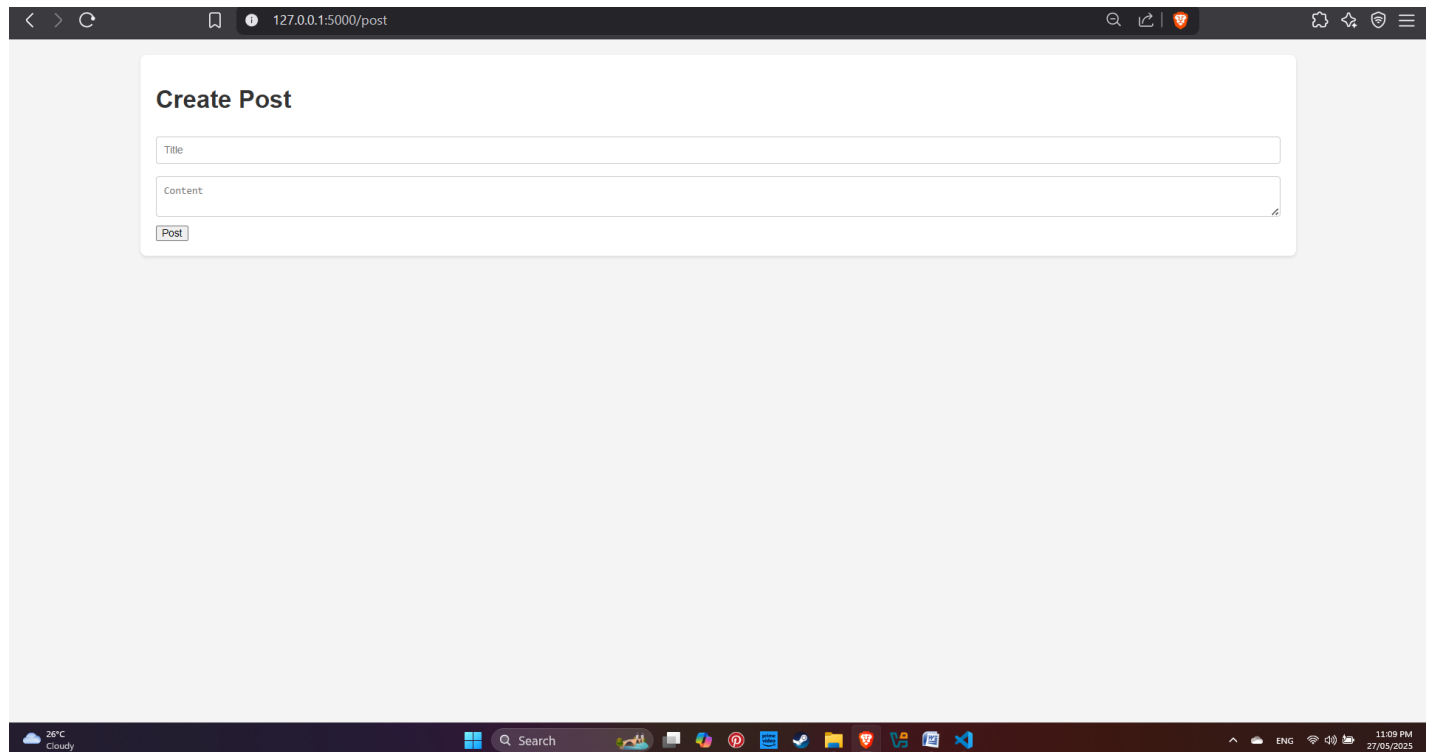


Page: /admin_panel

Description:

This screenshot shows the admin panel, where the full list of users and their roles is displayed. The panel was accessed by a normal user without proper role-based access control. This confirms a critical **authorization flaw**, where the server fails to validate if the logged-in user has an admin role before showing sensitive administrative features. This represents a direct violation of access control principles and can be exploited for privilege escalation.

Screenshot 6: Post Creation Page



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:5000/post". The page title is "Create Post". It features two input fields: "Title" and "Content". Below the "Content" field is a "Post" button. The browser's taskbar at the bottom shows the system clock as 11:09 PM on 27/05/2025, along with various application icons and network status indicators.

Page: /post

Description:

This image displays the **post creation interface**, available to all authenticated users. It features basic input fields for a post title and content. However, the lack of input sanitization or validation on this page allows users to submit malicious payloads (e.g., scripts), making this a key entry point for **Cross-Site Scripting (XSS)** attacks. This screenshot supports the demonstration of content injection and the need for secure input handling.

6.CONCLUSION

This project successfully demonstrated the construction and security assessment of a deliberately vulnerable forum web application developed using the Flask framework. The objective of the project was not merely to build a functioning system, but to intentionally introduce and exploit security flaws that are commonly found in real-world applications. Through this practical and controlled approach, the project provided a deep understanding of both application functionality and its associated security challenges.

Throughout the project, critical vulnerabilities were identified and validated through manual testing and hands-on exploitation. These included:

- **Cross-Site Scripting (XSS)** via input fields and uploaded HTML files.
- **SQL Injection** through unfiltered search parameters.
- **Insecure Direct Object Reference (IDOR)** in post deletion functionality.
- **Lack of Rate Limiting** on authentication endpoints, allowing brute-force attacks.
- **Lack of Input Validation**, making the system prone to injection and denial-of-service vectors.
- **Broken Access Control** in the admin panel, where non-admin users could view restricted data.
- **Remote Code Execution (RCE)** by uploading executable PHP files, leading to full server compromise.

Each vulnerability was carefully exploited and analyzed, and detailed evidence such as screenshots and payload samples were recorded. The impact of each issue was assessed not only in terms of technical consequences but also in terms of how they affect users, system integrity, and organizational risk.

The implementation phase helped solidify backend development skills using Flask and Python, while the exploitation phase enhanced understanding of ethical hacking, penetration testing, and secure development principles. The vulnerabilities chosen reflect real-world attack surfaces listed in the OWASP Top 10, making the project highly relevant to current security concerns in the web development industry.

Moreover, this project highlighted the importance of secure coding practices such as:

- Sanitizing user input
- Escaping output in templates
- Implementing proper authentication and authorization checks
- Restricting file uploads and execution permissions
- Using prepared statements and ORM features to prevent SQL injection
- Implementing security headers and access control mechanisms

This dual focus on building and breaking has reinforced the idea that **security must be an integral part of software design and development**, not just an afterthought. By intentionally exploring unsafe configurations and understanding their impact, the project has provided a hands-on foundation for future secure application development, penetration testing, and red-teaming efforts.

In conclusion, this project fulfilled its intended goals by serving as both a practical hacking lab and a comprehensive case study in web application security. It serves as a valuable reference for learners, developers, and cybersecurity professionals who seek to build or audit secure systems.

7.REFERENCES

These references include the resources used during development, vulnerability analysis, and security research:

1. OWASP Foundation. (2023). *OWASP Top 10: The Ten Most Critical Web Application Security Risks*. Retrieved from <https://owasp.org/www-project-top-ten/>
2. Flask Documentation. (2025). *Flask 2.x Documentation*. Retrieved from <https://flask.palletsprojects.com>
3. SQLAlchemy Documentation. (2025). *SQLAlchemy ORM Guide*. Retrieved from <https://docs.sqlalchemy.org>
4. Jinja2 Documentation. (2025). *Templating with Jinja2*. Retrieved from <https://jinja.palletsprojects.com>
5. CWE MITRE. (2025). *Common Weakness Enumeration (CWE)*. Retrieved from <https://cwe.mitre.org>
6. PortSwigger Web Security Academy. (2025). *Learning Web Security*. Retrieved from <https://portswigger.net/web-security>
7. Mozilla Developer Network. (2025). *Same-Origin Policy & Web Security*. Retrieved from <https://developer.mozilla.org>
8. Wireshark Foundation. (2025). *Wireshark Packet Analysis Tool*. Retrieved from <https://www.wireshark.org>
9. Bandit Security Tool. (2025). *Python Security Static Analyzer*. Retrieved from <https://bandit.readthedocs.io>
10. Flask-Limiter. (2025). *Rate Limiting for Flask*. Retrieved from <https://flask-limiter.readthedocs.io>

8.APPENDIX

The appendix includes supplementary materials such as screenshots, logs, payload samples, and scripts used during testing and documentation.

A. Screenshots of Exploits

Figure No.	Title	Description
A1	XSS via Malicious HTML Upload	Stored XSS exploiting uploaded HTML file.
A2	PHP RCE Shell Interface	Remote code execution via uploaded PHP.
A3	SQL Injection Exploit	Search function error due to SQL injection.
A4	IDOR on Post Deletion	Manual access to unauthorized delete route.
A5	Admin Panel Access without Role	Broken access control exposing user list.
A6	Post Creation with XSS Input	Demonstrating XSS entry via form.

B. Sample Payloads

Type	Payload Example
XSS	<code><script>alert('XSS');</script></code>
SQL Injection	<code>' OR 1=1 --</code>
PHP RCE Upload	<code><?php system(\$_GET['cmd']); ?></code>
Malicious HTML	File: malicious.html with embedded script.
IDOR Test	URL: <code>http://127.0.0.1:5000/delete_post/13</code>

C. Tools Used

- **VS Code** – Development environment.
- **Brave/Chrome Browser** – For testing XSS and session-based behaviors.
- **Python Logging** – For monitoring user actions and attack behavior.